

# 中国科学院大学

## 《计算机组成原理(研讨课)》实验报告

姓名 韦欣池 学号 2022K8009907004 专业 计算机科学与技术  
实验项目编号 5.2 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 推送到实验课 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 (比如 Verilog HDL 关键代码段 {包含注释} 及其对应的逻辑电路结构图、相应信号的仿真波形和信号变化的说明等)

- icache 设计。

关于 icache 的设计,主要是按照讲义中的状态转移图来设计:

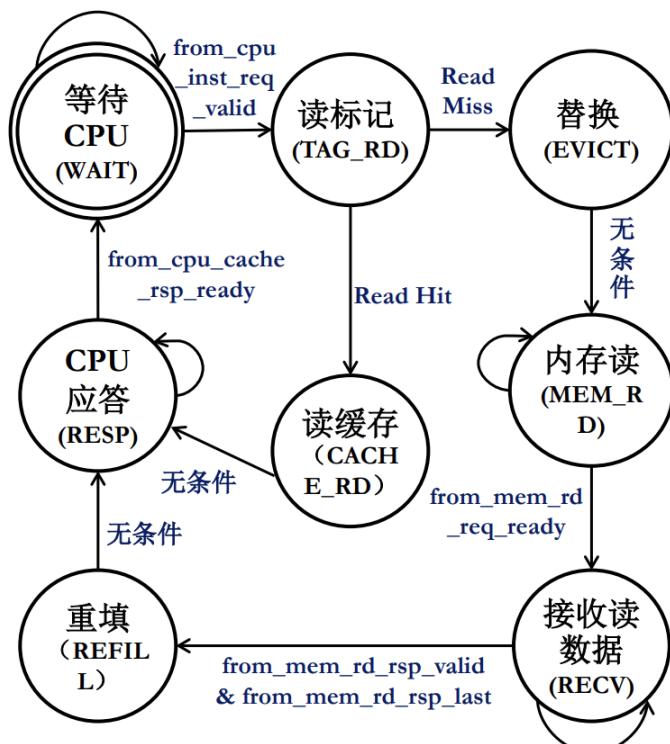


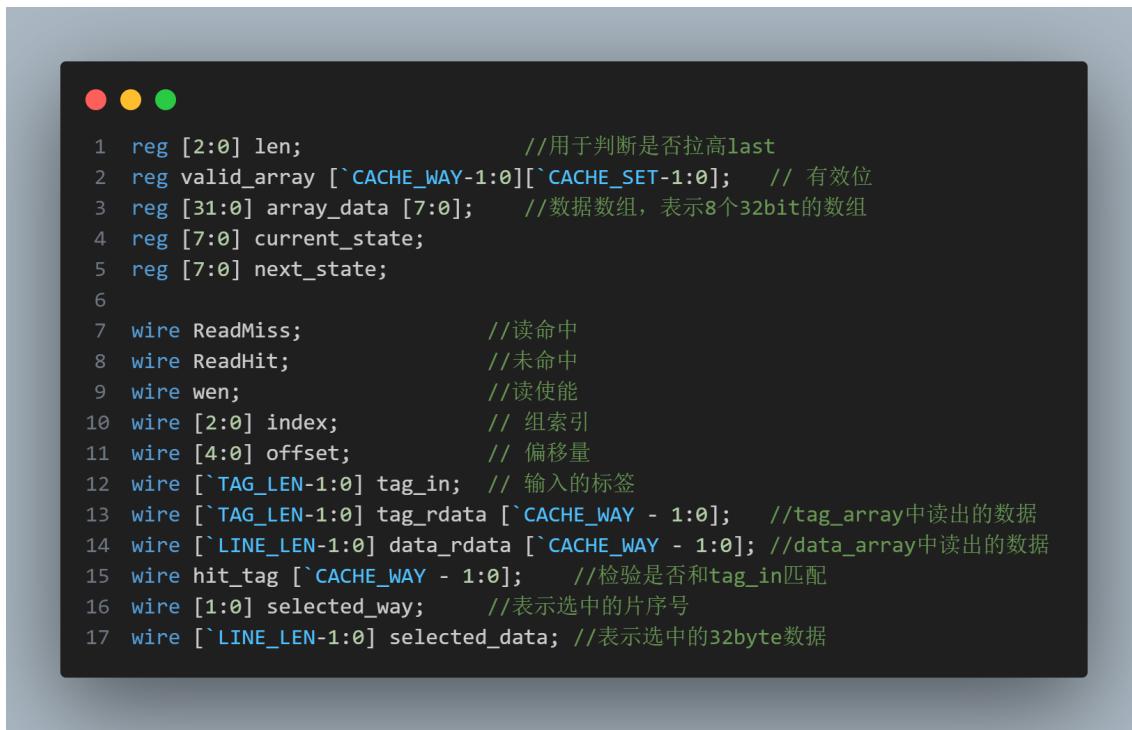
图 1: icache 状态转移图

对应的状态含义以及相应的操作如下：

- **WAIT:** 拉高to\_cpu\_inst\_req\_ready信号
- **TAG\_RD:** 根据输入地址index域，读出4路valid+tag，并与输入地址的tag域比较，产生Read Hit/Read miss信号
- **CACHE\_RD:** 从命中的cache block中读出32 byte，并根据输入地址的offset域，选择出要返回的指令码
- **RESP:** 拉高to\_cpu\_cache\_rsp\_valid端口信号，并将要返回的指令码输出到to\_cpu\_cache\_rsp\_data端口
- **EVICT:** 根据替换算法，从4路cache block中选择一个进行替换，将被替换block对应的valid清0
- **MEM\_RD:** 向内存发送输入请求所在的cache block地址（32 byte对齐地址），拉高to\_mem\_rd\_req\_valid
- **RECV:** 拉高to\_mem\_rd\_rsp\_ready。每当from\_mem\_rd\_rsp\_valid拉高时，接收4-byte from\_mem\_rd\_rsp\_data，直至from\_mem\_rd\_rsp\_last标记的最后一个4-byte数据已接收
- **REFILL:** 将已收到的32-byte数据填入选中cache block，同时更新对应的tag 和valid，并根据输入地址offset域，返回指令码

