浙江大学

本科实验报告

课程名称：计算机组成

姓　　名：应周骏

学　　院：计算机科学与技术学院

系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：3200103894

指导教师：马德

2022 年 6 月 20 日

浙江大学实验报告

课程名称：计算机组成 实验类型：综合

实验项目名称：Cache 设计与实现

学生姓名：应周骏 专业：计算机科学与技术 学号：3200103894

同组学生姓名：无 指导老师：马德

实验地点：东 4-509 实验日期：2022 年 6 月 5 日

一、实验目的和要求

1.理解高速缓存的基本概念和作用；

2.掌握缓存的组织结构和映射方式、替换策略；

3.理解缓存的工作原理、命中率和一致性问题；

4.设计缓存的控制器模块和存储模块；

二、实验内容和原理

目标：

熟悉数据缓存的工作原理，了解存储单元的组织结构，掌握缓存控制器的设计方法，设计并测试两路组关联 Cache。

内容：

数据缓存模块的设计（two-way set associate cache）并进行数据缓存模块的仿真验证。

原理：

1. Cache 的基本概念

Cache 又叫高速缓冲存储器，位于 CPU 与内存之间，是一种特殊的存储器子系统。

目前比较常见的是两极 cache 结构，即 cache 系统由一级高速缓存 L1 cache 和二级高速缓存 L2 cache 组成，L1 cache 通常又分为数据 cache（D-Cache）和指

令 cache（I-Cache），它们分别用来存放数据和执行这些数据的指令。

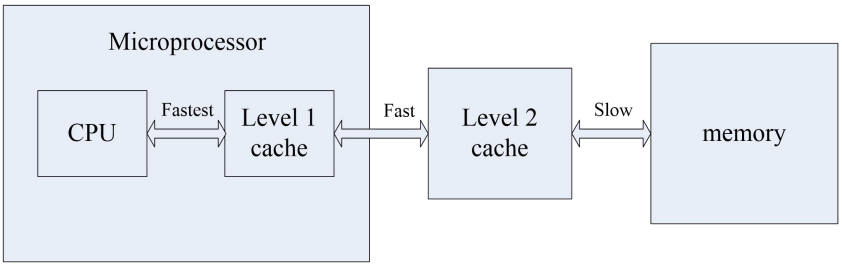

图 2.1 cache 示意图

2. Cache 的作用

Cache 的作用就是为了提高 CPU 对存储器的访问速度。

电脑的内存是以系统总线的时钟频率工作的，这个频率通常也就是 CPU 的外频。但是，CPU 的工作频率(主频)是外频与倍频因子的乘积。因此，内存的工作频率就远低于 CPU 的工作频率了。导致的直接结果是：CPU 在执行完一条指令后，常常需要“等待”一些时间才能再次访问内存，极大降了 CPU 工作效率。

3. Cache 的工作原理

CPU 运行程序是一条指令一条指令地执行的，而且指令地址往往是连续的，即 CPU 在访问内存时，在较短的一段时间内往往集中于某个局部，这时候可能会碰到一些需要反复调用的子程序。系统在工作时，可把这些活跃的子程序存入比主存快得多的 cache 中。

CPU 在访问内存时，首先判断所要访问的内容是否在 cache 中，如果在，则称为命中（hit），此时 CPU 直接从 cache 中调用该内容；否则称为未命中（miss），CPU 会通过 cache 对主存中的相应内容进行操作。

4. Cache 设计实现

• Cache 控制采用 FSM 实现。

• 实现标签比较，检测读写命中与否。

• 实现 Cache 的数据更新，当 miss 后采用 LRU 完成数据替换。

• 实现 Mem 的数据更新，当 write miss 后采用 Write Back，被改写的数据块在被替换出 cache 时写回到 Mem。同时实现 Write Allocate。

