《计算机组成原理》实验报告

年级、专业、班级		**姓名**	
实验题目	实验五 Cache		
实验时间		**实验地点**	
实验成绩	优秀/良好/中等	**实验性质**	□验证性 ☑设计性 □综合性
教师评价： □算法/实验过程正确； □源程序/实验内容提交； □程序结构/实验步骤合理； □实验结果正确； □语法、语义正确； □报告规范； 其他: 评价教师:			
实验目的 (1)加深对 Cache 原理的理解。 (2)通过使用 verilog 实现 Cache，加深对状态机的理解。			

报告完成时间: 2024 年 7 月 21 日

1 实验内容

根据实验指导书完成以下任务：

(1) 最低要求：参考指导书中直接映射写直达 Cache 的实现，实现写回策略的 Cache
(2) 替换实验环境中的 Cache 模块，并通过仿真测试
(3) (选做)(较高要求) 性能优化，实现 2 路组相联的 Cache
(4) (选做)(更高要求) 性能优化，使用伪 LRU 等替换策略实现 4 路以上组相联的 Cache
(5) (选做)(最高要求) 实现其它 Cache 性能优化方法，如 axi burst 传输

2 实验设计

2.1 DataCache 模块

2.1.1 功能描述

本次实验拟实现 MIPS Core 中数据存储器 (Data Memory) 所对应的 Cache 模块。当 MIPS Core 向 Cache 模块请求数据时,Cache 模块如果命中, 则可以马上返回数据, 否则需要访问内存。

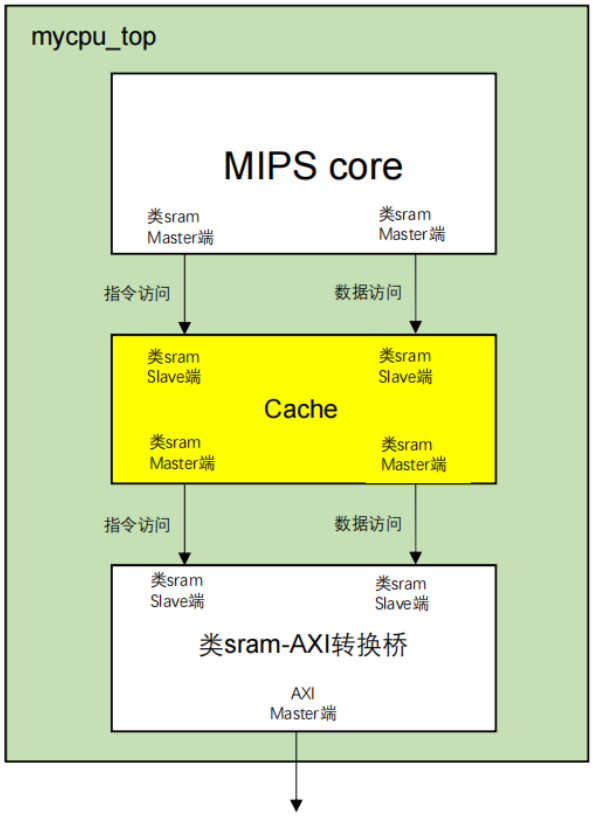

图 1: 顶层架构 (mycpu_top)

2.1.2 接口定义

表 1: DataCache 接口定义

信号名	方向	位宽	功能描述
clk	Input	1-bit	时钟信号
rst	Input	1-bit	复位信号
cpu_data_req	Input	1-bit	CPU 数据请求信号
cpu_data_wr	Input	1-bit	CPU 数据请求写信号
cpu_data_size	Input	2-bit	CPU 数据请求字节数
cpu_data_addr	Input	32-bit	CPU 数据请求地址
cpu_data_wdata	Input	32-bit	CPU 数据请求写数据
cpu_data_rdata	Output	32-bit	CPU 数据请求读数据
cpu_data_addr_ok	Output	1-bit	CPU 数据请求地址 (和数据) 被接收
cpu_data_data_ok	Output	1-bit	CPU 数据请求读数据返回/写数据写入完成
cache_data_req	Output	1-bit	Cache 数据请求信号
cache_data_wr	Output	1-bit	Cache 数据请求写信号
cache_data_size	Output	2-bit	Cache 数据请求字节数
cache_data_addr	Output	32-bit	Cache 数据请求地址
cache_data_wdata	Output	32-bit	Cache 数据请求写数据
cache_data_rdata	Input	32-bit	Cache 数据请求读数据
cache_addr_ok	Input	1-bit	主存接收到请求地址 (和数据)
cache_data_data_ok	Input	1-bit	Cache 数据请求读数据返回/写数据写入完成

