计算机组成与设计
实验报告

课程名称：	计算机组成与设计
姓 名:	王晓宇
学 院:	计算机科学与技术学院
专 业:	计算机科学与技术
指导教师:	刘海风
报告日期:	2024年 6 月 13 日

Lab6 Cache

课程名称： 计算机组成与设计 实验类型： 综合

实验项目名称： Lab6 Cache

学生姓名： 王晓宇 学号： 3220104364 同组学生姓名： 无

实验地点： 紫金港东四509室 实验日期： 2024 年 6 月 14 日

1 操作方法与实验步骤

这里我们使用FSM有限状态机来实现Cache，总共四个阶段IDLE、COMPARE、ALLOCATE、WRITE_BACK

```
module Cache(
    input wire clk, // clock
    input wire rst, // reset
    input wire [31:0] data_cpu_write, // data write in (to flush cache and flush memory)
    input wire [127:0] data_mem_read, // data from mem (to flush cache)
    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 [127:0] data_mem_write, // data to mem to change value in memory
    output reg [31:0] data_cpu_read, // data to cpu read from cache/memory
    output reg [31:0] addr_mem // memory addr. to get/write value from/to memory
);
localparam IDLE = 2'b00;
localparam COMPARE = 2'b01;
localparam ALLOCATE = 2'b10;
```

```
19  localparam WRITE_BACK = 2'b11;
20  reg [1:0] state;
21  reg [153:0]line[127:0][1:0];
22  wire [1:0] offset = addr_cpu[1:0];
23  wire [6:0] index = addr_cpu[8:2];
24  wire [22:0] tag = addr_cpu[31:9];
25
26  wire Hit_1pos = (line[index][0][153:131] == tag)&&(line[index][0]
    [128] == 1'b1);
27  wire Hit_2pos = (line[index][1][153:131] == tag)&&(line[index][1]
    [128] == 1'b1);
28  wire Dirty_1pos = line[index][0][129];
29  wire Dirty_2pos = line[index][1][129];
30  wire Dirty = Dirty_1pos || Dirty_2pos;
31
32  always@(posedge clk or posedge rst)begin
33      if(rst)begin
34          state <= IDLE;
35      end
36      else begin
37          case(state)
38              IDLE:begin
39                  wr_mem <= 1'b0;
40                  rd_mem <= 1'b0;
41                  if(wr_cpu||rd_cpu)begin
42                      state <= COMPARE;
43                  end
44                  else state <= IDLE;
45              end
46              COMPARE:begin
47                  if(Hit_1pos)begin
48                      if(wr_cpu)begin
49                          line[index][0][(offset*32)+:32] <=
    data_cpu_write;
50                          line[index][0][129] <= 1'b1;//dirty
51                          line[index][0][130] <= 1'b1;//lru
```