三、实验过程和数据记录

1. 工程文件建立

新建工程文件，命名为“OxExp06_cache”。

2. Cache 控制部分 FSM 设计

新建 Verilog 文件“cache.v”，输入以下代码。

```
module cache(
    input wire clk, // clock
    input wire rst, // reset
    input wire [31:0] data_cpu_write, // data write in
    input wire [31:0] data_mem_read, // data read in
    input wire [31:0] addr_cpu, // cpu addr
    input wire wr_cpu, // cpu write enable
    input wire rd_cpu, // cpu read enable
    input wire ready_mem, // memory ready
    output reg wr_mem, // memory write enable
    output reg rd_mem, // memory read enable
    output reg [31:0] data_mem_write, // data to mem
    output reg [31:0] data_cpu_read, // data to cpu
    output reg [31:0] addr_mem // memory addr
);
```

```
localparam   IDLE        = 3'd0,
             CompareTag  = 3'd1,
             Allocate    = 3'd2,
             WriteBack   = 3'd3,
             Update      = 3'd4,
             Read        = 3'd5;
reg [1:0]state, next_state;
wire [6:0]index;
reg en0, en1, ent0, ent1;
wire [25:0]rtag0,rtag1;
reg [25:0]wtag0,wtag1;
wire[22:0] tag;
reg[127:0]wdata;
reg[3:0] count;
wire[127:0] rdata0,rdata1;
wire hit,hit0,hit1,valid0,valid1,valid,LRU0,LRU1,dirty0,dirty1,dirty;
wire[1:0] wordsel;
wire[31:0] word0,word1;
reg[3:0]update;
reg[31:0] latch;
```

图 3.2.1 接口和临时变量

```
assign tag = addr_cpu[31:9];
assign valid0 = rtag0[25];
assign valid1 = rtag1[25];
assign valid = valid0 & valid1;
assign LRU0 = rtag0[24];
assign LRU1 = rtag1[24];
assign dirty0 = rtag0[23];
assign dirty1 = rtag1[23];
assign dirty = dirty0 | dirty1;
assign index = addr_cpu[8:2];
assign hit0 = valid0 & (tag == rtag0[22:0]);
assign hit1 = valid1 & (tag == rtag1[22:0]);
assign hit = hit0 | hit1;
assign wordsel = addr_cpu[1:0];
assign word0 = (wordsel == 2'd0) ? rdata0[31:0] :((wordsel == 2'd1) ? rdata0[63:32] :((wordsel == 2'd2) ? rdata0[95:64] : rdata0[127:96]));
assign word1 = (wordsel == 2'd0) ? rdata1[31:0] :((wordsel == 2'd1) ? rdata1[63:32] :((wordsel == 2'd2) ? rdata1[95:64] : rdata1[127:96]));
```

图 3.2.2 寄存器赋值

```
always@(posedge clk or posedge rst)
if(rst)
state<=IDLE;
else
state<=next_state;
```

图 3.2.3 时序逻辑状态转换

```
always@(*)
case(state)
IDLE:begin
    en0 <=1'b0;
    en1 <=1'b0;
    ent0 <= 1'b0;
    ent1 <= 1'b0;
    rd_mem <= 1'd0;
    wr_mem <= 1'd0;
    update <= 'd0;
    if(!wr_cpu)
    data_mem_write <= 32'dx;
    if(!rd_cpu)
    data_cpu_read <= 32'dx;
    count <= 4'b0;
    if(rd_cpu | wr_cpu) begin
        next_state=CompareTag;
    end
    else
        next_state=IDLE;
end
```

