中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2020K8009970001　姓名：金扬　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 格式保存在~/COD-Lab/reports 目录下。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 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。

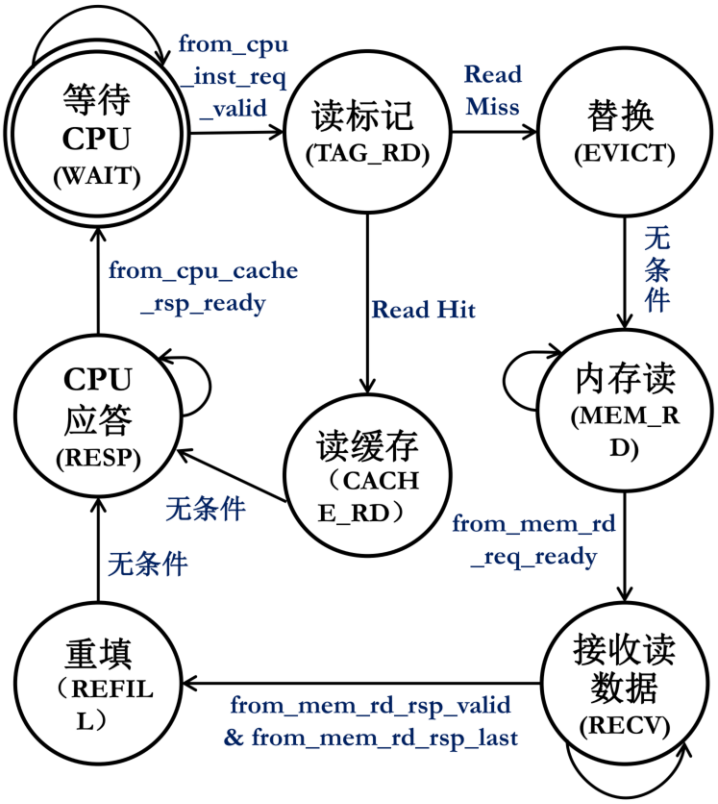
注：本报告中所有代码图片中的≤与≥符号均相当于<=和>=。这是代码编辑器的连体字显示功能，而非真正意义上的≤或≥字符。

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、相应信号的仿真波形和信号变化的说明等）

1. icache（指令 Cache）

a) 状态转移

状态转移图如 PPT 上所示：



b) 请求地址字段划分及信号生成

```
assign {tag, index, offset}     = from_cpu_inst_req_addr;
assign to_cpu_inst_req_ready    = current_state[isWAIT];
assign to_cpu_cache_rsp_valid   = current_state[isRESP];
assign to_mem_rd_req_valid      = current_state[isMEM_RD];
assign to_mem_rd_req_addr       = {tag, index, 5'b00000};
assign to_mem_rd_rsp_ready      = current_state[isRECV] || rst;
```

c) tag 和 data 数组的创建及 ReadHit 与 ReadMiss 的判定与处理

```
genvar i_way;
generate
    for (i_way=0;i_way<`CACHE_WAY;i_way=i_way+1) begin : ways
        tag_array tags(
            .clk    (clk),
            .waddr  (index),
            .raddr  (index),
            .wen    (wen[i_way]),
            .wdata  (tag),
            .rdata  (tag_rdata[i_way])
        );
        assign tag_hit[i_way] = valid_array[i_way][index] && (tag_rdata[i_way] == tag);
        data_array datas(
            .clk    (clk),
            .waddr  (index),
            .raddr  (index),
            .wen    (wen[i_way]),
            .wdata  ({
                data_recv[7],data_recv[6],data_recv[5],data_recv[4],
                data_recv[3],data_recv[2],data_recv[1],data_recv[0]
            }),
            .rdata  (data_rdata[i_way])
        );
    end
endgenerate
```

这里使用了 generate-for 语句来简化代码。

tag_hit 线路用来表示各 way 中 index 地址下的 tag 值是否与输入地址的 tag 域匹配。若四个 way 中存在一个 way 匹配，则 ReadHit，反之 ReadMiss。