3 实验过程记录

3.1 DataCache

3.1.1 问题 1: 检查延迟

问题描述:在写回 Cache 中, 由于无法重写块,Store 操作需要两个周期 (一个周期用来检查命中情况, 下一个周期才真正执行写操作)。

解决方案:增加写缓冲区。

3.1.2 问题 2: 同周期数据读入和比较的时序问题

问题描述:由于在同一个周期内既进行了数据读入, 又进行了命中检查所导致的时序问题。

解决方案:将数据读入设置为时钟上升沿, 将命中检查设置在紧随其后的时钟下降沿。

3.1.3 问题 3: 写缺失代价

问题描述: 当发生缺失替换一个被修改的块时, 一般需要和主存进行两次数据交互 (先写后读); 当下一次缺失没有立即发生时, 可以先执行读主存, 再隐式的执行写主存

解决方案: 增加写回缓冲区。

3.1.4 问题 4: 写回缓冲区的优先级

问题描述: 由于顶层模块是根据检查当前执行的指令是否经过 DataCache 来确定主存访问优先级的。对于写回缓冲区, 正在写回的数据与正在执行的指令没有对应关系, 但必须被优先执行。

解决方案: 将”mycpu_top” 模块下”bridge_2x1 bridge_2x1” 子模块的”.no_dcache” 信号修改为 (no_dcache & cache_data_wr)。

3.1.5 问题 5: 数组和多位宽信号大小端不一致

问题描述: 对于数组信号, 由低到高的访问顺序为自左向右, 对于多位宽信号则为自右向左。

解决方案: 将”reg [GROUP_WIDTH-1:0] cache_valid [CACHE_DEEPTH-1:0]” 修改为”reg cache_valid [CACHE_DEEPTH-1:0][GROUP_WIDTH-1:0]” (与”reg [TAG_WIDTH-1:0] cache_tag [CACHE_DEEPTH-1:0][GROUP_WIDTH-1:0]” 保持一致, 便于检查命中情况)。

4 实验结果及分析

4.1 DataCache

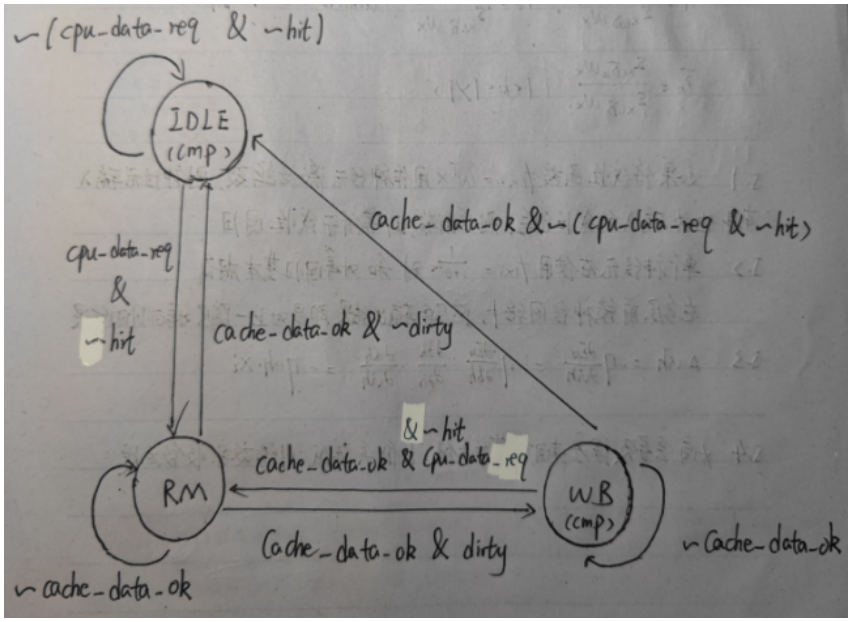

图 2: 有限状态机

```
============================================================
Test begin!
shell1 test begin.

dgemm PASS!

Total Count(SoC count) = 0x2cf24

Total Count(CPU count) = 0x146e2

============================================================
Test end!
----PASS!!!
$finish called at time : 2078220500 ps : File "E:/Vivaodo/Projects
```

