《计算机组成与设计》实验报告

姓名：龙永奇
学院：计算机科学与技术学院
专业：计算机科学与技术
邮箱：3220105907@zju.edu.cn

报告日期：2024年6月14日

Lab6 < Cache >

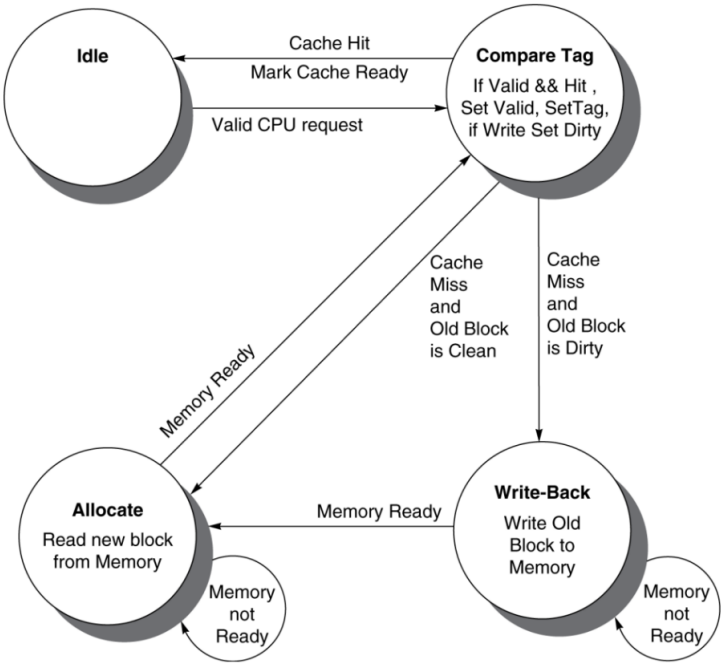
<3220105907> <龙永奇> 3220105907@zju.edu.cn

1 操作方法与实验步骤

1.1 缓存模块 Cache 设计

1.1.1 有限状态机

本实验所实现的 Cache 模块逻辑为 4 状态有限状态机：



- Idle 等待状态

  cache 在该状态下等待 CPU 发出有效的请求信号

  - 如果有有效的请求，进入 Compare Tag 状态，否则保持当前状态
  - 请求信号为 `MemRW`，0 表示无请求，1 表示读请求，2 表示写请求

- Compare Tag 标签比较状态

  根据 Index 找到 block，然后比较 Tag 找到具体数据，实际上就是检测是否命中

- 如果命中则进入 Idle 状态，否则进入 Allocate 状态

- Write-Back 写回状态

  如果未命中并且有脏块, 当前数据需要写回内存清空脏块

  - 如果 Memory 准备好则进入 Allocate 状态，否则保持当前状态

- Allocate 分配状态

  如果未命中并且没有脏块，就从内存中读数据

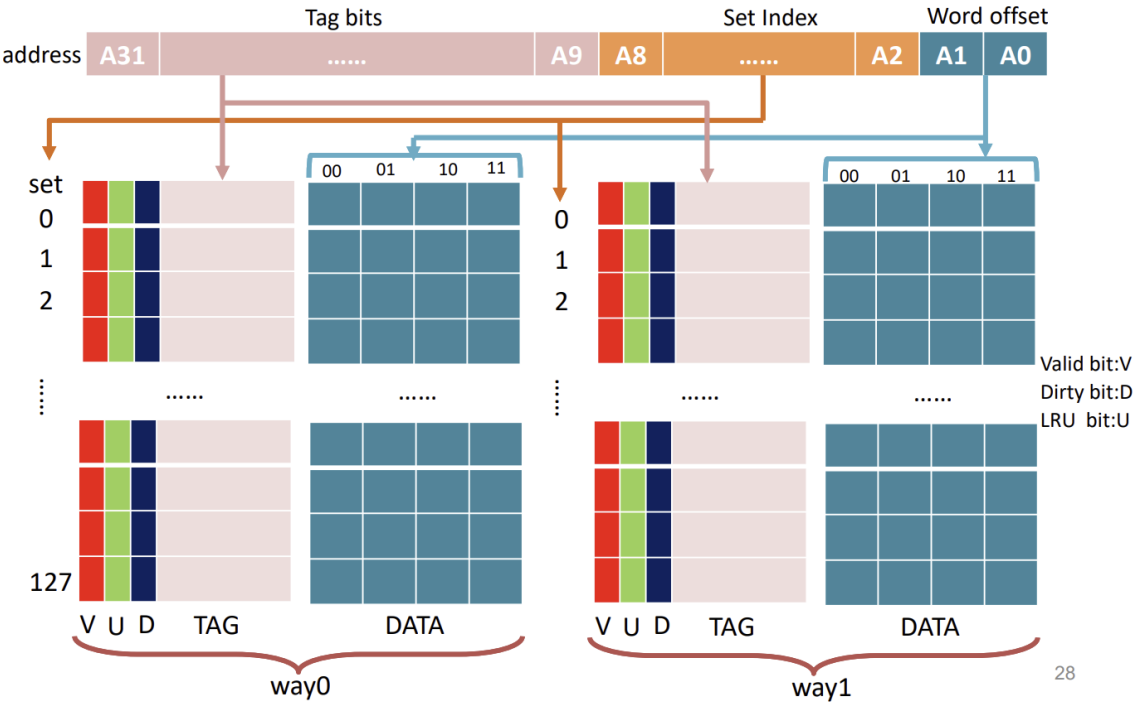
  - 如果 Memory 准备好则进入 Compare Tag 状态，否则保持当前状态

