浙江大学

# 本科实验报告

课程名称：计算机组成与设计

姓　　名：秦嘉俊

学　　院：竺可桢学院

专　　业：计算机科学与技术

指导教师：刘海风

2023 年 6 月 10 日

浙江大学实验报告

Lab6

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：秦嘉俊　学号：3210106182　同组学生姓名：张瑞

实验地点：玉湖七幢 629　实验日期：2023 年 6 月 11 日

# 一、操作方法与实验步骤

## 1.1 设计缓存模块 Cache

新建工程 Lab6, 进入工程新建源文件 cache.v, 输入如下 Verilog 代码

```
1
2  `define IDLE 2'd0
3  `define COMPARE_TAG 2'd1
4  `define ALLOCATE 2'd2
5  `define WRITE_BACK 2'd3
6
7  module cache(
8      input clk,
9      input rst,
10     input [31:0] addr,
11     input [31:0] write_data,  // 1 word 要写的字
12     input [127:0] mem_data,   // 4 words 从内存中来的
13     input [1:0] MemRW,        // 0 表示无请求，1 表示读，2 表示写
14     input memory_ready,
15     output reg MemRW_out,         // miss 时要从内存里读还是写
16     output reg ready,               // 表示 cache 读写是否完成
17     output reg [127:0] mem_data_out, // dirty bit 的时候要写回
18     output reg [31:0] data
```

```
19      );
20      reg [153:0] cache_data [127:0][1:0];
21
22      wire [1:0] offset = addr[1:0];
23      wire [6:0] index = addr[8:2];
24      wire [22:0] tag = addr[31:9];
25      reg [1:0] state;
26
27      always @(posedge clk or posedge rst) begin
28          if (rst) begin
29              state <= 'IDLE;
30          end
31          else begin
32              case(state)
33                  'IDLE: begin
34                      MemRW_out <= 0;
35                      ready <= 0;
36                      // 是否有有效请求
37                      if (MemRW == 1 || MemRW == 2) state <= 'COMPARE_TAG;
38                      else state <= 'IDLE;
39                  end
40                  'COMPARE_TAG: begin
41                      // hit
42                      if (cache_data[index][0][153] == 1'b1 && cache_data[
                            index][0][150:128] == tag) begin
43                          if (MemRW == 2) begin
44                              cache_data[index][0][(offset*32)+:32] <=
                                    write_data;
45                              cache_data[index][0][152] <= 1'b1;
46                              cache_data[index][0][151] <= 1'b1;
47                          end
48                          else data <= cache_data[index][0][(offset*32)+:32
                                ];
49                          state <= 'IDLE;
50                          ready <= 1;
51                      end
52                      else if (cache_data[index][1][153] == 1'b1 &&
                            cache_data[index][1][150:128] == tag) begin
```

```
53                  if (MemRW == 2) begin
54                      cache_data[index][1][(offset*32)+:32] <=
                            write_data;
55                      cache_data[index][1][152] <= 1'b1;
56                      cache_data[index][1][151] <= 1'b1;
57                  end
58                  else data <= cache_data[index][1][(offset*32)+:32
                        ];
59                  state <= 'IDLE;
60                  ready <= 1;
61              end
62              // miss
63              else begin
64                  if (cache_data[index][0][152] == 1 || cache_data[
                        index][1][152] == 1) begin
65                      state <= 'WRITE_BACK;
66                      MemRW_out <= 1;
67                  end
68                  else begin
69                      state <= 'ALLOCATE;
70                      MemRW_out <= 0;
71                  end
72                  ready <= 0;
73              end
74          end
75          'ALLOCATE: begin
76              if (memory_ready == 1) begin // 访问内存结束了
77              // lru 策略
78                  if (cache_data[index][0][151] == 1) begin // 替换
                        另一个缓存块
79                      cache_data[index][0][151] <= 1'b0;
80                      cache_data[index][1][151] <= 1'b1;
81                      cache_data[index][1][153] <= 1'b1;
82                      cache_data[index][1][152] <= 1'b0;
83                      cache_data[index][1][150:128] <= tag;
84                      cache_data[index][1][127:0] <= mem_data;
85                  end
86                  else begin
```

```
87                      cache_data[index][1][151] <= 1'b0;
88                      cache_data[index][0][151] <= 1'b1;
89                      cache_data[index][0][153] <= 1'b1;
90                      cache_data[index][0][152] <= 1'b0;
91                      cache_data[index][0][150:128] <= tag;
92                      cache_data[index][0][127:0] <= mem_data;
93                   end
94                   state <= 'COMPARE_TAG;
95                end
96                else begin
97                   state <= 'ALLOCATE;
98                end
99             end
100            'WRITE_BACK: begin
101               if (memory_ready == 1) begin
102                  MemRW_out <= 1;
103                  if (cache_data[index][0][152] == 1) begin
104                     mem_data_out <= cache_data[index][0][127:0];
105                     cache_data[index][0][152] <= 0; // reset dirty
                           bit
106                     state <= 'ALLOCATE;
107                  end
108                  else begin
109                     mem_data_out <= cache_data[index][1][127:0];
110                     cache_data[index][1][152] <= 0; // reset dirty
                           bit
111                     state <= 'ALLOCATE;
112                  end
113               end
114               else state <= 'WRITE_BACK;
115            end
116         endcase
117      end
118   end
119
120endmodule
```

