中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2020K8009926006 姓名：游昆霖 专业：计算机科学与技术

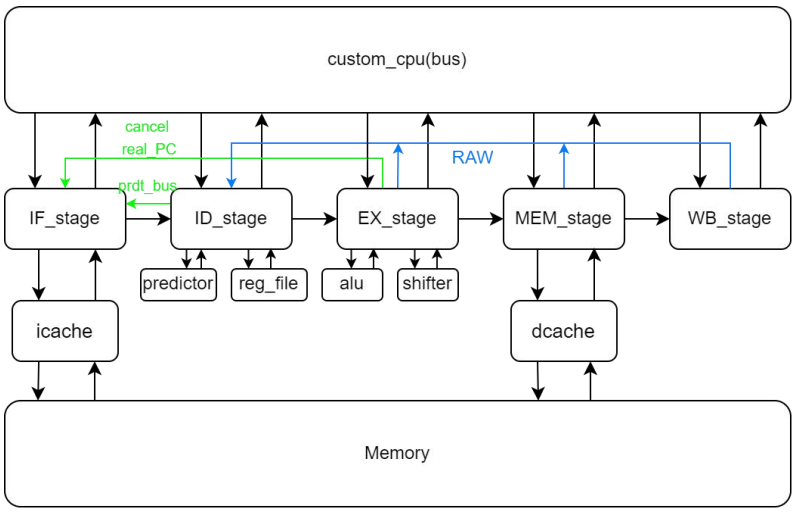
实验序号：5.4 实验名称：高速缓存（cache）设计

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。

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）

*说明：由于 icache 和 dcache 大部分代码类似，且 dcache 还需要处理写缺失和写命中问题，复杂程度较高。因此将只介绍 dcache 代码，icache 所有功能在 dcache 中均有实现，不再赘述。

