浙江大学

本科实验报告

课程名称：	计算机组成与设计
姓　　名：	张晋恺
学　　院：	竺可桢学院
系：	所在系
专　　业：	计算机科学与技术
学　　号：	3230102400
指导教师：	刘海风

2024 年 12 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 设计

学生姓名： 张晋恺 专业： 计算机科学与技术 学号： 3230102400

同组学生姓名： 指导老师： 刘海风

实验地点： 东 4-512 实验日期： 2024 年 12 月 16 日

一、操作方法与实验步骤

新建工程 cache，添加源代码如下

```
`timescale 1ns / 1ps

module cache(
    input clk,
    input rst,
    input [31:0] addr,          // 32 bit word address
    input [31:0] write_data,    // 1 word write data
    input [127:0] mem_data,     // 1 block data 4 word
    input [1:0] MemRW,          // 00: no operation, 01: read, 10: write
    input MIO_ready,            //
    output reg wb_op,       // when miss 0 read 1 write
    output reg hit,           // hit for cache
    output reg [127:0] mem_data_out, // dirty bit write back
    output reg [31:0] data
);


```

```
18  reg [153:0] cache [127:0][1:0];
19  // valid(1) dirty(1) lru(1) tag(23) data(128) = 153
20  // index 7 bit (128 line 2 way)  2 bit word offset
21
22  wire [1:0] offset = addr[1:0];
23  wire [6:0] index = addr[8:2];
24  wire [23:0] tag = addr[31:9];
25  reg [1:0] state;
26  integer i;
27
28  localparam  IDLE = 2'b00,
29              COMPARE_TAG = 2'b01,
30              ALLOCATE = 2'b10,
31              WRITE_BACK = 2'b11;
32
33  always @ (posedge clk or posedge rst) begin
34      if(rst) begin
35          for (i = 0; i < 128; i = i + 1) begin
36              cache[i][0] <= 154'h0;
37              cache[i][1] <= 154'h0;
38          end
39          state <= IDLE;
40          wb_op <= 0;
41          hit <= 0;
42          mem_data_out <= 128'h0;
43          data <= 32'h0;
44      end
45
46      else begin
47          case(state)
48          IDLE: begin
49              wb_op <= 0;
50              hit <= 0;
51              if (MemRW == 2'b01 || MemRW == 2'b10) begin
52                  state <= COMPARE_TAG;
53              end
54              else begin
55                  state <= IDLE;
56              end
57          end
58
59          COMPARE_TAG: begin
60              if(cache[index][0][153] == 1'b1 && cache[index][0][150:128]
                ↪  == tag) begin
61                  if(MemRW==2'b10) begin
```

```
                    cache[index][0][(offset*32)+:32] <= write_data;
                    cache[index][0][152] <= 1'b1;//dirty
                end
                else begin
                    data <= cache[index][0][(offset*32)+:32];
                end
                cache[index][0][151] <= 1'b1;//lru
                cache[index][1][151] <= 1'b0;//lru
                state <= IDLE;
                hit <= 1;
            end
            else if (cache[index][1][153] == 1'b1 &&
             ↪  cache[index][1][150:128] == tag) begin
                if(MemRW==2'b10) begin
                    cache[index][1][(offset*32)+:32] <= write_data;
                    cache[index][1][152] <= 1'b1;//dirty
                end
                else begin
                    data <= cache[index][1][(offset*32)+:32];
                end
                cache[index][1][151] <= 1'b1;//lru
                cache[index][0][151] <= 1'b0;//lru
                state <= IDLE;
                hit <= 1;
            end
            else begin
                if((cache[index][0][152] == 1'b1) ||
                 ↪  (cache[index][1][152] == 1'b1)) begin
                    state <= WRITE_BACK;
                end
                else begin
                    state <= ALLOCATE;
                    wb_op <= 0;
                end
                hit <= 0;
            end
        end

        ALLOCATE: begin
            if(MIO_ready == 1) begin
                if(cache[index][0][151] == 1'b1) begin
                    cache[index][0][151] <= 1'b0;

                    cache[index][1][151] <= 1'b1;
                    cache[index][1][152] <= 1'b0;
```

```
                    cache[index][1][153] <= 1'b1;
                    cache[index][1][150:128] <= tag;
                    cache[index][1][127:0] <= mem_data;
                end
                else begin
                    cache[index][1][151] <= 1'b0;

                    cache[index][0][151] <= 1'b1;
                    cache[index][0][152] <= 1'b0;
                    cache[index][0][153] <= 1'b1;

                    cache[index][0][150:128] <= tag;
                    cache[index][0][127:0] <= mem_data;
                end
                state <= COMPARE_TAG;
            end
            else begin
                state <= ALLOCATE;
            end
        end

        WRITE_BACK: begin
            if(MIO_ready == 1) begin
                if(cache[index][0][152] == 1'b1 && cache[index][1][151]
                ↪  == 1'b1 ) begin//第0块被替换且dirty
                    mem_data_out <= cache[index][0][127:0];
                    cache[index][0][152] <= 1'b0;
                    wb_op <= 1;
                end
                else if(cache[index][1][152] == 1'b1 &&
                ↪  cache[index][0][151] == 1'b1) begin//第1块被替换且dirty
                    mem_data_out <= cache[index][1][127:0];
                    cache[index][1][152] <= 1'b0;
                    wb_op <= 1;
                end
                else begin
                    mem_data_out <= 128'h0;
                    wb_op <= 0;
                end
                state <= ALLOCATE;
            end
            else begin
                state <= WRITE_BACK;
            end
        end
```

```
148
149         default: begin
150             state <= IDLE;
151         end
152
153         endcase
154     end
155 end
156
157 endmodule
```