这里我们实现的两路组相联的 Cache, 采用 LRU 算法进行替换，采用 WriteBack 写回

法进行写操作。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，故共有 $2^8$ 个缓存块。因此我们 cache 的大小为 $2^8 \times 16(4\ words) = 2^{12} = 4KiB$.

实际中 cache 还需要额外的位来存储信息，其中 valid bit(1) + dirty bit(1) + lru bit(1) + tag bit(23), 这里 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 154 bit.

具体实现 cache 是通过 FSM 的方式，我们按照 PPT 的方式定义了四个状态: `IDLE`(默认), `COMPARE_TAGE`(比较 tag), ALLOCATE(从内存中读数据到缓存), WRITE_BACK(将缓存中的数据写回内存). 其中的状态转换见下图：

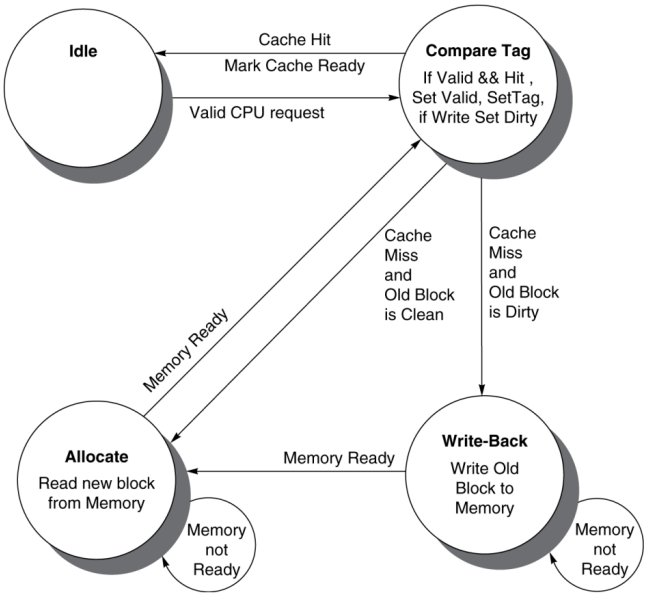


图 1: cache FSM

下面具体介绍每个部分的思路

- `IDLE`
  在这个状态下，我们首先判断是否有读请求，如果有则进入 `COMPARE_TAG` 状态，否则保持在 `IDLE` 状态。MemRW 表示我们的请求，如果为 0 说明没有对 cache 发出请求，为 1 为读请求, 2 为写请求，

- `COMPARE_TAGE`
  在这个状态下，我们首先判断是否命中，如果命中则进入 `IDLE` 状态，否则进入 `ALLOCATE` 状态。

判断是否命中的方法是将 cache 中的 tag 与 CPU 给出的 tag 进行比较，如果相同则命中，否则不命中。

- `ALLOCATE`
  在这个状态下，我们需要从内存中将 addr 对应的块搬运到 cache 中来，同时设置相关的位。但是如果对应组已经没有空闲位置，我们需要根据 LRU 策略将一个块替换出去，即根据 151 位判断最近是否有访问过。这里我们只需要看该组第 1 个缓存块即可，如果这个缓存块被访问过那么我们直接替换第 2 个缓存块，否则替换第一个缓存块。

