中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929034 姓名：侯汝垚 专业：计算机科学与技术

实验序号：5.4 实验名称：高速缓存（Cache）设计

注 1：撰写此 Word 格式实验报告后以 PDF 格式保存 SERVE CloudIDE 的 /home/serve-ide/cod-lab/reports 目录下（注意：reports 全部小写）。文件命名规则：prjN.pdf，其中“prj”和后缀名“pdf”为小写，“N”为 1 至 4 的阿拉伯数字。例如：prj1.pdf。PDF 文件大小应控制在 5MB 以内。此外，实验项目 5 包含多个选做内容，每个选做实验应提交各自的实验报告文件，文件命名规则：prj5-projectname.pdf，其中“-”为英文标点符号的短横线。文件命名举例：prj5-dma.pdf。具体要求详见实验项目 5 讲义。

注 2：使用 git add 及 git commit 命令将实验报告 PDF 文件添加到本地仓库 master 分支，并通过 git push 推送到 SERVE GitLab 远程仓库 master 分支（具体命令详见实验报告）。

注 3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

一、 逻辑电路结构与仿真波形的截图及说明

1. I-cache 设计

指令 cache 只需考虑读不用考虑写，相较而言比较简单，ppt 已经给出具体的状态转移图，主要工作在于如何判断 hit 或者 miss，如何选出 hit 的数据以及如何在 miss 时完成突发传输。

首先用 generate-for 语句生成这四路 cache_array 以及对应的 tag_array。

```
wire [23:0] rd_tag [3:0];
wire [255:0] rd_data [3:0];
wire [3:0] cache_wen;

genvar way_i;
generate
        for (way_i = 0; way_i < `CACHE_WAY; way_i = way_i + 1) begin: tag_way
                tag_array tag(
                        .clk(clk),
                        .waddr(req_pc[7:5]),
                        .raddr(req_pc[7:5]),
                        .wen(cache_wen[way_i] && current_state[6]),
                        .wdata(req_pc[31:8]),
                        .rdata(rd_tag[way_i])
                );
        end
endgenerate

genvar way_j;
generate
        for (way_j = 0; way_j < `CACHE_WAY; way_j = way_j + 1) begin: data_way
                data_array data(
                        .clk(clk),
                        .waddr(req_pc[7:5]),
                        .raddr(req_pc[7:5]),
                        .wen(cache_wen[way_j] && current_state[6]),
                        .wdata(write_data),
                        .rdata(rd_data[way_j])
                );
        end
endgenerate
```

每组有四个 cache 块，因此可以用一个四位的数据来表示当前有效的是哪几位，命中的是哪一位或者没有名字，途中的 valid_way 的四位分别对应四路，有效为 1，无效为 0，hit_way 同理。以此可以知道 hit 时 hit_way 至少有一路为 1，miss 时 hit_way 为全 0。

```
wire [3:0] valid_way = {valid_array[3][cache_set_addr],
                        valid_array[2][cache_set_addr],
                        valid_array[1][cache_set_addr],
                        valid_array[0][cache_set_addr]};

wire [3:0] hit_way = {rd_tag[3] == cache_tag && valid_way[3],
                      rd_tag[2] == cache_tag && valid_way[2],
                      rd_tag[1] == cache_tag && valid_way[1],
                      rd_tag[0] == cache_tag && valid_way[0]};
```

```
assign read_hit = |hit_way;
assign read_miss = ~read_hit;
```

Hit 以后 read_hit 即可作为选择信号，在四路里面选出命中的那一路，然后根据块内地址将数据返回。

```
wire [255:0] valid_data = {256{hit_way[0]}} & rd_data[0]
                        | {256{hit_way[1]}} & rd_data[1]
                        | {256{hit_way[2]}} & rd_data[2]
                        | {256{hit_way[3]}} & rd_data[3]
                        | {256{read_miss }} & write_data;

wire [7:0] set_offset = {
        cache_in_addr[4:2] == 3'b111,
        cache_in_addr[4:2] == 3'b110,
        cache_in_addr[4:2] == 3'b101,
        cache_in_addr[4:2] == 3'b100,
        cache_in_addr[4:2] == 3'b011,
        cache_in_addr[4:2] == 3'b010,
        cache_in_addr[4:2] == 3'b001,
        cache_in_addr[4:2] == 3'b000
};

assign read_hit = |hit_way;
assign read_miss = ~read_hit;
wire [31:0] read_hit_data = {32{set_offset[0]}} & valid_data[ 31: 0]
                          | {32{set_offset[1]}} & valid_data[ 63:32]
                          | {32{set_offset[2]}} & valid_data[ 95:64]
                          | {32{set_offset[3]}} & valid_data[127:96]
                          | {32{set_offset[4]}} & valid_data[159:128]
                          | {32{set_offset[5]}} & valid_data[191:160]
                          | {32{set_offset[6]}} & valid_data[223:192]
                          | {32{set_offset[7]}} & valid_data[255:224];
```

