中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929016　　姓名：李金明　专业：计算机科学与技术

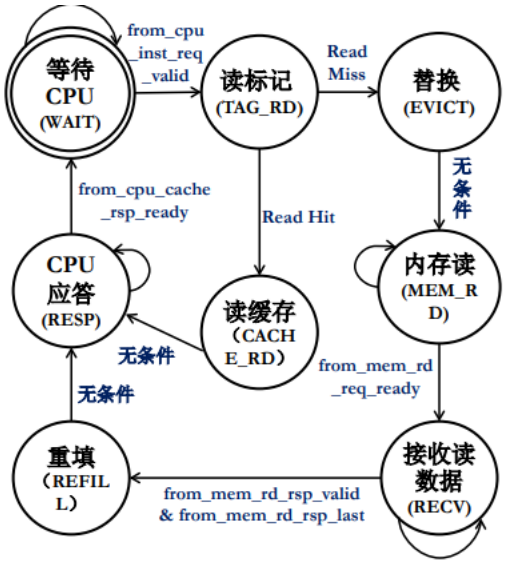
实验序号：5.4　　实验名称：高速缓存（Cache）设计

一、 关键代码及说明

1. 指令 Cache 部分

a. 状态机的设计

指令 Cache 的状态机由以下的状态转移图给出：



状态机的第二段由以下代码给出：

```
96    always @ (*) begin
97        case (current_state)
98            `S_WAIT: begin
99                if (from_cpu_inst_req_valid) next_state = `S_TAG_RD;
100               else next_state = `S_WAIT;
101           end
102           `S_TAG_RD: begin
103               if (read_hit) next_state = `S_CACHE_RD;
104               else next_state = `S_EVICT;
105           end
106           `S_EVICT: next_state = `S_MEM_RD;
107           `S_MEM_RD: begin
108               if (from_mem_rd_req_ready) next_state = `S_RECV;
109               else next_state = `S_MEM_RD;
110           end
111           `S_RECV: begin
112               if (from_mem_rd_rsp_valid & from_mem_rd_rsp_last) next_state = `S_REFILL;
113               else next_state = `S_RECV;
114           end
115           `S_REFILL: next_state = `S_RESP;
116           `S_CACHE_RD: next_state = `S_RESP;
117           `S_RESP: begin
118               if (from_cpu_cache_rsp_ready) next_state = `S_WAIT;
119               else next_state = `S_RESP;
120           end
121           default: next_state = `S_WAIT;
122       endcase
123   end
```

细节方面将在以下内容中给出。

b. 指令 Cache 内部存储结构

本实验中指令 Cache 使用的是 4 路组相联结构，字长 32bit，一个子块大小为 8 字，共有 32 块。

指令 Cache 一共有 4 路，每一路由 8 个块组成，每一块由标记该 set 对应的 cache 块是否存储了有效的内容的 valid、存放 cache 块对应地址的 tag 域的 tag 以及 256bit 数据组成，每个 way 内部各个 set 的对应字段组成一个 Array。在调用 tag array 和 data array 时，使用例化 4 个模块的方式：

```
    tag_array tag_0 (
        .clk(clk),
        .waddr(index),
        .raddr(index),
        .wen(wen[0]),
        .wdata(tag),
        .rdata(tag_rdata[0])
    );
    tag_array tag_1 (
        .clk(clk),
        .waddr(index),
        .raddr(index),
        .wen(wen[1]),
        .wdata(tag),
        .rdata(tag_rdata[1])
    );
    tag_array tag_2 (
        .clk(clk),
        .waddr(index),
        .raddr(index),
        .wen(wen[2]),
        .wdata(tag),
        .rdata(tag_rdata[2])
    );
    tag_array tag_3 (
        .clk(clk),
        .waddr(index),
        .raddr(index),
        .wen(wen[3]),
        .wdata(tag),
        .rdata(tag_rdata[3])
    );
```

```
    data_array data_0 (
        .clk(clk),
        .waddr(index),
        .raddr(index),
        .wen(wen[0]),
        .wdata(data_wdata),
        .rdata(data_rdata[0])
    );
    data_array data_1 (
        .clk(clk),
        .waddr(index),
        .raddr(index),
        .wen(wen[1]),
        .wdata(data_wdata),
        .rdata(data_rdata[1])
    );
    data_array data_2 (
        .clk(clk),
        .waddr(index),
        .raddr(index),
        .wen(wen[2]),
        .wdata(data_wdata),
        .rdata(data_rdata[2])
    );
    data_array data_3 (
        .clk(clk),
        .waddr(index),
        .raddr(index),
        .wen(wen[3]),
        .wdata(data_wdata),
        .rdata(data_rdata[3])
    );
```