图 2: icache 各状态含义及操作

然后我们根据该状态转移图以及代码中给出的输入输出，设置相应的变量：

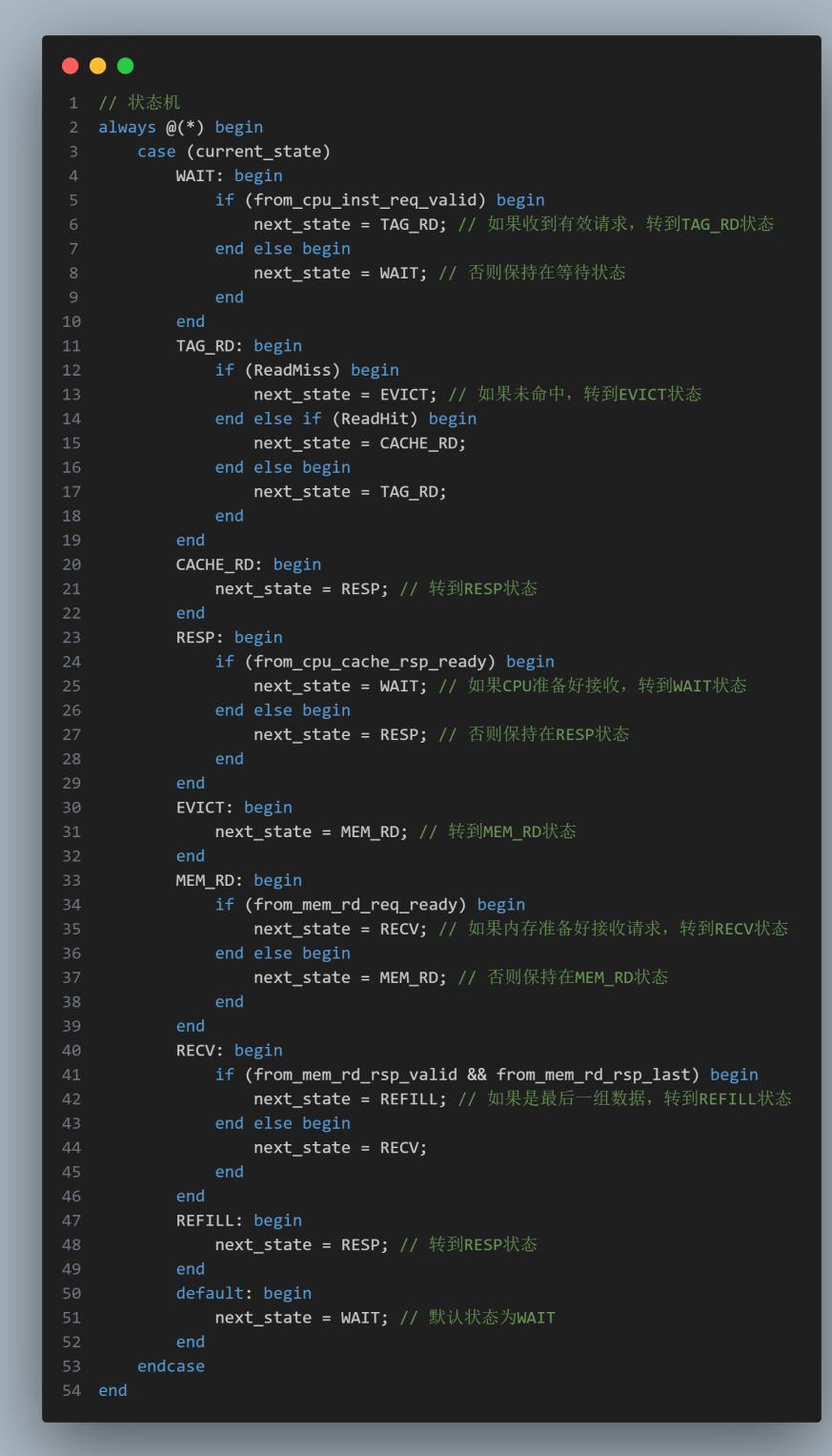


```
1 reg [2:0] len; //用于判断是否拉高last
2 reg valid_array [`CACHE_WAY-1:0][`CACHE_SET-1:0]; //有效位
3 reg [31:0] array_data [7:0]; //数据数组，表示8个32bit的数据
4 reg [7:0] current_state;
5 reg [7:0] next_state;
6
7 wire ReadMiss; //读命中
8 wire ReadHit; //未命中
9 wire wen; //读使能
10 wire [2:0] index; //组索引
11 wire [4:0] offset; //偏移量
12 wire [`TAG_LEN-1:0] tag_in; //输入的标签
13 wire [`TAG_LEN-1:0] tag_rdata [`CACHE_WAY - 1:0]; //tag_array中读出的数据
14 wire [`LINE_LEN-1:0] data_rdata [`CACHE_WAY - 1:0]; //data_array中读出的数据
15 wire hit_tag [`CACHE_WAY - 1:0]; //检验是否和tag_in匹配
16 wire [1:0] selected_way; //表示选中的片序号
17 wire [`LINE_LEN-1:0] selected_data; //表示选中的32byte数据
```

图 3: icache 中设置的相应的变量

图中设置的变量分为判断类和数组类。判断类如判断是否读命中等,用于在 icache、内存和 cpu 直接接口传输信号或数据时进行控制。数组类用于存放对应指令、判断结果等。

然后我们使用独热码对这 8 种状态进行编码,然后按照状态转移图完成相应代码的编写:



```
1 // 状态机
2 always @(*) begin
3     case (current_state)
4         WAIT: begin
5             if (from_cpu_inst_req_valid) begin
6                 next_state = TAG_RD; // 如果收到有效请求, 转到TAG_RD状态
7             end else begin
8                 next_state = WAIT; // 否则保持在等待状态
9             end
10        end
11        TAG_RD: begin
12            if (ReadMiss) begin
13                next_state = EVICT; // 如果未命中, 转到EVICT状态
14            end else if (ReadHit) begin
15                next_state = CACHE_RD;
16            end else begin
17                next_state = TAG_RD;
18            end
19        end
20        CACHE_RD: begin
21            next_state = RESP; // 转到RESP状态
22        end
23        RESP: begin
24            if (from_cpu_cache_rsp_ready) begin
25                next_state = WAIT; // 如果CPU准备好接收, 转到WAIT状态
26            end else begin
27                next_state = RESP; // 否则保持在RESP状态
28            end
29        end
30        EVICT: begin
31            next_state = MEM_RD; // 转到MEM_RD状态
32        end
33        MEM_RD: begin
34            if (from_mem_rd_req_ready) begin
35                next_state = RECV; // 如果内存准备好接收请求, 转到RECV状态
36            end else begin
37                next_state = MEM_RD; // 否则保持在MEM_RD状态
38            end
39        end
40        RECV: begin
41            if (from_mem_rd_rsp_valid && from_mem_rd_rsp_last) begin
42                next_state = REFILL; // 如果是最后一组数据, 转到REFILL状态
43            end else begin
44                next_state = RECV;
45            end
46        end
47        REFILL: begin
48            next_state = RESP; // 转到RESP状态
49        end
50        default: begin
51            next_state = WAIT; // 默认状态为WAIT
52        end
53    endcase
54 end
```

图 4: icache 中状态转移图的代码设计

然后是我们需要将 icache 和对应的 tag\_array 和 data\_array 进行连接:

```

1  /*创建tag_array和data_array*/
2  genvar i_way; //循环变量
3  generate
4      for (i_way=0;i_way<`CACHE_WAY;i_way=i_way+1) begin : ways
5          tag_array tag_array_module(
6              .clk    (clk),
7              .waddr (index),
8              .raddr (index),
9              .wen   (wen),
10             .wdata (tag_in),
11             .rdata (tag_rdata[i_way])
12         );
13         assign hit_tag[i_way] = (valid_array[i_way][index] && (tag_rdata[i_way] == tag_in));
14         data_array data_array_module(
15             .clk    (clk),
16             .waddr (index),
17             .raddr (index),
18             .wen   (wen),
19             .wdata ({array_data[7],array_data[6],array_data[5],array_data[4],
20                      array_data[3],array_data[2],array_data[1],array_data[0]}),
21             .rdata (data_rdata[i_way])
22         );
23     end
24 endgenerate

