浙江大學

本科生课程报告

学年、学期：2021 — 2022 学年 秋冬 学期

课程名称：计算机组成与设计

任课教师：赵武锋、沈继忠

题　　目：Cache Controller

学生姓名：黄嘉欣

学　　号：3190102060

目录

Project2：Cache Controller

3190102060 黄嘉欣 信工 1903 班

一、实验目的

① 设计一个一级数据缓存控制器；

② 加强对缓存结构和机制的理解。

二、实验原理

TMS320C621x/C671xDSP 采用有程序和数据两级内存架构，分别为第一级程序缓存 L1P、第一级数据缓存 L1D 和第二级内存 L2。其中，L1D 缓存的一些特定机制和参数如表 2.1 所示。

表 2.1 L1D 缓存部分机制和参数

A Simplified Model of TMS320C621x/C671x DSP	
Internal memory structure	Two Level
L1D size	4Kbytes
L1D organization	2-way set associative
L1D line size	32 bytes
L1D replacement strategy	Least Recently Used (LRU)
L1D read miss action	1 line allocates in L1D
L1D read hit action	Data read from L1D
L1D write miss action	No allocation in L1D, data sent to L2
L1D write hit action	Data updated in L1D, line marked dirty
L2 line size	128 bytes
L2 → L1D read path width	128-bit
L1D → L2 write path width	32-bit

注：为简单起见，假设 write buffer 的容量无限，不考虑 TLB；一次只允许执行一条 ld/st 指令。

① 地址分割

由表 2.1 可知，L1D 缓存的大小为 4K bytes，每行为 32=2^5 bytes，故总的 cache line 行数为$\frac{4\times1024}{32}=128$行，block offset 需要 5 位。又由于其采用 2 路组相联，则组数为$\frac{128}{2}=64=2^6$，组索引 Set Index 需占用 6 位。于是 L1D 的地址分割为：

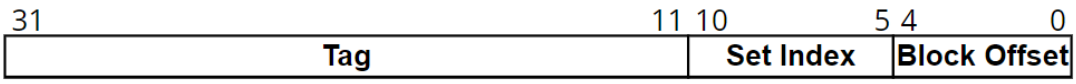

图 2.1 L1D 地址分割

② 读/写操作

由相关知识，在 CPU 发出读/写指令后，cache 与内存之间会进行数据交互，如下图所示：

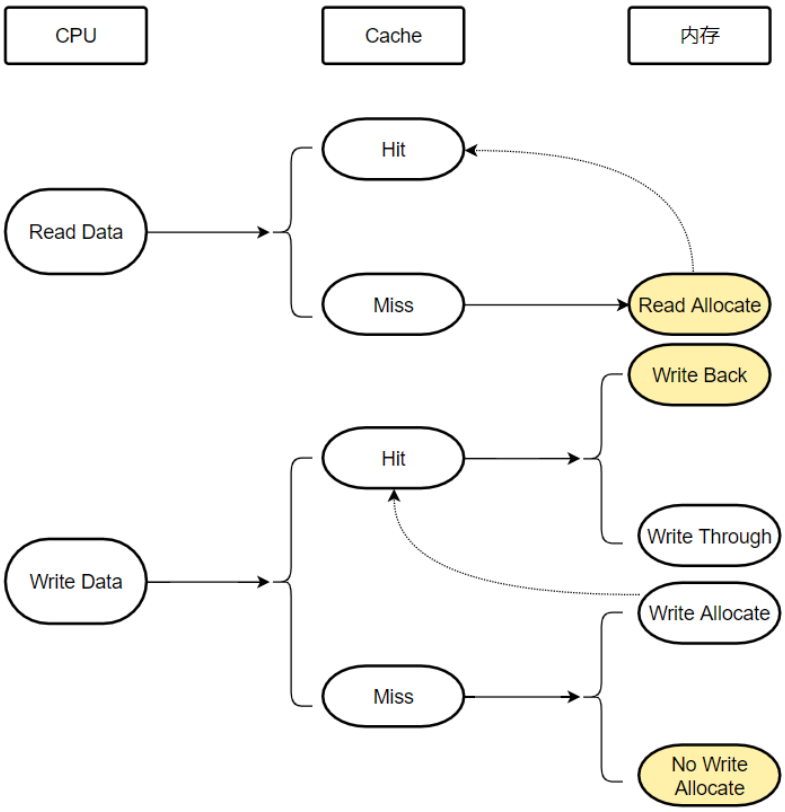

图 2.2 Cache 与内存的数据交互

根据表 2.1，当 L1D 读未命中时，系统会在 L1D 中分配 1 行；当读命中时，数据会直接从 L1D 中返回。当 L1D 写未命中时，系统不会分配行，而是直接将数据送到第二级内存 L2，因此其采用的是 No Write Allocate 机制；当写命中时，L1D 中的数据会更新，并将该行标记为脏块（dirty），故其采用的是 Write Back 方式，如图 2.2 中黄色填充标识所示。

由项目要求，在此项目中我们只需要考虑 L1D 的功能，写缓冲（Write Buffer）的容量无限，且不用考虑 TLB 和具体的数据替换（忽略 LRU 位）。当 CPU 需要读取数据时，若缓存命中（Read Data->Hit），则其可直接将数据读入；若未命中（Read Data->Miss），则系统会在 L1D 中分配一行，从 L2 中读出数据并将其存入 L1D，再实现读数据操作（此时一定会 Hit）。由于 L1D 应对写命中采用的是 Write Back 方式，写入数据仅被更新到 L1D，当该行被替换时才会将数据写入 L2，因此，在将数据从 L2 读到 L1D 之前，需要先判断被分配的一行是否为 dirty，若是，则需先回写数据，将该

行的数据保存到 L2 中，再从 L2 将所需数据读到 L1D；若不是，则可以直接从 L2 读出数据到 L1D。当 CPU 需要写入数据时，若缓存命中（Write Data->Hit），如上所述，新的写入数据会被暂时存储在 L1D 中的某一行，同时该行会被标记为 dirty；若未命中（Write Data->Miss），则第一级数据缓存 L1D 会被绕过，系统直接将数据写入缓冲，进而送到 L2。综上，由 L1D 对读指令和写指令的不同响应过程，可以绘制流程图如下：