图 3: 控制台 (写回策略)

```
============================================================
Test begin!
shell1 test begin.

dgemm PASS!

Total Count(SoC count) = 0x2c449

Total Count(CPU count) = 0x141f2

============================================================
Test end!
----PASS!!!
$finish called at time : 2043768500 ps : File "E:/Vivaodo/Projects
```

图 4: 控制台 (两路组相联)

```
============================================================
Test begin!
shell1 test begin.

dgemm PASS!

Total Count(SoC count) = 0x2c936

Total Count(CPU count) = 0x14430

============================================================
Test end!
----PASS!!!
$finish called at time : 2054174500 ps : File "E:/Vivaodo/Projects
```

图 5: 控制台 (八路组相联)

A 写回策略 DataCache 代码 (基于写缓冲区和写回缓冲区策略)

```
module DataCache_WB(
    input clk,rst,
    input cpu_data_req,
    input cpu_data_wr,
    input [1:0] cpu_data_size,
    input [31:0] cpu_data_addr,
    input [31:0] cpu_data_wdata,
    output [31:0] cpu_data_rdata,
    output cpu_data_addr_ok,
    output cpu_data_data_ok,

    output cache_data_req,
    output cache_data_wr,
    output [1:0] cache_data_size,
    output [31:0] cache_data_addr,
    output [31:0] cache_data_wdata,
    input [31:0] cache_data_rdata,
    input cache_addr_ok,//cache_data_addr_ok->cache_addr_ok(该信号不止与
        DataCache相关)
    input cache_data_data_ok
    );
//Cache的规格
    parameter INDEX_WIDTH=10;
    parameter OFFSET_WIDTH=2;
    localparam TAG_WIDTH=32-INDEX_WIDTH-OFFSET_WIDTH;
    localparam CACHE_DEEPTH=1<<INDEX_WIDTH;

    reg cache_valid [CACHE_DEEPTH-1:0];
    reg cache_dirty [CACHE_DEEPTH-1:0];
    reg [TAG_WIDTH-1:0] cache_tag [CACHE_DEEPTH-1:0];
    reg [31:0] cache_block [CACHE_DEEPTH-1:0];
//本次访问的地址解析
    wire [TAG_WIDTH-1:0] tag;
    wire [INDEX_WIDTH-1:0] index;
    wire [OFFSET_WIDTH-1:0] offset;

    assign tag=cpu_data_addr[31:INDEX_WIDTH+OFFSET_WIDTH];
    assign index=cpu_data_addr[INDEX_WIDTH+OFFSET_WIDTH-1:OFFSET_WIDTH];
    assign offset=cpu_data_addr[OFFSET_WIDTH-1:0];
//判断是否命中
    wire hit;
    assign hit=cache_valid[index] & (tag==cache_tag[index]);
//掩码和处理后的写入数据
    wire [3:0] mask;
    wire [31:0] write_data;
    wire [31:0] rwrite_data;
```

```
    assign mask=(cpu_data_size==2'b00)?1'b1<<offset:(cpu_data_size==2'b01)?2'
        b11<<offset:4'b1111;
    assign write_data=(cache_block[index] & ~{{8{mask[3]}},{8{mask[2]}},{8{mask
        [1]}},{8{mask[0]}}}) |
                    (cpu_data_wdata & {{8{mask[3]}},{8{mask[2]}},{8{mask
                        [1]}},{8{mask[0]}}});
    assign rwrite_data=(cache_data_rdata & ~{{8{mask[3]}},{8{mask[2]}},{8{mask
        [1]}},{8{mask[0]}}}) |
                    (cpu_data_wdata & {{8{mask[3]}},{8{mask[2]}},{8{mask
                        [1]}},{8{mask[0]}}});
//FSM
    localparam IDLE=2'b00,RM=2'b01,WB=2'b10;
    reg [1:0] state;

    always@(posedge clk or posedge rst)
    begin
        if(rst)
            state<=IDLE;
        else
        begin
            case(state)
                IDLE:state<=(cpu_data_req & ~hit)?RM:IDLE;
                RM:state<=(~cache_data_data_ok)?RM:
                            (cache_dirty[index])?WB:IDLE;
