中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929010 姓名：贾城昊 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。

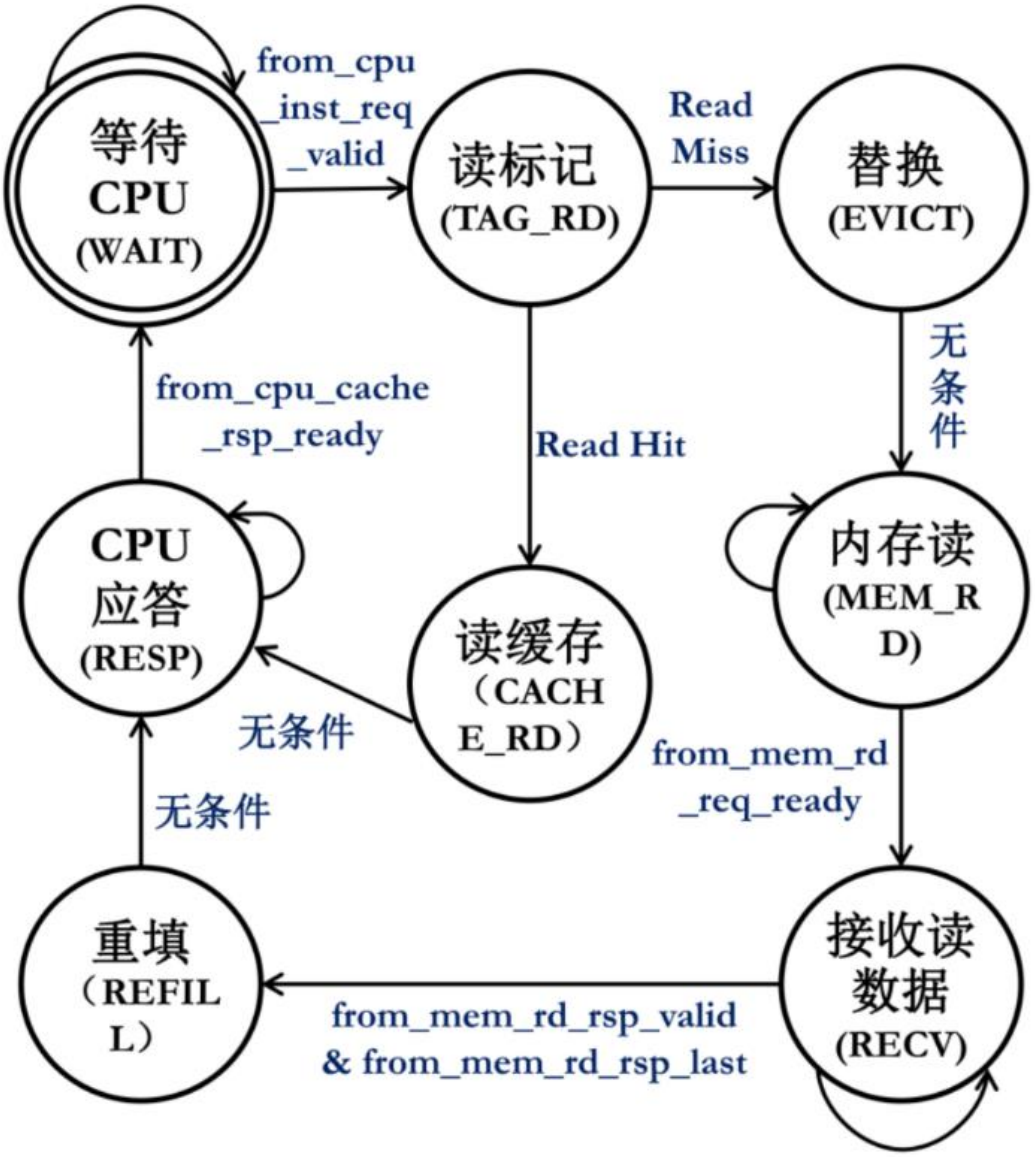
一、 逻辑电路结构与仿真波形的截图及说明（比如关键 RTL 代码段{包含注释}及其对应的逻辑电路结构图{自行画图，推荐用 PPT 画逻辑结构框图，复制到 word 中}、相应信号的仿真波形和信号变化的说明等）

高速缓存(cache)设计实验中基于 RSICV 五级流水线处理器主要完成 icache.v 与 dcache.v 两个文件的设计。其余文件 custom_cpu.v; IF_state.v, ID_state.v, EX_state.v, MEM_state.v, WB_state.v, regfile.v, alu.v, shifter.v; 在做必要修改后直接复用 prj3 以及 prj5.3 (复杂处理器设计实验)中的对应文件：

（一）. icache 设计以及相关代码:

