浙江大学

本科实验报告

课程名称：计算机体系结构

姓　　名：夏尤楷

学　　院：计算机科学与技术学院

系：计算机科学与技术

专　　业：计算机科学与技术

学　　号：3210104331

指导教师：陈文智

2023 年 11 月 20 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：缓存设计

学生姓名：夏尤楷 专业：计算机科学与技术 学号：3210104331

同组学生姓名：来思锐 指导老师：陈文智

实验地点：曹西 301 实验日期：2023 年 11 月 7 日

一、 实验目的和要求

目的：

1. 理解缓存；
2. 理解缓存管理单元及其状态机的原理；
3. 掌握缓存管理单元的设计方法；
4. 掌握缓存行（cache line）的设计方法；
5. 掌握缓存行的验证方法。

任务：

1. 设计缓存行和缓存管理单元；
2. 验证缓存行和缓存管理单元；
3. 观察仿真波形图。

二、 实验内容和原理

缓存行由 1 位的 LRU（least recently used，意为最近最少使用。用来标记相同索引（index）下数据最近被使用的次数的多少，1 表示更多地被使用，0 表示更少地被使用。LRU 是一种替换策略，意思是当某一索引的空间被填满后，后进入的该索引的数据会占据最近最少使用的数据的空间，将之替换掉）、1 位的 V（valid，用来标记这个缓存行是否正在被用来存储数据，1 表示是，0 表示否）、1 位的 D（dirty，用来标记该数据是否是“脏的”，即该数据是否与内存中对应地址的数据相一致，1 表示是。如果不一致，该数据在移出缓存时就必须要写回内

存中的对应地址）、若干位的标签（tag，建立起缓存中的数据和内存的地址之间的映射关系）和 128 位的数据（data）空间构成。

本实验中，把内存中的数据映射到缓存中时，使用 2 路组关联（2-way set associative），即同一索引对应有 2 个数据空间，当内存中某个地址（由地址可得索引值）的数据要映射到缓存中时，在 2 个数据空间中。缓存共有 32 组空间（即 64 个数据空间，每组 2 个），一个数据空间由 4 个字（word）构成，每个字占 4 字节。

内存地址由 32 位构成。在将地址上的数据映射到缓存中时，地址的各部分的作用如下。

地址（32 位）		
31 9	8 4	3 2	1 0
标签（23 位）	索引（5 位）	字偏移（2 位）	字节偏移（2 位）

本次实验只需编辑 cache.v 这一个源文件，内含 cache 模块，输入输出端口如下：

```
module cache (
    input wire clk,  // clock
    input wire rst,  // reset
    input wire [ADDR_BITS-1:0] addr,  // address
    input wire load,   //  read refreshes recent bit
    input wire store,  // set valid to 1 and reset dirty to 0
    input wire edit,  // set dirty to 1
    input wire invalid,  // reset valid to 0
    input wire [2:0] u_b_h_w, // select signed or not & data width
                        // please refer to definition of LB, LH, LW,
LBU, LHU in RV32I Instruction Set
    input wire [31:0] din,  // data write in
    output reg hit = 0,  // hit or not
    output reg [31:0] dout = 0,  // data read out
    output reg valid = 0,  // valid bit
    output reg dirty = 0,  // dirty bit
    output reg [TAG_BITS-1:0] tag = 0  // tag bits
    );
```

三、 实验过程和数据记录

1. 部分变量的准备

cache.v 中定义了许多 wire 变量，每个变量有其作用及含义。部分变量的赋值未被补全。补全后变量赋值如下：

```
assign addr_tag = addr[ADDR_BITS-1:ADDR_BITS-TAG_BITS];  //need to fill in
assign addr_index = addr[ADDR_BITS - TAG_BITS-1:
                    ADDR_BITS-TAG_BITS-SET_INDEX_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[1:0] == 2'b11 ? word2[31:24] :
                      (addr[1:0] == 2'b10 ? word2[23:16] :
                      (addr[1:0] == 2'b01 ? word2[15:8] :
                      (addr[1:0] == 2'b00 ? word2[7:0] : word2[31:24])));
                                                     //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
```

其中，变量后面带 1 的为第 1 路的缓存行的相关变量，后面带 2 的为第 2 路的缓存行的相关变量。recent 即 LRU 位。

2. 从缓存中读取数据

当 load == 1 时，说明读取内存中某地址数据时，要访问缓存，尝试从缓存中读取相应数据，被读取的数据要被输出。需要根据输入模块的数据长度和类型的限定输出指定长度和类型（有无符号）的数据，高位空位按照类型进行补足。并更新两路中的 recent 位。

