中国科学院大学
《计算机体系结构》实验报告

组员 1 武弋 2022K8009929002

组员 2 张家玮 2022K8009929010

组员 3 王泽黎 2022K8009929011

实验项目编号 7 实验名称 高速缓存设计专题实验

一、 实验任务

1. 设计 Cache 模块。
2. 在 CPU 中集成 ICache。
3. 在 CPU 中集成 DCache。
4. 在 CPU 中添加 CACOP 指令。

二、 Cache 模块的设计

本部分的主要内容是设计 Cache 模块并利用 Cache 模块级验证环境对所设计的 Cache 进行验证，通过仿真和上板验证。

- **Cache 端口与结构**。

  Cache 模块的端口定义如下：

```
module cache (
    input         clk,
    input         resetn,
    input         cachable, //是否可缓存
    // interface to CPU
    input         valid, //cache有效位
    input         op, // 0: read, 1: write
    input [ 7:0] index, //cache索引
    input [19:0] tag, //cache标记
    input [ 3:0] offset, //cache偏移
    input [ 3:0] wstrb, //写数据掩码
    input [31:0] wdata, //写数据
    output        addr_ok, //地址有效
    output        data_ok, //数据有效
    output [31:0] rdata, //读数据
    output        cache_recv_addr, //cache接收地址
    input         cacop, //cache操作
    input [ 4:0] code, //cache操作码
    output        cache_write, //cache写
    // interface to axi
    output        rd_req, //读请求
    output [ 2:0] rd_type, // 0: byte, 1: half, 2: word, 4: cache line
    output [31:0] rd_addr, //读地址
    input         rd_rdy, //读就绪
    input         ret_valid, //返回有效
```

```
    input         ret_last, //最后一个数据返回
    input [31:0]  ret_data, //返回数据
    output        wr_req, //写请求
    output [ 2:0] wr_type, // 0: byte, 1: half, 2: word, 4: cache line
    output [31:0] wr_addr, //写地址
    output [ 3:0] wr_wstrb, //写数据掩码
    output [127:0] wr_data, //写数据
    input         wr_rdy, //写就绪
    input         data_write_ok, //数据写入完成
    output        write_refill, //数据重填写回
    output reg    write_complete //数据写回完成
);
```

Cache 的逻辑结构如下：

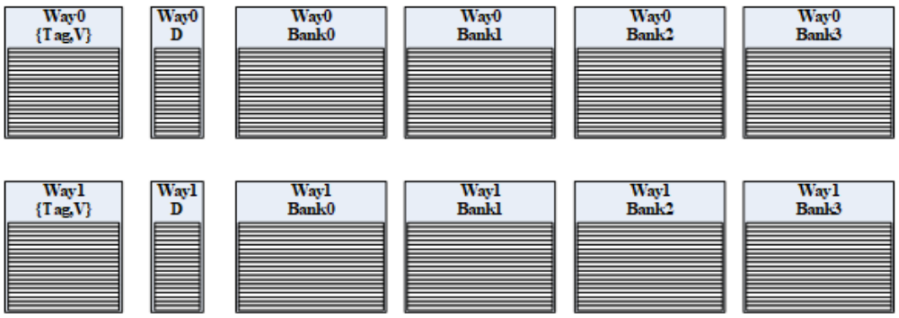

图 1: Cache 逻辑结构

如图所示，其含有两路组相连的 Cache，每路 Cache 包含 256 行，每行包含 4 个字，每个字包含 4 个字节，故 Cache 的总大小为 8KB。此外，每行还有一个 21 位的 tagv 和一个 dirty 位，分别用来存储标记、有效位和脏位。

Cache 存储结构的实现调用了 Xilinx 提供的 RAM IP 核，代码如下：

```
generate
    for (i = 0; i < 2; i = i + 1) begin
        TAGV_RAM u_tagv_ram (
            .clka  (clk          ),
            .wea   (tagv_we[i]   ),
            .addra (tagv_addr[i] ),
            .dina  (tagv_wdata[i] ),
            .douta (tagv_rdata[i] ),
            .ena   (1'b1         )
        );
    end
endgenerate

generate
    for (i = 0; i < 2; i = i + 1) begin
        for (j = 0; j < 4; j = j + 1) begin
            DATA_BANK_RAM u_data_bank_ram (
                .clka  (clk                 ),
                .wea   (data_bank_wstrb[i][j] ),
```

```
                .addra (data_bank_addr[i][j] ),
                .dina  (data_bank_wdata[i][j] ),
                .douta (data_bank_rdata[i][j] ),
                // .ena   (data_bank_we[i][j]  )
                .ena   (1'b1)
            );
        end
    end
endgenerate
```