  将块搬运到 cache 后，我们回到 `COMPARE_TAGE` 状态，相当于再次执行读写 cache 操作。

- `WRITE_BACK`
  在这个状态下，我们需要将 cache 中的数据写回内存，同时清空 dirty bit. 写回成功后我们进入 `ALLOCATE` 状态。

## 1.2 仿真验证

新建仿真文件 cache_tb.v, 输入如下代码

```
module cache_tb;
reg clk;
reg rst;
reg [31:0] cpu_addr;
reg [31:0] write_data;   // 1 word 要写的字
reg [127:0] mem_data;    // 4 words 从内存中来的
reg [1:0] MemRW;
reg memory_ready;
wire MemRW_out;          // miss 时要从内存里读还是写
wire hit;
wire [127:0] mem_data_out; // dirty bit 的时候要写回
wire [31:0] data;

initial begin
   clk = 1;
   rst = 1;
   MemRW = 0;
   #10;
```

```
        rst = 0;
        memory_ready = 1;
        // Read Miss
        cpu_addr = 32'h10000000;
        MemRW = 1;
        mem_data = 128'h11111111222222223333333344444444;
        #40;
        // Read Miss
        cpu_addr = 32'h20000000;
        mem_data = 128'h55555555666666667777777788888888;
        #40;
        // Read Hit
        cpu_addr = 32'h10000002;#20;
        cpu_addr = 32'h20000001;#20;
        // Write Hit
        MemRW = 2;
        cpu_addr = 32'h10000001;    // 写第一个字
        write_data = 32'hAAAAAAAA;
        #20;
        cpu_addr = 32'h20000002;    // 写第二个字
        write_data = 32'hBBBBBBBB;
        #20;
        // Read Hit 检验刚刚写的内容是否被写进去了
        MemRW = 1;
        cpu_addr = 32'h10000001;#20;
        cpu_addr = 32'h20000002;#20;

        // Write miss, write back and allocate
        MemRW = 2;
        cpu_addr = 32'h30000000;    // 需要驱赶一个块
        write_data = 32'hFFFF0000;
        mem_data = 128'hCCCCCCCCDDDDDDDDEEEEEEEEFFFFFFFF;
        #50;
        MemRW = 1;
        cpu_addr = 32'h30000000;#20;
        cpu_addr = 32'h30000001;#20;
    end
    always #5 clk = ~clk;
```

```
57
58      cache U1(.clk(clk), .rst(rst), .addr(cpu_addr), .write_data(
            write_data), .mem_data(mem_data), .MemRW(MemRW), .
            memory_ready(memory_ready), .MemRW_out(MemRW_out), .ready(hit
            ), .mem_data_out(mem_data_out), .data(data));
59  endmodule
```

具体的仿真结果及分析见第二部分。

# 二、实验结果与分析

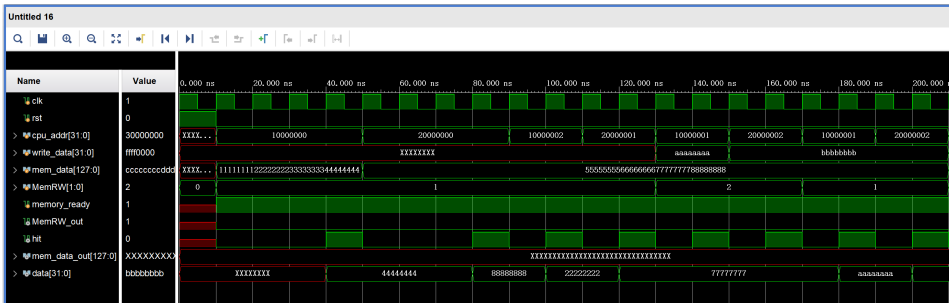


图 2: cache 仿真结果 (1)

首先我们 reset cache, 随后将 MemRW 置为 1, 表示我们要执行读操作。送入地址 1000_0000, 对应第 0 组的块, tag 为 100000. 此时 cache 里还没有任何数据，因此会 read miss. 于是我们会把 `mem_data` 写入到 cache 中去，随后我们再次读缓存数据（第四个时钟周期，即 40ns 时），可以看到 hit 变为了 1 说明缓存命中，而此时 `data` 也变为了我们要读取的数据，即 4444_4444. 过程中我们的状态依次是 IDLE->COMPARE_TAG->ALLOCATE->COMPARE_TAG->IDLE.