                WB:state<=(~cache_data_data_ok)?WB:
                            (cpu_data_req & ~hit)?RM:IDLE;
                 default state<=IDLE;
            endcase
        end
    end
//地址发送成功后拉低请求信号
    wire cache_data_addr_ok;
    reg send_req=1'b1;
    assign cache_data_addr_ok=cache_data_req & cache_addr_ok;

    always@(posedge clk or posedge rst)
    begin
        if(rst)
            send_req<=1'b1;
        else begin
            send_req<=(cache_data_addr_ok)?1'b0:
                        (cache_data_data_ok)?1'b1:send_req;
        end
    end
//写缓冲区
    reg store_dirty;
    reg store_write;
    reg [31:0] store_buffer;
    reg [31:0] store_addr;
```

```
wire [INDEX_WIDTH-1:0] store_index;
wire [TAG_WIDTH-1:0] store_tag;

assign store_tag=store_addr[31:INDEX_WIDTH+OFFSET_WIDTH];
assign store_index=store_addr[INDEX_WIDTH+OFFSET_WIDTH-1:OFFSET_WIDTH];
assign store_offset=store_addr[OFFSET_WIDTH-1:0];

always@(negedge clk)
begin
    if(state==RM)
    begin
        if(cpu_data_wr)
        begin
            store_buffer<=rwrite_data;
            store_dirty<=1'b1;
        end
        else begin
            store_buffer<=cache_data_rdata;
            store_dirty<=1'b0;
        end
        store_write<=cache_data_data_ok;
    end
    else begin
        if(cpu_data_req)
        begin
            store_buffer<=write_data;
            store_addr<=cpu_data_addr;
        end
        store_dirty<=1'b1;
        store_write<=cpu_data_req & cpu_data_wr & hit;
    end
end

integer i;
always@(posedge clk or posedge rst)
begin
    if(rst)
    begin
        store_buffer<=32'b0;
        for(i=0;i<CACHE_DEEPTH;i=i+1)
        begin
            cache_valid[i]<=1'b0;
            cache_dirty[i]<=1'b0;
        end
    end
    else if(store_write)
    begin
            cache_dirty[store_index]<=store_dirty;
            cache_valid[store_index]<=1'b1;
```

```
                cache_tag[store_index]<=store_tag;
                cache_block[store_index]<=store_buffer;
            end
        end
    //写回缓冲区
        reg [31:0] write_buffer;
        reg [31:0] write_addr;

        always@(negedge clk or posedge rst)
        begin
            if(rst)
            begin
                write_buffer<=32'b0;
                write_addr<=32'b0;
            end
            else if((state==RM) & cache_dirty[store_index])
            begin
                write_buffer<=cache_block[store_index];
                write_addr<={cache_tag[store_index],store_index,2'b00};
            end
        end
    //MIPS Core接口
        assign cpu_data_addr_ok=(cpu_data_req & hit) | cache_data_addr_ok;
        assign cpu_data_data_ok=(cpu_data_req & hit) | ((state==RM) &
            cache_data_data_ok);
        assign cpu_data_rdata=(hit)?cache_block[index]:cache_data_rdata;
    //AXI接口
        assign cache_data_req=(state!=IDLE) & send_req;
        assign cache_data_wr=(state==WB);
        assign cache_data_wdata=write_buffer;
        assign cache_data_size=(state==WB)?2'b10:cpu_data_size;
        assign cache_data_addr=(state==WB)?write_addr:cpu_data_addr;
    endmodule
```

