# 《计算机组成与设计》
# 实验报告


姓名: 刘韬

学院: 竺可桢学院

专业: 人工智能

邮箱: 3220103422@zju.edu.cn


报告日期: 2024-06-15

# cache 设计与实现

## 1 模块设计

### 1.1 cache 参数

首先是 cache 的参数

| Parameter | Value | Unit |
|---|---|---|
| Size of Cache | 4 | KB |
| Associativity | 2-way | - |
| NUM of Sets | 128 | - |
| Blocks/Cache line | 4 | words |
| Address width | 32 | bits |
| Data width | 32 | bits |
| TAG width | 23 | bits |
| Index width | 7 | bits |
| Word offset | 2 | bits |
| Valid width | 1 | bit |
| Dirty width | 1 | bit |
| LRU width | 1 | bit |

图 1：cache 参数

我们实现的是 2-ways set cache.每一个 block 是 4words.因此每个 cacheline 是 128bits data + 23 bits tags + 1 bit valid + 1 bit dirty + 1 bit lru = 153 bits.那么我们 cache 就是初始化 128 组，每组 2 个 cacheline，每个 cacheline 153 bits。

### 1.2 cache 实现

新建一个 cache 模块

```
1  define IDLE 2'd0
2  `define COMPARE_TAG 2'd1
3  `define ALLOCATE 2'd2
4  `define WRITE_BACK 2'd3
5
6  module cache(
7      input         clk,
8      input         rst,
9      input [31:0] addr_cpu,
10     input [31:0] data_cpu_write,
11     input [127:0]data_mem_read,
```

```
12      input [1:0]  memRW,
13      input        ready_mem,
14      output reg   memRW_out,
15      output reg [31:0] data_cpu_read,
16      output reg [127:0] data_mem_write,
17      output reg [31:0] addr_mem,
18      output reg ready,
19      output reg [1:0] state
20      );
21      reg [153:0] cache_data [127:0][1:0]; // 128 sets, 2 ways 153:valid,
    152:dirty, 151:lru, 150:128:tag, 127:0:data
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      // 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                      if(memRW == 1 || memRW == 2) begin
37                          state <= `COMPARE_TAG;
38                      end
39                      else begin
40                          state <= `IDLE;
41                      end
42                  end
43                  `COMPARE_TAG: begin
44                      // if valid and hit
45                      if(cache_data[index][0][153] == 1'b1 &&
    cache_data[index][0][150:128] == tag)begin
46                          if (memRW == 2) begin // Write
47                              cache_data[index][0][(offset*32)+:32] <=
    data_cpu_write;
48                              cache_data[index][0][152] <= 1'b1;
49                              cache_data[index][0][151] <= 1'b1;
50                              cache_data[index][1][151] <= 1'b0;
51                          end else begin // Read
52                              cache_data[index][0][151] <= 1'b1;
53                              cache_data[index][1][151] <= 1'b0;
54                              data_cpu_read <= cache_data[index][0]
    [(offset*32)+:32];
55                          end
56                          state <= `IDLE;
57                          ready <= 1;
58                      end
59                      else if(cache_data[index][1][153] == 1'b1 &&
    cache_data[index][1][150:128] == tag) begin
60                          if (memRW == 2) begin
61                              cache_data[index][1][(offset*32)+:32] <=
    data_cpu_write;
62                              cache_data[index][1][152] <= 1'b1;
63                              cache_data[index][1][151] <= 1'b1;
64                              cache_data[index][0][151] <= 1'b0;
65                          end else begin
66                              cache_data[index][1][151] <= 1'b1;
```

```
67                              cache_data[index][0][151] <= 1'b0;
68                              data_cpu_read <= cache_data[index][1]
    [(offset*32)+:32];
69                          end
