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

实验名称：Lab4:pipelined CPU with cache

指导教师：何水兵 完成日期：2023 年 11 月 14 日

实验分工

姓名：蒋奕 3210103803

同组学生姓名：黄俊涛 3210101831

分工情况：

蒋奕：cmu 模块 + 设计测试仿真

cmu 采用状态机实现，最终通过仿真和上板子结果验证。

黄俊涛：cmu 模块 + 设计测试仿真

1 实验目的和要求

- Understand the principle of Cache Management Unit (CMU) and State Machine of CMU.
- Understand the principle of Cache Management Unit (CMU) and State Machine of CMU.
- Master verification methods of CMU and compare the performance of CPU when it has cache or not.
- Master the design methods of CMU and Integrate it to the CPU.

2 实验内容和原理

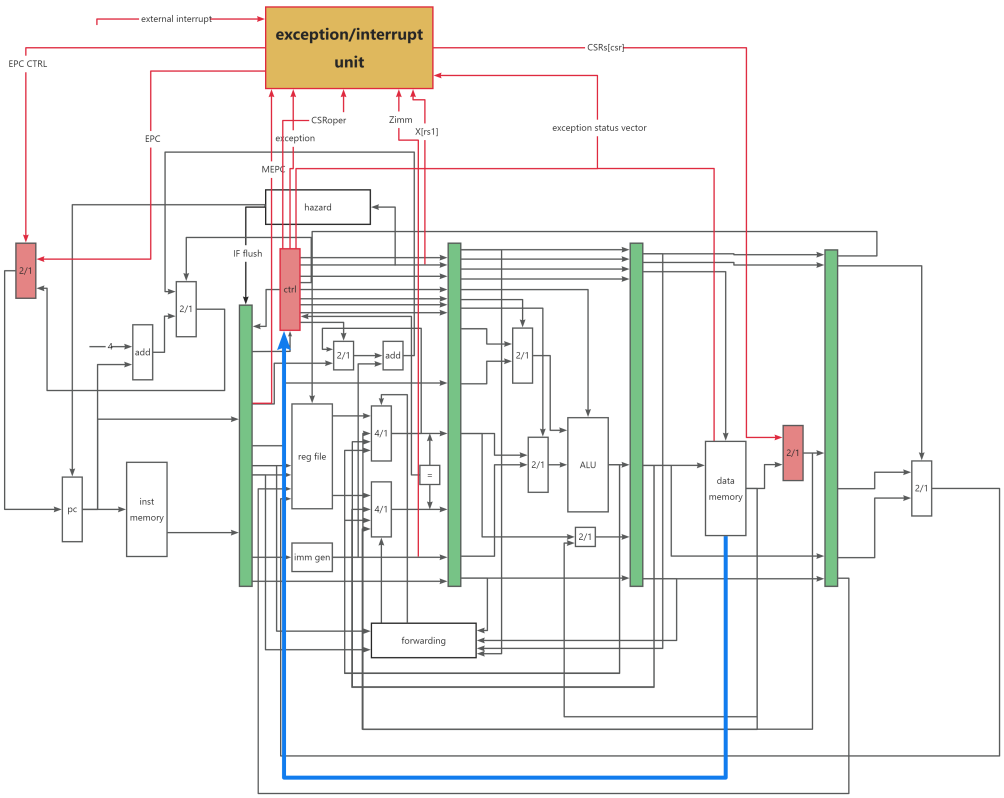
2.1 实验内容

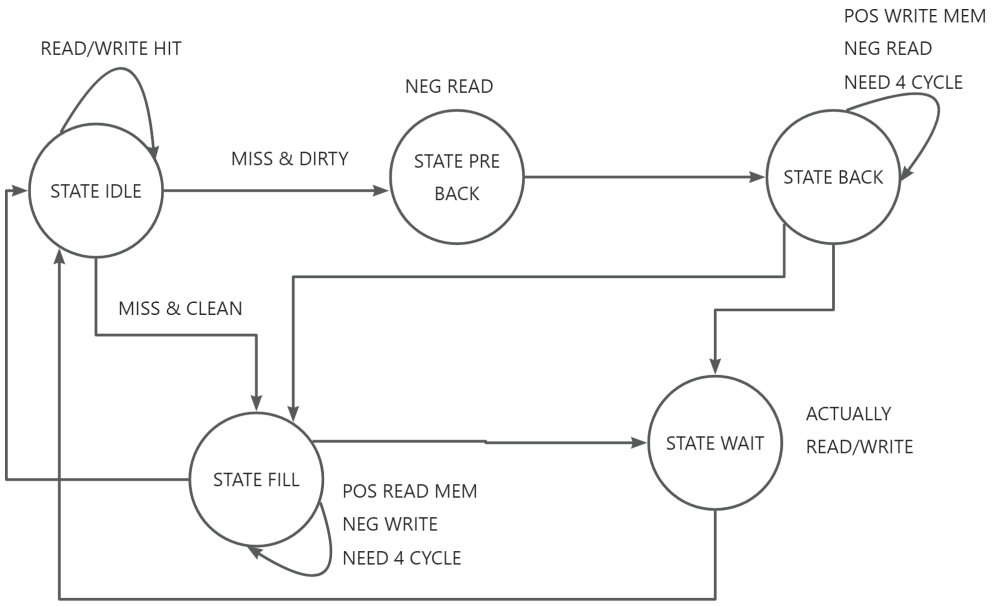
- Design of Cache Management Unit and integrate it to CPU.
- Observe and Analyze the Waveform of Simulation.
- Compare the performance of CPU when it has cache or not.

2.2 实验原理

- Structure of Circuit

  Compared to lab2, we add a line from data memory to ctrl unit. The curcuit is as follows:


external interrupt
exception/interrupt unit
EPC CTRL
CSRs[csr]
CSRoper
Zimm
X[rs1]
exception status vector
EPC
exception
MEPC
hazard
IF flush
ctrl
2/1
add
4
reg file
4/1
=
ALU
data memory
inst memory
pc
imm gen
forwarding


READ/WRITE HIT
NEG READ
POS WRITE MEM
NEG READ
NEED 4 CYCLE
STATE IDLE
MISS & DIRTY
STATE PRE BACK
STATE BACK
MISS & CLEAN
STATE FILL
STATE WAIT
ACTUALLY READ/WRITE
POS READ MEM
NEG WRITE
NEED 4 CYCLE

- Structure of State Machine

  The schematic is as above:

  Cache operations occur on the neg edge of the current state, while memory operations occur on the pos edge of the current state. Each state has meaning:

  1. **S_IDLE:** Idle state; no memory operations. Cache operations continue if there is a hit, keeping the system in this state.
  2. **S_PRE_BACK:** To facilitate writing back, perform a cache read operation.
  3. **S_BACK:** On the rising edge, write back the data from the previous state to memory. On the falling edge, read the data to be written back from the cache. This process continues until the entire cache line is written back. Since memory operations take 4 cycles to complete, wait for the memory acknowledgment signal before changing the state.