• **Cache 状态机设计**。

Cache 模块需要使用状态机来实现其对读写数据功能的管理，主状态机包括：

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。

负责具体写入数据的 write_buffer 状态机包括：

IDLE：Write Buffer 当前没有待写的数据。

WRITE：将待写数据写入到 Cache 中。在主状态机处于 LOOKUP 状态且发现 Store 操作命中 Cache 时，触发 Write Buffer 状态机进入 WRITE 状态，同时 Write Buffer 会寄存 Store 要写入的 Index、路号、offset、写使能（写 32 位数据里的哪些字节）和写数据。

其状态转换示意图如下：

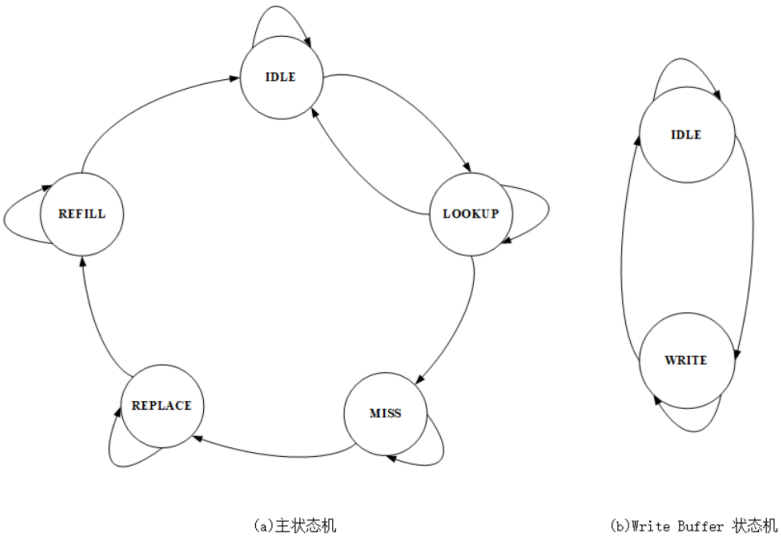

图 2: Cache 状态转换图

具体实现代码如下：

```
always @(posedge clk) begin
    if (!resetn) begin
        current_state <= IDLE;
        wbuf_current_state <= WBUF_IDLE;
```

```
    end else begin
        current_state <= next_state;
        wbuf_current_state <= wbuf_next_state;
    end
end

// Main State Transition
always @(*) begin
    case (current_state)
        IDLE: begin
            if (valid && !hit_write_hazard) begin
                next_state = LOOKUP;
            end
            else if (!valid || valid && hit_write_hazard) begin
                next_state = IDLE;
            end
        end
        LOOKUP: begin
            if (cache_hit & (!valid | valid & hit_write_hazard) & ~cache_write) begin
                next_state = IDLE;
            end
            else if (cache_hit & valid & !hit_write_hazard & ~cache_write) begin
                next_state = LOOKUP;
            end
            else if (~reg_op && (!dirty[replace_way][reg_index] || !tagv_rdata[replace_way][0]) &&
                reg_cachable && ~cacop) begin
                next_state = REPLACE;
            end
            else if (!cache_hit | cache_write) begin
                next_state = MISS;
            end
        end
        MISS: begin
            if (!wr_rdy) begin
                next_state = MISS;
            end
            else begin
                next_state = REPLACE;
            end
        end
        REPLACE: begin
            if(cache_write) begin
                next_state = IDLE;
            end
            else if (!rd_rdy) begin
                next_state = REPLACE;
            end
            else begin
                next_state = REFILL;
            end
```

```
        end
        REFILL: begin
            if ((ret_valid && ret_last) || ~reg_cachable) begin
                next_state = IDLE;
            end
            else begin
                next_state = REFILL;
            end
        end
        default: begin
            next_state = IDLE;
        end
    endcase
end

// Write Buffer State Transition
always @(*) begin
    case (wbuf_current_state)
        WBUF_IDLE: begin
            if (current_state == LOOKUP & hit_write & ~(cacop & code[0])) begin
                wbuf_next_state = WBUF_WRITE;
            end
            else begin
                wbuf_next_state = WBUF_IDLE;
            end
        end
        WBUF_WRITE: begin
            wbuf_next_state = WBUF_IDLE;
        end
        default: begin
            wbuf_next_state = WBUF_IDLE;
        end
    endcase
end
```

- **Cache 命中判断逻辑。**

  在 Cache 模块中，通过比对标记信息来判断 Cache 的命中，其逻辑如下：

```
wire way0_hit, way1_hit, cache_hit; //0路、1路命中和cache命中
wire hit_write; //写命中

assign way0_hit = tagv_rdata[0][0] & (tagv_rdata[0][20:1] == reg_tag) & reg_cachable;
assign way1_hit = tagv_rdata[1][0] & (tagv_rdata[1][20:1] == reg_tag) & reg_cachable;
assign cache_hit = (way0_hit | way1_hit) & ~(cacop & code[4:3] == 2'b01);
assign hit_write = cache_hit & reg_op;
```