整体数据流向示意图（基于 turbo）：



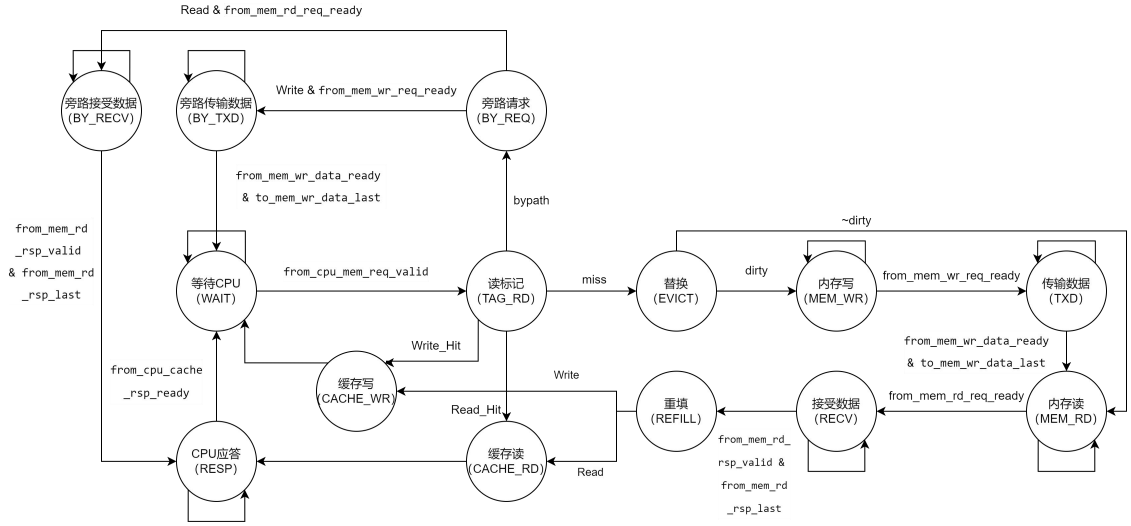
1、dcache 状态状态转移

```
localparam      WAIT    = 14'b00000000000001 ,
                TAG_RD  = 14'b00000000000010 ,
                EVICT   = 14'b00000000000100 ,
                MEM_WR  = 14'b00000000001000 ,
                TXD     = 14'b00000000010000 ,  //transmit data
                MEM_RD  = 14'b00000000100000 ,
                RECV    = 14'b00000001000000 ,
                REFILL  = 14'b00000010000000 ,
                CACHE_WR= 14'b00000100000000 ,
                CACHE_RD= 14'b00001000000000 ,
                RESP    = 14'b00010000000000 ,
                BY_REQ  = 14'b00100000000000 ,
                BY_TXD  = 14'b01000000000000 ,
                BY_RECV = 14'b10000000000000 ;

...

always @(*) begin
        case (cur_state)
                WAIT: begin
                        if(from_cpu_mem_req_valid)
                                next_state = TAG_RD;
                        else
                                next_state = WAIT;
                end

                TAG_RD: begin
                        if(bypath)
                                next_state = BY_REQ;
                        else if(cpu_mem_rw & Hit)  //write hit
                                next_state = CACHE_WR;
                        else if(~cpu_mem_rw & Hit) //read hit
                                next_state = CACHE_RD;
                        else
                                next_state = EVICT;
                end

                EVICT: begin
                        if(dirty)
                                next_state = MEM_WR;
                        else
                                next_state = MEM_RD;
                end
```

```
MEM_WR: begin
        if(from_mem_wr_req_ready)
                next_state = TXD;
        else
                next_state = MEM_WR;
end

TXD: begin
        if(from_mem_wr_data_ready & to_mem_wr_data_last)
                next_state = MEM_RD;
        else
                next_state = TXD;
end

MEM_RD: begin
        if(from_mem_rd_req_ready)
                next_state = RECV;
        else
                next_state = MEM_RD;
end

RECV: begin
        if(from_mem_rd_rsp_valid & from_mem_rd_rsp_last)
                next_state = REFILL;
        else
                next_state = RECV;
end

REFILL: begin
        if(cpu_mem_rw) //write
                next_state = CACHE_WR;
        else
                next_state = CACHE_RD;  //save a cycle than CACHE_RD
end

CACHE_WR: begin
        next_state = WAIT;
end

CACHE_RD: begin
        next_state = RESP;
end
```

```
                RESP: begin
                        if(from_cpu_cache_rsp_ready)
                                next_state = WAIT;
                        else
                                next_state = RESP;
                end

                BY_REQ: begin
                        if(cpu_mem_rw & from_mem_wr_req_ready)  //write
                                next_state = BY_TXD;
                        else if(~cpu_mem_rw & from_mem_rd_req_ready) //read
                                next_state = BY_RECV;
                        else
                                next_state = BY_REQ;
                end

                BY_TXD: begin
                        if(from_mem_wr_data_ready & to_mem_wr_data_last)
                                next_state = WAIT;
                        else
                                next_state = BY_TXD;
                end

                BY_RECV: begin
                        if(from_mem_rd_rsp_valid & from_mem_rd_rsp_last)
                                next_state = RESP;
                        else
                                next_state = BY_RECV;
                end

                default:
                        next_state = WAIT;
        endcase
end
```

状态转移图：



Cache 在接受到 CPU 的请求后，先进入 TAG_RD 状态，此处将分辨是否旁路请求并进入不同路径。

对于非旁路请求，如果命中将根据读或写分为 CACHE_RD 和 CACHE_WR 状态，分别进行 CACHE 中数据的读和写。如果未命中，则需进入 EVICT 替换状态。此时需要进一步根据 dirty 信号判断该替换块是否已被写过，若是，则需先后经历 MEM_WR 请求态和 TXD 传输数据态。然后，与替换块未写过的情形相同，先后进入 MEM_RD 请求态和 RECV 接受态，接受来自内存的替换数据，并进入 REFILL 状态进行数据重填。然后和命中情形一致，再进入 CACHE_RD 和 CACHE_WR 状态进行读写。

对于 IO 旁路请求，将从 TAG_RD 状态直接进入 BY_REQ 请求态，然后根据读写类型分别进入 BY_RECV 旁路接收状态和 BY_TXD 旁路传输状态。

对于 CACHE_WR 和 BY_TXD，当写缓存或是写内存完毕后可以直接回到初始态等待下一次使用。对于 CACHE_RD 和 BY_RECV，当读缓存或者读内存完毕后，需要额外经过 RESP 响应态，等待 CPU 准备好接收数据。

