(1) State machine design

1.State diagram

The state diagram of the state machine is shown in Figure 1.

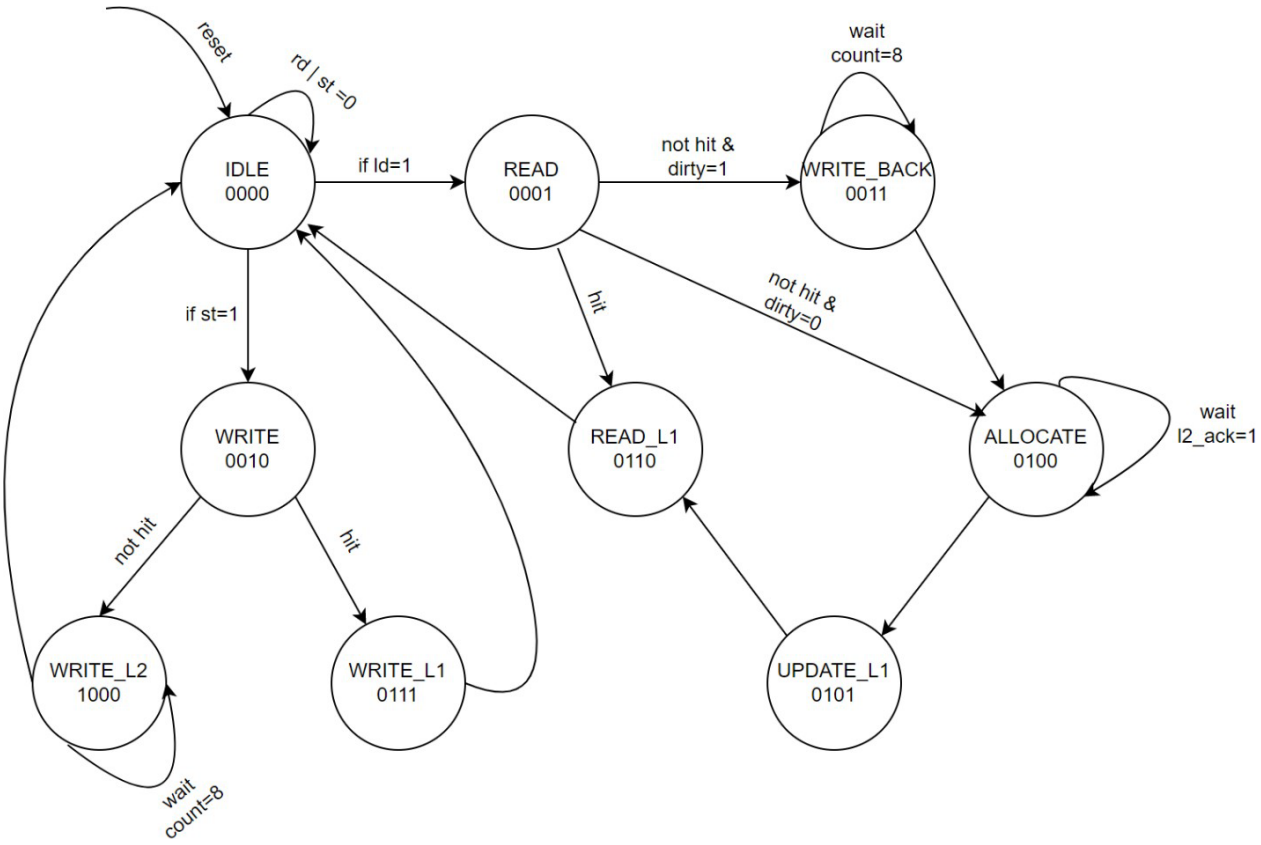


Figure 1 State Diagram

In the initial state, when the reset signal is set to 1, it jumps to the IDLE state, and when the rd and st signals are both 0, it remains in the state IDLE status.

In the IDLE state, jump from IDLE to READ state or when the input signal is rd=1 or st=1 WRITE status

state.

In the READ state, there are 3 jump conditions.

（1） When hit=1 corresponds to cache hit, it is transferred from the READ state to the READ_L1 state.

（2） When hit=0 & dirty=1 corresponds to cache miss, and the data in the block that needs to be replaced needs to be written back to L2, from the case READ jumps to the WRITE_BACK state.

（3） When hit=0 & dirty=0, the data in the data block corresponds to cache miss, and the data in the block that needs to be replaced does not need to be written back to L2, from READ Jump to the ALLOCATE state.

