中国科学院大学
《计算机组成原理(研讨课)》实验报告

姓名 薛翼舟 学号 2023K8009929044 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 的设计

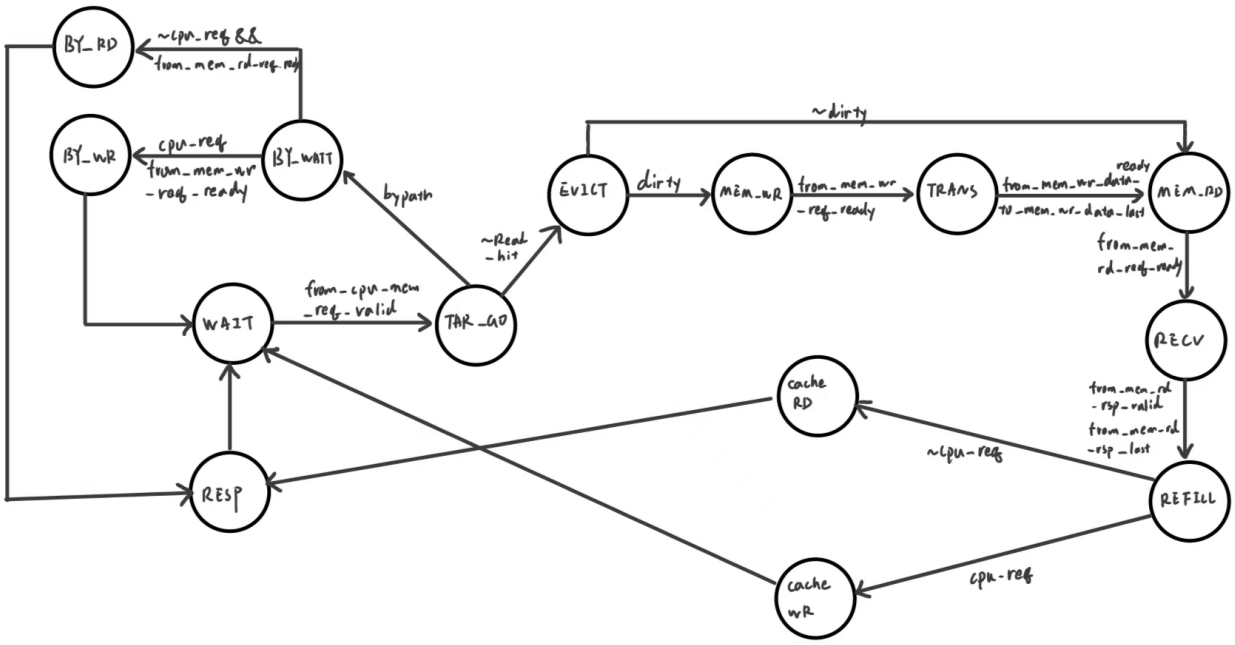
注 1:	撰写此 Word 格式实验报告后以 PDF 格式保存 SERVE CloudIDE 的 `/home/serve-ide/cod-lab/reports` 目录下(注意: reports 全部小写)。文件命名规则: `prjN.pdf`, 其中 `prj` 和后缀名 `pdf` 为小写, `N` 为 1 至 4 的阿拉伯数字。例如: `prj1.pdf`。PDF 文件大小应控制在 5MB 以内。此外, 实验项目 5 包含多个选做内容, 每个选做实验应提交各自的实验报告文件, 文件命名规则: `prj5-projectname.pdf`, 其中 "-" 为英文标点符号的短横线。文件命名举例: `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 {自行画图, 推荐用 PPT 画逻辑结构框图后保存为 PDF,再插入到 LaTeX. 中}、相应信号的仿真波形和信号变化的说明等)

这次实验主要分为两个部分, 分别是 icache 指令 cache 和 dcache 数据 cache, 由于 dcache 的作用能够完全覆盖 icache 的功能, 因此在本报告中只对 dcache 的设计进行讲解, icache 不多赘述, 本次实验的关键点在于要知道各个端口的作用, 以及 cache 整个的逻辑状态转变

