浙江大学

本科实验报告

Cache Controller

课程名称：	计算机组成与设计
姓　　名：	周灿松
学　　院：	信息与电子工程学院
系：	
专　　业：	信息工程
学　　号：	**3190105055**
指导老师：	刘鹏

2021 年 12 月 28 日

目录

一 有限状态机设计

1. 状态编码设计

如图 1所示，我参考教材 5.9 节设计了一共四个状态，分别是：空闲状态 IDLE，Tag 位比较状态 CompareTag，写回状态 WriteBack 以及分配状态 Allocate，各个状态作用如下

(1) IDLE：等待来自 CPU 的控制信号 ld 以及 st，当这两个信号有效时，转移到 CompareTag 状态。

(2) CompareTag：比较来自 Cache 的对应 Block 的 Tag 位与来自 CPU 的地址 Addr 的 [31:11] 位，同时根据对应 Block 的 Valid 与 Dirty 位判断是 Hit 或者 Miss，以及转移到对应的状态。

(3) WriteBack：在此状态将 Cache 中的 Block 写回 L2 Cache 中，当接收到来自 L2 的写回完成信号 write_done 后，写回完成，状态转移到 Allocate。

(4) Allocate：此状态用于在 L1 Cache miss 时将 L2 Cache 中的值加载到 L1 Cache 中，当得到写入完成信号 l2_ack 后，说明写入完成，状态切换到 CompareTag。

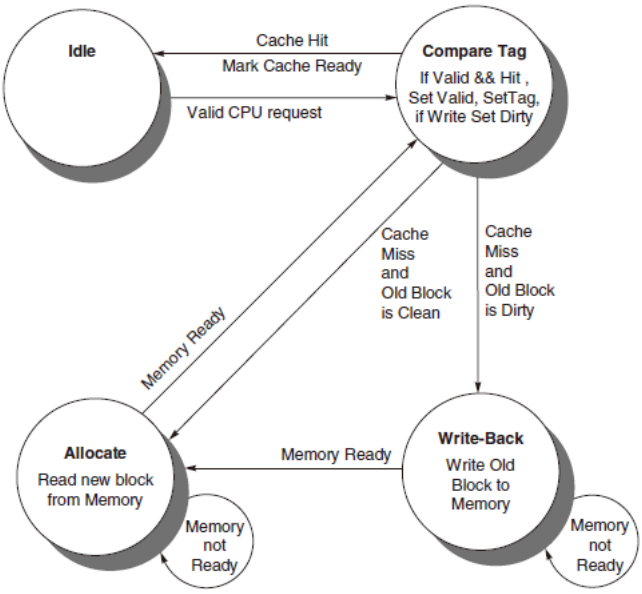

图 1: state diagram

各个状态的状态编码如表 1所示

状态编码	
IDLE	00
CompareTag	01
WriteBack	10
Allocate	11

表 1: 状态编码

2. 状态转换表

状态转移表如表 2所示，‘–’表示在当前状态转移情况下不关心该信号。

state	**ld**	**st**	**tag==addr[31:11]**	**dirty**	**valid**	**l2_ack**	**write_done**	**nextstate**
IDLE	0	0	–	–	–	–	–	IDLE
	0	1	–	–	–	–	–	CompareTag
	1	0	–	–	–	–	–	
	1	1	–	–	–	–	–	
CompareTag	–	–	–	0	0	–	–	Allocate
	–	–	–	1	0	–	–	WriteBack
	–	–	–	–	1	–	–	IDLE
WriteBack	–	–	–	–	–	–	1	Allocate
	–	–	–	–	–	–	0	WriteBack
Allocate	–	–	–	–	–	1	–	CompareTag
	–	–	–	–	–	0	–	Allocate