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

assign way_selected =   {(2){tag_hit[0]}} & 2'b00 |
                        {(2){tag_hit[1]}} & 2'b01 |
                        {(2){tag_hit[2]}} & 2'b10 |
                        {(2){tag_hit[3]}} & 2'b11 ;
assign data_selected = data_rdata[way_selected];
```

当 ReadHit 时，way_selected 用来表示 tag 匹配的那一个 way。data_selected 用来表示此时输入地址对应的 cache 中缓存的数据值。

为了生成向 cpu 回复的数据，还需要将 data_selected 偏移一段，并截取低位数据。

```
assign to_cpu_cache_rsp_data = data_selected >> {offset[4:0], 3'b000};
```

d) LRU_cnt 计数器

我采用了 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。

```
integer i;
always @(posedge clk) begin
    if (rst) begin
        for (i=0;i<8;i=i+1) begin
            LRU_cnt[0][i] <= 0;
            LRU_cnt[1][i] <= 0;
            LRU_cnt[2][i] <= 0;
            LRU_cnt[3][i] <= 0;
        end
    end else if (!rst && current_state[isRESP] && next_state[isWAIT]) begin
        if (way_selected==2'b00) LRU_cnt[0][index] <= LRU_cnt[0][index] + 1;
        if (way_selected==2'b01) LRU_cnt[1][index] <= LRU_cnt[1][index] + 1;
        if (way_selected==2'b10) LRU_cnt[2][index] <= LRU_cnt[2][index] + 1;
        if (way_selected==2'b11) LRU_cnt[3][index] <= LRU_cnt[3][index] + 1;
    end else if (!rst && current_state[isREFILL]) begin
        LRU_cnt[way_LRU[index]][index] <= 4'b0000;
    end
end
```

替换时，选取当前计数器值最大的 cache 块替换即可。

```
genvar i_set;
generate
    for (i_set=0;i_set<`CACHE_SET;i_set=i_set+1) begin : sets
        assign way_LRU[i_set] =
            {(2){~valid_array[0][i_set]}} & 2'b00 |
            {(2){ valid_array[0][i_set] & ~valid_array[1][i_set]}} & 2'b01 |
            {(2){ valid_array[0][i_set] &  valid_array[1][i_set] & ~valid_array[2][i_set]}} & 2'b10 |
            {(2){ valid_array[0][i_set] &  valid_array[1][i_set] &  valid_array[2][i_set] & ~valid_array[3][i_set]}} & 2'b11 |
            {(2){ valid_array[0][i_set] &  valid_array[1][i_set] &  valid_array[2][i_set] &  valid_array[3][i_set]}} & (
                LRU_cnt[0][i_set] >= LRU_cnt[1][i_set] ?
                LRU_cnt[0][i_set] >= LRU_cnt[2][i_set] ?
                LRU_cnt[0][i_set] >= LRU_cnt[3][i_set] ? 2'b00 : 2'b11 :
                LRU_cnt[2][i_set] >= LRU_cnt[3][i_set] ? 2'b10 : 2'b11 :
                LRU_cnt[1][i_set] >= LRU_cnt[2][i_set] ?
                LRU_cnt[1][i_set] >= LRU_cnt[3][i_set] ? 2'b01 : 2'b11 :
                LRU_cnt[2][i_set] >= LRU_cnt[3][i_set] ? 2'b10 : 2'b11
            );
    end
endgenerate
```

e) valid 数组的更新逻辑以及对内存数据的接受逻辑

```
integer j;
always @(posedge clk) begin
    if (rst) begin
        for (j=0;j<8;j=j+1) begin
            valid_array[0][j] <= 0;
            valid_array[1][j] <= 0;
            valid_array[2][j] <= 0;
            valid_array[3][j] <= 0;
        end
    end else if (!rst && current_state[isEVICT]) begin
        valid_array[way_LRU[index]][index]<=1'b0;
    end else if (!rst && current_state[isMEM_RD]) begin
        len <= 3'b000;
    end else if (!rst && current_state[isRECV] && from_mem_rd_rsp_valid) begin
        data_recv[len] <= from_mem_rd_rsp_data;
        len <= len + 1;
    end else if (!rst && current_state[isREFILL]) begin
        valid_array[way_LRU[index]][index] <= 1'b1;
    end
end

