浙江大学

本科实验报告

课程名称：	计算机组成与设计
姓　　名：	汪珉凯
学　　院：	计算机科学与技术学院
专　　业：	计算机
指导教师：	刘海风
报告日期：	2024 年 6　月 15　日

浙江大学实验报告

课程名称：计算机组成与设计 实验类型：综合

实验项目名称：Cache 的设计与实现

学生姓名：汪珉凯 学号：3220100975 同组学生姓名：赵桉、马扬松

实验地点：紫金港东四 509 室 实验日期：2024 年 6 月 15 日

一、操作方法与实验步骤

打开 vivado，新建工程 Lab6，创建代码 cache.v,具体代码如下：

```
`timescale 1ns / 1ps


module cache(
    input clk,
    input rst,
    input [127:0] from_mem_data,//data from memory
    input [31:0] from_cpu_data,//data from cpu
    input [31:0] addr,//address from memory
    input [1:0] request,//0:read, 1:write, 2\3:invalidate
    input ready_mem,//memory is ready
    output reg [127:0] to_mem_data,//data to memory
    output reg [31:0] to_cpu_data,//data to cpu
    output reg Mem_request,//memory request when miss,0 for read, 1 for write
    output reg finish //finish the request
    );
//FSM
//00:Idle ; 01:Compare Tag ; 10:Write Allocate ; 11:Write Back
reg [1:0] state;
//index bits = log2(128)=7
wire [6:0] index=addr[8:2];
//word offset bits = log2(4)=2
wire [1:0] word_offset=addr[1:0];
//tag bits =32(address bits)-2(word offset)-7(index bits)=23
wire [22:0] tag=addr[31:9];
//there are 128 cache regs and each has 2 ways
//each way has 154 bits( 3 bits for UDV ,23 bits for tag,128 bits for data)
reg [153:0] cache[127:0][1:0];
```

```
always @(posedge clk or posedge rst) begin
    if(rst) begin
        state<=2'b00;
    end
    else begin
        case(state)
            2'b00:begin//Idle
                finish<=0;//not finish
                Mem_request<=0;//no memory request
                if(request==2'b00 || request==2'b01) begin
                    state<=2'b01;//compare tag
                end
                else begin
                    state<=2'b00;//idle
                end
            end
            2'b01:begin//Compare Tag
                //hit:go to idle
                //first way
                if(cache[index][0][153] && cache[index][0][150:128]==tag) begin
                    if(request==2'b00) begin//read
                        to_cpu_data<=cache[index][0][(word_offset*32)+:32];//read data from cache
                        state<=2'b00;//idle
                        finish<=1;//finish
                    end
                    if(request==2'b01) begin//write
                        cache[index][0][(word_offset*32)+:32]<=from_cpu_data;//write data to cache
                        cache[index][0][152]<=1'b1;//set dirty bit
                        cache[index][0][151]<=1'b1;//set valid bit
                        state<=2'b00;//idle
                        finish<=1;//finish
                    end
                end
                //second way
                else if(cache[index][1][153] && cache[index][1][150:128]==tag) begin
                    if(request==2'b00) begin//read
                        to_cpu_data<=cache[index][1][(word_offset*32)+:32];//read data from cache
                        state<=2'b00;//idle
                        finish<=1;//finish
                    end
                    if(request==2'b01) begin//write
                        cache[index][1][(word_offset*32)+:32]<=from_cpu_data;//write data to cache
                        cache[index][1][152]<=1'b1;//set dirty bit
```

```
                    cache[index][1][151]<=1'b1;//set valid bit
                    state<=2'b00;//idle
                    finish<=1;//finish
                end
            end
            //miss
            //clean:write allocate
            else if(cache[index][0][152]==0 && cache[index][1][152]==0) begin
              state<=2'b10;//write allocate
              Mem_request<=0;//read
            end
            //dirty:write back
            else begin
              state<=2'b11;//write back
              Mem_request<=1;//write
            end
        end
```

```
        2'b10:begin//Write Allocate
            //if the memory isn't ready, wait
            if(!ready_mem) state<=2'b10;
            //if the memory is ready, allocate new cache line
            else begin
                //LRU strategy
                //cache[index][i][151]: 0=least used, 1=recently used
                if(cache[index][0][151]) begin
                    //cahche[index][0] is recently used,so replace cache[index][1]
                    cache[index][0][151]<=1'b0;//mark it as least used
                    cache[index][1][151]<=1'b1;//mark it as recently used
                    cache[index][1][153]<=1'b1;//valid
                    cache[index][1][152]<=1'b0;//clean
                    cache[index][1][150:128]<=tag;//update tag
                    cache[index][1][127:0]<=from_mem_data;//update data
                    state<=2'b01;//compare tag
                end
                else begin
                    //cahche[index][1] is recently used,so replace cache[index][0]
                    cache[index][1][151]<=1'b0;//mark it as least used
                    cache[index][0][151]<=1'b1;//mark it as recently used
                    cache[index][0][153]<=1'b1;//valid
                    cache[index][0][152]<=1'b0;//clean
                    cache[index][0][150:128]<=tag;//update tag
                    cache[index][0][127:0]<=from_mem_data;//update data
                    state<=2'b01;//compare tag
```

```
                end
            end
        end
        2'b11:begin//Write Back
            //if the memory isn't ready, wait
            if(!ready_mem) state<=2'b11;
            //if the memory is ready, write back
            else begin
                Mem_request<=1;//write
                state<=2'b10;//write allocate
                if(cache[index][0][152]) begin
                    to_mem_data<=cache[index][0][127:0];//write back data
                    cache[index][0][152]<=0;//reset dirty bit
                end
                else begin
                    to_mem_data<=cache[index][1][127:0];//write back data
                    cache[index][1][152]<=0;//reset dirty bit
                end
            end
        end
    endcase
end
end
endmodule
```