2、地址分析与路径确定

```
//Analyse mem addr, compare and judge path
        ...//相应数据初始化
        assign tag     = cpu_mem_addr[31:32-`TAG_LEN];
        assign set      = cpu_mem_addr[4 + `CACHE_SET_WD:5];
        assign offset   = cpu_mem_addr[4:0];

        //0x00 ~ 0x1F OR above 0x40000000
        assign bypath   = (~|cpu_mem_addr[31:5]) | (|cpu_mem_addr[31:30]);

        assign hit0     = valid0[set] & Tag0 == tag;
        assign hit1     = valid1[set] & Tag1 == tag;
        assign hit2     = valid2[set] & Tag2 == tag;
        assign hit3     = valid3[set] & Tag3 == tag;

        assign Hit      = hit0 | hit1 | hit2 | hit3;
```

根据不可缓存地址确定 bypath 旁路标志。当某组路的某块数据有效且 tag 相等时表示该路命中。为适配后续修改 cache 容量，针对可能的修改使用宏定义定义位宽。

3、替换策略

```
//EVICT : use PLRU algorithm
        //when some way are invalid, replace them first by order
        //only when all 4 way are valid, use PLRU to choose
        assign valid =  valid0[set] & valid1[set] & valid2[set] & valid3[set];
        assign way0 =   (valid & ~PLRU[set][1] & ~PLRU[set][0]) |
                        (~valid0[set]);
        assign way1 =   (valid &  PLRU[set][1] & ~PLRU[set][0]) |
                        (valid0[set] & ~valid1[set]);
        assign way2 =   (valid & ~PLRU[set][2] &  PLRU[set][0]) |
                        (valid0[set] & valid1[set] & ~valid2[set]);
        assign way3 =   (valid &  PLRU[set][2] &  PLRU[set][0]) |
                        (valid0[set] & valid1[set] & valid2[set] & ~valid3[set]);

        //refresh PLRU
        assign choose01 = choose0 | choose1;
        always @(posedge clk) begin
                if(rst) begin
```

```
                        PLRU[0] <= 3'b0; PLRU[1] <= 3'b0; PLRU[2] <= 3'b0; PLRU[3] <= 3'b0;
                        PLRU[4] <= 3'b0; PLRU[5] <= 3'b0; PLRU[6] <= 3'b0; PLRU[7] <= 3'b0;
                end
                else if (cur_state == CACHE_RD | cur_state == CACHE_WR) begin //after read or write
visit
                        PLRU[set][0] <= choose01;
                        if(choose01)
                                PLRU[set][1] <= choose0;
                        else
                                PLRU[set][2] <= choose2;
                end
        end
```

表示	L2	L1	L0
Way 0	x	1	1
Way 1	x	0	1
Way 2	1	x	0
Way 3	0	x	0
替换			
Way 0	x	0	0
Way 1	x	1	0
Way 2	0	x	1
Way 3	1	x	1

当有某些组路数据无效时，先按顺序选择无效组路进行更新，如果 4 组路均有效时，根据 PLRU 算法进行更新，具体规则如上图所示，其中下半部分为替换条件，上半部分为替换或是读写命中时的更新表示。通过 PLRU 算法，可以使得替换倾向于选择较少被访问的组路，从而增加 cache 读写命中率，减少访存次数。

4、读写路组选择

```
//choose logic :
        //hit : get cache_data from which way
        //miss: get refresh which way
```

```
        //note the init is all 0 because of the valid init
        always @(posedge clk) begin
                if(cur_state == TAG_RD) // if hit, get cache data way
                begin
                        Hit_tmp <= Hit;
                        choose0 <= hit0;
                        choose1 <= hit1;
                        choose2 <= hit2;
                        choose3 <= hit3;
                end
                if(cur_state == EVICT) //if miss, refresh which way, op before will be ignore
                begin
                        choose0 <= way0;
                        choose1 <= way1;
                        choose2 <= way2;
                        choose3 <= way3;
                end
        end
```