I. 状态转移及状态机：

状态转移如 PPT 所示：



状态机使用三段式描述方法，第一段和第二段如下：

```
//FSM
always @ (posedge clk) begin
        if (rst)
                current_state <= 8'b0;
        else
                current_state <= next_state;
end

always @ (*) begin
        case (current_state)
                WAIT: begin
                        if (from_cpu_inst_req_valid) begin
                                next_state = TAG_RD;
                        end
                        else begin
                                next_state = WAIT;
                        end
                end
                TAG_RD: begin
                        if (ReadHit) begin
                                next_state = CACHE_RD;
                        end
                        else begin
                                next_state = EVICT;
                        end
                end
                EVICT: begin
                        next_state = MEM_RD;
                end
                MEM_RD: begin
                        if (from_mem_rd_req_ready) begin
                                next_state = RECV;
                        end
                        else begin
                                next_state = MEM_RD;
                        end
                end
```

```
                RECV: begin
                        if (from_mem_rd_rsp_valid & from_mem_rd_rsp_last) begin
                                next_state = REFILL;
                        end
                        else begin
                                next_state = RECV;
                        end
                end
                REFILL: begin
                        next_state = RESP;
                end
                CACHE_RD: begin
                        next_state = RESP;
                end
                RESP: begin
                        if (from_cpu_cache_rsp_ready) begin
                                next_state = WAIT;
                        end
                        else begin
                                next_state = RESP;
                        end
                end
                default:
                        next_state = WAIT;
        endcase
end
```

II. icache 替换策略与替换过程

在本次实验中，本人采用了 LRU (least recently used)来作为替换算法。其基本思想是创建一个额外的计数器。当数据块从内存读入

时 cache 时,将其置为零。在接下来的每次访问时，没访问到的另外三路 cache 对应 index 下的计数器加一。

LRU 计数器代码如下：

```
//update LRU_cnt(用于替换算法)
integer i_set;
always @ (posedge clk) begin
        if (rst) begin
                for (i_set = 0; i_set < `CACHE_SET; i_set = i_set + 1) begin
                        LRU_cnt[0][i_set] <= 8'b0;
                        LRU_cnt[1][i_set] <= 8'b0;
                        LRU_cnt[2][i_set] <= 8'b0;
                        LRU_cnt[3][i_set] <= 8'b0;
                end
        end
        else if (current_state[is_RESP] && from_cpu_cache_rsp_ready)begin
                if (~hit_way[0])
                        LRU_cnt[0][index] <= LRU_cnt[0][index] + 1;
                if (~hit_way[1])
                        LRU_cnt[1][index] <= LRU_cnt[1][index] + 1;
                if (~hit_way[2])
                        LRU_cnt[2][index] <= LRU_cnt[2][index] + 1;
                if (~hit_way[3])
                        LRU_cnt[3][index] <= LRU_cnt[3][index] + 1;
        end
        else if (current_state[is_REFILL]) begin
                LRU_cnt[way_LRU][index] <= 8'b0;
        end
end
```

替换过程逻辑如下：

```
//LRU(选择算法)
assign way_LRU = (~valid_array[0][index])? 2'b00 :
                 (~valid_array[1][index])? 2'b01 :
                 (~valid_array[2][index])? 2'b10 :
                 (~valid_array[3][index])? 2'b11 :
                 (
                        (LRU_cnt[0][index] >= LRU_cnt[1][index])?
                        (LRU_cnt[0][index] >= LRU_cnt[2][index])?
                        (LRU_cnt[0][index] >= LRU_cnt[3][index])? 2'b00 : 2'b11 :
                        (LRU_cnt[2][index] >= LRU_cnt[3][index])? 2'b10 : 2'b11 :
                        (LRU_cnt[1][index] >= LRU_cnt[2][index])?
                        (LRU_cnt[1][index] >= LRU_cnt[3][index])? 2'b01 : 2'b11 :
                        (LRU_cnt[2][index] >= LRU_cnt[3][index])? 2'b10 : 2'b11
                 );
```

III. valid 数组的更新逻辑

初始时，valid 为 0，在 EVICT 状态时，把对应的 valid 清 0，

在 REFILL 阶段完成对 valid 的更新，具体如下：

```
//valid_array
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 (current_state[is_EVICT]) begin
                valid_array[way_LRU][index] <= 1'b0;
        end
        else if (current_state[is_REFILL]) begin
                valid_array[way_LRU][index] <= 1'b1;
        end
end
```

IV. 对内存数据的接受逻辑

在 RECV 状态从内存中接收数据,依次存放至 data_recv 各个寄存器。并在 REFILL 状态写回 cache,包括 tag 的写回和 data 的写回，及 valid 的更新，如下图所示：

RECV 状态接收数据：

```
//update read_data(从内存读数据)
always @ (posedge clk) begin
        if(rst) begin
                read_data_len <=  2'b0;
        end
        else if (current_state[is_MEM_RD] && from_mem_rd_req_ready) begin
                read_data_len <= 2'b0;
        end
        else if (current_state[is_RECV] && from_mem_rd_rsp_valid) begin
                read_data_len <= read_data_len + 1'b1;
        end
end

always @ (posedge clk) begin
        if (current_state[is_RECV] && from_mem_rd_rsp_valid) begin
                read_data[read_data_len] <= from_mem_rd_rsp_data;
        end
end
```

