《计算机组成原理》实验报告

年级、专业、班级	2021 级计算机科学与技术 04、05 班	**姓名**	胡鑫、冯宇馨
实验题目	实验五 Cache 设计与实现		
实验时间	2023 年 5 月 14 日	**实验地点**	DS1410
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: 冯永

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: 2023 年 5 月 23 日

1 实验任务

(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 Cache 模块

2.1.1 功能描述

当 MIPS core 向 Cache 模块请求指令和数据时，Cache 模块如果命中，则可以马上返回数据，不用再访问内存，否则需要访问内存。而访问内存时，Cache 模块只需产生类 sram 信号，该类 sram 信号经过一个类 sram-axi 转换桥后，会被转换成 axi 信号，因此 CPU 顶层对外接口为 axi 接口。

2.1.2 接口定义

表 1: 接口定义

信号名	方向	位宽	功能描述
cpu_data_req	input	1-bit	cpu->cache 的读写信号,1 为有读写请求
cpu_data_wr	input	1-bit	cpu->cache, 请求是否是写请求。一直保持电平状态直到请求成功
cpu_data_size	input	2-bit	结合地址最低两位,确定数据的有效字节
cpu_data_addr	input	32-bit	cpu->cache 的这次请求的地址
cpu_data_wdata	input	32-bit	cpu->cache 的写入的数据
cpu_data_rdata	output	32-bit	cache->cpu 的返回的读数据
cpu_data_addr_ok	output	1-bit	cache->cpu 该次请求的地址传输 OK, 读:地址被接收;写:地址和数据被接收
cpu_data_data_ok	output	1-bit	cache->cpu 该次请求的数据传输 OK, 读:数据返回;写:数据写入完成
cache_data_req	output	1-bit	cache->mem, 是不是数据请求。
cache_data_wr	output	1-bit	cache->mem, 当前请求是否是写请求。一直保持电平状态直到请求成功
cache_data_size	output	2-bit	cache_data_size=cpu_data_size; 结合地址最低两位,确定数据的有效字节
cache_data_addr	output	32-bit	cache->mem 的这次请求的地址
cache_data_wdata	output	32-bit	cache->mem 的写入的数据
cache_data_rdata	input	32-bit	mem->cache 返回的读数据
cache_data_addr_ok	input	1-bit	mem->cache 该次请求的地址传输 OK, 读: 地址被接收;写:地址和数据被接收
cache_data_data_ok	input	1-bit	mem->cache 该次请求的数据传输 OK, 读: 数据返回;写:数据写入完成

3 实验过程记录

3.1 实验设计

3.1.1 d_cache_wb 的设计

1. 当 CPU 发起读操作时，首先会检查 cache 中是否有对应的数据，如果有，直接从 cache 中读取数据返回给 CPU；如果没有，则向内存发起读操作，将读取到的数据存储在 cache 中，再将数据返回给 CPU。
2. 当 CPU 发起写操作时，先向 cache 写入数据，如果该数据对应的 cache line 已被修改过（dirty），则向内存发起一个写回操作，将修改后的数据写回内存。如果该数据对应的 cache line 未被修改过，则直接将修改后的数据存储在 cache 中即可。
3. 当有缓存读写操作时，使用 wire 和 reg 定义读写信号，通过 always@(posedge clk) 来触发状态机的状态改变，根据不同状态设置 read_req/write_req 的值，并通过 addr_rcv/waddr_rcv 的值判断地址是否接收成功，最后根据 cache 的命中情况和读写操作的完成情况，更新 cache 的状态和数据。
4. 在写入 cache 时，使用 write_mask 来设置写掩码，对需要进行更新的字进行标记，并根据具体情况进行写入，最后将 cache line 标记为 dirty，表示该行已被修改过。

3.1.2 d_cache_wb_2way 的设计

1. 读取内存，当有一个读请求时，cache 向内存发出请求，接收到地址后，将状态设置为 RM，等待数据返回。当数据返回成功后，将 Cache line 置为有效，将数据写入 Cache。读请求成功（read_finish）后，需要将地址的 tag 和 index 保存下来，因为在之后的操作中可能会修改地址的值。
2. 写入内存，当有一个写请求时，cache 向内存发出请求，接收到地址和数据后，将状态设置为 WM。在完成写操作之前，需要等待数据返回成功（write_finish），防止写入过程中被新的操作覆盖。写入完成后，如果写入的 Cache line 已经存在，就将其标记为 dirty；否则将其标记为有效，并将数据写入 Cache。
3. Cache->CPU，当需要从 Cache 读取数据时，如果该 Cache line 已经标记为有效（hit），就直接从 Cache 中读取数据；否则需要向内存发起请求，并等待数据返回成功。
4. Cache->Memory，当有一个读请求或写请求时，cache 向内存发出请求，并等待对方接受（addr_rcv 或 waddr_rcv），如果地址和数据都接受成功，就将 cache 的请求标记为有效（read_req 或 write_req）。
5. 更新 Cache，当一个读请求或写请求完成（read_finish 或 isIDLE write (hit | inRM)）时，根据具体情况更新 Cache。如果是读请求完成，就将 Cache line 置为有效、清除 dirty 标记，并将从内存读取的数据写入 Cache；如果是写请求完成，根据该 Cache line 是否存在，标记为 dirty 或有效，并将新的数据写入 Cache。在对 Cache 进行写入操作时，无论是命中还是未命中，都需要回到 IDLE 状态才能继续写入，因此需要 isIDLE 的判断。在更新 Cache line 使用的是 Cache line 的 index 而不是 tag+index 组合，因为此时 tag 和 index 的值可能已经修改，不能直