代码解释

本次 cache 设计采用有限状态机的方式，其时序图如下

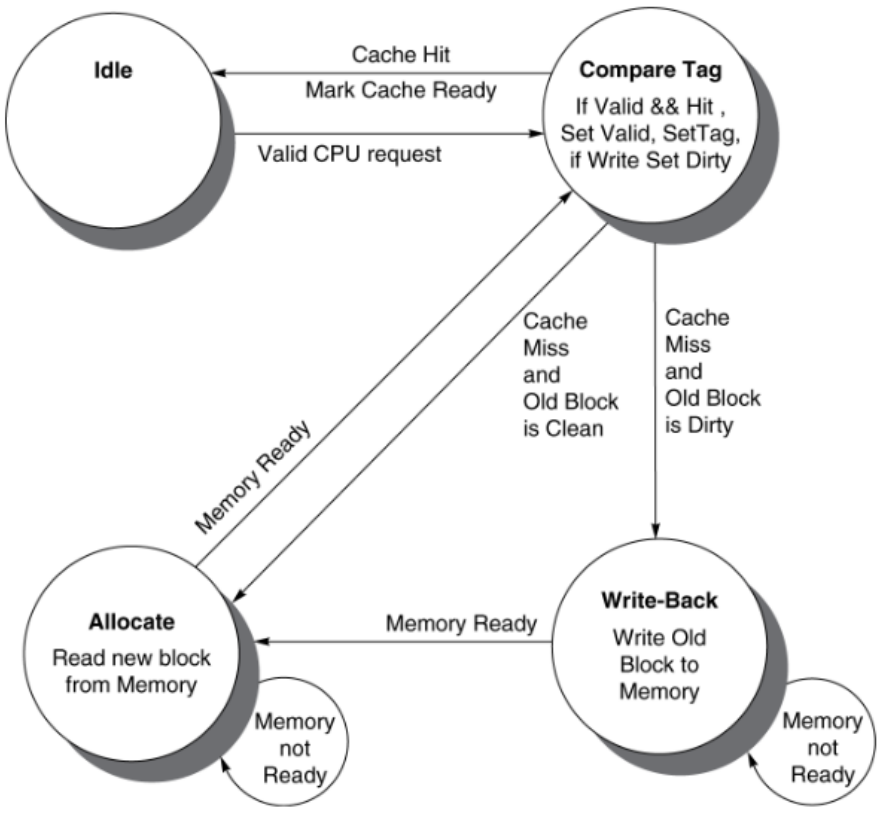

图 1: cache 时序图

首先，定义 cache 模块，本次实现的是2 路组相连，组数为 128，每一 block 为 4 个 word的 cache，该 cache 的接受输入的字地址为 32 位，其中前 23 位为 tag，中间

