《计算机组成原理》实验报告

年级、专业、班级	2022 级信息安全 01 班,2022 级信息安全 02 班	**姓名**	杨小艺,姚凡
实验题目	实验五 cache 设计		
实验时间	2024 年 6 月 2 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: 谭玉娟

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 6 月 10 日

1 实验内容

cache 设计实验 实验要求实现以下功能：首先，阅读实验环境介绍，明白 Cache 模块所处的位置与作用。然后，阅读示例 Cache 实现过程，了解设计并实现 Cache 的主要步骤。其中类 sram 接口部分可以参考文档《A12_ 类 SRAM 接口说明》接着，阅读 Cache 性能提升章节写回 Cache 内容，并参照示例代码，实现写回 Cache。其中以下过程比较重要分析写回 Cache 的流程图分析写回 Cache 的状态机分析写回 Cache 输入输出波形图接着，阅读调试章节的内容，用自己实现的 Cache 模块，替换掉示例 Cache 模块，然后运行仿真程序，进行调试。(选做) 可以阅读 Cache 性能提升章节其余内容，来提升 Cache 性能。

2 实验设计

2.1 Cache

2.1.1 功能描述

MIPS core 模块内部包含一个 5 级流水的 CPU 核，实现了 MIPS I 的 57 条基本指令。MIPS core 通过两个类 sram 接口对外进行指令访问和数据访问。

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。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

如下表所示。

2.2 直接映射 cache 结构

原始的内存数据可以看成一个一维的字节数组，内存地址作为一个索引，可以通过索引访问到对应的内存字节。地址和数据是一一对应的。而直接映射的 Cache 可以看成一个二维数组，内存地址被拆成 Tag, index 和 offset 三部分。其中 index 和 offset 构成这个二维数组的索引，通过 index 访问一个 cache line，通过 offset 确定 cache line 中数据块中对应的字。而由于 Cache 的容量远小于内存的容量，因此无法做到一一对应。当地址的 index 和 offset 都相同时，就会将两个地址映射到 cache 中的同一个位置造成冲突。因此此时就需要 tag 来表示两个不同的地址。

直接映射 Cache 结构如图所示。

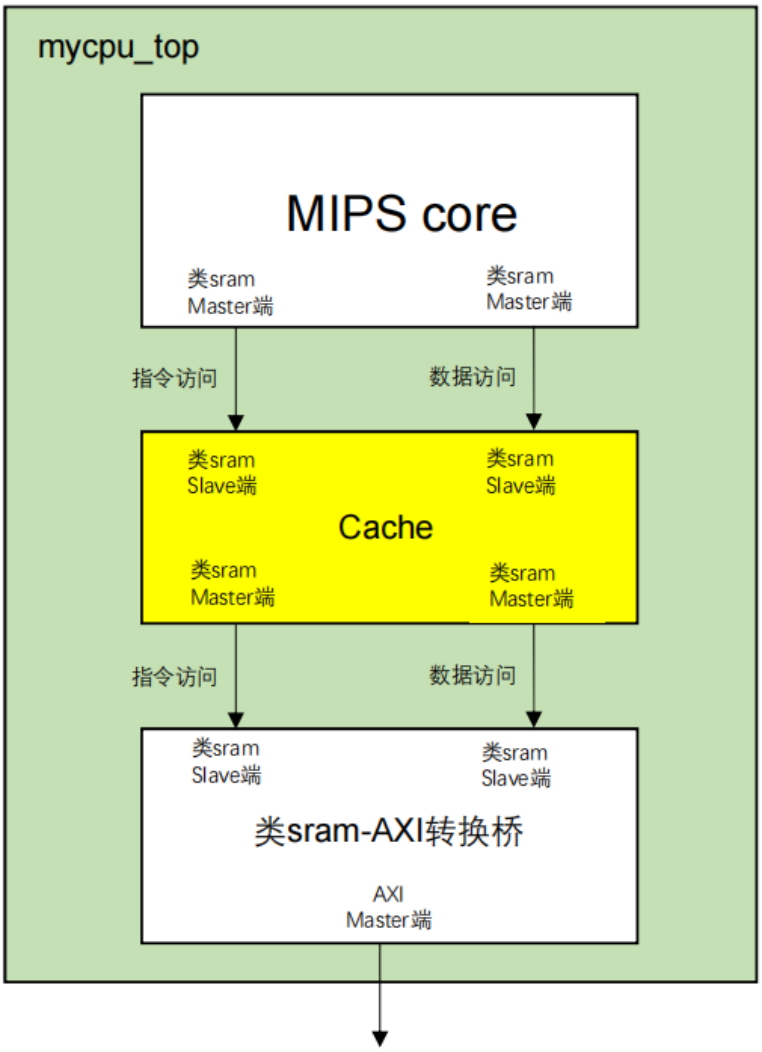
mycpu_top
MIPS core
类sram
Master端
类sram
Master端
指令访问
数据访问
类sram
Slave端
类sram
Slave端
Cache
类sram
Master端
类sram
Master端
指令访问
数据访问
类sram
Slave端
类sram
Slave端
类sram-AXI转换桥
AXI
Master端