接使用。

6. 更新 Cache, 当一个读请求或写请求完成(read_finish 或 isIDLE write (hit | inRM))时, 根据具体情况更新 Cache。如果是读请求完成,就将 Cache line 置为有效、清除 dirty 标记,并将从内存读取的数据写入 Cache; 如果是写请求完成, 根据该 Cache line 是否存在, 标记为 dirty 或有效,并将新的数据写入 Cache。在对 Cache 进行写入操作时,无论是命中还是未命中,都需要回到 IDLE 状态才能继续写入, 因此需要 isIDLE 的判断。在更新 Cache line 使用的是 Cache line 的 index 而不是 tag+index 组合,因为此时 tag 和 index 的值可能已经修改,不能直接使用。

3.1.3 d_cache_wb_4way_fLRU 的设计

1. 根据伪 LRU 算法原理, 当需要替换某一组的数据时, 需要将最久未使用或未被访问的路使用权选择为新的数据写入。
2. 在 read 状态下, 如果命中, 则将该路选择为最近使用过, 如果未命中, 则选择最久未使用的路进行替换; 在 write 状态下, 如果命中, 将该路的脏位置为 1, 并将数据写入; 如果未命中, 则选择最久未使用的路进行替换,并将该路的脏位置为 1。
3. 在 write 状态下,如果命中,将该路的脏位置为 1,并将数据写入;如果未命中,则选择最久未使用的路进行替换,并将该路的脏位置为 1。
4. 在写状态下也可能出现 read_finish 的情况(例如在 write 状态下, 同时需要读取一段数据)。此时需要先进行读操作,然后再进行写操作。

3.2 问题以及解决方案

问题描述:在做实验的时候, 由于教学进度还没有到 cache 章节, 我们对 cache 的设计很不熟悉

解决方案: 阅读实验指导书, 在网站上找相关的视频和文章学习理论知识, 组内成员积极进行交流讨论

问题描述:仿真结果未出现 PASS

解决方案:仔细阅读调试步骤,发现是没发现 run all 按钮,一直在点击 relaunch 按钮

4 实验结果及分析

4.1 实现写回策略的 Cache 的运行结果

图 1: 控制台打印输出—Pass!!!

4.2 实现 2 路组相联的 Cache 的运行结果

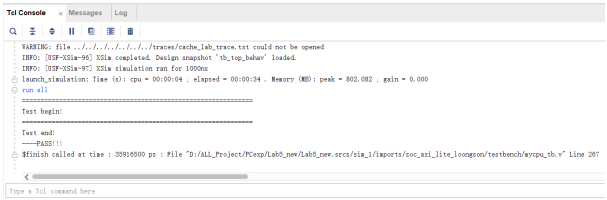

图 2: 控制台打印输出—Pass!!!

4.3 使用伪 LRU 替换策略实现 4 路组相联的 Cache 的运行结果

图 3: 控制台打印输出—Pass!!!