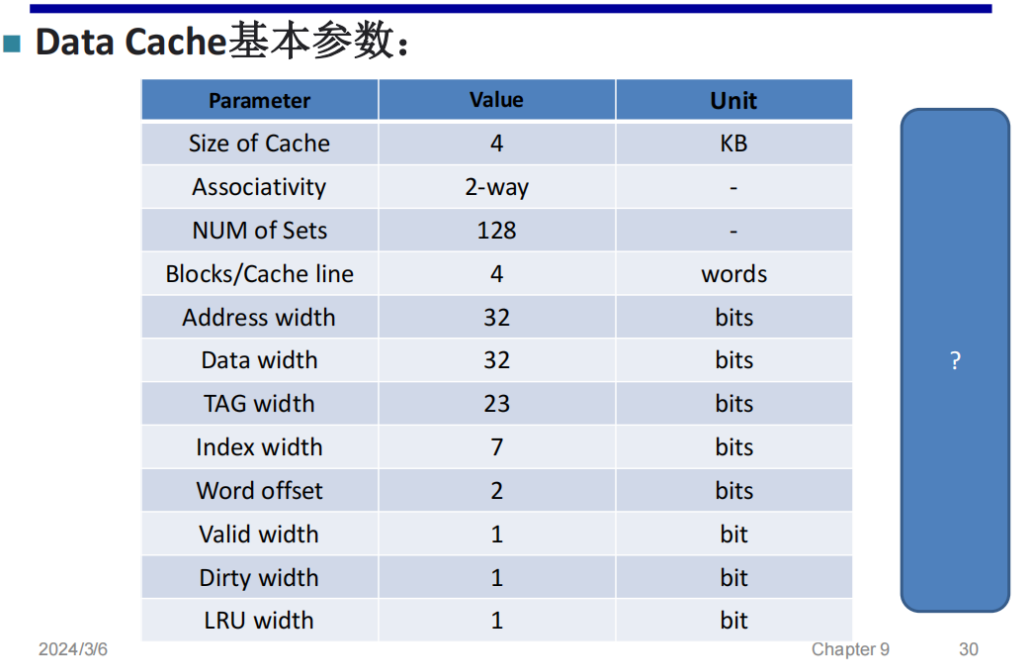
```
52                          line[index][1][130] <= 1'b0;
53                          state <= IDLE;
54                      end
55                      else begin
56                          data_cpu_read <= line[index][0]
[(offset*32)+:32];
57                          state <= IDLE;
58                          rd_mem <=1'b0;
59                      end
60                  end
61                  else if(Hit_2pos)begin
62                      if(wr_cpu)begin
63                          line[index][1][(offset*32)+:32] <=
data_cpu_write;
64                          line[index][1][129] <= 1'b1;//dirty
65                          line[index][1][130] <= 1'b1;//lru
66                          line[index][0][130] <= 1'b0;//lru
67                          state <= IDLE;
68                      end
69                      else begin
70                          data_cpu_read <= line[index][1]
[(offset*32)+:32];
71                          state <= IDLE;
72                          rd_mem <=1'b0;
73                      end
74                  end
75                  else begin
76                      if(Dirty)begin
77                          state<= WRITE_BACK;
78                          wr_mem <= 1'b1;
79                          rd_mem <= 1'b0;
80                      end
81                      else begin
82                          state <= ALLOCATE;
83                          rd_mem <= 1'b1;
84                          rd_mem <= 1'b0;
```

```
                end
            end
        end

        ALLOCATE:begin
            if(ready_mem)begin
                if(line[index][0][130]==1'b0)begin
                    line[index][0][128] <= 1'b1;
                    line[index][0][129] <= 1'b0;
                    line[index][0][127:0] <= data_mem_read;
                    line[index][0][153:131] <= tag;
                end
                else begin
                    line[index][1][128] <= 1'b1;
                    line[index][1][129] <= 1'b0;
                    line[index][1][127:0] <= data_mem_read;
                    line[index][1][153:131] <= tag;
                end
                state <= COMPARE;
            end
            else begin
                state <= ALLOCATE;
            end
        end
        WRITE_BACK:begin
            if(ready_mem)begin
                wr_mem <= 1'b1;
                if(Dirty_1pos)begin
                    data_mem_write <= line[index][0][127:0];
                    line[index][0][129] <=1'b0;//dirty
                end
                else begin
                    data_mem_write <= line[index][1][127:0];
                    line[index][1][129] <=1'b0;//dirty
                end
                state <= ALLOCATE;
```

```
                end
                else begin
                    state <= WRITE_BACK;
                end
            end
        endcase
    end
    addr_mem <= {addr_cpu[31:2],2'b00};
end
endmodule
```