valid array 则用一个 4 元素 8 位数组表示：

```
    reg [`CACHE_SET - 1 : 0] valid_array [`CACHE_WAY - 1 : 0];
```

初始时所有 valid 值置 0，在 EVICT 阶段根据替换算法将对应位置的 valid 置 0，在 REFILL 阶段将填入有效数据位置的对应 valid 置 1：

```
    integer i_valid;
    always @ (posedge clk) begin
        if (rst) begin
            for (i_valid = 0; i_valid < `CACHE_WAY; i_valid = i_valid + 1)
                valid_array[i_valid] <= 8'b0;
            end
        else if (state_EVICT) valid_array[replace_position][index] <= 1'b0;
        else if (state_REFILL) valid_array[replace_position][index] <= 1'b1;
    end
```

c. WAIT 阶段

该阶段表示 Cache 处于待机状态，可以接受 cpu 发出的指令请求信号，只需拉高 to_cpu_inst_req_ready 信号即可。

```
125     assign to_cpu_inst_req_ready = state_WAIT;
```

d. TAG_RD 阶段

此阶段根据输入地址的 index 域，读出 4 路的 valid 和 tag，与输入地址的 tag 相比较，从而确定是否命中。

这里我采用独热码的方式确定命中的是哪一路，由以下代码给出：

```
183     assign hit_way[0] = valid_array[0][index] & (tag_rdata[0] == tag);
184     assign hit_way[1] = valid_array[1][index] & (tag_rdata[1] == tag);
185     assign hit_way[2] = valid_array[2][index] & (tag_rdata[2] == tag);
186     assign hit_way[3] = valid_array[3][index] & (tag_rdata[3] == tag);
```

读命中信号也在此时产生（为 1 表示命中，反之则为不命中）：

```
150     assign read_hit = |hit_way;
```

e. CACHE_RD 阶段

这是 Cache 命中的情况，此阶段根据 TAG_RD 阶段得到的命中路数读出 32 byte 数据，并根据输入地址的 offset 域，来确定要返回的指令码，由以下代码给出：

```
220     assign data_hit = {`LINE_LEN{hit_way[0]}} & data_rdata[0] |
221                       {`LINE_LEN{hit_way[1]}} & data_rdata[1] |
222                       {`LINE_LEN{hit_way[2]}} & data_rdata[2] |
223                       {`LINE_LEN{hit_way[3]}} & data_rdata[3];
224     assign to_cpu_cache_rsp_data = data_hit >> {offset, 3'b0};
```

其中 data_hit 是读出的 32 byte 数据。

f. RESP 阶段

此阶段拉高 to_cpu_cache_rsp_valid 端口信号即可将返回的指令码传递给 CPU：

```
126     assign to_cpu_cache_rsp_valid = state_RESP;
```

g. EVICT 阶段

这是 Cache 未命中的情况。此阶段首先要根据替换算法选择出替换的 Cache 块。这里我使用的是 LRU 算法，其基本思路是对每个 Cache 块设置一个计数器，其计数 Cache 块未访问到的次数。每次重填数据后将对应的 Cache 块的计数器置 0。当访问到一个 Cache 块时，将相同 index 值的另外三个 way 的 Cache 块的计数器加 1。每次替换时，选择计数器值最大的（即重填后被访问次数最少的）Cache 块进行替换。由以下代码给出：