assign wen[0] = current_state[isREFILL] && way_LRU[index] == 2'b00;
assign wen[1] = current_state[isREFILL] && way_LRU[index] == 2'b01;
assign wen[2] = current_state[isREFILL] && way_LRU[index] == 2'b10;
assign wen[3] = current_state[isREFILL] && way_LRU[index] == 2'b11;
```

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。

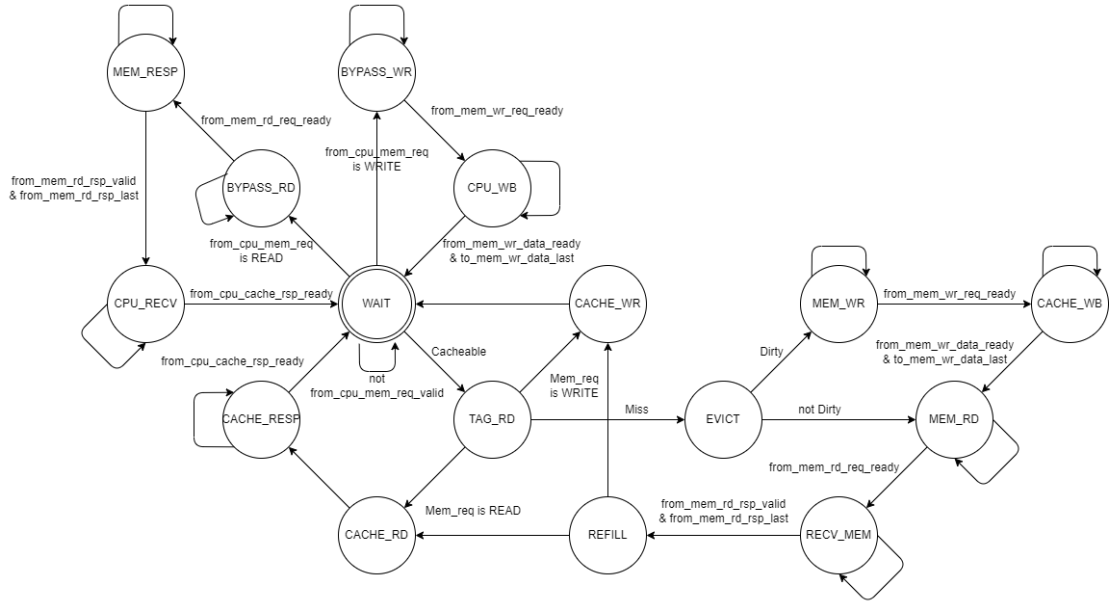
```
tag_array tags(
    .clk    (clk),
    .waddr  (index),
    .raddr  (index),
    .wen    (wen[i_way]),
    .wdata  (tag),
    .rdata  (tag_rdata[i_way])
);
```

```
data_array datas(
    .clk    (clk),
    .waddr  (index),
    .raddr  (index),
    .wen    (wen[i_way]),
    .wdata  ({
        data_recv[7],data_recv[6],data_recv[5],data_recv[4],
        data_recv[3],data_recv[2],data_recv[1],data_recv[0]
    }),
    .rdata  (data_rdata[i_way])
);
```

2. dcache（数据 Cache）

a) 状态转移

一共设计了十六个状态，其中后五个状态在旁路访问使用。状态转移图如下：



各状态具体功能见后一页的表。

这十六个状态分别为：

编号	状态名	描述
0	WAIT	等待 CPU 访存请求 缓冲请求地址、请求类型（读/写）、数据、选通信号
1	TAG_RD	判断当前 TAG 对应块是否已被缓存
2	EVICT	根据 LRU 算法选出待替换的缓存块 判断该块是否需要写回内存
3	MEM_WR	向内存发送写请求
4	CACHE_WB	利用 Burst 传输，将待替换的缓存块写回内存
5	MEM_RD	向内存发送读请求
6	RECV_MEM	利用 Burst 传输，从内存接收新缓存块的数据
7	REFILL	将收到的数据填入对应缓存块 重置 LRU 计数器， valid、dirty、tag 标记
8	CACHE_WR	将来自 CPU 的修改写入对应缓存块
9	CACHE_RD	从对应缓存块读取数据
10	CACHE_RESP	将读取到的数据返还 CPU，更新 LRU 计数器
以下为旁路读写状态		
11	BYPASS_RD	向内存发送读请求
12	MEM_RESP	接受内存回复的数据
13	CPU_RECV	向 CPU 发送从内存送来的数据
14	BYPASS_WR	向内存发送写请求
15	CPU_WB	向内存发送先前缓冲的来自 CPU 的数据

b) 各信号的生成

较多地使用了数据选择逻辑。