表 2: 状态转移表

3. FSM 图

FSM 图如图 2所示。

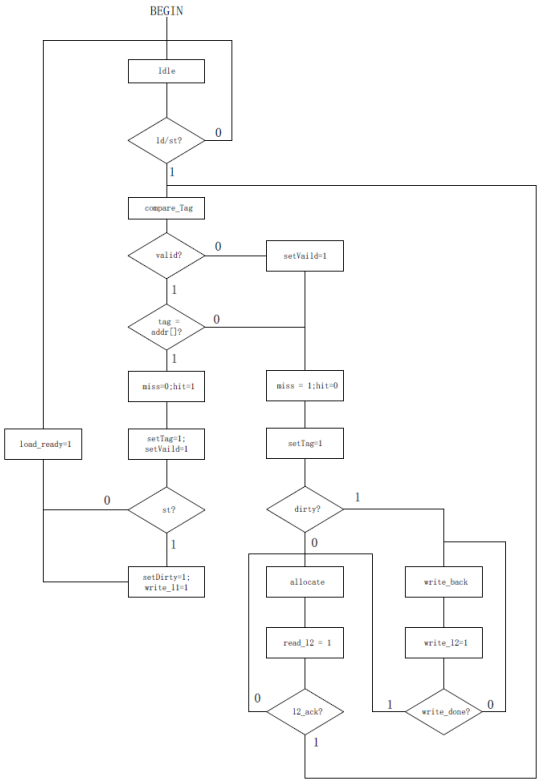

图 2: FSM 图

二 电路设计

我通过 Vivado 软件进行了电路设计，电路图如图 3所示。

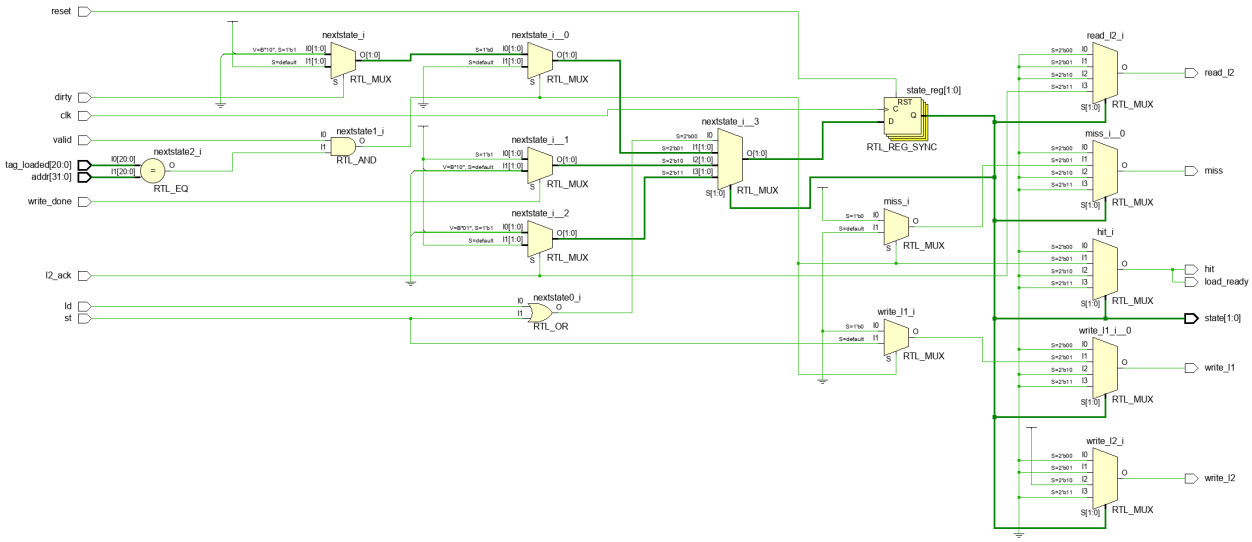

图 3: 电路图

三 Verilog 实现

1. 信号含义详细阐释

(1) clk: 时钟信号

(2) reset: 重置信号

(3) ld: 数据由 L1 加载到 CPU

(4) st: 数据由 CPU 写入 L1

(5) addr: 数据所在地址

(6) valid:L1 的有效位

(7) dirty:L1 的脏位

(8) tag_loaded:L1 对应位置的 Tag

(9) l2_ack: 数据是否完成从 L2 写入 L1 的过程

(10) write_done: 数据是否完成从 L1 写回 buffer 的过程

(11) hit:hit 信号

(12) miss:miss 信号

(13) load_ready:L1 数据是否准备就绪

(14) write_l1: 使能 L1 的写入功能

(15) read_l2: 发向 L2 的 load 指令

(16) write_l2: 使能 L1 写回 buffer

2. Verilog 代码

Cache 控制器的代码 Listing 1所示。

3. 测试样例

所用测试代码如 Listing 2所示。

4. 测试结果

Cache Controller 测试结果如图 4所示，各个控制信号正确，功能能够正常实现

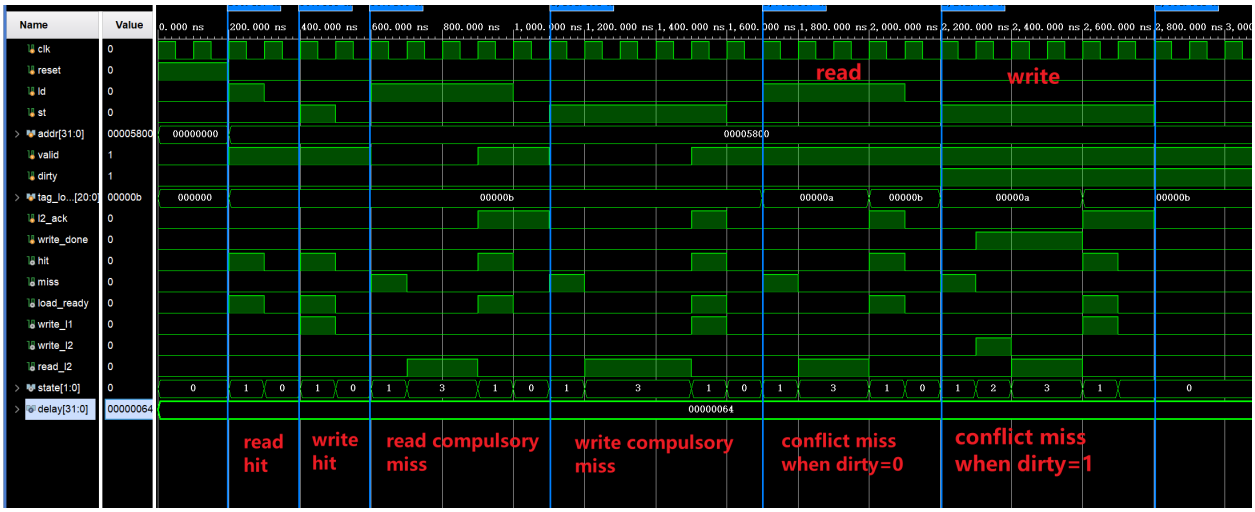

图 4: 测试结果

四 心得体会

通过本次实验，我对 Cache 的整个控制机制有了一个更加具体的理解，同时也补足了自己在学习 Cache 相关内容时所遗漏的知识点。

除此之外，这次实验也是我第一次编写 testbench 文件，对于 Verilog 语言的测试方式有了一个更加透彻的掌握。

五 附录

1. cache 代码

Listing 1: Cache 控制器代码

```
module cache_controller (
    input clk,reset,
    input ld , st,
    input [31:0] addr,
    input valid , dirty,
    input [20:0] tag_loaded,
    input l2_ack, write_done,
    output reg hit , miss,
    output reg load_ready , write_l1,
    output reg write_l2 , read_l2,
    output state
);
    parameter IDLE = 0 , CompareTag = 1 , WriteBack = 2 , Allocate = 3;
    reg [1:0] state , nextstate;

    always @(posedge clk) begin
        if(reset) state = IDLE;
        else state = nextstate;
    end

    always @(*) begin
        hit = 0 ; miss = 0;
        load_ready = 0 ; write_l1 = 0;
        write_l2 = 0 ;read_l2 = 0;
        case (state)
            IDLE : begin
                if(ld || st) begin
                    nextstate = CompareTag;
                end
                else nextstate = IDLE;
            end

            CompareTag : begin
                if(!(valid && tag_loaded == addr[31:11]))begin
                    miss = 1 ;
                    if(dirty)begin
                        nextstate = WriteBack;
                    end
                    else begin
                        nextstate = Allocate;
                    end
```

```
                end
                else begin
                    if(st)begin
                        hit = 1 ;
                        load_ready = 1 ; write_l1 = 1;
                    end
                    else begin
                        hit = 1 ; miss = 0;
                        load_ready = 1;write_l1 = 0;
                        write_l2 = 0 ; read_l2 = 0;
                    end
                    nextstate = IDLE;
                end
            end

            WriteBack : begin
                write_l2 = 1;
                if(write_done)begin
                    nextstate  = Allocate;
                end
                else begin
                    nextstate = WriteBack;
                end
            end

            Allocate : begin
                read_l2=1;
                if(l2_ack)begin
                    nextstate = CompareTag;
                end
                else begin
                    nextstate = Allocate;
                end
            end
        endcase
    end
endmodule
```