计数器的设置：

```
226     integer i_call;
227     always @ (posedge clk) begin
228         if (rst) begin
229             for (i_call = 0; i_call < `CACHE_SET; i_call = i_call + 1) begin
230                 call_cnt[0][i_call] <= 8'b0;
231                 call_cnt[1][i_call] <= 8'b0;
232                 call_cnt[2][i_call] <= 8'b0;
233                 call_cnt[3][i_call] <= 8'b0;
234             end
235         end
236         else if (state_RESP && from_cpu_cache_rsp_ready)begin
237             if (~hit_way[0]) call_cnt[0][index] <= call_cnt[0][index] + 1;
238             if (~hit_way[1]) call_cnt[1][index] <= call_cnt[1][index] + 1;
239             if (~hit_way[2]) call_cnt[2][index] <= call_cnt[2][index] + 1;
240             if (~hit_way[3]) call_cnt[3][index] <= call_cnt[3][index] + 1;
241         end
242         else if (state_REFILL) call_cnt[replace_position][index] <= 8'b0;
243     end
```

根据计数器的值选择替换的路：

```
245     assign replace_position = (~valid_array[0][index])? 2'd0 :
246                               (~valid_array[1][index])? 2'd1 :
247                               (~valid_array[2][index])? 2'd2 :
248                               (~valid_array[3][index])? 2'd3 :
249                               (call_cnt[0][index] >= call_cnt[1][index])?
250                               (call_cnt[0][index] >= call_cnt[2][index])?
251                               (call_cnt[0][index] >= call_cnt[3][index])? 2'd0 : 2'd3 :
252                               (call_cnt[2][index] >= call_cnt[3][index])? 2'd2 : 2'd3 :
253                               (call_cnt[1][index] >= call_cnt[2][index])?
254                               (call_cnt[1][index] >= call_cnt[3][index])? 2'd1 : 2'd3 :
255                               (call_cnt[2][index] >= call_cnt[3][index])? 2'd2 : 2'd3;
```

在此阶段还需将 valid 值置 0，已在前面的代码给出。

h. MEM_RD 阶段

这个阶段是发送访存请求的阶段，需要保证发送到内存的地址有效。由于Cache 块是 32 byte，故这个地址是按 32 位对齐的。该阶段还需拉高to_mem_rd_req_valid 值，由以下代码给出：

```
131     assign to_mem_rd_req_addr = {from_cpu_inst_req_addr[31:5], 5'b0};
127     assign to_mem_rd_req_valid = state_MEM_RD;
```

i. RECV 阶段

此阶段进行数据的读取，一共需要读取 32 byte 数据，每次传输 32 bit，按照突发（Burst）传输的方式进行。首先要拉高 to_mem_rd_rsp_ready。

```
128     assign to_mem_rd_rsp_ready = state_RECV;
```

每当 from_mem_rd_rsp_valid 拉高时，准备接受数据。我使用了 8 元素的 32 bit 数组记录读出的 32 byte 数据，使用计数器确定此次读入的是第几个数据：

```
257     always @ (posedge clk) begin
258         if (rst || state_MEM_RD && from_mem_rd_req_ready) read_cnt <= 2'b0;
259         else if (state_RECV && from_mem_rd_rsp_valid) read_cnt <= read_cnt + 1'b1;
260     end
261
262     always @ (posedge clk) begin
263         if (state_RECV && from_mem_rd_rsp_valid) read_data[read_cnt] <= from_mem_rd_rsp_data;
264     end
```