```
assign to_cpu_mem_req_ready     = current_state[isWAIT];
assign to_cpu_cache_rsp_valid   = current_state[isCACHE_RESP] || current_state[isCPU_RECV];
assign to_cpu_cache_rsp_data    = {(32){current_state[isCACHE_RESP]}}   & data_selected >> {offset[4:0], 3'b000}
                                | {(32){current_state[isCPU_RECV]}}     & data_bypass;
assign to_mem_rd_req_valid      = current_state[isMEM_RD] || current_state[isBYPASS_RD];
assign to_mem_rd_req_addr       = {(32){current_state[isMEM_RD]}}    & {tag, index, 5'b00000}
                                | {(32){current_state[isBYPASS_RD]}} & Address;
assign to_mem_rd_req_len        = {( 8){current_state[isMEM_RD]}}    & 7
                                | {( 8){current_state[isBYPASS_RD]}} & 0;
assign to_mem_rd_rsp_ready      = current_state[isRECV_MEM] || current_state[isMEM_RESP] || rst;
assign to_mem_wr_req_valid      = current_state[isMEM_WR] || current_state[isBYPASS_WR];
assign to_mem_wr_req_addr       = {(32){current_state[isMEM_WR]}}    & {tag_rdata[way_LRU[index]], index, 5'b00000}
                                | {(32){current_state[isBYPASS_WR]}} & Address;
assign to_mem_wr_req_len        = {( 8){current_state[isMEM_WR]}}    & 7
                                | {( 8){current_state[isBYPASS_WR]}} & 0;
assign to_mem_wr_data_valid     = current_state[isCACHE_WB] || current_state[isCPU_WB];
assign to_mem_wr_data           = {(32){current_state[isCACHE_WB]}} & data_wb
                                | {(32){current_state[isCPU_WB]}}   & Write_data;
assign to_mem_wr_data_strb      = {( 4){current_state[isCACHE_WB]}}
                                | {( 4){current_state[isCPU_WB]}}   & Write_strb;
assign to_mem_wr_data_last      = current_state[isCACHE_WB] && len == 3'b000
                                || current_state[isCPU_WB];

assign {tag, index, offset} = Address;

assign Cacheable    = (|from_cpu_mem_req_addr[31:5]) && !from_cpu_mem_req_addr[31] && !from_cpu_mem_req_addr[30];
assign Miss         = !(tag_hit[0] || tag_hit[1] || tag_hit[2] || tag_hit[3]);
assign Dirty        = valid_array[way_LRU[index]][index] && dirty_array[way_LRU[index]][index];
```

其中 Cacheable 信号通过判断输入地址是否落在特定范围实现。Miss 信号和前文一样，通过判断 tag_hit 线路实现。Dirty 信号通过读取 valid 数组和 dirty 数组实现。值得注意的是，只有当前 cache 块是有效时，才需要写回，因此 dirty 才有效。

c) wdata 和 wen 连线

相较 icache，dcache 除了从内存能写数据到 cache 外，cpu 也能通过 write_data，write_strb 修改 cache 中的值。因此对于 data 寄存器的 wdata 和 wen 接口，在接入前也需要对数据进行选择。

```
data_array datas(
    .clk    (clk),
    .waddr  (index),
    .raddr  (index),
    .wen    (wen[i_way]),
    .wdata  ({(256){current_state[isREFILL]}} & {
        data_recv[7],data_recv[6],data_recv[5],data_recv[4],
        data_recv[3],data_recv[2],data_recv[1],data_recv[0]
    } | {(256){current_state[isCACHE_WR]}} & (
        data_rdata[i_way] & ~({
            224'b0,
            {(8){Write_strb[3]}},{(8){Write_strb[2]}},{(8){Write_strb[1]}},{(8){Write_strb[0]}}
        }<<{offset,3'b000}) | ({
            224'b0,
            Write_data & {{(8){Write_strb[3]}},{(8){Write_strb[2]}},{(8){Write_strb[1]}},{(8){Write_strb[0]}}}
        }<<{offset,3'b000})
    )),
    .rdata  (data_rdata[i_way])
);
```

