

# 《计算机组成原理》实验报告

年级、专业、班级	2023 级计算机科学与技术 06 班 2023 级计算机科学与技术 04 班 2023 级计算机科学与技术 04 班	姓名	饶格奇 刘雨霜 李隆征
实验题目	实验二 Cache 设计与实现		
实验时间	2025 年 5 月 23 日	实验地点	DS1410
实验成绩	优秀/良好/中等	实验性质	<input type="checkbox"/> 验证性 <input checked="" type="checkbox"/> 设计性 <input type="checkbox"/> 综合性
<b>教师评价：</b> <input type="checkbox"/> 算法/实验过程正确； <input type="checkbox"/> 源程序/实验内容提交； <input type="checkbox"/> 程序结构/实验步骤合理； <input type="checkbox"/> 实验结果正确； <input type="checkbox"/> 语法、语义正确； <input type="checkbox"/> 报告规范； 其他： <div>评价教师: 任骛</div>			
<b>实验目的</b> (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: 2025 年 5 月 20 日

# 1 实验内容

阅读实验原理实现以下模块：

- (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) 性能优化，实现 2 路组相联的 Cache

## 2 实验设计

### 2.1 Cache 模块

其中 i\_cache 模块由于无写功能无需变动，只需实现 d\_cache 模块的写回和写分配策略。

#### 2.1.1 功能描述

Cache 减少 CPU 访问主存的时间，提高访问速度。

#### 2.1.2 状态机的设计

1. 在读命中的情况下，CPU 直接读取对应的 cacheline 的数据；
2. 在读缺失的情况，如果索引到的 cacheline 是干净的，那么发送读请求，从内存读取数据，然后返回给 CPU，同时将数据写入到索引到的 cacheline 中；如果索引到的 cacheline 是脏的，那么首先要发送写请求，将这个 cacheline 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，从内存中读取对应的数据，然后再把数据返回给 CPU，同时将数据写入到索引的 cacheline 中。
3. 在写命中的情况下，如果索引到的 cacheline 是干净的，那么直接将数据写入到对应的 cacheline 中，并且将 dirty 位置为 1；如果索引到的 cacheline 是脏的，直接把数据写入到 cache 中。
4. 在写缺失的情况下，如果索引到的 cacheline 是干净的，那么将数据写入到 cacheline 中，覆盖掉原来的数据。如果索引到的 cacheline 是脏的，那么首先发送写请求，将脏的 cacheline 的数据更新到内存中；然后等待第一个写请求处理完成后，然后将数据写入到索引到的 cacheline 中，并且将脏位标志位置为 1；