In the WRITE_BACK state, implement the write-back function of the cache, which writes the data in L1 back to L2, thus achieving synchronization, because L1D-> L2 has a bit width of 32-bit The line size of L1 is 32-bytes, so it needs to be transmitted 8 times, so a counter is used to count, And determine the transition condition, when count=8, jump to the ALLOCATE state.

In the ALLOCATE state, two operations are implemented. The first thing you need to read from L2 is to determine the block of data to be replaced. When
l2_ack signal is 1, the data in L2 has arrived and moved to the UPDATE_L1 state.

In the ALLOCATE state, data that needs to be written to L1 is written to L1 and then transferred to the READ_L1 state. In the READ_L1 state, the required data is read from the corresponding location in L1 and then transferred to the IDLE state. In the WRITE state, there are 2 jump cases.

（1） When hit=1 corresponds to the cache hit, it is transferred from the WRITE state to the WRITE_L1 state.

（2） When hit=0 corresponds to cache miss, it is transferred from the WRITE state to the WRITE_L2 state.

In the WRITE_L1 state, the data that needs to be written is written to the corresponding location of L1, and dirty position 1 is then transferred to IDLE
state.

In the WRITE_L2 state, because write allocation is not used, the data is written directly to the corresponding location in L2. Since L1D-> L2 has a bit width of 32-bit and L1 has a line size of 32-bytes, So 8 transfers are required, so a counter is used to count, and the transition condition is determined, and when count=8, jumps to the IDLE state.

2.State transition table

Because there are too many input signals, the state transition table is difficult to implement, so instead of Table 1, the table shows the relationship between the substate and the input signal in a given state. thereinto

```
hit = hit1 | hit2 = valid1 & (tag_loaded1 == addr[31:11]) | valid1 &
(tag_loaded1 == addr[31:11]);  countend = （count==6）
dirtyget = (!ref[1] & valid1 & dirty1) |  (!ref[0] & valid2 & dirty2)
```

Table 1 The secondary state determined
by the present state, the input signal

Status quo	input						Sub morp hism
	ld	st	hit	countend	dirtyget	l2_ack	
IDLE(0000)	0	0	x	x	x	x	IDLE(0000)
IDLE(0000)	1	0	x	x	x	x	READ(0001)
IDLE(0000)	0	1	x	x	x	x	WRITE(0010)
READ(0001)	x	x	1	x	x	x	READ_L1(0110)
READ(0001)	x	x	0	x	1	X	WRITE_BACK(0011)

READ(0001)	x	x	0	x	0	x	ALLOCATE(0100)
WRITE_BACK(0011)	x	x	x	0	x	x	WRITE_BACK(0011)
WRITE_BACK(0011)	x	x	x	1	x	x	ALLOCATE(0100)
ALLOCATE(0100)	x	x	x	x	x	0	ALLOCATE(0100)
ALLOCATE(0100)	x	x	x	x	x	1	UPDATE_L1(0101)
UPDATE_L1	x	x	x	x	x	x	READ_L1(0110)

READ_L1	x	x	x	x	x	x	IDLE(0000)
WRITE(0010)	x	x	1	x	x	x	WRITE_L1(0111)
WRITE(0010)	x	x	0	x	x	x	WRITE_L2(1000)
WRITE_L1(0111)	x	x	x	x	x	x	IDLE(0000)
WRITE_L2(1000)	x	x	x	0	x	x	WRITE_L2(1000)
WRITE_L2(1000)	x	x	x	1	x	x	IDLE(0000)

3.Controller flowchart

The flowchart of the controller working is shown in Figure 2. After receiving the request, first determine whether it is a Read operation or a Write operation. If it is a Read operation, determine whether it is hit. If hit, the required data is returned directly from L1. If missed, at L1

If the dirty bit of the block is 1, the data in the block is first written back to L2 and dirty is set to 0. The required data is then read from L2 and finally returned.

If it is a Write operation, determine whether it is hit. If hit, the data is written directly to the corresponding location in L1 and dirty is set to 1. If it is missed, the data is written directly to the corresponding location in L2.