REFILL 状态完成 data 和 tag 的写回：

```
//wen && data_wdata
assign wen[0] = current_state[is_REFILL] & (way_LRU == 2'b00);
assign wen[1] = current_state[is_REFILL] & (way_LRU == 2'b01);
assign wen[2] = current_state[is_REFILL] & (way_LRU == 2'b10);
assign wen[3] = current_state[is_REFILL] & (way_LRU == 2'b11);

assign data_wdata = {read_data[7], read_data[6], read_data[5], read_data[4], read_data[3], read_data[2], read_data[1], read_data[0]};
```

```
//例化
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])
);
```

V. 请求地址字段划分及信号生成

```
//HandShake Signals
assign to_cpu_inst_req_ready = current_state[is_WAIT];
assign to_cpu_cache_rsp_valid = current_state[is_RESP];
assign to_mem_rd_req_valid = current_state[is_MEM_RD];
assign to_mem_rd_rsp_ready = current_state[is_RECV];

//Input Decode
assign {tag, index, offset} = from_cpu_inst_req_addr;
assign to_mem_rd_req_addr = {from_cpu_inst_req_addr[31:5], 5'b0};

//ReadHit or ReadMiss
assign hit_way[0] = valid_array[0][index] & (tag_rdata[0] == tag);
assign hit_way[1] = valid_array[1][index] & (tag_rdata[1] == tag);
assign hit_way[2] = valid_array[2][index] & (tag_rdata[2] == tag);
assign hit_way[3] = valid_array[3][index] & (tag_rdata[3] == tag);

assign ReadHit = hit_way[0] | hit_way[1] | hit_way[2] | hit_way[3];
assign ReadMiss = ~ReadHit;
```

VI. 传给 CPU 的数据

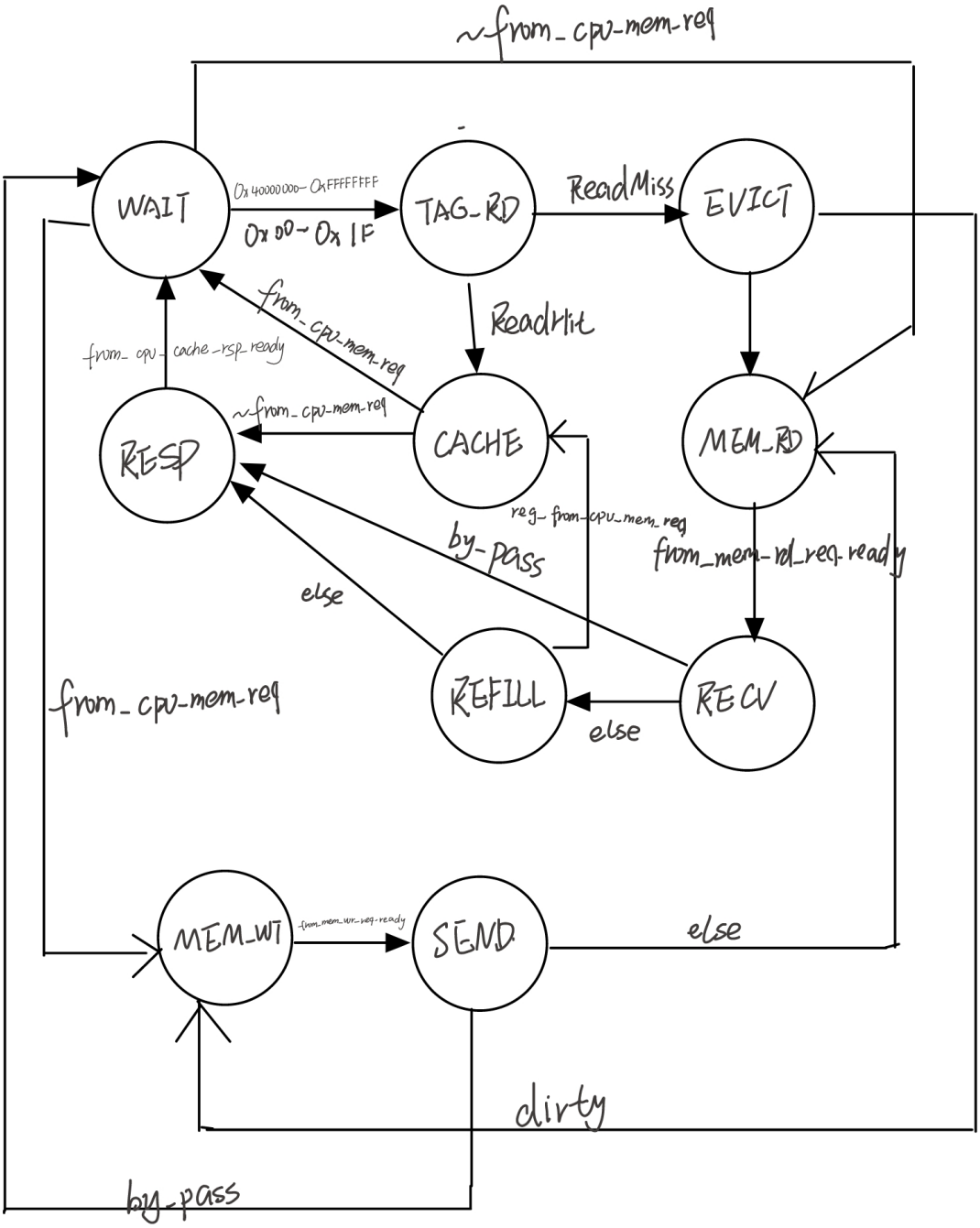
根据 tag 进行数据选择，并根据 offset 进行偏移，如下所示：

```
//data to cpu
assign data_selected =  {`LINE_LEN{hit_way[0]}} & data_rdata[0] |
                        {`LINE_LEN{hit_way[1]}} & data_rdata[1] |
                        {`LINE_LEN{hit_way[2]}} & data_rdata[2] |
                        {`LINE_LEN{hit_way[3]}} & data_rdata[3];
assign to_cpu_cache_rsp_data = data_selected >> {offset, 3'b0};
```