若不命中则需要替换。用一个排队器选出一个空的 cache 块。若某个组已经满，则用 LRU 算法找出需要替换的 cache 块。LRU 算法通过一个 4*4 矩阵实现，对应每组的四个块，每次命中就将对应列全部清零，对应行除了对应命中块的那个位置外的三个位置置 0，这样当 cache 块满后，对应行全为 0 的 cache 块就是最近最少使用的 cache 块。

```
wire [3:0] empty_block = {
        ~valid_array[3][cache_set_addr],
        ~valid_array[2][cache_set_addr],
        ~valid_array[1][cache_set_addr],
        ~valid_array[0][cache_set_addr]
};

wire block_full = ~(|empty_block);

// write way when cache is empty
wire [3:0] empty_cache_addr = {
        empty_block[3],
        ~empty_block[3] && empty_block[2],
        ~empty_block[3] && ~empty_block[2] && empty_block[1],
        ~empty_block[3] && ~empty_block[2] && ~empty_block[1] && empty_block[0]
};
```

```
integer matrix_i, matrix_j;
reg [3:0] visit_matrix [7:0][3:0];
always @(posedge clk) begin
        if (rst) begin
                for (matrix_i = 0; matrix_i < 8; matrix_i = matrix_i + 1) begin
                        for (matrix_j = 0; matrix_j < 3; matrix_j = matrix_j + 1) begin
                                visit_matrix[matrix_i][matrix_j] <= 4'b0;
                        end
                end
        end
        else if (current_state[7] && from_cpu_cache_rsp_ready) begin
                case (hit_way)
                        4'b0001: begin
                                visit_matrix[req_pc[7:5]][0] <= 4'b1110;
                                visit_matrix[req_pc[7:5]][1] <= visit_matrix[req_pc[7:5]][1] & 4'b1110;
                                visit_matrix[req_pc[7:5]][2] <= visit_matrix[req_pc[7:5]][2] & 4'b1110;
                                visit_matrix[req_pc[7:5]][3] <= visit_matrix[req_pc[7:5]][3] & 4'b1110;
                        end
                        4'b0010: begin
                                visit_matrix[req_pc[7:5]][0] <= visit_matrix[req_pc[7:5]][0] & 4'b1101;
                                visit_matrix[req_pc[7:5]][1] <= 4'b1101;
                                visit_matrix[req_pc[7:5]][2] <= visit_matrix[req_pc[7:5]][2] & 4'b1101;
                                visit_matrix[req_pc[7:5]][3] <= visit_matrix[req_pc[7:5]][3] & 4'b1101;
                        end
                        4'b0100: begin
                                visit_matrix[req_pc[7:5]][0] <= visit_matrix[req_pc[7:5]][0] & 4'b1011;
                                visit_matrix[req_pc[7:5]][1] <= visit_matrix[req_pc[7:5]][1] & 4'b1011;
                                visit_matrix[req_pc[7:5]][2] <= 4'b1011;
                                visit_matrix[req_pc[7:5]][3] <= visit_matrix[req_pc[7:5]][3] & 4'b1011;
                        end
                        4'b1000: begin
                                visit_matrix[req_pc[7:5]][0] <= visit_matrix[req_pc[7:5]][0] & 4'b0111;
                                visit_matrix[req_pc[7:5]][1] <= visit_matrix[req_pc[7:5]][1] & 4'b0111;
                                visit_matrix[req_pc[7:5]][2] <= visit_matrix[req_pc[7:5]][2] & 4'b0111;
                                visit_matrix[req_pc[7:5]][3] <= 4'b0111;
                        end
                        default: begin
                                visit_matrix[req_pc[7:5]][0] <= visit_matrix[req_pc[7:5]][0];
                                visit_matrix[req_pc[7:5]][1] <= visit_matrix[req_pc[7:5]][1];
                                visit_matrix[req_pc[7:5]][2] <= visit_matrix[req_pc[7:5]][2];
                                visit_matrix[req_pc[7:5]][3] <= visit_matrix[req_pc[7:5]][3];
                        end
                endcase
        end
end
```