图 3.2.4 初始状态 IDLE

```
CompareTag:begin
en0 <=1'b0;
en1 <=1'b0;
ent0 <= 1'b0;
ent1 <= 1'b0;
rd_mem <= 1'd0;
wr_mem <= 1'd0;
if(update == 3'b100) begin
next_state <= Allocate;
update <= 3'd0;
end
else if(hit) begin
    if(wr_cpu) begin
        next_state <= IDLE;
    ent0 <= 1'b1;
    ent1 <= 1'b1;
    if(hit0)
        begin
        en0 <= 1'd1;
        case(wordsel)
            2'd0: wdata <= {rdata0[127:32],data_cpu_write};
            2'd1: wdata <= {rdata0[127:64],data_cpu_write,rdata0[31:0]};
            2'd2: wdata <= {rdata0[127:96],data_cpu_write,rdata0[63:0]};
            2'd3: wdata <= {data_cpu_write,rdata0[95:0]};
        endcase

      else if(rd_cpu) begin
          next_state <= IDLE;
          ent0 <= 1'b1;
          ent1 <= 1'b1;
          if(hit0) begin
              data_cpu_read <= word0;
              if(LRU0) //LRU0 -> 0
                  wtag0 <= {rtag0[25],1'd0,rtag0[23:0]};
              else
                  wtag0 <= {rtag0[25],1'd0,rtag0[23:0]};
              if(LRU1) //LRU1 = 1
                  wtag1 <= {rtag1[25],1'd1,rtag1[23:0]};
              else
                  wtag1 <= {rtag1[25],1'd1,rtag1[23:0]};
          end
          else if(hit1) begin
              data_cpu_read <= word1;
              //LRU0 0->1
              if(LRU0)
                  wtag0 <= {rtag0[25],1'd1,rtag0[23:0]};
              else
                  wtag0 <= {rtag0[25],1'd1,rtag0[23:0]};
              if(LRU1)
                  wtag1 <= {rtag1[25],1'd0,rtag1[23:0]};
              else
                  wtag1 <= {rtag1[25],1'd0,rtag1[23:0]};
          end
      end
  end
  else if(valid & dirty)begin
      next_state = WriteBack;
  end
  else
```

```
    if(LRU0) //LRU is 1
        wtag0 <= {rtag0[25],1'd0,1'd1,rtag0[22:0]};
    else
        wtag0 <= {rtag0[25],1'd0,1'd1,rtag0[22:0]};
    if(LRU1) //1
        wtag1 <= {rtag1[25],1'd1,rtag1[23:0]};
    else
        wtag1 <= {rtag1[25],1'd1,rtag1[23:0]};
    end
else
    begin
    en1 <= 1'd1;

    case(wordsel)
        2'd0: wdata <= {rdata1[127:32],data_cpu_write};
        2'd1: wdata <= {rdata1[127:64],data_cpu_write,rdata1[31:0]};
        2'd2: wdata <= {rdata1[127:96],data_cpu_write,rdata1[63:0]};
        2'd3: wdata <= {data_cpu_write,rdata1[95:0]};
    endcase

    if(LRU1)
        wtag1 <= {rtag1[25],1'd0,1'd1,rtag1[22:0]};
    else
        wtag1 <= {rtag1[25],1'd0,1'd1,rtag1[22:0]};
    if(LRU0)
        wtag0 <= {rtag0[25],1'd1,rtag0[23:0]};
    else
        wtag0 <= {rtag0[25],1'd1,rtag0[23:0]};
    end
```

```
        end
    end
    else if(valid & dirty)begin
        next_state = WriteBack;
    end
    else
    next_state=Allocate;
    end
```

图 3.2.5 CompareTag 状态

```
Allocate:begin
en0 <=1'b0;
en1 <=1'b0;
ent0 <= 1'b0;
ent1 <= 1'b0;
rd_mem <= 1'd1;
wr_mem <= 1'd0;
addr_mem = {addr_cpu[31:2],2'b0};
if(data_mem_read != latch) begin
wdata = {data_mem_read, wdata[127:32]};
update = update + 2'b01;
end
count = {1'd1,count[3:1]};
latch = data_mem_read;
if(LRU1) begin
    en1 = 1'b1;
    ent0 <= 1'b1;
    ent1 <= 1'b1;
    wtag1 = {3'b100,tag};
    wtag0 = {3'b010, rtag0[22:0]};
end
end
else begin
        en0 = 1'b1;
        ent0 <= 1'b1;
        ent1 <= 1'b1;
        wtag0 = {3'b100, tag};
        wtag1 = {3'b010, rtag1[22:0]};
    end
if(ready_mem & count == 4'b1111) begin
next_state=CompareTag;
end
else begin
if(!ready_mem)
count = 4'b0000;
next_state=Allocate;
end
end
```