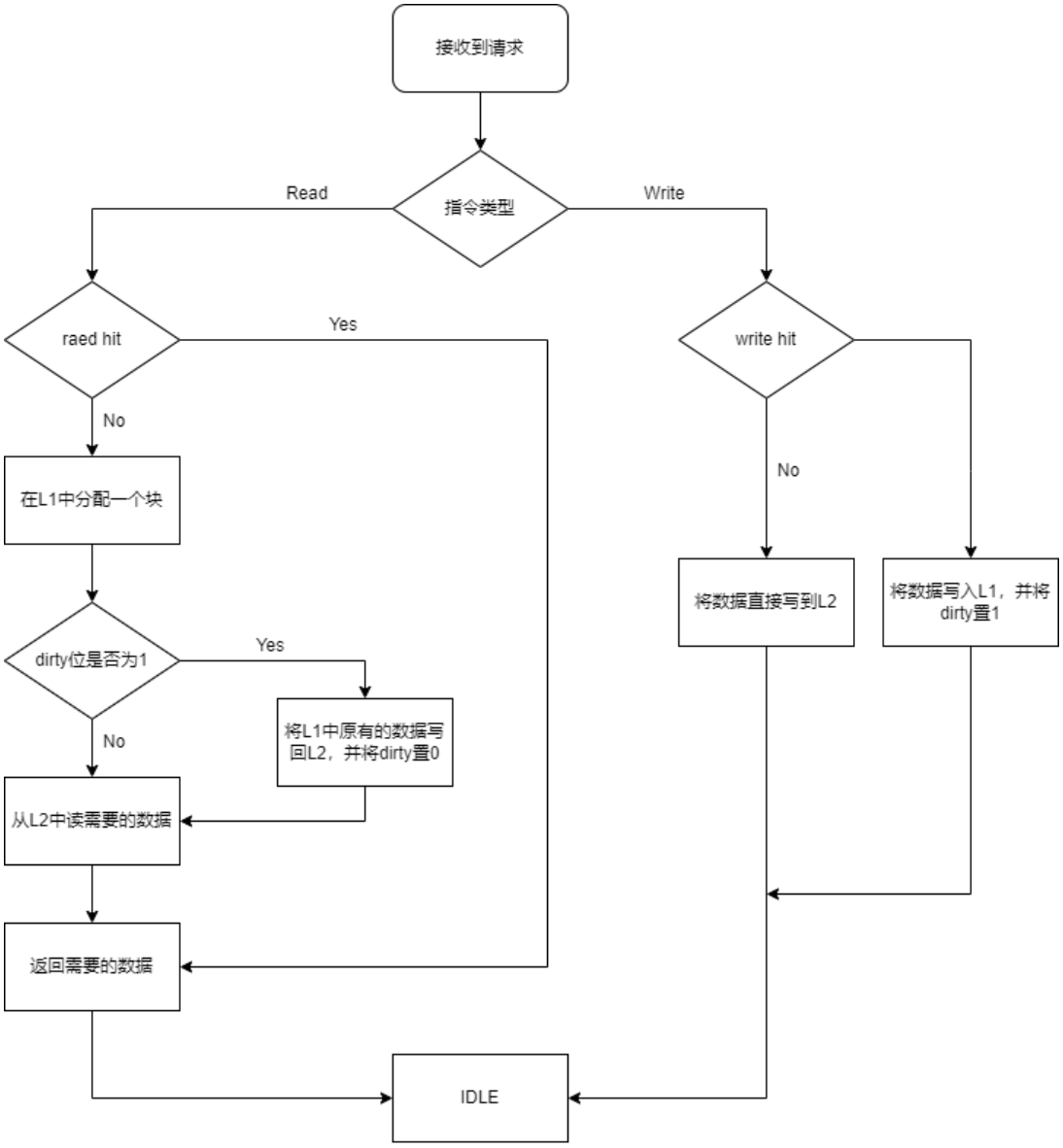


Figure 2 Controller Workflow Diagram

(2) Circuit design

The state machine circuit is shown in Figure 3. It is divided into three parts.

The secondary logic part first passes the input signal through the combinatorial logic circuit to obtain the input signal of 16 selectors, and determines the output of the selector according to the current state, so as to obtain the secondary state.

The status register consists of a D trigger. The input is secondary, the output is present, and there is a reset signal input. The output has a total of 4 bits of signal, which is not drawn to save space.