- **命中时的读写数据生成逻辑。**

  在 cache 命中时，如果需要将 Cache 中的数据读出，其首先需要选择对应的数据，代码如下：

```
wire [ 31:0] way0_load_word, way1_load_word; //0路、1路读数据
wire [ 31:0] load_res; //读数据结果

assign way0_load_word = data_bank_rdata[0][reg_offset[3:2]];
assign way1_load_word = data_bank_rdata[1][reg_offset[3:2]];
assign load_res = {32{ way0_hit}} & way0_load_word
                | {32{ way1_hit}} & way1_load_word
                | {32{ret_valid}} & ret_data;
```

如果需要将数据写入 Cache，其要准备好待写入的各项数据如下：

```
reg       hitwr_way; //写路
reg [ 1:0] hitwr_bank; //写bank
reg [ 7:0] hitwr_index; //写索引
reg [ 3:0] hitwr_wstrb; //写数据掩码
reg [31:0] hitwr_wdata; //写数据

always @(posedge clk) begin
    if (!resetn) begin
        hitwr_way <= 1'b0;
        hitwr_bank <= 2'b00;
        hitwr_index <= 8'b0;
        hitwr_wstrb <= 4'b0;
        hitwr_wdata <= 32'b0;
    end
    else if (current_state == LOOKUP && hit_write) begin
        hitwr_way  <= way0_hit ? 1'b0 : 1'b1;
        hitwr_bank <= reg_offset[3:2];
        hitwr_index <= reg_index;
        hitwr_wstrb <= reg_wstrb;
        hitwr_wdata <= reg_wdata;
    end
end
```

- **未命中时需要的操作。**

  若 Cache 未命中，就需要向 AXI 总线发出读写请求，等待返回数据，代码如下：

```
assign rd_req = (current_state == REPLACE & (~reg_op | (reg_op & reg_cachable &
    write_complete))) & ~rd_rdy & ~(~reg_cachable & reg_op) & ~cache_write;
assign rd_type = reg_cachable ? 3'b100 : 3'b010;
assign rd_addr = {reg_tag, reg_index, ((~reg_cachable & ~reg_op) ? reg_offset : 4'b0)};
assign wr_req = reg_wr_req;
assign wr_type = reg_cachable | cacop ? 3'b100 : 3'b010;
assign wr_wstrb = reg_cachable | cacop ? 4'b1111 : reg_wstrb;
assign wr_addr = reg_cachable & ~cacop ? {tagv_rdata[replace_way][20:1], reg_index, reg_offset} :
             cacop & code[4:3] == 2'b01 ? {reg_tagv_dcacop[20:1], reg_index, reg_offset[3:1],
                 1'b0} :
             {reg_tag, reg_index, reg_offset};
assign wr_data = reg_cachable | cacop ? {8'hff, replace_data[119:0]} : {4{reg_wdata}};
```

```
assign replace_data = cacop & code[4:3] == 2'b01 ? {data_bank_rdata[offset[0]][3],
    data_bank_rdata[offset[0]][2],
                data_bank_rdata[offset[0]][1], data_bank_rdata[offset[0]][0]} :
                cacop & code[4:3] == 2'b10 & dcacop_hit0 ? {data_bank_rdata[0][3],
                    data_bank_rdata[0][2],
                data_bank_rdata[0][1], data_bank_rdata[0][0]} :
                cacop & code[4:3] == 2'b10 & dcacop_hit1 ? {data_bank_rdata[1][3],
                    data_bank_rdata[1][2],
                data_bank_rdata[1][1], data_bank_rdata[1][0]} :
                {data_bank_rdata[replace_way][3], data_bank_rdata[replace_way][2],
                data_bank_rdata[replace_way][1], data_bank_rdata[replace_way][0]};
```

由代码可见，请求信号的生成需要考虑 cacop 指令和非缓存访存的特殊情况。

- **操作的最终完成**。

  各个操作完成后，需要给 CPU 返回结果信号，如下：

```
assign addr_ok = current_state == IDLE | current_state == LOOKUP & valid & cache_hit & (op | !op
    & !hit_write_hazard);
assign data_ok = (current_state == LOOKUP & cache_hit & ~cache_write)
             | (current_state == LOOKUP & reg_op & ~(hit_write_hazard | (current_state ==
                 LOOKUP & next_state == MISS)))
             | (current_state == REFILL & ret_valid & !reg_op & ((ret_data_num ==
                 reg_offset[3:2] & reg_cachable) | !reg_cachable));
assign rdata = load_res;
```

cache 中的其他逻辑代码将在后两个部分中解释。

三、 将 Cache 集成至 CPU 中

本部分需要将实践任务 20 完成 Cache 模块作为 ICache 和 DCache 集成到实践任务 19 完成的 CPU 中；修改 CPU 中的 AXI 转换桥，以支持 Burst 传输。然后在采用 AXI 总线的 SoC 验证环境里完成 exp21、exp22 对应 func 的功能验证，要求成功通过仿真和上板验证。