```
assign wen[0] = current_state[isREFILL] && way_LRU[index] == 2'b00 ||
                current_state[isCACHE_WR] && way_selected == 2'b00;
assign wen[1] = current_state[isREFILL] && way_LRU[index] == 2'b01 ||
                current_state[isCACHE_WR] && way_selected == 2'b01;
assign wen[2] = current_state[isREFILL] && way_LRU[index] == 2'b10 ||
                current_state[isCACHE_WR] && way_selected == 2'b10;
assign wen[3] = current_state[isREFILL] && way_LRU[index] == 2'b11 ||
                current_state[isCACHE_WR] && way_selected == 2'b11;
```

d) LRU 计数器

逻辑与前文类似，不再赘述

e) 请求数据缓冲

注意到 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，因此需要对请求数据进行缓冲。

```
always @(posedge clk) begin
    if (current_state[isWAIT]) begin
        Address     <= from_cpu_mem_req_addr;
        Write_data  <= from_cpu_mem_req_wdata;
        Write_strb  <= from_cpu_mem_req_wstrb;
        Mem_req     <= from_cpu_mem_req;
    end
end
```

f) 主时序逻辑

```
integer j;
always @(posedge clk) begin
    if (rst) begin
        for (j=0;j<8;j=j+1) begin
            valid_array[0][j] <= 0;
            valid_array[1][j] <= 0;
            valid_array[2][j] <= 0;
            valid_array[3][j] <= 0;
            LRU_cnt[0][j] <= 0;
            LRU_cnt[1][j] <= 0;
            LRU_cnt[2][j] <= 0;
            LRU_cnt[3][j] <= 0;
        end
    end else if (!rst && current_state[isEVICT]) begin
        valid_array[way_LRU[index]][index] <= 1'b0; //keep zero till finish clean
    //end else if (next_state[isMEM_WR]) begin //for mutually excluded if
        len <= 3'b000; //initialize the counter `len`
    end else if (!rst && !current_state[isCACHE_WB] && next_state[isCACHE_WB]) begin
        data_wb <= data_rdata[way_LRU[index]] >> {len, 5'b00000};
        len <= len + 1;
    end else if (!rst && current_state[isCACHE_WB] && from_mem_wr_data_ready) begin
        data_wb <= data_rdata[way_LRU[index]] >> {len, 5'b00000};
        len <= len + 1;
    end else if (!rst && current_state[isMEM_RD]) begin
        len <= 3'b000;
    end else if (!rst && current_state[isRECV_MEM] && from_mem_rd_rsp_valid) begin
        data_recv[len] <= from_mem_rd_rsp_data;
        len <= len + 1;
    end else if (!rst && current_state[isREFILL]) begin
        LRU_cnt[way_LRU[index]][index] <= 4'b0000;
        valid_array[way_LRU[index]][index] <= 1'b1; //finish clean
        dirty_array[way_LRU[index]][index] <= 1'b0;
    end else if (!rst && current_state[isCACHE_RESP] && next_state[isWAIT]) begin
        if (way_selected==2'b00) LRU_cnt[0][index] <= LRU_cnt[0][index] + 1;
        if (way_selected==2'b01) LRU_cnt[1][index] <= LRU_cnt[1][index] + 1;
        if (way_selected==2'b10) LRU_cnt[2][index] <= LRU_cnt[2][index] + 1;
        if (way_selected==2'b11) LRU_cnt[3][index] <= LRU_cnt[3][index] + 1;
    end else if (!rst && current_state[isCACHE_WR]) begin
        dirty_array[way_selected][index] <= 1'b1;
    end else if (!rst && current_state[isMEM_RESP] && from_mem_rd_rsp_valid) begin
        data_bypass <= from_mem_rd_rsp_data;
    end
end
```

3. custom_cpu.v 中 inst_retire 连线

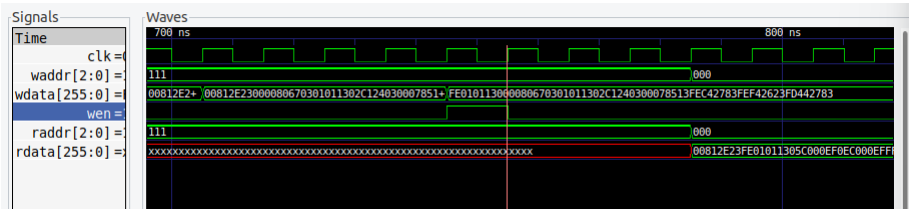
```
assign inst_retire [69:69] = RF_wen;
assign inst_retire [68:64] = RF_waddr;
assign inst_retire [63:32] = RF_wdata;
assign inst_retire [31: 0] = PC_normal;
```

其中 PC_normal 在 ID 状态更新，取 PC 在新一轮更新前的值。