2. 测试代码

Listing 2: Cache 控制器测试文件

```
`timescale 1ns / 1ps

module testbench;
    parameter delay = 100;
    //inputs
    reg clk,reset;
    reg ld , st;
    reg [31:0] addr;
    reg valid , dirty;
    reg [20:0] tag_loaded;
    reg l2_ack;
    reg write_done;

    //outputs
    wire hit , miss;
    wire load_ready , write_l1;
    wire write_l2 , read_l2;
    wire [1:0]state;

    //instantiation of the cache controller
    cache_controller cache(
        .clk(clk),
        .reset(reset),
        .ld(ld),
        .st(st),
        .addr(addr),
        .valid(valid),
        .dirty(dirty),
        .tag_loaded(tag_loaded),
        .l2_ack(l2_ack),
        .hit(hit),
        .miss(miss),
        .load_ready(load_ready),
        .write_l1(write_l1),
        .write_l2(write_l2),
        .read_l2(read_l2),
        .write_done(write_done),
        .state(state)
    );

    //set the free running clock
    initial begin
```

```
        clk = 1;
        forever begin
            #(delay/2) clk = ¬clk;
        end
    end

    //set up the reset signal
    initial begin
        reset = 1 ;
        #(delay*2) reset = 0;
    end

    initial begin
        //initial input
        ld = 0;
        st = 0;
        addr = 32'b0000_0000_0000_0000_0000_0000_0000_0000;
        tag_loaded = 21'b0000_0000_0000_0000_0000_0;
        valid = 0;
        dirty = 0;
        l2_ack = 0;
        write_done = 0;


        //read hit
        #(delay*2)
            ld = 1 ; st = 0;
            addr = 32'b0000_0000_0000_0000_0101_1000_0000_0000;
            tag_loaded = 21'b0000_0000_0000_0000_0101_1;
            valid = 1;
        //wait for next instruction
        #delay
            ld = 0 ; st = 0;
        //write hit
        #delay
            ld = 0 ; st = 1;
        //wait for next instruction
        #delay
            ld = 0 ; st = 0;

        //read compulsory miss
        #(delay)
            ld = 1 ; st = 0;
```

```
86              valid = 0;
            //wait for the l2_ack
88          #(delay*3)
                l2_ack = 1;
90              valid = 1;
            //wait for next instruction
92          #delay
                ld = 0 ; st = 0;
94
            //write compulsory miss
96          #(delay)
                ld = 0 ; st = 1;
98              valid = 0;
                l2_ack = 0;
100         //wait for the l2_ack
            #(delay*4)
102             l2_ack = 1;
                valid = 1;
104         //wait for next instruction
            #delay
106             ld = 0 ; st = 0;
                l2_ack = 0;
108
            //conflict miss with dirty=0
110         #(delay)
                ld = 1 ; st = 0;
112             tag_loaded = 21'b0000_0000_0000_0000_0101_0;
                valid = 1;
114         //wait for the l2_ack
            #(delay*3)
116             l2_ack = 1;
                tag_loaded = 21'b0000_0000_0000_0000_0101_1;
118         #delay
                ld = 0 ; st = 0;
120             l2_ack=0;

122         //conflict miss with dirty=1
            #(delay)
124             ld = 0 ; st = 1;
                tag_loaded = 21'b0000_0000_0000_0000_0101_0;
126             dirty = 1;
            //wait for the write_done signal
128         #(delay)
```

```
            write_done = 1;
130         #(delay*3)
            write_done = 0;
132         l2_ack = 1;
            tag_loaded = 21'b0000_0000_0000_0000_0101_1;
134         #(delay*2)
            ld = 0 ;st = 0;
136         l2_ack=0;

138         #(delay*3)

140         $stop;

142         end

144 endmodule
```