突发传输过程中用一个 256 位宽的寄存器保存读出来的数据，用一个计数器标记需要替换然后统一替换。

```
reg [2:0] from_mem_rd_counter;
always @(posedge clk) begin
        if (rst)
                from_mem_rd_counter <= 3'b0;
        else if (from_mem_rd_rsp_valid && to_mem_rd_rsp_ready)
                from_mem_rd_counter <= from_mem_rd_counter + 1'b1;
end

reg [255:0] write_data;
always @(posedge clk) begin
        if (rst)
                write_data <= 256'b0;
        else if (from_mem_rd_rsp_valid && to_mem_rd_rsp_ready) begin
                case (from_mem_rd_counter)
                        3'b000: write_data[ 31:  0] <= from_mem_rd_rsp_data;
                        3'b001: write_data[ 63: 32] <= from_mem_rd_rsp_data;
                        3'b010: write_data[ 95: 64] <= from_mem_rd_rsp_data;
                        3'b011: write_data[127: 96] <= from_mem_rd_rsp_data;
                        3'b100: write_data[159:128] <= from_mem_rd_rsp_data;
                        3'b101: write_data[191:160] <= from_mem_rd_rsp_data;
                        3'b110: write_data[223:192] <= from_mem_rd_rsp_data;
                        3'b111: write_data[255:224] <= from_mem_rd_rsp_data;
                endcase
        end
end
```