```
always @(posedge clk) begin
    if (current_state[isID]) begin
        PC_normal <= PC_reg;
    end
end
```

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．cache 寄存器未正常写入



如图，这里写地址 waddr 为 111，写入数据 wdata 有效，wen 为高电平，理应在红线所示时刻触发，将数据写入。但是，通过观察读地址 raddr 对应的数据 rdata 可知，该数据并没有正常写入。

出错原因是老师之前提供的 data_array.v 和 tag_array.v 中有些小 bug。

```
1   `timescale 10 ns / 1 ns
2
3   `define TARRAY_DATA_WIDTH 24
4   `define TARRAY_ADDR_WIDTH 3
5
6   module tag_array(
7       input                           clk,
8       input  [`TARRAY_ADDR_WIDTH - 1:0] waddr,
9       input  [`TARRAY_ADDR_WIDTH - 1:0] raddr,
10      input                           wen,
11      input  [`TARRAY_DATA_WIDTH - 1:0] wdata,
12      output [`TARRAY_DATA_WIDTH - 1:0] rdata
13  );
14
15      reg [`TARRAY_DATA_WIDTH-1:0] array[ 1 << `TARRAY_ADDR_WIDTH - 1 : 0];
16
17      always @(posedge clk)
18      begin
19          if(wen)
20              array[waddr] <= wdata;
21      end
22
23  assign rdata = array[raddr];
24
25  endmodule
```

左移优先级低于减法

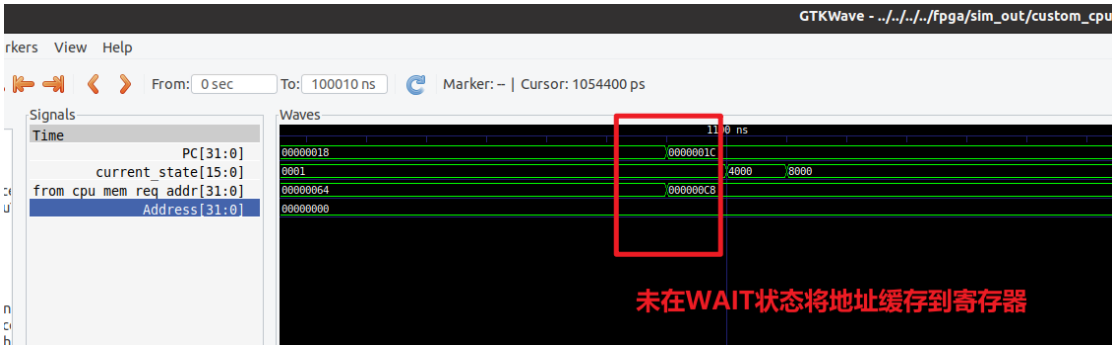
这里在定义 array 数组的长度时，使用了如红线所示的运算。但事实上，因为左移的优先级低于减法，这个操作会先执行减一运算，再执行左移运算。导致数组大小的定义出了问题。

解决方案是加个括号。

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

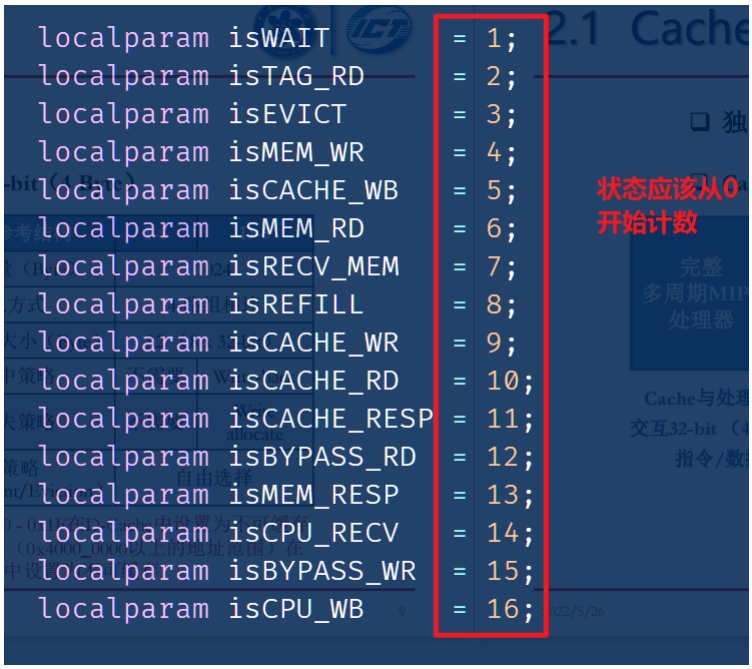
```
reg [`DARRAY_DATA_WIDTH-1:0] array[ (1 << `DARRAY_ADDR_WIDTH) - 1 : 0];
```

2. 状态下标定义出错



如图，这里的 from_cpu_mem_req_addr 应该在 WAIT 状态结束时，缓冲到 Address 寄存器。但是波形显示并未成功缓冲。

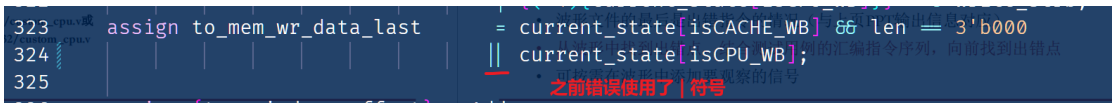