（二）. dcache 设计以及相关代码

I. 状态转移及状态机

本次实验中，本人的 dcache 一共设计了 10 个状态，其状态转移大致如下图所示：



状态机采用三段式的结构描述，第一段和第二段的代码具体如下：

```
// current_state
always @(posedge clk) begin
    if (rst)
        current_state <= WAIT;
    else
        current_state <= next_state;
end

// next_state
always @(*) begin
    case (current_state)
        WAIT: begin
            if (from_cpu_mem_req_valid) begin
                if (|from_cpu_mem_req_addr[31:30] | ~|from_cpu_mem_req_addr[31:5]) begin
                    if (from_cpu_mem_req) begin  //MemWrite
                        next_state = MEM_WT;
                    end
                    else begin                   //MemRead
                        next_state = MEM_RD;
                    end
                end
                else begin
                    next_state = TAG_RD;     //not bypass
                end
            end
            else begin
                next_state = WAIT;
            end
        end
        TAG_RD: begin
            if (hit) begin                   //Hit Cache
                next_state = CACHE;
            end
            else begin
                next_state = EVICT;
            end
        end
```

```
CACHE: begin
    if (reg_from_cpu_mem_req)  //MeMWrite -> don't need resp
        next_state = WAIT;
    else                       // MemRead -> need resp
        next_state = RESP;
end
RESP: begin
    if (from_cpu_cache_rsp_ready) //握手成功
        next_state = WAIT;
    else
        next_state = RESP;
end
EVICT: begin
    if (dirty)                  //dirty -> need Write
        next_state = MEM_WT;
    else
        next_state = MEM_RD;
end
MEM_RD: begin
    if (from_mem_rd_req_ready) //内存请求握手成功
        next_state = RECV;
    else
        next_state = MEM_RD;
end
RECV: begin
    if (from_mem_rd_rsp_valid & from_mem_rd_rsp_last) begin
        if (bypass)             //bypass -> don't need refill Cache
            next_state = RESP;
        else
            next_state = REFILL;
    end
    else
        next_state = RECV;
end
```

```
            REFILL: begin
                if (reg_from_cpu_mem_req)  // MemWrite -> need to write Cache
                    next_state = CACHE;
                else                       // MemRead -> don't need to write Cache
                    next_state = RESP;
            end
            MEM_WT: begin
                if (from_mem_wr_req_ready) //内存请求握手成功
                    next_state = SEND;
                else
                    next_state = MEM_WT;
            end
            SEND: begin
                if (from_mem_wr_data_ready & to_mem_wr_data_last) begin
                    if (bypass)                 //bypass -> end
                        next_state = WAIT;
                    else                        //not bypass -> continue to write Cache
                        next_state = MEM_RD;
                end
                else
                    next_state = SEND;
            end
            default: begin
                next_state = WAIT;
            end
        endcase
    end
```