限定存储在变量 u_b_h_w[2:0]中。u_b_h_w[2]代表输出的数据是否为无符号数（1 代表是）。u_b_h_w[1:0]为 2’b11 或 2’b10 时，代表数据为字；u_b_h_w[1:0]为 2’b01 时，代表数据为半字；u_b_h_w[1:0]为 2’b00 时，代表数据为字节。

源代码已经给出第 1 路命中（hit）时读取出的数据的表达式。仿照第 1 路即可给出第 2 路命中时读取出的数据的表达式。代码如下：

```
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
```

3. 编辑缓存内数据

当 edit == 1 时，说明如果根据数据在内存中的地址，在缓存中命中了相应的数据，则需要对编辑缓存内的该数据。这样，缓存内的数据与内存中对应地址的数据就出现了不同，就要标记缓存内的数据为“脏”。还要更新两路中的 recent 位。

编辑的数据的长度储存在 u_b_h_w[1:0]中，其内数据代表的含义与 load 中相同。

源代码已经给出第 1 路经编辑后的数据的表达式。仿照第 1 路即可给出第 2 路经编辑后的数据的表达式。代码如下：

```
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_element2] <= 1'b1;
    inner_recent[addr_element1] <= 1'b0;
end
```

4. 往缓存中存入数据

当 store == 1 时，表明要往缓存中存入数据。当存入数据的索引对应的空间已满，要存入新数据就必须替换掉缓存内该索引中的原有的数据时，根据两路中 recent 的值的大小，保留大的那一路的数据，替换掉小的那一路的数据。同时，对应的 dirty 要置 0，valid 要置 1。

源代码中已经给出 recent1 == 1 时把数据存入第 2 路，替换掉原有数据的代码。仿照把存入第 2 路时的代码即可写出把数据存入第 1 路的代码，代码如下：

```
// recent2 == 1 => replace 1
// recent2 == 0 => no data in this set, place to 1
```

```
if (recent2) begin
     inner_data[addr_word1] <= din;
     inner_valid[addr_element1] <= 1'b1;
     inner_dirty[addr_element1] <= 1'b0;
     inner_tag[addr_element1] <= addr_tag;
end else begin
     inner_data[addr_word1] <= din;
     inner_valid[addr_element1] <= 1'b1;
     inner_dirty[addr_element1] <= 1'b0;
     inner_tag[addr_element1] <= addr_tag;
end
//need to fill in
```

5. 一些访问缓存结果的输出

模块有一些输出端口是用来输出访问缓存的信息的，这些信息包括是否命中（1 代表是），访问到的缓存空间是否有效（valid）、是否为脏、标签为多少等等，以供外部模块使用。

由于 LRU 策略的缘故，输出的访问到的缓存空间的信息为 recent 较小的信息，以供数据在被替换时模块外的写回内存等一系列操作。

代码如下：

```
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 <= hit2 | hit1;                 //need to fill in
```