- **将 ICache 集成到 CPU**。

  首先需要在 CPU 中添加 ICache 的例化模块，代码如下：

```
cache icache(
    .clk        (aclk),
    .resetn     (aresetn),
    .cachable   (1'b1),
    // interface to CPU
    .valid      (icache_valid),
    .op         (icache_op),
    .index      (icache_index),
    .tag        (icache_tag),
    .offset     (icache_offset),
    .wstrb      (icache_wstrb),
    .wdata      (icache_wdata),
    .addr_ok    (icache_addr_ok),
```

```
    .data_ok    (icache_data_ok),
    .rdata      (icache_rdata),
    .cache_recv_addr (icache_cache_recv_addr),
    // interface to bridge
    .rd_req     (icache_rd_req),
    .rd_type    (icache_rd_type),
    .rd_addr    (icache_rd_addr),
    .rd_rdy     (icache_rd_rdy),
    .ret_valid  (icache_ret_valid),
    .ret_last   (icache_ret_last),
    .ret_data   (icache_ret_data),
    .wr_req     (icache_wr_req),
    .wr_type    (icache_wr_type),
    .wr_addr    (icache_wr_addr),
    .wr_wstrb   (icache_wr_wstrb),
    .wr_data    (icache_wr_data),
    .wr_rdy     (icache_wr_rdy),
    .data_write_ok (1'b1),
    .write_refill (icache_wr_req),
    .write_complete (icache_write_complete),
    // exp23
    .cacop      (icacop),
    .code       (cacop_code),
    .cache_write ()
);
```

这些信号的含义已经在前面介绍，故直接给出其组成逻辑代码如下：

```
assign icacop_vaddr  = nextpc & {32{~icacop | cacop_code[4:3]!=2'b10 | icacop_compelete}}|
   icacop_addr & {32{icacop & ~icacop_compelete & cacop_code[4:3]==2'b10}};
assign icache_valid  = inst_sram_req;
assign icache_op     = inst_sram_wr;
assign icache_index  = inst_sram_addr[11: 4] & {8{~icacop | cacop_code[4:3]==2'b10 |
    icacop_compelete}}|
   icacop_addr[11: 4] & {8{icacop & ~icacop_compelete & cacop_code[4:3]!=2'b10}};
assign icache_tag    = inst_sram_addr[31:12];
assign icache_offset = inst_sram_addr[ 3: 0] & {4{~icacop | cacop_code[4:3]==2'b10 |
    icacop_compelete}}|
   icacop_addr[ 3: 0] & {4{icacop & ~icacop_compelete & cacop_code[4:3]!=2'b10}};
assign icache_wstrb  = inst_sram_wstrb;
assign icache_wdata  = inst_sram_wdata;
assign icache_rd_rdy = arready && arid == 4'b0;
assign icache_ret_valid = rvalid && rid == 4'b0;
assign icache_ret_last = rlast && rid == 4'b0;
assign icache_ret_data = inst_sram_rdata;
assign icache_wr_rdy = 1'b1;
```

其中读写控制信号大多复用了 bridge 和 sram 的信号，只有 exp23 中部分信号需要根据 icacop 的值进行选择。

然后，CPU（以及 AXI Bridge）中原有的取值相关控制信号就可以用 ICache 输出的信号进行改写，代码过于零散，且意义不大，不再给出。

- **将 DCache 集成到 CPU**。

Dcache 的集成相对比较复杂。由于其具有缓存访问和非缓存访问两种情况，故需要分别处理这两种情况。并且，由于一次写入内存的数据为 16 字节，故需要对 Bridge 中的写入部分进行较多修改。

首先，需要在 CPU 中添加 DCache 的例化模块，代码如下：

```
cache dcache(
    .clk        (aclk),
    .resetn     (aresetn),
    .cachable   (dcache_cachable),
    // interface to CPU
    .valid      (dcache_valid),
    .op         (dcache_op),
    .index      (dcache_index),
    .tag        (dcache_tag),
    .offset     (dcache_offset),
    .wstrb      (dcache_wstrb),
    .wdata      (dcache_wdata),
    .addr_ok    (dcache_addr_ok),
    .data_ok    (dcache_data_ok),
    .rdata      (dcache_rdata),
    .cache_recv_addr (dcache_cache_recv_addr),
    // interface to bridge
    .rd_req     (dcache_rd_req),
    .rd_type    (dcache_rd_type),
    .rd_addr    (dcache_rd_addr),
    .rd_rdy     (dcache_rd_rdy),
    .ret_valid  (dcache_ret_valid),
    .ret_last   (dcache_ret_last),
    .ret_data   (dcache_ret_data),
    .wr_req     (dcache_wr_req),
    .wr_type    (dcache_wr_type),
    .wr_addr    (dcache_wr_addr),
    .wr_wstrb   (dcache_wr_wstrb),
    .wr_data    (dcache_wr_data),
    .wr_rdy     (dcache_wr_rdy),
    .data_write_ok(bvalid),
    .write_refill (dcache_write_refill),
    .write_complete(dcache_write_complete),
    // exp23
    .cacop      (inst_cacop_MEM),
    .code       (cacop_code_MEM),
    .cache_write (dcacop_write)
);
```