B 写回策略 DataCache 代码 (两路组相联)

```
module DataCache_WB(
    input clk,rst,
    input cpu_data_req,
    input cpu_data_wr,
    input [1:0] cpu_data_size,
    input [31:0] cpu_data_addr,
    input [31:0] cpu_data_wdata,
    output [31:0] cpu_data_rdata,
    output cpu_data_addr_ok,
    output cpu_data_data_ok,
```

```
    output cache_data_req,
    output cache_data_wr,
    output [1:0] cache_data_size,
    output [31:0] cache_data_addr,
    output [31:0] cache_data_wdata,
    input [31:0] cache_data_rdata,
    input cache_addr_ok,//cache_data_addr_ok->cache_addr_ok(该信号不止与
        DataCache相关)
    input cache_data_data_ok
    );
//Cache的规格
    parameter ASSOCIATIVITY_WIDTH=1;
    parameter INDEX_WIDTH=10-ASSOCIATIVITY_WIDTH;
    parameter OFFSET_WIDTH=2;
    localparam TAG_WIDTH=32-INDEX_WIDTH-OFFSET_WIDTH;
    localparam CACHE_DEEPTH=1<<INDEX_WIDTH;
    localparam GROUP_WIDTH=1<<ASSOCIATIVITY_WIDTH;

    reg cache_valid [CACHE_DEEPTH-1:0][GROUP_WIDTH-1:0];
    reg cache_dirty [CACHE_DEEPTH-1:0][GROUP_WIDTH-1:0];
    reg [TAG_WIDTH-1:0] cache_tag [CACHE_DEEPTH-1:0][GROUP_WIDTH-1:0];
    reg [31:0] cache_block [CACHE_DEEPTH-1:0][GROUP_WIDTH-1:0];
//本次访问的地址解析
    wire [TAG_WIDTH-1:0] tag;
    wire [INDEX_WIDTH-1:0] index;
    wire [OFFSET_WIDTH-1:0] offset;

    assign tag=cpu_data_addr[31:INDEX_WIDTH+OFFSET_WIDTH];
    assign index=cpu_data_addr[INDEX_WIDTH+OFFSET_WIDTH-1:OFFSET_WIDTH];
    assign offset=cpu_data_addr[OFFSET_WIDTH-1:0];
//判断是否命中
    wire [ASSOCIATIVITY_WIDTH-1:0] target;
    wire [GROUP_WIDTH-1:0] hits;
    wire hit;

    genvar i;
    generate
        for(i=0;i<GROUP_WIDTH;i=i+1)
        begin:Hits
            assign hits[i]=cache_valid[index][i] & (tag==cache_tag[index][i]);
        end
    endgenerate

    assign target=(hits==2'b10)?1'b1:1'b0;
    assign hit=| hits;
//替换策略
    reg [ASSOCIATIVITY_WIDTH-1:0] replace [CACHE_DEEPTH-1:0];

    integer j;
```

```
    always@(negedge clk)
    begin
        if(rst)
        for(j=0;j<CACHE_DEEPTH;j=j+1)
        begin
            replace[j]<=0;
        end
        else if(hit)
        begin
            replace[index]<=~target;
        end
    end
//掩码和处理后的写入数据
    wire [3:0] mask;
    wire [31:0] write_data;
    wire [31:0] rwrite_data;
    assign mask=(cpu_data_size==2'b00)?1'b1<<offset:(cpu_data_size==2'b01)?2'
        b11<<offset:4'b1111;
    assign write_data=(cache_block[index][target] & ~{{8{mask[3]}},{8{mask
        [2]}},{8{mask[1]}},{8{mask[0]}}}) |
                        (cpu_data_wdata & {{8{mask[3]}},{8{mask[2]}},{8{mask
                            [1]}},{8{mask[0]}}});
    assign rwrite_data=(cache_data_rdata & ~{{8{mask[3]}},{8{mask[2]}},{8{mask
        [1]}},{8{mask[0]}}}) |
                        (cpu_data_wdata & {{8{mask[3]}},{8{mask[2]}},{8{mask
                            [1]}},{8{mask[0]}}});
//FSM
    localparam IDLE=2'b00,RM=2'b01,WB=2'b10;
    reg [1:0] state;
    reg [ASSOCIATIVITY_WIDTH-1:0] store_target;

    always@(posedge clk or posedge rst)
    begin
        if(rst)
            state<=IDLE;
        else
        begin
            case(state)
                IDLE:state<=(cpu_data_req & ~hit)?RM:IDLE;
                RM:state<=(~cache_data_data_ok)?RM:
                            (cache_dirty[index][store_target])?WB:IDLE;
                WB:state<=(~cache_data_data_ok)?WB:
                            (cpu_data_req & ~hit)?RM:IDLE;