6. 仿真代码的设计

仿真代码要涉及读命中、读未命中、写命中、写未命中四种情况。

仿真代码设计如下：

```
module cache_sim;

    // Inputs
    reg clk;
    reg rst;
    reg [31:0] addr;
    reg load;
    reg store;
    reg edit;
    reg invalid;
    reg [2:0] u_b_h_w;
    reg [31:0] din;

    // Outputs
    wire hit;
    wire [31:0] dout;
    wire valid;
    wire dirty;
    wire [22:0] tag;

    // Instantiate the Unit Under Test (UUT)
    cache uut (
        .clk(~clk),
        .rst(rst),
```

```
    .addr(addr),
    .load(load),
    .store(store),
    .edit(edit),
    .invalid(invalid),
    .u_b_h_w(u_b_h_w),
    .din(din),
    .hit(hit),
    .dout(dout),
    .valid(valid),
    .dirty(dirty),
    .tag(tag)
);

initial begin
    clk = 1;
    forever #10 clk = ~clk ;
end

reg [31:0]counter = 0;

always @(posedge clk) begin
    counter <= counter + 32'b1;

    case (counter)
        // Initialize Inputs
        32'd0: begin
            rst <= 0;
            addr <= 0;
            load <= 0;
            store <= 0;
            edit <= 0;
            invalid <= 0;
            u_b_h_w <= 0;
            din <= 0;
        end

        // init
        32'd10: begin
            load <= 0;
            store <= 1;
            edit <= 0;

            din <= 32'h11111111;
            addr <= 32'h00000004;
        end

        32'd11: begin
            addr <= 32'h0000000C;
        end

        32'd12: begin
            addr <= 32'h00000010;
        end

        32'd13: begin
            addr <= 32'h00000014;
        end

        // read hit
        32'd14: begin
```

```
    load <= 1;
    store <= 0;
    edit <= 0;

    u_b_h_w <= 3'b010;
    din <= 0;
    addr <= 32'h00000014;
end

// read miss
32'd15: begin
    u_b_h_w <= 3'b010;
    addr <= 32'h00000030;
end

// write hit
32'd16: begin
    load <= 0;
    store <= 0;
    edit <= 1;

    u_b_h_w <= 3'b010;
    din <= 32'h22222222;
    addr <= 32'h00000010;
end

// write miss
32'd17: begin
    u_b_h_w <= 3'b010;
    addr <= 32'h000000024;
end

// read line 0 of set 0, set recent bit
32'd18: begin
    load <= 1;
    store <= 0;
    edit <= 0;

    u_b_h_w <= 3'b010;
    din <= 0;
    addr <= 32'h00000014;
end

// store to line 1 of set 1 due to line 0 recent
32'd19: begin
    load <= 0;
    store <= 1;
    edit <= 0;

    u_b_h_w <= 3'b010;
    din <= 32'h33333333;
    addr <= 32'h00000214;
end

// edit line 1 of set 1, set dirty & recent
32'd20: begin
    load <= 0;
    store <= 0;
    edit <= 1;

    u_b_h_w <= 3'b010;
```

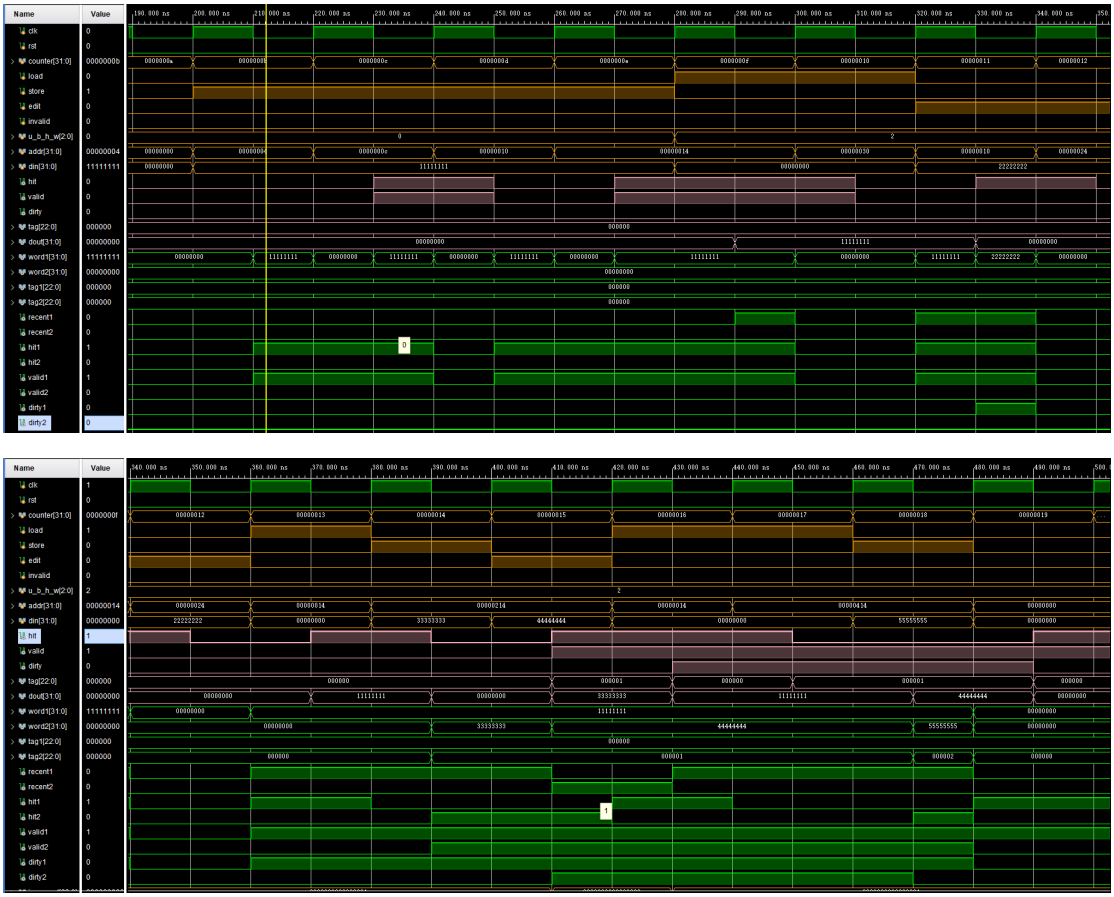
```
            din <= 32'h44444444;
            addr <= 32'h00000214;
        end

        // read line 0 of set 1, set recent bit
        32'd21: begin
            load <= 1;
            store <= 0;
            edit <= 0;

            u_b_h_w <= 3'b010;
            din <= 0;
            addr <= 32'h00000014;
        end

// read miss, tag mismatch. output tag (of line 1), valid and dirty == 1
        32'd22: begin
            load <= 1;
            store <= 0;
            edit <= 0;

            u_b_h_w <= 3'b010;
            din <= 32'h0;
            addr <= 32'h00000414;
        end

        // auto replace line 1 of set 0
        32'd23: begin
            load <= 0;
            store <= 1;
            edit <= 0;

            u_b_h_w <= 3'b010;
            din <= 32'h55555555;
            addr <= 32'h00000414;
        end

        // clear
        default: begin
            load <= 0;
            store <= 0;
            edit <= 0;
            din <= 0;
            addr <= 0;
        end
    endcase
end
```