1.1 cache 的状态转移图

我一共对 dcache 设计了 14 个状态, 状态转移图如下



j 这 14 个状态相比 icache 所给出的简单的状态机, 添加了几个部分, 分别是对于写部分的逻辑, 添加了 cache_wr 和 dirty 替换等有关写的逻辑和连接 I/O 接口等的数据旁路的逻辑, 主要思想是如果没有命中, 那么就需要替从内存中替换, 如果将要替换的 cache 块中是 dirty 的, 那么就需要将其写回内存, 之后再从内存中取出, 状态机部分的代码如下

```
always @(*) begin
    case (dcache_current_state)
            WAIT: begin
                    if(from_cpu_mem_req_valid) begin
                            dcache_next_state = TAG_RD;
                    end
                    else begin
                            dcache_next_state = WAIT;
                    end
            end
            TAG_RD: begin
                    if(bypath) begin//表示旁路读
                            dcache_next_state = BY_WAIT;
                    end
                    else if(~cpu_req && Read_hit) begin      //表示读命中
                            dcache_next_state = CACHE_RD;
                    end
```

```
18                      else if(cpu_req && Read_hit) begin      //表示写命中
19                              dcache_next_state = CACHE_WR;
20                      end
21                      else if(~Read_hit) begin
22                              dcache_next_state = EVICT;
23                      end
24                      else begin
25                              dcache_next_state = TAG_RD;
26                      end
27              end
28              BY_WAIT: begin
29                      if(~cpu_req && from_mem_rd_req_ready) begin//表示旁路读
30                              dcache_next_state = BY_RD;
31                      end
32                      else if(cpu_req && from_mem_wr_req_ready) begin
33          //表示旁路写
34                              dcache_next_state = BY_WR;
35                      end
36                      else begin
37                              dcache_next_state = BY_WAIT;
38                      end
39              end
40              EVICT: begin
41                      if(dirty) begin
42                              dcache_next_state = MEM_WR;//需要写来进行替换
43                      end
44                      else begin
45                              dcache_next_state = MEM_RD;
46                      end
47              end
48              MEM_WR: begin
49                      if(from_mem_wr_req_ready) begin
50                              dcache_next_state = TRANS;
51                      end
52                      else begin
53                              dcache_next_state = MEM_WR;
54                      end
55              end
```

```
56                  TRANS: begin
57                          if(from_mem_wr_data_ready && to_mem_wr_data_last) begin
58                                  dcache_next_state = MEM_RD;
59                  //由于最终的结果是write_strb和内存32位共同组成的结果，所以有读的过程
60                          end
61                          else begin
62                                  dcache_next_state = TRANS;
63                          end
64                  end
65                  MEM_RD: begin
66                          if(from_mem_rd_req_ready) begin
67                                  dcache_next_state = RECV;
68                          end
69                          else begin
70                                  dcache_next_state = MEM_RD;
71                          end
72                  end
73                  RECV:   begin
74                          if(from_mem_rd_rsp_valid && from_mem_rd_rsp_last) begin
75                                  dcache_next_state = REFILL;
76                  //接受数据一直到RECV位置
77                          end
78                          else begin
79                                  dcache_next_state = RECV;
80                          end
81                  end
82                  REFILL: begin
83                          if(cpu_req) begin
84                                  dcache_next_state = CACHE_WR;
85                          end
86                          else begin
87                                  dcache_next_state = CACHE_RD;
88                          end
89                  end
90                  CACHE_RD:begin
91                          dcache_next_state = RESP;
92                  end
93                  CACHE_WR:begin
```

```
                        dcache_next_state = WAIT;
                        //dcache_next_state = RESP;写不需要CPU应答
                end
                BY_WR:begin
                        if(from_mem_wr_data_ready && to_mem_wr_data_last) begin
                                dcache_next_state = WAIT;
                        end
                        else begin
                                dcache_next_state = BY_WR;
                        end
                end
                BY_RD:begin
                        if(from_mem_rd_rsp_last && from_mem_rd_rsp_valid) begin
                                dcache_next_state = RESP;
                        end
                        else begin
                                dcache_next_state = BY_RD;
                        end
                end
                RESP:begin
                        dcache_next_state = WAIT;
                end
                default: begin
                        dcache_next_state = WAIT;
                end
        endcase
end
```

