浙江大學
ZHEJIANG UNIVERSITY

Lab 3 Cache Design

2024-2025 春夏学期 计算机体系结构

课程实验报告

姓名 王浩雄

学号 3230106032

年级 2023 级

专业 混合班（计算机科学与技术）

班级 混合 2303 班

2025 年 3 月 28 日

Lab 3 Report

1 实验目的

本实验要求设计一个满足如下要求的 Cache：

- 2 路组相连，具有 64 个 Cache Line，每个 Cache Line 的大小为 16 字节；
- 使用 LRU 替换策略；
- 使用写返回、写分配策略。

2 实验设计

2.1 输入输出接口

代码片段：

```
module cache (
        input wire clk,                 // 时钟信号
        input wire rst,                 // 复位信号
        input wire [ADDR_BITS-1:0] addr,  // 地址输入
        input wire load,                // 指定操作类型：读操作
        input wire store,               // 指定操作类型：存储操作
        input wire edit,                // 指定操作类型：写操作（编辑）
        input wire invalid,             // 指定操作类型：失效操作
        input wire [2:0] u_b_h_w,       // 指定数据宽度与符号扩展控制
        input wire [31:0] din,          // 要写入的数据
        output reg hit = 0,             // 命中信号
        output reg [31:0] dout = 0,     // 读出的数据
        output reg valid = 0,           // 有效位
        output reg dirty = 0,           // 脏位
        output reg [TAG_BITS-1:0] tag = 0  // tag信息
);
```

解释：

该部分定义了模块的输入输出接口：

- `clk` 和 `rst` 分别用于驱动同步逻辑与系统复位；
- `addr` 为存取数据时提供的地址，后续将进行地址解码；

- load、store、edit 控制缓存的读、替换、写操作；
- u_b_h_w 决定数据读取/写入的宽度和是否有符号扩展；
- din 为写入数据，dout 为读出数据；
- hit、valid、dirty、tag 分别反馈命中状态、有效性、脏数据标识及标签信息。

具体地，当我们考虑 Cache 模块与 Cache 控制器进行交互时，输出命中状态、有效性、脏数据标识及标签信息具有以下作用：

- hit: 反馈命中状态，使缓存控制器能够判断当前访问请求的数据是否已存在于缓存中。如果 hit 为高，控制器便不需要进一步从主存取数据；否则，必须进行缺失处理。
- valid: 反馈有效位，确保缓存控制器判断缓存块中的数据是否为有效数据。即使地址匹配（tag 比较成功），如果 valid 为低，数据也被认为是无效的。
- dirty: 反馈脏数据标识，用于写回策略。当缓存替换时，若 dirty 为高，控制器就知道需要将数据写回主存，以防止数据丢失。
- tag: 反馈 tag 信息。结合地址解码后获得的 tag，控制器能够验证命中情况并保持缓存一致性，支持对缓存行的管理。

2.2 存储结构设计

缓存内部采用二维数组来保存数据与状态信息。每个缓存块内含有 tag、数据、有效位、脏位及最近使用位。

代码片段：

```
reg [ELEMENT_NUM-1:0] inner_recent = 0;
reg [ELEMENT_NUM-1:0] inner_valid = 0;
reg [ELEMENT_NUM-1:0] inner_dirty = 0;
reg [TAG_BITS-1:0] inner_tag [0:ELEMENT_NUM-1];
reg [31:0] inner_data [0:ELEMENT_NUM*ELEMENT_WORDS-1];

integer i;
initial begin
for (i = 0; i < ELEMENT_NUM; i = i + 1)
inner_tag[i] = 23'b0;
for (i = 0; i < ELEMENT_NUM*ELEMENT_WORDS; i = i + 1)
inner_data[i] = 32'b0;
end
```