A 实现写回策略的 Cache 代码

```
module d_cache_wb (
    input wire clk, rst,
    //mips core
    input         cpu_data_req     , // cpu->cache的读写信号, 1为有读写请求
    input         cpu_data_wr      , // cpu->cache,请求是否是写请求。一直保持电平状
        态直到请求成功
    input  [1 :0] cpu_data_size    , //结合地址最低两位, 确定数据的有效字节
    input  [31:0] cpu_data_addr    , // cpu->cache的这次请求的地址
    input  [31:0] cpu_data_wdata   , // cpu->cache的写入的数据
    output [31:0] cpu_data_rdata   , // cache->cpu的返回的读数据
    output        cpu_data_addr_ok , // cache->cpu该次请求的地址传输OK, 读: 地址被
        接收; 写: 地址和数据被接收
    output        cpu_data_data_ok , // cache->cpu该次请求的数据传输OK, 读: 数据返
        回; 写: 数据写入完成

    //axi interface
    output         cache_data_req     , //cache->mem, 是不是数据请求。
    output         cache_data_wr      , //cache->mem, 当前请求是否是写请求。一直保
        持电平状态直到请求成功
```

```
    output  [1 :0] cache_data_size    ,
    output  [31:0] cache_data_addr    , //cache->mem的这次请求的地址
    output  [31:0] cache_data_wdata   , //cache->mem的写入的数据
    input   [31:0] cache_data_rdata   , //mem->cache返回的读数据
    input          cache_data_addr_ok , //mem->cache该次请求的地址传输OK, 读: 地址
        被接收; 写: 地址和数据被接收
    input          cache_data_data_ok   //mem->cache该次请求的数据传输OK, 读: 数据
        返回; 写: 数据写入完成
);

    //使用LUT方法实现存储
    //Cache配置
    parameter  INDEX_WIDTH  = 10, OFFSET_WIDTH = 2;
    localparam TAG_WIDTH    = 32 - INDEX_WIDTH - OFFSET_WIDTH;
    localparam CACHE_DEEPTH = 1 << INDEX_WIDTH;

    //Cache存储单元
    //使用reg定义二维数组的方式实现存储
    reg                  cache_valid [CACHE_DEEPTH - 1 : 0]; //有效位
    reg                  cache_dirty [CACHE_DEEPTH - 1 : 0]; //修改位
    reg [TAG_WIDTH-1:0]  cache_tag   [CACHE_DEEPTH - 1 : 0]; //标记存储体
    reg [31:0]           cache_block [CACHE_DEEPTH - 1 : 0]; //数据存储体

    //访问地址分解
    wire [OFFSET_WIDTH-1:0] offset;
    wire [INDEX_WIDTH-1:0] index;
    wire [TAG_WIDTH-1:0] tag;

    assign offset = cpu_data_addr[OFFSET_WIDTH - 1 : 0];
    assign index = cpu_data_addr[INDEX_WIDTH + OFFSET_WIDTH - 1 : OFFSET_WIDTH];
    assign tag = cpu_data_addr[31 : INDEX_WIDTH + OFFSET_WIDTH];

    //访问Cache line
    wire c_valid;
    wire c_dirty;
    wire [TAG_WIDTH-1:0] c_tag;
    wire [31:0] c_block;

   //根据index确定cache的某一行或某几行
    assign c_valid = cache_valid[index];
    assign c_dirty = cache_dirty[index];
    assign c_tag   = cache_tag  [index];
    assign c_block = cache_block[index];

    //判断是否命中
    wire hit, miss;
    assign hit = c_valid & (c_tag == tag);  //cache line的valid位为1, 且tag与地址中
        tag相等
    assign miss = ~hit;//不命中
```

```
//读或写
wire read, write;
assign write = cpu_data_wr; //写请求
assign read =~write; // 不是写就是读

//cache当前位置是否修改
wire dirty, clean;
assign dirty = c_dirty; //修改, 脏就要换出
assign clean = ~dirty;  //不修改

//FSM状态机
//IDLE:空闲状态
//RM:读取内存状态
//WM: 写内存状态
parameter IDLE = 2'b00, RM = 2'b01, WM = 2'b11;
reg [1:0] state;
reg inRM; //用于写cache时, 发生写缺失且dirty的情况, 需要先RM再到IDLE再回写cache
    , 所以置一位来判断之前是否是在RM

always @(posedge clk) begin
    if(rst) begin
        state <= IDLE;
        inRM <= 1'b0;
    end
    else begin
        case(state)
            IDLE:
            begin
                state <= cpu_data_req & read & hit? IDLE:
                    //读命中直接读取对应的cache的数据
                        cpu_data_req & read & miss? (clean? RM : WM):
                            //读缺失, clean:从内存读取数据, 并写入cache;dirty:
                            首先要将这个cache line的脏数据写入到内存中, 等待写
                            请求处理完成后, 再发送读请求
                        cpu_data_req & write & hit? IDLE:
                            //写命中, clean:直接写cache line, drity位置1; dirty
                            : 直接写cache line
                        cpu_data_req & write & miss? (clean? IDLE:WM): IDLE;
                            //写缺失, clean:直接写cache line; dirty: 先将cache
                            的数据写进内存
                inRM <= 1'b0;
            end
            WM:   state <= cache_data_data_ok ? IDLE : WM;       //写入内存数据完
                成时, 返回00, 否则继续

            RM:
            begin
```

```
                    state <= cache_data_data_ok ? IDLE : RM; //读取内存数据完成时，返回00，否则继续
                    inRM <= 1'b1;
                end