这个 Verilog 代码实现了一个简单的 2 路组相连（2-way set associative）缓存模块。缓存模块的主要功能是从内存中读取数据并将其缓存，以加速 CPU 的访问速度。它通过一个有限状态机（FSM）管理缓存操作，包括读、写、写回和分配等。

Cache 的主要结构

- 数据宽度：
  - 每个缓存行的大小为 128 位。
  - 从 CPU 来的数据为 32 位。
  - 地址宽度为 32 位。

- 缓存方式：
  - 2 路组相连，每组包含 2 个缓存块。
  - 每个缓存块包含 154 位数据（3 位 UDV，23 位标记，128 位数据）。

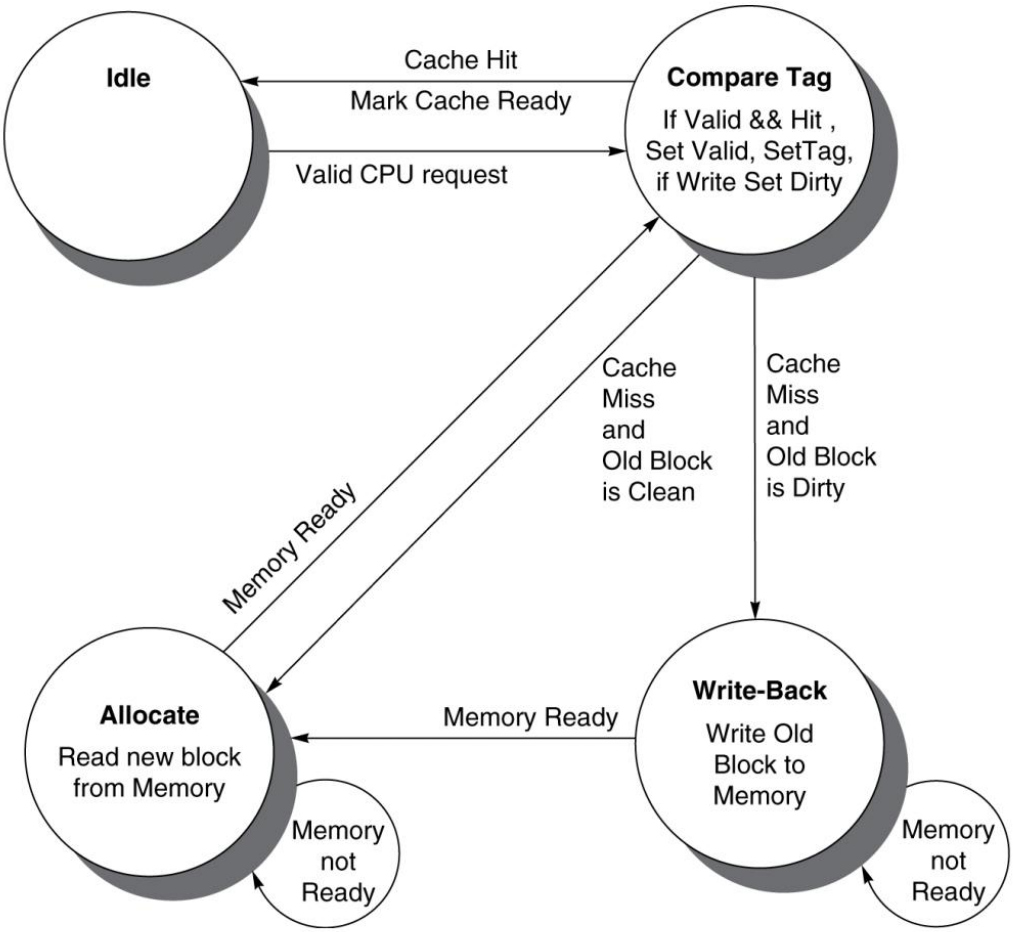
- 地址分割：
  - 7 位的索引位：用于选择缓存行。
  - 2 位的字偏移：用于选择缓存行中的字。
  - 23 位的标记位：用于匹配地址标记。