```

图 5: 对应接口代码设计

这里我使用 generate 语块来进行连接, 是因为本实验的 icache 是包含四片的相当于二维数组, 因此在连接的时候需要分别连接, 所以使用了 for 循环来完成。

然后我们还需要完成的是当读 cache 没有命中的时候的操作, 即从内存中读出数据并替换 cache 中某一处的数据, 然后再进行 icache 和 cpu 的传输。对于替换算法, 老师说可以任意实现, 这里由于时间限制, 我没有使用性能较为优异的近期最小使用频率算法(LRU 算法)或先入先出算法(FIFO 算法)等等, 而是直接替换固定位置, 在代码中我设置了常量 fixed\_replace\_way, 取其值为 0, 表示每次替换都替换第 0 片 icache 中对应位置的数据。

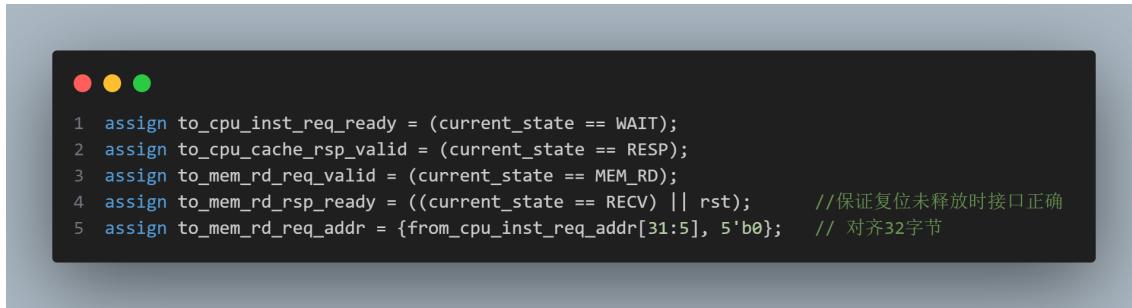
```

1  integer j_way; //循环变量
2  always @(posedge clk) begin
3      if (rst) begin
4          for (j_way=0;j_way<`CACHE_SET;j_way=j_way+1) begin
5              valid_array[0][j_way] <= 0;
6              valid_array[1][j_way] <= 0;
7              valid_array[2][j_way] <= 0;
8              valid_array[3][j_way] <= 0;
9          end
10     end else if (!rst && (current_state == EVICT)) begin //替换算法: 每次都是将第0片中对应的位置进行替换
11         valid_array[fixed_replace_way][index]<=1'b0; //将对应的valid置为0
12     end else if (!rst && (current_state == MEM_RD)) begin
13         len <= 3'b000;
14     end else if (!rst && (current_state == RECV) && from_mem_rd_rsp_valid) begin
15         array_data[len] <= from_mem_rd_rsp_data; //接收4-byte from_mem_rd_rsp_data
16         len <= len + 1; //直至from_mem_rd_rsp_last标记的最后一个4-byte数据已接收
17     end else if (!rst && (current_state == REFILL)) begin
18         valid_array[fixed_replace_way][index] <= 1'b1;
19     end
20 end

```

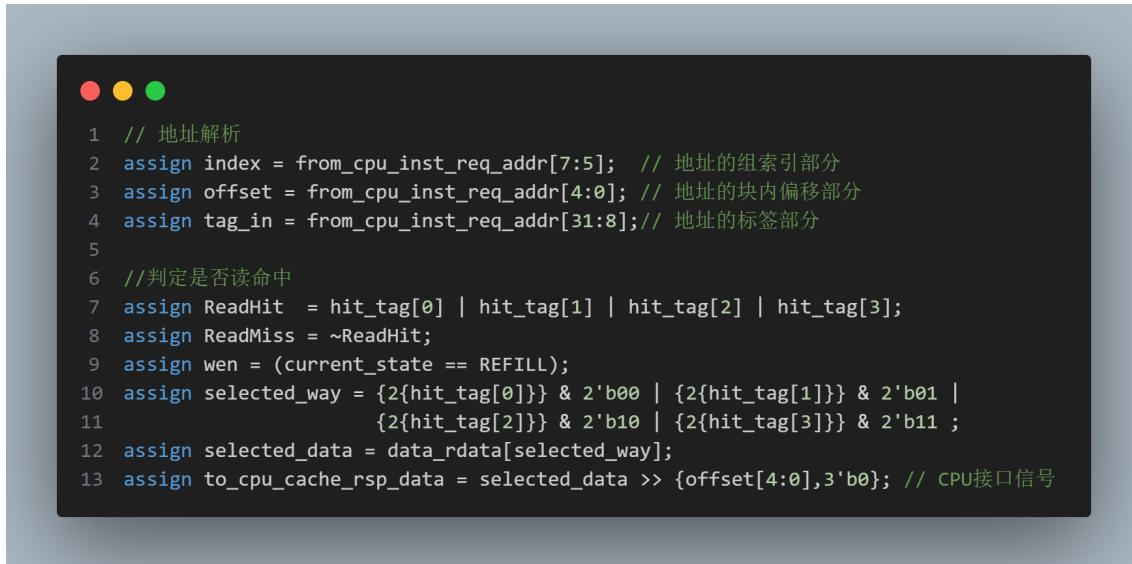
图 6: 对应替换策略代码设计

最后就是完成 icache\_module 中对应输出的赋值,以及我设置的对应的变量的赋值:



```
● ● ●
1 assign to_cpu_inst_req_ready = (current_state == WAIT);
2 assign to_cpu_cache_rsp_valid = (current_state == RESP);
3 assign to_mem_rd_req_valid = (current_state == MEM_RD);
4 assign to_mem_rd_rsp_ready = ((current_state == RECV) || rst);      //保证复位未释放时接口正确
5 assign to_mem_rd_req_addr = {from_cpu_inst_req_addr[31:5], 5'b0};    // 对齐32字节
```

图 7: 对应输出的赋值



```
● ● ●
1 // 地址解析
2 assign index = from_cpu_inst_req_addr[7:5]; // 地址的组索引部分
3 assign offset = from_cpu_inst_req_addr[4:0]; // 地址的块内偏移部分
4 assign tag_in = from_cpu_inst_req_addr[31:8];// 地址的标签部分
5
6 //判定是否读命中
7 assign ReadHit = hit_tag[0] | hit_tag[1] | hit_tag[2] | hit_tag[3];
8 assign ReadMiss = ~ReadHit;
9 assign wen = (current_state == REFILL);
10 assign selected_way = {2{hit_tag[0]}} & 2'b00 | {2{hit_tag[1]}} & 2'b01 |
11           {2{hit_tag[2]}} & 2'b10 | {2{hit_tag[3]}} & 2'b11 ;
12 assign selected_data = data_rdata[selected_way];
13 assign to_cpu_cache_rsp_data = selected_data >> {offset[4:0],3'b0}; // CPU接口信号
```