图 3.2.6 Allocate 状态

```
WriteBack:begin
en0 <=1'b0;
en1 <=1'b0;
ent0 <= 1'b0;
ent1 <= 1'b0;
rd_mem <= 1'd0;
wr_mem <= 1'd1;
wdata <= 'h0;
addr_mem <= {addr_cpu[31:2], 2'd0};
if(LRU0 & dirty0) begin
    data_mem_write <= rdata0;
end
else if(LRU1 & dirty1)begin
    data_mem_write <= rdata1;
end
else begin
wr_mem <= 1'b0;
next_state <= Allocate;
end
if(ready_mem)
next_state = Allocate;
else
next_state = WriteBack;
end
default:next_state=IDLE;
endcase
// Instantiation Data RAM for Way 0
```

图 3.2.7 WriteBack 状态

```
Data_ram0 d0 (
    .clk(clk),
    .addr(index),
    .din(wdata),
    .en(en0),
    .dout(rdata0)
);
// Instantiation Data RAM for Way 1
Data_ram1 d1 (
    .clk(clk),
    .addr(index),
    .din(wdata),
    .en(en1),
    .dout(rdata1)
);
// Instantiation Tag RAM for Way 0
Tag_Ram0 t0 (
    .clk(clk),
    .addr(index),
    .din(wtag0),
    .en(ent0),
    .dout(rtag0)
);
// Instantiation Tag RAM for Way 1
Tag_Ram1 t1 (
    .clk(clk),
    .addr(index),
    .din(wtag1),
    .en(ent1),
    .dout(rtag1)
);
```

图 3.2.8 存储器接口

3. 存储器模块设计

新建 “Data_ram0.v” “Data_ram1.v” “Tag_ram0.v” “Tag_ram1.v” ，实现对应的存储模块。

```
module Data_ram0#(
    parameter Index_width = 7,      // Address Width Parameter, also used to calculate depth
    parameter Block_width = 128        // Data width parameter
)(
    input wire clk, // clock
    input wire en, // enable
    input wire [Index_width-1:0] addr, // address
    input wire [Block_width-1:0] din, // data write in
    output [Block_width-1:0] dout // data read out
    );

    localparam Num_of_sets  = 128;

    // Memory Array Decaration
    reg [Block_width-1:0] cache_tag [0:Num_of_sets-1];

    // Memory initialization
    initial
    begin
        $readmemh("dram0.mem", cache_tag);
    end

    // Internal Register to latch address for synchronous read operation
    reg [Index_width-1:0] rd_addr;

    // Internal Register to latch address for synchronous read operation
    reg [Index_width-1:0] rd_addr;

    // Sequential Block to write data in memory as well as latch read address
    // depending on 'we' write enable signal
    always@(posedge clk)
    begin
        if(en)
            cache_tag[addr] <= din;
        rd_addr <= addr;
    end

    // Data out during read operation with latched address to keep output stable
    assign dout = cache_tag[rd_addr];

endmodule
```