写回-写分配策略的直接映射 Cache 流程图如图所示：

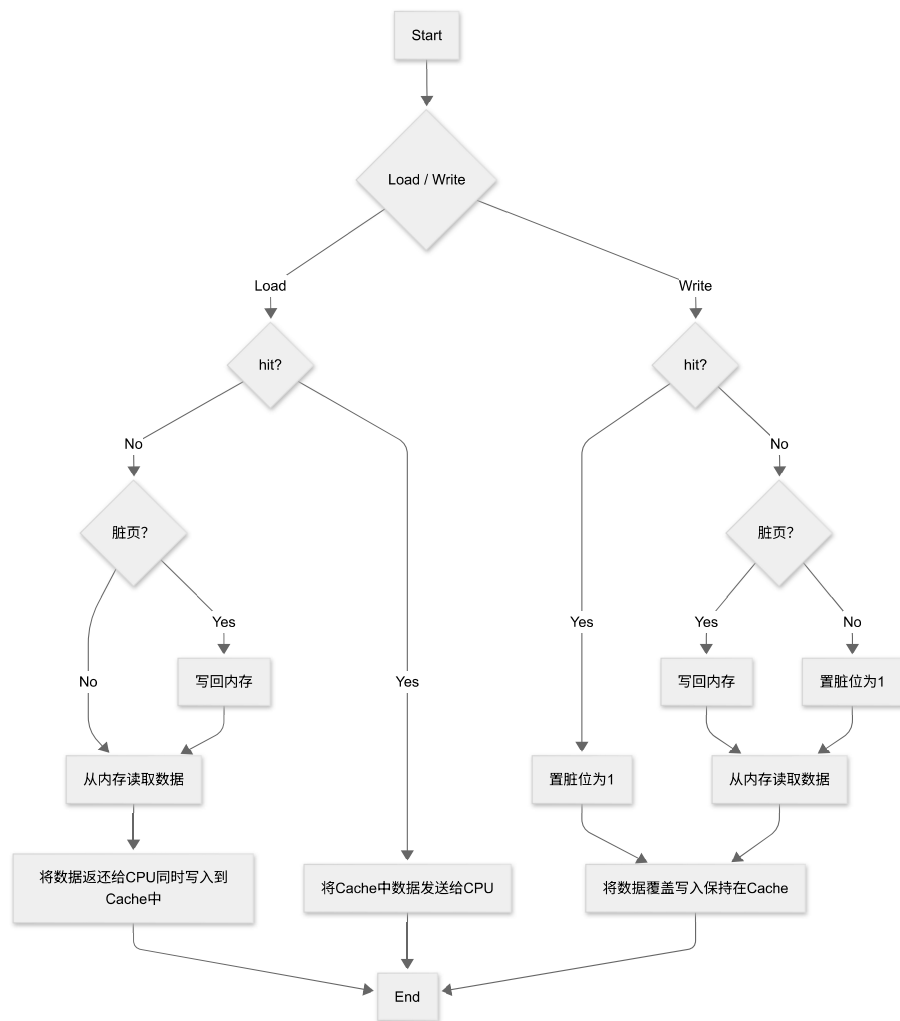


图 1: 写回-写分配策略的直接映射 Cache 流程图

写回-写分配策略的直接映射 Cache 状态图如图所示:

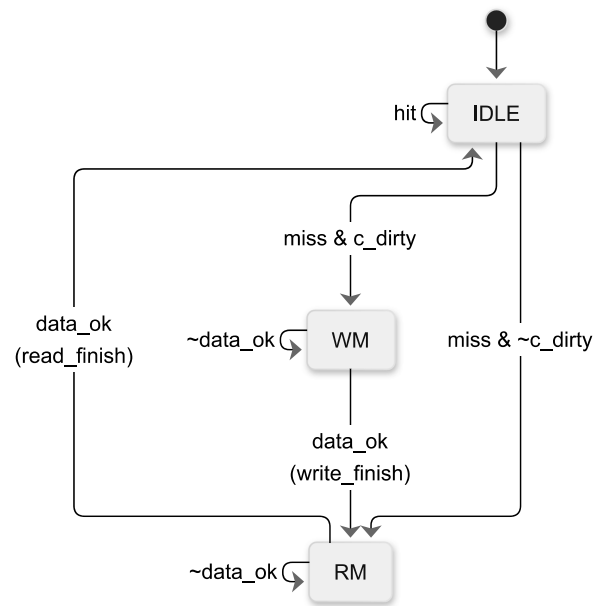


图 2: 写回-写分配策略的直接映射 Cache 状态图

### 2.1.3 接口定义

表 1: d\_cache 模块接口信号表

信号名	方向	位宽	功能描述
<b>MIPS Core 接口</b>			
cpu_data_req	input	1	MIPS CPU 请求 Cache 访问(读或写)
cpu_data_wr	input	1	写使能信号, 高电平表示写操作
cpu_data_size	input	2	访问字节数: 00=byte, 01=halfword, 10/11=word
cpu_data_addr	input	32	CPU 发出的访存地址
cpu_data_wdata	input	32	写操作时由 CPU 提供的数据
cpu_data_rdata	output	32	读操作时返回给 CPU 的数据
cpu_data_addr_ok	output	1	地址阶段握手信号, 表示地址接收成功
cpu_data_data_ok	output	1	数据返回握手信号, 表示数据有效返回
<b>AXI 接口</b>			
cache_data_req	output	1	Cache 请求主存读/写数据
cache_data_wr	output	1	高电平表示写请求, 低电平为读请求
cache_data_size	output	2	访问大小: 同上
cache_data_addr	output	32	发往主存的地址
cache_data_wdata	output	32	写回主存的数据
cache_data_rdata	input	32	从主存读取回的数据
cache_data_addr_ok	input	1	主存接受地址成功
cache_data_data_ok	input	1	主存返回数据成功