j. REFILL 阶段

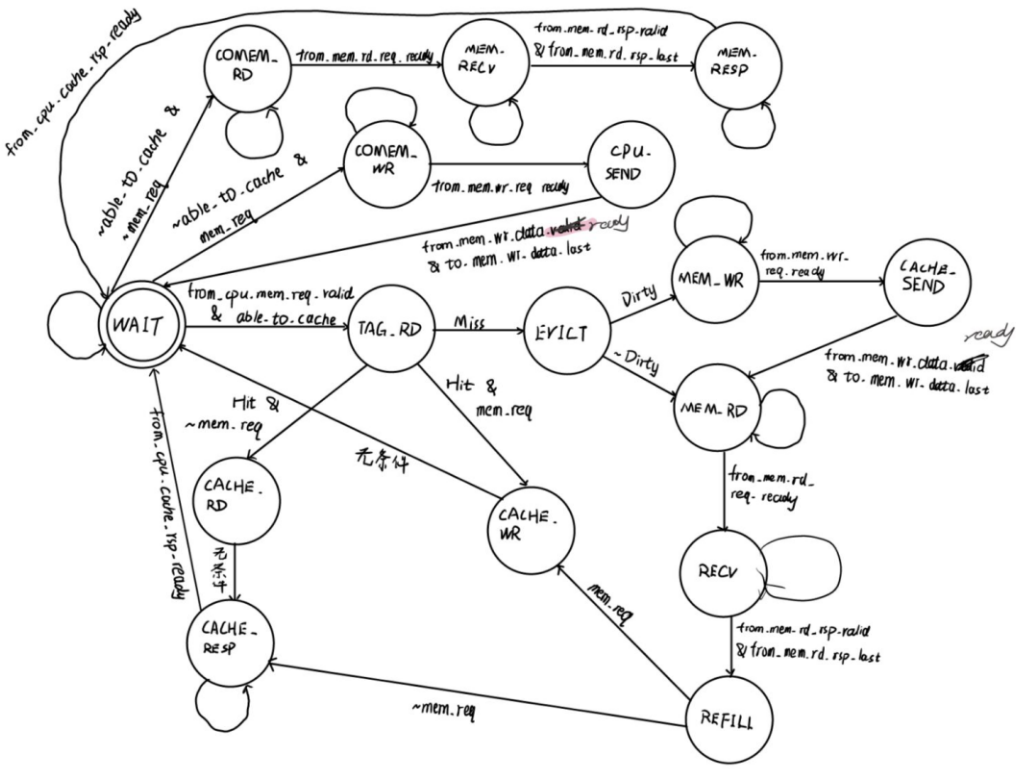
Cache 在 REFILL 阶段将读到的 32 byte 数据重新填回 cache 块，由以下代码给出：

```
143     assign wen[0] = state_REFILL & (~replace_position[1] & ~replace_position[0]);
144     assign wen[1] = state_REFILL & (~replace_position[1] & replace_position[0]);
145     assign wen[2] = state_REFILL & (replace_position[1] & ~replace_position[0]);
146     assign wen[3] = state_REFILL & (replace_position[1] & replace_position[0]);
147     assign data_wdata = {read_data[7], read_data[6], read_data[5], read_data[4],
148                          read_data[3], read_data[2], read_data[1], read_data[0]};
```