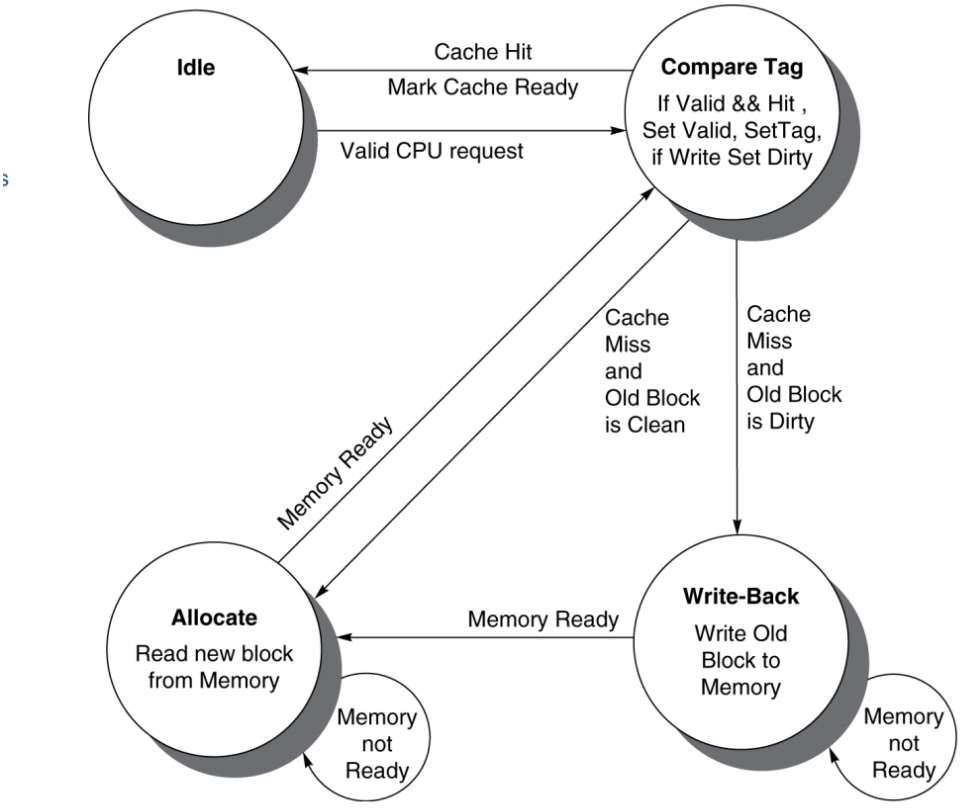
先给出我们要实现的Cache的参数：



看得出来我们要实现两路组相联的 Cache, 采用 LRU 算法进行替换，采用 WriteBack 进行写操作。从 CPU 得到的地址为 32 位，分别为 23 tag bits + 7 index bits + 2 offset bits. 而 cache 一个块是 128 bit 的，共有 $2^7 = 128$ 组，因为是两路组相联的 Cache，所以一共有 2^8个缓存块。我们cache 的大小为 $2^8 \times 16(4\ words) = 2^{12} = 4KB$.

Cache 还需要额外的位来存储信息, valid bit 表示该块是否有效， dirty bit 表示该块是否被修改， lru bit 表示该块的 lru 信息， tag bit 表示该块的 tag 信息。因此实际上 cache 每一个块的大小为154bit

FSM实现Cache，其中的状态转换见下图：



现在分析一下各个状态的作用：

- IDLE:在这个阶段，cache没有进行任何操作。它处于等待状态，准备响应来自CPU的请求。当CPU需要访问数据时，跳入COMPARE。同时如果cache命中之后，我们也会回到这个状态。
- COMPARE:这个状态是比较tag的状态，如果命中返回IDLE状态；如果发现不匹配(1和2路均没有命中)，则进入分情况讨论
  - 若不是脏数据，我们选择跳入ALLOCATE阶段进行数据更新，使用LRU策略
  - 若是脏数据，我们进入WRITEBACK状态去进行数据写入Memory的操作，使得Cache中数据与Memory中数据保持一致，接着进ALLOCATE状态进行处理Miss的状况
- ALLOCATE:处理Miss的状况，从Memory中读取数据，读取完成后更新数据后回到COMPARE状态。这里使用一个LRU替换策略，即更新掉此索引下的LRU位最小的那个，即0。