以下是对输入命令的地址解释 (块内偏移, 块号, 字节地址)

```
//在cache中有不可缓存的空间，用bypath来标记
//0x00~0x1F 0x4000 0000
assign bypath = (cpu_req_addr[31] || cpu_req_addr[30])
|| (cpu_req_addr[31:5]==27'b0);

//cpu发出请求的分段
//分段cpu请求
wire    [`TAG_LEN-1:0]  tag     =       cpu_req_addr[31:8];
wire    [2:0]           index   =       cpu_req_addr[7:5];
//index标记了在一路cache块中的地址
```

```
wire    [4:0]           offset  =       cpu_req_addr[4:0];
```

总体的对于 cpu 内存相应的过程就是, cpu 向 cache 发出内存请求, 之后如果检查 cpu 输入的地址是否是不可缓存的, 如果是不可缓存的, 就转入旁路处理流程。如果是可缓存的, 那么就检查 cache 中是否有对应的块, 如果有, 那么就直接返回数据, 如果没有, 那么就需要替换, 替换的过程分为两种情况, 如果替换的块是 dirty 的, 那么就需要先将其写回内存, 然后再从内存中读取新的数据块。另外如果是写请求, 需要再最后写回之后将 dirty 标记为 1, 用来表示当前块内的这些数据是需要写回的。

1.2 命中与否的逻辑判断

在 dcache 中, 需要判断是否命中, 其具体判断的逻辑如下

```
wire    [23:0]  tag_way0;
wire    [23:0]  tag_way1;
wire    [23:0]  tag_way2;
wire    [23:0]  tag_way3;

assign  hit_way0 = valid_way0[index] && (tag_way0==tag);
assign  hit_way1 = valid_way1[index] && (tag_way1==tag);
assign  hit_way2 = valid_way2[index] && (tag_way2==tag);
assign  hit_way3 = valid_way3[index] && (tag_way3==tag);

assign  Read_hit = hit_way0 || hit_way1 || hit_way2 || hit_way3;
```

就是比较当前四路中对应的 valid 是否为 1, 如果为 1, 那么就比较 tag 是否相同, 如果相同, 那么就表示命中, 否则表示未命中。最终的命中结果是 Read_hit, 如果四路中有一个命中, 那么 Read_hit 就为 1, 否则为 0。

1.3 替换的逻辑

在本次实验中, 我采用的替换逻辑是 PLRU 树的替换 (由于 LRU 矩阵的空间复杂度比较高), 其主要的思想就是利用树来表示四路 cache 的替换逻辑, 根据每次访问的目标来更改这个树的逻辑状态, dcache 中对于访问目标块的逻辑如下

```
always @(posedge clk) begin
        if(rst) begin
                Read_hit_reg <= 0;
                visit_way0   <= 0;
                visit_way1   <= 0;
                visit_way2   <= 0;
                visit_way3   <= 0;
```

```
        end
        else if(dcache_current_state == TAG_RD) begin//对于命中的部分
                Read_hit_reg <= Read_hit;
                visit_way0   <= hit_way0;
                visit_way1   <= hit_way1;
                visit_way2   <= hit_way2;
                visit_way3   <= hit_way3;
        end
        else if(dcache_current_state == EVICT) begin//对于更新的部分
                visit_way0   <= replace_way0;
                visit_way1   <= replace_way1;
                visit_way2   <= replace_way2;
                visit_way3   <= replace_way3;
        end
        else begin
                Read_hit_reg <= Read_hit_reg;
                visit_way0   <= visit_way0;
                visit_way1   <= visit_way1;
                visit_way2   <= visit_way2;
                visit_way3   <= visit_way3;
        end
end
```