1.1.2 Cache 模块实现

- 本实验中的 Cache 模块类型为 2-way associative，替换策略为 LRU，写回策略为 Write back
- Cache 数据储存实现：

```
reg  [153:0] cache1 [127:0];
reg  [153:0] cache2 [127:0];
```

由于是两路组相联，因此需要两个 128 * 154 大小寄存器



  - [153:151]

    高 3 位分别表示 Vaild bit（该块是否有效）、Dirty bit（该块是否被修改） 以及 LRU 状态

  - [150:128]

    由于 CPU 生成的数据为 32 位，32 = 2 byte offset + 7 index + 23 tag，因此 tag 位需要 23 bits

  - [127:0]

    cache 中每个 data 数据大小为 1 word，共 4 个 data，因此一共 128 bits

  综上可知，cache 的整体大小为 2^12 = 4KB

整体实现代码如下：

```
module Cache(
    input               clk,
```

```
    input               rst,
    input [31:0]        addr_cpu_in,     // cpu给出从内存读取或写入数据的地址
    input [31:0]        write_data_cpu_in,// cpu给出需要写入内存的数据
    input [1:0]         MemRW_cpu_in,     // cpu发出请求，00无请求，01读请求，10写请求
    input [127:0]       mem_data,         // 从内存中读出的数据
    input               mem_ready,        // 内存是否准备好


    output reg [127:0]  mem_data_out,     // Dirty为1时，需要将cache中的数据写回内存
    output reg [31:0]   read_data,        // 读出cache中的数据
    output reg          MemRW_out         // 发生miss，从内存中读或者写
    );

    reg  [153:0] cache1 [127:0];          // set1
    reg  [153:0] cache2 [127:0];          // set2
    reg  [1:0]   state;
    wire [22:0]  tag = addr_cpu_in[31:9];    // 23 bits tag
    wire [6:0]   index = addr_cpu_in[8:2];   // 7 bits index
    wire [1:0]   boffset = addr_cpu_in[1:0]; // 2 bits byte offset
    localparam
        idle = 2'b00,
        cmptag = 2'b01,
        wback = 2'b10,
        alloc = 2'b11;

    always @(posedge clk or posedge rst) begin
        if(rst) begin
            state <= idle;
        end
        else begin
        case(state)
            idle: begin
                if(MemRW_cpu_in == 2'b00) begin
                    state <= idle;
                end
                else if(MemRW_cpu_in == 2'b01 || MemRW_cpu_in == 2'b10)begin
                    state <= cmptag;
                end
                MemRW_out <= 0;
            end
            cmptag: begin
                // hit
                // 第一路
                if(cache1[index][153] == 1'b1 && cache1[index][150:128] == tag)
begin
                    if(MemRW_cpu_in == 2'b01) begin        // 读
                        read_data <= cache1[index][(boffset * 32)+:32]; // 位选
择，如果写成 boffset*32+31:boffset*32 就会出现变量
                    end
                    else if(MemRW_cpu_in == 2'b10) begin   // 写
                        cache1[index][151] <= 1;
                        cache1[index][152] <= 1;
                        cache1[index][(boffset * 32) +:32] <= write_data_cpu_in;
                    end
                    state <= idle;
                end
```

```
                // 第二路
                else if(cache2[index][153] == 1'b1 && cache2[index][150:128] ==
tag) begin
                    if(MemRW_cpu_in == 2'b01) begin
                        read_data <= cache2[index][(boffset * 32) +:32];
                    end
                    else if(MemRW_cpu_in == 2'b10) begin
                        cache2[index][151] <= 1;
                        cache2[index][152] <= 1;
                        cache2[index][(boffset * 32) +:32] <= write_data_cpu_in;
                    end
                    state <= idle;
                end
                // miss
                else begin
                    // old block is dirty
                    if(cache1[index][152] || cache1[index][152]) begin
                        state <= wback;
                        MemRW_out <= 1;
                    end
                    // old block is clean
                    else begin
                        state <= alloc;
                        MemRW_out <= 0;
                    end
                end
            end
            wback: begin
                if(mem_ready) begin
                    // 第一个block