解释：

- `inner_recent` 用于记录每个缓存块的最近使用状态，采用 LRU 替换策略。
- `inner_valid` 和 `inner_dirty` 分别存储缓存块是否有效以及是否被修改的状态。
- `inner_tag` 数组存放每个缓存块的 tag 信息，用于缓存命中判断。
- `inner_data` 存储实际数据，每个缓存块包含多个字。
- `initial` 块用于上电初始化，通过遍历将 tag 和数据均置零。

2.3 地址解码

地址解码部分将输入地址分解为 tag、索引和字内偏移，用于定位具体缓存块和数据字。

代码片段：

```
// 地址字段分解
assign addr_tag = addr[ADDR_BITS-1:ADDR_BITS-TAG_BITS];
assign addr_index = addr[SET_INDEX_WIDTH+ELEMENT_WORDS_WIDTH+
   WORD_BYTES_WIDTH-1:
ELEMENT_WORDS_WIDTH+WORD_BYTES_WIDTH];
assign addr_element1 = {addr_index, 1'b0};
assign addr_element2 = {addr_index, 1'b1};
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]};
```

解释：

- `addr_tag` 提取地址高位，作为缓存块的标签。
- `addr_index` 缓存组的索引，决定数据在二维数组中的组位置。
- `addr_element1` 和 `addr_element2` 表示同一组内两个路（way）的索引。
- `addr_word1` 和 `addr_word2` 进一步结合组内索引和地址内的字选择位，定位到具体数据存储单元。

具体地，以下两行代码实现了对同一组内两个缓存块的定位。地址中的 `addr_index` 部分用于确定缓存的组号。将 `addr_index` 与一个额外的位（1'b0 或 1'b1）拼接，得到 `addr_element1` 和 `addr_element2`，分别代表同一组内的第一个和第二个缓存块。

```
assign addr_element1 = {addr_index, 1'b0};
assign addr_element2 = {addr_index, 1'b1};
```

2.4 数据读取与控制信号的设定

代码片段：

```
assign word1 = inner_data[addr_word1];
assign word2 = inner_data[addr_word2];
assign half_word1 = addr[1] ? word1[31:16] : word1[15:0];
assign half_word2 = addr[1] ? word2[31:16] : word2[15:0];
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]);

assign recent1 = inner_recent[addr_element1];
assign recent2 = inner_recent[addr_element2];
assign valid1 = inner_valid[addr_element1];
assign valid2 = inner_valid[addr_element2];
assign dirty1 = inner_dirty[addr_element1];
assign dirty2 = inner_dirty[addr_element2];
assign tag1 = inner_tag[addr_element1];
assign tag2 = inner_tag[addr_element2];

assign hit1 = valid1 & (tag1 == addr_tag);
assign hit2 = valid2 & (tag2 == addr_tag);
```

解释：

- **数据选择：**根据 `addr_word1` 和 `addr_word2` 直接索引 `inner_data` 数组，获取对应字；再通过地址中较低位确定半字和字节的选取。
- **控制信号提取：**通过 `inner_recent`、`inner_valid`、`inner_dirty` 和 `inner_tag` 数组分别获得两路的状态信号。
- **命中判断：**利用比较得到的 tag 与 `addr_tag` 及有效位，生成 `hit1` 和 `hit2` 信号，表明对应路是否命中。

2.5 时钟上升沿的同步逻辑

所有缓存的状态更新和数据传输均在时钟上升沿完成，根据不同的操作信号进行读、写、替换以及失效操作。

代码片段：

```
always @ (posedge clk) begin

valid <= recent1 ? valid2 : valid1;
dirty <= recent1 ? dirty2 : dirty1;
tag   <= recent1 ? tag2   : tag1;
hit   <= hit1 | hit2;

// 读操作
if (load) begin
        if (hit1) begin
                dout <= u_b_h_w[1] ? word1 :
                u_b_h_w[0] ? {u_b_h_w[2] ? 16'b0 : {16{half_word1
                   [15]}}, half_word1} :
                {u_b_h_w[2] ? 24'b0 : {24{byte1[7]}}, byte1};
                inner_recent[addr_element1] <= 1'b1;
                inner_recent[addr_element2] <= 1'b0;
        end
        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;
        end
end
else begin
        dout <= inner_data[ recent1 ? addr_word2 : addr_word1 ];
end

// 写操作
if (edit) begin
        if (hit1) begin
                inner_data[addr_word1] <= /* 根据u_b_h_w选择不同写入
                   方式 */;
```