其中 visit_way0 visit_way3 是四路 cache 的访问状态, 访问有两种, 一种就是直接命中, 另一种就是被替换, 访问状态为 1 表示访问了该块, 访问状态为 0 表示没有访问该块, 之后我们根据 visit_way 的值来更新 PLRU 树的值, 代码如下

```
always @(posedge clk) begin
        if(rst) begin
                plru_tree[0]=3'b0; plru_tree[1]=3'b0;
        plru_tree[2]=3'b0; plru_tree[3]=3'b0;
                plru_tree[4]=3'b0; plru_tree[5]=3'b0;
        plru_tree[6]=3'b0; plru_tree[7]=3'b0;
        end
        //每次应答的时候更新plru树，dache中由于有写操作，修改一下
        //而且旁路不需要访问cache中的存储
        else if (dcache_current_state==CACHE_RD || dcache_current_state==CACHE_WR
    || dcache_current_state == REFILL) begin
                if(visit_way0 || visit_way1) begin
                        plru_tree[index][0] <= 1;
```

```
                        plru_tree[index][1] <= visit_way0;
            //如果访问了0, 就指向右边的1
                end
                else begin
                        plru_tree[index][0] <= 0;
                        plru_tree[index][2] <= visit_way2;
                end
        end
end
```

其主要思路就是根据当前访问的对象, 来更新 PLRU 树的状态, 使得我们每次更新替换, 替换的都是相对而言访问的比较少的块, 来保证 cache 的命中率

1.4 dirty 逻辑的判断

我们会遇到 dirty 的逻辑, dirty 的逻辑主要是用来标记当前块是否需要写回内存, 其主要的逻辑就是在写的时候将 dirty 标记为 1, 在替换的时候如果 dirty 为 1, 那么就需要将其写回内存, dirty 信号的具体判断逻辑和 hit 差不多, 代码如下

```
//检查要替换的是否是dirty的数据, 来决定是否要写回
wire    dirty;
assign dirty =
                (replace_way0 && valid_way0[index] && dirty_way0[index]) ||
                (replace_way1 && valid_way1[index] && dirty_way1[index]) ||
                (replace_way2 && valid_way2[index] && dirty_way2[index]) ||
                (replace_way3 && valid_way3[index] && dirty_way3[index]);
```

其判断的逻辑就是, 我们会遇到未命中, 需要替换的情景, 然后在替换的时候, 我们根据 PLRU 树的状态来判断替换的是哪一路, 然后如果我们将要替换的块是 dirty 的, 那么就需要将其写回内存, 之后再从内存中读取新的数据块。

1.5 在写和读过程完成后, 对于 valid 和 dirty 的更新

每次我们读, 写完一次 cache 之后, 有可能要对 cache 中 valid 和 dirty 的值进行更新, valid 的更新如下

```
//更新valid信号
always @(posedge clk) begin
        if(rst) begin
                valid_way0 <= 8'b0; valid_way1 <= 8'b0;
                valid_way2 <= 8'b0; valid_way3 <= 8'b0;
        end
```

```
        else if (dcache_current_state == REFILL) begin//在REFILL阶段更新
                if(visit_way0)
                        valid_way0[index] <= 1;
                else if(visit_way1)
                        valid_way1[index] <= 1;
                else if(visit_way2)
                        valid_way2[index] <= 1;
                else if(visit_way3)
                        valid_way3[index] <= 1;
        end
end
```

valid 信号比较简单, 就是每次 REFILL 会在某个 cache 块中写入某个数据, 那么就将该块的 valid 标记为 1, 表示该块中有数据

dirty 的更新如下

```
//更新dirty信号
always @(posedge clk) begin
        if(rst) begin
                dirty_way0[7:0] <= 8'b0; dirty_way1[7:0] <= 8'b0;
                dirty_way2[7:0] <= 8'b0; dirty_way3[7:0] <= 8'b0;
        end
        else begin
                //在WR阶段拉高对应的位置的dirty值
                if(dcache_current_state == CACHE_WR) begin
                        if(visit_way0)
                                dirty_way0[index] <= 1;
                        else if(visit_way1)
                                dirty_way1[index] <= 1;
                        else if(visit_way2)
                                dirty_way2[index] <= 1;
                        else if(visit_way3)
                                dirty_way3[index] <= 1;
                end
                //在REFILL阶段, 重置dirty值
                else if(dcache_current_state == REFILL) begin
                        if(visit_way0)
                                dirty_way0[index] <= 0;
                        else if(visit_way1)
                                dirty_way1[index] <= 0;
```

```
                        else if(visit_way2)
                                dirty_way2[index] <= 0;
                        else if(visit_way3)
                                dirty_way3[index] <= 0;
                end
        end
end
```