图 3.3.1 Data_ram 模块

```
module Tag_Ram0#(
    parameter Index_width = 7,      // Address Width Parameter, also used to calculate depth
    parameter Block_width = 26      // Data width parameter
)(
    input wire clk, // clock
    input wire en, // enable
    input wire [Index_width-1:0] addr, // address
    input wire [Block_width-1:0] din, // data write in
    output [Block_width-1:0] dout // data read out
    );

    localparam Num_of_sets  = 128;

    // Memory Array Decaration
    reg [Block_width-1:0] cache_tag [0:Num_of_sets-1];

    // Memory initialization
    initial
    begin
        $readmemh("tram0.mem", cache_tag);
    end

    // Internal Register to latch address for synchronous read operation
    reg [Index_width-1:0] rd_addr;

    // Internal Register to latch address for synchronous read operation
    reg [Index_width-1:0] rd_addr;

    // Sequential Block to write data in memory as well as latch read address
    // depending on 'we' write enable signal
    always@(posedge clk)
    begin
        if(en)
            cache_tag[addr] <= din;
        rd_addr <= addr;
    end

    // Data out during read operation with latched address to keep output stable
    assign dout = cache_tag[rd_addr];

endmodule
```

图 3.3.2 Tag_ram 模块

4. 仿真设计

依据给出的仿真代码，根据本实验状态机进行修改，代码如下。

```
wr_cpu = 0;
ready_mem = 1;
data_cpu_write = 0;
data_mem_read = 0;
repeat(4)
delay;

rst = 0;

delay;
delay;
//  Read from location
rd_cpu = 1'd1;
addr_cpu = 32'b0000_0000_0000_0000_0000_0010_0000_0111;
delay;
rd_cpu = 1'd1;
delay;
delay;
rd_cpu = 1'd0;
delay;

//  Write to same location
wr_cpu = 1'd1;
data_cpu_write = 32'hababababab;
addr_cpu = 32'b0000_0000_0000_0000_0000_0010_0000_0111;
delay;
delay;
wr_cpu = 1'd0;
delay;
delay;
// Read from same location to check updated data
```

```
// Read from same location to check updated data
rd_cpu = 1'd1;
addr_cpu = 32'b0000_0000_0000_0000_0000_0010_0000_0111;
delay;
rd_cpu = 1'd1;
delay;
rd_cpu = 1'd0;
delay;

//      Read Miss with dirty bit 0 policy check, reads data from Main Memory

delay;
rd_cpu = 1'd1;
addr_cpu = 32'b0000_0000_0000_0000_0000_0010_0000_1010;
delay;
rd_cpu = 1'd1;
@(posedge rd_mem);
ready_mem = 0;
repeat(4)
delay;

ready_mem = 1;

//delay;

data_mem_read = 32'h11111111;
```

图 3.4.1 cache 仿真（详见附件）

四、实验结果分析

1. cache 控制模块设计思路

本实验状态机依据实验 PPT 中给出的状态机设计。在 IDLE 状态时，对一些基本变量进行赋值。有请求时，进入 CompareTag 状态，此时判断是否命中，若命中，则直接读写数据，在写数据时设置 dirty 位，并更新 LRU 状态；若没有命中且当前块不是 dirty，则直接进入 Allocate 状态，若为 dirty，则进入 WriteBack 状态。在 Allocate 状态下，从 mem 中连续读入 4 个 block 值，采用比较读入值是否改变（类似上升沿触发）来判断是否读入，从而防止因为时序而导致反复读入。在 WriteBack 状态下，写回 cache 中当前 block 到 mem 中，并进入 Allocate 状态。具体细节依据下面的流程图（来自 wiki 百科）。