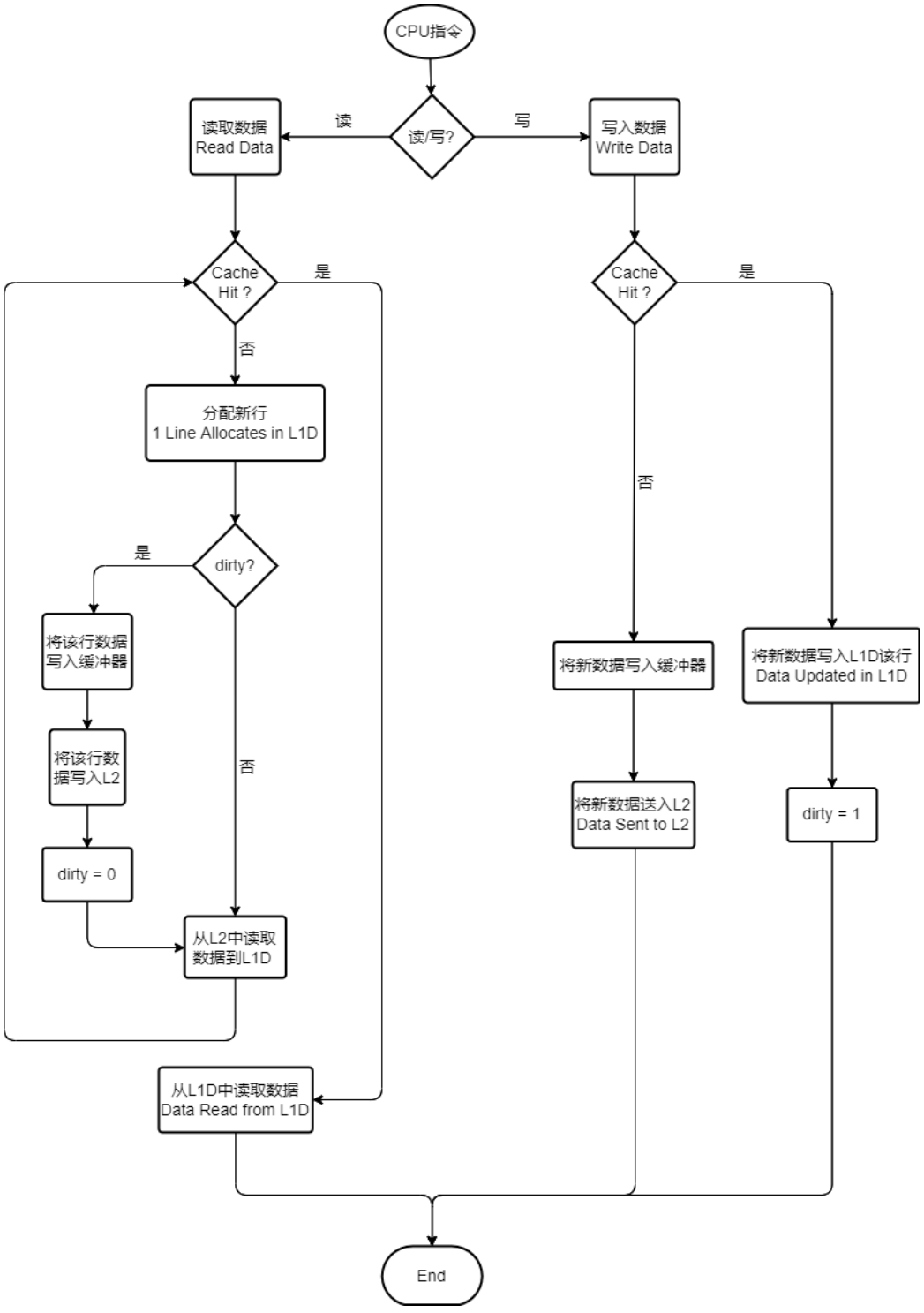

图 2.3 L1D 的读写操作

三、实验步骤

① 状态机设计

(1) 绘制状态图，为每个状态分配唯一的二进制码并标记转移条件信号；

(2) 由状态图，列出状态转移表。其根据所有可能的条件信号，给出每个当前状态的下一个状态；

(3) 绘制缓存控制器的流程图。

② 电路设计

(1) 根据流程图设计缓存控制器并绘制电路；

(2) 在线外标出重要的信号，并在图旁列出每个信号的含义和功能（如有必要，还应列出它们的源和目的地）。

③ Verilog HDL 实现

(1) 根据电路设计，使用 Verilog HDL 描述电路；

(2) 添加足够的注释，使程序易于理解。项目的重点在于行为的描述，如果需要的话，可以添加一些结构和数据流的描述；项目实现为缓存控制器的输出控制信号和状态变化：可以忽略特定的数据。

注：模块名称和一些信号名称如下，其不能在程序中改变。每个信号的含义已经给出，以供参考。

```
module cache_controller(…);
    input ld, st;           // load or store request
    input addr[31:0];       // cache access address
    input tag_loaded[20:0]; // the tag of the indexed cache data address
    input valid;            // valid indication of the block in cache
    input dirty;            // indicate whether the cache line is dirty
    input l2_ack;           // indicate that the data loaded from L2 is arrived
    output hit, miss;       // indicate the access is a hit or miss
    output load_ready;      // indicate the data is successfully loaded and the
                            // data is ready for processor
    output write_l1;        // enable for L1 cache write
    output read_l2;         // load request to L2 cache
    output write_l2;        // enable for write back to write buffer
```

四、实验设计

① 状态机设计

(1) 确定状态与流程

由图 2.3 及输入输出信号的定义，可以发现缓存控制器存在以下 4 个状态：

i) 闲置（Idle）：没有指令输入，控制器处于闲置状态，此时所有输出均为 0。当 CPU 发出 ld/st 指令后，转移到标志比较状态；

ii) 标志比较（CompareTag）：比较地址的 tag 与相应缓存的 tag、确定 valid 的值，以判断缓存是否命中。若命中（hit=1），则输出信号 load_ready=1，使处理器从 L1D 中读取数据；或输出信号 write_l1=1，将新数据写入 L1D。完成上述操作后，控制器会返回闲置状态。若未命中（miss=1），则会根据指令类型进行分类。当执行 ld 读指令时，若分配到的新行为脏块（dirty=1），则输出 write_l2=1，转移到写入缓冲状态，否则输出信号 read_l2=1，次态为读入数据状态；当执行 st 写指令时，则直接令 write_l2=1 并进入写入缓冲状态；

iii) 写入缓冲（WriteBuffer）：读取数据时，将 L1D 中脏块的数据送入 L2，再输出信号 write_l2=0，read_l2=1，进入读入数据状态；或在写入数据时，将新的数据送入 L2，输出 write_l2=0，然后转移到闲置状态。注意，由于 L1D->L2 的通路宽度为 32 位，而 L1D 每行的大小为 32 字节，故需要 8 个时钟周期才能完成数据的写入。定义变量 WB_ack，当写入未完成时，WB_ack=0，否则将其赋值为 1，此过程可在系统中加入计数器完成；

iv) 读入数据（ReadData）：从 L2 中读取需要的数据到 L1D，读取完毕后输出信号 read_l2=0。与 iii)同理，由于 L2->L1D 的通路宽度为 128 位，而 L2 每行大小为 128 字节，故需要 8 个时钟周期才能完成数据的读取，此时输入 l2_ack=1。之后控制器回到标志比较状态，完成数据的读取操作。

(2) 绘制状态图

显然，四个状态只需要两位二进制数即可编码。分别为上述四个状态指定二进制编码为:Idle—2’b00,CompareTag—2’b01,WriteBuffer—2’b10,ReadData—2’b11，则根据流程，可以画出状态图为：