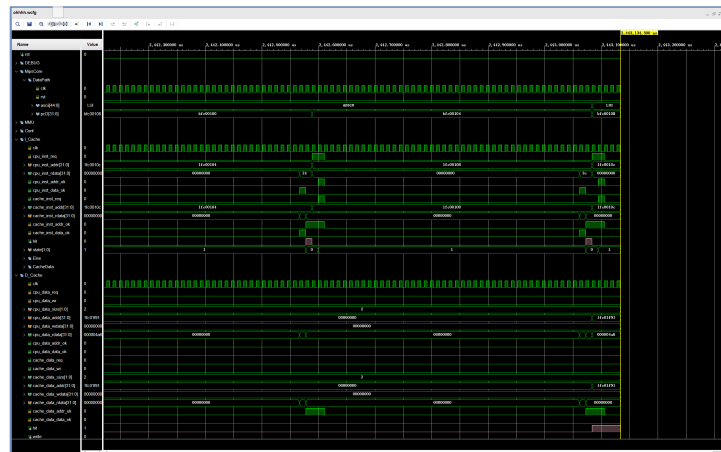
## 3 实验过程记录

### 3.1 问题 1: 仿真多数据出现不定态 X

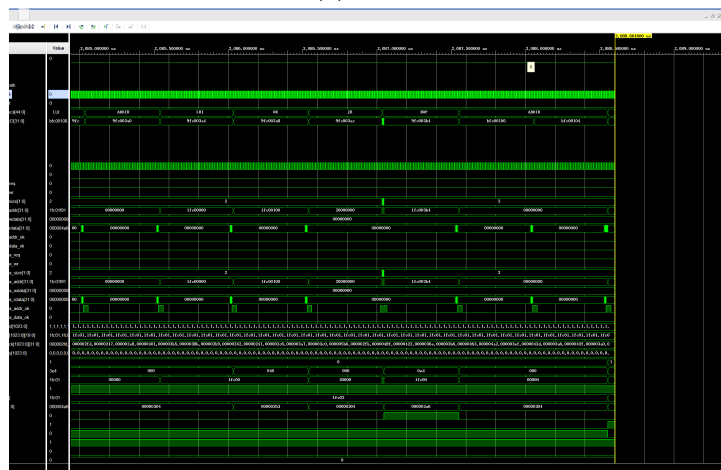
**问题描述:**refile 的内部信号未初始化 0, 加之 controller 中的控制信号在某些情况规定为 x, 虽然逻辑上在某些情况这些控制信号是 0 是 1 都是可以的, 但是会影响到其他运算, 导致不定态在运算中不断传递。