7 位为组 index，后 2 位为块内 offset。

因此该 cache 的每一行需要

- 1 位 valid 位，表示该行是否有效
- 1 位 dirty 位，表示该行是否被修改
- 1 位 lru 位，表示该行是否被使用
- 23 位 tag，表示该行的 tag
- 128 位 data，表示该行的数据

一共是 154 位，因此 cache 的每一行需要 154 位来存储，而 cache 有 128 组，一组 2 路，因此 cache 定义为 `reg [153:0] cache [127:0][1:0];`

然后四个状态之间的转换如下

IDLE

在 IDLE 状态，cache 等待 CPU 的请求，如果 CPU 的请求是读写请求，则 cache 进入 COMPARE_TAG 状态，否则 cache 继续在 IDLE 状态。

COMPARE_TAG

在 COMPARE_TAG 状态，cache 比较输入的地址找到对应的组，然后比较 tag 和 cache 中的 tag，如果匹配 (hit)，则根据 MemRW 的值决定是否将数据写入 cache，

- 如果 MemRW 为 10，则将数据写入 cache，并设置 dirty 位为 1，lru 位为 1，同组的另外一行设置 lru 位为 0；
- 如果 MemRW 为 01，则将数据从 cache 中读出，并设置 lru 位为 1，同组的另外一行设置 lru 位为 0；

如果未匹配 (miss)，则 cache 需要从内存中取出数据，然后写入 cache，在这之前，需要判断 cache 中是否有 dirty 的行，如果有，则需要进入 WRITE_BACK 状态，判断是不是需要写回，然后进入 ALLOCATE 状态，将数据从内存中取出，写入 cache。否则，直接进入 ALLOCATE 状态，将数据从内存中取出，写入 cache。

WRITE_BACK

在 WRITE_BACK 状态，cache 需要判断是不是需要写回，逻辑如下

- 如果第 0 块是 dirty 且第一块 lru 为 1，这说明需要替换第 0 块，则将第 0 块的数据写回内存，并设置 dirty 位为 0，然后进入 ALLOCATE 状态；
- 如果第 1 块是 dirty 且第 0 块 lru 为 1，这说明需要替换第 1 块，则将第 1 块的数据写回内存，并设置 dirty 位为 0，然后进入 ALLOCATE 状态；
- 否则，不写回，直接进入 ALLOCATE 状态；

ALLOCATE

在 ALLOCATE 状态，cache 需要从内存中取出数据，然后写入 cache，逻辑如下

- 如果第 0 块的 lru 为 1，则替换第一块，设置第 1 块的 tag，dirty=0，lru=1，data 写入第 1 块，然后将第 0 块的 lru 设置为 0；进入 COMPARE_TAG 状态；
- 如果第 1 块的 lru 为 1，则替换第 0 块，设置第 0 块的 tag，dirty=0，lru=1，data 写入第 0 块，然后将第 1 块的 lru 设置为 0；进入 COMPARE_TAG 状态；

需要注意的是，在 WRITE_BACK 和 ALLOCATE 状态，如果 MIO_ready 为 0，则 cache 继续在 WRITE_BACK 和 ALLOCATE 状态，直到 MIO_ready 为 1。才开始工作。