                    if(cache1[index][152]) begin
                        cache1[index][152] <= 0;
                        mem_data_out <= cache1[index][127:0];
                    end
                    // 第二个block
                    else begin
                        cache2[index][152] <= 0;
                        mem_data_out <= cache2[index][127:0];
                    end
                    state <= alloc;
                end
                else begin
                    state <= wback;
                end
            end
            alloc: begin
                if(mem_ready) begin
                    // 0 0, 0 1 的时候替换第一个block
                    if((!cache1[index][151] && !cache2[index][151]) |
(!cache1[index][151] && cache2[index][151])) begin
                        cache2[index][151] <= 0;
                        cache1[index][151] <= 1;  // U
                        cache1[index][152] <= 0;  // D
                        cache1[index][153] <= 1;  // V
                        cache1[index][127:0] <= mem_data;
                        cache1[index][150:128] <= tag;
```

```
                    end
                    // 1 0，1 1 的时候替换第二个block
                    else begin
                        cache1[index][151] <= 0;
                        cache2[index][151] <= 1;  // U
                        cache2[index][152] <= 0;  // D
                        cache2[index][153] <= 1;  // V
                        cache2[index][127:0] <= mem_data;
                        cache2[index][150:128] <= tag;
                    end
                    state <= cmptag;
                end
                else begin
                    state <= alloc;
                end
            end
        endcase
        end
    end
endmodule
```

1.2 仿真测试文件设计

```
`timescale 1ns / 1ps

module Cache_tb;
    reg             clk, rst;
    reg [31:0]      addr_cpu_in;
    reg [31:0]      write_data_cpu_in;
    reg [1:0]       MemRW_cpu_in;
    reg [127:0]     mem_data;
    reg             mem_ready;

    wire [127:0]    mem_data_out;
    wire [31:0]     read_data;
    wire            MemRW_out;

    Cache U1(
        .clk(clk),
        .rst(rst),
        .addr_cpu_in(addr_cpu_in),
        .write_data_cpu_in(write_data_cpu_in),
        .MemRW_cpu_in(MemRW_cpu_in),
        .mem_data(mem_data),
        .mem_ready(mem_ready),

        .mem_data_out(mem_data_out),
        .read_data(read_data),
        .MemRW_out(MemRW_out)
    );

    initial begin
```

```
        clk = 1;
        rst = 1;
        mem_ready = 1;
        MemRW_cpu_in = 0;
        #10;
        rst = 0;


        // read miss
        MemRW_cpu_in = 1;   // read
        addr_cpu_in = 32'h0001_0000; // 第0组，tag=1
        mem_data = 128'h1111_1111_3333_3333_5555_5555_7777_7777;
        #50;

        // read hit
        addr_cpu_in = 32'h0002_0000; // 第0组，tag=1
        #50;

        addr_cpu_in = 32'h0002_0001;
        mem_data = 128'h2222_2222_4444_4444_6666_6666_8888_8888;
        #50;
        addr_cpu_in = 32'h0002_0001;
        #50;

        // write hit
        MemRW_cpu_in = 2;   // write
        addr_cpu_in = 32'h0001_0001;
        write_data_cpu_in = 32'hABCD_ABCD;
        #50;
        addr_cpu_in = 32'h0002_0002;
        write_data_cpu_in = 32'hFEDC_FEDC;
        #50;

        // 检查刚才写进来的
        MemRW_cpu_in = 1;   // read
        addr_cpu_in = 32'h0001_0001;
        #50;
        addr_cpu_in = 32'h0002_0002;
        #50;

        // write miss
        MemRW_cpu_in = 2;   // write
        addr_cpu_in = 32'h0003_0000;
        write_data_cpu_in = 32'h1145_1419;
        #50;

        // 检查刚才写进来的
        MemRW_cpu_in = 1;   // read
        addr_cpu_in = 32'h0003_0000;
        #50;
        addr_cpu_in = 32'h0003_0001;
        #50;
    end
    always begin
        #5;
        clk = ~clk;
```