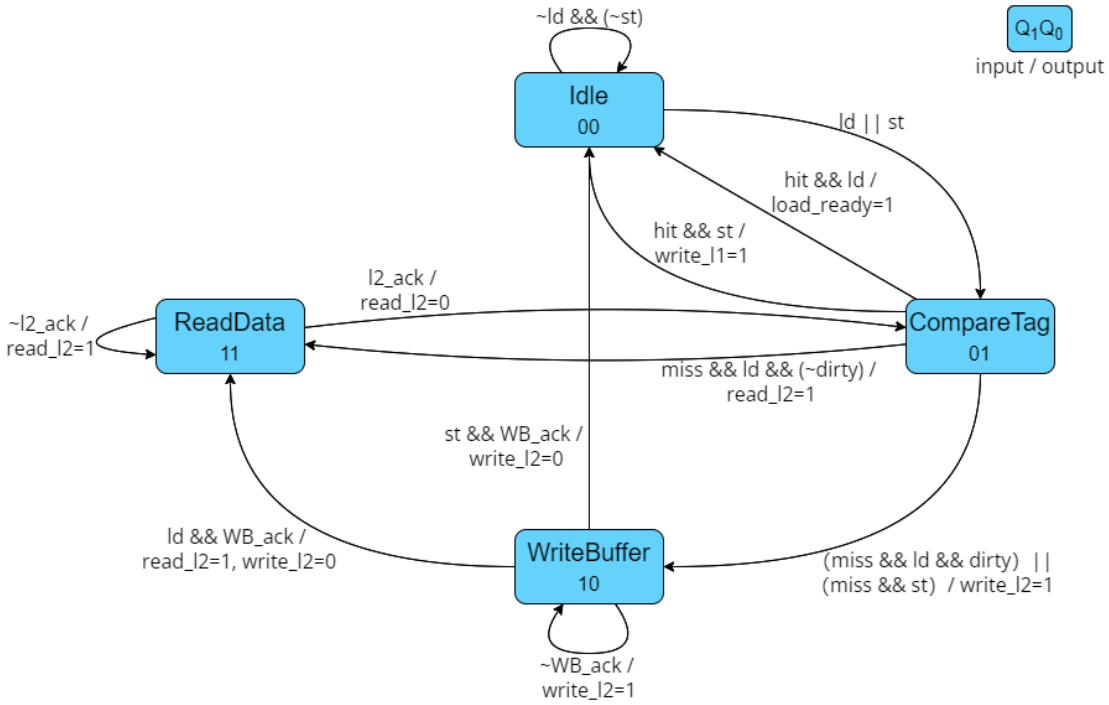

图 4.1 缓存控制器的状态图

(3) 绘制状态转移表

根据前述流程，可以画出状态转移表为：

表 4.1 缓存控制器的状态转移表

现态		次态		转移条件	输出
Q_1	Q_0	Q_1'	Q_0'		
0	0	0	0	~ld&&(~st)	/
		0	1	ld||st	/
0	1	0	0	hit&&ld	load_ready=1
				hit&&st	Write_l1=1
		1	0	(miss&&ld&&dirty) ||(miss&&st)	write_l2=1
		1	1	miss&&ld&&(~dirty)	read_l2=1
1	0	0	0	st&&WB_ack	write_l2=0
		1	0	~WB_ack	Write_l2=1
		1	1	ld&&WB_ack	read_l2=1 write_l2=0
1	1	0	1	l2_ack	read_l2=0
		1	1	~l2_ack	read_l2=1

(4) 绘制流程图

根据状态转换图 4.1，可以画出缓存控制器的 ASM 图为：

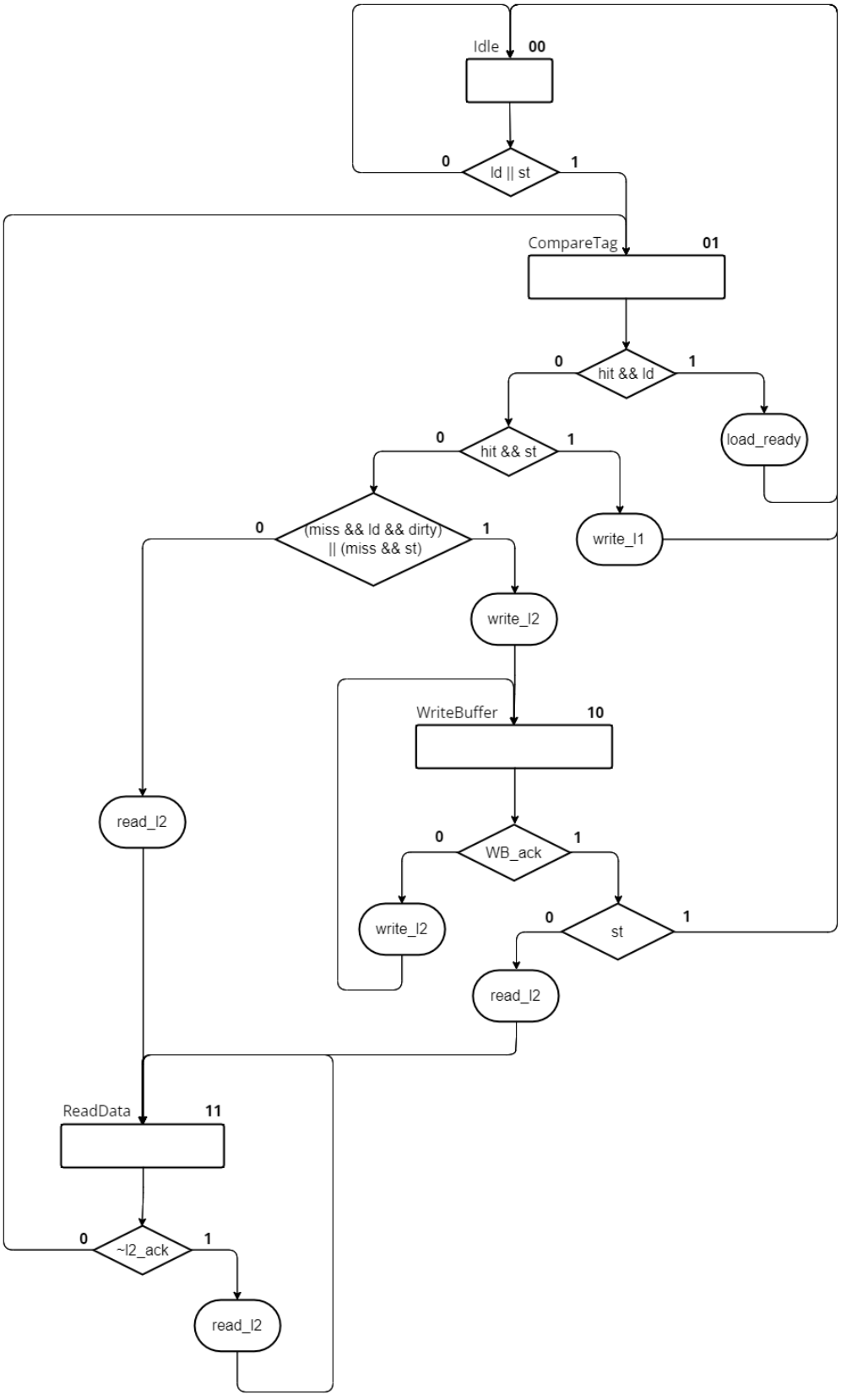

图 4.2 缓存控制器的 ASM 图

② 电路设计

根据①中的状态机设计，可以使用数据选择器型控制器完成相应的电路设计，将当前状态作为选择器的控制信号，并对数据选择器的各个输入值进行选通。相比于计数器型控制器，当 ASM 图发生变化时，我们只需要对电路中相应的 MUX 输入作适当改变即可，不至于牵动全局、重新计算各状态的激励函数。由于状态编码为 2 位，故电路需要两个 4 选 1 数据选择器，分别记为 MUX0、MUX1，其中 MUX0 产生 Q_0'，MUX1 产生 Q_1'。将数据选择器的输出信号连接到两个 D 型触发器的输入端，即为触发器的次态激励函数，触发器的 Q 输出端为控制器的状态编码 Q_1Q_0，经译码后即可得知其具体状态名。与此同时，将 Q_1、Q_0 分别接回数据选择器的两个控制端，作为地址信号，从而可以选出下一个状态的编码。

(1) 数据选择器输入

由状态转移表和 ASM 图，可以得到数据选择器的输入为：

MUX0(0) = ld + st，　　MUX1(0) = 0

MUX0(1) = miss·ld·$\overline{\text{dirty}}$，　　MUX1(1) = $\overline{\text{hit}\cdot(\text{ld}+\text{st})}$ = miss

XMU0(2) = ld·WB_ack，　　MUX1(2) = ld + $\overline{\text{WB\_ack}}$

MUX0(3) = l2_ack + $\overline{\text{l2\_ack}}$ = 1，　　MUX1(3) = $\overline{\text{l2\_ack}}$

(2) 输出信号

由状态转移表和 ASM 图，可以得到电路各输出信号的逻辑表达式为：

load_ready = $\overline{Q}_1\cdot Q_0$·hit·ld

Write_l1 = $\overline{Q}_1\cdot Q_0$·hit·st

write_l2 = $\overline{Q}_1\cdot Q_0$·((miss·ld·dirty) + (miss·st)) + $Q_1\cdot\overline{Q}_0\cdot\overline{\text{WB\_ack}}$

read_l2 = $\overline{Q}_1\cdot Q_0$·miss·ld·$\overline{\text{dirty}}$ + $Q_1\cdot\overline{Q}_0$·ld·WB_ack + $Q_1\cdot Q_0\cdot\overline{\text{l2\_ack}}$