对 cache 测试的仿真代码如下

```
`timescale 1ns / 1ps


module tb();
    reg clk;
    reg rst;
    reg [31:0] addr;
    reg [31:0] write_data;
    reg [127:0] mem_data;
    reg [1:0] MemRW;
    reg MIO_ready;
```

```
    wire hit;
    wire wb_op;
    wire [31:0] data_out;
    wire [127:0] mem_out;

    cache U1 (
        .clk(clk),
        .rst(rst),
        .addr(addr),
        .write_data(write_data),
        .mem_data(mem_data),
        .MemRW(MemRW),
        .MIO_ready(MIO_ready),
        .hit(hit),
        .wb_op(wb_op),
        .data(data_out),
        .mem_data_out(mem_out)
    );

    initial begin
        clk = 1;
        rst = 1;
        MemRW = 2'b00;
        #10;
        rst = 0;
        MIO_ready = 1;
        // read miss
        addr = 32'h10000000;
        MemRW = 2'b01;
        mem_data = 128'h11111111222222223333333344444444;
        #40;
        // read miss
        addr = 32'h20000000;
        mem_data = 128'h55555555666666667777777788888888;
        #40;
        // read hit
        addr = 32'h10000002;
        #20;
        addr = 32'h10000003;
        #20;
        // write hit
        MemRW = 2'b10;
        addr = 32'h10000002;
        write_data = 32'haaaaaaaa;
        #20;
```

```
        addr = 32'h10000003;
        write_data = 32'hbbbbbbbb;
        #20;
        // read hit
        MemRW = 2'b01;
        addr = 32'h10000002;
        #20;
        addr = 32'h10000003;
        #20;
        addr = 32'h20000000;//recently used
        #20;
        // write miss 替换 tag10000000 mem_data 为修改后
        MemRW = 2'b10;
        addr = 32'h30000000;
        write_data = 32'h99999999;
        mem_data = 128'heeeeeeeeffffffffccccccccdddddddd;
        #50;
        MemRW = 2'b01;
        addr = 32'h30000000;
        #20;
        addr = 32'h30000002;
        #20;
        addr = 32'h10000003;//read miss 替换tag20000000 不用写回 mem_data
        ↪  默认0
        mem_data = 128'h11111111222222223333333344444444;
        #50;
        $finish;
    end
    always begin
        #5 clk = ~clk;
    end

endmodule
```

仿真波形以及解释在下一章节中分析；

二、实验结果与分析

对 cache 进行仿真测试，得到如下结果：

第一次 read miss:

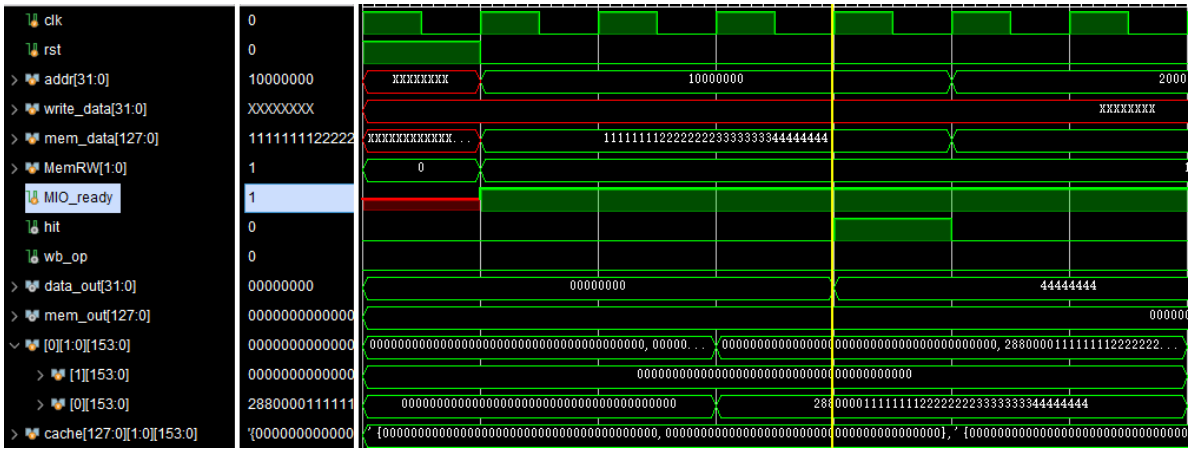

图 2: read miss 1

首先，reset 之后，需要读 addr 为`32'h10000000`的数据，由于 cache 中没有这个数据，所以会产生 read miss，此时，所以在第三个时钟周期，cache 将数据从 memory 中读取到 cache 中，注意到此时第 0 组的第 0 块数据变为了`128'h11111111.....44444444`；同时在下一个时钟周期，hit 信号被置为 1，成功读出`32'h44444444`;