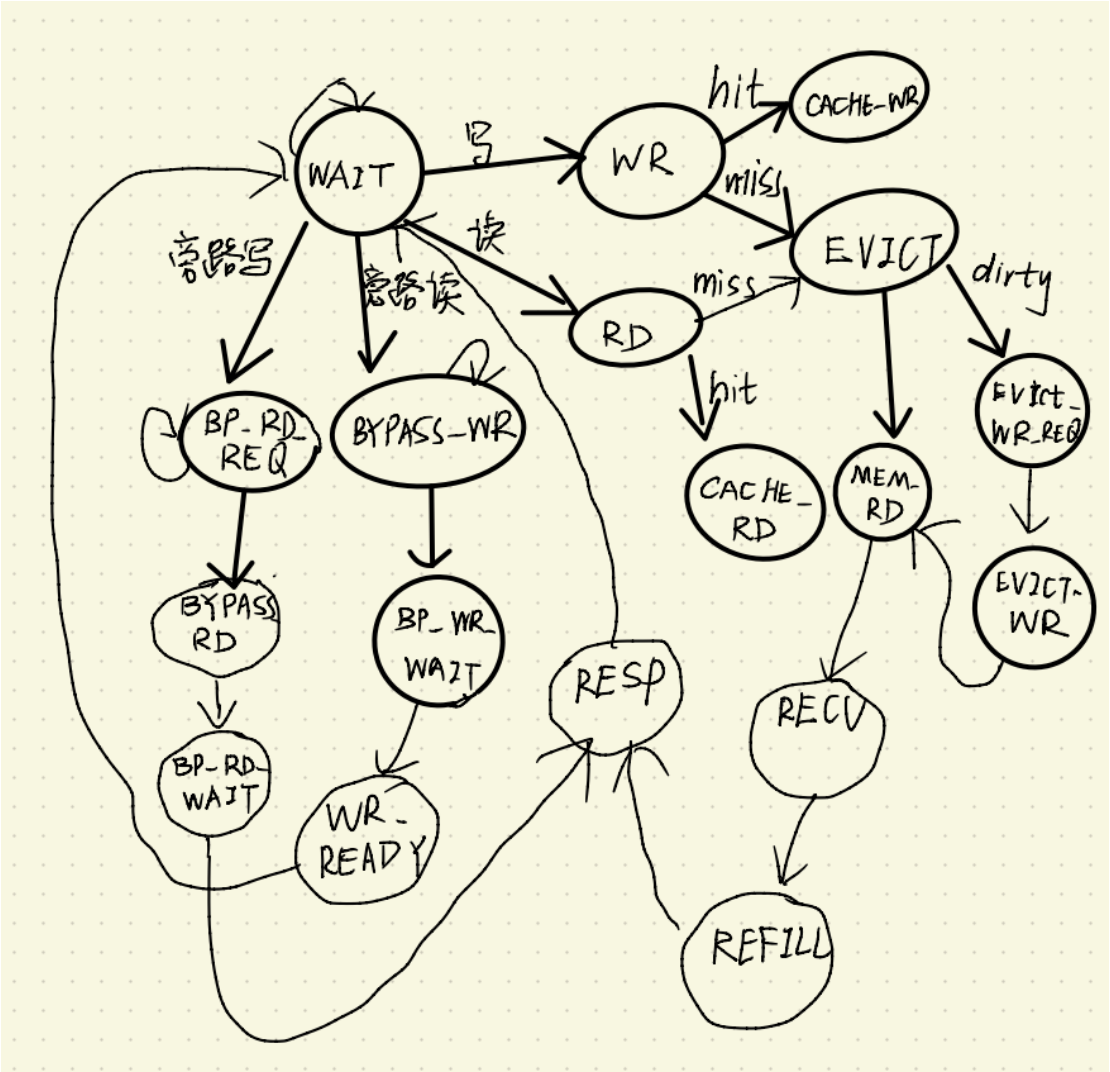
2. D-cache 设计

D-cache 需要额外处理不可缓存区域的读写以及内存写操作。

首先修改状态机，其中共有 19 个状态，各状态功能如下。对于不可缓存区域只需按照正常访存完成。

```
//TODO: Please add your D-Cache code here
    localparam WAIT         = 18'b000000000000000001,
               WR           = 18'b000000000000000010,
               RD           = 18'b000000000000000100,
               BYPASS_WR    = 18'b000000000000001000,
               BYPASS_RD    = 18'b000000000000010000,
               CACHE_WR     = 18'b000000000000100000,
               CACHE_RD     = 18'b000000000001000000,
               EVICT        = 18'b000000000010000000,
               EVICT_WR_REQ = 18'b000000000100000000,
               EVICT_WR     = 18'b000000001000000000,
               MEM_RD       = 18'b000000010000000000,
               RECV         = 18'b000000100000000000,
               REFILL       = 18'b000001000000000000,
               RESP         = 18'b000010000000000000,
               BP_RD_WAIT   = 18'b000100000000000000,
               BP_WR_WAIT   = 18'b001000000000000000,
               WR_READY     = 18'b010000000000000000,
               BP_RD_REQ    = 18'b100000000000000000;
```

WAIT	等待
WR	接受写请求
RD	接受读请求
BYPASS_WR	接受旁路写请求
BYPASS_RD	接受旁路读请求
CACHE_WR	cache块写
CACHE_RD	cache块读
EVICT	替换
EVICT_WR_REQ	发送突发传输内存写请求
EVICT_WR	突发传输内存写
MEM_RD	读内存
RECV	突发传输内存读
REFILL	充填
RESP	应答
BP_RD_WAIT	旁路读等待CPU响应
BP_WR_WAIT	旁路写等待CPU响应
WR_READY	拉高写请求Ready
BP_RD_REQ	发送旁路读请求



旁路包括 0-0x1ff 即 io 的地址，即 0x40000000 以外的地址。

```
wire bypass = ~(|req_addr[31:5]) || (req_addr[30] || req_addr[31] || req_addr[31]);
```

Cache 读操作同 I-cache，写操作同样分为 hit 和 miss。若 hit 只需修改对应的 cache 块。若 miss，先通过突发传输将对应数据写入 cache 块，然后再执行 cache 写。替换算法同 i-cache 中的 LRU。内存写需要用寄存器存需要写的数据以及 strb。

```
reg [31:0] wdata;
always @(posedge clk) begin
        if (rst)
                wdata <= 32'b0;
        else if (from_cpu_mem_req_valid && current_state[0] && from_cpu_mem_req)
                wdata <= from_cpu_mem_req_wdata;
end

reg [3:0] wstrb;
always @(posedge clk) begin
        if (rst)
                wstrb <= 4'b0;
        else if (from_cpu_mem_req_valid && current_state[0] && from_cpu_mem_req)
                wstrb <= from_cpu_mem_req_wstrb;
end
```