当命中时，根据命中路组进行选择；当缺失时，根据替换数据进行选择。

5、各路组相关数组

```
//valid / tag / data / dirty array: initialization or refresh
        //valid array
        always @(posedge clk) begin
                if(rst) begin
                        valid0[`CACHE_SET-1:0] <= {`CACHE_SET{1'b0}};
                        valid1[`CACHE_SET-1:0] <= {`CACHE_SET{1'b0}};
                        valid2[`CACHE_SET-1:0] <= {`CACHE_SET{1'b0}};
                        valid3[`CACHE_SET-1:0] <= {`CACHE_SET{1'b0}};
                end
                if(cur_state == REFILL) begin
                        if(choose0)             valid0[set] <= 1'b1;
                        else if(choose1)        valid1[set] <= 1'b1;
                        else if(choose2)        valid2[set] <= 1'b1;
                        else if(choose3)        valid3[set] <= 1'b1;
                end
        end
        //tag array : when rst, no need to refresh because valid = 0
                //wdata is tag
        assign TagWen0 = cur_state[7] & choose0; //REFILL
        assign TagWen1 = cur_state[7] & choose1;
```

```
    assign TagWen2 = cur_state[7] & choose2;
    assign TagWen3 = cur_state[7] & choose3;

    //data array
    //读写 REFILL 均需更新，因为 CACHE 写只会进行部分更新
    assign DataWen0 = (cur_state[7] | cur_state[8]) & choose0; //REFILL or CACHE_WR
    assign DataWen1 = (cur_state[7] | cur_state[8]) & choose1;
    assign DataWen2 = (cur_state[7] | cur_state[8]) & choose2;
    assign DataWen3 = (cur_state[7] | cur_state[8]) & choose3;
     assign Array_Wdata =    {`LINE_LEN{cur_state[7]}} & mem_block_data | //REFILL
                            {`LINE_LEN{cur_state[8]}} & cache_modified_block ;

    //dirty array
    always @(posedge clk) begin
            if (rst) begin
                    dirty0[`CACHE_SET-1:0] <= {`CACHE_SET{1'b0}};
                    dirty1[`CACHE_SET-1:0] <= {`CACHE_SET{1'b0}};
                    dirty2[`CACHE_SET-1:0] <= {`CACHE_SET{1'b0}};
                    dirty3[`CACHE_SET-1:0] <= {`CACHE_SET{1'b0}};
            end
            else if (cur_state == CACHE_WR) begin
                    if (choose0)            dirty0[set] <= 1'b1;
                    else if (choose1)       dirty1[set] <= 1'b1;
                    else if (choose2)       dirty2[set] <= 1'b1;
                    else if (choose3)       dirty3[set] <= 1'b1;
            end
            else if (cur_state == REFILL) begin //refill cache with mem data, reset dirty
                    if (choose0)            dirty0[set] <= 1'b0;
                    else if (choose1)       dirty1[set] <= 1'b0;
                    else if (choose2)       dirty2[set] <= 1'b0;
                    else if (choose3)       dirty3[set] <= 1'b0;
            end
    end
```

初始化时只需初始化 valid 和 dirty 数据即可让其余两个数组无效化。在 REFILL 重填阶段需要根据替换选择对 4 个数组都进行更新。CACHE_WR 在向 CACHE 写数据阶段需要根据选择更新对应的 dirty 和 data 数组，data 数组需要将 cache 块拿出，根据 offset 和 strb 修改对应位置，在重新将修改后的块填入 data 数组。

6、CPU 读数据相关

```
//READ : final data to cpu  -- source : cache(hit)/mem(miss)/bypath
        //data from cache
        assign cache_block_data =       ( {`LINE_LEN{choose0}} & Data0 ) |
                                        ( {`LINE_LEN{choose1}} & Data1 ) |
                                        ( {`LINE_LEN{choose2}} & Data2 ) |
                                        ( {`LINE_LEN{choose3}} & Data3 ) ;
        assign cache_final_data = cache_block_data[ {offset,3'b0} +: 32 ];

        //data from mem / bypath
        assign to_mem_rd_req_addr[31:5] = cpu_mem_addr[31:5];
        assign to_mem_rd_req_addr[4:0]  = bypath ? cpu_mem_addr[4:0] : 5'b0;

        assign to_mem_rd_req_len = {5'b0 , {3{~bypath}} }; //bypath 0 other 7

        always @(posedge clk) begin
                if((cur_state == RECV | cur_state == BY_RECV) & from_mem_rd_rsp_valid)
                        mem_block_data <= {from_mem_rd_rsp_data , mem_block_data[255:32]};
        end     //result : {data7,...,data0} or {data,x,..,x}

        assign mem_final_data = mem_block_data[ {offset,3'b0} +: 32 ];
        assign bypath_read_data = mem_block_data [255:224];

        //choose source
        assign to_cpu_cache_rsp_data =  ( {32{ bypath}} & bypath_read_data)     |
                                        ( {32{~bypath}} & cache_final_data)     ;
```