**解决方案:**将 controller 中的控制信号为 x 改成 0。

## 4 实验结果及分析

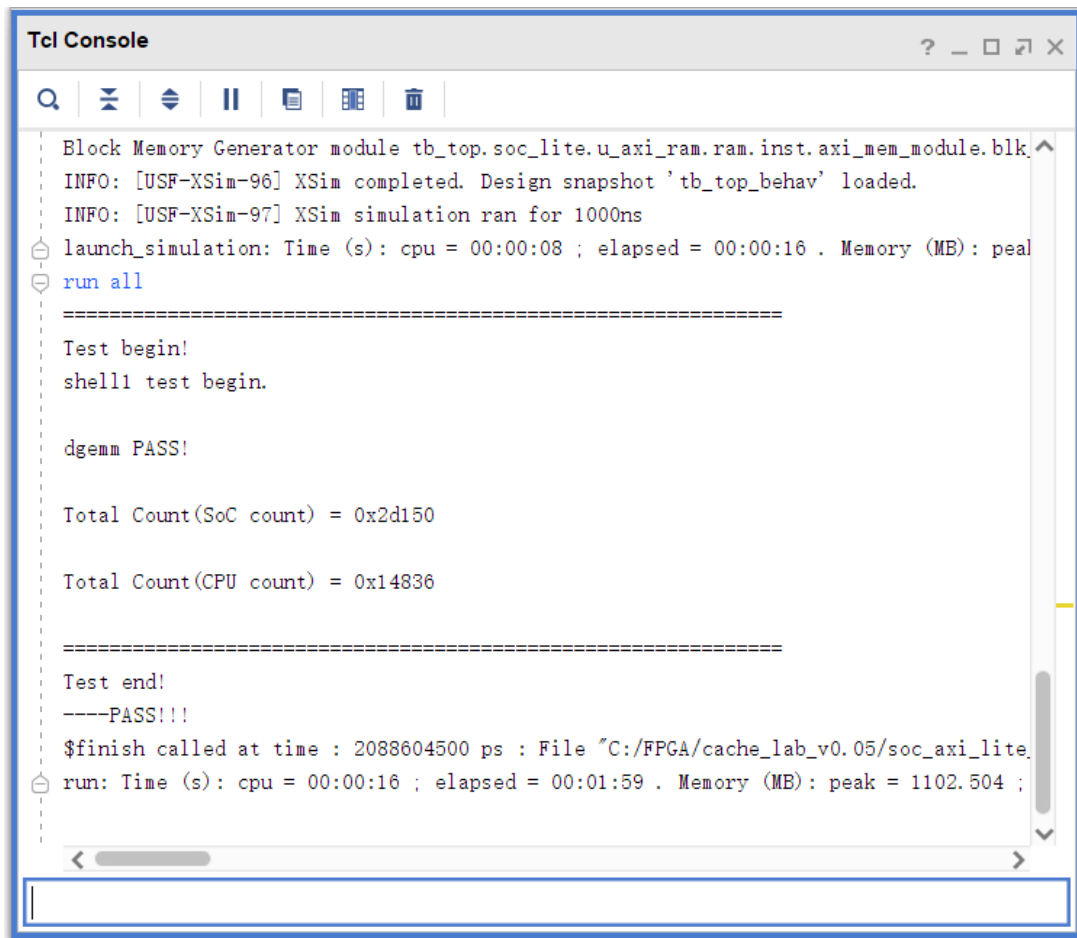


(a) 写直达



(b) 写回-写分配

图 3: 写直达和写回-写分配的仿真对比



The screenshot shows a 'Tcl Console' window with a toolbar at the top containing icons for search, zoom, execution, and other functions. The main text area displays the following output:

```
Block Memory Generator module tb_top.soc_lite.u_axi_ram.ram.inst.axi_mem_module.blk
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_top_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:08 ; elapsed = 00:00:16 . Memory (MB): peak
run all
=====
Test begin!
shell1 test begin.

dgemm PASS!

Total Count(SoC count) = 0x2d150

Total Count(CPU count) = 0x14836

=====
Test end!
---PASS!!!
$finish called at time : 2088604500 ps : File "C:/FPGA/cache_lab_v0.05/soc_axi_lite.
run: Time (s): cpu = 00:00:16 ; elapsed = 00:01:59 . Memory (MB): peak = 1102.504 ;
```

The window includes a horizontal scrollbar at the bottom of the text area and a vertical scrollbar on the right side.