70                          state <= `IDLE;
71                          ready <= 1;
72                      end
73                      // miss
74                      else begin
75                          // if dirty
76                          if(cache_data[index][0][152] == 1 ||
    cache_data[index][1][152] == 1) begin
77                              memRW_out <= 1;
78                              state <= `WRITE_BACK;
79                          end
80                          // if not dirty
81                          else begin
82                              memRW_out <= 0;
83                              state <= `ALLOCATE;
84                          end
85                          ready <= 0;
86                      end
87                  end
88                  `ALLOCATE: begin
89                      // if memory has read
90                      if(ready_mem == 1)begin
91                          if(cache_data[index][0][151] == 1)begin
92                              cache_data[index][0][151] <= 0;
93                              cache_data[index][1][153] <= 1;
94                              cache_data[index][1][152] <= 0;
95                              cache_data[index][1][151] <= 1;
96                              cache_data[index][1][150:128] <= tag;
97                              cache_data[index][1][127:0] <= data_mem_read;
98                          end else begin
99                              cache_data[index][1][151] <= 0;
100                             cache_data[index][0][153] <= 1;
101                             cache_data[index][0][152] <= 0;
102                             cache_data[index][0][151] <= 1;
103                             cache_data[index][0][150:128] <= tag;
104                             cache_data[index][0][127:0] <= data_mem_read;
105                         end
106                         state <= `COMPARE_TAG;
107                     end else begin // wait to memory read
108                         if(cache_data[index][0][151] == 1) begin
109                             addr_mem <= {cache_data[index][1][150:128],
    index, 2'b00};
110                         end else begin
111                             addr_mem <= {cache_data[index][0][150:128],
    index, 2'b00};
112                         end
113                         state <= `ALLOCATE;
114                     end
115                 end
116                 `WRITE_BACK: begin
117                     if(ready_mem == 1)begin
118                         if(cache_data[index][0][152] == 1)begin
119                             addr_mem <= {cache_data[index][0][150:128],
    index, 2'b00};
120                             data_mem_write <= cache_data[index][0][127:0];
121                             cache_data[index][0][152] <= 0;
122                             if(cache_data[index][1][152] == 0)begin
123                                 state <= `ALLOCATE;
```

```
124                             end else
125                                 state <= `WRITE_BACK;
126                         end
127                         else if(cache_data[index][1][152] == 1)begin
128                             addr_mem <= {cache_data[index][1][150:128],
    index, 2'b01};
129                             data_mem_write <= cache_data[index][1][127:0];
130                             cache_data[index][1][152] <= 0;
131                             state <= `ALLOCATE;
132                         end
133                     end
134                 end
135             endcase
136         end
137     end
138 endmodule
```