输入与输出

- 输入：
  - `clk`：时钟信号。
  - `rst`：复位信号。
  - `from_mem_data`：来自内存的数据（128 位）。
  - `from_cpu_data`：来自 CPU 的数据（32 位）。
  - `addr`：内存地址（32 位）。
  - `request`：请求类型（2 位，00：读，01：写，2\3：失效）。
  - `ready_mem`：内存是否准备好。

- 输出：
  - `to_mem_data`：写入内存的数据（128 位）。
  - `to_cpu_data`：返回给 CPU 的数据（32 位）。
  - `Mem_request`：内存请求（0：读，1：写）。
  - `finish`：请求完成标志。

有限状态机（FSM）



缓存模块的状态机有四个状态：空闲（Idle），比较标签（Compare Tag），写分配（Write Allocate）和写回（Write Back）。

状态 00: 空闲（Idle）

在空闲状态下，缓存模块等待请求。如果接收到读或写请求（`request` 为 00 或 01），则进入比较标签状态。如果请求为失效（2\3），则保持在空闲状态。

- 输入：`request`（请求类型）
- 输出：`finish`（完成标志）置 0，`Mem_request`（内存请求）置 0
- 下一状态：如果请求为读或写，则进入比较标签状态（01）；否则保持空闲状态。

状态 01: 比较标签（Compare Tag）

在比较标签状态下，缓存模块检查请求地址的标记是否与缓存中的标记匹配。如果命中（hit），则进行相应的读或写操作。如果未命中（miss），则根据缓存行的状态选择进入写分配或写回状态。

- 命中（Hit）：如果地址标记匹配且有效：
  - 读操作：从缓存中读取数据返回给 CPU。
  - 写操作：将数据写入缓存，并设置脏位（dirty bit）和有效位（valid bit）。
- 未命中（Miss）：如果两路缓存均未命中：
  - 干净（Clean）：进入写分配状态（10）。
  - 脏（Dirty）：进入写回状态（11）。

- 输入：`addr`（地址），`request`（请求类型）
- 输出：根据命中情况设置`to_cpu_data`（返回数据），更新缓存块数据和状态
- 下一状态：根据命中情况选择进入空闲、写分配或写回状态。

状态 10: 写分配（Write Allocate）

在写分配状态下，缓存模块等待内存数据准备好，然后根据最近最少使用（LRU）策略替换缓存行。

- 内存准备好：更新缓存行的数据和标记，设置最近使用位，并返回比较标签状态（01）。
- 内存未准备好：继续等待。

- 输入：`ready_mem`（内存准备好），`from_mem_data`（来自内存的数据）
- 输出：更新缓存行的数据和状态
- 下一状态：内存准备好则进入比较标签状态；否则保持在写分配状态。