图 4: 通过仿真测试

## A Datapath 代码

```
module d_cache (
    input wire clk, rst,
    //mips core
    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  ,

    //axi interface
    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_data_addr_ok  ,
    input      cache_data_data_ok  ,
);

//Cache 配置
parameter INDEX_WIDTH  = 10, OFFSET_WIDTH = 2;
localparam TAG_WIDTH    = 32 - INDEX_WIDTH - OFFSET_WIDTH;
localparam CACHE_DEPTH  = 1 << INDEX_WIDTH;

//Cache 存储单元
reg      cache_valid [CACHE_DEPTH - 1 : 0];
reg [TAG_WIDTH-1:0] cache_tag  [CACHE_DEPTH - 1 : 0];
reg [31:0] cache_block [CACHE_DEPTH - 1 : 0];
reg      cache_dirty [CACHE_DEPTH - 1 : 0];

//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 [TAG_WIDTH-1:0] c_tag;
wire [31:0] c_block;
```



```

wire c_dirty;

assign c_valid = cache_valid[index];
assign c_tag   = cache_tag   [index];
assign c_block = cache_block[index];
assign c_dirty = cache_dirty[index];

//判断是否命中
wire hit, miss;
assign hit = c_valid & (c_tag == tag);
assign miss = ~hit;

//读或写
wire read, write;
assign write = cpu_data_wr;
assign read = ~write;

//FSM
parameter IDLE = 2'b00, RM = 2'b01, WM = 2'b11;
reg [1:0] state;
always @(posedge clk) begin
    if(rst) begin
        state <= IDLE;
    end
    else begin
        case(state)
            IDLE: state <= cpu_data_req & hit ? IDLE :
                        cpu_data_req & miss & ~c_dirty ? RM :
                        cpu_data_req & miss & c_dirty ? WM :
                        IDLE; //Writeback和WriteAllocate策略下的状态机
            RM: state <= cache_data_data_ok ? IDLE : RM; //由于块只有1个字可以不用回填
            WM: state <= cache_data_data_ok ? RM : WM;

            default: state <= IDLE;
        endcase
    end
end

//读内存
wire read_req;
reg addr_rcv;
wire read_finish;

always @(posedge clk) begin
    addr_rcv <= rst ? 1'b0 :
        read_req & cache_data_addr_ok ? 1'b1 : //AXI接收地址成功
        read_finish ? 1'b0 : addr_rcv; //结束

```

```

end

assign read_req = (state == RM);
assign read_finish = read_req & cache_data_data_ok;

//写内存
wire write_req;
reg waddr_rcv;
wire write_finish;

always @(posedge clk) begin
    waddr_rcv <= rst ? 1'b0 :
        write_req & & cache_data_addr_ok ? 1'b1 :
        write_finish ? 1'b0 : waddr_rcv;
end

assign write_req = (state == WM);
assign write_finish = write_req & cache_data_data_ok;

//output to mips core
assign cpu_data_rdata = hit ? c_block : cache_data_rdata; //直至读命中把
    CacheLine中的数据发送
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) | (read_finish);

//output to axi interface
assign cache_data_req = (read_req & ~addr_rcv) | (write_req & ~waddr_rcv)
    ;
assign cache_data_wr = write_req; // 只有在写回状态才是写操作
assign cache_data_size = cpu_data_size;
assign cache_data_addr = write_req ? {c_tag, index, offset} :
    cpu_data_addr; //写回脏数据，读取新数据
assign cache_data_wdata = write_req ? c_block : cpu_data_wdata; // 写回时
    发送脏数据

//保存地址中的tag，index，防止地址发生改变
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

//写掩码生成
wire [31:0] write_cache_data;
wire [3:0] write_mask;

```

```

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);

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]}}}};

// 缺失flag, 1说明写缺失未处理, 缺失写回后进入RM状态进行写分配, 如何判断是
// 写缺失还是读缺失
reg write_miss_pending;
always @(posedge clk) begin
    if(rst) begin
        write_miss_pending <= 1'b0;
    end
    else if(cpu_data_req & write & miss) begin
        write_miss_pending <= 1'b1;
    end
    else if(read_finish & write_miss_pending) begin
        write_miss_pending <= 1'b0;
    end
end
// 写入Cache
integer t;
always @(posedge clk) begin
    if(rst) begin
        for(t=0; t<CACHE_DEPTH; t=t+1) begin
            cache_valid[t] <= 0;
            cache_dirty[t] <= 0;
        end
    end
    else begin
        if(read_finish) begin//写缺失或读缺失
            cache_valid[index_save] <= 1'b1;
            cache_tag[index_save] <= tag_save;
            cache_block[index_save] <= write_miss_pending ?
                write_cache_data : cache_data_rdata;
            cache_dirty[index_save] <= write_miss_pending ? 1'b1 : 1'b0;
        end
        else if(cpu_data_req & write & hit) begin
            // 写命中, 直接更新缓存并设置脏位
            cache_block[index] <= write_cache_data;
        end
    end
end

```

```
        cache_dirty[index] <= 1'b1;
    end
end
end
endmodule
```