其信号的生成逻辑如下：

```
assign dcache_cachable = (~csr_crmd_pg) ? (csr_crmd_datm == 2'b01) :
    data_flag_dmw0_hit ? (data_dmw0_mat == 2'b01) :
    data_flag_dmw1_hit ? (data_dmw1_mat == 2'b01) :
    dcacop_write ? 1'b1 :
```

```
    (s1_mat == 2'b01);
assign dcache_valid  = data_sram_req;
assign dcache_op     = data_sram_wr;
assign dcache_index  = data_sram_addr[11: 4];
assign dcache_tag    = data_sram_addr[31:12];
assign dcache_offset = data_sram_addr[ 3: 0];
assign dcache_wstrb  = data_sram_wstrb;
assign dcache_wdata  = data_sram_wdata;
assign dcache_rd_rdy = (~data_sram_wr & arready & arid == 4'b1) | (data_sram_wr &
    ~dcache_cachable) | (data_sram_wr & dcache_cachable & arready & arid == 4'b1);
assign dcache_ret_valid = rvalid & rid == 4'b1;
assign dcache_ret_last = rlast & rid == 4'b1;
assign dcache_ret_data = data_sram_rdata;
assign dcache_wr_rdy = 1'b1;
assign dcache_req    = dcache_wr_req | dcache_rd_req;
assign dcache_addr   = dcache_op ? dcache_wr_addr : dcache_rd_addr;
assign dcacop        = inst_cacop_MEM & (cacop_code_MEM[2:0] == 3'b001);
```

其大部分信号也是复用了 bridge 和 sram 的信号，但是需要使用 dmw 命中情况与 csr 寄存器中的相关信息来判断访存是否使用缓存。

与 ICache 类似地，其他 CPU 中访存相关的控制信号也要使用 DCache 的输出信号进行代替，不再给出。

DCache 的集成需要对 bridge 进行较大规模的修改，以适应两种缓存访问和同时写 16 字节数据的情况。首先是 bridge 端口的添加，新端口如下：

```
input wire [ 2:0] dcache_rd_type, // exp22
input wire [ 2:0] dcache_wr_type, // exp22
input wire [127:0]dcache_wr_data,// exp22
input wire        dcache_cachable,// exp22
input wire        dcache_write_refill,// exp22
input wire        dcache_write_complete,// exp22
input wire        dcacop_write //exp23
```

为了实现写入 4 字功能，需要使用一个数据 buffer 来储存这四个字，代码如下：

```
always @(posedge aclk) begin
    if (~aresetn) begin
        {wdata_buffer[3],wdata_buffer[2],wdata_buffer[1],wdata_buffer[0]} <= 128'b0;
        wstrb <= 4'b0;
        wid   <= 4'b1;
    end
    else if(data_sram_req & (data_sram_wr & ~write_to_read)) begin
        {wdata_buffer[3],wdata_buffer[2],wdata_buffer[1],wdata_buffer[0]} <= dcache_wr_data;
        wstrb <= data_sram_wstrb;
    end
end
```

写入 16 字节数据的关键控制信号是 wlen 和 wlast，分别标记还需要写入的数据长度和是否是最后一次写入，代码如下：

```
always @(posedge aclk) begin
```

```
    if(~aresetn) begin
        wlen <= 8'b0;
    end
    else if(data_sram_req & (data_sram_wr & ~write_to_read)) begin
        wlen <= {6'b0, {2{dcache_wr_type[2]}}};
    end
    else if(wvalid & wready) begin
        wlen <= wlen - 1;
    end
end
assign wlast = ~|wlen[1:0]; //写最后一个数据
```

然后，用这两个信号来控制写入内存的端口信号即可完成 4 字的写入，代码如下：

```
assign wlast = ~|wlen[1:0]; //写最后一个数据
assign wdata = wdata_buffer[~wlen[1:0]]; //写数据
assign wvalid = data_waddr_ok & !data_wdata_ok; //写请求数据有效
//(以下部分位于CPU中，传入到bridge内)
always @(posedge aclk) begin
    if(reset) begin
        data_wdata_ok <= 1'b0;
    end
    else if(wready && wvalid && & ((wlast & dcache_cachable) | ~dcache_cachable)) begin
        data_wdata_ok <= 1'b1;
    end
    else if(pipe_ready_go[3]) begin
        data_wdata_ok <= 1'b0;
    end
end
```

至于是否缓存访问的片选信息已经分布在前面 Cache 模块部分给出的代码中以及 bridge 的各个控制信号中，内容过多且零散，不再集中展示。

四、 cacop 指令的添加

本部分将在实践任务 22 完成的 CPU 中增加 CACOP 指令实现，并在采用 AXI 总线的 SoC 验证环境里完成 exp23 对应 func 的功能验证，要求成功通过仿真和上板验证。