                default state<=IDLE;
            endcase
        end
    end
//地址发送成功后拉低请求信号
    wire cache_data_addr_ok;
```

```
    reg send_req=1'b1;
    assign cache_data_addr_ok=cache_data_req & cache_addr_ok;

    always@(posedge clk or posedge rst)
    begin
        if(rst)
            send_req<=1'b1;
        else begin
            send_req<=(cache_data_addr_ok)?1'b0:
                        (cache_data_data_ok)?1'b1:send_req;
        end
    end
//写缓冲区
    reg store_dirty;
    reg store_write;
    reg [31:0] store_buffer;
    reg [31:0] store_addr;
    wire [INDEX_WIDTH-1:0] store_index;
    wire [TAG_WIDTH-1:0] store_tag;

    assign store_tag=store_addr[31:INDEX_WIDTH+OFFSET_WIDTH];
    assign store_index=store_addr[INDEX_WIDTH+OFFSET_WIDTH-1:OFFSET_WIDTH];
    assign store_offset=store_addr[OFFSET_WIDTH-1:0];

    always@(negedge clk)
    begin
        if(state==RM)
        begin
            if(cpu_data_wr)
            begin
                store_buffer<=rwrite_data;
                store_dirty<=1'b1;
            end
            else begin
                store_buffer<=cache_data_rdata;
                store_dirty<=1'b0;
            end
            store_write<=cache_data_data_ok;
            store_target<=replace[store_index];
        end
        else begin
            if(cpu_data_req)
            begin
                store_buffer<=write_data;
                store_addr<=cpu_data_addr;
                if(hit)
                store_target<=target;
            end
            store_dirty<=1'b1;
```

```
            store_write<=cpu_data_req & cpu_data_wr & hit;
        end
    end

    integer k;
    always@(posedge clk or posedge rst)
    begin
        if(rst)
        begin
            store_buffer<=32'b0;
            for(j=0;j<CACHE_DEEPTH;j=j+1)
            for(k=0;k<GROUP_WIDTH;k=k+1)
            begin
                cache_valid[j][k]<=0;
                cache_dirty[j][k]<=0;
                cache_tag[j][k]<=0;
            end
        end
        else if(store_write)
        begin
            cache_dirty[store_index][store_target]<=store_dirty;
            cache_valid[store_index][store_target]<=1'b1;
            cache_tag[store_index][store_target]<=store_tag;
            cache_block[store_index][store_target]<=store_buffer;
        end
    end
//写回缓冲区
    reg [31:0] write_buffer;
    reg [31:0] write_addr;

    always@(negedge clk or posedge rst)
    begin
        if(rst)
        begin
            write_buffer<=32'b0;
            write_addr<=32'b0;
        end
        else if((state==RM) & cache_dirty[store_index][store_target])
        begin
            write_buffer<=cache_block[store_index][store_target];
            write_addr<={cache_tag[store_index][store_target],store_index,2'b00
                };
        end
    end
//MIPS Core接口
    assign cpu_data_addr_ok=(cpu_data_req & hit) | cache_data_addr_ok;
    assign cpu_data_data_ok=(cpu_data_req & hit) | ((state==RM) &
        cache_data_data_ok);
```

```
    assign cpu_data_rdata=(hit)?cache_block[index][store_target]:
        cache_data_rdata;
//AXI接口
    assign cache_data_req=(state!=IDLE) & send_req;
    assign cache_data_wr=(state==WB);
    assign cache_data_wdata=write_buffer;
    assign cache_data_size=(state==WB)?2'b10:cpu_data_size;
    assign cache_data_addr=(state==WB)?write_addr:cpu_data_addr;
endmodule
```