图 1: cpu 顶层模块

表 1: 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 对数据的请求信号。当它为 1 时,表示 CPU 正在请求数据。
cpu_data_wr	input	1-bit	CPU 的写请求信号。当它为 1 时,表示 CPU 正在进行写操作。
cpu_data_size	input	2-bits	请求数据的大小。它的值可以是 00(字节)、01(半字)、10(字)或 11(双字)。
cpu_data_addr	input	32-bits	CPU 请求的数据地址。
cpu_data_wdata	input	32-bits	CPU 写入的数据。
cpu_data_rdata	output	32-bits	从 Cache 中读取的数据。
$cpu_data_addr_ok$	output	1-bit	数据地址确认信号。当它为 1 时,表示数据地址已经被 Cache 确认。
$cpu_data_data_ok$	output	1-bit	数据确认信号。当它为 1 时,表示数据已经被 Cache 确认。
$cache_data_req$	output	1-bit	Cache 对内存的请求信号。当它为 1 时,表示 Cache 正在请求内存数据。
$cache_data_wr$	output	1-bit	Cache 的写请求信号。当它为 1 时,表示 Cache 正在向内存写入数据。
$cache_data_size$	output	2-bits	Cache 请求的数据大小。
$cache_data_addr$	output	32-bits	Cache 请求的数据地址。
$cache_data_wdata$	output	32-bits	Cache 写入内存的数据。
$cache_data_rdata$	input	32-bits	从内存中读取的数据。
$cache_data_addr_ok$	input	1-bit	内存地址确认信号。当它为 1 时,表示内存地址已经被确认。
$cache_data_data_ok$	input	1-bit	内存数据确认信号。当它为 1 时,表示内存数据已经被确认。