- WRITEBACK:进入WRITEBACK状态去进行数据写入Memory的操作，使得Cache中数据与Memory中数据保持一致，同时修改脏位为0

2 实验结果与分析

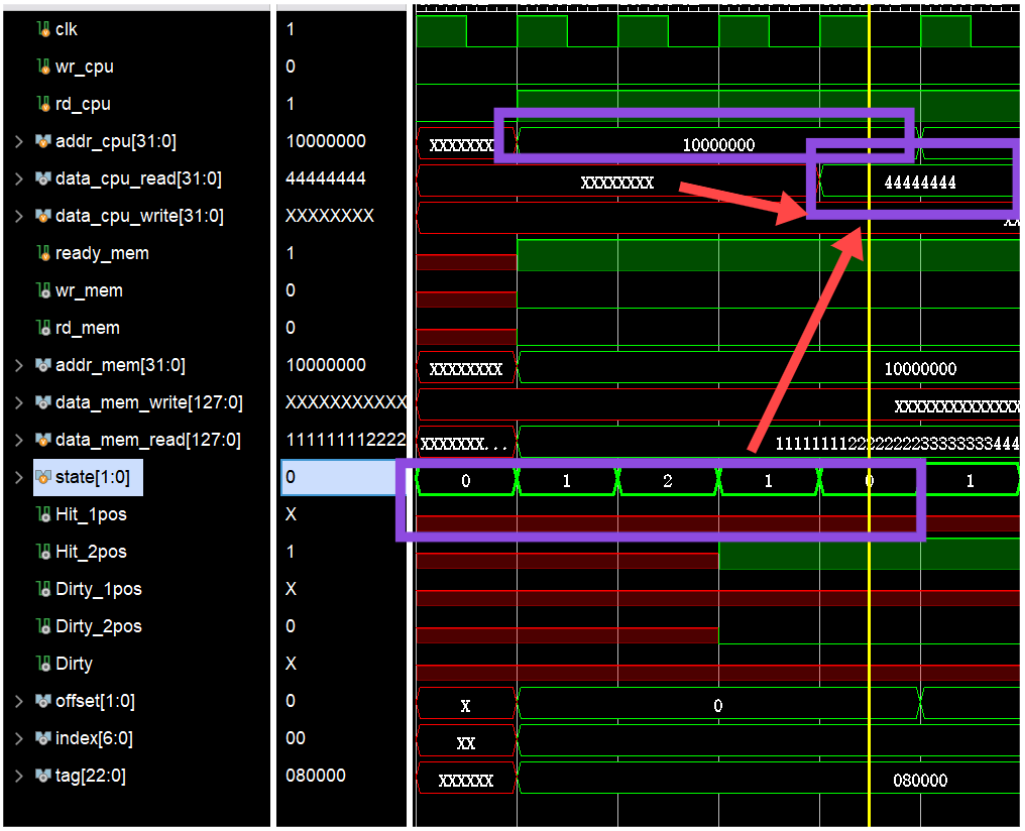
仿真代码：

```
module sim_cache;
reg clk;
reg rst;
reg [31:0] addr_cpu;
reg [31:0] data_cpu_write;
reg [127:0] data_mem_read;
reg wr_cpu;
reg ready_mem;
reg rd_cpu;
wire wr_mem;
wire rd_mem;
wire [127:0] data_mem_write;
wire [31:0] data_cpu_read;
wire [31:0] addr_mem;
initial begin
    clk = 1;
    rst = 1;
    wr_cpu = 0;
    rd_cpu = 0;
    #10;
    rst = 0;
    ready_mem = 1;
    /*Read Miss and then update from the Memory*/
    wr_cpu = 0;
    rd_cpu = 1;
    //Read Miss
    addr_cpu = 32'h10000000;
    data_mem_read = 128'h11111111222222223333333344444444;
    #40;
    //Read mem_read
```

```
31      addr_cpu = 32'h10000002; #40;
32      //Read Miss
33      addr_cpu = 32'h20000000;
34      data_mem_read = 128'h55555555666666667777777788888888;
35      #40;
36      //Read mem_read
37      addr_cpu = 32'h20000001; #40;
38      //Read Miss
39      addr_cpu = 32'h30000002;
40      #40;
41      /*Write into Cache and update memory*/
42      wr_cpu = 1;
43      rd_cpu = 0;
44      addr_cpu = 32'h00000207;
45      data_cpu_write = 32'hAAAAAAAA;
46      #40;
47      //Write to same location
48      addr_cpu = 32'h00000207;
49      data_cpu_write = 32'hFFFFFFFF;
50      #40;
51      //read mem_read
52      rd_cpu = 1'd1;
53      wr_cpu = 1'd0;
54      addr_cpu = 32'h00000207;
55      #40;
56      addr_cpu = 32'h30000207;
57      data_mem_read = 128'hAAAAAAAABBBBBBBBCCCCCCCCDDDDDDDD; #40;
58  end
59  always #5 clk = ~clk;
60  Cache U1 (
61      .clk(clk),
62      .rst(rst),
63      .addr_cpu(addr_cpu),
64      .data_cpu_write(data_cpu_write),
65      .data_mem_read(data_mem_read),
66      .wr_cpu(wr_cpu),
```