            endcase
        end
    end
//读内存
    //变量 read_req，addr_rcv，read_finish用于构造类sram信号。
    wire read_req;          //是否有cache->mem的读请求
    reg addr_rcv;           //地址接收成功(addr_ok)后到结束
    wire read_finish;       //数据接收成功(data_ok)，即读请求结束
    always @(posedge clk) begin
        addr_rcv <= rst ? 1'b0 :
                    cache_data_req & read & cache_data_addr_ok ? 1'b1 : //地址接收成功
                    read_finish ? 1'b0 :
                    addr_rcv;
    end
    assign read_req = state==RM;
    assign read_finish = read & cache_data_data_ok;    //数据返回完成

    //写内存
    wire write_req;     //是否有cache->mem的写请求
    reg waddr_rcv;
    wire write_finish;
    always @(posedge clk) begin
        waddr_rcv <= rst ? 1'b0 :
                     cache_data_req& write & cache_data_addr_ok ? 1'b1 :  //地址和数据接收成功
                     write_finish ? 1'b0 :
                     waddr_rcv;
    end
    assign write_req = state==WM;
    assign write_finish = write & cache_data_data_ok;       //数据写入完成

    //cache->cpu
    assign cpu_data_rdata   = hit ? c_block : cache_data_rdata;  //cache->cpu读的数据，命中则来自于cache的数据块，不命中从主存中读
    assign cpu_data_addr_ok = cpu_data_req & hit | cache_data_req &
        cache_data_addr_ok;
    assign cpu_data_data_ok = cpu_data_req & hit | cache_data_data_ok;

    // cache->mem
    assign cache_data_req   = read_req & ~addr_rcv | write_req & ~waddr_rcv; //有请求且地址（和数据）未被mem接受
    assign cache_data_wr    = write_req;  //只要处在WM状态就一直有写请求
    assign cache_data_size  = cpu_data_size;
```

```
assign cache_data_addr  = cache_data_wr ? {c_tag, index, offset}:  //写内存，写
    回mem的地址为原cache line对应的地址
                                          cpu_data_addr;            //读内存，对
                                              应mem的地址为cpu_data_addr
assign cache_data_wdata = c_block;     //写回mem的数据是原来cache line的数据

//写入Cache
//保存地址中的tag, index, 防止addr发生改变
reg [TAG_WIDTH-1:0] tag_save;
reg [INDEX_WIDTH-1:0] index_save;
always @(posedge clk) begin
    tag_save   <= rst ? 0 :
                  cpu_data_req ? tag : tag_save;
    index_save <= rst ? 0 :
                  cpu_data_req ? index : index_save;
end

wire [31:0] write_cache_data;
wire [3:0] write_mask;

//根据地址低两位和size，生成写掩码（针对sb, sh等不是写完整一个字的指令），4位对
    应1个字（4字节）中每个字的写使能
assign write_mask = cpu_data_size==2'b00 ?
                        (cpu_data_addr[1] ? (cpu_data_addr[0] ? 4'b1000 : 4'
                            b0100):
                                            (cpu_data_addr[0] ? 4'b0010 : 4'
                                                b0001)) :
                        (cpu_data_size==2'b01 ? (cpu_data_addr[1] ? 4'b1100 :
                            4'b0011) : 4'b1111);

//掩码的使用：位为1的代表需要更新的。
//位拓展：{8{1'b1}} -> 8'b11111111
//new_data = old_data & ~mask | write_data & mask 旧数据不更新或写进去的数据
assign write_cache_data = cache_block[index] & ~{{8{write_mask[3]}}, {8{
    write_mask[2]}}, {8{write_mask[1]}}, {8{write_mask[0]}}} |
                            cpu_data_wdata & {{8{write_mask[3]}}, {8{write_mask
                                [2]}}, {8{write_mask[1]}}, {8{write_mask[0]}}};



integer t;
wire isIDLE;  //写cache时，不论write命中与否，都要回到IDLE状态才能写入
assign isIDLE = state==IDLE;
always @(posedge clk) begin
    if(rst) begin
        for(t=0; t<CACHE_DEEPTH; t=t+1) begin    //刚开始将Cache置为无效 //*
            dirty 为 0
            cache_valid[t] <= 0;
            cache_dirty[t] <= 0;
```

```
            end
        end
        else begin
            if(read_finish) begin //读请求完成后
                cache_valid[index_save] <= 1'b1;              //将Cache line置为有效
                cache_dirty[index_save] <= 1'b0;              // 读取内存的数据后，一
                    定是clean，未被修改的
                cache_tag  [index_save] <= tag_save;
                cache_block[index_save] <= cache_data_rdata; //写入Cache line
            end
            else if (write & isIDLE & (hit | inRM))
              begin
                cache_dirty[index] <= 1'b1; // 修改，置1
                cache_block[index] <= write_cache_data;       //写入Cache line，使用
                    index而不是index_save
            end
        end
    end
endmodule
```