其中，hit 和 miss 存在关系：miss = ~hit

由于系统采用两路组相联，故当且仅当一个组的两块中有一个块的 tag 与地址的 tag 匹配且该块的 valid=1 时，缓存命中。不妨设一个组中两块的 tag 分别为 tag0_loaded、tag1_loaded，valid 位分别为 valid0、valid1，是否命中为 hit0、hit1，则根据地址分配，有：

```
hit0 = (tag0_loaded==addr[31:11])&&valid0
hit1 = (tag1_loaded==addr[31:11])&&valid1
```

hit = (ld||st)&&(hit0||hit1)

令一个方面，由于在读取数据、未命中情况中需要分配新行，此涉及到具体的数据替换，需要根据 LRU 位选择 cache line，再检查该行的 dirty 位是否为 1。因此，考虑到项目要求，可以直接令输入信号 dirty 表示该行是否是脏块，而非设两路 block 的 dirty 位分别为 dirty0、dirty1，进而避免对 LRU 位的判断和修改。

(3) 绘制电路图

根据数据选择器的输入和电路输出信号表达式，可以绘制电路图如下：

i) 命中判断电路

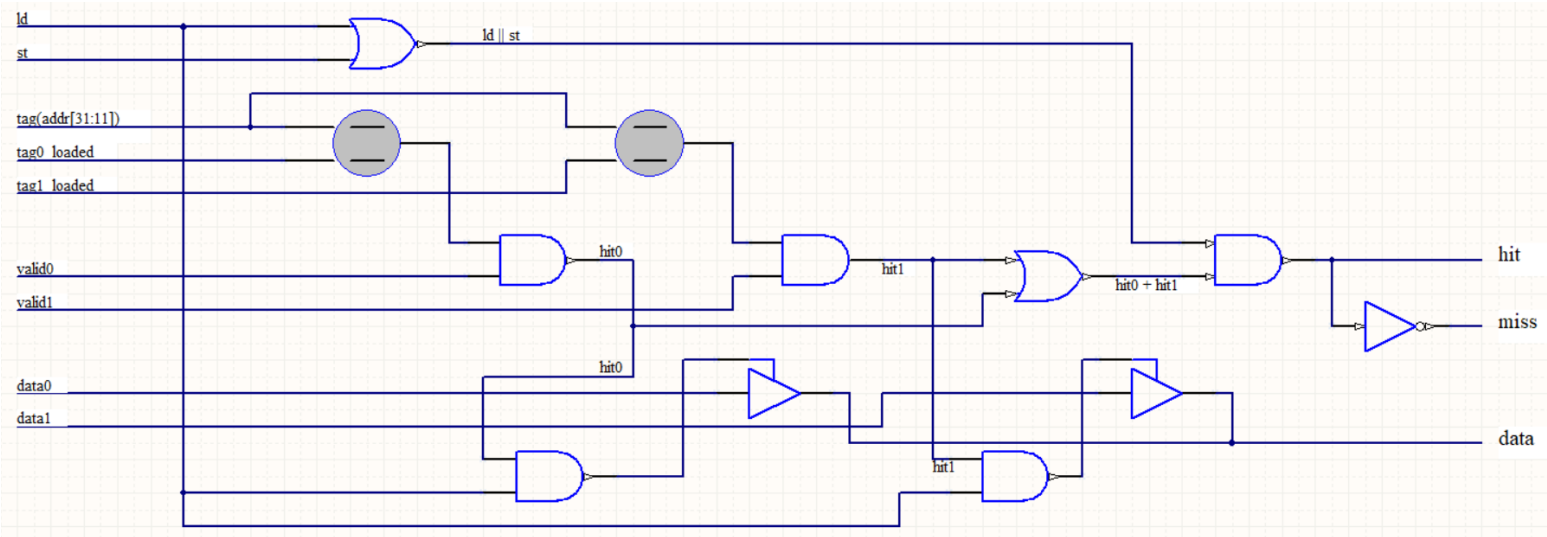

图 4.3 命中判断电路

图中各信号的含义和功能如下表所示（信号类型是指信号在总电路设计中的类型）：

表 4.2 命中判断电路信号含义表

信号类型	信号	含义及功能
输入信号	ld	CPU 读取指令，高电平有效
	st	CPU 写入指令，高电平有效
	tag	等于 addr[31:11]，用于与 cache 块中的 tag_loaded 比较，以判断是否命中
	tag0_loaded / tag1_loaded	cache 组中第一/第二个块的 tag
	valid0 / valid1	cache 组中第一/第二个块的 valid 位

内部信号	hit0 / hit1	组中第一/第二个块命中，高电平有效
	hit	缓存命中，高电平有效
	miss	缓存未命中，高电平有效
\	data0 / data1	cache 组中第一/第二个块的 data
	data	从 cache 中取出的数据

显然，由于系统采用了 2 路组相联，组中的两个块都含有 tag、valid 位和 data 等，如电路中 tag0_loaded、tag1_loaded 等所示。当根据 Set Index 找到对应的组后，我们需要将输入地址中的 tag 分别与两块中的 tag 进行比较，确定 cache line，再根据其 valid 位判断是否命中。若命中，则 hit0 或 hit1 为高电平，根据(2)中表达式，利用与或非门即可完成 hit 等信号的赋值。需要注意的是，我在图中画出了数据的选通电路（逻辑上实现，实际可能并非如此），当组中的两块之一被命中时，不妨设第一块被命中，则 hit0=1，此时前一个三态门使能高电平，正常输出；而 hit1=0，对应的三态门输出为高阻态，故有 data=data0。若两块都未命中，则三态门输出都是高阻态，无数据输出。