根据路组选择可以从 data 数组中取得数据块，然后根据 offset 可以确定 cache 所读的 4byte 数据。读写缺失时，需要根据替换算法从内存取得数据块。根据 bypath 旁路标志确定请求地址和长度，然后在对应状态通过 mem_block_data 寄存器移位接收内存传递的数据，此时由于旁路只传输一拍，对应的数据即最高 32 位。最后根据旁路标志选择 CPU 读响应数据的来源。

7、CPU 写数据相关

```
//WRITE : final data from cpu --target : cache(hit or miss) / mem(bypath)
                //note write back to MEM is dirty block or bypath
        //cpu write
```

```
        assign cache_modified_final = {
                                        { ({8{cpu_mem_wstrb[3]}} & cpu_mem_wdata[31:24]) |
({8{~cpu_mem_wstrb[3]}} & cache_final_data[31:24]) },
                                        { ({8{cpu_mem_wstrb[2]}} & cpu_mem_wdata[23:16]) |
({8{~cpu_mem_wstrb[2]}} & cache_final_data[23:16]) },
                                        { ({8{cpu_mem_wstrb[1]}} & cpu_mem_wdata[15: 8]) |
({8{~cpu_mem_wstrb[1]}} & cache_final_data[15: 8]) },
                                        { ({8{cpu_mem_wstrb[0]}} & cpu_mem_wdata[ 7: 0]) |
({8{~cpu_mem_wstrb[0]}} & cache_final_data[ 7: 0]) }
                                         };

        //finally refresh to data array
        assign cache_modified_block =
        {`LINE_LEN{offset[4:2]==3'b000 }} & {cache_block_data[`LINE_LEN-1 :
32],cache_modified_final} |
        {`LINE_LEN{offset[4:2]==3'b001 }} & {cache_block_data[`LINE_LEN-1 :
64],cache_modified_final,cache_block_data[31:0]} |
        {`LINE_LEN{offset[4:2]==3'b010 }} & {cache_block_data[`LINE_LEN-1 :
96],cache_modified_final,cache_block_data[63:0]} |
        {`LINE_LEN{offset[4:2]==3'b011 }} & {cache_block_data[`LINE_LEN-1 :
128],cache_modified_final,cache_block_data[95:0]} |
        {`LINE_LEN{offset[4:2]==3'b100 }} & {cache_block_data[`LINE_LEN-1 :
160],cache_modified_final,cache_block_data[127:0]} |
        {`LINE_LEN{offset[4:2]==3'b101 }} & {cache_block_data[`LINE_LEN-1 :
192],cache_modified_final,cache_block_data[159:0]} |
        {`LINE_LEN{offset[4:2]==3'b110 }} & {cache_block_data[`LINE_LEN-1 :
224],cache_modified_final,cache_block_data[191:0]} |
        {`LINE_LEN{offset[4:2]==3'b111 }} & {cache_modified_final,cache_block_data[223:0]} ;

        assign bypath_write_data = cpu_mem_wdata;
        //write back to MEM
        assign to_mem_wr_req_addr = ( {32{bypath}} & cpu_mem_addr )                    |
                                    ( {32{~bypath & choose0}} & {Tag0,set,5'b0} )      |
                                    ( {32{~bypath & choose1}} & {Tag1,set,5'b0} )      |
                                    ( {32{~bypath & choose2}} & {Tag2,set,5'b0} )      |
                                    ( {32{~bypath & choose3}} & {Tag3,set,5'b0} )      ;
        assign to_mem_wr_req_len = {5'b0,{3{~bypath}}};

        //use last shifter to get last signal
        always @(posedge clk) begin
                if(cur_state == MEM_WR | cur_state == BY_REQ) begin
                        if(bypath)
                                last_shifter <= 8'b1;
                        else
```

```
                        last_shifter <= {1'b1,7'b0};
                end
                else if(cur_state == TXD & from_mem_wr_data_ready) begin
                        last_shifter <= {1'b0,last_shifter[7:1]};
                end
        end

        //use dirty block shifter to transmit it 4byte each time
        always @(posedge clk) begin
                if(cur_state == MEM_WR) begin
                        dirty_block_data <= cache_block_data;
                end
                else if(cur_state == TXD & from_mem_wr_data_ready) begin
                        dirty_block_data <= {32'b0 , dirty_block_data[255:32]};
                end
        end

        assign to_mem_wr_data_last = last_shifter[0] & from_mem_wr_data_ready &
to_mem_wr_data_valid;
        assign to_mem_wr_data = ( {32{ bypath}} & bypath_write_data)      |
                                ( {32{~bypath}} & dirty_block_data[31:0]) ;
        //for dirty block, strb are always 4'b1111 to ensure complete data transmit
        //for bypath , depends on input port
        assign to_mem_wr_data_strb = {4{~bypath}} | cpu_mem_wstrb;
```