图 8: 对应变量的赋值

### • dcache 设计。

相比较 icache, dcache 需要完成的事情则要多出许多,首先需要判断是否可以使用 dcache,然后再判断是否命中,由于 dcache 包含读写两种操作,所以需要分别完成这两种操作,然后在 dcache 中的每个 set 都增加 1-bit 的 dirty 标记,用于标记已修改过的 cache block,也就是说,当操作为写操作时,我们首先将输入的数据存入 dcache,然后在我们需要再对该位置进行写入数据的时候,我们首先应该将原来的数据写回到内存中,然后再存放当前写操作的数据。这是我们需要额外设计的状态。

如果不可以使用 dcache,则需要进行旁路操作,即直接将 cpu 和内存进行读写操作。

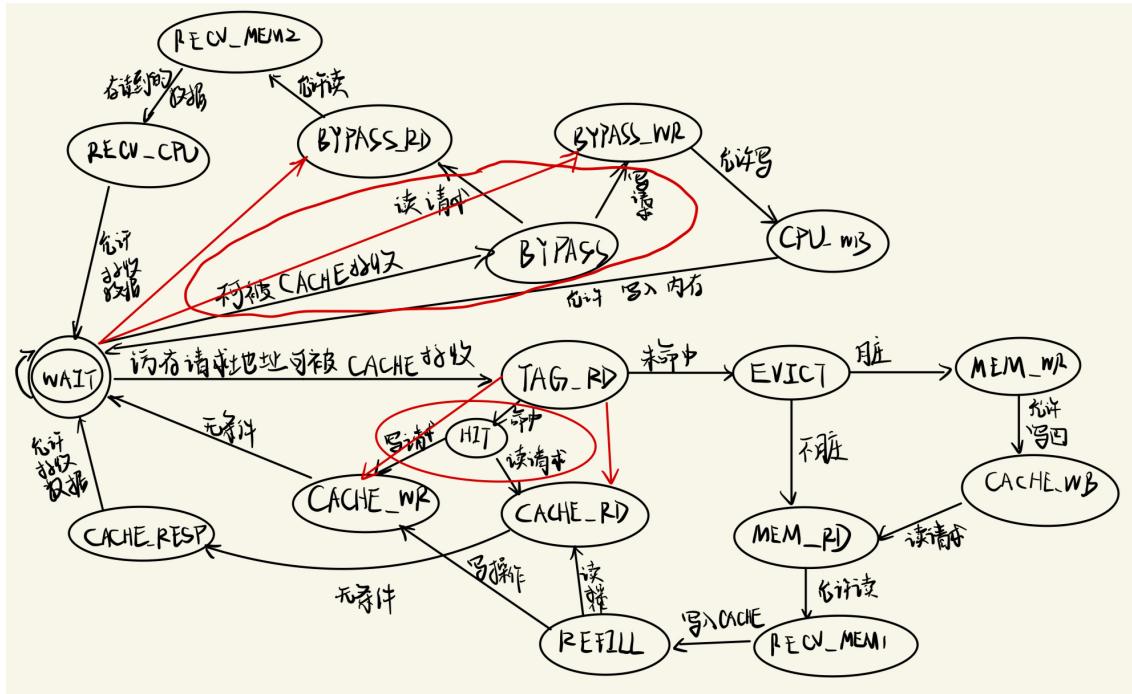


图 9: dcache 状态转移图设计

我一开始设计了 18 个状态,后来进行了优化,即将 BYPASS 和 HIT 两种状态去掉,直接将判断读写操作和判断是否命中或判断是否旁路操作合并,最终得到了如图所示 16 种状态,每种状态的含义及相应操作写在状态转移的过程中。

关于 dcache 的代码编写过程与 icache 类似,我同样设置了相应的变量来辅助我状态转移图和相应接口的完成:

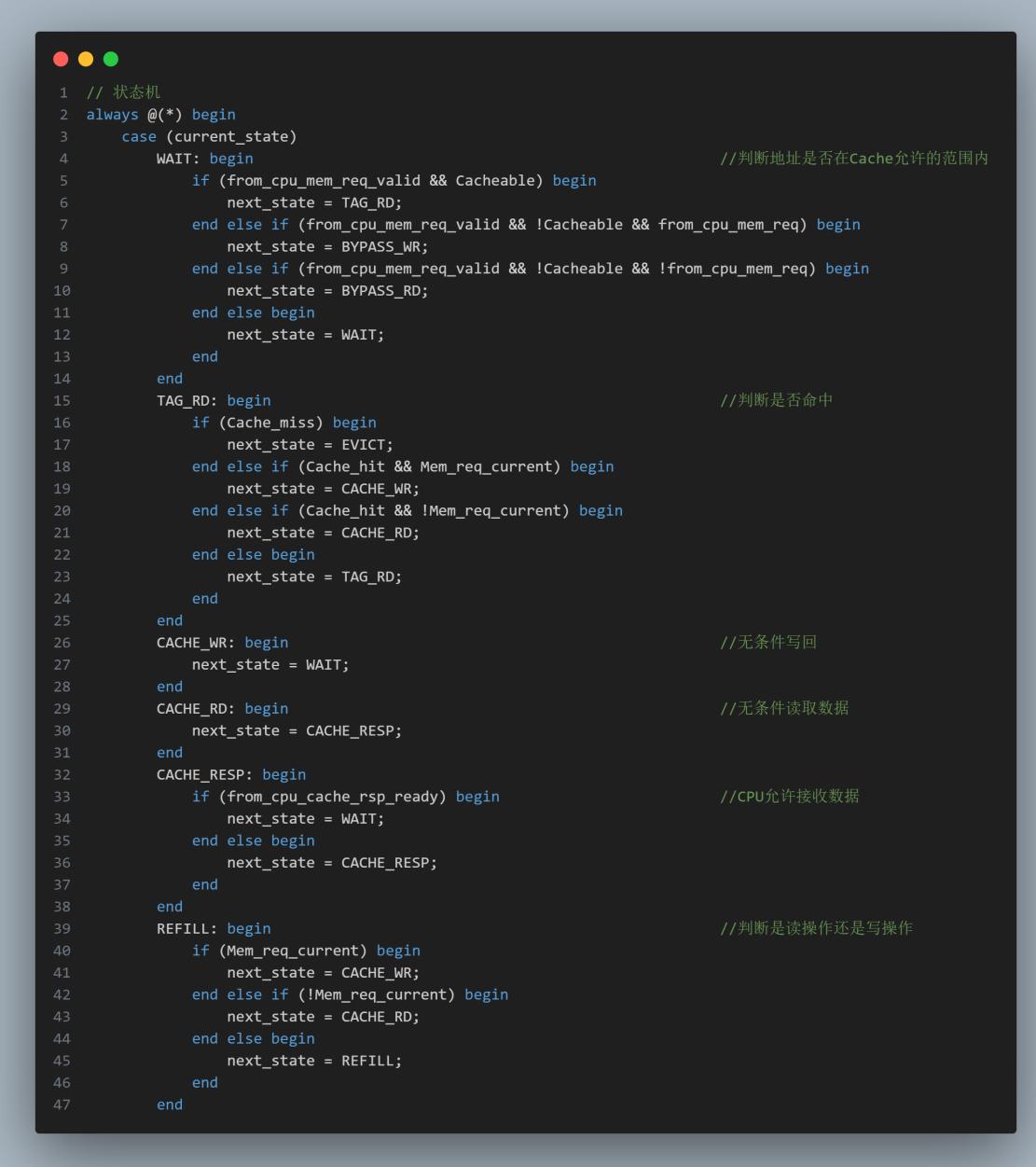
```

1 reg [2:0] len; // 用于判断是否拉高last
2 reg valid_array [`CACHE_WAY - 1:0][`CACHE_SET - 1:0]; // 有效位数组
3 reg dirty_array [`CACHE_WAY - 1:0][`CACHE_SET - 1:0]; // 脏数组
4 reg [31:0] array_data [7:0]; // 数据数组, 表示8个32bit的数据
5 reg [31:0] data_wb; // 从内存中写回的数据
6 reg [31:0] data_bypass; // 旁路写回CPU的数据
7 reg [31:0] Bypass_address_current; // 旁路读写地址
8 reg [31:0] Write_data_current; // 旁路写回内存的数据
9 reg [3:0] Write_strb_current; // 旁路写回使能
10 reg Mem_req_current; // 存储WAIT状态时的from_cpu_mem_req值
11 reg [17:0] current_state; // Cache使能
12 reg [17:0] next_state; // Cache命中
13
14 wire Cacheable; // Cache未命中
15 wire Cache_hit; // Cache脏
16 wire Cache_miss; // Cache脏
17 wire Cache_dirty; // 输入的标签
18 wire [TAG_LEN-1:0] tag_in; // 组索引
19 wire [2:0] index; // 偏移量
20 wire [4:0] offset; // 读写使能
21 wire wen [`CACHE_WAY - 1:0]; // tag_array中读出的数据
22 wire [ `TAG_LEN - 1:0] tag_rdata [ `CACHE_WAY - 1:0]; // data_array中读出的数据
23 wire [ `LINE_LEN - 1:0] data_rdata [ `CACHE_WAY - 1:0]; // 检验是否和tag_in匹配
24 wire hit_tag [ `CACHE_WAY - 1:0]; // 表示选中的片序号
25 wire [1:0] selected_way; // 表示选中的32byte数据
26 wire [ `LINE_LEN - 1:0] selected_data; // 表示替换的数据
27 wire [ `LINE_LEN - 1:0] replaced_data; // 表示替换的数据