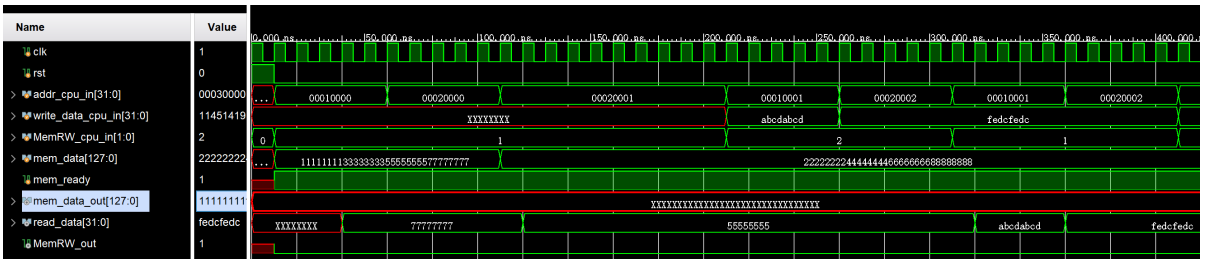
```
    end

endmodule
```

测试结果见"实验结果分析"部分

2 实验结果与分析

2.1 仿真结果分析



- read miss

  初始化后，将 MemRW 置 1，进行读操作，地址为 1000_0000

  此时 cache 中无数据，因此发生 read miss

- read hit 1

  将 mem_data 设为 128'h1111_1111_3333_3333_5555_5555_7777_7777

  此时内存中的数据会写入 cache，在下一个周期得到 7777_7777

- read hit 2

  设置地址为 h0002_0001，在下一个周期得到 5555_5555（后一个 byte）

- write hit

  将 MemRW 设为 2，进行写操作

  地址 0001_0001，写入数据 32'hABCD_ABCD

  地址 0002_0002，写入数据 32'hFEDC_FEDC

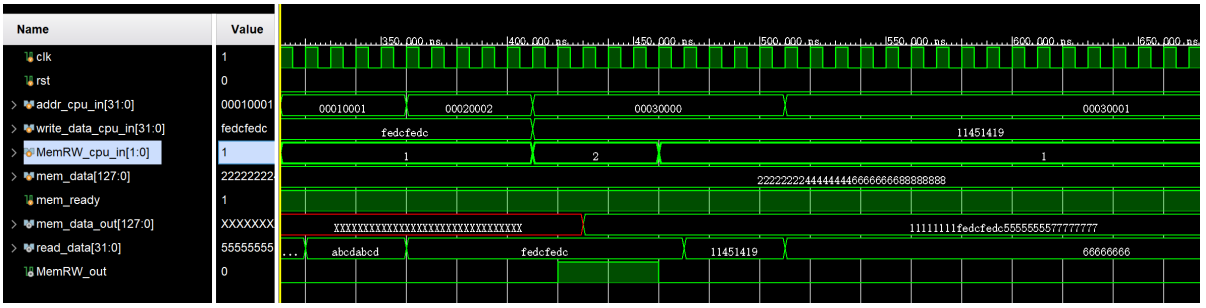
- 检查写入是否成功

  将 MemRW 设为 1，进行读操作

  地址为 0001_0001 时，读出数据 32'hABCD_ABCD

  地址为 0002_0002 时，读出数据 32'hFEDC_FEDC

  说明写入成功



- write miss

  将 MemRW 设为 2，进行写操作

地址 0003_0000，写入数据为 32'h1145_1419

此时 0 和 1 两个 block 已经写满数据，需要使用 LRU 进行替换

- 检查替换结果

  将 MemRW 设为 1，进行读操作

  地址为 0003_0000 时，读出数据 32'hFEDC_FEDC

  地址为 0003_0001 时，读出数据 32'h1145_1419

  最早被修改过的苏剧被替换，说明替换成功

3 讨论、心得

本次实验主要内容为 CPU 设计一个简单的两路组相联 cache，相比于之前的单周期流水线 CPU 设计，cache 实验简单了很多，也大体掌握了 cache 的结构以及实现方法，加深了理论课的印象