- **CACOP 指令基本数据通路。**

首先需要添加 cacop 指令的译码信号，代码如下：

```
// cache instruction
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          = inst_cacop_MEM & cacop_code_MEM[2:0] == 3'b001; //该信号传入DCache
```

然后将指令译码信号添加到其他的通用数据通路中，不再赘述。

- **CPU 中 ICache 对 CACOP 指令的实现。**

为了尽量减少修改 ICache 对后续指令取值造成的影响，我们将 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，代码如下：

```
assign icacop_vaddr  = nextpc & {32{~icacop | cacop_code[4:3]!=2'b10 | icacop_complete}}|
   icacop_addr & {32{icacop & ~icacop_complete & cacop_code[4:3]==2'b10}};
assign icache_valid  = inst_sram_req;
assign icache_op     = inst_sram_wr;
assign icache_index  = 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 icache_tag    = inst_sram_addr[31:12];
assign icache_offset = 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}};
```

可以看到其中的 icacop_complete 信号，用来标记 ICache 的 cacop 指令是否执行完毕。其代码如下：

```
reg icacop_complete;
always @(posedge aclk) begin
   if (reset) 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 if (pipe_ready_go[0]) begin
      icacop_complete <= 1'b0;
   end
end
```

以上设计是因为 op 为 10 时，需要两个周期才能完成，op 不为 10 时需一个周期完成。

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

```
assign icacop_vaddr  = nextpc & {32{~icacop | cacop_code[4:3]!=2'b10 | icacop_complete}}|
icacop_addr & {32{icacop & ~icacop_complete & cacop_code[4:3]==2'b10}};
```

此时可能会触发 tlb 的异常，需要在异常处理流程添加这种情况，代码如下：

```
assign csr_ecode_ID_m = csr_ecode_ID != 6'b0 ? csr_ecode_ID
   : ({6{inst_syscall }} & 6'hb
   | {6{inst_break  }} & 6'hc
   | {6{inst_not_exist || invtlb_op_not_exist}} & 6'hd) & {6{~inst_need_refetch_ID}}
   | {6{ecode_MMU_preIF == 6'h3f & icacop}} & 6'h3f
   | {6{ecode_MMU_preIF == 6'h01 & icacop}} & 6'h01;
```

由于 ICache 的 cacop 指令在 ID 级完成，而其异常判断也挤占的取指级的数据通路，所以触发异常时，需要在 IF

级将异常码越级传递到 ID 级。

- **CPU 中 DCache 对 CACOP 指令的实现。**

相比 ICache，DCache 的 cacop 指令时序较为简单，只需要将其执行放在正常的 MEM 级即可。

由于 dcache 可能涉及写内存操作，所以 cache 需要向 CPU 输出一个写入标志信号用于标志进入 CPU 访存状态，以下为被其控制的信号的两个例子：

```
.cache_write (dcacop_write) //例化cache模块时的信号
assign dcache_cachable = (~csr_crmd_pg) ? (csr_crmd_datm == 2'b01) :
                  data_flag_dmw0_hit ? (data_dmw0_mat == 2'b01) :
                  data_flag_dmw1_hit ? (data_dmw1_mat == 2'b01) :
                  dcacop_write ? 1'b1 :
                  (s1_mat == 2'b01)
                  ;
assign memory_access = (inst_ld_w_MEM | inst_ld_b_MEM | inst_ld_h_MEM | inst_ld_bu_MEM |
                inst_ld_hu_MEM | inst_st_w_MEM | inst_st_b_MEM | inst_st_h_MEM | dcacop_write) &
                ~ex_MEM_m & ~inst_need_refetch_MEM & ~ex_WB;
```

在其他的数据通路中，只需要将 DCache 的 cacop 数据添加到共用的片选逻辑中（如 tlb_s1 访问、data_sram 地址片选和 MEM 级异常等）即可，比较简单，不再给出。

- **Cache 模块中对 CACOP 指令的实现。**

所有的 cacop 指令都可能涉及对于 tagv 的读写，回顾其写入信号，代码如下：

```
assign tagv_we[i] = ret_valid & ret_last & replace_way == i & reg_cachable | cacop & offset[0]
    == i & code[4:3] != 2'b10 |
                cacop_wr_tagv & tagv_rdata[i][0] & (tagv_rdata[i][20:1] == tag) & reg_cachable
                    & code[4:3] == 2'b10;
assign tagv_addr[i] = current_state == IDLE ? index : reg_index;
assign tagv_wdata[i] = {reg_tag, 1'b1} & {21{ret_valid & ret_last & replace_way == i &
    reg_cachable}}|
                {reg_tagv_dcacop[20:1], 1'b0} & {21{cacop & offset[0] == i & (code[4:3] ==
                    2'b01 | code[4:3] == 2'b10)}}|
                {21'b0} & {21{cacop & offset[0] == i & code[4:3] == 2'b00}};
```