B 实现 2 路组相联的 Cache 代码

```
module d_cache_wb_2way (
    input wire clk, rst,
    //mips core
    input          cpu_data_req     , // 是不是数据请求(load 或 store指令)。一个周期
        后就清除了
    input          cpu_data_wr      , // mips->cache，当前请求是否是写请求(是不是
        store指令)。一直保持电平状态直到请求成功
    input  [1 :0] cpu_data_size     , // 结合地址最低两位，确定数据的有效字节（用于
        sb、sh等指令）
    input  [31:0] cpu_data_addr     , // cpu->cache的这次请求的地址
    input  [31:0] cpu_data_wdata    , // cpu->cache的写入的数据
    output [31:0] cpu_data_rdata    , // cache->cpu的返回的读数据
    output         cpu_data_addr_ok , // cache->cpu该次请求的地址传输OK，读：地址被
        接收；写：地址和数据被接收
    output         cpu_data_data_ok , // cache->cpu该次请求的数据传输OK，读：数据返
        回；写：数据写入完成

    //axi interface
    output          cache_data_req     , // cache->mem，是不是数据请求
    output          cache_data_wr      , // cache->mem，当前请求是否是写请求(是不是
        处于WM状态)。一直保持电平状态直到请求成功
    output  [1 :0] cache_data_size    ,
    output  [31:0] cache_data_addr    , //cache->mem的这次请求的地址
    output  [31:0] cache_data_wdata   , //cache->mem的写入的数据
    input   [31:0] cache_data_rdata   , //mem->cache返回的读数据
```

```
    input           cache_data_addr_ok , // mem->cache该次请求的地址传输OK, 读: 地址
        被接收; 写: 地址和数据被接收
    input           cache_data_data_ok   //mem->cache该次请求的数据传输OK, 读: 数据
        返回; 写: 数据写入完成
);
    //Cache配置
    parameter  INDEX_WIDTH  = 10, OFFSET_WIDTH = 2;
    localparam TAG_WIDTH    = 32 - INDEX_WIDTH - OFFSET_WIDTH;
    localparam CACHE_DEEPTH = 1 << INDEX_WIDTH;

    //Cache存储单元
    //两路, 有两个cache
    reg                  cache_valid [CACHE_DEEPTH - 1 : 0][1:0]; //有效位
    reg                  cache_dirty [CACHE_DEEPTH - 1 : 0][1:0]; //修改位
    reg                  cache_used    [CACHE_DEEPTH - 1 : 0][1:0]; //使用标记位
    reg [TAG_WIDTH-1:0]  cache_tag   [CACHE_DEEPTH - 1 : 0][1:0]; //标记存储体
    reg [31:0]           cache_block [CACHE_DEEPTH - 1 : 0][1:0]; //数据存储体

    //访问地址分解
    wire [OFFSET_WIDTH-1:0] offset;
    wire [INDEX_WIDTH-1:0] index;
    wire [TAG_WIDTH-1:0] tag;

    assign offset = cpu_data_addr[OFFSET_WIDTH - 1 : 0];
    assign index = cpu_data_addr[INDEX_WIDTH + OFFSET_WIDTH - 1 : OFFSET_WIDTH];
    assign tag = cpu_data_addr[31 : INDEX_WIDTH + OFFSET_WIDTH];

    //访问Cache line
    wire                 c_valid[1:0];
    wire                 c_dirty[1:0];
    wire                 c_used   [1:0]; //recently used
    wire [TAG_WIDTH-1:0] c_tag  [1:0];
    wire [31:0]          c_block[1:0];

    //根据index确定cache的某一行或某几行
    assign c_valid[0] = cache_valid[index][0];
    assign c_valid[1] = cache_valid[index][1];
    assign c_dirty[0] = cache_dirty[index][0];
    assign c_dirty[1] = cache_dirty[index][1];
    assign c_used   [0] = cache_used   [index][0];
    assign c_used   [1] = cache_used   [index][1];
    assign c_tag  [0] = cache_tag  [index][0];
    assign c_tag  [1] = cache_tag  [index][1];
    assign c_block[0] = cache_block[index][0];
    assign c_block[1] = cache_block[index][1];

    //判断是否命中
    wire hit,hit0,hit1,miss;
    //0路是否命中
```

```
assign hit0=c_valid[0] & (c_tag[0] == tag);//cache line中0路的valid位为1, 且tag
    与地址中tag相等
//1路是否命中
assign hit1=c_valid[1] & (c_tag[1] == tag);//cache line中1路的valid位为1, 且tag
    与地址中tag相等
assign hit = hit0| hit1 ; //0或1路有一路命中即可
assign miss = ~hit;

//后面的cache处理应访问哪一路
wire c_way;
//如果hit, 则way没有意义, 若miss, 则way表示分配那路
assign c_way = hit ? (hit0 ? 1'b0 : 1'b1) :
               c_used[0] ? 1'b1 : 1'b0;

// cpu指令读或写
wire read, write;
assign write = cpu_data_wr;
assign read = cpu_data_req & ~write; //是数据读写请求, 不是写就是读

//cache当前位置是否修改
wire dirty, clean;
assign dirty = c_dirty[c_way];//修改, 脏就要换出
assign clean = ~dirty;//未修改

//FSM状态机
//IDLE:空闲状态
//RM:读取内存状态
//WM:写内存状态
parameter IDLE = 2'b00, RM = 2'b01, WM = 2'b11;
reg [1:0] state;
reg inRM;//用于写cache时, 发生写缺失且dirty的情况, 需要先RM再到IDLE再写回cache
    , 所以置一位来判断之前是否是在RM

always @(posedge clk) begin
    if(rst) begin
        state <= IDLE;
        inRM <= 1'b0;
    end
    else begin
        case(state)
            IDLE: begin
                state <= cpu_data_req & read & hit ? IDLE: //读命中直接读取对应
                    的cache的数据
                      cpu_data_req & read & miss ? (clean ? RM : WM):
                      //读缺失, clean:从内存读取数据, 并写入cache;
                      //dirty:首先要将这个cache line的脏数据写入到内存中, 等
                          待写请求处理完成后, 再发送读请求
                      cpu_data_req & write & hit ? IDLE://写命中, clean:直接
                          写cache line, drity位置1; dirty: 直接写cache line
```

```
                        cpu_data_req & write & miss ? (clean? IDLE : WM):IDLE;
                            //写缺失, clean:直接写cache line; dirty: 先将cache
                            的数据写进内存
                    inRM <= 1'b0;
                end

                WM:  state <= cache_data_data_ok ? IDLE : WM;      //写入内存数据完
                    成时, 返回00, 否则继续

                RM: begin
                    state <= cache_data_data_ok ? IDLE : RM; //读取内存数据完成时,
                        返回00, 否则继续