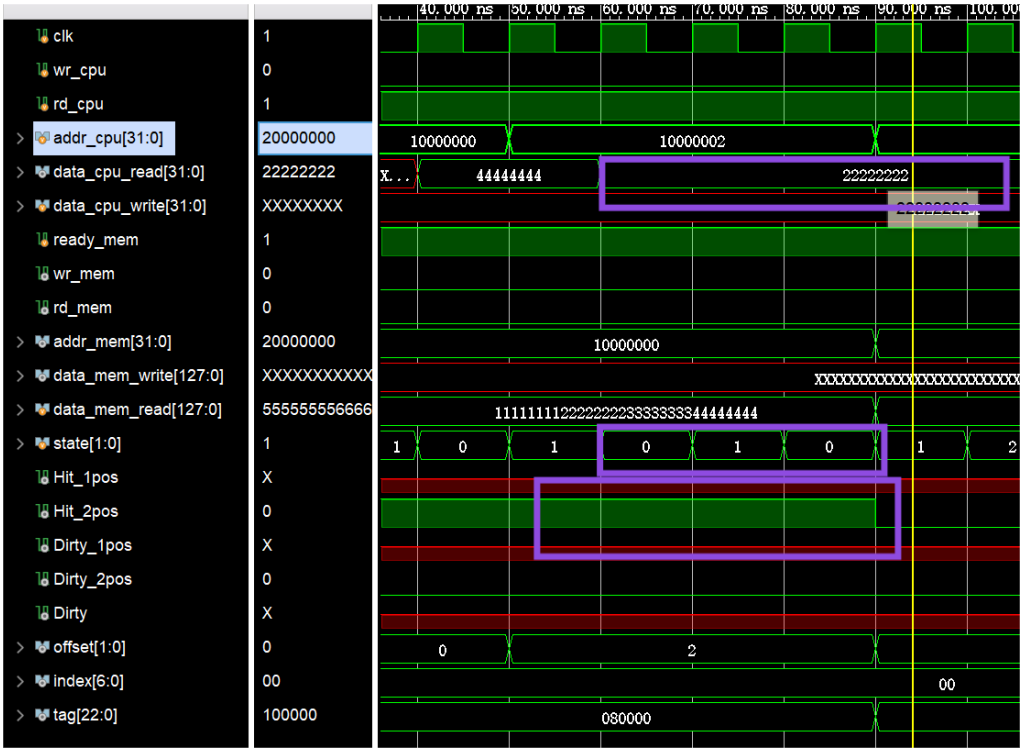
```
    .rd_cpu(rd_cpu),
    .ready_mem(ready_mem),
    .wr_mem(wr_mem),
    .rd_mem(rd_mem),
    .data_mem_write(data_mem_write),
    .data_cpu_read(data_cpu_read),
    .addr_mem(addr_mem)
);
endmodule
```