```

图 10: dcache 中设置的相应变量

然后我们使用独热码对这 16 种状态进行编码,然后按照状态转移图完成相应代码的编写:



```
1 // 状态机
2 always @(*) begin
3     case (current_state)
4         WAIT: begin
5             if (from_cpu_mem_req_valid && Cacheable) begin
6                 next_state = TAG_RD;
7             end else if (from_cpu_mem_req_valid && !Cacheable && from_cpu_mem_req) begin
8                 next_state = BYPASS_WR;
9             end else if (from_cpu_mem_req_valid && !Cacheable && !from_cpu_mem_req) begin
10                next_state = BYPASS_RD;
11            end else begin
12                next_state = WAIT;
13            end
14        end
15        TAG_RD: begin
16            if (Cache_miss) begin
17                next_state = EVICT;
18            end else if (Cache_hit && Mem_req_current) begin
19                next_state = CACHE_WR;
20            end else if (Cache_hit && !Mem_req_current) begin
21                next_state = CACHE_RD;
22            end else begin
23                next_state = TAG_RD;
24            end
25        end
26        CACHE_WR: begin
27            next_state = WAIT;
28        end
29        CACHE_RD: begin
30            next_state = CACHE_RESP;
31        end
32        CACHE_RESP: begin
33            if (from_cpu_cache_rsp_ready) begin
34                next_state = WAIT;
35            end else begin
36                next_state = CACHE_RESP;
37            end
38        end
39        REFILL: begin
40            if (Mem_req_current) begin
41                next_state = CACHE_WR;
42            end else if (!Mem_req_current) begin
43                next_state = CACHE_RD;
44            end else begin
45                next_state = REFILL;
46            end
47        end
48    end
49 end
```

图 11: dcache 状态转移图的代码设计 1

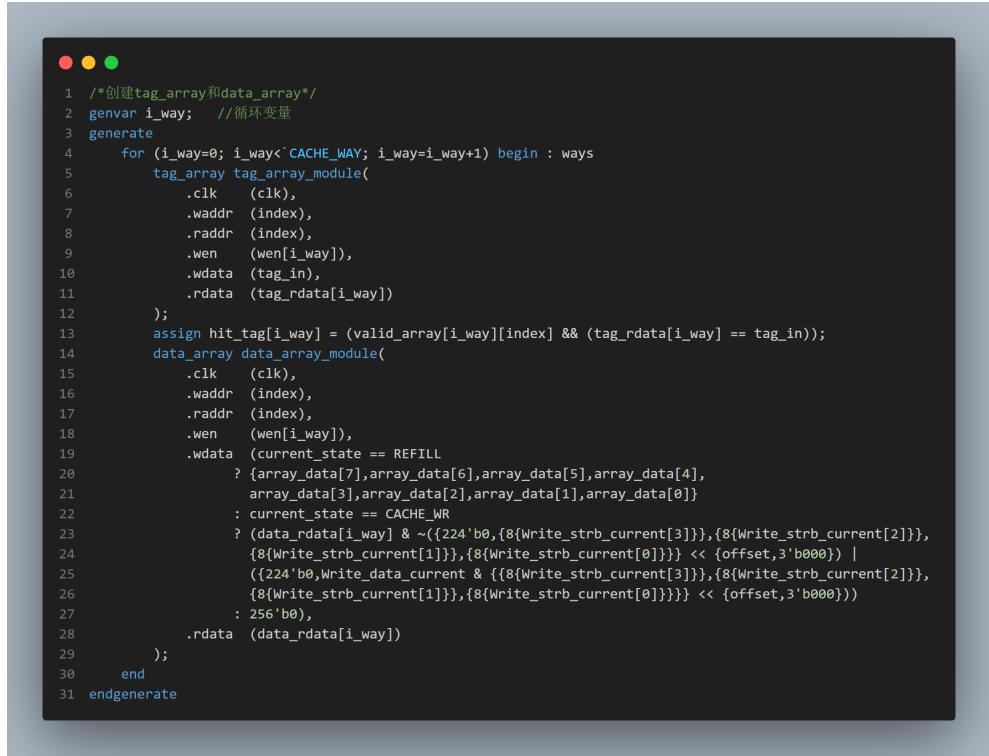
```

1      EVICT: begin                                //判断是否为脏数据
2          if (Cache_dirty) begin
3              next_state = MEM_WR;
4          end else if (!Cache_dirty) begin
5              next_state = MEM_RD;
6          end else begin
7              next_state = EVICT;
8          end
9      end
10     MEM_RD: begin                               //允许接收内存读请求
11         if (from_mem_rd_req_ready) begin
12             next_state = RECV_MEM1;
13         end else begin
14             next_state = MEM_RD;
15         end
16     end
17     MEM_WR: begin                               //允许接收内存写请求
18         if (from_mem_wr_req_ready) begin
19             next_state = CACHE_WB;
20         end else begin
21             next_state = MEM_WR;
22         end
23     end
24     CACHE_WB: begin                            //允许内存接收数据 (32byte突发传输)
25         if (from_mem_wr_data_ready && to_mem_wr_data_last) begin
26             next_state = MEM_RD;
27         end else begin
28             next_state = CACHE_WB;
29         end
30     end
31     RECV_MEM1: begin                           //允许从内存将数据读入Cache (突发传输)
32         if (from_mem_rd_rsp_valid && from_mem_rd_rsp_last) begin
33             next_state = REFILL;
34         end else begin
35             next_state = RECV_MEM1;
36         end
37     end
38     BYPASS_RD: begin                            //允许接收内存读请求
39         if (from_mem_rd_req_ready) begin
40             next_state = RECV_MEM2;
41         end else begin
42             next_state = BYPASS_RD;
43         end
44     end
45     BYPASS_WR: begin                            //允许接收内存写请求
46         if (from_mem_wr_req_ready) begin
47             next_state = CPU_WB;
48         end else begin
49             next_state = BYPASS_WR;
50         end
51     end
52     RECV_MEM2: begin                            //允许从内存读出数据 (突发传输)
53         if (from_mem_rd_rsp_valid && from_mem_rd_rsp_last) begin
54             next_state = RECV_CPU;
55         end else begin
56             next_state = RECV_MEM2;
57         end
58     end
59     RECV_CPU: begin                            //CPU允许接收数据
60         if (from_cpu_cache_rsp_ready) begin
61             next_state = WAIT;
62         end else begin
63             next_state = RECV_CPU;
64         end
65     end
66     CPU_WB: begin                             //允许内存接收数据 (突发传输)
67         if (from_mem_wr_data_ready && to_mem_wr_data_last) begin
68             next_state = WAIT;
69         end else begin
70             next_state = CPU_WB;
71         end
72     end
73     default: next_state = WAIT;
74 endcase
75 end