状态 11: 写回（Write Back）

在写回状态下，缓存模块等待内存准备好，然后将脏的缓存行数据写回内存。

- 内存准备好：将脏数据写回内存，重置脏位，进入写分配状态（10）。
- 内存未准备好：继续等待。

- 输入：`ready_mem`（内存准备好）
- 输出：`to_mem_data`（写回内存的数据），重置缓存行的脏位
- 下一状态：内存准备好则进入写分配状态；否则保持在写回状态。

2.针对 cache.v 写一段仿真代码 cache_tb.v，具体如下所示：

```
`timescale 1ns / 1ps


module cache_tb;
    // Inputs
    reg clk;
    reg rst;
    reg [127:0] from_mem_data;
    reg [31:0] from_cpu_data;
    reg [31:0] addr;
    reg [1:0] request;
    reg ready_mem;
    // Outputs
    wire [127:0] to_mem_data;
    wire [31:0] to_cpu_data;
    wire Mem_request;
    wire finish;
    // Instantiate the Unit Under Test (UUT)
    cache uut (
        .clk(clk),
        .rst(rst),
        .from_mem_data(from_mem_data),
        .from_cpu_data(from_cpu_data),
        .addr(addr),
        .request(request),
        .ready_mem(ready_mem),
        .to_mem_data(to_mem_data),
        .to_cpu_data(to_cpu_data),
        .Mem_request(Mem_request),
        .finish(finish)
    );
    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end
```

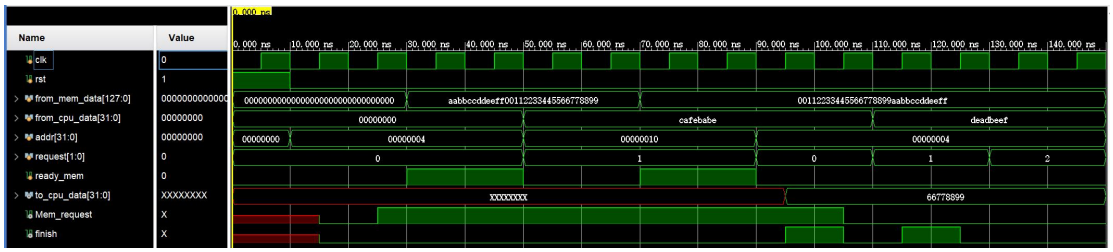
```
// Test sequence
initial begin
    // Initialize Inputs
    rst = 1;
    from_mem_data = 0;
    from_cpu_data = 0;
    addr = 0;
    request = 0;
    ready_mem = 0;
    // Apply reset
    #10;
    rst = 0;
    // Test case 1: Read miss -> Allocate
    addr = 32'h00000004;
    request = 2'b00; // Read request
    // Wait for a few clock cycles to observe the behavior
    #20;
    ready_mem = 1;
    from_mem_data = 128'hAABBCCDDEEFF00112233445566778899; // Simulated memory data
    // Wait for the cache to process the memory data
    #20;
    ready_mem = 0;
    // Test case 2: Write miss -> Write Allocate
    addr = 32'h00000010;
    request = 2'b01; // Write request
    from_cpu_data = 32'hCAFEBABE; // Data to write
    // Wait for a few clock cycles to observe the behavior
    #20;
    ready_mem = 1;
    from_mem_data = 128'h00112233445566778899AABBCCDDEEFF; // Simulated memory data for allocation
    // Wait for the cache to process the memory data
    #20;
    ready_mem = 0;
    // Test case 3: Read hit
    addr = 32'h00000004;
    request = 2'b00; // Read request
    // Wait for a few clock cycles to observe the behavior
    #20;
    // Test case 4: Write hit
    addr = 32'h00000004;
    request = 2'b01; // Write request
    from_cpu_data = 32'hDEADBEEF; // Data to write
    // Wait for a few clock cycles to observe the behavior
    #20;
```

```
        // Test case 5: Invalidate
        request = 2'b10; // Invalidate request
        // Wait for a few clock cycles to observe the behavior
        #20;
        // End of test
        $stop;
    end
endmodule
```