1. 首先检查Miss后的读取Memory和再读取的命中
    - 首先Cache是空的，现查找`32'h10000000`地址的值，先转入`COMPARE`阶段，由于初始化Cache为空，`Miss`。
    - 进入`ALLOCATE`阶段读入`data_mem_read`传入的值表示从`Memory`读入值更新Cache
    - 更新结束回到`COMPARE`阶段表示hit命中，再恢复`IDLE`阶段
    - 阶段变化为0->1->2->1->0
    - 将`128'h11111111222222223333333344444444`写入Cache后读取第1字节得到`32'h44444444`。



2. 检查一次直接命中

- 首先查找`32'h10000002`地址的值，先转入`COMPARE`阶段，由于初始化Cache已经存在值，`Hit`。
- 命中结束回到`COMPARE`阶段表示hit命中，`Hit_2pos`为1，再恢复`IDLE`阶段
- 阶段变化为0->1->0
- 将`128'h11111111222222223333333344444444`写入Cache后读取第3字节得到`32'h22222222`。



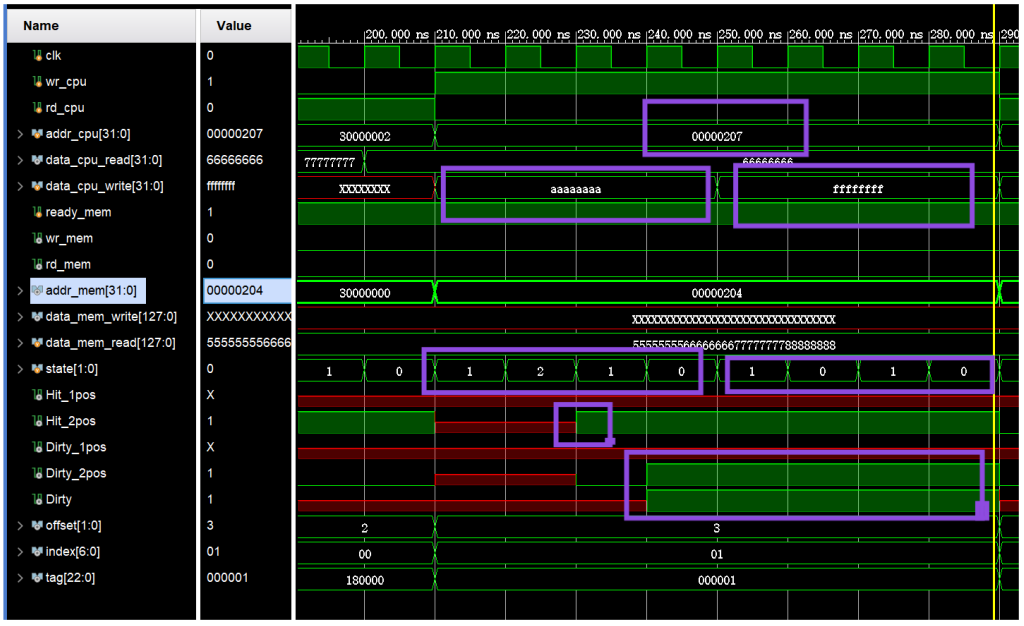
3. 再一次检查Miss后的读取Memory和再读取的命中

- 首先Cache是空的，现查找`32'h20000000`地址的值，先转入`COMPARE`阶段，由于初始化Cache为空，`Miss`。
- 进入`ALLOCATE`阶段读入`data_mem_read`传入的值表示从`Memory`读入值更新Cache
- 更新结束回到`COMPARE`阶段表示hit命中，再恢复`IDLE`阶段
- 阶段变化为0->1->2->1->0
- 将`128'h55555555666666667777777788888888`写入Cache后读取第1字节得到`32'h88888888`。