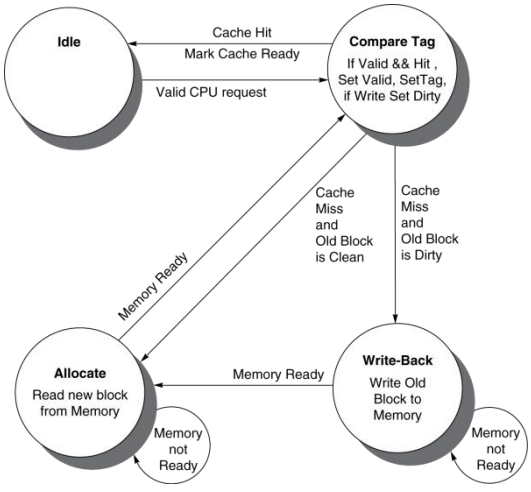

图 4.1.1 cache 控制状态机

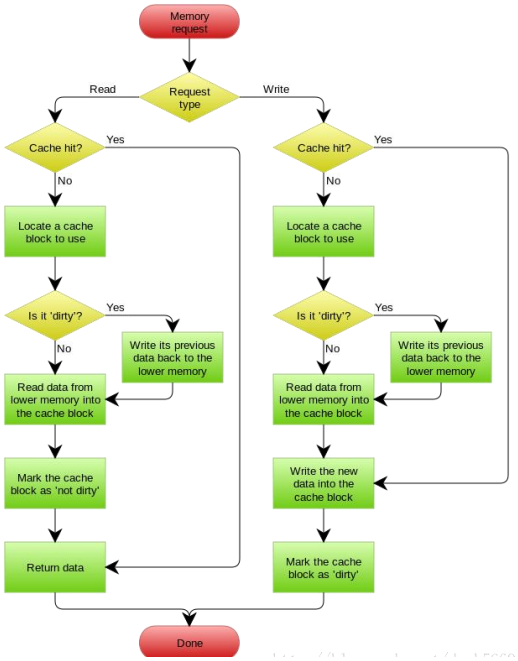

图 4.1.2 cache with WB&WA

2. 仿真结果分析

依据实验给出的仿真，测试了 read hit, read miss with dirty bit 0, write miss with dirty bit 0 三种情况，结果如图，符合实验预期。