各个状态的功能如下：

编号	名称	描述
0	WAIT	等待 CPU 访存请求
1	TAG_RD	判断当前 TAG 对应块是否已被缓存
2	CACHE	完成对 cache 块的读与写
3	RESP	将读取到的数据返还 CPU
4	EVICT	根据 LRU 算法选出待替换的缓存块(dirty 需要写回内存)
5	MEM_RD	向内存发送读请求
6	RECV	从内存接收新数据（包括旁路和 cache）
7	REFILL	将收到的数据填入对应缓存块，并重置 LRU 计数器，

		valid、dirty、tag 标记
8	MEM_WT	向内存发送写请求
9	SEND	将数据写回内存（包括 cache 和旁路）

II. dache 替换策略

与 icache 替换策略相同，这里不赘述了

```
//update LRU_cnt(用于替换算法)
integer i_set;
always @ (posedge clk) begin
    if (rst) begin
        for (i_set = 0; i_set < `CACHE_SET; i_set = i_set + 1) begin
            LRU_cnt[0][i_set] <= 8'b0;
            LRU_cnt[1][i_set] <= 8'b0;
            LRU_cnt[2][i_set] <= 8'b0;
            LRU_cnt[3][i_set] <= 8'b0;
        end
    end
    else if (current_state[is_CACHE])begin
        if (~hit_way[0])
                LRU_cnt[0][index] <= LRU_cnt[0][index] + 1;
        if (~hit_way[1])
                LRU_cnt[1][index] <= LRU_cnt[1][index] + 1;
        if (~hit_way[2])
                LRU_cnt[2][index] <= LRU_cnt[2][index] + 1;
        if (~hit_way[3])
                LRU_cnt[3][index] <= LRU_cnt[3][index] + 1;
    end
    else if (current_state[is_REFILL]) begin
            LRU_cnt[way_LRU][index] <= 8'b0;
    end
end
```

```
//LRU(选择算法)
assign way_LRU = (~valid_array[0][index])? 2'b00 :
        (~valid_array[1][index])? 2'b01 :
        (~valid_array[2][index])? 2'b10 :
        (~valid_array[3][index])? 2'b11 :
        (
            (LRU_cnt[0][index] >= LRU_cnt[1][index])?
            (LRU_cnt[0][index] >= LRU_cnt[2][index])?
            (LRU_cnt[0][index] >= LRU_cnt[3][index])? 2'b00 : 2'b11 :
            (LRU_cnt[2][index] >= LRU_cnt[3][index])? 2'b10 : 2'b11 :
            (LRU_cnt[1][index] >= LRU_cnt[2][index])?
            (LRU_cnt[1][index] >= LRU_cnt[3][index])? 2'b01 : 2'b11 :
            (LRU_cnt[2][index] >= LRU_cnt[3][index])? 2'b10 : 2'b11
        );
```

III. 控制信号及握手信号

(1). 握手信号

根据状态的含义，确定握手信号如下：

```
//HandShake Signals
assign to_cpu_mem_req_ready = current_state[is_WAIT];
assign to_cpu_cache_rsp_valid = current_state[is_RESP];

assign to_mem_rd_req_valid = current_state[is_MEM_RD];
assign to_mem_rd_rsp_ready = current_state[is_RECV];

assign to_mem_wr_req_valid = current_state[is_MEM_WT];
assign to_mem_wr_data_valid = current_state[is_SEND];

// decode
assign {tag, index, offset} = reg_from_cpu_mem_req_addr;
```

(2) . 旁路判断信号，dirty 判断信号， cache 命中信号

旁路根据 cpu 请求的地址判断，dirty 直接从 dirty 数组读出，hit 则是根据 valid 数组和 tag 比较得到：

```
//bypass
assign bypass = |reg_from_cpu_mem_req_addr[31:30] | ~|reg_from_cpu_mem_req_addr[31:5];

//Hit
assign hit_way[0] = valid_array[0][index] & (tag_rdata[0] == tag);
assign hit_way[1] = valid_array[1][index] & (tag_rdata[1] == tag);
assign hit_way[2] = valid_array[2][index] & (tag_rdata[2] == tag);
assign hit_way[3] = valid_array[3][index] & (tag_rdata[3] == tag);

assign hit = |hit_way;
assign hit_way_num = {hit_way[3] | hit_way[2],
                      hit_way[3] | hit_way[1]};
```

```
// dirty
assign dirty = dirty_array[way_LRU][index];
```

IV. dirty 数组和 valid 数组

valid 数组与 icache 类似

```
//valid array
always @(posedge clk) begin
    if (rst) begin
        valid_array[0] <= 8'b0;
        valid_array[1] <= 8'b0;
        valid_array[2] <= 8'b0;
        valid_array[3] <= 8'b0;
    end
    else if (current_state[is_EVICT]) begin  // EVICT
        valid_array[way_LRU][index] <= 0;
    end
    else if (current_state[is_REFILL]) begin  // REFILL
        valid_array[way_LRU][index] <= 1;
    end
end
```

dirty 数组在 REFILL 状态，对应位置清 0（数据刚填到 cache 中），在对 cache 进行写后，对应位置改为 1，如下所示：

```
//dirty array
always @(posedge clk) begin
    if (rst) begin
        dirty_array[0] <= 8'b0;
        dirty_array[1] <= 8'b0;
        dirty_array[2] <= 8'b0;
        dirty_array[3] <= 8'b0;
    end
    else if (current_state[is_CACHE] & reg_from_cpu_mem_req) begin  // CACHE & write
        dirty_array[hit_way_num][index] <= 1;
    end
    else if (current_state[is_REFILL]) begin                       // REFILL
        dirty_array[way_LRU][index] <= 0;
    end
end
```