其实基本的思路差不多, 如果我们写入 cache, 则会进入到 CACHE_WR 的阶段, 那么就将对应的 dirty 标记为 1, 表示该块是 dirty 的, 另外在 REFILL 过程中, 由于我们读出来的数据是未命中后更新的数据, 这时候我们将 dirty 值进行重置, 如果是写, 则之后再 CACHE_WR 阶段拉高 dirty 就可以

1.6 对于 cache 片的写使能信号

这部分的思路比较平凡, 就是根据我们之前所选择的片, 来决定写使能信号, 代码如下

```
//对于各cache片的写使能信号
wire    tag_wen0;
wire    tag_wen1;
wire    tag_wen2;
wire    tag_wen3;

assign  tag_wen0 = (dcache_current_state == REFILL) && replace_way0;
assign  tag_wen1 = (dcache_current_state == REFILL) && replace_way1;
assign  tag_wen2 = (dcache_current_state == REFILL) && replace_way2;
assign  tag_wen3 = (dcache_current_state == REFILL) && replace_way3;

//在dcache中, 写逻辑的时候也更新data, 因为在refill后需要用
//cpu传入的wdata程序来部分改变cache中的内容
assign  data_wen0 = (dcache_current_state == REFILL && replace_way0)
        || (dcache_current_state == CACHE_WR && hit_way0);
assign  data_wen1 = (dcache_current_state == REFILL && replace_way1)
        || (dcache_current_state == CACHE_WR && hit_way1);
assign  data_wen2 = (dcache_current_state == REFILL && replace_way2)
        || (dcache_current_state == CACHE_WR && hit_way2);
assign  data_wen3 = (dcache_current_state == REFILL && replace_way3)
        || (dcache_current_state == CACHE_WR && hit_way3);
```

另外值得一提的是, 我们在整个过程中只需要在 CACHE_WR 和 CACHE_RD 状态读一次 tag 片和 data 片, 因为即使是需要替换, 我们也是在之前的状态将 cache 对应的片中对

应的位置中的数据替换, 替换之后再 CACHE_WR 和 CACHE_RD 状态中将数据拿出来进行更改或者选取

1.7 在和内存交互中, last 等信号的逻辑

在读和写过程中, 我们会和内存产生交互, 在这个过程中我们需要用 last 信号来标记读入的最后一个字节, 其赋值逻辑如下

```
reg     [7:0]   last_string;
always @(posedge clk) begin
        if(dcache_current_state == MEM_WR ||dcache_current_state == BY_WAIT) begin
                if(bypath) begin
                        last_string <= 8'b00000001;
                end
                else begin
                        last_string <= 8'b10000000;
                end
        end
        else if(dcache_current_state == TRANS && from_mem_wr_data_ready) begin
    //在传输的时候,每传输32位右移一位
                last_string <= {1'b0,last_string[7:1]};
        end
end
```

如果不是旁路, 就读/写 8 个字节, 如果是旁路, 则一次 1 个字节

写和读的数据产生的逻辑如下

```
//对于读的数据, 一直读入直到last
always@(posedge clk) begin
        if(rst) begin
                mem_data_raw <= 256'b0;
        end
        else if((dcache_current_state == BY_RD || dcache_current_state==RECV)
                    && from_mem_rd_rsp_valid) begin
                //每次读入四字节, 要让高地址放在高位, 这样的偏移才是正确的
                mem_data_raw <= {from_mem_rd_rsp_data,mem_data_raw[255:32]};
        end
        else begin
                mem_data_raw <= mem_data_raw;
        end
end
```

```
//对于写回的数据
reg [255:0] write_data_raw;
always @(posedge clk) begin
        if(rst) begin
                write_data_raw <= 256'b0;
        end
        else if(dcache_current_state == MEM_WR) begin
                //在dirty写回的时候，将cache_data_raw的数据取出来
                write_data_raw <= cache_data_raw;
        end
        else if(dcache_current_state == TRANS && from_mem_wr_data_ready) begin
                //每次写32位
                write_data_raw <= {32'b0,write_data_raw[255:32]};
        end
        else begin
                write_data_raw <= write_data_raw;
        end
end

assign  to_mem_wr_data_last = last_string[0] &&
from_mem_wr_data_ready && to_mem_wr_data_valid;
```