四、 实验结果分析

仿真结果如下两图。图中上方黄色变量是计数器 counter 和输入模块的自变量，只在时钟正边沿处更新；中间粉红色变量是缓存内随时钟负边沿更新的数据（reg 变量），赋给它们的值由下方绿色变量决定，它们不在时钟正边沿处更新；下方绿色变量的值是由输入变量和模块内在时钟负边沿更新的数据（reg 变量）通过组合逻辑得到，故既可能在时钟正边沿变化，也可能在时钟负边沿变化。

对缓存内数据的编辑和存放数据进入缓存这两个操作都在时钟负边沿时完

成。



counter 在 32’h0000000b~32’h0000000e 之间时，store == 1，程序正在向缓存中存入初始的数据，每个周期内存入一个。

首先存入的是内存地址为 32’h0000004 的数据，数据为 32’h11111111，将数据及其标签存入 set 0 的索引为 0 处的空间中，标签为 23’h00000。存入前，该空间内是无效数据，valid1 = 0。时钟负边沿后存入数据，此时 valid1 自然更新为 1。store 操作不改变 recent1，故存入后 recent1 仍为 0，。存入后，word1 显示刚存入的数据，tag1 显示刚存入的数据的标签，由于 recent1 == 0，tag 取 tag1 = 23’h00000。易得标签和索引必然匹配，故 hit1 由 0 变为 1。易得数据存入后，存入缓存空间内的标签和索引自然与存入数据的地址是匹配的，所以 hit1 由 0 变为 1。

再存入的是内存地址为 32’h0000000c 的数据，数据为 32’h11111111，由于 recent1 仍为 0，该数据依然存入 set 0 的索引为 0 处的空间中。存入前，该空间内数据有效，valid1 == 1，由于 recent1 = 0，时钟负边沿后，valid 被更新 valid1 的值；存入前，地址 32’h0000000c 的标签与该空间的标签 tag1 相同，均为 23’h00000，故命中，所以 hit1 在存入前就显示为 1，时钟负边沿后 hit 也由 0 更

新为 1。同样，store 操作不改变 recent1，故 recent1 仍为 0。存入后，word1 显示刚存入的数据，tag1 显示刚存入的数据的标签，由于 recent1 == 0，tag 取 tag1 = 23'h00000。