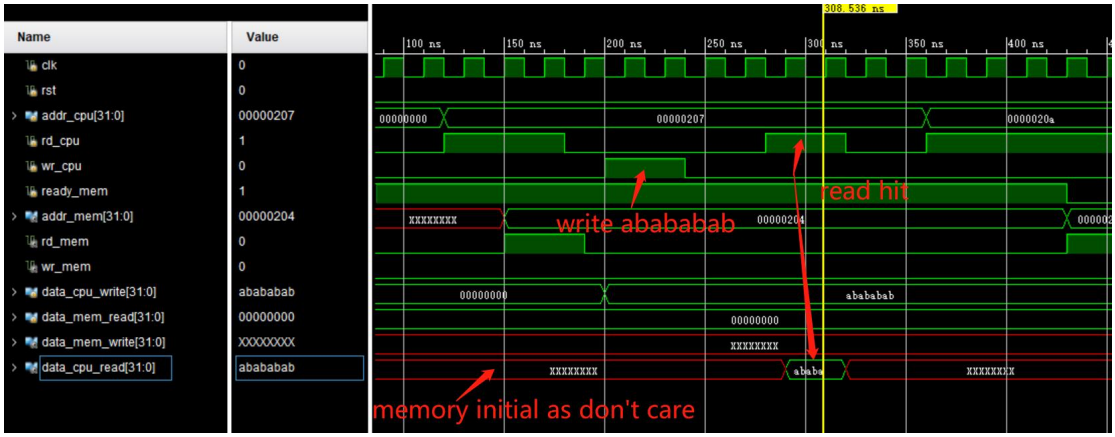

图 4.2.1 read hit

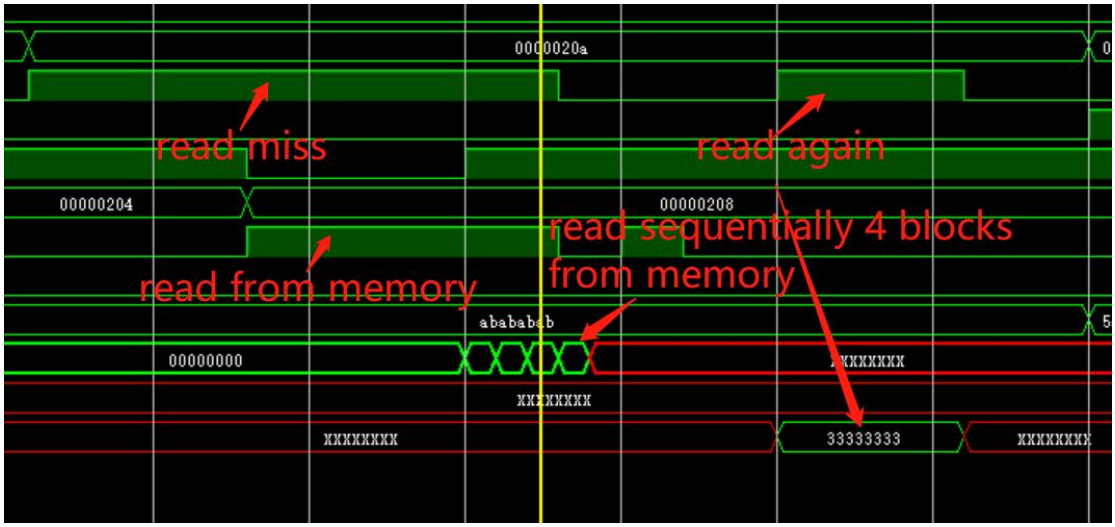

图 4.2.2 read miss with dirty bit 0

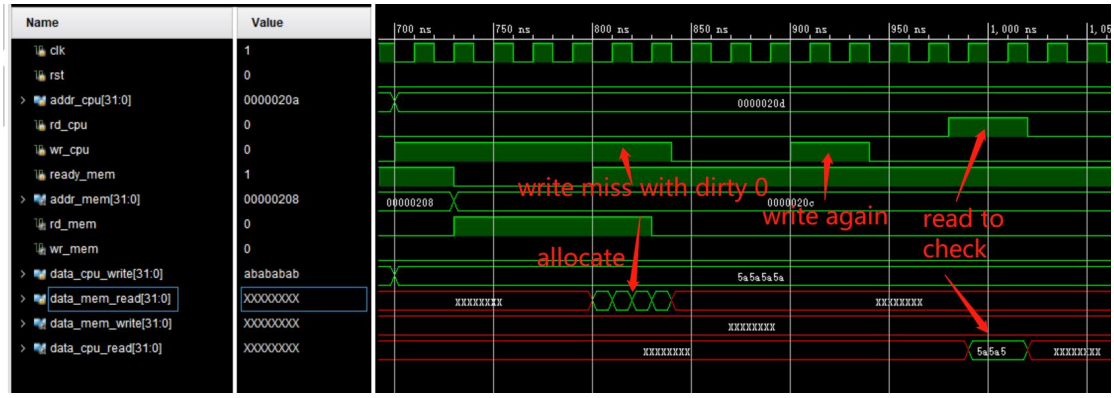

图 4.2.3 write miss with dirty bit 0

五、讨论与心得

1. 通过本次实验，我掌握了 Cache 的基本原理，基本了解了 Cache 存储的优势以及 LRU、Write Back、Write Allocate 等策略的具体实现，厘清了 cache 内部存储以及数据的组织（包括几位 tag/index、LRU 位、dirty 位、valid 位等等）。

2. 本次实验中，虽然给出了相应的教师版参考答案，但是其中的状态机与 PPT 中给出的状态机有较大区别，其中细分了很多二级的状态，因此我重新按照 PPT 给出状态机书写了 cache 控制器，基本能够实现功能。在过程中遇到了不少状态转移问题，例如涉及较多 assign 操作时，其顺序与我预想的不一致，导致了状态内部操作赋值存在一些问题；同时，对阻塞和非阻塞赋值，也花费了我不少时间，整体上比较艰难，不过通过这个实验，加深对 cache 的印象，在日后学习过程中也不会忘记。

3. 该实验进一步锻炼了我书写复杂状态机的能力，熟练度得到更多提升。