替换过程需要额外考虑是否 dirty，若 dirty 则进入 EVICT_WR_REQ 状态将 dirty

数据写回，否则安装 I-cache 的方法正常替换。

```
EVICT: begin
        if (dirty)
                next_state = EVICT_WR_REQ;
        else
                next_state = MEM_RD;
end
```

3. Cache 命中率统计

因为 cache 块内没有接入 CPU 的计数器，可以在行为仿真中统计几个简答测试案例的 cache 命中率，用$display 打印。挑选了几个有代表性的 benchmark，可以看出 cache 命中率非常客观，都大于 95%，对性能提升很大。

	I-cache	d-cache写	d-cache读
add-longlong	1859/1868	6/7	371/390
bit	414/432	16/18	414/432
fact	23527/23539	141/151	149/150
recursion	57176/57197	875/877	449/457
prime	34808/34816	4/5	12/14
总	117784/117852	1042/1058	1395/1443
命中率	99.94%	98.49%	96.67%

从 coremark 和 dhrystone 的分数来看，icache 对性能的提升最大，提升近 15 倍，而 dcache 提升比较有限，猜测原因在于因为每条指令的执行都需要用到 icache，而访存指令在程序中占比一般比较少，因此 dcache 带来的性能提升不大。

	coremark	dhrystone
流水线	93089124us	454
流水线 + icache	6646296us	4034
流水线 + icache + dcache	6441674us	5691
流水线 + icache + dcache + freq*2	3222789us	10554

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）

1. 如何获得 memory 类型变量的波形?

Tag_array 和 data_array 都是深度为 4memory 型变量，似乎 GTKwave 不能直接获得波形，通过搜索找到。

2. Cache 无法写入

查看波形时发现明明写使能已经拉高，但是写入后数据依然为 X，经微信群中的同学提醒才发现原因在于实验框架已经写好的代码中移位和减运算的优先级出错，增加括号即可。

```
reg [`TARRAY_DATA_WIDTH-1:0] array[ (1 << `TARRAY_ADDR_WIDTH) - 1 : 0];
```

3. LRU 替换算法的实现

参考 https://blog.csdn.net/qq_70829439/article/details/127640013，采用矩阵法实现。

4. 内存写什么时候返回 ready?

实验框架中内存写操作是半互锁的，因此只有一次握手，对于内存写操作若在初始状态就拉高 ready 信号，在访存请求握手成功后 cpu 就认为存数指令已经完成，所以内存每次内存写只用一个周期就实现，这也会导致 trace_end 之前就已经往 0xC 写 0，行为仿真无法停止。解决办法是将内存写和读分开操作，内存读必须完全写完后才返回 ready。

5. 水仙花行为仿真超时。

除了行为仿真中的水仙花外其他所有测试都能通过，包括 fpga_run 中的所有测试，同时 fpga_emu 中水仙花也能正常通过。在行为仿真中打断点让它能够停止，多次测试发现仿真停在了最后几条汇编指令，一旦从 nemu_assert 返回后，行为仿真就无法停止，即使强行提交错误的指令也无法报错。因此这个 bug 测试了很久也无法找到根源，因为行为仿真一旦到那个点之后就无法停下，那条指令附加也没有访存指令，理论上 d-cache 应该不会影响，但是摘下 d-cache 后就正常通过。多次测试后我猜测问题可能是我的 PC 跳转逻辑有些问题。我最开始的设计是 ID 阶段如果需要跳转就直接通过组合逻辑改变 PC，使得当前访存请求的指令是正确的，这样分支预测失败时只损失一条指令，就是 IW 中的指令。这是一个比较激进的策略，PC 的逻辑也比较混乱，可能这里存在一些没有考虑到的问题，我修改成了比较稳妥的策略，就是直接将 IF 和 IW 对应的指令无效，之后水仙花能够正常通过行为仿真。

三、 在课后，你花费了大约 30 小时完成此次实验。

四、 对于此次实验的心得、感受和建议

I-cache 相对来说比较简单，主要难点在于判断 hit、miss、选出 hit 的数据以及 cache 组满的时候的替换算法的实现，D-cache 的难点在于对内存写的处理。在亲自用 Verilog 实现了组相联 cache 后对理论课上讲的知识也有了更深的理解。希望可以将内存写操作改为全互锁，这样在 cache 和流水线结合后可以避免一些问题。也建议可以为 cache 增加一些计数器的接口，可以统计命中率，便于对比不同替换算法如 FIFO、LRU、LRU-2 的性格。