其中写使能是根据替换算法拉高的。

2. dcache 部分

a. 状态机的设计



状态机的第二段由以下代码给出：

```
164     always @ (*) begin
165         case (current_state)
166             `S_WAIT: begin
167                 if (!from_cpu_mem_req_valid) next_state = `S_WAIT;
168                 else if (able_to_cache) next_state = `S_TAG_RD;
169                 else if (from_cpu_mem_req) next_state = `S_COMEM_WR;
170                 else next_state = `S_COMEM_RD;
171             end
172             `S_TAG_RD: begin
173                 if (~read_hit) next_state = `S_EVICT;
174                 else if (~mem_req) next_state = `S_CACHE_RD;
175                 else next_state = `S_CACHE_WR;
176             end
177             `S_CACHE_RD: next_state = `S_CACHE_RESP;
178             `S_CACHE_RESP: next_state = `S_WAIT;
179             `S_CACHE_WR: next_state = `S_WAIT;
180             `S_EVICT: begin
181                 if (dirty) next_state = `S_MEM_WR;
182                 else next_state = `S_MEM_RD;
183             end
184             `S_MEM_WR: begin
185                 if (from_mem_wr_req_ready) next_state = `S_CACHE_SEND;
186                 else next_state = `S_MEM_WR;
187             end
188             `S_CACHE_SEND: begin
189                 if (from_mem_wr_data_ready & to_mem_wr_data_last) next_state = `S_MEM_RD;
190                 else next_state = `S_CACHE_SEND;
191             end
192             `S_MEM_RD: begin
193                 if (from_mem_rd_req_ready) next_state = `S_RECV;
194                 else next_state = `S_MEM_RD;
195             end
```

```
196             `S_RECV: begin
197                 if (from_mem_rd_rsp_valid & from_mem_rd_rsp_last) next_state = `S_REFILL;
198                 else next_state = `S_RECV;
199             end
200             `S_REFILL: begin
201                 if (mem_req) next_state = `S_CACHE_WR;
202                 else next_state = `S_CACHE_RESP;
203             end
204             `S_COMEM_WR: begin
205                 if (from_mem_wr_req_ready) next_state = `S_CPU_SEND;
206                 else next_state = `S_COMEM_WR;
207             end
208             `S_CPU_SEND: begin
209                 if (from_mem_wr_data_ready & to_mem_wr_data_last) next_state = `S_WAIT;
210                 else next_state = `S_CPU_SEND;
211             end
212             `S_COMEM_RD: begin
213                 if (from_mem_rd_req_ready) next_state = `S_MEM_RECV;
214                 else next_state = `S_COMEM_RD;
215             end
216             `S_MEM_RECV: begin
217                 if (from_mem_rd_rsp_valid & from_mem_rd_rsp_last) next_state = `S_MEM_RESP;
218                 else next_state = `S_MEM_RECV;
219             end
220             `S_MEM_RESP: begin
221                 if (from_cpu_cache_rsp_ready) next_state = `S_WAIT;
222                 else next_state = `S_MEM_RESP;
223             end
224             default: next_state = `S_WAIT;
225         endcase
226     end
```

数据 Cache 相比指令 Cache 增加的主要是对内存的写回和不可缓存区域的旁路处理，其他地方与指令 Cache 基本相同，且其中比如请求状态、接收内存数据大同小异，故不再按照每一个状态细化解释工作过程。

b. 内存写操作

对于 Cache 写回操作，首先需要设置一个 dirty array，其表示 Cache 中的数据是否更改过，由此确定在替换时是否需要向主存写入数据。Dirty array 的定义与 valid array 相同，赋值由以下代码给出

```
238     integer i_dirty;
239     always @ (posedge clk) begin
240         if (rst) begin
241             for (i_dirty = 0; i_dirty <`CACHE_WAY; i_dirty = i_dirty + 1)
242                 dirty_array[i_dirty] <= 8'b0;
243         end
244         else if (state_CACHE_WR)
245             dirty_array[write_way][index] <= 1'b1;
246         else if (state_CACHE_SEND & from_mem_wr_data_ready & to_mem_wr_data_last)
247             dirty_array[replace_position][index] <= 1'b0;
248     end
```