ii) 控制器电路

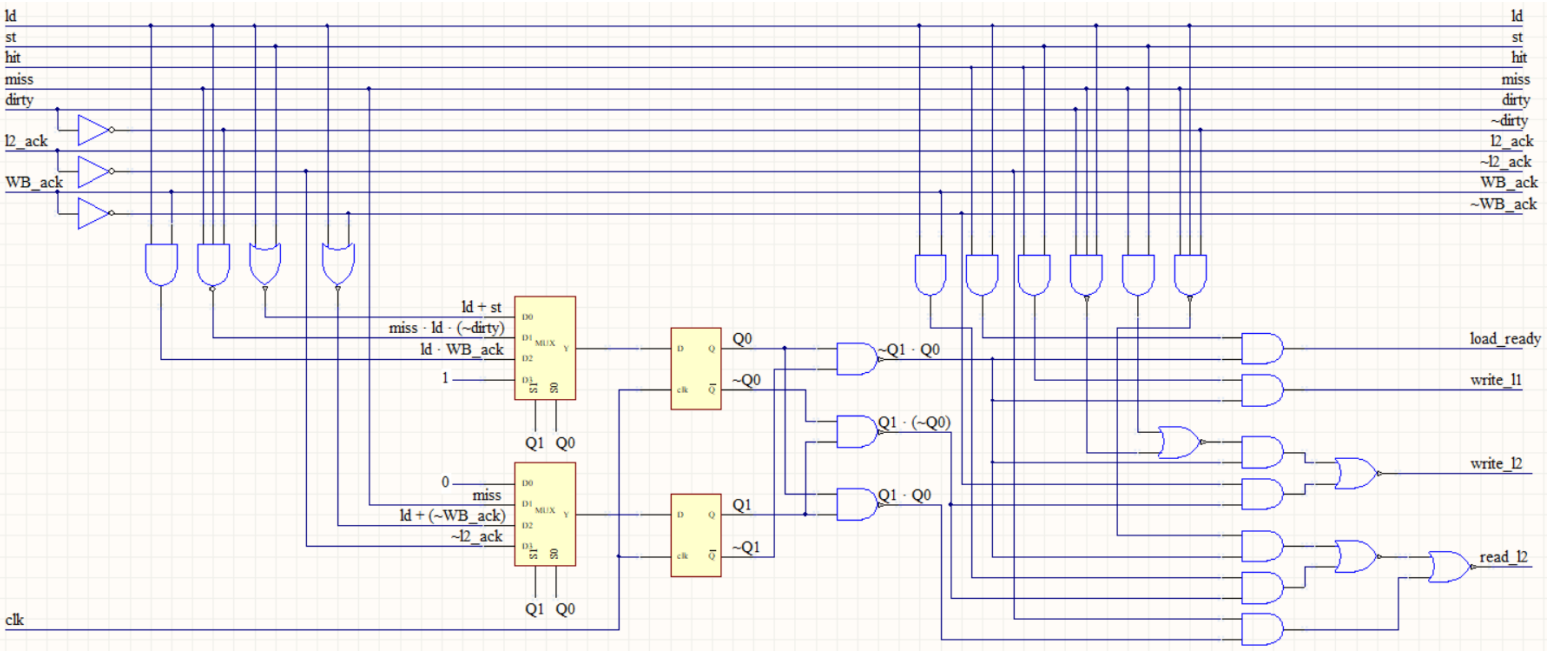

图 4.4 控制器电路

图中各信号的含义和功能如下表所示（信号类型是指信号在总电路设计中的类型）：

表 4.3 控制器电路信号含义表

信号类型	信号	含义及功能
输入信号	clk	时钟信号
	ld / st	CPU 读取/写入指令，高电平有效
	dirty	脏块，高电平有效
	l2_ack	已从 L2 中读取数据到 L1D
内部信号	hit	缓存命中，高电平有效
	miss	缓存未命中，高电平有效
	WB_ack	已将数据从 L1D 写入缓冲，由计数器赋值，当计时满 8 个周期后置为 1
	Q0 / Q1	D 寄存器输出，状态编码
输出信号	load_ready	可从 L1D 读取数据到 CPU，高电平有效
	write_l1	允许向 L1D 中写入数据
	write_l2	允许向缓冲/L2 中写入数据
	read_l2	请求从 L2 中读取数据

如图 4.4，根据(1)中数据选择器的输入函数，可以将对应的输入信号或内部信号通过逻辑门进行组合，最终传递到选择器的输入端。由于将状态编码作为了数据选择器的控制信号，当 D 触发器输出编码之后，电路能够得到反馈，根据该编码选通选择器输入，即次态编码，从而完成状态的转移。根据(2)中的输出信号表达式，可以画出相应的组合电路。除此之外，作为判定写回缓冲操作是否完成的信号，WB_ack 由内置的计数器（电路图中未画出）进行赋值，当开始写入时计数器即开始计数，满 8 个周期后，数据已完全写入缓冲，WB_ack 置 1。

③ Verilog HDL 设计

由项目要求，此缓存控制器只关注控制信号的输出以及状态的转换，忽略特定的数据传输。因此，在 Verilog HDL 实现时，我们不需要考虑组的确定和数据的定位，也不需要考虑 LRU 位以及数据的具体读/写操作，对应变量和流程将不会在设计中体现。

当然，正如输出信号和命中判断电路设计中所述，由于每个组中含有两路，在判断是否缓存命中时，需要对两路 block 都进行比较，故定义变量 tag0_loaded/tag1_loaded、hit0/hit1 等，与电路中一致。除此之外，为了能够初始化控制器，还需添加复位信号 reset，当其高电平时控制器回到 Idle 状态；在从 L2 中读取数据（ReadData 状态）或向缓冲/L2 中写入数据（WriteBuffer 状态）时，需要经历 8 个时钟周期才可完成，此由信号 l2_ack 和 WB_ack 表征。需要注意的是，由于 l2_ack 是输入信号，在编写测试文件时需确保自控制器进入 ReadData 状态后连续 8 个周期都有 l2_ack=0；而 WB_ack 是内部信号，由计数器赋值，当控制器进入 WriteBuffer 状态后计时 8 个周期，完成后令 WB_ack=1。综上，结合设计电路，可以编写 Verilog HDL 代码如下：

(1) 控制器有限状态机实现

```
module CacheController_FSM(clk, reset, ld, st, addr, tag0_loaded, tag1_loaded,
                           valid0, valid1, dirty, l2_ack,
                           hit, miss, load_ready, write_l1, read_l2, write_l2);
    input clk, reset;
    input ld, st;              // 读/写指令，高电平有效
    input [31:0] addr;         // 输入地址
    input [20:0] tag0_loaded;  // 组中第一个块的 tag
    input [20:0] tag1_loaded;  // 组中第二个块的 tag
    input valid0, valid1;      // 组中两块的 valid 位
    input dirty;               // 是否为脏块，高电平有效
    input l2_ack;              // 数据已从 L2 读取到 L1D，高电平有效

    output hit, miss;          // 缓存命中/未命中，高电平有效
    output reg load_ready;     // 可从 L1D 读取数据到 CPU，高电平有效
    output reg write_l1;       // 可将数据从 CPU 写入 L1D，高电平有效
    output reg read_l2;        // 可从 L2 读取数据到 L1D，高电平有效
    output reg write_l2;       // 可将数据从 L1D 写回缓冲/L2，高电平有效

    wire [20:0] tag;           // 输入地址中的 tag
    wire hit0, hit1;           // 命中组中第一/第二个块
    wire WB_ack;               // 数据已从 L1D 写入到缓冲/L2，高电平有效

    // 状态编码
    parameter Idle = 2'b00, CompareTag = 2'b01, WriteBuffer = 2'b10, ReadData =
2'b11;
    reg [1:0] state, nextstate;

    // 命中判断电路部分
    assign tag = addr[31:11];
```

```
assign hit0 = (tag0_loaded==tag)&&valid0;       // 组中第一个块是否命中
assign hit1 = (tag1_loaded==tag)&&valid1;       // 组中第二个块是否命中
assign hit = (ld||st)&&(hit0||hit1);            // 缓存是否命中
assign miss = ~hit;                             // 缓存是否未命中

// WriteBuffer 计数器
counter_n #(.n(8), .counter_bits(3)) WB_counter(.clk(clk),
             .en(write_l2),        // 当可将数据从 L1D 写回 L2 时，开始计数
             .r(~write_l2|reset),  // 当不可写回数据或系统重置时，计数器不工作
             .co(WB_ack),          // 计数满 8 个周期，数据写完，输出 WB_ack=1
             .q());
// 控制器(有限状态机)
always @ (posedge clk)
begin
    if (reset) state = Idle; // 重置为空闲态
    else state = nextstate;
end

always @ (*)
begin
    case (state)
        Idle:       begin
                        // 状态转移
                        if (ld||st) nextstate = CompareTag;
                        // 指令输入，转移到 CompareTag
                        else        nextstate = Idle;       // 否则保持 Idle
                        // 空闲态输出控制信号为 0
                        load_ready = 0; write_l1 = 0; read_l2 = 0; write_l2 = 0;
                    end
        CompareTag: begin
                        // 输出和状态转移与输入条件有关
                        if (hit&&ld)
                        begin
                            // 读命中，CPU 从 L1D 读取数据
                            nextstate = Idle; load_ready = 1;
                        end
                        else if (hit&&st)
                        begin
                            // 写命中，CPU 将数据写入 L1D
                            nextstate = Idle; write_l1 = 1;
                        end
                        else if ((miss&&ld&&dirty)||(miss&&st))
                        begin
                            // 读未命中，分配的行为脏块或写未命中，将数据写入缓冲/L2
```

```
                        nextstate = WriteBuffer; write_l2 = 1;
                    end
                    else // 读未命中，分配的行不是脏块，从 L2 中读取数据到 L1D
                    begin
                        nextstate = ReadData; read_l2 = 1;
                    end
                end
            WriteBuffer:begin
                    // 进入 WriteBuffer 态，向缓冲/L2 写数据，WB_counter 开始计数
                    if (~WB_ack)
                    begin
                        // 未写满 8 个时钟周期，状态保持，同时计数允许信号保持为 1
                        nextstate = WriteBuffer; write_l2 = 1;
                    end
                    else if (ld&&WB_ack)
                    begin
                        // 读操作且写满 8 个周期（从读未命中且脏块而来）
                        // 从 L2 读取数据到 L1D，同时计数结束
                        nextstate = ReadData; read_l2 = 1; write_l2 = 0;
                    end
                    else // 写操作且写满 8 个周期（从写未命中而来），指令完成
                    begin
                        nextstate = Idle; write_l2 = 0;
                    end
                end
            ReadData:   begin
                    // 从 L2 读取数据到 L1D，需要 8 个时钟周期
                    if (~l2_ack)
                    begin
                        // 未满 8 个周期，状态保持
                        nextstate = ReadData; read_l2 = 1;
                    end
                    else // 数据读取完毕，重新比较标志
                    begin
                        nextstate = CompareTag; read_l2 = 0;
                    end
                end
            default:    begin // 默认为空闲态，输出控制信号为 0
                    nextstate = Idle;
                    load_ready = 0; write_l1 = 0; read_l2 = 0; write_l2 = 0;
                end
        endcase
    end
```

```
endmodule
```