在这一部分要注意的是小尾端, 在读数据的时候, 每次在高地址加入 32 位的数据, 之后 last 右移, 在写地址的时候, 每次写入低 32 位, 之后数据右移, 另外在写的时候, 需要利用 last_string 来标记最后一个信号

1.8 最终读数据的产生

在进行了之前的替换之后, 我们接下来要产生最红在 RESP 阶段向 cpu 返回的最终的数据, 其相关的赋值逻辑如下

```
//由于cache旁路的内存部分返回的32bit数据，所以只取最新鲜的32位
wire [31:0] bypath_data_out;
assign bypath_data_out  =       mem_data_raw[255:224];
assign cache_data_out   =       cache_data_raw[{offset,3'b0} +: 32];
assign to_cpu_cache_rsp_data =  (bypath)?       bypath_data_out:
                                                        cache_data_out;
```

在这里有两个要点, 首先是旁路的时候只返回 32 位, 所以结合我们之前的读逻辑, 我们只取最新鲜, 也就是最高的 32 位 (因为在读是逐渐向右移的过程), 另外就是对于 cache_data_out 产生的选取, 我们需要根据 offset 来选取, 由于 offset 是 5 位的, 所以我们可以将其左移 3 位,

之后再从 cache_data_raw 中取出 32 位数据, 在本次实验中, 我也学习到了 [{offset,3'b0} +: 32] 这种选取的方式

```
//对于写回内存的部分
wire    [31:0]  cpu_write_data_out;
wire    [7:0]   write_byte_0;
wire    [7:0]   write_byte_1;
wire    [7:0]   write_byte_2;
wire    [7:0]   write_byte_3;

assign  write_byte_0 = (cpu_write_strb[0])? cpu_write_data[7:0]   : cache_data_out[7:0];
assign  write_byte_1 = (cpu_write_strb[1])? cpu_write_data[15:8]  : cache_data_out[15:8];
assign  write_byte_2 = (cpu_write_strb[2])? cpu_write_data[23:16] : cache_data_out[23:16];
assign  write_byte_3 = (cpu_write_strb[3])? cpu_write_data[31:24] : cache_data_out[31:24];

assign  cpu_write_data_out = {write_byte_3,write_byte_2,write_byte_1,write_byte_0};

//根据offset的相对位置来对写回的data进行赋值, offset低两位对齐, 直接取出来高三位
wire    [255:0] cache_data_write;
wire    [2:0]   eff;
assign  eff = offset[4:2];
assign  cache_data_write   =
                (eff==3'b000)?  {cache_data_raw[255:32],cpu_write_data_out}:
                (eff==3'b001)?  {cache_data_raw[255:64],cpu_write_data_out,cache_data_raw[31:0]}:
                (eff==3'b010)?  {cache_data_raw[255:96],cpu_write_data_out,cache_data_raw[63:0]}:
                (eff==3'b011)?  {cache_data_raw[255:128],cpu_write_data_out,cache_data_raw[95:0]}:
                (eff==3'b100)?  {cache_data_raw[255:160],cpu_write_data_out,cache_data_raw[127:0]}:
                (eff==3'b101)?  {cache_data_raw[255:192],cpu_write_data_out,cache_data_raw[159:0]}:
                (eff==3'b110)?  {cache_data_raw[255:224],cpu_write_data_out,cache_data_raw[191:0]}:
                                {cpu_write_data_out,cache_data_raw[223:0]};

wire    [255:0] final_write_data;
assign  final_write_data   =    (dcache_current_state==REFILL)?     mem_data_raw:
                                (dcache_current_state==CACHE_WR)?   cache_data_write://分别是REFILL和CACHE_WR
                                                                    256'b0;
```