再存入的是内存地址为 32'h00000010 的数据，数据为 32'h11111111，将数据及其标签存入 set 0 的索引为 1 处的空间中，标签为 23'h00000。存入前，该空间内是无效数据，valid1 = 0。时钟负边沿后存入数据，此时 valid1 自然更新为 1。store 操作不改变 recent1，故 recent1 仍为 0。存入后，word1 显示刚存入的数据，tag1 显示刚存入的数据的标签，由于 recent1 == 0，tag 取 tag1 = 23'h00000。易得数据存入后，存入缓存空间内的标签和索引自然与存入数据的地址是匹配的，所以 hit1 由 0 变为 1。

再存入的是内存地址为 32'h00000014 的数据，数据为 32'h11111111，由于 recent1 仍为 0，该数据依然存入 set 0 的索引为 1 处的空间中。存入前，该空间内数据有效，故 valid1 == 1，时钟负边沿后，由于 recent1 == 0，valid 被更新为 valid1 的值；存入前，地址 32'h00000014 的标签与该空间的标签 tag1 相同，均为 23'h00000，故命中，所以 hit1 在存入前就显示为 1，时钟负边沿后 hit 也由 0 更新为 1。同样，store 操作不改变 recent1，故 recent1 仍为 0。存入后，word1 显示刚存入的数据，tag1 显示刚存入的数据的标签，由于 recent1 == 0，tag 取 tag1 = 23'h00000。

counter == 32'h0000000f 时，load == 1，进行读取操作，读取内存地址为 32'h00000014 的数据。该地址对应的索引为 1，显然在时钟负边沿前，set 0 中索引 1 数据有效（valid1 == 1），且其标签 tag1 与该地址的标签同为 23'h00000，故读命中，hit1 为 1，时钟负边沿后 hit 与 valid 仍为 1。原本该数据的 recent 为 0，时钟负边沿后，更新该数据的 recent 为 1，故 recent1 变为 1。时钟负边沿后，dout 更新为被读取出的数据，即 word1 = 32'h11111111，输出到缓存外。

counter == 32'h00000010 时，load == 1，进行读取操作，读取内存地址为 32'h00000030 的数据。该地址对应的索引为 3，很明显在时钟负边沿前，缓存内该索引的空间中没有任何有效数据，故读未命中。时钟负边沿前，hit1 = hit2 = 0，valid1 = valid2 = 0，时钟负边沿后，valid 和 hit 也都更新为零。

counter == 32'h00000011 时，edit == 1，对缓存内数据进行编辑，输入的数据为 32'h22222222。u_b_h_w == 2，说明将输入的 1 个字的数据全部用来编辑缓存内数据。数据来自内存地址 32'h00000010，索引为 1，标签为 23'h00000，很明显，在时钟负边沿前，缓存的 set 0 中，索引为 1 的数据标签（tag1）同为 23'h00000，且数据有效（valid1 == 1），所以写命中，hit1 = 1，时钟负边沿后 hit 也更新为 1。在时钟负边沿后，由于 recent1 == 1，valid 被更新为 valid2 的值，故仍为 0。时钟负边沿后，写被执行，空间内数据由 32'h11111111 变为 32'h22222222，故 word1 也发生了相应的更新；同时数据变“脏”，dirty 位由 0 变为 1，故 dirty1 也出现相应变化。

counter == 32'h00000012 时，edit == 1，对缓存内数据进行编辑，输入的数据为 32'h22222222。u_b_h_w == 2，说明将输入的 1 个字的数据全部用来编辑缓存内数据。数据来自内存地址 32'h00000010，索引为 2，标签为 23'h00000，很明显，在时钟负边沿前，缓存的两个 set 的索引为 2 的空间中均没有有效数据，valid1 = valid2 = 0。故写未命中，hit1 = hit2 = 0；时钟负边沿后，valid 在更新后自然仍为 0。

counter == 32'h00000013 时，load == 1，进行读取操作，读取地址为 32'h00000014 处的数据。该地址对应的索引为 1，显然在时钟负边沿前，set 0 中索引 1 数据有效（valid1 == 1），且其标签 tag1 与该地址的标签同为 23'h00000，故读命中，hit1 为 1，时钟负边沿后 hit 与 valid 仍为 1。时钟负边沿后，更新该数据的 recent 值为 1，故 recent1 变为 1。时钟负边沿后，dout 更新为被读取出的数据，即 word1 = 32'h11111111，输出到缓存外。

counter == 32'h00000014 时，store == 1，往缓存中存储数据，数据为 32'h33333333，来自地址 32'h00000214，索引为 1，标签为 23'h00001。显然在时钟负边沿前，只有 set 0 中的索引 1 数据有效（valid1 == 1），但是其标签 tag1 为 23'h00000，与要存入的数据来源的地址不同，所以 hit1 = hit2 = 0。由于 recent1 == 1，于是 set 0 中的数据不用被替换，数据存入 set 1 中。时钟负边沿后，hit 自然被更新为 0，由于负边沿前 recent1 == 1，valid 在负边沿后被更新为负边沿前的 valid2 的值，即 0，tag 取负边沿前的 tag1 = 23'h00000。时间负边沿后，数据存入，于是 word2 更新为存入的数据，valid2 由 0 变 1，tag2 变成存入的数据的