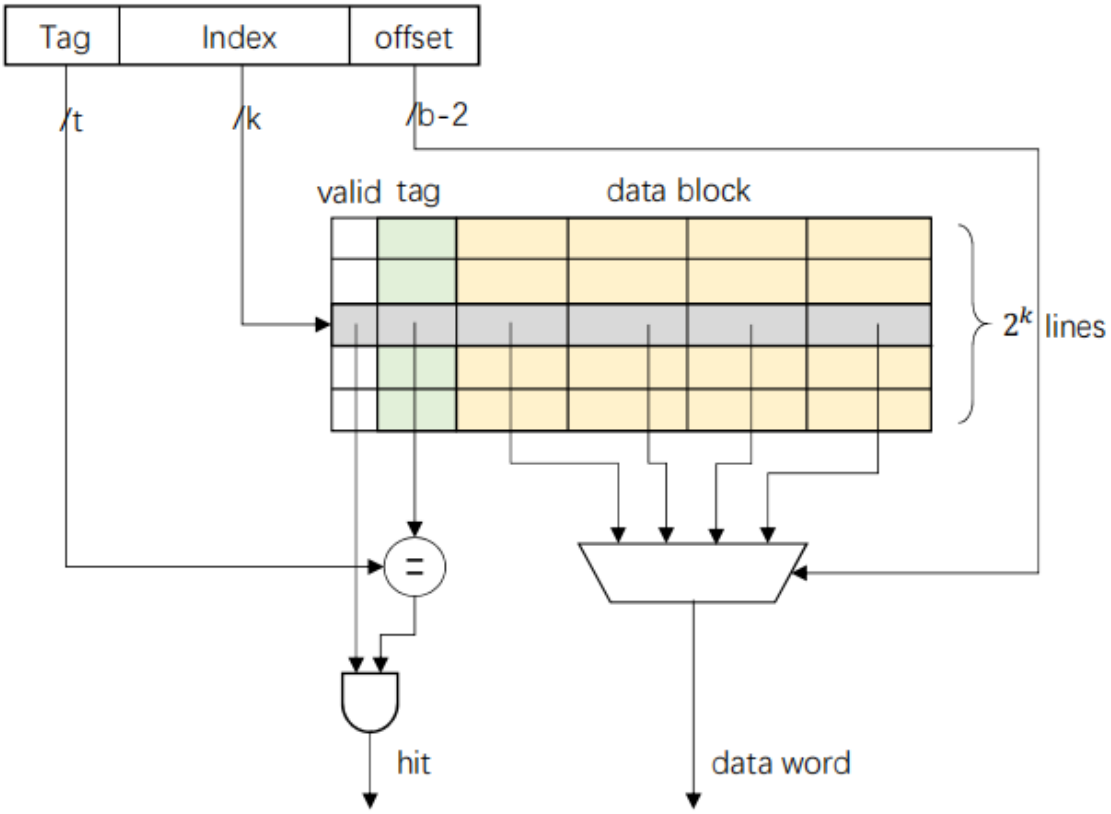

图 2: 直接映射 cache 结构

2.3 写直达——写不分配

2.3.1 策略

当 CPU 执行 store 类指令时，对 Cache 应该如何操作呢？当 Cache 命中时，最简单的想法是将数据同时写入到 Cache 和内存。这样保证了 Cache 和内存中的数据都是修改后最新的数据，保持了一致性。但也导致了大量的写内存操作，因此效率非常低。而另一种想法是只写入 Cache，同时标记该 Cache line 为已修改，等到万不得已的时候（该 cache line 被替换时）再写入内存。这可以大大减少写内存的次数。这两种策略分别为写直达 (Write Through) 和写回 (Write Back)。本示例出于简单考虑，使用写直达策略。上面讨论的是写命中的情况，当写缺失时。针对是否需要将数据写入 Cache，可以分为写分配 (Write Allocate) 和写不分配 (Non-Write Allocate)。写直达通常结合写不分配策略一起使用，即写缺失时，直接写入内存，而不写入 Cache。而写回通常结合写分配策略一起使用，即写缺失时，只写入 Cache，而不写入内存。

2.3.2 状态机设计

由于指令 Cache 不包含写的情况，我们以数据 Cache 为例进行分析。经过上面的分析，我们可以画出 Cache 读写处理的流程图，如下图所示。CPU 发来的访存请求可能是读 (load 类指令)，也可能是写 (store 类指令)。如果是读请求的话，判断是否命中。如果命中了的话，便立刻返回数

据。而如果缺失了，则需要从内存读取数据。再将数据写入 Cache，并返回数据给 CPU，这两个操作可以同时进行。如果是写请求的话，判断是否命中。如果缺失的话，将数据写入内存。如果命中了的话，则既需要将数据写入内存，也需要写入 Cache。因此，我们可以设计一个简单的状态机如图 5。图中定义了三种状态，IDLE 表示空闲状态也就是初始状态，RM 表示读取内存的状态，WM 代表写内存的状态。以读为例，当读命中时，因为可以马上返回数据，因此仍然保持 IDLE 状态。当读缺失时，则进入 RM 状态，直到读取内存数据完成时（data_ok），Cache 返回数据给 CPU，写入 Cache，然后返回到 IDLE 状态。