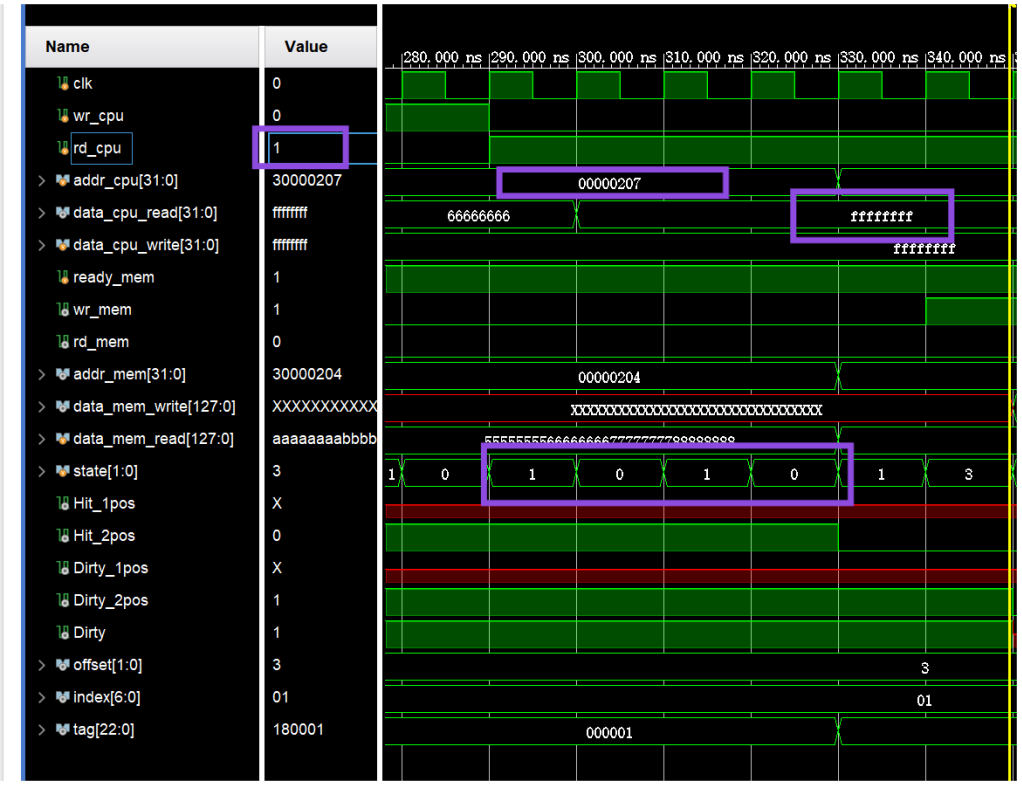
4. 检查一次直接命中
   - 首先查找`32'h20000001`地址的值，先转入`COMPARE`阶段，由于初始化Cache已经存在值，`Hit`。
   - 命中结束回到`COMPARE`阶段表示hit命中，`Hit_2pos`为1，再恢复`IDLE`阶段
   - 阶段变化为0->1->0
   - 已经将`128'h55555555666666667777777788888888`写入Cache后读取第2字节得到`32'h77777777`。

5. 检查脏数据后的更新

- 首先向`32'h00000207`写入数据`32'hAAAAAAAA`，由于`Miss`，需要从Memory中读取，这里没有对`data_mem_read`修改，仍然是`128'h55555555666666667777777788888888`，
- 读取到之后写入Cache，之后对Cache的第4位进行修改`32'hFFFFFFFF`，同时脏位变1
- 之后再对Cache同一block再次进行修改，脏位仍不变，为1，现在的`32'h00000207`地址的数据应该变为`128'hffffffff66666666777777778888888`