V. burst 传输

设计一个计数器，用来进行与内存的数据传输：

```
//rd_buffer
always @(posedge clk) begin
    if (to_mem_rd_rsp_ready & from_mem_rd_rsp_valid) begin
        rd_buffer[burst_num] <= from_mem_rd_rsp_data;
    end
end

//burst_num
always @(posedge clk) begin
    if(rst) begin
        burst_num <= 8'b0;
    end
    else if (current_state[is_MEM_RD] | current_state[is_MEM_WT]) begin   // reset or MEM_RD or MEM_WT
        burst_num <= 8'b0;
    end
    else if ((to_mem_rd_rsp_ready & from_mem_rd_rsp_valid) | (from_mem_wr_data_ready & to_mem_wr_data_valid)) begin
        burst_num <= burst_num + 1;
    end
end
```

VI. Memory 的读写地址，burst 长度以及返还 cpu 的数据

根据是否是旁路，对地址，burst 长度以及返回 cpu 的数据进行选择：

```
// rsp_data
assign to_cpu_cache_rsp_data = bypass ? rd_buffer[0]
                             : cache_data >> {offset, 3'b0};

//Mem_Read request addr & len
assign to_mem_rd_req_addr = bypass ? {reg_from_cpu_mem_req_addr[31:2], 2'b0}
                                   : {reg_from_cpu_mem_req_addr[31:5], 5'b0};
assign to_mem_rd_req_len = bypass ? 8'b0 : 8'b111;

//Mem_Write request addr & len
assign to_mem_wr_req_addr = bypass ? {reg_from_cpu_mem_req_addr[31:2], 2'b0}
                                   : {evict_tag, index, 5'b0};
assign to_mem_wr_req_len = bypass ? 8'b0 : 8'b111;

// Mem_Write
assign to_mem_wr_data_strb = bypass ? reg_from_cpu_mem_req_wstrb : 4'b1111;

assign to_mem_wr_data = bypass ? reg_from_cpu_mem_req_wdata
                               : evict_data >> {burst_num, 5'b0};

assign to_mem_wr_data_last = (burst_num == to_mem_rd_req_len);
```

VII. data 数组以及 tag 数组

data 数组的写数据需要根据是 cpu 还是 mem 来的进行选择，本人在这个过程中使用了掩码，方便处理，其它信号根据状态机状态的含义确定即可，如下图所示：

```
//  data array
assign cache_write = current_state[is_CACHE] & reg_from_cpu_mem_req; //Write Cache Signal
assign write_mask = {{8{reg_from_cpu_mem_req_wstrb[3]}},
                     {8{reg_from_cpu_mem_req_wstrb[2]}},
                     {8{reg_from_cpu_mem_req_wstrb[1]}},
                     {8{reg_from_cpu_mem_req_wstrb[0]}}} << {offset, 3'b000}; //Mask --字节掩码

assign data_wdata = cache_write ? (((reg_from_cpu_mem_req_wdata) << {offset, 3'b000}) & write_mask) | (cache_data & ~write_mask)
                  : {rd_buffer[7], rd_buffer[6], rd_buffer[5], rd_buffer[4], rd_buffer[3], rd_buffer[2], rd_buffer[1], rd_buffer[0]};

assign data_wen = {4{current_state[is_REFILL]}} & evict_way |  //REFILL on  read / write miss
                 {4{cache_write}} & hit_way;  //  write not miss;

assign cache_data =  hit_way[0] ? data_rdata[0]
                   : hit_way[1] ? data_rdata[1]
                   : hit_way[2] ? data_rdata[2]
                   : data_rdata[3];

assign evict_data =   evict_way[0] ? data_rdata[0]
                    : evict_way[1] ? data_rdata[1]
                    : evict_way[2] ? data_rdata[2]
                    : data_rdata[3];
```

```
//例化
data_array data_way_0 (
    .clk(clk),
    .waddr(index),
    .raddr(index),
    .wen(data_wen[0]),
    .wdata(data_wdata),
    .rdata(data_rdata[0])
);
data_array data_way_1 (
    .clk(clk),
    .waddr(index),
    .raddr(index),
    .wen(data_wen[1]),
    .wdata(data_wdata),
    .rdata(data_rdata[1])
);
data_array data_way_2 (
    .clk(clk),
    .waddr(index),
    .raddr(index),
    .wen(data_wen[2]),
    .wdata(data_wdata),
    .rdata(data_rdata[2])
);
data_array data_way_3 (
    .clk(clk),
    .waddr(index),
    .raddr(index),
    .wen(data_wen[3]),
    .wdata(data_wdata),
    .rdata(data_rdata[3])
);
```

```
// tag array
assign evict_tag =   evict_way[0] ? tag_rdata[0]
                   : evict_way[1] ? tag_rdata[1]
                   : evict_way[2] ? tag_rdata[2]
                   : tag_rdata[3];

//例化
tag_array tag_way_0 (
    .clk(clk),
    .waddr(index),
    .raddr(index),
    .wen(data_wen[0]),
    .wdata(tag),
    .rdata(tag_rdata[0])
);
tag_array tag_way_1 (
    .clk(clk),
    .waddr(index),
    .raddr(index),
    .wen(data_wen[1]),
    .wdata(tag),
    .rdata(tag_rdata[1])
);
tag_array tag_way_2 (
    .clk(clk),
    .waddr(index),
    .raddr(index),
    .wen(data_wen[2]),
    .wdata(tag),
    .rdata(tag_rdata[2])
);
tag_array tag_way_3 (
    .clk(clk),
    .waddr(index),
    .raddr(index),
    .wen(data_wen[3]),
    .wdata(tag),
    .rdata(tag_rdata[3])
);
```