导致这个问题的原因在于状态下标定义出了失误。



如图，因为状态的最低位为第 0 位，状态应该从 0 开始计数。

3. to_mem_wr_data_last 信号未正常拉高

如图所示，在向内存传输最后一个数据时，to_mem_wr_data_last 未正常拉高。其原因如下图所示：



在生成该信号时，之前错误使用了|符号而非||符号。虽然对于 1bit 数据两者功能类似，但是在这里仍然只能使用||符号。

这是因为，该符号前的语句为 len == 3’b000。如果使用|符号，由于|符号优先级高于==符号，代码将被解释为 current_state[isCACHE_WB] && len == (3’b000 | current_state[isCPU_WB])，与期望功能不符。

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。

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

在第 0 个数据 valid 时，拉高 last 信号。

3. 性能分析与优化

在最初的设计（**Pipeline #74206**）中，对于 Dhystone 测试，五次 Dhrystones per second 值分别为：123，120，118，116，113。

其平均值为 118，高于 PPT 上所有的性能评估结果样例。

对于 Coremark 测试，五次 Total us 以及 Iterations 值分别为

Total us	Iterations	Iterations/Sec
1640400	5	3.048
2477274	5	2.018
3314148	5	1.509
4151023	5	1.205
4987898	5	1.002

其平均值为 1.756，与 PPT 上样例的最高值 1.843 相近。

在编写实验报告和绘制状态转移图的过程中，我发现我的状态仍然存在简化空间。我尝试将 CACHE_RD 和 CACHE_RESP 合为一个状态。但是由于 FPGA_RUN 一步始终 failed，无法得到具体的测评结果。failed 原因至今未知，猜测是因为评测板卡不太稳定。（我将曾经 passed 的点重新 retry 一遍，也得到了 failed 的结果）

我也曾查看延时最长路径的组合逻辑情况，发现造成延时的首要原因是 custom_cpu.v 中实现的 MULResult 线路。这条线路通过对两数相乘得到，乘法运算的复杂度较高，造成了高延时。若要进一步优化，可以考虑将乘法指令的执行过程拆分为多周期完成，从而降低每个周期的耗时。

四、 在课后，你花费了大约 30 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）

这次实验的 icache 部分原理简单，由于资料比较完善，易于实现。dcache 部分相较 icache 部分复杂一些，但是由于之前实现过 icache，加上 dcache 部分自由空间较大，因此相关设计和实现并不算非常复杂。

另外由于 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。所幸由于本地仿真工具较为便捷，该错误不难发现、纠正。

非常感谢刘子扬同学告诉我本地仿真工具的使用方法，让我大大加快了调试速度。希望老师能升级云平台相关硬件设施，减少硬件故障。