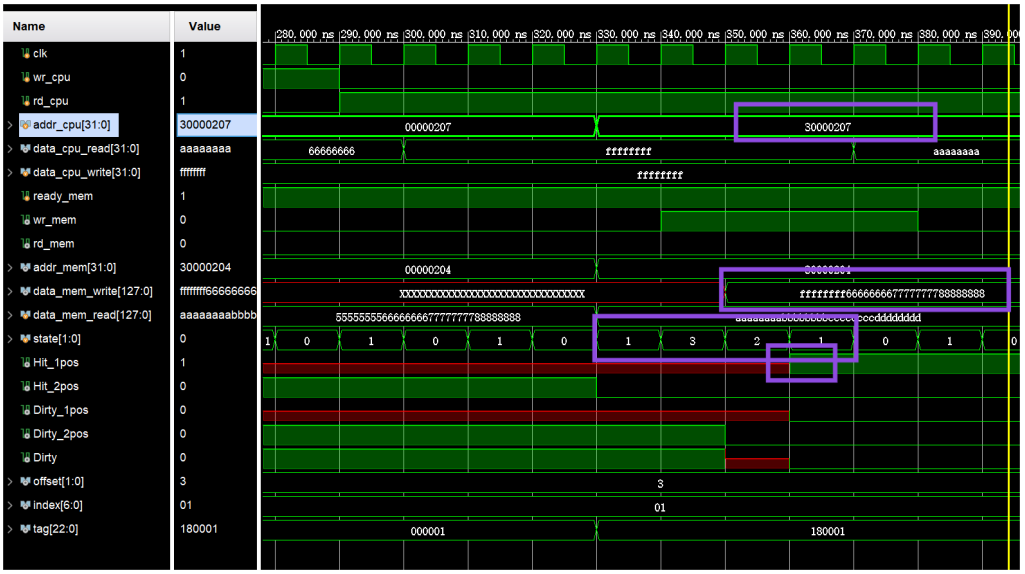


- 读取一下cache内容是否真的更改：



6. 进行Cache 写回

- 对32'h30000207地址数据进行读取，再COMPARE处miss，而且有脏数据，先进入WRITEBACK阶段进行写回操作
- 由于是第二个位置的脏数据，所以这里输出信号可以看到上一步在cache中更改完的128'hffffffff66666666777777788888888，与之前的128'h55555555666666667777777788888888不同。更改脏位为0
- 之后跳入ALLOCATE阶段，对32'h30000207的数据进行cache读，之后跳回COMPARE阶段， HIT
- 阶段变化为0->1->3->2->1->0



3 讨论、心得

Cache利用了有限状态机去实现了对内存的快速访问，同时本次实验也帮助我们复习了组相联的有关知识，对Cache的结构有了进一步认识，难度不算很大

这次是正儿八经的收官（，计组我真的很爱你啊、、、你猜我说的是真的不

今天是2024年6月13日，致敬传奇热带风味破防哥，祝瓜助教毕业快乐&其他小伙伴期末复习顺利，门门满绩！

海阔山遥，未知何处是潇湘！

4 思考题

> 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？

- 设计方法基本一致，两者都可能使用相同的替换策略，如最近最少使用（LRU）或先进先出（FIFO）等，来决定哪些缓存行应该被替换，只是指令内容一般是固定的，不存在写回，也就是说可以减少状态机状态数。
- 不存在写回、写分派现象。在数据缓存中，写回是指当缓存行被修改后，最终将这些修改写回主存储器的过程。指令缓存通常不涉及写回，因为指令不应该被修改。
- 如果需要修改时，如果命中，则在cache上进行修改即可，另外将脏位设计为0，使得下次寻找机会写回ROM；如果Miss，使用LRU策略去选择合适的位置，将ROM中的值读入到Cache中，进行Cache的更新。

> 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？

- 流水线将指令执行过程分解为多个阶段，如取指（IF）、解码（ID）、执行（EX）、访存（MEM）和写回（WB），我们这里先只考虑对数据的缓存，对指令的缓存原理接近。
- 对数据的访存一定发生在MEM阶段，这个阶段利用cache接口进行对数据的访存，如果命中，则在Cache上进行原地修改，假如Cache失效，而Cache取存数据的的周期小于流水线CPU的时钟频率对于的一个周期，则不必停顿，等待取回即可；假如Cache失效，而Cache取存数据的的周期大于流水线CPU的时钟频率对于的一个周期，则流水线CPU需要整体STALL进行数据提取和存储。根据替换策略的不同，Cache访问Memory的速度不同，此时要根据实际情况选择合适的方案来停顿(如前文提到)。