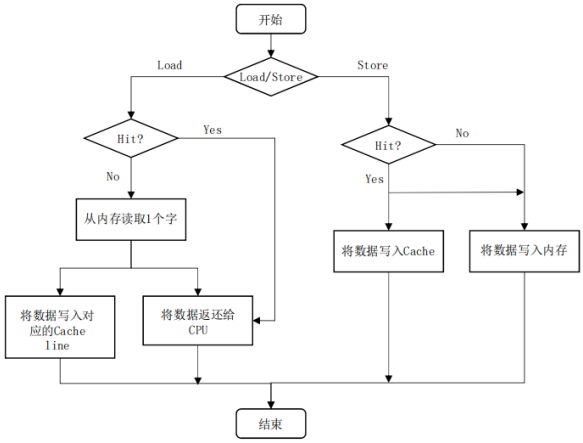

图 3: 示例 cache 流程图

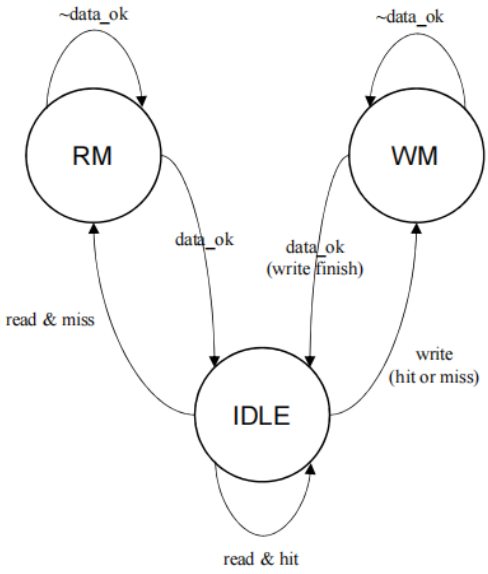

图 4: 示例数据状态机

2.4 写回——写分配

2.4.1 策略

写回策略是在写命中的情况下，并不直接将数据写入到下级存储器中，而是只将数据写入到索引到的 cache line 中，当且仅当一个脏的 cache line 要被替换的时候再将数据更新到内存中。写回策略一般是跟写分配策略配合的，写分配策略就是在写缺失的情况下，同时将数据写入到 cache 跟内存中。现在稍微总结一下采用写回–写分配策略、直接映射的 cache 的实现：

1 在读命中的情况下，CPU 直接读取对应的 cache line 的数据；

2 在读缺失的情况，如果索引到的 cache line 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 cache line 中；如果索引到的 cache line 是脏的，那么首先要发送写请求，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，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 cache line 中。

3 在写命中的情况下，如果索引到的 cache line 是干净的，那么直接将数据写入到对应的 cache line 中，并且将 dirty 位置为 1；如果索引到的 cache line 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 cache line 是干净的，那么将数据写入到 cache line 中，覆盖掉原来的数据。如果索引到的 cache line 是脏的，那么首先发送写请求，将脏的 cache line 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 cache line 中，并且将脏位标志位置为 1；

我们可能会多次将数据写入到某个相同的地址。针对这种情况，我们可以发现，写回策略仅需要在被替换出去的时候访问内存，而写通策略每次写操作都要访问内存。所以写回–写分配的策略有助于提升 cache 性能。

2.4.2 状态机设计

下图中定义了三种状态，IDLE 表示空闲状态也就是初始状态，RM 表示读取内存的状态，WM 代表写内存的状态。以读为例，当读命中时，因为可以马上返回数据，因此仍然保持 IDLE 状态。当读缺失且该 cache block 不是 dtrty 时，则进入 RM 状态，直到读取内存数据完成时（data_ok），Cache 返回数据给 CPU，写入 Cache，然后返回到 IDLE 状态；以写为例，当写命中时，直接写入 cache block，因此很快回到 IDLE 状态，当写缺失且 cache block 为 dirty 时，则进入 WM 状态，将当前 cache 里的数据写入内存，再从内存读取数据到该 cache block 上，然后返回到 IDLE 状态。

3 实验过程记录

在整个实验过程中完成了以下工作：计算机组成原理中，Cache 是一 种高速缓存存储器，旰加速 CPU 对于内存数据的访问。Cache 的设计实验我们完成了一下几个步骤:

1. 确定 Cache 的大小和映射方式: Cache 的大小和映射方式会影响 Cache 的性能和命中率。

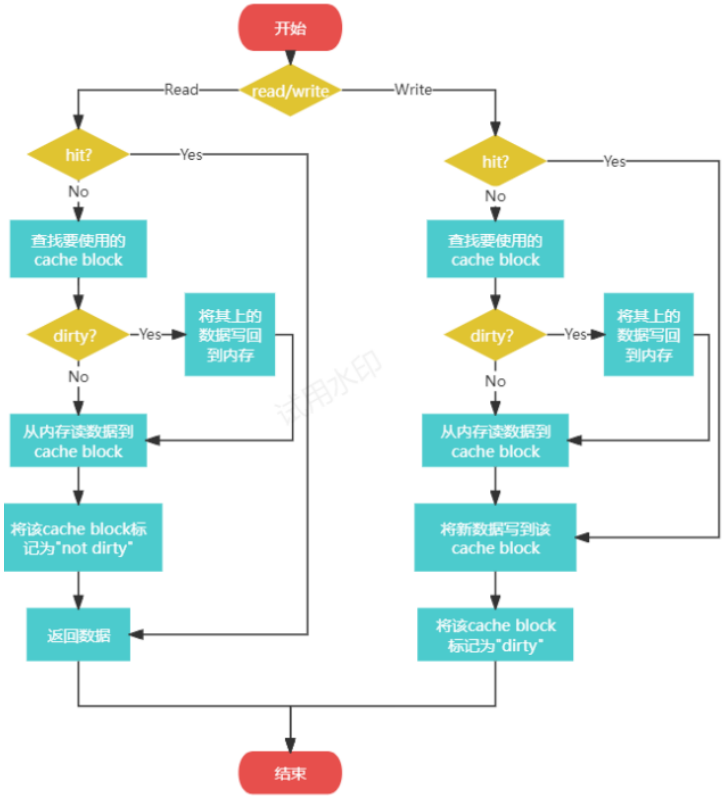

图 5: 写回流程图

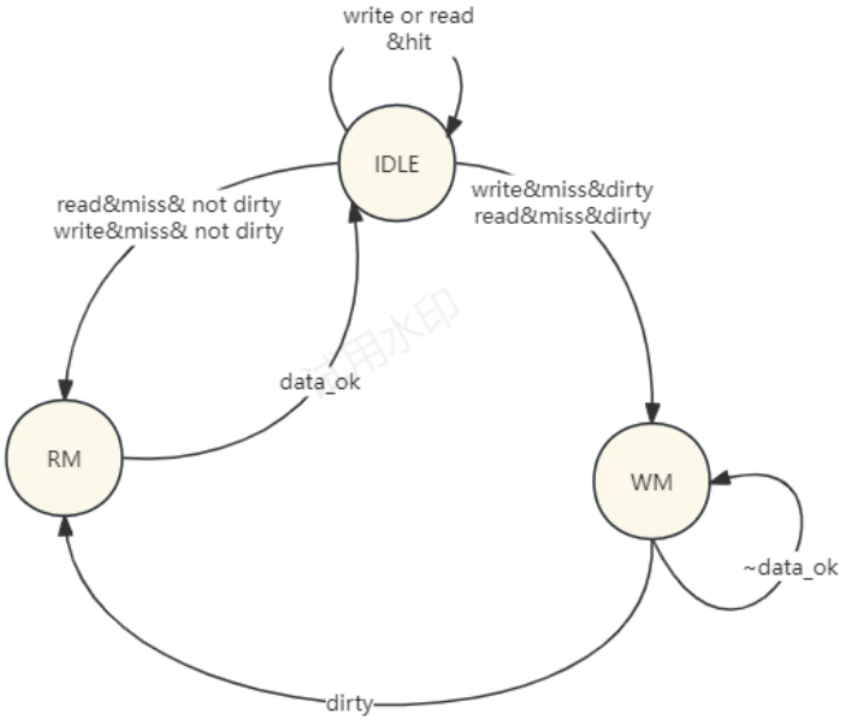

图 6: 写回状态机设计

通常可以选择直接映射、全相联映射或组相联映射等方式。

2. 编写 Cache 的读写模块: Cache 的读写模块用于实现 CPU 对 Cache 的访问。中读模块需要实现 Cache 的命中和替换算法, 写模块需要实现 Cache 的写回和写直通两种方式。

3. 编写 Cache 的控制模块: Cache 的控制模块用于实现 Cache 的状态转换和控制信号的生成。其中状态转换包括 Cache 的初始化、读写操作等,控制信号包括读写请求信号、命中信号等。

4. 进行仿真和测试: 在完成 Cache 的设计后,需要对 Cache 进行仿真和测试。使用 Verilog 语言进行仿真和测试,以验证 Cache 的正确性和性能。

5. 思考优化 Cache 的设计: 在完成初步设计后,可以通过修改 Cache 的映射方式、替换算法和控制模块等, 来优化 Cache 的性能和命中率。同时还可以对 Cache 的大小进行调整, 以达到最佳的性能和成本效益。