本次实验是本课程最后一次实验了，回顾本学期，在一共 6 次实验的过程中，由简入繁，由浅入深，深入学习掌握了单周期 CPU、五周期流水线 CPU 的原理及特性，课程与实验内容结合良好，在提高工程能力的同时，加深了对课程内容的理解，收获颇丰，对考试以及之后的计算机体系结构课程学习大有裨益，成就感满满！非常感谢两位助教在本学期实验过程中的悉心指导与解答，同时也感谢同学们的帮助！下学期再见！

4 思考题

1. **实验只设计实现数据缓存，若实现指令缓存，设计方法是否一样？指令缓存也会存在写回、写分派现象吗？指令缓存的内容如果需要修改，如何操作？**

   相同点：

   - 缓存结构: 数据缓存和指令缓存都采用类似的结构，如直接映射（Direct Mapped）、全相联（Fully Associative）、组相联（Set Associative）
   - 缓存大小和块大小: 两者都需要决定缓存的总大小和每个缓存块的大小。
   - 替换策略: 常见的替换策略，如LRU（Least Recently Used）、FIFO（First In First Out）、随机替换等
   - 命中和失效: 两者都需要处理缓存命中和失效的情况，从主存中加载数据或指令

   不同点：

   - 写策略: 数据缓存需要考虑写策略，如 Write-Back 和 Write-Through。但指令缓存不需要写策略，因为指令是从主存加载的，只读不写
   - 一致性问题: 数据缓存需要考虑数据一致性问题，如在多处理器系统中的缓存一致，指令缓存不涉及这些问题
   - 指令修改: 在大多数情况下，指令一旦加载到缓存中，就不会修改。但在一些特殊情况下（比如 python 的自修改代码），需要刷新缓存来确保正确性

   在自修改代码或动态生成代码时，需要处理指令缓存内容的更新，有以下几种方法：

   - 刷新缓存: 自修改代码在修改内存中的指令后，需要通知处理器刷新相关的指令缓存。这可以通过指令集中的特定指令实现
   - 同步缓存和主存: 自修改代码需要保证缓存和主存中的指令同步。这可以通过刷新指令缓存或整个缓存来实现
   - 缓存无效化: 在修改指令后，可以使特定缓存行无效，使处理器重新从主存中加载更新后的指令

2. **带缓存的流水线CPU如何实现，当发生缺失的情况 时CPU应该如何应对?**

   带缓存的流水线 CPU 实现

   - 指令缓存（I-Cache）: 在 IF 阶段，CPU 从指令缓存中读取指令。如果命中，则继续流水线。如果缺失，需要从主存中加载指令
   - 数据缓存（D-Cache）: 在 MEM 阶段，CPU 从数据缓存中读取或写入数据。如果命中，则继续流水线。如果缺失，则需要从主存中加载或写入数据

   当发生缓存缺失时:

   指令缓存缺失

   - 暂停流水线: 暂停流水线中的后续指令，避免数据依赖和冒险竞争
   - 加载指令: 从主存中加载缺失的指令块到指令缓存
   - 重新 IF : 缓存更新后，重新从指令缓存中取指，恢复流水线的执行

   数据缓存缺

   - 暂停流水线: 暂停当前指令以及所有后续指令，直到数据加载完成。
   - 加载数据: 从主存中加载缺失的数据块到数据缓存
   - 继续执行: 数据加载完成后，继续执行当前指令并恢复流水线

   缓存缺失的流水线处理

   - 非阻塞缓存（Non-blocking Cache）: 允许缓存缺失期间继续处理其他指令，减少停顿时间
   - 预取（Prefetching）: 提前将可能需要的数据和指令加载到缓存中，减少缺失概率
   - 多级缓存（Multi-level Cache）: 使用L1、L2、甚至L3缓存来分层次存储数据，进一步降低主存访问的延迟