

# 浙江大学

## 本科实验报告

课程名称:	计算机体系结构
姓 名:	姜雨童
学 院:	计算机科学与技术学院
专 业:	计算机科学与技术
邮 箱:	3220103450@zju.edu.cn
QQ 号:	1369218489
电 话:	18867766468
指导教师:	王小航
报告日期:	2024 年 11 月 14 日

# 浙江大学实验报告

课程名称： 计算机体系结构 实验类型： 综合

实验项目名称： Lab03 Cache Design

学生姓名： 姜雨童 学号： 33220103450 同组学生姓名：       /      

实验地点： 玉泉曹西 301 实验日期： 2024 年 11 月 14 日

## 一、目标与原理、

### 1-1 实验目的

- Design of Cache Line
- Verify the Cache Line
- Observe the Waveform of Simulation

### 1-2 实验要求

本次实验需要实现一个 2 路组关联的 Cache，具体参数如下：

- 64 cache lines
  - Cache Line Size = 16 Bytes = 4 words = 4 \* 32 bits
- 2-way set associative
- Replacement policy: LRU
- Write policy
  - Write Back
  - Write Allocate

## 二、操作方法与实验步骤

### 2-1 表示内部信号

为了后续方便表示和书写，首先给出 tag、index 等信号的地址段，同样地，给出 dirty、valid 等信息。因为实验中的 Cache 为两路组关联的，因此这里的信号分为两路，可以根据给出的一路补全另一路。

```

    assign addr_tag = addr[ADDR_BITS-1:ADDR_BITS-TAG_BITS];           //need to fill in
    assign addr_index = addr[ADDR_BITS-TAG_BITS-
1:ELEMENT_WORDS_WIDTH+WORD_BYTES_WIDTH];           //need to fill in
    assign addr_element1 = {addr_index, 1'b0};
    assign addr_element2 = {addr_index, 1'b1};           //need to fill in
    assign addr_word1 = {addr_element1, addr[ELEMENT_WORDS_WIDTH+WORD_BYTES_WIDTH-
1:WORD_BYTES_WIDTH]};
    assign addr_word2 = {addr_element2, addr[ELEMENT_WORDS_WIDTH+WORD_BYTES_WIDTH-
1:WORD_BYTES_WIDTH]};           //need to fill in

    assign word1 = inner_data[addr_word1];
    assign word2 = inner_data[addr_word2];           //need to fill in
    assign half_word1 = addr[1] ? word1[31:16] : word1[15:0];
    assign half_word2 = addr[1] ? word2[31:16] : word2[15:0];           //need to fill in
    assign byte1 = addr[1] ?
        addr[0] ? word1[31:24] : word1[23:16] :
        addr[0] ? word1[15:8] : word1[7:0] ;
    assign byte2 = addr[1] ?
        addr[0] ? word2[31:24] : word2[23:16] :
        addr[0] ? word2[15:8] : word2[7:0] ;           //need to fill in

    assign recent1 = inner_recent[addr_element1];
    assign recent2 = inner_recent[addr_element2];           //need to fill in
    assign valid1 = inner_valid[addr_element1];
    assign valid2 = inner_valid[addr_element2];           //need to fill in
    assign dirty1 = inner_dirty[addr_element1];
    assign dirty2 = inner_dirty[addr_element2];           //need to fill in
    assign tag1 = inner_tag[addr_element1];
    assign tag2 = inner_tag[addr_element2];           //need to fill in

    assign hit1 = valid1 & (tag1 == addr_tag);
    assign hit2 = valid2 & (tag2 == addr_tag);           //need to fill in

```

## 2-2 补充输出信号

由于 Cache 控制是实验四的内容，因此查看实验四相关指导后可以知道这里输出信号的作用是给出将被替换 block 的信息。实验中采用 LRU（Least-Recent-Used）替换策略，因此使用 block1 时需要替换 block2，由此给出本模块输出。

```

always @ (posedge clk) begin
    valid <= recent1 ? valid2 : valid1;           //need to fill in
    dirty <= recent1 ? dirty2 : dirty1;           //need to fill in
    tag <= recent1 ? tag2 : tag1;           //need to fill in
    hit <= hit1 | hit2;           //need to fill in

```

## 2-3Cache 功能实现

这里分为 load、edit 和 store 情况：load 对应 load 指令，即从 cache 中读取数据；edit 对应 store 指令，即保存数据时对 cache 中的内容进行修改；store 则对应把数据从 memory 中写道 cache 的过程。