The output logic section, where the output is determined by both the present state and the current input

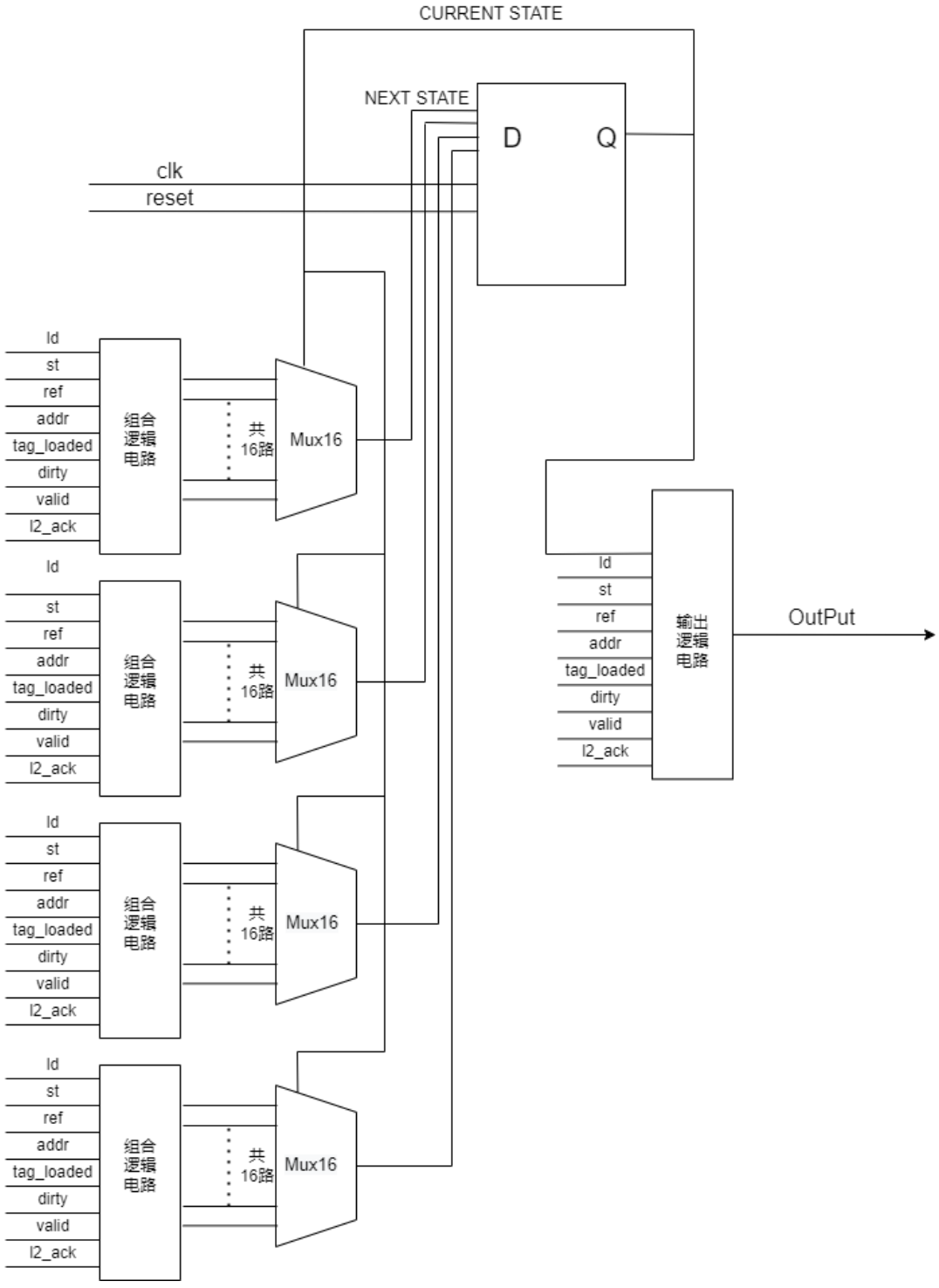


Figure 3 State machine circuit diagram

(c) Verilog implementation and simulation process

1.Verilog implementation

This design uses the standard three-segment notation method to implement the state machine. Because Cache is connected in two groups, the input and output signals are defined as follows.

```
input   clk,
input   reset,
input   ld, st,
input   [1:0]   ref,
input   [31:0]  addr,
input   [20:0]  tag_loaded1, tag_loaded2,
input   valid1, dirty1, valid2, dirty2,
input   l2_ack,
output  reg     hit, miss,
output  reg     tag_en1,tag_en2,valid_en1,valid_en2,dirty_en1,dirty_en2,
                //tag、valid、dirty使能信号
output  reg     load_ready,
output  reg     [1:0] write_l1,
output  reg     read_l2, write_l2,
output  reg     [1:0] ref_new,       //更改后的reference
output  reg     ref_en,
output  reg     dirty,
output  reg [3:0]   state,
output  reg [3:0]   next_state,
output  reg [3:0]   count,
output  reg count_en                //计数器使能
```