```

图 12: dcache 状态转移图的代码设计 2

然后是我们需要将 dcache 和对应的 tag\_array 和 data\_array 进行连接:



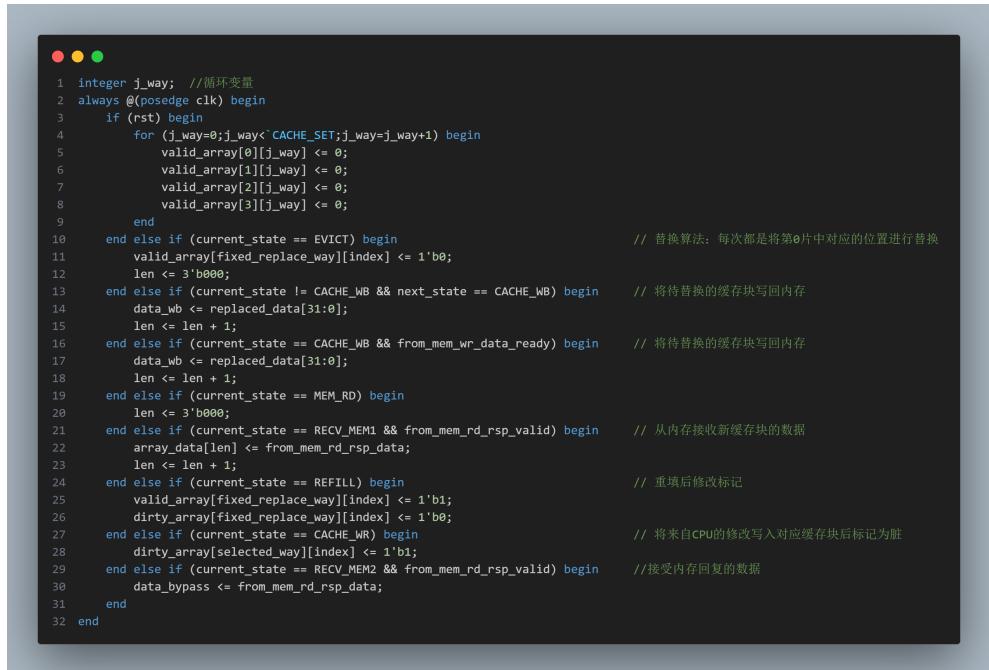
```

1  /*创建tag_array和data_array*/
2  genvar i_way; //循环变量
3  generate
4      for (i_way=0; i_way<CACHE_WAY; i_way=i_way+1) begin : ways
5          tag_array tag_array_module(
6              .clk    (clk),
7              .waddr (index),
8              .raddr (index),
9              .wen   (wen[i_way]),
10             .wdata  (tag_in),
11             .rdata  (tag_rdata[i_way])
12         );
13         assign hit_tag[i_way] = (valid_array[i_way][index] && (tag_rdata[i_way] == tag_in));
14         data_array data_array_module(
15             .clk    (clk),
16             .waddr (index),
17             .raddr (index),
18             .wen   (wen[i_way]),
19             .wdata  (current_state == REFILL
20                 ? {array_data[7],array_data[6],array_data[5],array_data[4],
21                   array_data[3],array_data[2],array_data[1],array_data[0]}
22                 : current_state == CACHE_WR
23                 ? {data_rdata[i_way],{8{Write_stb_current[3]}},{8{Write_stb_current[2]}},
24                   {8{Write_stb_current[1]}},{8{Write_stb_current[0]}} <> {offset,3'b000} |
25                   ({224'b0,Write_data_current & {{8{Write_stb_current[3]}},{8{Write_stb_current[2]}}},
26                     {8{Write_stb_current[1]}},{8{Write_stb_current[0]}}}) <> {offset,3'b000})
27                 : 256'b0},
28             .rdata  (data_rdata[i_way])
29         );
30     end
31 endgenerate

```

图 13: 对应接口代码设计

接口设计同样是使用 generate 语块来完成。然后关于替换算法,我仍然使用的是固定替换策略:



```

1  integer j_way; //循环变量
2  always @(posedge clk) begin
3      if (rst) begin
4          for (j_way=0;j_way<CACHE_SET;j_way=j_way+1) begin
5              valid_array[0][j_way] <= 0;
6              valid_array[1][j_way] <= 0;
7              valid_array[2][j_way] <= 0;
8              valid_array[3][j_way] <= 0;
9          end
10     end else if (current_state == EVICT) begin // 替换算法: 每次都是将第0片中对应的位置进行替换
11         valid_array[fixed_replace_way][index] <= 1'b0;
12         len <= 3'b000;
13     end else if (current_state != CACHE_WB && next_state == CACHE_WB) begin // 将待替换的缓存块写回内存
14         data_wb <= replaced_data[31:0];
15         len <= len + 1;
16     end else if (current_state == CACHE_WB && from_mem_wr_data_ready) begin // 将待替换的缓存块写回内存
17         data_wb <= replaced_data[31:0];
18         len <= len + 1;
19     end else if (current_state == MEM_RD) begin
20         len <= 3'b000;
21     end else if (current_state == RECV_MEM1 && from_mem_rd_rsp_valid) begin // 从内存接收新缓存块的数据
22         array_data[len] <= from_mem_rd_rsp_data;
23         len <= len + 1;
24     end else if (current_state == REFILL) begin // 重填后修改标记
25         valid_array[fixed_replace_way][index] <= 1'b1;
26         dirty_array[fixed_replace_way][index] <= 1'b0;
27     end else if (current_state == CACHE_WR) begin // 将来自CPU的修改写入对应缓存块后标记为脏
28         dirty_array[selected_way][index] <= 1'b1;
29     end else if (current_state == RECV_MEM2 && from_mem_rd_rsp_valid) begin // 接受内存回复的数据
30         data_bypass <= from_mem_rd_rsp_data;
31     end
32 end