在写命中或写缺失后数据重填完毕后，需要将 cache 中对应位置的 32 位数据取出并根据 wstrb 将相应位置修改为 cpu 写入的数据，然后在根据 offset 修改 cache 块的对应位置，将修改后的块重填至 data 数组。

旁路写内存或读写缺失时，需要根据旁路标志或替换算法的选择确定写内存的地址和长度。对于读写缺失的情形，写内存地址由被替换块的 tag 等相关信息确定。

向内存 burst 传输数据时，预先在 MEM_WR 请求阶段即将 cache 上的数据块搬移到一个寄存器中，通过每次传输低 32 位并右移，以从低向高往内存传输数据。同时设置 last_shifter 寄存器，以最低位表示 last 信号。该寄存器将 1 放置在高位并不断右移，从而使得最后一个数据传输时 last 信号拉高。由于 last

最好在传输结束释放，额外考虑了向内存的写数据握手信号。

最后，需要根据旁路标志确定向内存写数据的来源，及相应 strb 信号。

8、控制信号

```
//handshake signals related to state machine
        assign to_cpu_mem_req_ready   = cur_state == WAIT;
        assign to_cpu_cache_rsp_valid = cur_state == RESP;

        assign to_mem_rd_req_valid  = (cur_state == MEM_RD) | (cur_state == BY_REQ & ~cpu_mem_rw);
        assign to_mem_rd_rsp_ready  = (cur_state == RECV) | (cur_state == BY_RECV) | (cur_state
== WAIT);

        assign to_mem_wr_req_valid  = (cur_state == MEM_WR) | (cur_state == BY_REQ & cpu_mem_rw);
        assign to_mem_wr_data_valid = (cur_state == TXD) | (cur_state == BY_TXD);
```

根据相应状态含义确定 CPU 和内存的请求和响应相关信号。由于旁路读写请求合并为一个状态，而读写请求不能同时有效，此处应当考虑读写类型。特别的，由于内存接口设计，在初始态应当将 to_mem_rd_rsp_ready 拉高。