(2) 控制器逻辑电路实现

```
module CacheController_Logic(clk, reset, ld, st, addr, tag0_loaded, tag1_loaded,
                             valid0, valid1, dirty, l2_ack,
                             hit, miss, load_ready, write_l1, read_l2, write_l2);
    input clk, reset;
    input ld, st;              // 读/写指令，高电平有效
    input [31:0] addr;         // 输入地址
    input [20:0] tag0_loaded;  // 组中第一个块的 tag
    input [20:0] tag1_loaded;  // 组中第二个块的 tag
    input valid0, valid1;      // 组中两块的 valid 位
    input dirty;               // 是否为脏块，高电平有效
    input l2_ack;              // 数据已从 L2 读取到 L1D，高电平有效

    output hit, miss;          // 缓存命中/未命中，高电平有效
    output load_ready;         // 可从 L1D 读取数据到 CPU，高电平有效
    output write_l1;           // 可将数据从 CPU 写入 L1D，高电平有效
    output read_l2;            // 可从 L2 读取数据到 L1D，高电平有效
    output write_l2;           // 可将数据从 L1D 写回缓冲/L2，高电平有效

    wire [20:0] tag;           // 输入地址中的 tag
    wire hit0, hit1;           // 命中组中第一/第二个块
    wire WB_ack;               // 数据已从 L1D 写入到缓冲/L2，高电平有效
    wire D0, D1;               // D 触发器输入
    wire Q0, Q1;               // D 触发器输出
    wire w1, w2, w3;           // 内部信号，D 触发器输出的三个组合

    // 命中判断电路部分
    assign tag = addr[31:11];
    assign hit0 = (tag0_loaded==tag)&&valid0;       // 组中第一个块是否命中
    assign hit1 = (tag1_loaded==tag)&&valid1;       // 组中第二个块是否命中
    assign hit = (ld||st)&&(hit0||hit1);            // 缓存是否命中
    assign miss = ~hit;                             // 缓存是否未命中

    // WriteBuffer 计数器
    counter_n #(.n(8), .counter_bits(3)) WB_counter(.clk(clk),
                .en(write_l2),        // 当可将数据从 L1D 写回 L2 时，开始计数
                .r(~write_l2|reset),  // 当不可写回数据或系统重置时，计数器不工作
                .co(WB_ack),          // 计数满 8 个周期，数据写完，输出 WB_ack=1
                .q());

    // 控制器(逻辑电路)
```

```
    // 两个 4 选 1 数据选择器
    mux4_1 mux0(.in({1'b1, ld&&WB_ack, miss&&ld&&(~dirty), ld||st}),
                .addr({Q1, Q0}), .out(D0));
    mux4_1 mux1(.in({~l2_ack, ld||(~WB_ack), miss, 1'b0}),
                .addr({Q1, Q0}), .out(D1));

    // 两个 D 触发器
    dffre d0(.d(D0), .en(1'b1), .r(reset), .clk(clk), .q(Q0));
    dffre d1(.d(D1), .en(1'b1), .r(reset), .clk(clk), .q(Q1));

    assign w1 = (~Q1)&&Q0;
    assign w2 = Q1&&(~Q0);
    assign w3 = Q1&&Q0;

    // 输出信号
    assign load_ready = w1&&hit&&ld;
    assign write_l1 = w1&&hit&&st;
    assign write_l2 = (w1&&((miss&&ld&&dirty)||(miss&&st)))||(w2&&(~WB_ack));
    assign read_l2 = (w1&&miss&&ld&&(~dirty))||(w2&&ld&&WB_ack)||(w3&&(~l2_ack));

endmodule
```

(3) 测试代码

```
`timescale 1ns/1ps

module CacheController_tb;
    parameter dely = 10;

    // inputs
    reg clk, reset;
    reg ld, st;             // 读/写指令，高电平有效
    reg [31:0] addr;        // 输入地址
    reg [20:0] tag0_loaded; // 组中第一个块的 tag
    reg [20:0] tag1_loaded; // 组中第二个块的 tag
    reg valid0, valid1;     // 组中两块的 valid 位
    reg dirty;              // 是否为脏块
    reg l2_ack;

    // outputs
    wire hit, miss;         // 缓存命中/未命中，高电平有效
    wire load_ready;        // 可从 L1D 读取数据到 CPU，高电平有效
    wire write_l1;          // 可将数据从 CPU 写入 L1D，高电平有效
    wire read_l2;           // 可从 L2 读取数据到 L1D，高电平有效
```

```
    wire write_l2;          // 可将数据从 L1D 写回缓冲/L2，高电平有效

    // Instantiate the Unit Under Test(UUT)
    // CacheController_Logic inst(
    CacheController_FSM inst(
        // inputs
        .clk(clk),
        .reset(reset),
        .ld(ld),
        .st(st),
        .addr(addr),
        .tag0_loaded(tag0_loaded),
        .tag1_loaded(tag1_loaded),
        .valid0(valid0),
        .valid1(valid1),
        .dirty(dirty),
        .l2_ack(l2_ack),

        // outputs
        .hit(hit),
        .miss(miss),
        .load_ready(load_ready),
        .write_l1(write_l1),
        .read_l2(read_l2),
        .write_l2(write_l2)
    );

    // clk
    initial begin
        clk = 0;
        forever #(dely/2) clk = ~clk;
    end

    // reset
    initial begin
        reset = 1;
        #(dely) reset = 0;    // 初始化完成，此时控制器处于 Idle
    end

    // inputs
    initial begin
        ld = 0; st = 0;
        addr = {21'h1, 6'h0, 5'h0};       // 地址
        tag0_loaded = 21'h1; valid0 = 0;  // 第一个块
```

```
        tag1_loaded = 21'h3; valid1 = 0;  // 第二个块
        dirty = 0; l2_ack = 0;

        // 读未命中、非脏块，假设新分配的行为第一个块
        #(dely*3) ld = 1;
        #(dely)               // 发出读取指令，控制器转移到 CompareTag
                              // 第一个块 valid 位为 0，第二个块 tag 不匹配，故未命中
        #(dely*8) l2_ack = 1; // 进入 ReadData 状态，等待 8 个周期从 L2 读取数据到 L1D
                  valid0 = 1;
        #(dely)   l2_ack = 0; // 数据读取完毕，valid 位置为 1，回到 CompareTag
        #(dely)   ld = 0;     // 输出 load_ready=1，回到 Idle

        // 读未命中、脏块，假设新分配的行为第二个块
        #(dely)   addr = {21'h3, 6'h0, 5'h0}; // 修改地址，第二个块
                  dirty = 1;  // 第二个块改为脏块
        #(dely)   ld = 1;
        #(dely)               // 发出读取指令，控制器转移到 CompareTag
                              // 第一个块 tag 不匹配，第二个块 valid 位为 0，故未命中
        #(dely*8)             // 进入 WriteBuffer 状态，等待 8 个周期将数据写进缓冲
        #(dely*8) l2_ack = 1; // 进入 ReadData 状态，等待 8 个周期从 L2 读取数据到 L1D
                  valid1 = 1;
        #(dely)   l2_ack = 0; // 数据读取完毕，valid 位置为 1，回到 CompareTag
        #(dely)   ld = 0;     // 输出 load_ready=1，回到 Idle

        // 读命中
        #(dely)   addr = {21'h1, 6'h0, 5'h0}; // 修改地址，第一个块
        #(dely)   ld = 1;
        #(dely)               // 发出读取指令，控制器转移到 CompareTag
                              // 第一个块 tag 匹配，且 valid 位为 1，故命中
        #(dely)   ld = 0;     // 输出 load_ready = 1，回到 Idle

        // 写未命中
        #(dely)   valid0 = 0; // 第一个块 valid 位改为 0
        #(dely)   st = 1;
        #(dely)               // 发出写入指令，控制器转移到 CompareTag
                              // 第一个块 valid 位为 0，第二个块 tag 不匹配，故未命中
        #(dely*8)             // 进入 WriteBuffer 状态，等待 8 个周期将数据写入缓冲
        #(dely)   st = 0;     // 数据写入完毕，回到 Idle

        // 写命中
        #(dely)   addr = {21'h3, 6'h0, 5'h0}; // 修改地址，第二个块
        #(dely)   st = 1;
        #(dely)               // 发出写入指令，控制器转移到 CompareTag
                              // 第二个块 tag 匹配，且 valid 位为 1，故命中
```

```
        #(dely)    st = 0;      // 输出 write_l1 = 1，回到 Idle

        #(dely*2) $stop;
    end
endmodule
```