```

图 14: 替换策略代码设计

然后是对应 dcache\_module 的输出赋值,这里需要注意的是要分是否使用 dcache 两种情况来进行设计:

```

1 //输出赋值
2 assign to_cpu_mem_req_ready = (current_state == WAIT);
3 assign to_cpu_cache_rsp_valid = (current_state == CACHE_RESP || current_state == RECV_CPU);
4 assign to_cpu_cache_rsp_data = {32{current_state == CACHE_RESP}} & (selected_data[31:0])
| {32{current_state == RECV_CPU}} & data_bypass;
5 assign to_mem_rd_req_valid = current_state == MEM_RD || current_state == BYPASS_RD;
6 assign to_mem_rd_req_addr = {32{current_state == MEM_RD}} & {tag_in, index, 5'b00000}
| {32{current_state == BYPASS_RD}} & Bypass_address_current;
7 assign to_mem_rd_req_len = {8{current_state == MEM_RD}} & 8'b111
| {8{current_state == BYPASS_RD}} & 8'be;
8 assign to_mem_rd_rsp_ready = (current_state == RECV_MEM1) || (current_state == RECV_MEM2) || rst;
9 assign to_mem_rd_req_addr = {32{current_state == MEM_WR}} & {tag_rdata[fixed_replace_way], index, 5'b00000}
| {32{current_state == BYPASS_WR}} & Bypass_address_current;
10 assign to_mem_rd_req_len = {8{current_state == MEM_WR}} & 8'b111
| {8{current_state == BYPASS_WR}} & 8'be;
11 assign to_mem_wr_data_valid = (current_state == CACHE_WB) || (current_state == CPU_WB);
12 assign to_mem_wr_data = {32{current_state == CACHE_WB}} & data_wb
| {32{current_state == CPU_WB}} & Write_data_current;
13 assign to_mem_wr_data_strb = {4{current_state == CACHE_WB}}
| {4{current_state == CPU_WB}} & Write_strb_current;
14 assign to_mem_wr_data_last = (current_state == CACHE_WB && len == 3'b000)
|| current_state == CPU_WB;
15 assign to_mem_wr_req_len = 23

```

图 15: 替换策略代码设计

关于其他变量的赋值和 icache 类似:

```

1 // 地址解析
2 assign index = Bypass_address_current[7:5]; // 地址的组索引部分
3 assign offset = Bypass_address_current[4:0]; // 地址的块内偏移部分
4 assign tag_in = Bypass_address_current[31:8]; // 地址的标签部分
5
6 // 判定是否可以访问Cache、是否读命中、是否脏
7 assign Cacheable = (!from_cpu_mem_req_addr[31:5]) && ~from_cpu_mem_req_addr[31] && ~from_cpu_mem_req_addr[30];
8 assign Cache_hit = hit_tag[0] | hit_tag[1] | hit_tag[2] | hit_tag[3];
9 assign Cache_miss = ~Cache_hit;
10 assign Cache_dirty = valid_array[fixed_replace_way][index] && dirty_array[fixed_replace_way][index];
11
12 // 选中的片序号和数据
13 assign selected_way = {2{hit_tag[0]}} & 2'b00 | {2{hit_tag[1]}} & 2'b01 |
| {2{hit_tag[2]}} & 2'b10 | {2{hit_tag[3]}} & 2'b11 ;
14 assign selected_data = data_rdata[selected_way] >> {offset[4:0], 3'b000};
15 assign replaced_data = data_rdata[fixed_replace_way] >> {len, 5'b00000};
16
17 //使能判断
18 //由于分为从旁路读写数据和从Cache读写数据，因此使能与Cache的片数有关
19 assign wen[0] = current_state == REFILL || current_state == CACHE_WR && selected_way == 2'b00;
20 assign wen[1] = current_state == REFILL || current_state == CACHE_WR && selected_way == 2'b01;
21 assign wen[2] = current_state == REFILL || current_state == CACHE_WR && selected_way == 2'b10;
22 assign wen[3] = current_state == REFILL || current_state == CACHE_WR && selected_way == 2'b11;
23

```

图 16: 替换策略代码设计

然后 dcache 与 icache 不同,还有额外需要注意的地方是:CPU 在成功发送请求后并不会等待 cache 完成一系列之后的操作,而是会继续按照 cpu 的设计来进行接下来的操作,就和多周期 cpu 的编写一样,我们需要将在取指令的时候的指令信息存下来,直到再次取指令的时候再进行更新:

```

1 //请求数据缓冲
2 always @(posedge clk) begin
3     if (current_state == WAIT) begin
4         Bypass_address_current <= from_cpu_mem_req_addr;
5         Write_data_current    <= from_cpu_mem_req_wdata;
6         Write_strb_current   <= from_cpu_mem_req_wstrb;
7         Mem_req_current      <= from_cpu_mem_req;
8     end
9 end

```

图 17: dcache 数据缓冲

## 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)

- icache 设计问题。

在 icache 的设计过程中,由于以及给出状态转移图,所以该部分的编写大体上思路是没有难度的,主要存在的问题是正确的将 cache 和 cpu 以及内存连接起来,在仿真的过程中,我遇到了多数测试样例可以通过,但是有几个样例总是不能通过的情况,在观察报错之后,我发现是我位宽没有对齐。

```

73 %Warning-WIDTH: fpga/design/ucas-cod/hardware/sim/..../sources/custom_cpu/cache/icache_top.v:197:15: Bit extraction of array
74 [7:0] requires 3 bit index, not 1 bits.
75           : ... In instance custom_
76           cpu_test.u_cpu_test.u_icache_wrapper.u_icache
77           197 | valid_array[from_mem_rd_rsp_last][index]<=1'b0;
78           | ^
79 %Warning-WIDTH: fpga/design/ucas-cod/hardware/sim/..../sources/custom_cpu/cache/icache_top.v:197:37: Bit extraction of var
80 [3:0] requires 2 bit index, not 3 bits.
81           : ... In instance custom_
82           cpu_test.u_cpu_test.u_icache_wrapper.u_icache
83           197 | valid_array[from_mem_rd_rsp_last][index]<=1'b0;
84           | ^
85 %Warning-WIDTH: fpga/design/ucas-cod/hardware/sim/..../sources/custom_cpu/cache/icache_top.v:204:15: Bit extraction of array
86 [7:0] requires 3 bit index, not 1 bits.
87           : ... In instance custom_
88           cpu_test.u_cpu_test.u_icache_wrapper.u_icache
89           204 | valid_array[from_mem_rd_rsp_last][index] <= 1'b1;
90           | ^
91 %Warning-WIDTH: fpga/design/ucas-cod/hardware/sim/..../sources/custom_cpu/riscv32/custom_cpu.v:225:21: Logical operator LOGA
92 ND expects 1 bit on the LHS, but LHS's VARREF 'ALU_result' generates 32 bits.
93           : ... In instance custo
94           m_cpu_test.u_cpu_test.u_cpu
95           225 | (ALU_result && (funct3 == 3'b110 || funct3 == 3'b100)) ||
96           | ^
97 %Warning-WIDTH: fpga/design/ucas-cod/hardware/sim/..../sources/custom_cpu/riscv32/custom_cpu.v:226:7: Logical operator LOGNO
98 T expects 1 bit on the LHS, but LHS's VARREF 'ALU_result' generates 32 bits.
99           : ... In instance custom_
100          _cpu_test.u_cpu_test.u_cpu
101          226 | (!ALU_result && (funct3 == 3'b101 || funct3 == 3'b111))) ? 1 : 0;
102          | ^
103 %Error: Exiting due to 8 warning(s)