C 写回策略 DataCache 代码 (基于伪 LRU 替换策略的八路组相联)

```
module DataCache_WB(
    input clk,rst,
    input cpu_data_req,
    input cpu_data_wr,
    input [1:0] cpu_data_size,
    input [31:0] cpu_data_addr,
    input [31:0] cpu_data_wdata,
    output [31:0] cpu_data_rdata,
    output cpu_data_addr_ok,
    output cpu_data_data_ok,

    output cache_data_req,
    output cache_data_wr,
    output [1:0] cache_data_size,
    output [31:0] cache_data_addr,
    output [31:0] cache_data_wdata,
    input [31:0] cache_data_rdata,
    input cache_addr_ok,//cache_data_addr_ok->cache_addr_ok(该信号不止与
        DataCache相关)
    input cache_data_data_ok
    );
//Cache的规格
    parameter ASSOCIATIVITY_WIDTH=3;
    parameter INDEX_WIDTH=10-ASSOCIATIVITY_WIDTH;
    parameter OFFSET_WIDTH=2;
    localparam TAG_WIDTH=32-INDEX_WIDTH-OFFSET_WIDTH;
    localparam CACHE_DEEPTH=1<<INDEX_WIDTH;
    localparam GROUP_WIDTH=1<<ASSOCIATIVITY_WIDTH;

    reg cache_valid [CACHE_DEEPTH-1:0][GROUP_WIDTH-1:0];
    reg cache_dirty [CACHE_DEEPTH-1:0][GROUP_WIDTH-1:0];
    reg [TAG_WIDTH-1:0] cache_tag [CACHE_DEEPTH-1:0][GROUP_WIDTH-1:0];
    reg [31:0] cache_block [CACHE_DEEPTH-1:0][GROUP_WIDTH-1:0];
//本次访问的地址解析
    wire [TAG_WIDTH-1:0] tag;
```

```
    wire [INDEX_WIDTH-1:0] index;
    wire [OFFSET_WIDTH-1:0] offset;

    assign tag=cpu_data_addr[31:INDEX_WIDTH+OFFSET_WIDTH];
    assign index=cpu_data_addr[INDEX_WIDTH+OFFSET_WIDTH-1:OFFSET_WIDTH];
    assign offset=cpu_data_addr[OFFSET_WIDTH-1:0];
//判断是否命中
    wire [ASSOCIATIVITY_WIDTH-1:0] target;
    wire [GROUP_WIDTH-1:0] hits;
    wire hit;

    genvar i;
    generate
        for(i=0;i<GROUP_WIDTH;i=i+1)
        begin:Hits
            assign hits[i]=cache_valid[index][i] & (tag==cache_tag[index][i]);
        end
    endgenerate

    assign target=(hits==8'b1000_0000)?3'b111:
                  (hits==8'b0100_0000)?3'b110:
                  (hits==8'b0010_0000)?3'b101:
                  (hits==8'b0001_0000)?3'b100:
                  (hits==8'b0000_1000)?3'b011:
                  (hits==8'b0000_0100)?3'b010:
                  (hits==8'b0000_0010)?3'b001:3'b000;
    assign hit=| hits;
//替换策略
    wire [ASSOCIATIVITY_WIDTH-1:0] replace [CACHE_DEEPTH-1:0];
    wire enable [CACHE_DEEPTH-1:0];

    generate
    for(i=0;i<CACHE_DEEPTH;i=i+1)
    begin:LRUS
        assign enable[i]=hit & (index==i);
        PseudoLRU LRU(clk,rst,enable[i],target,replace[i]);
    end
    endgenerate
//掩码和处理后的写入数据
    wire [3:0] mask;
    wire [31:0] write_data;
    wire [31:0] rwrite_data;
    assign mask=(cpu_data_size==2'b00)?1'b1<<offset:(cpu_data_size==2'b01)?2'
        b11<<offset:4'b1111;
    assign write_data=(cache_block[index][target] & ~{{8{mask[3]}},{8{mask
        [2]}},{8{mask[1]}},{8{mask[0]}}}) |
                          (cpu_data_wdata & {{8{mask[3]}},{8{mask[2]}},{8{mask
                              [1]}},{8{mask[0]}}});
```

```
    assign rwrite_data=(cache_data_rdata & ~{{8{mask[3]}},{8{mask[2]}},{8{mask
        [1]}},{8{mask[0]}}}) |
                        (cpu_data_wdata & {{8{mask[3]}},{8{mask[2]}},{8{mask
                            [1]}},{8{mask[0]}}});
//FSM
    localparam IDLE=2'b00,RM=2'b01,WB=2'b10;
    reg [1:0] state;
    reg [ASSOCIATIVITY_WIDTH-1:0] store_target;

    always@(posedge clk or posedge rst)
    begin
        if(rst)
            state<=IDLE;
        else
        begin
            case(state)
                IDLE:state<=(cpu_data_req & ~hit)?RM:IDLE;
                RM:state<=(~cache_data_data_ok)?RM:
                            (cache_dirty[index][store_target])?WB:IDLE;
                WB:state<=(~cache_data_data_ok)?WB:
                            (cpu_data_req & ~hit)?RM:IDLE;
                default state<=IDLE;
            endcase
        end
    end
//地址发送成功后拉低请求信号