                    inRM <= 1'b1;
                end
            endcase
        end
    end

    //读内存
    //变量 read_req, addr_rcv, read_finish用于构造类sram信号。
    wire read_req;      //是否有cache->mem的读请求
    reg addr_rcv;       //地址接收成功(addr_ok)后到结束
    wire read_finish;   //数据接收成功(data_ok), 即读请求结束
    always @(posedge clk) begin
        addr_rcv <= rst ? 1'b0 :
                    cache_data_req & read & cache_data_addr_ok ? 1'b1 :
                    read_finish ? 1'b0 :
                    addr_rcv;
    end
    assign read_req = state==RM;
    assign read_finish = read & cache_data_data_ok;//数据返回完成

    //写内存
    wire write_req;      //是否有cache->mem的写请求
    reg waddr_rcv;
    wire write_finish;
    always @(posedge clk) begin
        waddr_rcv <= rst ? 1'b0 :
                     cache_data_req & write & cache_data_addr_ok ? 1'b1 ://地址和数
                         据接收成功
                     write_finish ? 1'b0 :
                     waddr_rcv;
    end
    assign write_req = state==WM;
    assign write_finish = write & cache_data_data_ok;//数据写入完成

    //cache->cpu
    assign cpu_data_rdata   = hit ? c_block[c_way] : cache_data_rdata;//cache->cpu
        读的数据, 命中则来自于cache的数据块, 不命中从主存中读
```

```
assign cpu_data_addr_ok = cpu_data_req & hit | cache_data_req & read_req &
    cache_data_addr_ok;
assign cpu_data_data_ok = cpu_data_req & hit | read_req & cache_data_data_ok;

//output to axi interface
// cache->mem
assign cache_data_req   = read_req & ~addr_rcv | write_req & ~waddr_rcv;//有请
    求且地址（和数据）未被mem接受
assign cache_data_wr    = write_req;//只要处在WM状态就一直有写请求
assign cache_data_size  = cpu_data_size;
assign cache_data_addr  = cache_data_wr ? {c_tag[c_way], index, offset}://写，
    写回mem的地址为原cache line对应的地址(一路)
                                          cpu_data_addr;//读，对应mem的地址为
                                              cpu_data_addr
assign cache_data_wdata = c_block[c_way];//写回mem的数据是原来cache line（一
    路）的数据

//写入Cache
//保存地址中的tag，index，防止addr发生改变
reg [TAG_WIDTH-1:0] tag_save;
reg [INDEX_WIDTH-1:0] index_save;
always @(posedge clk) begin
    tag_save   <= rst ? 0 :
                  cpu_data_req ? tag : tag_save;
    index_save <= rst ? 0 :
                  cpu_data_req ? index : index_save;
end

wire [31:0] write_cache_data;
wire [3:0] write_mask;

//根据地址低两位和size，生成写掩码（针对sb，sh等不是写完整一个字的指令），4位对
    应1个字（4字节）中每个字的写使能
assign write_mask = cpu_data_size==2'b00 ?
                        (cpu_data_addr[1] ? (cpu_data_addr[0] ? 4'b1000 : 4'
                            b0100):
                                            (cpu_data_addr[0] ? 4'b0010 : 4'
                                                b0001)) :
                        (cpu_data_size==2'b01 ? (cpu_data_addr[1] ? 4'b1100 :
                            4'b0011) : 4'b1111);

//掩码的使用：位为1的代表需要更新的。
//位拓展：{8{1'b1}} -> 8'b11111111
//new_data = old_data & ~mask | write_data & mask
assign write_cache_data = cache_block[index][c_way] & ~{{8{write_mask[3]}}, {8{
    write_mask[2]}}, {8{write_mask[1]}}, {8{write_mask[0]}}} |
                          cpu_data_wdata & {{8{write_mask[3]}}, {8{write_mask
                              [2]}}, {8{write_mask[1]}}, {8{write_mask[0]}}};
```

```
    integer t, way;
    wire isIDLE;//写cache时，不论write命中与否，都要回到IDLE状态才能写入
    assign isIDLE = state==IDLE;
    always @(posedge clk) begin
        if(rst) begin
        //reset
            for(t=0; t<CACHE_DEEPTH; t=t+1) begin
                for (way = 0; way<2; way=way+1) begin
                    cache_valid[t][way] <= 0;
                    cache_dirty[t][way] <= 0;
                    cache_used [t][way] <= 0;
                end
            end
        end
        else begin
            if(read_finish) begin //读请求完成后
                cache_valid[index_save][c_way] <= 1'b1;              //将Cache line
                    置为有效
                cache_dirty[index_save][c_way] <= 1'b0; // 读取内存的数据后，一定是
                    clean
                cache_tag  [index_save][c_way] <= tag_save;
                cache_block[index_save][c_way] <= cache_data_rdata; //将从mem读到数
                    据写入Cache
            end
            else if (write & isIDLE & (hit | inRM)) begin
                cache_dirty[index][c_way] <= 1'b1; // 改了数据，变dirty
                cache_block[index][c_way] <= write_cache_data;       //写入Cache
                    line，使用index而不是index_save
            end
            //c_way的写入，一路使用了,一路就未使用
            if (read & isIDLE & (hit | inRM)) begin
                //读cache：hit直接IDLE;miss,clean:RM,dirty:WM->RM 上一状态都是RM,
                    所以要求inRM
                cache_used[index][c_way]   <= 1'b1;
                cache_used[index][1-c_way] <= 1'b0;
            end
            else if(write & isIDLE & (hit |inRM)) begin
            //写cache：hit直接IDLE;miss,clean:覆盖,dirty:RM
                 cache_used[index][c_way]   <= 1'b1;
                cache_used[index][1-c_way] <= 1'b0;
            end
        end
    end
endmodule
```