4 实验结果及分析

4.1 仿真结果

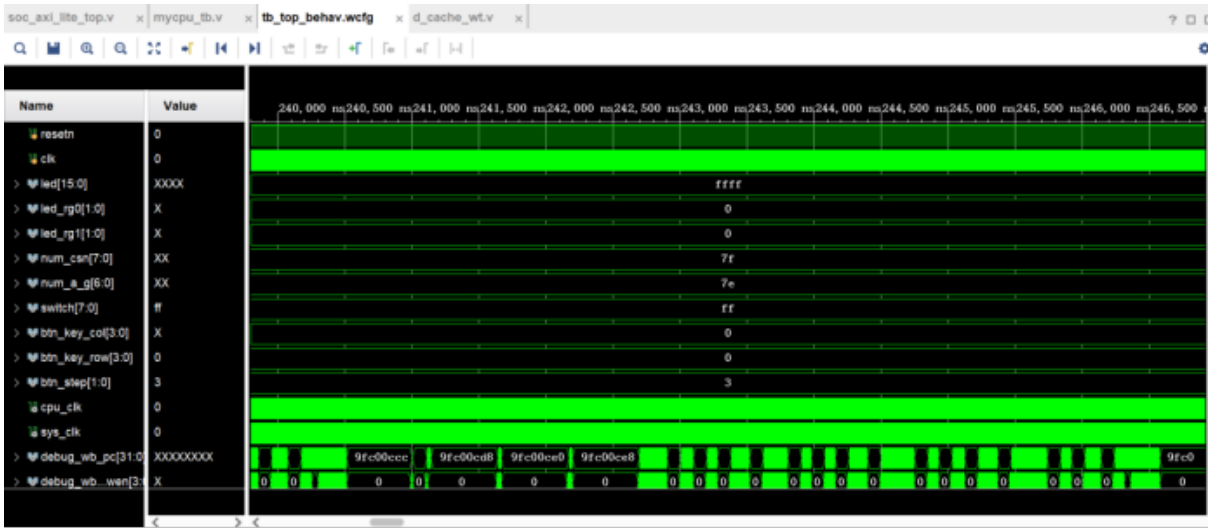

图 7

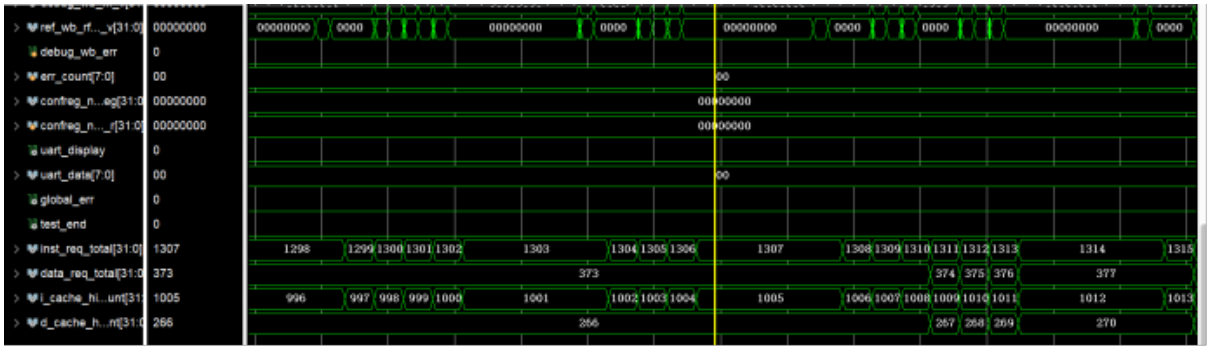

图 8

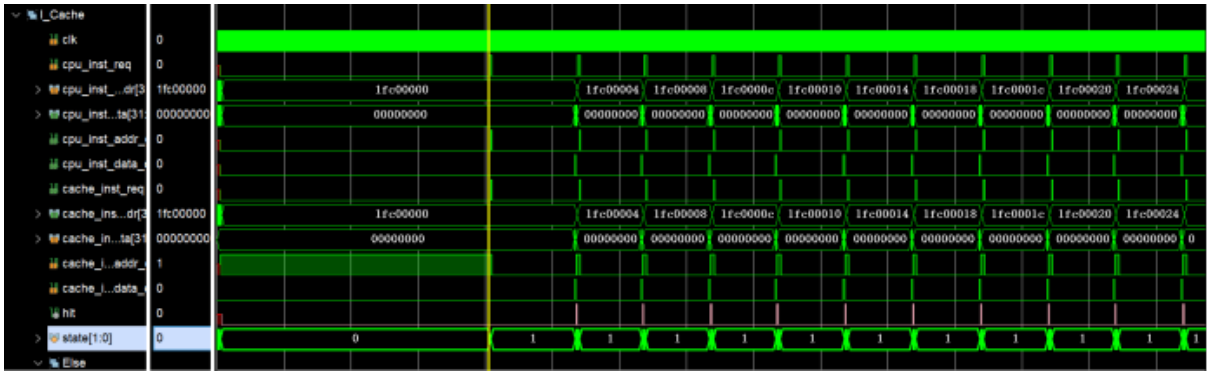

图 9

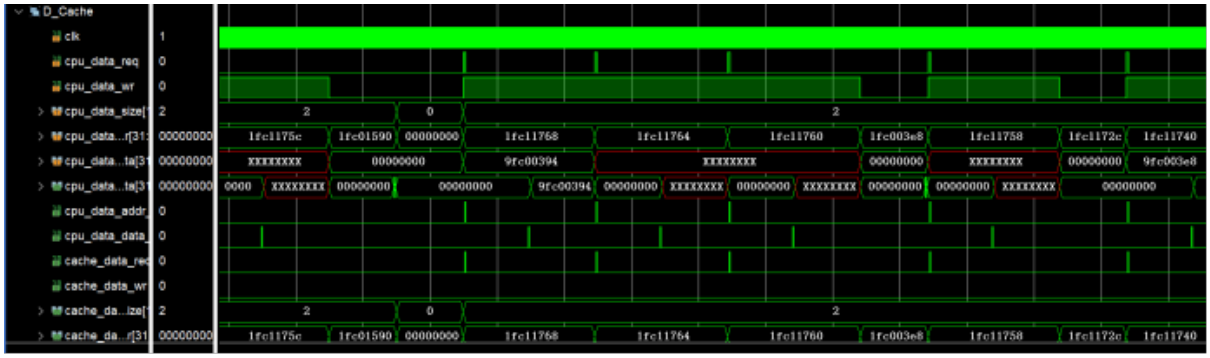

图 10

4.2 比较 write back 和 write through 时间上的区别

```
run all
shell1 test begin.

dgemm PASS!

Total Count(SoC count) = 0x2d2d6

Total Count(CPU count) = 0x14890

==============================================================
Test end!
----PASS!!!
$finish called at time : 2092454500 ps : File "C:/Users/17673/Desktop/cache_lab_after/soc_axi_lite_loongson/testbench/mycpu_tb.v" Line 267
run: Time (s): cpu = 00:00:23 ; elapsed = 00:05:24 . Memory (MB): peak = 1298.273 ; gain = 0.000
```