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）

本次实验的硬件部分代码 debug 较为麻烦，因为流水线的握手较为复杂，且还需要考虑与内存的握手，所以 debug 耗费了本人特别多的时

间。对于本次实验中遇到的 bug，主要有以下几个：

遇到的重要 bug 以及解决过程：

1. data_array.v 和 tag_array.v 中移位逻辑出错：

本人最开始发现 waddr 和 wen 有效时，data_array 的数据却没有写入，后面经过排查。发现是因为老师之前提供的 data_array.v 和 tag_array.v 中有些小 bug，原来的代码如下图所示：

```
module data_array(
        input                               clk,
        input  [`DARRAY_ADDR_WIDTH - 1:0] waddr,
        input  [`DARRAY_ADDR_WIDTH - 1:0] raddr,
        input                               wen,
        input  [`DARRAY_DATA_WIDTH - 1:0] wdata,
        output [`DARRAY_DATA_WIDTH - 1:0] rdata
);

        reg [`DARRAY_DATA_WIDTH-1:0] array[ 1 << `DARRAY_ADDR_WIDTH - 1 : 0];

        always @(posedge clk)
        begin
                if(wen)
                        array[waddr] <= wdata;
        end

assign rdata = array[raddr];
```

这里在定义 array 数组的长度时，使用了如红线所示的运算。但事实上，因为左移的优先级低于减法，这个操作会先执行减一运算，再执行左移运算。导致数组大小的定义出了问题。

解决方案是加个括号即可：

```
module data_array(
        input                               clk,
        input  [`DARRAY_ADDR_WIDTH - 1:0] waddr,
        input  [`DARRAY_ADDR_WIDTH - 1:0] raddr,
        input                               wen,
        input  [`DARRAY_DATA_WIDTH - 1:0] wdata,
        output [`DARRAY_DATA_WIDTH - 1:0] rdata
);

        reg [`DARRAY_DATA_WIDTH-1:0] array[ (1 << `DARRAY_ADDR_WIDTH )- 1 : 0];

        always @(posedge clk)
        begin
                if(wen)
                        array[waddr] <= wdata;
        end

assign rdata = array[raddr];

endmodule
```

（tag_array 一样的，就不重复放图了）

2. 加上 cache 后，访存太快，导致部分行为仿真无法结束：

本人写完 cache 后，发现部分行为仿真会卡死在最后一句，而 fpga 和仿真加速，均没问题，其最后波形图如下：

可以看到，trace_end 已经拉高了，但行为仿真并未结束，起初

本人以为是自己的问题，但在查询无果后，本人检查了框架代码，发现判断结束的条件如下：

```
// End
always @(posedge sys_clk)
begin
    if (trace_end & (`MEM_WEN == 1'b1) & (`MEM_ADDR == 32'h0C) & (`MEM_WDATA == 32'h0))
    begin
        $display("================================================");
        $display("Benchmark simulation passed!!!");
        $display("================================================");
        $fclose(trace_file);
        $finish;
    end
end
```

他需要在 trace_end 拉高的同时，MEM_WEN,即 MemWrite 拉高，但波形中两者刚好差了一个周期，这是因为本人退役指令使用的是时序逻辑，而加了 cache 后，访存很快，导致这里出错，讲退役指令改为如下组合逻辑，行为仿真即可完成：

```
assign inst_retire = (MEM_to_WB_Valid & WB_Allow_in) ? MEM_to_WB_Bus : 70'b0;
```

（原本时序逻辑可以看 prj5-turbo 的实验报告）

3. 其它错误：

除上述错误外，本人也犯过一些小错误，例如由于本人状态机设计的状态设计的较少（例如本人室友状态机状态便设计了 16 个，而本人只设计了 10 个），所以出现了对旁路的兼容出错的情况，又或者是声明位宽出错等等，但这些大多是自己的逻辑问题，通过查看波形图可以较快的定位到出错位置，这里就不一一赘述了。

三、 对讲义中思考题（如有）的理解和回答

1. 如何实现大于 4 byte 的指令/数据交互?