C 使用伪 LRU 替换策略实现的 4 路组相联的 Cache 代码

```
module d_cache (
    input wire clk, rst,
    //mips core
    input          cpu_data_req     , // cpu->cache的读写信号, 1为有读写请求
    input          cpu_data_wr      , // cpu->cache,请求是否是写请求。一直保持电平状
        态直到请求成功
    input  [1 :0] cpu_data_size     , //结合地址最低两位, 确定数据的有效字节
    input  [31:0] cpu_data_addr     , // cpu->cache的这次请求的地址
    input  [31:0] cpu_data_wdata    , // cpu->cache的写入的数据
    output [31:0] cpu_data_rdata    , // cache->cpu的返回的读数据
    output         cpu_data_addr_ok , // cache->cpu该次请求的地址传输OK, 读: 地址被
        接收; 写: 地址和数据被接收
    output         cpu_data_data_ok , // cache->cpu该次请求的数据传输OK, 读: 数据返
        回; 写: 数据写入完成

    //axi interface
    output          cache_data_req      , //cache->mem, 是不是数据请求。
    output          cache_data_wr       , //cache->mem, 当前请求是否是写请求。一直保
        持电平状态直到请求成功
    output  [1 :0] cache_data_size     ,
    output  [31:0] cache_data_addr     , //cache->mem的这次请求的地址
    output  [31:0] cache_data_wdata    , //cache->mem的写入的数据
    input   [31:0] cache_data_rdata    , //mem->cache返回的读数据
    input          cache_data_addr_ok , // mem->cache该次请求的地址传输OK, 读: 地址
        被接收; 写: 地址和数据被接收
    input          cache_data_data_ok   //mem->cache该次请求的数据传输OK, 读: 数据
        返回; 写: 数据写入完成
);
    //Cache配置
    parameter  INDEX_WIDTH  = 10, OFFSET_WIDTH = 2;
    localparam TAG_WIDTH    = 32 - INDEX_WIDTH - OFFSET_WIDTH;
    localparam CACHE_DEEPTH = 1 << INDEX_WIDTH;

    //Cache存储单元
    //cache分成四部分
    reg                  cache_valid [CACHE_DEEPTH - 1 : 0][3:0];//有效位
    reg                  cache_dirty [CACHE_DEEPTH - 1 : 0][3:0]; //修改位
    reg [TAG_WIDTH-1:0] cache_tag   [CACHE_DEEPTH - 1 : 0][3:0]; //标记存储体
    reg [31:0]           cache_block [CACHE_DEEPTH - 1 : 0][3:0]; //数据存储体
    reg [2:0]            cache_bit   [CACHE_DEEPTH - 1 : 0];   //(4-1)位的bits去确定
        cache line
    //* tree 为对应cache set的查找树, tree[2]为根节点, tree[1]右子树, tree[0]左子树
    wire [2:0] tree;


    //访问地址分解
    wire [OFFSET_WIDTH-1:0] offset;
    wire [INDEX_WIDTH-1:0] index;
    wire [TAG_WIDTH-1:0] tag;
```

```
    assign offset = cpu_data_addr[OFFSET_WIDTH - 1 : 0];
    assign index = cpu_data_addr[INDEX_WIDTH + OFFSET_WIDTH - 1 : OFFSET_WIDTH];
    assign tag = cpu_data_addr[31 : INDEX_WIDTH + OFFSET_WIDTH];

    //访问Cache line
    wire                    c_valid[3:0];
    wire                    c_dirty[3:0];
    wire [TAG_WIDTH-1:0]    c_tag  [3:0];
    wire [31:0]             c_block[3:0];

   //根据index确定cache line
    assign c_valid[0] = cache_valid[index][0];
    assign c_valid[1] = cache_valid[index][1];
    assign c_valid[2] = cache_valid[index][2];
    assign c_valid[3] = cache_valid[index][3];
    assign c_dirty[0] = cache_dirty[index][0];
    assign c_dirty[1] = cache_dirty[index][1];
    assign c_dirty[2] = cache_dirty[index][2];
    assign c_dirty[3] = cache_dirty[index][3];
    assign c_tag  [0] = cache_tag  [index][0];
    assign c_tag  [1] = cache_tag  [index][1];
    assign c_tag  [2] = cache_tag  [index][2];
    assign c_tag  [3] = cache_tag  [index][3];
    assign c_block[0] = cache_block[index][0];
    assign c_block[1] = cache_block[index][1];
    assign c_block[2] = cache_block[index][2];
    assign c_block[3] = cache_block[index][3];
    assign tree =  cache_bit[index];
    //判断是否命中
    wire hit, hit0,hit1,hit2,hit3,miss;
    assign hit0 = c_valid[0] & (c_tag[0] == tag);
    assign hit1 = c_valid[1] & (c_tag[1] == tag);
    assign hit2 = c_valid[2] & (c_tag[2] == tag);
    assign hit3 = c_valid[3] & (c_tag[3] == tag);
    assign hit = hit0 | hit1 | hit2 | hit3;  //查看set中是否有line命中
    assign miss = ~hit;

    //0-3路cache line
    wire [1:0] c_way;
    //如果hit，则way没意义，若miss，则way表示分配那路
    //根节点为1，表示最近访问的是0，1路；根节点为0，表示最近访问的是2，3路
    assign c_way = hit ? (hit0 ? 2'b00 :
                          hit1 ? 2'b01 :
                          hit2 ? 2'b10 : 2'b11) : tree[2] ?
                          {tree[2], tree[0]} : //next_state是2，3路
                          {tree[2], tree[1]};  //next_states是0，1路
// cpu指令读或写
    wire read, write;
```

```
assign  write = cpu_data_wr;
assign  read = ~write; // 不是写就是读

//cache当前位置是否修改
wire dirty, clean;
assign dirty = c_dirty[c_way]; //修改，脏就要换出
assign clean = ~dirty;  //不修改


//FSM状态机
//IDLE:空闲状态
//RM:读取内存状态
//WM: 写内存状态
parameter IDLE = 2'b00, RM = 2'b01, WM = 2'b11;
reg [1:0] state;
reg inRM; //用于写cache时，发生写缺失且dirty的情况，需要先RM再到IDLE再回写cache
    ，所以置一位来判断之前是否是在RM

always @(posedge clk) begin
    if(rst) begin
        state <= IDLE;
        inRM <= 1'b0;
    end
    else begin
        case(state)
            IDLE:
            begin
                state <= cpu_data_req & read & hit? IDLE:
                    //读命中直接读取对应的cache的数据
                        cpu_data_req & read & miss? (clean? RM : WM):
                            //读缺失，clean:从内存读取数据，并写入cache;dirty:
                            首先要将这个cache line的脏数据写入到内存中，等待写
                            请求处理完成后，再发送读请求
                        cpu_data_req & write & hit? IDLE:
                            //写命中，clean:直接写cache line，drity位置1；dirty
                            : 直接写cache line
                        cpu_data_req & write & miss? (clean? IDLE:WM): IDLE;
                            //写缺失，clean:直接写cache line；dirty: 先将cache
                            的数据写进内存
                inRM <= 1'b0;
            end
            WM:  state <= cache_data_data_ok ? IDLE : WM;      //写入内存数据完
                成时，返回00，否则继续

            RM:
            begin
                state <= cache_data_data_ok ? IDLE : RM; //读取内存数据完成时，
                    返回00，否则继续