下面是 cache 的设计思路，我们基于下面的有限状态机进行设计。

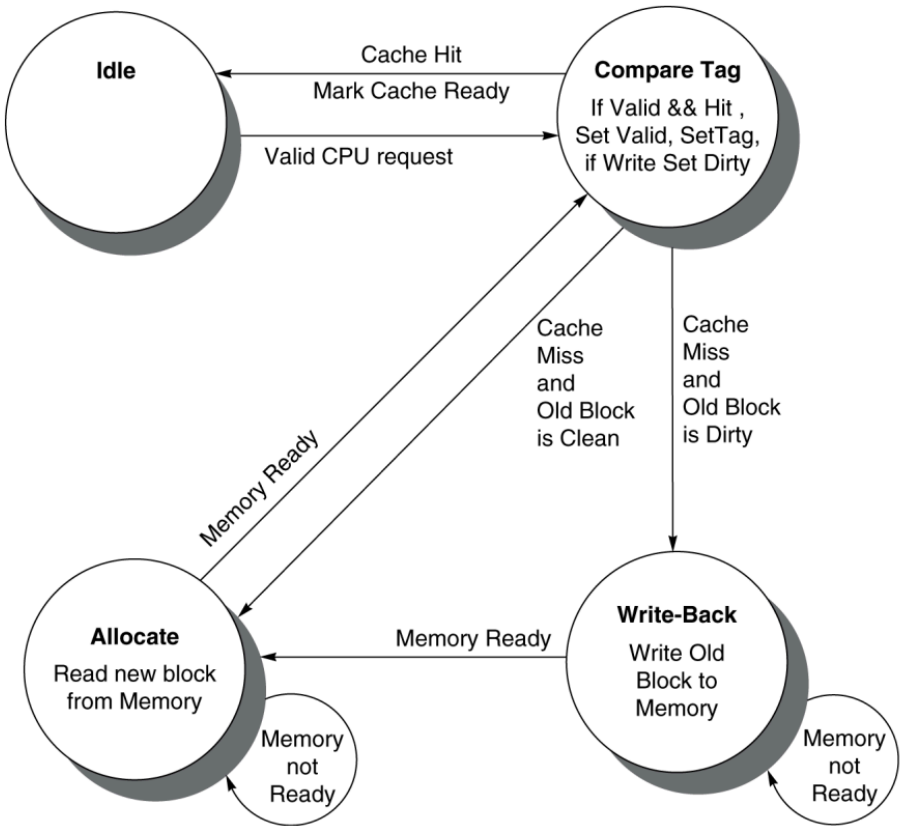


图 2：有限状态机

整体的思路是根据请求判断是否命中，如果命中就执行请求，如果没有命中说明要去内存中找，这里先判断 cache 中是否与内存一致，如果不一致先写回内存，然后再去内存中找。注意要做整个 block 的替换。

### 1.2.1 IDLE

在 IDLE 状态下，cache 不做任何事情，等待 CPU 给出 memRW 信号。这也是 cache 的初始状态。在接收到 CPU 的有效信号后，将状态转移到 COMPARE_TAG 状态。

### 1.2.2 COMPARE_TAG

这个状态首先根据 Tag 判断是否命中。如果命中，那么判断读写类型，如果是读的话就读入相应的数据，如果是写的话写入并且更新 dirty_bits 和 lru_bits；如果没有命中，那么根据 dirty 位判断是否需要写回。如果需要写回，那么将状态转移到 WRITE_BACK 状态。如果不需要写回，那么将状态转移到 ALLOCATE 状态。

### 1.2.3 ALLOCATE

在这个状态下，cache 等待 memory 的数据。如果 memory 的数据准备好了，那么我们需要从内存中将 addr 对应的块搬运到 cache 中来，并且更新 dirty_bits 和 lru_bits。因此这里需要输出相应的 mem_addr 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。如果 memory 的数据没有准备好，那么 cache 将继续等待。

### 1.2.4 WRITE_BACK

在这个状态下，cache 等待 memory 的数据。如果 memory 的准备好了，那么将数据写入 memory 中，并且清空 dirty_bits，这里同样要写入 mem_addr 然后返回 ALLOCATE。如果 memory 的数据没有准备好，那么 cache 将继续等待。