当未命中且替换的 Cache 块是“脏块”时，或者不可缓存区域要写回时，需要进行内存写回操作。写回时各项数据赋值如下：

```
356     always @ (posedge clk) begin
357         if (state_MEM_WR & from_mem_wr_req_ready)
358             {cache_to_mem_data[7], cache_to_mem_data[6], cache_to_mem_data[5], cache_to_mem_data[4],
359             cache_to_mem_data[3], cache_to_mem_data[2], cache_to_mem_data[1], cache_to_mem_data[0]} <= data_rdata[replace_position];
360     end
361
362     assign to_mem_wr_data = {32{state_CACHE_SEND}} & cache_to_mem_data[operate_cnt] |
363                             {32{state_CPU_SEND}} & cpu_data;
364     assign to_mem_wr_data_last = (state_CPU_SEND) ? (operate_cnt == 3'b0)? 1 : 0 :
365                                  (operate_cnt == 3'd7)? 1 : 0;
366     assign to_mem_wr_req_addr = {32{state_MEM_WR}} & {tag_rdata[replace_position], index, 5'b0} |
367                                 {32{state_COMEM_WR}} & cpu_address;
368     assign to_mem_wr_data_strb = {4{state_MEM_WR | state_CACHE_SEND}} & 4'b1111 |
369                                  {4{state_COMEM_WR | state_CPU_SEND}} & cpu_strb;
370     assign to_mem_wr_req_len = {8{state_MEM_WR}} & 8'b111 |
371                                {8{state_COMEM_WR}} & 8'b0;
```

```
396     always @ (posedge clk) begin
397         if (rst || state_MEM_WR && from_mem_wr_req_ready || state_MEM_RD && from_mem_rd_req_ready)
398             operate_cnt <= 3'b0;
399         else if (state_CACHE_SEND && from_mem_wr_data_ready || state_RECV && from_mem_rd_rsp_valid)
400             operate_cnt <= operate_cnt + 3'b1;
401     end
```

每次向主存填回的数据为 32 bit，根据填回次数确定填回的数据。对于 Cache 写回，需要写回 8 次，而对于旁路写回，只需要写回 1 次，由此确定 to_mem_wr_req_len 的值。使用 operate_cnt 作为计数器计数写回的次数，达到总共写回次数时，则拉高 to_mem_wr_data_last 值。该计数器值还用来确定每次填回所填回的数据。

c. 内存读操作

在内存读操作中主要是要增加对旁路的支持。当需要读取不可缓存区域的数据时，我们置 to_mem_rd_req_len 为 0，即只读一个字长，to_mem_rd_req_addr 也不再按 32 byte 对齐，由以下代码给出：

```
391     assign to_mem_rd_req_len = {8{state_MEM_RD}} & 8'b111 |
392                                {8{state_COMEM_RD}} & 8'b0;
393     assign to_mem_rd_req_addr = {32{state_MEM_RD}} & {cpu_address[31:5], 5'b0} |
394                                 {32{state_COMEM_RD}} & cpu_address;
```

二、 实验过程中遇到的问题、对问题的思考过程及解决方法

1. 加入数据 Cache 后即使 FPGA 能跑过，行为仿真中的 max，min3，if_else 和 select_sort 也会不通过。

这里定位问题的过程较为复杂，最后发现这是跟行仿真责检查的工作原理有关。行为仿真通过的标准是提交的指令比对完成且 MenWrite 拉高时退出。出

现问题的原因是加入数据 Cache 后 MemWrite 拉高较快（Cache 只要处于空闲状态即可），导致指令比对完成前 MemWrite 就已经拉高了。

解决方法是我将提交指令的 inst_retire 由时序逻辑赋值改为了组合逻辑赋值，这样能使得指令提交提前一个周期，在 MemWrite 拉高前就完成提交指令的比对。

三、 在课后，你花费了大约 35 小时完成此次实验?

四、 对于此次实验的心得、感受和建议

这部分实验难度不大。我认为一个原因是我完成了 DMA 的设计，对突发传输有所了解；另一个原因是课上 PPT 已经给出了详细的指令 Cache 工作过程及状态转移图，设计指令 Cache 较为简单。数据 Cache 虽然需要增加较多情况的处理，但都大同小异，自行设计状态转移图也并非是件难事。设计完状态转移图，数据 Cache 的设计水到渠成。

该部分遇到的困难主要就是行为仿真部分不通过的情况，这里定位问题花费了较长的时间。在设计出数据 Cache 后不久，两个 Cache 协同工作就已经能通过 FPGA 了，但是行为仿真中的部分情况迟迟不通过，设计数据 Cache 一半以上的时间都在试图找出问题所在。

最后感谢主讲老师及各位助教老师的讲解，让我深入了解到了 Cache 的工作原理，通过编写 Cache，我明显体会到了 Cache 对于提升运行速度的巨大作用。