地址对应的标签。易得数据存入后，缓存内的标签和索引自然与存入数据的地址是匹配的，所以 hit2 由 0 变为 1。

counter == 32'h00000015 时，edit == 1，编辑缓存内数据，输入的数据为 32'h44444444。u_b_h_w == 2，说明将输入的 1 个字的数据全部用来编辑缓存内数据。数据来自内存地址 32'h00000214，索引为 1，标签为 23'h00001，很明显，在时钟负边沿前，缓存的 set 1 中，索引为 1 的数据标签（tag2）同为 23'h00001，且数据有效（valid2 == 1），所以写命中，hit2 = 1，时钟负边沿后 hit 也更新为 1。时钟负边沿前，recent1 == 1，recent2 == 0，故在时钟负边沿后，更新 recent2 为 1，recent1 为 0。时钟负边沿后，由于 recent1 == 0，所以 valid 被更新为 valid1 的值，tag 被更新为 tag1 的值，故 valid 变为 1，tag 变为 23'h00001。时钟负边沿后，写被执行，空间内数据由 32'h33333333 变为 32'h44444444，故 word2 也发生了相应的更新；同时数据变“脏”，dirty 位由 0 变为 1，故 dirty2 也出现相应变化。

counter == 32'h00000016 时，load == 1，读取内存地址为 32'h00000014 的数据。该地址对应的索引为 1，显然在时钟负边沿前，缓存的 set 0 中索引 1 数据有效（valid1 == 1），且其标签 tag1 与该地址的标签同为 23'h00000，故读命中，hit1 为 1，时钟负边沿后 hit 为 1。缓存中的该数据为脏，故 dirty1 显示为 1。时钟负边沿后，更新该数据的 recent 值为 1，故 recent1 变为 1；相应地，相同索引的 set 1 中的 recent 变为 0，故 recent2 变为 0。时钟负边沿后，dout 更新为被读取出的数据，即 word1 = 32'h11111111，输出到缓存外。由于时钟负边沿前 recent1 等于 0，时钟负边沿后，valid 更新为 valid1 = 1，dirty 更新为 dirty1 = 1。

counter == 32'h00000017 时，load == 1，读取内存地址为 32'h00000414 的数据。该地址对应的索引为 1，显然在时钟负边沿前，缓存中所有索引为 1 的数据的标签（tag1 为 23'h00000，tag2 为 23'h00001）均不与该地址的标签（32'h00002）相匹配，故读未命中，hit1 = hit2 = 0。时钟负边沿后，hit 更新为 0。

counter == 32'h00000018 时，store == 1，往内存中存储数据 32'h55555555，数据来源于地址 32'h00000414，索引为 1，标签为 23'h00002。在时钟负边沿前，该数据的标签与缓存内索引为 1 的所有有效数据的标签（tag1 = 23'h00000，tag2 = 23'h00001）均不匹配，故 hit1 = hit2 = 0。缓存内在索引为 1 的空间中，由于

recent1 == 1，所以新数据存储于 set 1 中，覆盖掉原有的数据。在时钟负边沿后存入数据，于是 set 1 中索引为 1 的数据变为 32’h55555555，标签变为 23’h00002，故 word2、tag2 作出相应变化。同时该数据不脏，dirty 被置为 0。易得数据存入后，存入缓存空间内的标签和索引自然与存入数据的地址是匹配的，所以 hit2 由 0 变为 1。

五、　讨论与心得

这次实验让我对缓存的运行原理有了更深刻的认识。尤其是标签和索引的比对这方面的实现上，本实验是先得到要访问的地址的索引，再在缓存中转到相应的索引空间，看是否有有效数据的标签与要访问的地址的标签相同，如果有则命中，没有则未命中。我一开始没有看懂这种实现方式，后来看懂之后直呼巧妙。