# 2 仿真测试

我们使用下面的 testbench 进行仿真测试

```
`include "cache.v"

module cache_tb;
    reg clk;
    reg rst;
    reg [31:0] addr_cpu;
    reg [31:0] data_cpu_write;
    reg [127:0] data_mem_read;
    reg [1:0] memRW;
    reg ready_mem;
    wire memRW_out;
    wire [31:0] data_cpu_read;
    wire [127:0] data_mem_write;
    wire [31:0] addr_mem;
    wire ready;
    wire [1:0] state;

    initial begin
        $dumpfile("cache_tb.vcd");
        $dumpvars(0, cache_tb);

        clk = 1;
        rst = 1;
        memRW = 0;
        #10 rst = 0;
        ready_mem = 1;
        // Read miss
        addr_cpu = 32'h10000000;
        memRW = 1;
```

```
30          data_mem_read = 128'h11111111222222223333333344444444;
31          #40;
32          // Read miss
33          addr_cpu = 32'h20000000;
34          data_mem_read = 128'h55555555666666667777777788888888;
35          #40;
36          // Read hit
37          addr_cpu = 32'h10000002;#20;
38          addr_cpu = 32'h20000001;#20;
39          // Write Hit
40          memRW = 2;
41          addr_cpu = 32'h10000001;
42          data_cpu_write = 32'hAAAAAAAA;
43          #20;
44          addr_cpu = 32'h20000003;
45          data_cpu_write = 32'hFFFFFFFF;
46          #20;
47          // Read hit
48          memRW = 1;
49          addr_cpu = 32'h10000001;
50          #20;
51          addr_cpu = 32'h20000003;
52          #20;
53          // Write miss, write back and allocate
54          memRW = 2;
55          addr_cpu = 32'h30000000;
56          data_cpu_write = 32'hAAAAAAAA;
57          data_mem_read = 128'hBBBBBBBBCCCCCCCCDDDDDDDDEEEEEEEE;
58          #50;
59          memRW = 1;
60          addr_cpu = 32'h30000000;#20;
61          addr_cpu = 32'h30000001;#20;
62      end
63      always begin
64          #5 clk = ~clk;
65      end
66
67      cache cache_inst(
68          .clk(clk),
69          .rst(rst),
70          .addr_cpu(addr_cpu),
71          .data_cpu_write(data_cpu_write),
72          .data_mem_read(data_mem_read),
73          .memRW(memRW),
74          .ready_mem(ready_mem),
75          .memRW_out(memRW_out),
76          .data_cpu_read(data_cpu_read),
77          .data_mem_write(data_mem_write),
78          .addr_mem(addr_mem),
79          .ready(ready),
80          .state(state)
81      );
82  endmodule
```

我们包含了读 miss，写 miss，读 hit，写 hit，写 miss 的情况。我们可以看到仿真结果如下：

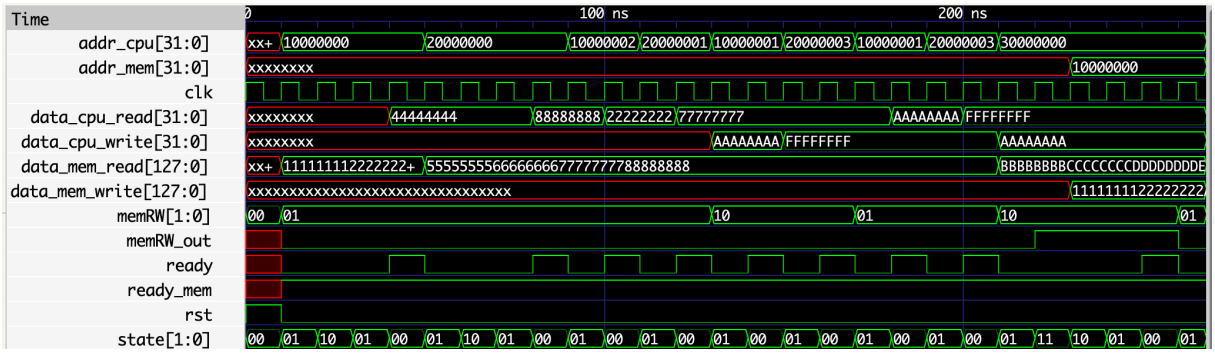


图 3：仿真结果 1

首先进行 reset，随后我们将 memRW 置 1，这表示我们要进行读操作。接着我们将地址 10000000 送入，这表示我们要读的地址，对应第 0 组的块，tag 是 100000。由于此时 cache 中还没有任何的数据，所以是 read miss。于是进入 allocate 阶段，我们将从 memory 中读到的数据写入 cache，然后再回到 compare tag 阶段，将读到的数据输出。

接着读地址 20000000 的操作是相同的，那么现在有两个块中已经写入数据。然后我们验证前面的写操作是否有效，我们尝试读入这个块中的其他的字，读地址 10000002，20000001，这两个满足 read hit，可以看到我们读出的数据也是符合预期的。10000002 对应的是第 0 组的 tag 为 100000 的第二个字，因此读出了 22222222；20000001 对应的是第 1 组的 tag 为 200000 的第一个字，因此读出了 77777777。