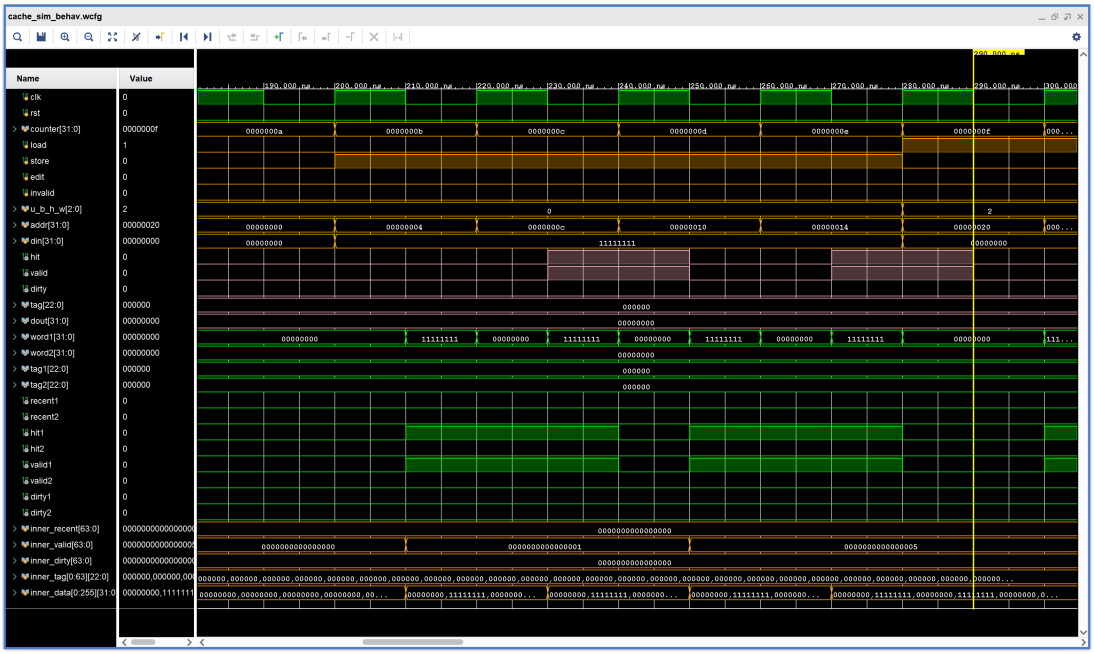
```
33                  inner_dirty[addr_element1] <= 1'b1;
34                  inner_recent[addr_element1] <= 1'b1;
35                  inner_recent[addr_element2] <= 1'b0;
36          end
37          else if (hit2) begin
38                  inner_data[addr_word2] <= /* 根据u_b_h_w选择不同写入
                        方式 */;
39                  inner_dirty[addr_element2] <= 1'b1;
40                  inner_recent[addr_element1] <= 1'b0;
41                  inner_recent[addr_element2] <= 1'b1;
42          end
43  end
44
45  // 存储操作
46  if (store) begin
47          if (recent1) begin
48                  inner_data[addr_word2] <= din;
49                  inner_valid[addr_element2] <= 1'b1;
50                  inner_dirty[addr_element2] <= 1'b0;
51                  inner_tag[addr_element2] <= addr_tag;
52          end
53          else begin
54                  inner_data[addr_word1] <= din;
55                  inner_valid[addr_element1] <= 1'b1;
56                  inner_dirty[addr_element1] <= 1'b0;
57                  inner_tag[addr_element1] <= addr_tag;
58          end
59  end
60
61  // 失效操作
62  if (invalid) begin
63          inner_recent[addr_element1] <= 1'b0;
64          inner_recent[addr_element2] <= 1'b0;
65          inner_valid[addr_element1] <= 1'b0;
66          inner_valid[addr_element2] <= 1'b0;
67          inner_dirty[addr_element1] <= 1'b0;
68          inner_dirty[addr_element2] <= 1'b0;
69  end
70  end
```