1. 对于 load 情况，如果 hit 了，输出相应数据并修改最近访问信息 inner\_recent（如果 miss 了，替换 recent 为 0 的数据，且数据为 dirty 时还需将其写回内存，实验中已经给出代码，不需要考虑，而具体的 dirty/recent 信息在 2-2 部分输出）；
2. 对于 edit 情况，修改数据并将标记位 dirty 置一，最后修改最近访问信息；
3. 对于 store 情况，从内存中读入数据并修改相应的标记。

```
// load
else if (hit2) begin
    dout <=
        u_b_h_w[1] ? word2 :
        u_b_h_w[0] ? {u_b_h_w[2] ? 16'b0 : {16{half_word2[15]}}, half_word2} :
        {u_b_h_w[2] ? 24'b0 : {24{byte2[7]}}, byte2};

    inner_recent[addr_element1] <= 1'b0;
    inner_recent[addr_element2] <= 1'b1;    //need to fill in
end
```

```
// edit
else if (hit2) begin
    inner_data[addr_word2] <=
        u_b_h_w[1] ?          // word
        din
        :
        u_b_h_w[0] ?          // half word
        addr[1] ?              // upper / lower?
        {din[15:0], word2[15:0]}
        :
        {word2[31:16], din[15:0]}
        : // byte
        addr[1] ?
        addr[0] ?
        {din[7:0], word2[23:0]} // 11
        :
        {word2[31:24], din[7:0], word2[15:0]} // 10
        :
        addr[0] ?
        {word2[31:16], din[7:0], word2[7:0]} // 01
        :
        {word2[31:8], din[7:0]} // 00
        ;
    inner_dirty[addr_element2] <= 1'b1;
```

```

inner_recent[addr_element1] <= 1'b0;
inner_recent[addr_element2] <= 1'b1;    //need to fill in
end

```

```

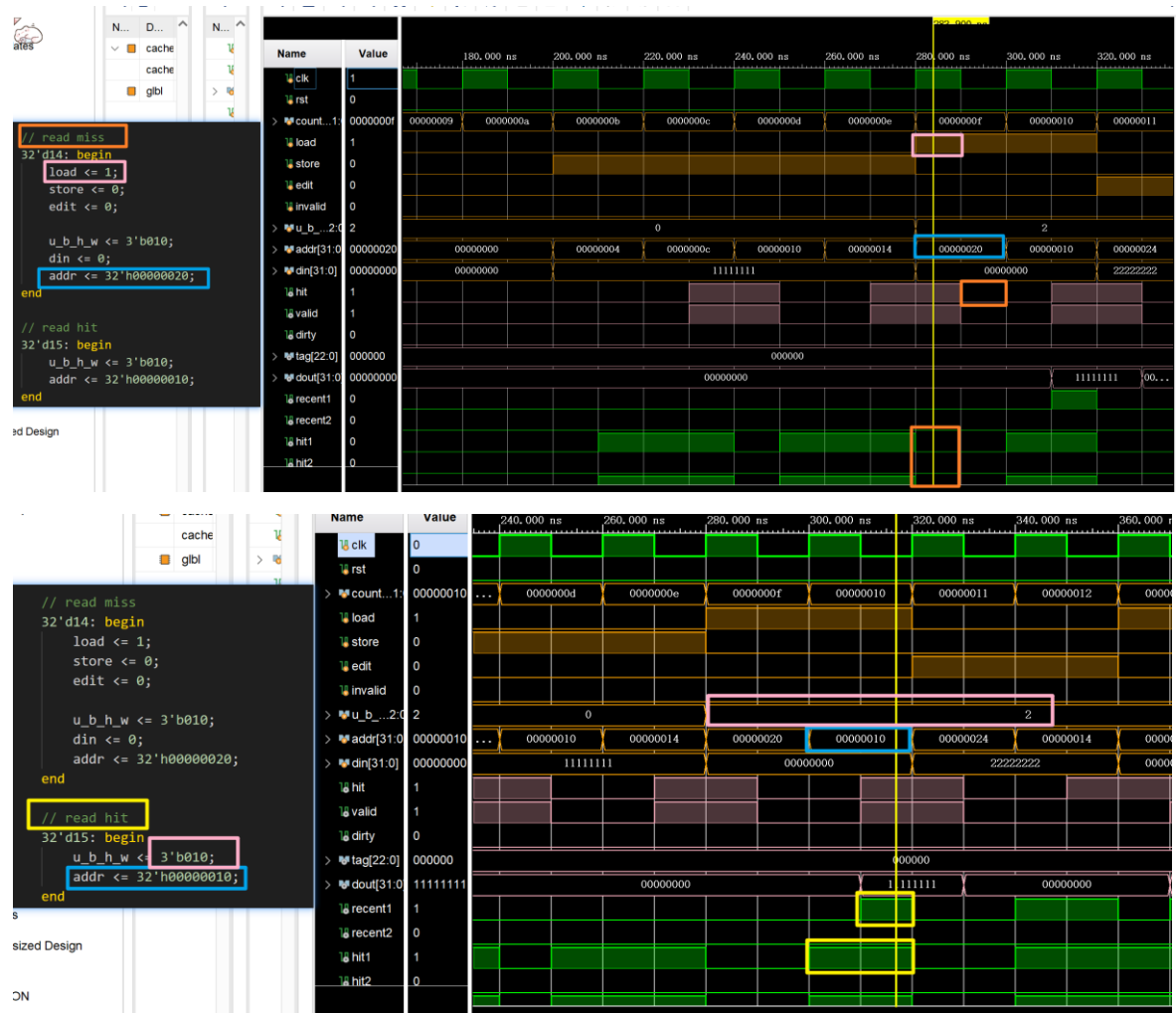
// store
    // recent2 == 1 => replace 1
    // recent2 == 0 => no data in this set, place to 1
    inner_data[addr_word1] <= din;
    inner_valid[addr_element1] <= 1'b1;
    inner_dirty[addr_element1] <= 1'b0;
    inner_tag[addr_element1] <= addr_tag;    //need to fill in
end