                inRM <= 1'b1;
```

```
            end
        endcase
    end
end

//读内存
//变量 read_req, addr_rcv, read_finish用于构造类sram信号。
wire read_req;          //是否有cache->mem的读请求
reg addr_rcv;           //地址接收成功(addr_ok)后到结束
wire read_finish;     //数据接收成功(data_ok), 即读请求结束
always @(posedge clk) begin
    addr_rcv <= rst ? 1'b0 :
                cache_data_req & read & cache_data_addr_ok ? 1'b1 : //地址接收
                    成功
                read_finish ? 1'b0 :
                addr_rcv;
end
assign read_req = state==RM;
assign read_finish = read & cache_data_data_ok;    //数据返回完成

//写内存
wire write_req;     //是否有cache->mem的写请求
reg waddr_rcv;
wire write_finish;
always @(posedge clk) begin
    waddr_rcv <= rst ? 1'b0 :
                 cache_data_req& write & cache_data_addr_ok ? 1'b1 :  //地址和
                     数据接收成功
                 write_finish ? 1'b0 :
                 waddr_rcv;
end
assign write_req = state==WM;
assign write_finish = write & cache_data_data_ok;       //数据写入完成


//cache->cpu
assign cpu_data_rdata   = hit ? c_block[c_way] : cache_data_rdata; //cache->cpu
    读的数据, 命中则来自于cache的数据块, 不命中从主存中读
assign cpu_data_addr_ok = cpu_data_req & hit | cache_data_req & read_req &
    cache_data_addr_ok;
assign cpu_data_data_ok = cpu_data_req & hit | read_req & cache_data_data_ok;

// cache->mem
assign cache_data_req   = read_req & ~addr_rcv | write_req & ~waddr_rcv;//有请
    求且地址(和数据)未被mem接受
assign cache_data_wr    = write_req;//只要处在WM状态就一直有写请求
assign cache_data_size  = cpu_data_size;
assign cache_data_addr  = cache_data_wr ? {c_tag[c_way], index, offset}:  //
    写, 写回mem的地址为原cache line对应的地址(一路)
```

```
                                                cpu_data_addr;                          //
                                                    读，对应mem的地址为cpu_data_addr
    assign cache_data_wdata = c_block[c_way];                                           //写
        回mem的数据是原来cache line（一路）的数据
//写入Cache
    //保存地址中的tag，index，防止addr发生改变
    reg [TAG_WIDTH-1:0] tag_save;
    reg [INDEX_WIDTH-1:0] index_save;
    always @(posedge clk) begin
        tag_save   <= rst ? 0 :
                      cpu_data_req ? tag : tag_save;
        index_save <= rst ? 0 :
                      cpu_data_req ? index : index_save;
    end

    wire [31:0] write_cache_data;
    wire [3:0] write_mask;
    //根据地址低两位和size，生成写掩码（针对sb，sh等不是写完整一个字的指令），4位对
        应1个字（4字节）中每个字的写使能
    assign write_mask = cpu_data_size==2'b00 ?
                            (cpu_data_addr[1] ? (cpu_data_addr[0] ? 4'b1000 : 4'
                                b0100):
                                                (cpu_data_addr[0] ? 4'b0010 : 4'
                                                    b0001)) :
                            (cpu_data_size==2'b01 ? (cpu_data_addr[1] ? 4'b1100 :
                                4'b0011) : 4'b1111);

    //掩码的使用：位为1的代表需要更新的。
    //位拓展：{8{1'b1}} -> 8'b11111111
    //new_data = old_data & ~mask | write_data & mask
    assign write_cache_data = cache_block[index][c_way] & ~{{8{write_mask[3]}}, {8{
        write_mask[2]}}, {8{write_mask[1]}}, {8{write_mask[0]}}} |
                                cpu_data_wdata & {{8{write_mask[3]}}, {8{write_mask
                                    [2]}}, {8{write_mask[1]}}, {8{write_mask[0]}}};

    integer t, way;
    wire isIDLE;  //写cache时，不论write命中与否，都要回到IDLE状态才能写入
    assign isIDLE = state==IDLE;
    always @(posedge clk) begin
        if(rst) begin
            for(t=0; t<CACHE_DEEPTH; t=t+1) begin    //刚开始将Cache初始化为无效，
                dirty 初始化为 0
                for ( way = 0;  way<4;  way= way+1) begin
                    cache_valid[t][way] <= 0;
                    cache_dirty[t][way] <= 0;
                end
                //* tree初始化为000
                 cache_bit[t] <= 3'b000;
            end
```

```
        end
        else begin
            if(read_finish)
                begin // 处于RM状态，且已得到mem读取的数据
                  cache_valid[index_save][c_way] <= 1'b1;              //将Cache line
                      置为有效
                  cache_dirty[index_save][c_way] <= 1'b0; // 读取内存的数据后，一定是
                      clean
                  cache_tag  [index_save][c_way] <= tag_save;
                  cache_block[index_save][c_way] <= cache_data_rdata; //写入Cache
                      line
                end
            else if (write & isIDLE & (hit | inRM))
                begin
                  cache_dirty[index][c_way] <= 1'b1; // 修改，置1
                  cache_block[index][c_way] <= write_cache_data;       //写入Cache
                      line，使用index而不是index_save
                end

            //更新树的标识位
            if ( read & isIDLE & (hit | inRM))
                begin
                //读cache：hit直接IDLE;miss,clean:RM,dirty:WM->RM 上一状态都是RM，所
                    以要求inRM
                  if (c_way[1] == 1'b0)  //当前state为0，1路，则更新tree[2]和tree[1]
                      { cache_bit[index][2],  cache_bit[index][1]} <= ~c_way;
                  else  //当前state为2，3路，则更新tree[2]和tree[0]
                      { cache_bit[index][2],  cache_bit[index][0]} <= ~c_way;
                end
             else if ( write & isIDLE & (hit | inRM))
             begin
                //写cache：hit直接IDLE;miss,clean:覆盖,dirty:RM
                if (c_way[1] == 1'b0)  //当前state为0，1路，则更新tree[2]和tree[1]
                      { cache_bit[index][2],  cache_bit[index][1]} <= ~c_way;
                 else  //当前state为2，3路，则更新tree[2]和tree[0]
                      { cache_bit[index][2],  cache_bit[index][0]} <= ~c_way;
             end

        end
    end
endmodule
```