随后我们测试 write 的功能，将 memRW 置为 2 表示写操作；我们将从 CPU 中得到的要写入的数据分别写入地址 10000001 和 20000003，这两个地址对应的是第 0 组的 tag 为 100000 的第一个字和第 1 组的 tag 为 200000 的第三个字。可以看到我们写入的数据也是符合预期的。然后我们再次读取这两个地址，可以看到我们读出的数据也是是与传入的数据一致的。

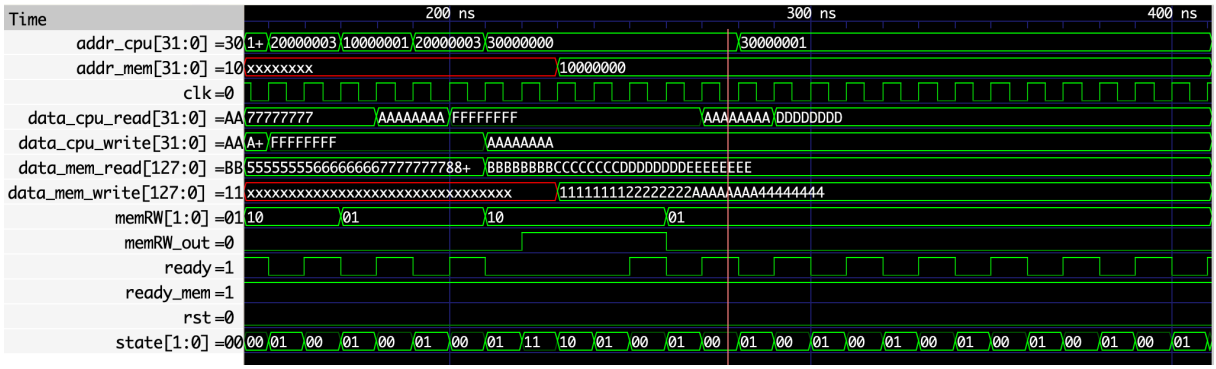


图 4：仿真结果 2

最后我们测试 LRU 替换策略，我们进行 30000000 地址的写，因为按照之前的操作，Index=0 的组已经被填满，我们需要根据 LRU 策略选择一个块替换出去。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。因此可以在第三个周期看到 memRW_out 被置为 1，表示要写回内存。在第五个周期看到 ready=1，说明已经完成了修改。然后我们再次读取 30000000 和 30000001，可以看到我们读出的数据也是符合预期的。AAAAAAAA 刚刚写入的数据，符合 LRU 策略。而 DDDDDDDD 是从磁盘中搬入的数据符合预期。

# 3 思考题

## 3.1 思考题 1

指令缓存在大多数情况下都是只读的，通常不需要考虑写回和写分配的问题。如果要修改指令也可以通过写操作来实现，但是要先将 cpu 暂停直到写操作完成为止。

## 3.2 思考题 2

带缓存的流水线发生缺失时应当阻塞流水线的运行，直到恢复正常为止。

# 4 心得体会

这下是真的完结了。本来没有打算写 cache 的，听牢 Q 说 cache 比较简单而且平时分确实有点欠缺于是就来写 cache 了。实际上也确实就是一个状态机的设计，如果按照 ppt 上的实现还要再另外创建模块，反倒不如直接用寄存器堆省事（x）。现在终于可以完结撒花开香槟了！（lhf：不，明天还有小测）计组的实验到这里完全结束，从开始基本模块的实现到单周期的实现，解决异常中断的实现，再到流水线不解决冲突和解决冲突，以及最后内存部分 cache 的实现，全部圆满完成，感恩助教，感恩老师，感恩马上要被卸载掉的 vivado。

祝愿考试顺利，江湖路远，有缘再会！