随后读地址 2000_0000 过程与刚刚相同，也是 read miss. 从内存中读出块之后, cache 第 0 组里的两个块都已经放置了内存的有效数据。

于是接下来我们尝试读这个块里的其他 word, 送入地址 1000_0002. 这里对应的是第 0 组的 tag 为 100000 的块的第 2 个字，因此我们取出了 2222_2222. 同理我们后面送入地址 2000_0001, 对应第 0 组的 tag 为 200000 的块的第 1 个字，因此我们取出了 7777_7777. 这个过程中我们的状态依次是 IDLE->COMPARETAG->IDLE, 因此可以看到在第二个周期时 hit 就已经变为了 1 而 `data` 也变为了要读取的数据。

随后我们将 MemRW 变为 2 表示执行写操作。首先我们送入地址 1000_0001 表示对第 0 组的 tag 为 100000 的块的第 1 个字进行写操作 (写为 *aaaa_aaaa*)。同理 2000_0002 表示对第 0 组的 tag 为 200000 的块的第 2 个字进行写操作 (写为 *bbbb_bbbb*). 这个过程

中我们的状态依次是 IDLE->COMPARETAG->IDLE, 因此可以看到在第二个周期时 hit 就已经变为了 1.

写操作结束后，我们立刻读刚刚的地址，可以通过 `data` 看到我们依次读出了 *aaaa_aaaa*,*bbbb_bbbb*, 说明我们成功写入了 cache.

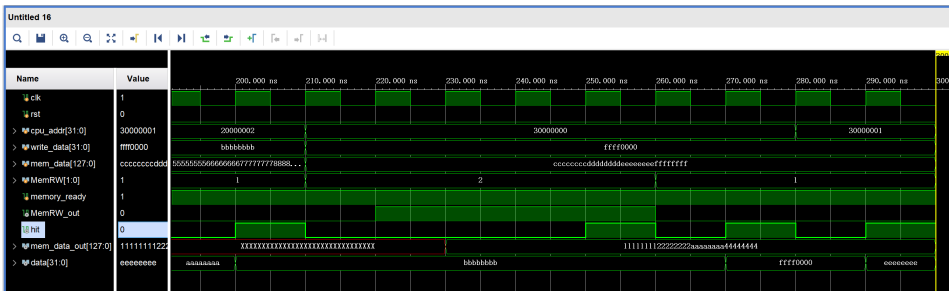


图 3: cache 仿真结果 (2)

最后我们尝试写地址 3000_0000, 这对应第 0 组的 tag 为 300000 的块的第 0 个字。这里因为之前的操作第 0 组的 cache 已经填满，我们需要根据 LRU 策略选择一个块驱赶出去。这里因为 tag 为 100000 的块访问的时间更早，我们会选择驱赶这个块，同时因为这个块已经被修改 (即脏位为 1), 在驱赶之前还要把数据写回到内存中。因此在第三个周期可以看到 `mem_data_out` 变为了这个块被修改后的数据, `MemRW_out` 也变为了 1 表示要对内存进行写操作。写回脏位后我们要从内存里把这个新的块的数据搬到 cache 中来，随后第五个周期 (250ns) 可以看到 hit 变为了 1, 这时就完成了对数据的修改. 这个过程中我们的状态依次是 IDLE->COMPARE_TAG->WriteBack->ALLOCATE->COMPARE_TAG->IDLE.

随后我们读地址 3000_0000 和 3000_0001, 可以看到 3000_0000 的值就是我们刚刚写入的 *ffff*_0000, 而 3000_0001 的值就是我们从内存中搬进来的块的数据，即 *eeee_eeee*.

# 三、讨论、心得

## 心得

提前开香槟了，原来这里还有一次实验（雾）。

Cache 本身的设计还是比较简单的，但是在实现的过程中还是遇到了一些问题，好在没有花费太多时间，理解了理论课上的知识之后，实现起来还是比较顺利的。虽然我们实现的 Cache 比较简陋，但是在实际应用中，Cache 的设计还是比较复杂的，比如 Intel 的 Cache 有三级，而且还有很多优化的技巧，比如预取，写缓冲等等。这些都是我们可以继续学习的。以后龙芯杯也可能用到。

终于终于结束了硬件实验，颇有感慨，收获甚多，感谢老师，感谢助教，感谢不放弃的自己。

江湖路远，后会有期。