9、模块例化

*说明：由于对于 4 个路组的例化较为重复，仅分别展示 0 路组的例化。

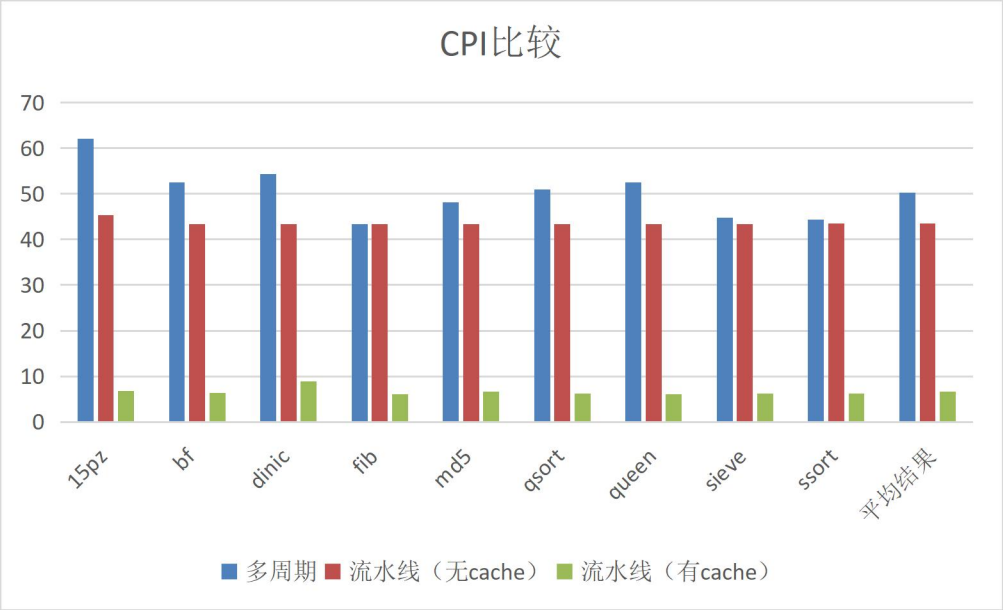
```
//inst of tag_array and data_array
        tag_array tag_way0(
                .clk    (clk),
                .waddr  (set),
                .raddr  (set),
                .wen    (TagWen0),
                .wdata  (tag),
                .rdata  (Tag0)
        );

...

        data_array data_way0(
                .clk    (clk),
                .waddr  (set),
```

```
            .raddr  (set),
            .wen    (DataWen0),
            .wdata  (Array_Wdata),
            .rdata  (Data0)
    );
```

其中 set 对于不同路组均一致，且写数据同样一致，根据不同路的写使能信号选择写入哪个路组。

10、性能比较(100hz)

	处理器(RISCV)	时钟周期数	指令总数	访存延迟周期数	CPI
15pz	多周期	324238832	5224477	90004486	62.06
	流水线(无 cache)	236498270	5224477	94278671	45.27
	流水线(有 cache)	35492152	5224477	249547	6.79
bf	多周期	23775610	452850	3956925	52.50
	流水线(无 cache)	19655008	452850	3947242	43.40
	流水线(有 cache)	2847937	452850	630209	6.29
dinic	多周期	906059	16687	171982	54.30
	流水线(无 cache)	723066	16687	172866	43.33
	流水线(有 cache)	148090	16687	9475	8.87
fib	多周期	110676382	2549521	183886	43.41
	流水线(无 cache)	110479201	2549521	185795	43.33
	流水线(有 cache)	15598973	2549521	1853649	6.12
md5	多周期	236203	4911	22927	48.10
	流水线(无 cache)	212774	4911	22781	43.33
	流水线(有 cache)	32305	4911	774	6.58
qsort	多周期	483070	9476	65405	50.98
	流水线(无 cache)	410526	9476	66011	43.32
	流水线(有 cache)	58659	9476	6992	6.19
queen	多周期	4284106	81486	637557	52.57
	流水线(无 cache)	3532577	81486	640402	43.35
	流水线(有 cache)	492974	81486	43914	6.05
sieve	多周期	456038	10191	14335	44.75
	流水线(无 cache)	441542	10191	14452	43.33
	流水线(有 cache)	63947	10191	4836	6.27
ssort	多周期	27431823	619041	514622	44.31
	流水线(无 cache)	26922713	619041	554778	43.49
	流水线(有 cache)	3808486	619041	413803	6.15
平均结果	多周期	54720902.56	996515.56	10619125	54.91
	流水线(无 cache)	44319519.67	996515.56	11098110.89	44.47
	流水线(有 cache)	6504835.89	996515.56	357022.11	6.59



由上表可见，添加 cache 后大幅度降低了流水线的访存延迟，增加了指令并行度。添加 cache 的流水线性能约达到了未加 cache 流水线的 7 倍，多周期处理器的 9 倍。但是加 cache 流水线的 CPI 仍只达到 6 左右，未能降至理想的 1，一方面是因为存在一定读写缺失的概率，当读写缺失时，由于需要进行缓存块替换，比直接从内存获得 32 位数据消耗更多的时钟周期。另一方面，由于 icache 和 dcache 仍采用多周期实现，即使读写命中，也需经过多个状态，一定程度上降低了运行效率。如果使用流水线实现 cache，有望进一步降低 CPI。