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。

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：

```
assign cache_write_new = dirty[offset[0]][index] & tagv_rdata[offset[0]][0] & cacop & code[4:3]
    == 2'b01|
   dirty[0][index] & tagv_rdata[0][0] & cacop & code[4:3] == 2'b10 & way0_hit |
   dirty[1][index] & tagv_rdata[1][0] & 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
assign cache_write = (cache_write_reg | cache_write_new) & cacop & (code[4:3] == 2'b01 |
    code[4:3] == 2'b10 & (cacop_hit | cache_hit));
```

然后，需要使用该信号控制状态机，使其进入 MISS 状态，代码如下：

```
LOOKUP: begin
    if (cache_hit & (!valid | valid & hit_write_hazard) & ~cache_write) begin
        next_state = IDLE;
    end
    else if (cache_hit & valid & !hit_write_hazard & ~cache_write) begin
        next_state = LOOKUP;
    end
    else if (~reg_op && (!dirty[replace_way][reg_index] || !tagv_rdata[replace_way][0]) &&
        reg_cachable && ~cacop) begin
        next_state = REPLACE;
    end
    else if (!cache_hit | cache_write) begin
        next_state = MISS;
    end
end
```

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 = reg_wr_req;
assign wr_type = reg_cachable | cacop ? 3'b100 : 3'b010;
assign wr_wstrb = reg_cachable | cacop ? 4'b1111 : reg_wstrb;
assign wr_addr = reg_cachable & ~cacop ? {tagv_rdata[replace_way][20:1], reg_index, reg_offset} :
               cacop & code[4:3] == 2'b01 ? {reg_tagv_dcacop[20:1], reg_index, reg_offset[3:1],
                   1'b0} :
               {reg_tag, reg_index, reg_offset};
assign wr_data = reg_cachable | cacop ? {8'hff, replace_data[119:0]} : {4{reg_wdata}};
```

五、 Debug 过程

- **exp20**。

本实验中印象最深刻的问题是 hit_write_hazard 中判断冲突时误将 reg_offset[3:2]==offset[3:2] 写为 reg_offset==offset，导致条件错误无法正确判断，通过看波形发现 cache 行中的值不对定位到了该问题并很快解决。总体来讲本实验复杂程度较低，没有遇到过多的困难。

- **exp21、exp22**。

错误 1：数据通路异常

（1）错误现象：pc 在 preIF 级未发出取值指令时就立即更新为 nextpc

（2）分析定位过程：通过观察仿真波形，发现是由于将 bridge 中的 inst_addr_ok 直接替换为 icache_addr_ok，导致 pc 更新相关控制信号持续拉高

(3)错误原因:cache 中的 addr_ok 信号代表 cache 允许 CPU 发起请求,而 bridge 中的 addr_ok 意味着地址握手成功,二者不能起相同作用
(4)修正效果:在 cache 中加入新的 cache_recv_addr 输出信号,将其接入到 CPU 中,发挥原先 inst_addr_ok 的作用

错误 2:pc 值在流水线传递中异常被覆盖
(1)错误现象:WB 级 pc 与 reference 不符
(2)分析定位过程:通过观察仿真波形,发现是由于流水线阻塞期间 preIF 级因流水线控制信号异常拉高,进而错误发起取值请求,导致 pc 更新,覆盖了原先的正确 pc 及其指令
(3)错误原因:引入 ICache 后,未对 IF 级流水线相关控制信号做出正确调整,导致产生错误
(4)修正效果:对 pipe_allowin,pipe_valid 等流水线控制信号进行赋值逻辑调整,使 CPU 正常工作

错误 3:除法器运算结果异常
(1)错误现象:除法 IP 核输出错误运算结果
(2)分析定位过程:通过观察仿真波形,发现错误 pc 中向除法 IP 核输入的信号和数据正确,但输出结果错误,向前定位,发现此前异常向除法 IP 核发起了运算请求