第二次 read miss:

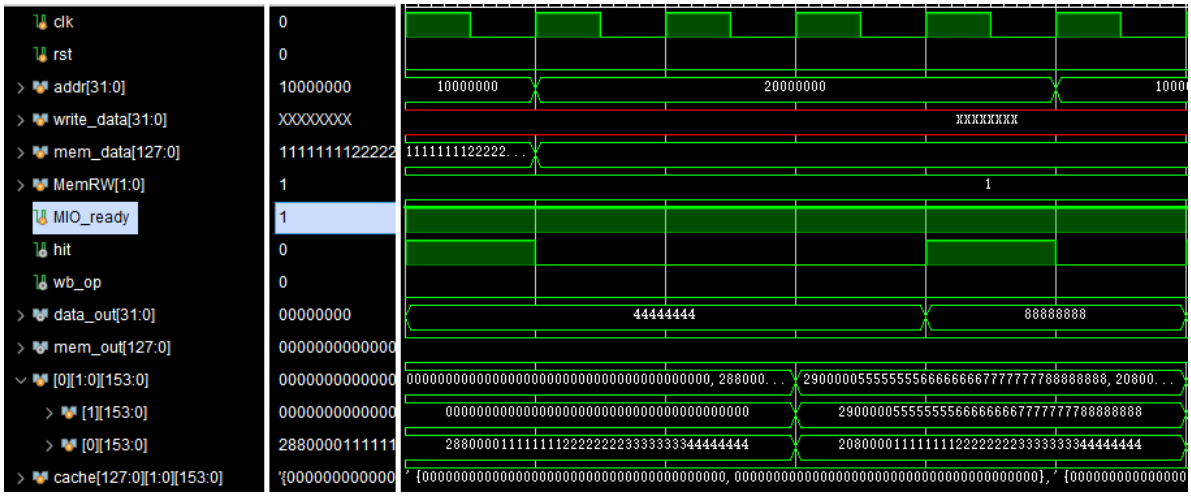

图 3: read miss 2

与第一次 read miss 类似，需要读 addr 为`32'h20000000`的数据，由于 cache 中没有这个数据，所以会产生 read miss，此时，所以在第三个时钟周期，cache 将数据从 memory 中读取到 cache 中，注意到此时第 0 组的第 1 块数据变为了`128'h55555555.....88888888`；同时在下一个时钟周期，hit 信号被置为 1，成功读出`32'h88888888`;

read hit 之后 write hit:

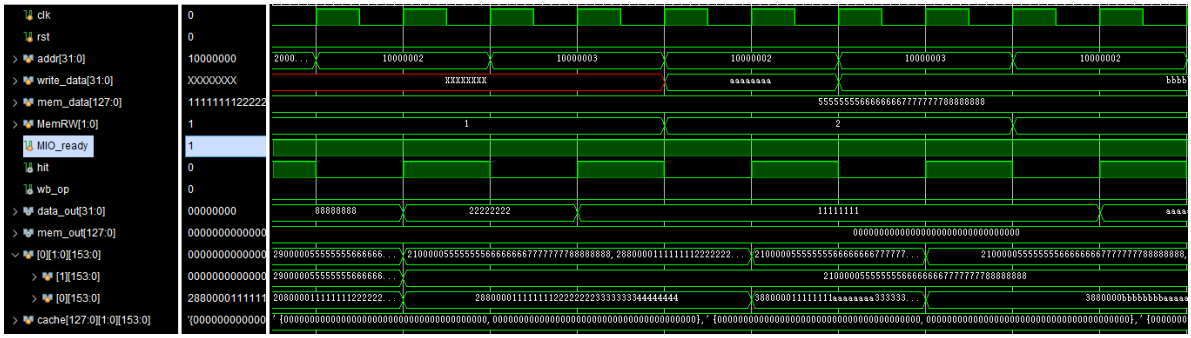

图 4: read hit 和 write hit

如图，在 hit 读出`32'h10000002`的数据和`32'h10000003`的数据之后，紧接着进行 write hit，将`32'h10000002`的数据修改为`32'haaaaaaaa`，此时，cache 中第 0 组的第 1 块数据有相应的变化，然后将`32'h10000003`的数据修改为`32'hbbbbbbbb`，此时，cache 中第 0 组的第 1 块数据有相应的变化，

write hit **之后** read hit:

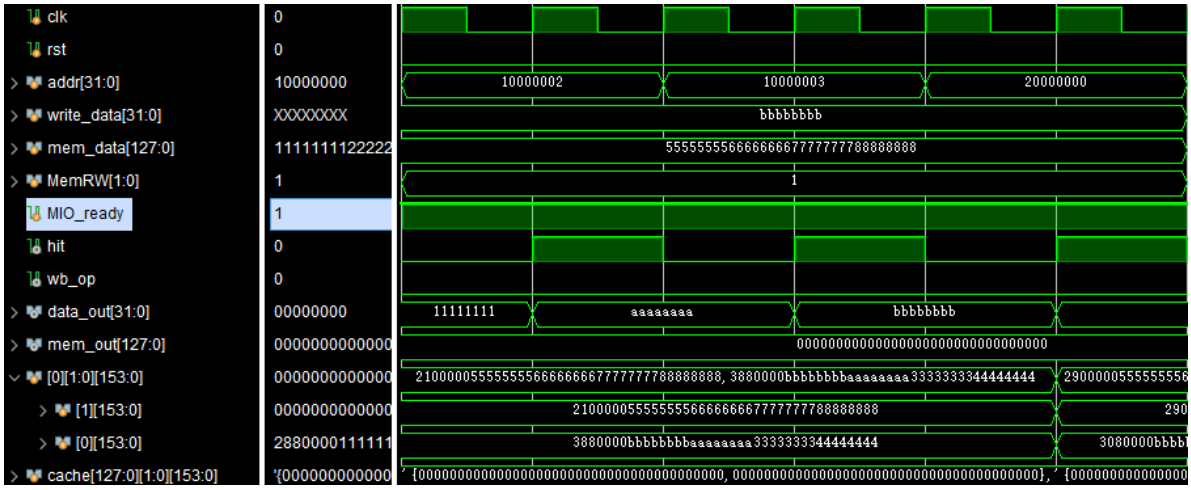

图 5: write hit 和 read hit

如图，依次读出修改后的`32'h10000002`和`32'h10000003`的数据，为`32'haaaaaaaa`和`32'hbbbbbbbb`，之后再读出`32'h20000000`的数据，为之后验证 dirty 位 write back 的正确性做准备；

write miss **验证** write back:

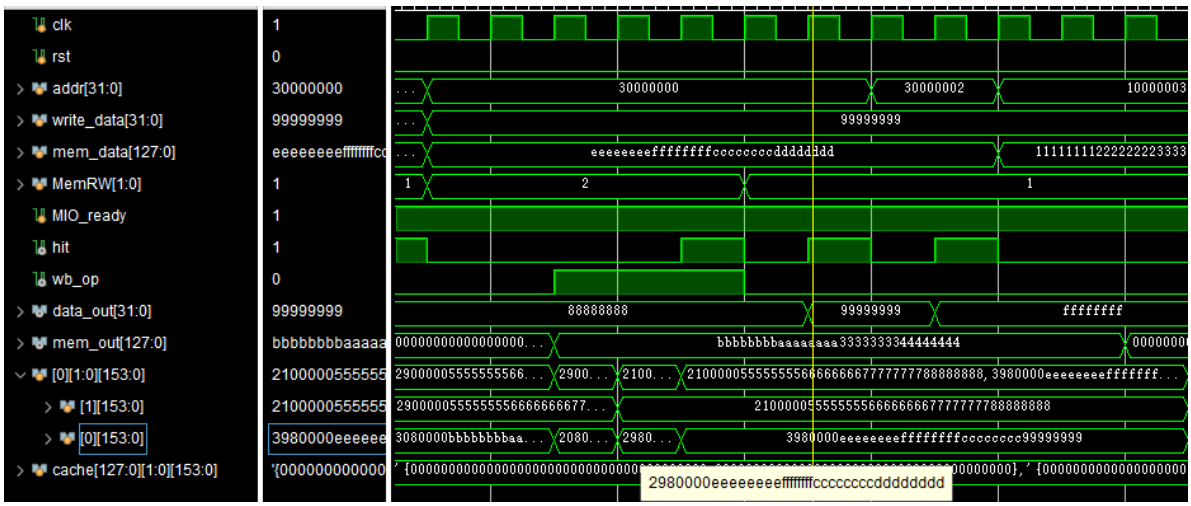

图 6: write miss 验证 write back

由图可知，在 addr 为`32'h30000000`的 write miss 之后，在第三个时钟周期进入 write back 状态，此时 wb_op 为 1，代表写回同时 mem out 的值为经过修改的第 0

块的值 (因为最近 read 了第 1 块), 然后进入 allocate 状态，此时 cache 中第 0 组的第 0 块数据变为了 `128'heeeeeeeeffffffffcccccccddddddddd`，再下一个时钟周期，write hit, 将新写入的第 0 块最后一个字写入 memory 中，此时的值为 `128'heeeeeeeeffffffffcccccccc99999999`，

read miss 验证没有 write back 替换:

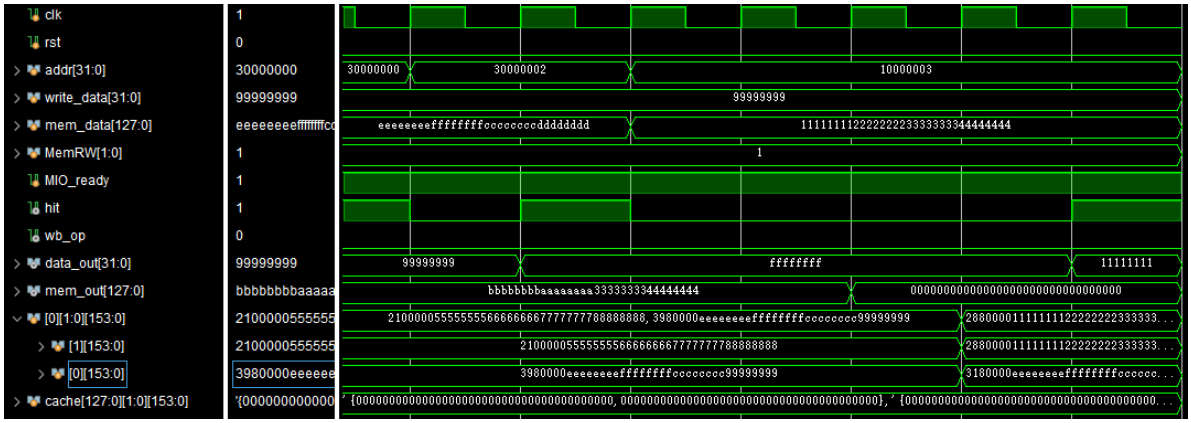

图 7: read miss 验证没有 write back 替换

读出 `32'h30000000` 的数据为 `32'h99999999`, 读出 `32'h30000002` 的数据为不受影响的 `32'hffffffff`，之后读出 `32'h10000003` 的数据，此时遇到 read miss，虽然此时有一块的数据被修改过，但是由于它的 lru 为 1，所以不会被替换，write back 阶段输出的值为 0，在接下来的 allocate 阶段，将最不常用的第 1 块数据替换为 `128'h11111111222222223333333344444444`，最后 hit 读出 `32'h10000003` 的数据为 `32'h44444444`，

至此，实验的验证完成。

三、讨论与实验心得

本次实验作为计组这门课的最后一次实验，与单周期处理器和流水线处理器相比，cache 的实验难度就比较小了，也让我们开心地、结束了这门课的所有实验，总得来说，cache 只需要按照 PPT 上给的流程图，用有限状态机实现即可，与第一次实验的信任检测器类似；在仿真时，我一开始没有考虑好需要 write back 的条件，导致仿真结果与预期不符，后来经过仔细思考解决了这个问题；

完结撒花！

思考题

本实验没有思考题