State definitions and intermediate signal definitions

```
localparam      IDLE            = 0,
                READ            = 1,
                WRITE           = 2,
                WRITEBACK_L2    = 3,
                ALLOCATE        = 4,
                UPDATE_L1       = 5,
                READ_L1         = 6,
                WRITE_L1        = 7,
                WRITE_L2        = 8;

wire        hit1, hit2;
assign hit1 = valid1 & (tag_loaded1 == addr[31:11]);      //block0 hit
assign hit2 = valid2 & (tag_loaded2 == addr[31:11]);      //block1 hit
```

The minor logic section is implemented using always@ (*) combined logic blocks and the case statement, and only the parts are shown below

```
always@(*)                                  //状态转移逻辑
begin
    case(state)
            IDLE:
                    if(ld)
                    begin
                        next_state = READ;
                    end
                    else if(st)
                    begin
                        next_state = WRITE;
                    end
                    else
                    begin
                        next_state = IDLE;
                    end
```

Status registers are implemented using always@ (posedge) timing logic blocks

```
always@(posedge clk or negedge reset)  //状态寄存器
begin
    if(reset) begin  //复位
    state <= IDLE;
    end
    else begin
    state <= next_state;
    end
end
```

The output logic section is implemented using always@ (*) combining logic blocks and case statements, and only the following sections are shown

```
always@(posedge clk or negedge reset)  //输出逻辑
begin
    if(reset) begin
            hit         <= 0;
            miss        <= 0;
            tag_en1     <= 0;
            tag_en2     <= 0;
            valid_en1   <= 0;
            valid_en2   <= 0;
            dirty_en1   <= 0;
            dirty_en2   <= 0;
            load_ready  <= 0;
            write_l1    <= 0;
            read_l2     <= 0;
            write_l2    <= 0;
            ref_en      <= 0;
            ref_new     <= 0;
            count_en    <= 0;
            dirty       <= 0;
    end
```

Counter modules are implemented using always@ (posedge) timing logic blocks

```
always@(posedge clk)    //计数器模块
begin
    if(count_en == 1)
    begin
        count <= count + 1;
    end
    else
    begin
        count <= 0;
    end
end
```

2.Simulation process

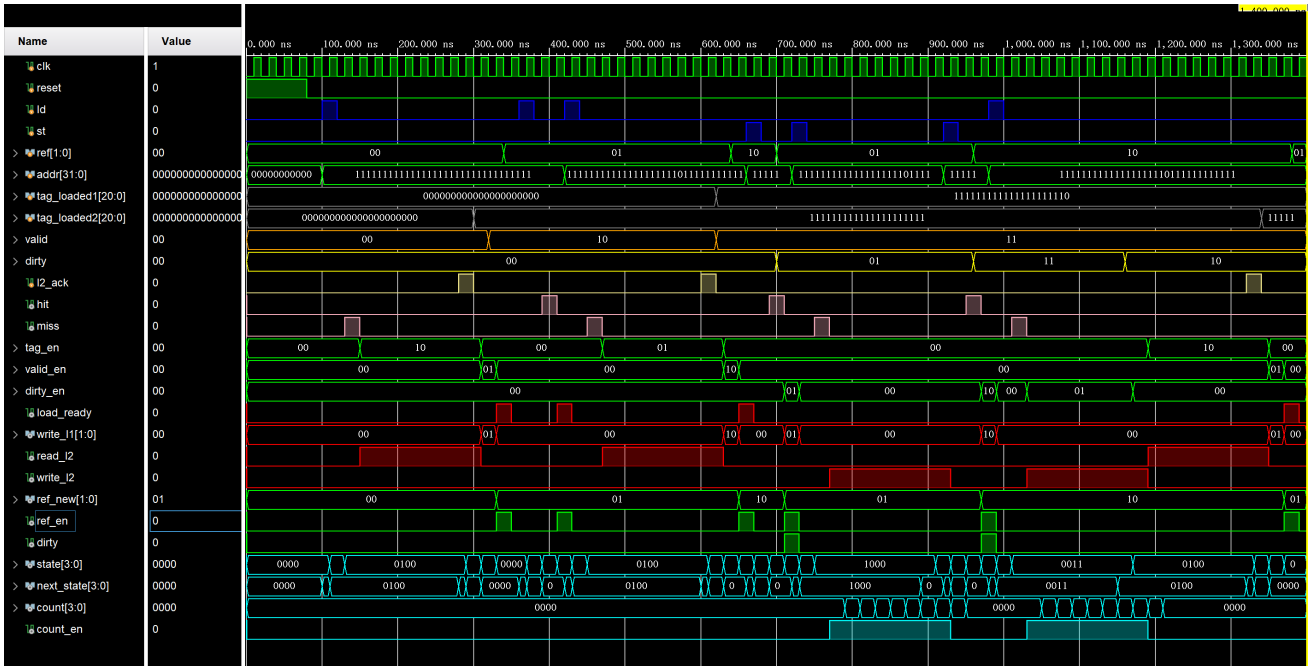
In simulation, write testbench to simulate the following process