(3)错误原因:在异常处理后刷新流水线时,未刷新除法器相关控制信号寄存器,导致额外发起异常运算请求,进而覆盖正确结果
(4)修正效果:新增除法器相关控制信号寄存器相关刷新逻辑,保证除法器正常工作

exp22:
错误 1:ld 指令错误取出全零数据
(1)错误现象:ld 指令错误取出全零数据,与 reference 不符
(2)分析定位过程:通过观察仿真波形,发现是由于此前在同一地址的 st 指令未能成功写入数据
(3)错误原因:接入 bridge 的 wstrb 掩码出现错误
(4)修正效果:接入正确 wstrb 掩码后问题解决

错误 2:流水线在 MEM 级异常阻塞
(1)错误现象:流水线在 MEM 级永久阻塞
(2)分析定位过程:通过观察仿真波形,发现是由于 bridge 在 DCache 写时,wvalid 和 wready 只进行了一次握手,进而导致 bvalid 和 bready 未进行握手,访存过程卡死
(3)错误原因:由于引入 DCache 后,写入一个 cache 行时需要进行四次 wvalid 和 wready 握手,才可以正常进行访存过程
(4)修正效果:修改 wvalid 赋值逻辑,保证访存正常进行

错误 3:DCache 写重填过程无法正常工作
(1)错误现象:流水线在 MEM 级永久阻塞
(2)分析定位过程:通过观察仿真波形,发现是由于 bridge 在 DCache 写重填时,只发起了一次写请求,而未进行后续读请求
(3)错误原因:在原先的 AXI-bridge 的数据部分访存设计中,并未考虑在同一个流水级中进行多次访存操作,导致只会向内存发起一次数据类型访存请求
(4)修正效果:修改 data_sram_req 赋值逻辑,但仍然存在问题,两次访存请求均为写请求

错误 4:DCache 写重填过程无法正常工作
(1)错误现象:bridge 在 DCache 写重填过程中向内存发起两次写请求

(2)分析定位过程:通过观察仿真波形,发现是由于 bridge 在 DCache 写重填时,只发起了两次写请求,而不是一次写请求和一次读请求
(3)错误原因:在原先的 AXI-bridge 的数据部分访存设计中,并未考虑在同一个流水级中进行读请求和写请求两类请求,导致在发起第一次写请求后 wr 信号未发生变化,进而第二次访存请求仍为写请求
(4)修正效果:添加新的控制信号 write_to_read,协助 wr 信号进行数据访存控制

错误 5:DCache 写重填过程流水线异常流动
(1)错误现象:流水线在 MEM 级异常流动
(2)分析定位过程:通过观察仿真波形,发现是由于 bridge 在 DCache 写重填时,只发起了发起第一次写请求后,流水级相关控制信号就允许流动
(3)错误原因:在原先的 MEM 级别流水线设计中,并未考虑在同一个流水级中进行两次数据类访存请求,导致在发起第一次写请求后 CPU 认为 MEM 级工作完成,流水线可以正常流动
(4)修正效果:修改 MEM 级流水线控制信号赋值逻辑,保证 CPU 在 DCache 写重填过程中正常工作

错误 6:项目在第一次仿真或上板运行时正常 Pass,reset 后出现异常
(1)错误现象:项目在 reset 后第一次取指取出异常数据
(2)分析定位过程:通过修改 testbench 文件,发现在第一次仿真时,在内存不应该允许写入的地址写入了数据
(3)错误原因:cache 设计中忽略了写入时对有效位和脏位的判断
(4)修正效果:对 cache 写过程中加入了对有效位和脏位的判断

• **exp23**。

1. 本实验中,遇到最困难的问题是对 DCache 中写回数据的控制,完成 code==5'b01001 这一测试耗费了 10 多个小时,因为本人对于 DCache 状态机和写回机制的研究均不足,并且逻辑信号中存在着大量的意想不到的联动,在调试过程中不断产生新的 bug。花费时间最长的 bug 是 DCache 已经正确发出了写回请求所需的信号,但读地址时仍然读不到数据。经过排查波形,发现是该功能没有适配 4 字同时写回的 DCache 写回机制,导致其在 bridge 中的写回数据和地址发生错误,读取时出现问题。

2. 第二个问题是在 op[4:3]==2'b10 时,需要把虚址放入完整的数据通路中,但是 ICache 发生了严重时序问题,出现了两路 Cache 同时命中,同时内存返回重填数据的糟糕情况。这个问题的解决过程是经过努力地查看波形与思考,还是没有解决问题,于是去睡觉。第二天早上醒来重新分析波形,搞清了 tagv_ram 的运行机制,想到这种情况需要先读数再写入,需要将指令在 ICache 中的执行分为两拍进行,于是解决了问题。

3. 第三个问题是在 ICache 的 cacop 加入 tlb 异常时发现 ICache 之前从未触发过正确的重填例外,并且将 cacop 指令触发的异常连接到数据通路后,异常返回入口 PC 总是滞后一级,导致异常处理错误。通过分析设计架构,发现是本指令的执行在 ID 级,但异常标在 IF 级导致的,于是将异常标记提前到 ID 级,解决了问题。

六、 实验总结

本实验难度较高,花费了小组三位同学大量时间,锻炼了我们的团队合作能力和解决问题的能力,使我们对于 cache 的工作原理有了更深刻的理解。在实验中,我们遇到了许多问题,如 cache 的写回机制、异常处理、流水线控制等,通过不断地思考、查阅资料、分析波形,我们最终解决了这些问题。长时间的合作和共同 debug 也加深了我们之间的友谊,总体来讲本次实验我们收获颇丰。

七、 实验分工

exp20:张家玮
exp21、exp22:王泽黎

exp23、实验报告：武弋

以上分工为实验主要编写人员，大部分调试过程为组内三人共同完成。