图 11: 控制台输出

A I_cache 代码

```
module i_cache (
    input  wire clk, rst,
    //mips core
    input           cpu_inst_req     ,
    input           cpu_inst_wr      ,
    input  [1 :0]   cpu_inst_size    ,
    input  [31:0]   cpu_inst_addr    ,
```

```
    input  [31:0] cpu_inst_wdata    ,
    output [31:0] cpu_inst_rdata    ,
    output        cpu_inst_addr_ok  ,
    output        cpu_inst_data_ok  ,

    //axi interface
    output         cache_inst_req     ,
    output         cache_inst_wr      ,
    output  [1 :0] cache_inst_size    ,
    output  [31:0] cache_inst_addr    ,
    output  [31:0] cache_inst_wdata   ,
    input   [31:0] cache_inst_rdata   ,
    input          cache_inst_addr_ok ,
    input          cache_inst_data_ok
);

    parameter  INDEX_WIDTH  = 10, 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 [TAG_WIDTH-1:0]  cache_tag   [CACHE_DEEPTH - 1 : 0];
    reg [31:0]           cache_block [CACHE_DEEPTH - 1 : 0];

    //
    wire [OFFSET_WIDTH-1:0] offset;
    wire [INDEX_WIDTH-1:0] index;
    wire [TAG_WIDTH-1:0] tag;

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

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

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

    //
    wire hit, miss;
    assign hit = c_valid & (c_tag == tag);
    assign miss = ~hit;

    //
```

```
parameter IDLE = 2'b00, RM = 2'b01;
reg [1:0] state;
always @(posedge clk) begin
    if(rst) begin
        state <= IDLE;
    end
    else begin
        case(state)
            IDLE:    state <= cpu_inst_req & miss ? RM : IDLE;
            RM:      state <= cache_inst_data_ok ? IDLE : RM;
        endcase
    end
end

//

wire read_req;
reg addr_rcv;
wire read_finish;
always @(posedge clk) begin
    addr_rcv <= rst ? 1'b0 :
                cache_inst_req & cache_inst_addr_ok ? 1'b1 :
                read_finish ? 1'b0 : addr_rcv;
end
assign read_req = state==RM;
assign read_finish = cache_inst_data_ok;

//output to mips core
assign cpu_inst_rdata   = hit ? c_block : cache_inst_rdata;
assign cpu_inst_addr_ok = cpu_inst_req & hit | cache_inst_req &
   cache_inst_addr_ok;
assign cpu_inst_data_ok = cpu_inst_req & hit | cache_inst_data_ok;

//output to axi interface
assign cache_inst_req   = read_req & ~addr_rcv;
assign cache_inst_wr    = cpu_inst_wr;
assign cache_inst_size  = cpu_inst_size;
assign cache_inst_addr  = cpu_inst_addr;
assign cache_inst_wdata = cpu_inst_wdata;


//
reg [TAG_WIDTH-1:0] tag_save;
reg [INDEX_WIDTH-1:0] index_save;
always @(posedge clk) begin
    tag_save   <= rst ? 0 :
                  cpu_inst_req ? tag : tag_save;
    index_save <= rst ? 0 :
                  cpu_inst_req ? index : index_save;
```

```
    end

    integer t;
    always @(posedge clk) begin
        if(rst) begin
            for(t=0; t<CACHE_DEEPTH; t=t+1) begin    //
                cache_valid[t] <= 0;
            end
        end
        else begin
            if(read_finish) begin //
                cache_valid[index_save] <= 1'b1;              //
                cache_tag  [index_save] <= tag_save;
                cache_block[index_save] <= cache_inst_rdata;
            end
        end
    end
endmodule
```

B d_cache 文件代码

```
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
    parameter  INDEX_WIDTH  = 10, 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 [TAG_WIDTH-1:0]   cache_tag   [CACHE_DEEPTH - 1 : 0];
reg [31:0]            cache_block [CACHE_DEEPTH - 1 : 0];

//
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];

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

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

//
wire hit, miss;
assign hit = c_valid & (c_tag == tag);
assign miss = ~hit;

//
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 & read & miss ? RM :
                             cpu_data_req & read & hit  ? IDLE :
                             cpu_data_req & write       ? WM : IDLE;
            RM:     state <= read & cache_data_data_ok ? IDLE : RM;
            WM:     state <= write & cache_data_data_ok ? IDLE : WM;
```

```
        endcase
    end
end


wire read_req;
reg addr_rcv;
wire read_finish;
always @(posedge clk) begin
    addr_rcv <= rst ? 1'b0 :
                read & cache_data_req & 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;


wire write_req;
reg waddr_rcv;
wire write_finish;
always @(posedge clk) begin
    waddr_rcv <= rst ? 1'b0 :
                 write & cache_data_req & cache_data_addr_ok ? 1'b1 :
                 write_finish ? 1'b0 : waddr_rcv;
end
assign write_req = state==WM;
assign write_finish = write & cache_data_data_ok;

//output to mips core
assign cpu_data_rdata   = hit ? c_block : cache_data_rdata;
assign cpu_data_addr_ok = read & cpu_data_req & hit | cache_data_req &
    cache_data_addr_ok;
assign cpu_data_data_ok = read & cpu_data_req & hit | cache_data_data_ok;

//output to axi interface
assign cache_data_req   = read_req & ~addr_rcv | write_req & ~waddr_rcv;
assign cache_data_wr    = cpu_data_wr;
assign cache_data_size  = cpu_data_size;
assign cache_data_addr  = cpu_data_addr;
assign cache_data_wdata = cpu_data_wdata;


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;


    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);

    //new_data = old_data & ~mask | write_data & mask
    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;
    always @(posedge clk) begin
        if(rst) begin
            for(t=0; t<CACHE_DEEPTH; t=t+1) begin
                cache_valid[t] <= 0;
            end
        end
        else begin
            if(read_finish) begin
                cache_valid[index_save] <= 1'b1;
                cache_tag  [index_save] <= tag_save;
                cache_block[index_save] <= cache_data_rdata;
            end
            else if(write & cpu_data_req & hit) begin
                cache_block[index] <= write_cache_data;
            end
        end
    end
endmodule
```