```

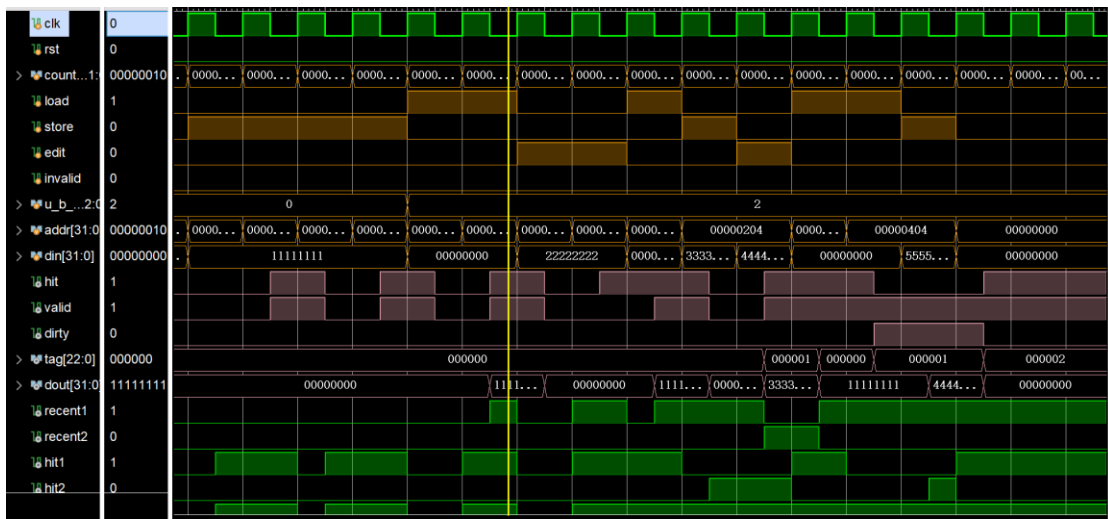
### 三、实验结果与分析

本次实验只需要仿真，不需要上板验证。对照 testbench 和仿真波形后发现结果符合预期，这里分析一次 read miss 后 read hit 的过程，其他部分不再赘述。

图一为 read miss，黄线标记处为 load 情况，且 addr 为 0x20；图二为 read hit。





完整仿真波形如下：







## 四、讨论、心得





整体实验并不难，大部分代码也给出了参考样例，可以根据 xxx1 写出 xxx2，因此整体实验过程并没有遇到特别的困难或 bug。

唯一需要注意的是，最开始仿真时出现了报错信息，检查后发现是因为给的框架把仿真测试文件(cache\_sim.v)设成了 top（文件前有标识 ），更改后就可以正常跑仿真了。

 Synthesis (3 errors)

-  [Synth 8-462] no clock signal specified in event control [cache\_sim.v:70]
-  [Synth 8-6156] failed synthesizing module 'cache\_sim' [cache\_sim.v:25]
-  [Common 17-69] Command failed: Vivado Synthesis failed

 Design Sources (2)

-  Verilog Header (1)
  -  addr\_define.vh
-  **cache\_sim** (cache\_sim.v) (1)
  -  uut : cache (cache.v)