    wire cache_data_addr_ok;
    reg send_req=1'b1;
    assign cache_data_addr_ok=cache_data_req & cache_addr_ok;

    always@(posedge clk or posedge rst)
    begin
        if(rst)
            send_req<=1'b1;
        else begin
            send_req<=(cache_data_addr_ok)?1'b0:
                        (cache_data_data_ok)?1'b1:send_req;
        end
    end
//写缓冲区
    reg store_dirty;
    reg store_write;
    reg [31:0] store_buffer;
    reg [31:0] store_addr;
    wire [INDEX_WIDTH-1:0] store_index;
    wire [TAG_WIDTH-1:0] store_tag;

    assign store_tag=store_addr[31:INDEX_WIDTH+OFFSET_WIDTH];
    assign store_index=store_addr[INDEX_WIDTH+OFFSET_WIDTH-1:OFFSET_WIDTH];
```

```
assign store_offset=store_addr[OFFSET_WIDTH-1:0];

always@(negedge clk)
begin
    if(state==RM)
    begin
        if(cpu_data_wr)
        begin
            store_buffer<=rwrite_data;
            store_dirty<=1'b1;
        end
        else begin
            store_buffer<=cache_data_rdata;
            store_dirty<=1'b0;
        end
        store_write<=cache_data_data_ok;
        store_target<=replace[store_index];
    end
    else begin
        if(cpu_data_req)
        begin
            store_buffer<=write_data;
            store_addr<=cpu_data_addr;
            if(hit)
            store_target<=target;
        end
        store_dirty<=1'b1;
        store_write<=cpu_data_req & cpu_data_wr & hit;
    end
end

integer j,k;
always@(posedge clk or posedge rst)
begin
    if(rst)
    begin
        store_buffer<=32'b0;
        for(j=0;j<CACHE_DEEPTH;j=j+1)
        for(k=0;k<GROUP_WIDTH;k=k+1)
        begin
            cache_valid[j][k]<=0;
            cache_dirty[j][k]<=0;
            cache_tag[j][k]<=0;
        end
    end
    else if(store_write)
    begin
        cache_dirty[store_index][store_target]<=store_dirty;
        cache_valid[store_index][store_target]<=1'b1;
```

```
                cache_tag[store_index][store_target]<=store_tag;
                cache_block[store_index][store_target]<=store_buffer;
            end
        end
    //写回缓冲区
        reg [31:0] write_buffer;
        reg [31:0] write_addr;

        always@(negedge clk or posedge rst)
        begin
            if(rst)
            begin
                write_buffer<=32'b0;
                write_addr<=32'b0;
            end
            else if((state==RM) & cache_dirty[store_index][store_target])
            begin
                write_buffer<=cache_block[store_index][store_target];
                write_addr<={cache_tag[store_index][store_target],store_index,2'b00
                    };
            end
        end
    //MIPS Core接口
        assign cpu_data_addr_ok=(cpu_data_req & hit) | cache_data_addr_ok;
        assign cpu_data_data_ok=(cpu_data_req & hit) | ((state==RM) &
            cache_data_data_ok);
        assign cpu_data_rdata=(hit)?cache_block[index][store_target]:
            cache_data_rdata;
    //AXI接口
        assign cache_data_req=(state!=IDLE) & send_req;
        assign cache_data_wr=(state==WB);
        assign cache_data_wdata=write_buffer;
        assign cache_data_size=(state==WB)?2'b10:cpu_data_size;
        assign cache_data_addr=(state==WB)?write_addr:cpu_data_addr;
    endmodule
```

D 伪 LRU 代码

```
module PseudoLRU(
    input clk,rst,enable,
    input [2:0] path,
    output [2:0] replace
    );

    reg root;
    reg [1:0] inner;
    reg [3:0] leaf;
```

```
    always@(negedge clk)
    begin
        if(rst)
        begin
            root<=1'b0;
            inner<=2'b0;
            leaf<=4'b0;
        end
        else if(enable)
        begin
            {root,inner[path[2]],leaf[path[2:1]]}<=~path;
        end
    end

    assign replace[2]=root;
    assign replace[1]=inner[replace[2]];
    assign replace[0]=leaf[replace[2:1]];
endmodule
```