10、结构设计优化

由于同时实现流水线和 cache，对二者的结构优化同时进行，并使用 Dhrystone 和 Coremark 进行性能评测。

初始版的 icache 和 dcache 均未例化 data_array 和 tag_array，而是在模块内部直接定义寄存器，一方面不便于修改 cache 容量大小，另一方面，由于寄存器摆放较不对齐，会消耗额外的布线资源，因此当 cache 修改为 4x32（4 路组，每个路组中 32 个 cache 块）之后，bit-gen 资源就已经不够了。

在通过例化 tag_array 和 data_array 后，分别尝试了增大路组数和增大路组内 cache 块数的尝试。发现使用 8 路组时，每个路组只能放置 16 个 cache 块，且通过性能评测，发现和 4x16 相比几乎无性能优化，查阅资料得知，路组数达到 4 以上后，增大路组数提高的效率已经微乎其微，因此后续容量在 4 路组基础上增大每路组数据块数。

经过尝试，cache 容量最大可开至 4x128。在验收过程中，通过刘士祺助教的讲解得知 vivado 会对成块的 cache 块组进行一定的布局布线优化，且可节省布局布线资源，将 cache 容量增大到 4x128 后，根据 Dhrystone 的性能评测结果，性能约提高了 6%，根据 Coremark 也有了一定的性能提升。

然后，通过增大时钟频率加快处理器的运行。经过测试发现，在 280hz 时可以保证所有 bit-gen 任务 WNS 均大于 0，在 300hz 时也只有部分任务 WNS 小于 0，且处理器通过 fpga 上板任务的最高频率（不稳定）可以达到 400hz 以上。对于 WNS 小于 0 的任务，查看 bit-gen 阶段报告可见，关键路径主要为框架 dcache 相关框架代码部分，优化空间较小，扇出也均符合规范，不需特别优化。

综上，最后得到稳定的处理器为 280hz 时钟频率下，icache 和 dcache 容量均为 4x128 的流水线处理器，和最开始 100hz 下 cache 容量 4x8 的流水席处理器性能对比如下：

流水线(有 cache)	Dhrystones per second	Comark Iterations/Sec
100hz 4x8	5411	0.676
100hz 4x128	5774	0.676
280hz 4x128	16168	1.894
稳定优化倍数	2.988	2.802

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）

问题 1：仿真通过但上板和硬件仿真无法结束

通过手动确定硬件仿真时间范围，查看波形可见最终 PC 可以到达 hit good trap 位置，且通过旁路向内存 0x0c 位置写 0。分析与内存交互信号发现，读写有效请求同时拉高，内存优先获取读请求有效信号，而忽略写请求。进一步查看代码相关逻辑，发现旁路读写请求用同一个状态表示，应当进一步根据读写类型进行分类，保证读写请求不同时有效。

问题 2：上板每组任务只有第 1 个可以通过，其余任务可以通过硬件仿真

检查 cache 内部功能实现无误情况下，将问题定位至与内存交互信号。与多周期处理器和未加 cache 的流水线处理器对比发现，应当在初始状态拉高向内存的读数据准备信号。添加后即可正确通过。

问题 3：框架代码 tag_array 和 data_array 问题

在行为仿真过程中，发现部分任务出现了 wdata=xxxx 的不定态，查看波形发现 set 改变后，data_array 反馈的数据并未变化，进一步查看发现，问题在于框架代码中声明数组时优先级错误。由于减法优先级高于移位优先级，应当在移位运算外加上括号。

三、 对讲义中思考题（如有）的理解和回答

本次实验无思考题。

四、 在课后，你花费了大约_____15_____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）

心得感受:

本次实验难度适中，较好的加深了对于组相联缓存结构，和缺失替换策略的理解。同时讲义对组路中数据结构的讲解也较为清楚明晰，整体实验实现过程较为顺畅。同时在进行结构设计优化的过程中，通过增大路组数的比较和增大路组内容量的比较，更清晰的了解了组相联缓存策略的替换算法和 vivado 对于布局布线的优化。

建议:

实验过程中可以强调和内存接口的交互信号问题，鼓励同学们通过查看框架代码掌握内存接口逻辑，从而定位和修改自身代码错误。

致谢:

感谢常老师和陈欲晓助教在硬件仿真工具上的帮助，感谢刘士祺助教在结构设计优化及 vivado 优化布局布线上的讲解。