Because during the simulation, the addresses in the accessed active memory are 11111111111111111111_11111111111 , respectively 11111111111111111110 11111111111 and 11111111111111111101 11111111111, the three pieces of data are in the cache , so only tag bits (i.e. 21 bits higher) are used to represent their addresses

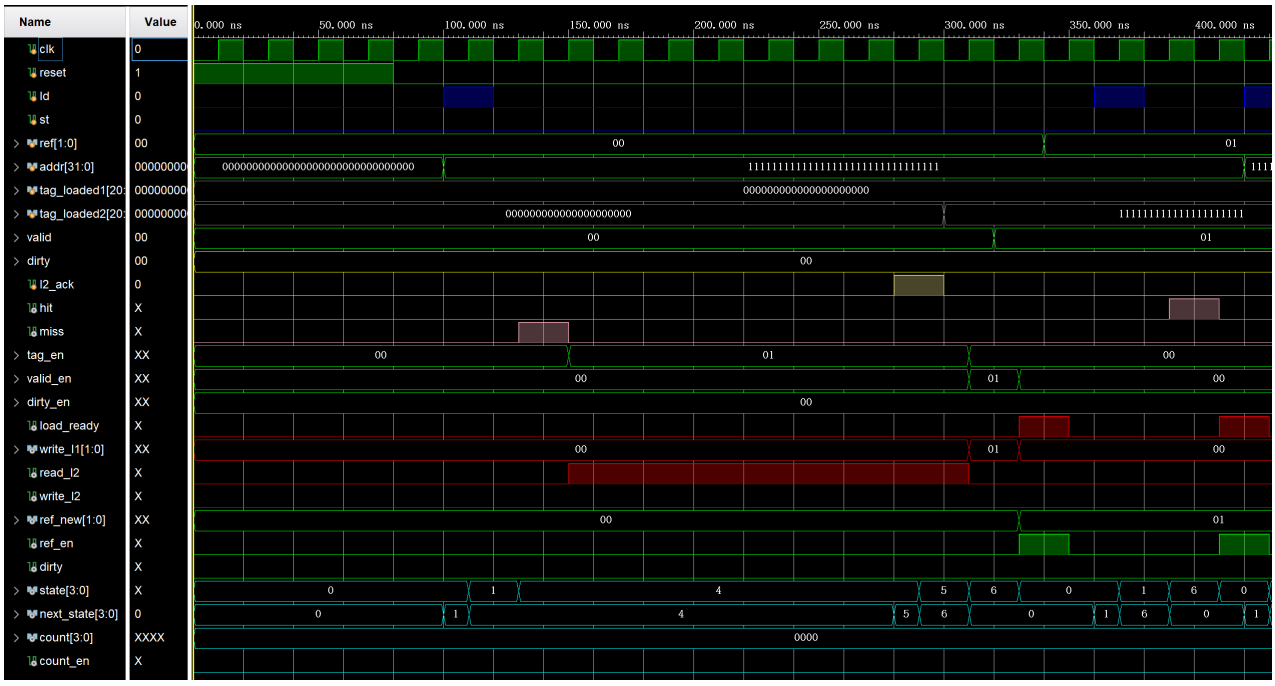
Table 2 Values for related data in the simulation process