使用突发(Burst)传输。在发起请求后,连续传输多个数据。用 last 信号标志本次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。（与 dma 有类似之处）

2. 如果只传输 32-bit 数据（len=0），last 在哪个位置拉高?

直接把 len 设置为 0,让在第 0 个数据 valid 时,就拉高 last 信号即可。

3. 性能对比:

未加 cache 的五级流水线的周期计数器如下图所示:

```
[ssort] Suffix sort: * Passed.
Result:
       Cycles: at least 47921756
       Instruction Count: at least 686410
       Memory Read Instruction Count: at least 11290
       Memory Write Instruction Count: at least 7427
       Branch Instruction Count: at least 132631
       Jump Instruction Count: at least 869
benchmark finished
time 1294.85ms
```

```
[sieve] Eratosthenes sieve: * Passed.
Result:
        Cycles: at least 822381
        Instruction Count: at least 11601
        Memory Read Instruction Count: at least 344
        Memory Write Instruction Count: at least 129
        Branch Instruction Count: at least 2496
        Jump Instruction Count: at least 46
benchmark finished
time 34.32ms
```

```
[queen] Queen placement: * Passed.
Result:
        Cycles: at least 6103882
        Instruction Count: at least 87084
        Memory Read Instruction Count: at least 14419
        Memory Write Instruction Count: at least 12356
        Branch Instruction Count: at least 6078
        Jump Instruction Count: at least 4118
benchmark finished
```

加了 cache 的五级流水线的性能计数器如下图所示

```
[ssort] Suffix sort: * Passed.
Result:
        Cycles: at least 3857867
        Instruction Count: at least 752561
        Memory Read Instruction Count: at least 11290
        Memory Write Instruction Count: at least 7427
        Branch Instruction Count: at least 132631
        Jump Instruction Count: at least 869
benchmark finished
time 120.74ms
```

```
[sieve] Eratosthenes sieve: * Passed.
Result:
       Cycles: at least 65341
       Instruction Count: at least 12742
       Memory Read Instruction Count: at least 344
       Memory Write Instruction Count: at least 129
       Branch Instruction Count: at least 2496
       Jump Instruction Count: at least 46
benchmark finished
time 25.51ms
```

```
[queen] Queen placement: * Passed.
Result:
       Cycles: at least 459993
       Instruction Count: at least 91691
       Memory Read Instruction Count: at least 14419
       Memory Write Instruction Count: at least 12356
       Branch Instruction Count: at least 6078
       Jump Instruction Count: at least 4118
benchmark finished
time 29.05ms
```

可以看出，cache 的效果是非常显著的，相比于未加 cache 的流水线，其周期数只有原来的十四分之一到十二分之一，其 CPI 大多在 4-6 之间。

对于 CPI 稳定在 4-6 之间,并非理想中的 1 左右。本人认为可能原因如下:

1）． cpu 访问 cache 命中率有限,存在一定与内存交互的时间;若程序的指令/数据分布比较分散, cache 命中率将进一步降低,导致访问内存次数增加,进而使得 CPI 上升。

2）．本次实验中实现的 icache 与 dcache 没有使用流水线,而是多周期。即便命中 cache, cache 也需要多个周期完成数据写入/输出。

四、 在课后，你花费了大约___50__小时完成此次实验。

五、 对于此次实验的心得、感受和建议（比如实验是否过于简单或复杂，是否缺少了某些你认为重要的信息或参考资料，对实验项目的建议，对提供帮助的同学的感谢，以及其他想与任课老师交流的内容等）

相较于流水线，这次实验较为简单。icache 部分原理简单，而且由于资料比较完善，易于实现。dcache 部分相较 icache 部分复杂一点，但是由于之前实现过 icache,加上 dcache 部分自由空间很大，因此实际设计和实现并不算复杂。

但是由于 data_array.v 和 tag_array.v 如之前提到的那样存在一些错误，给波形调试增加了一些工作。而且行为仿真的结束判定，导致本人在这上面花费的时间也较久，最后查看框架代码才知道原因。所幸本次实验 debug 较为方便，其它的逻辑错误已经各种小错误并不难发现、纠正

最后特别感谢老师，助教们和实验群的同学对本人提出的问题的帮助与解答。在本次实验中，行为仿真框架检查的问题也是本人和实验群里多位同学努力后才得以发现。

最后希望计组研讨课越来越好，能给学弟学妹更好的体验，本学期本

人做了四个选座实验，对于 Verilog 语言以及计算机相关知识了解了很多，感谢这门课让我学到了这么多的知识。