```

图 18: icache 行为仿真中的报错

然后我在修改完对应的接口,解决了位宽不对的问题之后,可以顺利通过 icache 的仿真测试。

- dcache 设计问题。

在编写 cache 的过程中, 老师上课讲到了给出的文件中可能存在的一些问题, 是由于本地仿真和在线仿真行为逻辑不同导致的对表达式优先级处理的不同, 解决办法就是在可能引起错误的地方加上括号保证该操作优先执行即可:

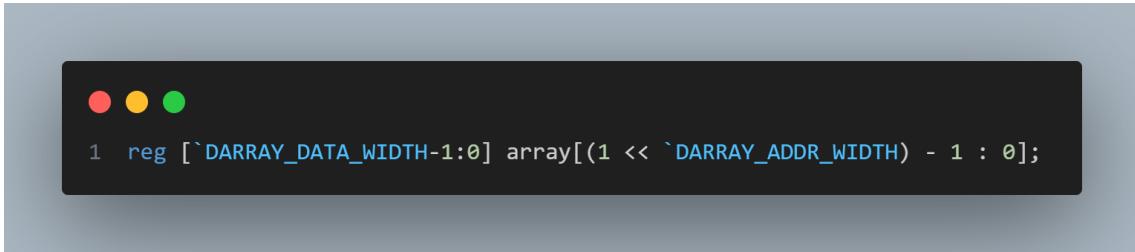


图 19: data\_array.v 中对问题的修改

同样的在 tag\_array.v 中也做相应的修改即可。

此外, 在 dcache 的仿真过程中, 还遇到了超时的情况:

```
45 time 10641.63ms
46 custom cpu running time out
47 wait_for_finish: AXI firewall status: 00000000
```

图 20: fpga 仿真中 hello 仿真超时

然后我检查发现, 是由于状态转移图的代码编写有问题, 导致某些情况下不能正确跳转到应该跳转的状态, 经过修改之后可以正确完成 fpga 仿真。

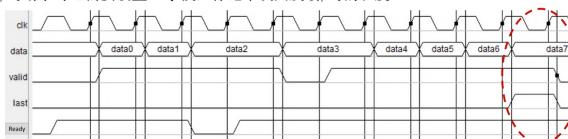
我还遇到了在课程群中有同学询问关于行为仿真中只有 basic 仿真出现问题, 其余仿真均可通过的情况, 经老师和助教说明后得出是由于本实验的仿真是和流水线仿真相同, 由于不同同学流水线设计不同, 没办法按照时序进行仿真, 只能等待完成某一条指令后再进行对照, 然后我编写的代码可能存在多完成一条 nop 指令的现象, 导致和仿真测试结果不同, 从而出现问题, 因此只需要保证其他行为仿真和 fpga 仿真通过即可。

### 三、对讲义中思考题的理解和回答

- 思考题 1: 如何实现大于 4 byte 的指令/数据交互。

本实验中我们采用突发传输的方式来完成大于 4byte 的指令和数据交互:

- 发起一次内存读/写请求后, 连续传输多个数据
- 基于之前实验中使用过的Valid/Ready握手信号, 实现突发传输接口
  - 每个数据的传输都需要Valid-Ready握手
  - 添加last信号, 标志本次突发传输最后一个有效数据
  - 在发送读/写请求时还需要设置len字段, 标记本次突发读/写的长度



对于32-bit位宽的内存访问接口, 一次Cache Block数据交互就是突发读/写8个数据 (一般使用最后一个数据的下标值来标记len)

图 21: 突发传输

用 last 信号标记突发传输最后一个有效数据,用 len 字段标记本次突发读/写的长度。

- 思考题 2: 如果用上述接口只传输 32-bit 数据( $len = 0$ ), last 在哪个位置拉高。

如果只传输 4byte 数据,则在第 0 个数据传输时拉高 last。

## 四、实验所耗时间

在课后,你花费了大约 20 小时完成此次实验。

## 五、实验思考与心得

- 性能分析。

RISCV	15pz	bf	dinic	fib	md5	qsort	queen	sieve	ssort
时钟周期数	522802366	38811687	1479194	181088157	384627	786885	6885618	744437	44725243
指令数	5224458	452831	16668	2549502	4892	9457	81467	10172	619022
RISCV-icache	15pz	bf	dinic	fib	md5	qsort	queen	sieve	ssort
时钟周期数	187067086	10676372	460733	18403232	75415	183847	1771798	94544	5359734
指令数	5224458	452831	16668	2549502	4892	9457	81467	10172	619022
RISCV-icache&dcache	15pz	bf	dinic	fib	md5	qsort	queen	sieve	ssort
时钟周期数	69895502	7298098	353759	18110066	62891	84754	641931	73130	4778518
指令数	5224458	452831	16668	2549502	4892	9457	81467	10172	619022
无cache对比icache	1.79	2.64	2.21	8.84	4.10	3.28	2.89	6.87	7.34
无cache对比icache&dcache	6.48	4.32	3.18	9.00	5.12	8.28	9.73	9.18	8.36
icache对比icache&dcache	1.68	0.46	0.30	0.02	0.20	1.17	1.76	0.29	0.12
	dhystone	coremark							
RISCV-icache	3126	8789157							
RISCV-icache&dcache	4046	8632367							

图 22: 性能对比

从 coremark 和 dhystone 的分数来看, icache 对性能的提升较 dcache 来说更大, 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, 因此只有在使用较多访存指令的测试中增加 dcache 的优势才更加明显, 比如图中的 15pz, qsort 和 queen。

- 实验心得。

本实验是选做实验中较难的一个,也是工作量比较大的一个。不过本实验循序渐进,比较适合我们进行选做。

icache 部分比较容易,因为讲义中已经给出了状态转移图,只需要我们按照图中信息完成即可,剩下的就是对于命中的判断和接口的连接。

dcache 部分难点在于需要自己设计状态转移图,这个部分每个人的设计都可能有不同,不同的设计也会对性能产生相应的影响,需要我们仔细思考,先画出状态转移图再进行代码的编写。

同时希望以后能够实现打印出 cache 命中率的功能,这样有利于对比不同设计的性能,从而更好地帮助我们实现性能更高的 cache。