五、仿真结果

如测试代码的注释所描述的流程，我对控制器可能面对的五种都进行了测试，其状态转换的整个过程应为：Idle->CompareTag->ReadData->CompareTag->Idle->CompareTag->WriteBuffer->ReadData->CompareTag->Idle->CompareTag->Idle->CompareTag->WriteBuffer->Idle->CompareTag->Idle，各状态下的输出将在以下具体分析。

① 有限状态机实现

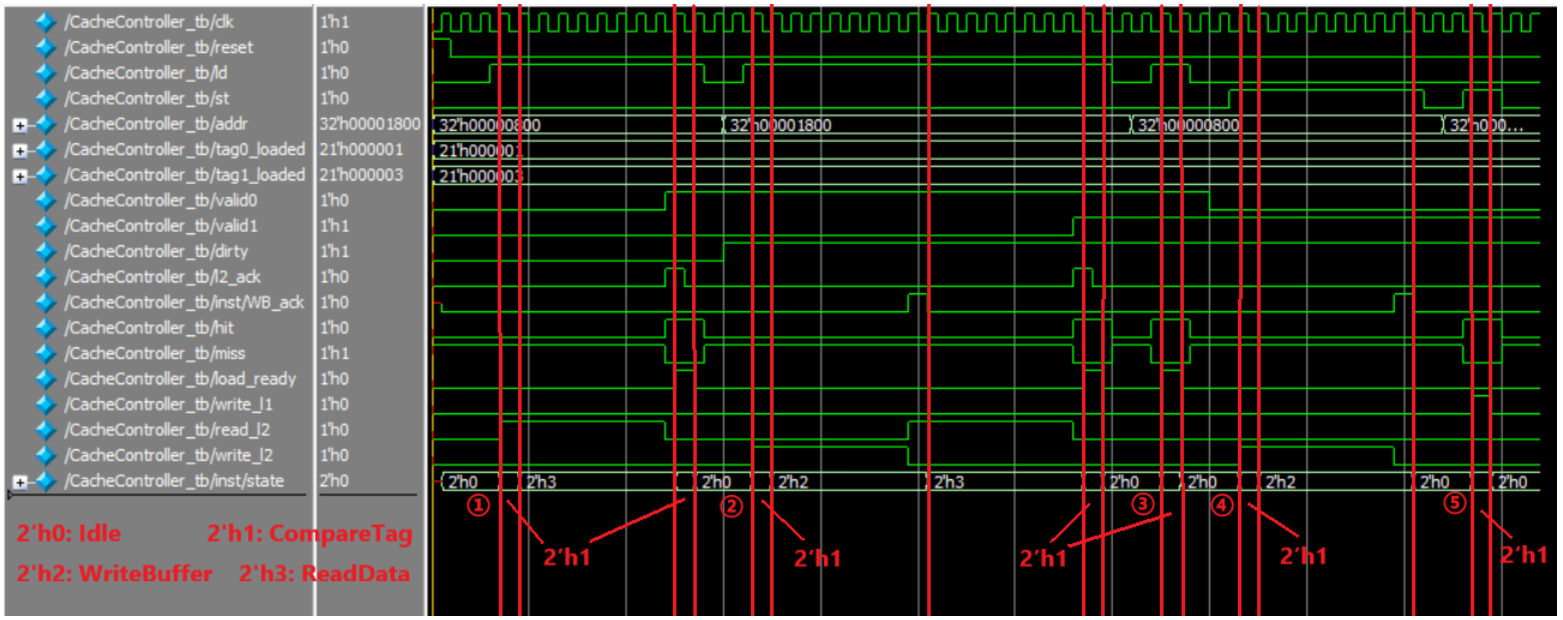

图 5.1 有限状态机实现仿真结果

分析：如图 5.1，控制器状态转移过程与理论分析相同。

在图中①标识的过程中，控制器在 Idle 态接收到 ld 指令，进入 CompareTag 态。由于输入信号设置为未命中，且假设新分配的行为第一个块，其非脏块（dirty=0），控制器将进入 ReadData 态，从 L2 读取数据到 L1D。因此，在 CompareTag 态，控制器输出信号 read_l2=1；开始读取数据后，控制器将在 ReadData 停留 8 个时钟周期，期间保持读允许 read_l2=1，直到输入信号 l2_ack=1，说明数据读取完毕，valid 位被置为 1， L2 读允许信号 read_l2 清 0，控制器回到 CompareTag 态。显然，修改 cache

line 为有效后，缓存命中，CPU 能够从 L1D 读取数据，因此输出信号 load_ready=1。当下一个时钟上升沿到来时，控制器进入 Idle 态，所有输出控制信号清 0。

在图中②标识的过程中，控制器在 Idle 态接收到 ld 指令，进入 CompareTag 态。由于输入信号设为未命中，且假设新分配的行为第一个块，其是脏块（dirty=1），当下一个时钟上升沿到来时，控制器将进入 WriteBuffer 态，将脏块中的内容写到缓冲/L2 中，因此，在 CompareTag 态，控制器输出信号 write_l2=1；开始写入数据后，控制器会在 WriteBuffer 态停留 8 个时钟周期，保持 write_l2=1，与输出波形一致。当计数器计满 8 个周期，WB_ack=1 表示数据写入完毕，此时写缓冲允许信号 write_l2 清 0，控制器将进入 ReadData 状态，从 L2 中读取数据到该 cache line，因此输出控制信号 read_l2=1。在 ReadData 停留 8 个时钟周期后，输入信号 l2_ack=1，数据读取完毕，该行 valid 位置为 1，read_l2 清零，控制器回到 CompareTag 状态。与①类似，修改 cache line 的 valid 位后，缓存命中，CPU 可从 L1D 读取数据，输出信号 load_ready=1。当下一个时钟上升沿到来时，控制器进入 Idle 态，所有输出控制信号清 0。

在图中③标识的过程中，控制器在 Idle 态接收到 ld 指令，进入 CompareTag 态。由于输入信号设置为命中，CPU 可直接从 L1D 读取数据，故输出控制信号 load_ready=1。当下一个时钟上升沿到来时，控制器回到 Idle 态，所有输出控制信号清 0。

在图中④标识的过程中，控制器在 Idle 态接收到 st 指令，进入 CompareTag 状态。由于输入信号设置为未命中，数据将绕过 L1D，直接写入 L2，故输出信号 write_l2=1。进入 WriteBuffer 态后，需停留 8 个周期等待数据写入，当 WB_ack=1，表明写入完成，L2 写允许信号 write_l2 清零。之后，控制器回到 Idle 状态，所有输出控制信号清 0。