The address of the main memory block accessed	Hit or Miss	The contents of the cache block after access	
		Group 111111	Group 111111
Read 11111111111111111111	Miss		Memory[11111111111111111111], dirty=0，reference=1
Read 11111111111111111111	Hit		Memory[11111111111111111111], dirty=0，reference=1
Read 11111111111111111110	Miss	Memory[11111111111111111110], dirty=0，reference=1	Memory[11111111111111111111], dirty=0，reference=0
Write 11111111111111111111	Hit	Memory[11111111111111111110], dirty=0，reference=0	Memory[11111111111111111111], dirty=1，reference=1
Write 11111111111111111101	Miss	Memory[11111111111111111110], dirty=0，reference=0	Memory[11111111111111111111], dirty=1，reference=1
Write 11111111111111111110	Hit	Memory[11111111111111111110], dirty=1，reference=1	Memory[11111111111111111111], dirty=1，reference=0
Read 11111111111111111101	Miss	Memory[11111111111111111110], dirty=1，reference=0	Memory[11111111111111111101], dirty=0，reference=1

The complete simulation results are as follows



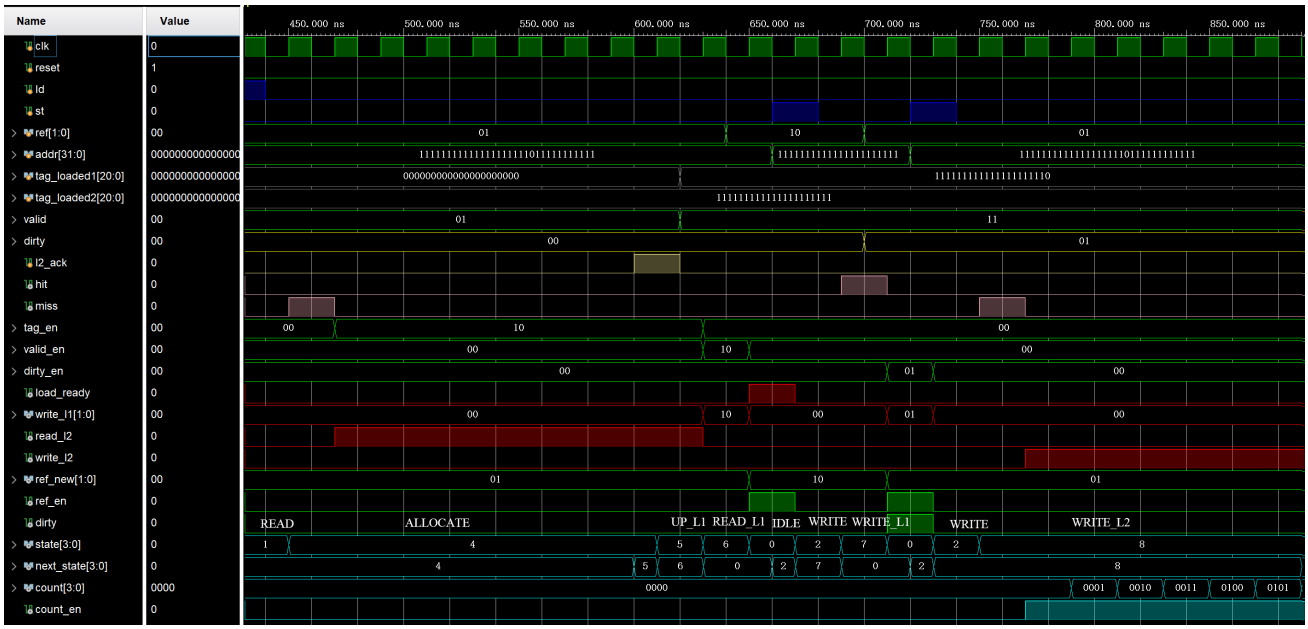
Simulation results analysis:



The controller is initialized first, the reset signal is valid, and you can see that it is in the IDLE state.

Later, when the ld signal is 1, it is transferred to the READ state, and since there is no data in L1 at this time , you can see that the output miss signal is 1, hopping Go to ALLLOCATE status. The ALLOCATE status is that the data is allocated in L1 blocks and waiting for the data in L2 to be read in, and according to the LRU rules, you can see that the tag_en is at this time 01。When the l2_ack signal is 1, the data in L2 has been fully read in, jumps to the UPDATE_L1 state, and you can see that the write_l1 later and valid_en signals are set to 01. After the update is complete, jump to the READ_L1 state, read the data from L1, and you can see that the load_ready signal is valid and the ref_en signal is valid, ref_ new output 01. Finally, jump back to the IDLE state.