具体仿真结果见二。

二、实验结果与分析

仿真结果：



预期结果及波形分析：

测试用例 1：读操作缺失 -> 分配新缓存行

作用：发起一个地址为 0x00000004 的读请求。 预期结果:

addr = 0x00000004

request = 2'b00（读请求）

ready_mem 在 20 ns 后变为 1，表示内存数据准备好，提供模拟内存数据 0xAABBCCDDEEFF00112233445566778899。

等待 20 ns 后，ready_mem 变为 0。

结果：缓存模块检测缓存未命中，进入写分配状态，等待内存数据准备好后，将数据写入缓存，并更新标签和使用位，然后返回空闲状态。

测试用例 2：写操作缺失 -> 写分配

作用：发起一个地址为 0x00000010 的写请求，提供要写入的数据 0xCAFEBABE。 预期结果:

addr = 0x00000010

request = 2'b01（写请求）

from_cpu_data = 0xCAFEBABE

ready_mem 在 20 ns 后变为 1，提供模拟内存数据 0x00112233445566778899AABBCCDDEEFF。

等待 20 ns 后，ready_mem 变为 0。

结果：缓存模块检测缓存未命中，进入写分配状态，等待内存数据准备好后，将数据写入缓存，并更新标签和使用位，然后返回空闲状态。

测试用例 3：读操作命中
作用：发起一个地址为 0x00000004 的读请求。 预期结果:
addr = 0x00000004
request = 2'b00（读请求）
结果：缓存模块检测缓存命中，直接从缓存读取数据 0x66778899（模拟数据），并返回给 CPU，然后返回空闲状态。

测试用例 4：写操作命中
作用：发起一个地址为 0x00000004 的写请求，提供要写入的数据 0xDEADBEEF。 预期结果:
addr = 0x00000004
request = 2'b01（写请求）
from_cpu_data = 0xDEADBEEF
结果：缓存模块检测缓存命中，将数据 0xDEADBEEF 写入缓存，并设置缓存行的脏位和有效位，然后返回空闲状态。

测试用例 5：无效化操作
作用：发起一个无效化请求。 预期结果:
request = 2'b10（无效化请求）
结果：缓存模块将相应的缓存行标记为无效，然后返回空闲状态。

三、讨论、心得

本次实验总体较为简单，只需要以有限状态机的形式实现一个 cache 就可以。本次实验的难点在于，理解 cache 的工作原理，也是对我变相的一次复习。

思考题

1.指令缓存和数据缓存的方法类似，但是不会存在写回和写分派现象，因为指令通常来说是只读的。指令缓存中的内容一般不需要修改，但在一些特殊情况下可能需要更新。可以通过以下 2 种方式处理：
（1）无效化缓存：将相关缓存行标记为无效，下次访问时强制从内存重新加载。
（2）刷新缓存：强制将缓存内容更新为内存中的最新内容。

2.带 cache 的流水线 CPU，可以将以下两类缓存接入流水线中：
指令缓存（I-cache）：用于存储即将执行的指令，减少从内存中取指令的延迟。
数据缓存（D-cache）：用于存储即将访问的数据，减少从内存中取数据的延迟。
缓存缺失处理：
(1)指令缓存缺失（I-cache miss）：
1.1 暂停流水线：当取指阶段发生指令缓存缺失时，流水线会暂停，等待指令从内存加载到缓存。

1.2 加载新指令：从内存加载缺失的指令到指令缓存，然后继续执行。
1.3 指令预取：使用指令预取技术可以减少指令缓存缺失的概率。

(2)数据缓存缺失（D-cache miss）：
2.1 暂停流水线：当存储访问阶段发生数据缓存缺失时，流水线会暂停，等待数据从内存加载到缓存。
2.2 加载新数据：从内存加载缺失的数据到数据缓存，然后继续执行。
非阻塞缓存：使用非阻塞缓存（non-blocking cache）技术允许在等待数据加载时继续执行其他指令，以减少停顿时间。