在图中⑤标识的过程中，控制器在 Idle 态接收到 st 指令，进入 CompareTag 状态。由于输入信号设置为命中，CPU 可直接将数据写入 L1D 该行，故输出控制信号 write_l1=1。当下一个时钟上升沿到来时，控制器回到 Idle 态，所有输出控制信号清 0。

综上，控制器的状态转换过程与测试文件所预测的过程相同，且各输出控制信号波形符合预期，控制器设计正确。

② 逻辑电路实现

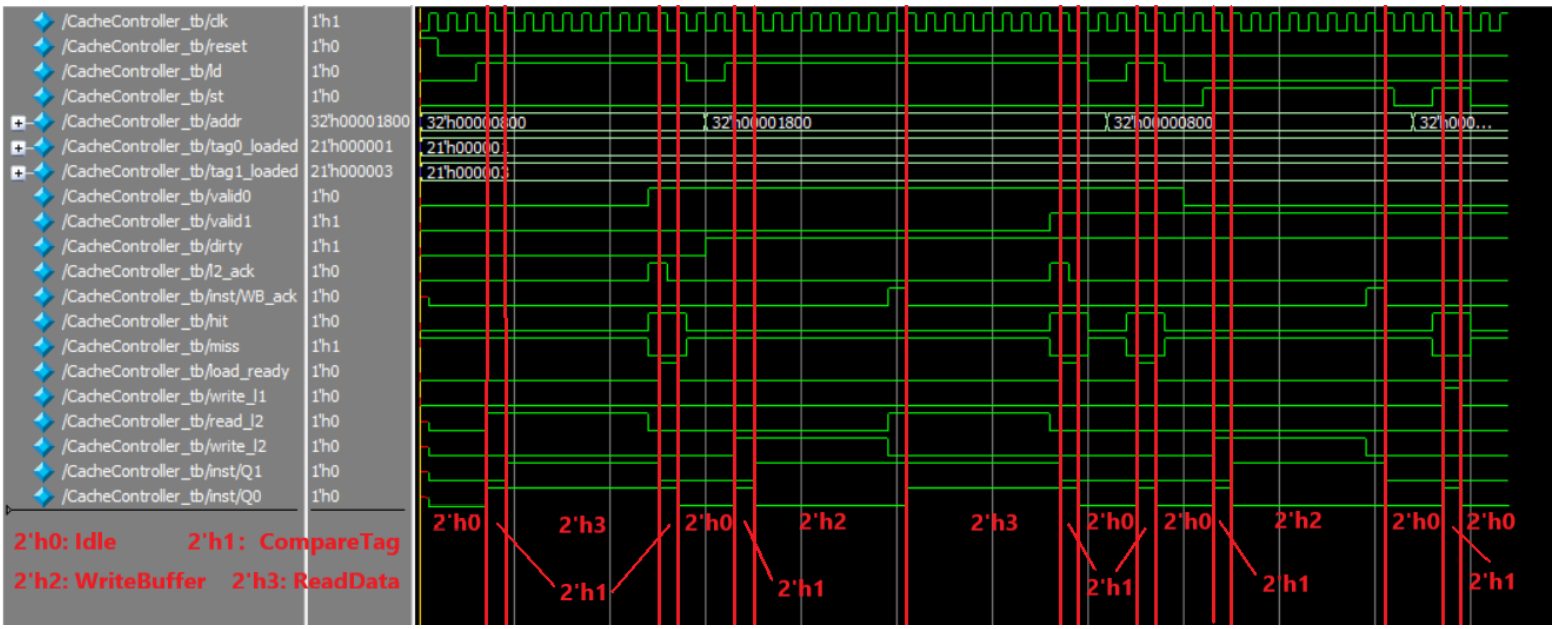

图 5.2 逻辑电路实现仿真结果

分析：如图 5.2 所示，逻辑电路实现下的输出控制信号、状态编码转移与有限状态机实现完全相同，故不再赘述，设计正确。

六、心得体会

此次实验，我们利用有限状态机完成了一个简单的缓存控制器，对缓存和 CPU、内存的交互流程进行了比较全面的熟悉，极大地加深了自己对缓存结构和机制的理解。当然，在实验的过程当中，我也遇到过一些问题，解决过程记录如下：

① 在查看测试输出时，发现当 addr 的高 21 位与 tag1_load 相等且 valid1=1 时，hit1 始终为 0。多次检查代码中 hit1 的赋值部分，都没有发现逻辑错误。最后查看变量的定义过程，发现 tag 虽已定义，但位数为 1 而非 21，改正后，hit1 的值输出正确。

② 在输出信号中，load_ready、write_l2 始终为低电平，与逻辑不符。通过分析状态转换，发现在读未命中、非脏块的条件下，控制器本应从 CompareTag 状态回到 Idel 状态，但在实际测试中却从 CompareTag 状态再次进入了 ReadData 状态。因此，我对测试代码的逻辑进行了梳理，发现是由于数据读取完毕后应当为 1 的 valid 位的位置不小心写错，导致在 CompareTag 态 hit 不为 1。修改后，输出信号恢复正常。

③ 在最初的电路设计中，我将两路 block 的 dirty 位分别设为 dirty0 和 dirty1，并令 dirty=(hit0&&dirty0)||(hit1&&dirty1)。然而，在进行测试时，若输入为读

未命中、脏块，控制器应由 CompareTag 进入 WriteBuffer 态，实际的状态转移却是由 CompareTag 进入到了 ReadData 态。通过比较各信号与状态的转移条件表达式，我发现此问题是由于 dirty 始终为 0 导致，进而定位到代码中的 dirty 赋值语句，发现其存在逻辑错误。显然，当未命中时，dirty 将永远处于低电平。基于此，我又重新回顾了一遍 dirty 在设计中的作用，发现此方案并不能满足设计要求，必须由 LRU 位确定替换掉哪一行后，才能够判断 dirty 是否为 1。因此，结合项目要求，我将 dirty 修改为输入信号，令其直接表示该行是否为脏块，从而解决了此逻辑冲突。

④ 在修改完上述问题后，控制器能够进入 WriteBuffer 状态，但仿真时出现了迭代超时的报错。究其原因，我发现在代码设计中出现了无限循环的情况：WB_ack 用来控制了 write_l2 的赋值，而 write_l2 作为了计数器的使能输入，导致组合电路的输出又作为了组合电路的输入，最终使得仿真器计算超出迭代限制。通过将计数器的输出修改为寄存器类型，此问题被成功解决。

虽然为了简单起见，此项目忽略了许多因素，如数据替换、缓冲容量等等，但其难度和复杂度仍是一个不小的挑战，只有全面认识、掌握了缓存对 CPU 读/写指令的响应逻辑和流程，才能够确保设计的正确。从输入/输出、内部信号和状态的确定，到状态转换图的设计，再到电路的绘制和代码的实现，我们从无到有，根据基本的缓存机制和参数确定控制器的响应方式，以此为基点完成思维的扩张，也不失为一次良好的学习体验。除此之外，在编写 test bench 的过程中，我也深深体会到了测试文件的重要性：其并非简单的设计一些输入条件，而是要综合考虑到每种情况下系统的变化和所需的输入信号，如 l2_ack、valid0 等的赋值。也正是因此，在对测试文件进行编写和修改的过程中，我对控制器的状态变化流程也有了更好的把握，可谓相辅相成。

总的来说，此次实验不仅加深了我对缓存机制的理解，也回顾了控制器的设计方法，既遇到过困难，也因此锻炼了自己的创造力、收获了知识。看到最后的测试结果完全符合预期，我的心中难掩激动。虽然可能实际中的设计与我现在所实现的简单控制器存在有很大的差异，但我却借此过程加深了对相关知识的掌握，培养了自己的兴趣，受益匪浅。