Later, when the ld signal is 1, it is transferred to the READ state, and the input address is consistent with the previous ld, and L1 hits, and the output hit can be seen The signal is 1. After jumping to the READ_L1 state, reading the data from L1, you can see that the load_ready signal is valid, and the ref_en signal is valid, ref_new Output 01. Finally, jump back to the IDLE state.



Then when the ld signal is 1, it is transferred to the READ state, and the addr=32'b11111111111111111110_11111111111 at this time. Does not exist in L1, you can see that the output miss signal is 1, jumping to the ALLOCATE state. The ALLOCATE status is that the data is allocated in L1 in chunks and waiting for data to be read into L2, at which point you can see that the tag_en is 10。When the l2_ack signal is 1, the data in L2 has been fully read in, jumps to the UPDATE_L1 state, and you can see that the write_l1 later And valid_en signals are set to 10. After the update is complete, jump to the READ_L1 state, read the data from L1, and you can see that the load_ready signal is valid and the ref_en signal is valid ref_new Output 10. Finally, jump back to the IDLE state.

Then when the st signal is 1, it moves to the WRITE state, where addr=32'b11111111111111111111_11111111111, Exists in L1 and you can see that the output hit signal is 1, jumping to the WRITE_L1 state. After you can see that the write_l1 signal is set to 01 and the dirty_en signal is set to 01, according to the LRU rules ref_en valid, ref_new output 01. Finally, jump back to the IDLE state. Then when the st signal is 1, it moves to the WRITE state, where addr=32' b11111111111111111101_11111111111, Does not exist in L1, you can see the output miss=1, jump to the WRITE_L2 state. You can see that write_l2 1, wait 8 cycles, and jump back to the IDLE state.

Then when the st signal is 1, it moves to the WRITE state, where addr=32'b11111111111111111110_11111111111, at L1 , you can see that the output hit signal is 1, jumping to the WRITE_L1 state. You can see that after write_l1 signal is set to 10 and dirty_en signal is set to 10, according to the LRU rules ref_en is valid, ref_new output 10. Finally, jump back to the IDLE state. Then when the ld signal is 1, it moves to the READ state, where addr=32'b11111111111111111101_11111111111 , Does not exist in L1, and you can see that the output miss signal is 1. According to the LRU rules, the tag=11111111111111111111 of the replaced block. The block is dirty=1, so it needs to be written back to jump to the WRITE_BACK state. You can see that the write_l2 signal is 1, the dirty_en signal is 01, and dirty is 0, while writing back, put dirty to 0. WRITE_BACK lasts for 8 cycles, after which it jumps to the ALLOCATE state, where the ALLOCATE status is Data In Blocks are allocated in L1 and wait for data to be read into L2, and according to LRU rules, you can see that the tag_en is 01. When the l2_ack signal is 1, the data in L2 has been completely read in, jumping to the UPDATE_L1 state, and you can see that the next write_ Both the l1 and valid_en signals are set to 01. After the update is complete, jump to the READ_L1 state, read the data from L1, and you can see that the load_ready signal is valid, and the ref_en signal is valid, root According to LRU rules, ref_new output 01. Finally, jump back to the IDLE state.

Fifth, problems, solutions and experiences

When writing state machine code, in order to reduce the amount of code, the state machine is written in a piece of time. However, due to the relatively large number of states involved, the one-stage state machine is prone to problems, and it is not conducive to the subsequent problem elimination, so the writing method of the three-stage state machine is changed later.

Since the counter is required for the WRITE_BACK and WRITE_L2 phases, the counter's enable signal count_en can only be turned on when the first time it enters the above state. Therefore, the counter counts only 6 times in total . The situation is not taken into account when the code is first written, resulting in an error in the state machine state.

In addition, because in the design requirements, there is no need to give the specific implementation of the data path, so for the input and output signals of the state machine, we need to increase or decrease on the existing signals, and in this process, we need to consider the read and write mode and storage structure of the cache, which brings us a lot of challenges.

At the same time, due to the case of reading miss, the allocation method is used , but for write miss, it is directly written back to L2. Therefore, in the state design, it is necessary

to divide the read and written tag comparison into two states. This adds to the state, but it brings convenience to subsequent designs.

The design of the cache controller not only gives us a deeper understanding of how the cache is composed and works, but also makes me aware of the complexity of the cache structure in the actual computer and its importance. At the same time, this design experiment also made me re-review The knowledge of digital circuits learned in the lower grades, especially the writing of finite state machines, has also played a good role in consolidating the knowledge of the past.

Or

de

ri

ng