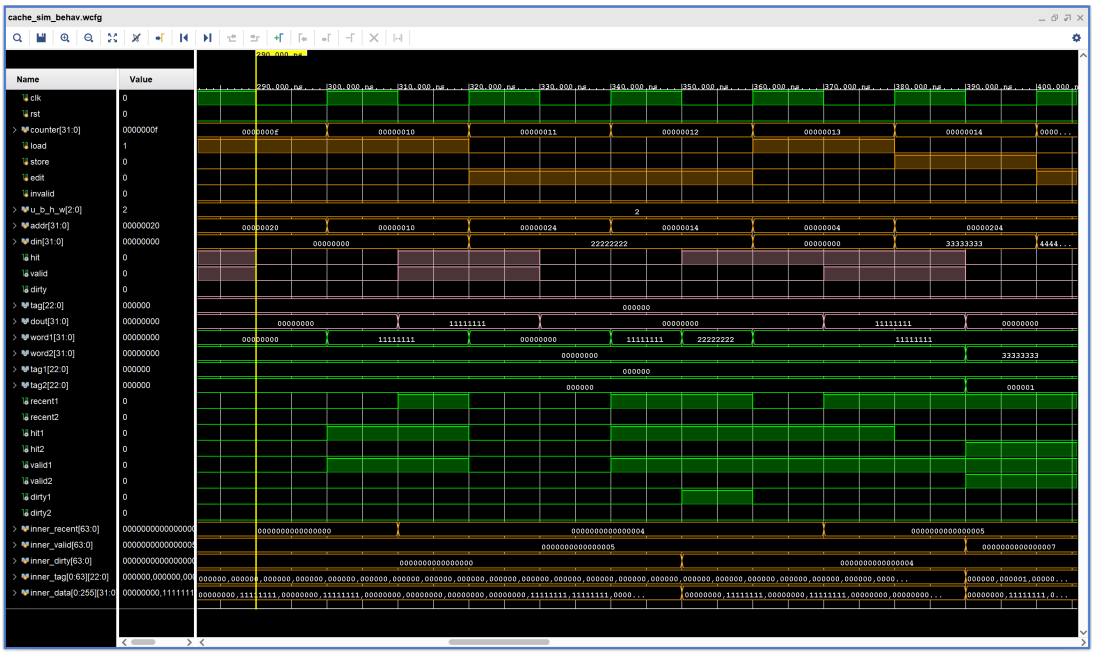
解释：

- **输出选择**：根据最近使用位，在两路之间选取当前不活跃的缓存块，并输出其 `valid`、`dirty` 与 `tag` 状态；命中信号为两个路命中结果的或运算。
- **读操作**：当 `load` 信号为高时，判断命中路（hit1 或 hit2），根据 `u_b_h_w` 选择字、半字或字节数据输出，并更新最近使用位。
- **写操作**：当 `edit` 信号为高时，根据命中路更新相应数据，同时设置对应路的 `dirty` 位，并刷新最近使用位。
- **存储替换**：当 `store` 信号为高时，根据最近使用位决定替换哪一路的数据，写入新数据的同时更新 `valid`、`dirty` 和 `tag`。
- **失效操作**：当 `invalid` 信号为高时，将当前组中两路的 `recent`、`valid` 和 `dirty` 均清零，实现缓存块失效。

3 实验结果

3.1 仿真波形





3.2 仿真测试点分析

通过仿真测试，我验证了缓存模块在不同操作下对数据、状态位及 tag 信息的更新逻辑，确保了缓存命中判断、写回和替换策略的正确实现。下面选取部分关键测试点进行详细说明：