有两种写的情况, 一种是在 REFILL 中读出来的数据, 这时候直接写入, 另一种情况就是在 CACHE_WR 中, 这时候需要将 cpu 传入的 wdata 进行写入, 根据 write_strb 来选出更改的字节以及内容再根据 offset 来选出更改的字节

二、 实验过程中遇到的问题、对问题的思考过程及解决方法

dcache 的调试过程对我而言可能是最痛苦的, 由于代码比较复杂并且模块化的程度较低, 导致我在调试的过程中遇到了很多麻烦, 包括一些非常低级的错误, 比如 vscode 的自动联想变量补全把我的一个 rd 信号补全成了一个 wr 信号, 把 visit 补全成了 valid(浪费了我将近 4 个小时), 另外还有一些逻辑上更有价值的问题

2.1 数据旁路部分的 bug

一开始在我的状态机设计中, 我并没有设置 BY_WAIT 这个状态, 也就是从 TAR_GO 直接分别跳转到 BY_RD 和 BY_WR, 但是在测试的时候, 我发现旁路的读和写都无法进行,

一直卡住不动经过调试发现是因为在旁路的读和写过程中, 需要等待内存的响应信号, 而这个状态转移导致从 cache 发射出去的握手信号提前了一个周期, 于是我加入了 BY_WAIT 状态来解决这个问题. 但是之后我忘记更改 last_string 的修改状态, 导致 last_string 又延后了一个周期, 然后很久之后我才发现 (大约 3 个小时)

2.2 一些调试方面的心得

写的时候一定不要着急, 要有耐心, 一点一点写, 避免出现低级的错误, 在调试的过程中, 利用 sim_main 中可以更改的 TIME_LIMIT 宏和限定周期的加速仿真波形来判断出现的问题, 这样问题基本上就能迎刃而解了

三、 性能评估与比较

根据 microbench 中的 9 个测试集, 我们比较有 cache 和没有 cache 的流水线 riscv 处理器的性能, 根据我们的性能计数器, 得到的结果如下

		指令数	周期数	时间(ms)	CPI	CPI加速比	时间加速比
15pz	无cache	5224461	397491252	3989.62	76.08	10.08350191	9.8172199
	有cache	5224461	39419961	406.39	7.55		
bf	无cache	452834	32335500	551.84	71.41	10.52747242	8.6890254
	有cache	452834	3071535	63.51	6.78		
dinic	无cache	16671	1184976	4873.24	71.08	6.655560735	9.820131
	有cache	16671	178043	496.25	10.68		
fib	无cache	2549505	180782438	1823.42	70.91	10.90142747	10.267583
	有cache	2549505	16583373	177.59	6.50		
md5	无cache	4895	346992	161.40	70.89	9.388819741	6.4098491
	有cache	4895	36958	25.18	7.55		
queen	无cache	81470	5776753	70.57	70.91	10.88552655	4.2232196
	有cache	81470	530682	16.71	6.51		
sieve	无cache	10175	721380	20.87	70.90	10.4278817	1.6368627
	有cache	10175	69178	12.75	6.80		
ssort	无cache	619025	43979362	1179.16	71.05	10.80454544	9.8402737
	有cache	619025	4070450	119.83	6.58		
qsort	无cache	9460	670637	753.02	70.89	10.75497145	9.1977525
	有cache	9460	62356	81.87	6.59		
平均CPI加速比		平均时间加速比					
10.04774527		7.766879622					

我们可以看到, 除了一些复杂度比较低的测试集, 其他的测试集都有明显的性能提升, CPI 相比没有 cache 的处理器有了明显的下降, 下降到原本的, 这也说明了 cache 的设计是成功的, 另外在一些复杂度较低的测试集上, 由于 cache 的开销, 反而导致性能下降,

四、 实验所耗时间

在课后，你花费了大约 23 小时完成此次实验。