3.2.1 读操作：读命中与读缺失测试

- 读缺失

```
32'd14: begin
load <= 1;
store <= 0;
edit <= 0;
u_b_h_w <= 3'b010;
din <= 0;
addr <= 32'h00000020;
end
```

该测试点发起一次读操作，地址为 0x00000020。由于此前未在缓存中写入该地址对应的数据，故出现读缺失。本例中观察到输出信号 hit 为低，与预期相符。

- **读命中**

```
32'd15: begin
u_b_h_w <= 3'b010;
addr <= 32'h00000010;
end
```

该时刻读操作的地址为 0x00000010，在 32'd12 时已经存入数据，因此应产生缓存命中。同时，根据 u_b_h_w 信号的设置，缓存模块将以指定数据宽度输出数据，并更新对应缓存行的最近使用位（recent）。本例中观察到输出信号 hit 、valid 为高，dout 为 0x11111111，与预期相符。

3.2.2 写操作：写命中与写缺失测试

- **写缺失**

```
32'd16: begin
load <= 0;
store <= 0;
edit <= 1;
u_b_h_w <= 3'b010;
din <= 32'h22222222;
addr <= 32'h000000024;
end
```

此时执行写操作（edit），地址 0x00000024 在缓存中未命中，故发生写缺失。在缺失时，缓存需要进行替换或先行加载后修改。本例中观察到输出信号 hit 、valid 为低，与预期相符。

- **写命中**

```
32'd17: begin
u_b_h_w <= 3'b010;
addr <= 32'h00000014;
end
```

该测试点继续执行写操作（edit），地址 0x00000014 之前已经写入数据（在 32'd13 时），因此会产生写命中。此时，缓存模块会根据写操作对数据进行更新，同时修改该缓存行的 dirty 位置，表示数据已经修改，需在替换时写回主存。本例中观察到信号 hit1 、dirty1 为高，word1 由 0x11111111 变为 0x22222222，与预期相符。

3.2.3 recent 位的影响与替换测试

- **读操作设置 recent 位**

```
32'd18: begin
load <= 1;
store <= 0;
edit <= 0;
u_b_h_w <= 3'b010;
din <= 0;
addr <= 32'h00000004;
end
```

读取地址 0x00000004，由缓存数据知，该地址对应缓存组中的第一路。读操作成功后，缓存模块将把对应行的 recent1 位更新为 1，表明第一路为最近使用的，从而在替换策略中起到保护作用。本例中观察到信号 recent1 为高，与预期相符。

- **替换操作——存储到非最近使用行**

```
32'd19: begin
load <= 0;
store <= 1;
edit <= 0;
u_b_h_w <= 3'b010;
din <= 32'h33333333;
addr <= 32'h00000204;
```

```
end
```

此时存储操作针对地址 0x00000204,在本例的缓存实现中,该地址与地址 0x00000004 被映射到同一行（即具有相同的 index）。由于该行 recent1 为高，缓存模块会将第二路作为替换目标。更新时会同时设置 valid 位（置 1）、清除 dirty 位，并写入新的 tag。本例中观察到信号 hit2 、valid2 为高，与预期相符。

- **写操作使替换行变脏**

```
32'd20: begin
load <= 0;
store <= 0;
edit <= 1;
u_b_h_w <= 3'b010;
din <= 32'h44444444;
addr <= 32'h00000204;
end
```

对同一地址 0x00000204 进行写操作，使刚替换进入的数据被修改。此时，该缓存行的 dirty 位被置高，同时 recent 位更新，标识该行为最新使用且数据已修改，后续替换时需写回主存。本例中观察到信号 recent2 、dirty2 为高，与预期相符。同时，dout 输出 0x33333333，该数值是该行该路原来存储的数值，将其作为 dout 输出给缓存控制器，此后缓存控制器操作该数据写入内存。