  4. **S_FILL:** On the rising edge, read data from memory. On the falling edge, write data to the cache. This process continues until the entire cache line is filled. Similar to S_BACK, wait for the memory acknowledgment signal.
  5. **S_WAIT:** Execute cache operations that couldn't be performed earlier due to a miss.

3 实验过程和数据记录及结果分析

3.1 实验过程

The details of cache can be seen in comments of the corresponding code.

- state IDLE

```
S_IDLE: begin
        if (en_r || en_w) begin
                if (cache_hit)
                        next_state = S_IDLE;
                else if (cache_valid && cache_dirty)
                        next_state = S_PRE_BACK;
                else
                        next_state = S_FILL;
        end
        next_word_count = 2'b00;
end
```

- state BACK

  In this lab, in order to simulate Memory being slower than Cache, a counter has been manually set up. The state transition occurs only every 4 clock cycles.

```
S_BACK: begin
        if (mem_ack_i &&
```

```
word_count == {ELEMENT_WORDS_WIDTH{1'b1}})
        next_state = S_FILL;
else
        next_state = S_BACK;
if (mem_ack_i)
        next_word_count = word_count + 2'b01;
else
        next_word_count = word_count;
end
```

- state FILL

  In this lab, in order to simulate Memory being slower than Cache, word count has been manually set up. The state transition occurs only every 4 clock cycles.

```
S_FILL: begin
        if (mem_ack_i &&
        word_count == {ELEMENT_WORDS_WIDTH{1'b1}})
                next_state = S_WAIT;
        else
                next_state = S_FILL;
        if (mem_ack_i)
                next_word_count = word_count + 2'b01;
        else
                next_word_count = word_count;
        end
```

- state WAIT

```
S_WAIT: begin
        next_state = S_IDLE;
        next_word_count = 2'b00;
end
```

For the control part of the cache, we should do as follows:

- S_IDLE and S_WAIT

  The cache operates normally. The `cache_store` signal is set to 1 because there is no need to transfer data from memory to the cache in these states.

```
S_IDLE, S_WAIT: begin
        cache_addr = addr_rw;
        cache_load = en_r;
        cache_edit = en_w;
        cache_store = 1'b0;
        cache_u_b_h_w = u_b_h_w;
```

```
            cache_din = data_w;
        end
```

- S_BACK, S_PRE_BACK

  Due to the need to write back dirty data from the cache to memory, CPU should not operate on the cache during this time. Therefore, we ought to set `cache_load`, `cache_edit`, and `cache_store` to 0. Additionally, no data needs to be loaded into the cache during this operation.

```
        S_BACK, S_PRE_BACK: begin
            cache_addr = {addr_rw[ADDR_BITS-1:BLOCK_WIDTH],
                    next_word_count,
                    {ELEMENT_WORDS_WIDTH{1'b0}}};
            cache_load = 1'b0;
            cache_edit = 1'b0;
            cache_store = 1'b0;
            cache_u_b_h_w = 3'b010;
            cache_din = 32'b0;
        end
```

- S_FILL

  In `S_FILL`, data is read from memory and placed into cache. To prevent unexpected issues like overwriting data, it's necessary to stop CPU from writing to cache or modifying data in cache. If `mem_ack_i` is 1, indicating that the cache has successfully read data from memory, the cache data can be written back to memory. Therefore, `cache_store` is set to 1. Since data is read from memory, `cache_din` is set to `mem_data_i`.

```
        S_FILL: begin
            cache_addr = {addr_rw[ADDR_BITS-1:BLOCK_WIDTH],
                     word_count,
                    {ELEMENT_WORDS_WIDTH{1'b0}}};
            cache_load = 1'b0;
            cache_edit = 1'b0;
            cache_store = mem_ack_i;
            cache_u_b_h_w = 3'b010;
            cache_din = mem_data_i;
        end
```

- stall signal

  In CMU unit, stall signal is used to pause the execution of instructions in the CPU. When there is communication between the cache and memory, stall signal needs to be set to 1. Therefore, except for the `S_IDLE` state, a pause in the execution of instructions is required in all other states.

```
        assign stall = (next_state != S_IDLE);
```

- Complete the cache.v file

```
module cmu (
        // CPU side
                input clk,
                input rst,
                input [31:0] addr_rw,
                input en_r,
                input en_w,
        input [2:0] u_b_h_w,
                input [31:0] data_w,
                output [31:0] data_r,
                output stall,

        // mem side
                output reg mem_cs_o = 0,
                output reg mem_we_o = 0,
                output reg [31:0] mem_addr_o = 0,
                input [31:0] mem_data_i,
                output [31:0] mem_data_o,
                input mem_ack_i,

        // debug info
        output [2:0] cmu_state
        );

    `include "addr_define.vh"

    reg [ADDR_BITS-1:0] cache_addr = 0;
    reg cache_load = 0;
    reg cache_store = 0;
    reg cache_edit = 0;
    reg [2:0] cache_u_b_h_w = 0;
    reg [WORD_BITS-1:0] cache_din = 0;
    wire cache_hit;
    wire [WORD_BITS-1:0] cache_dout;
    wire cache_valid;
    wire cache_dirty;
    wire [TAG_BITS-1:0] cache_tag;

    cache CACHE (
        .clk(~clk),
```

```
41          .rst(rst),
42          .addr(cache_addr),
43          .load(cache_load),
44          .store(cache_store),
45          .edit(cache_edit),
46          .invalid(1'b0),
47          .u_b_h_w(cache_u_b_h_w),
48          .din(cache_din),
49          .hit(cache_hit),
50          .dout(cache_dout),
51          .valid(cache_valid),
52          .dirty(cache_dirty),
53          .tag(cache_tag)
54      );
55
56      localparam
57          S_IDLE = 0,
58          S_PRE_BACK = 1,
59          S_BACK = 2,
60          S_FILL = 3,
61          S_WAIT = 4;
62
63      reg [2:0]state = 0;
64      reg [2:0]next_state = 0;
65      reg [ELEMENT_WORDS_WIDTH-1:0]word_count = 0;
66      reg [ELEMENT_WORDS_WIDTH-1:0]next_word_count = 0;
67      assign cmu_state = state;
68
69      always @ (posedge clk) begin
70          if (rst) begin
71              state <= S_IDLE;
72              word_count <= 2'b00;
73          end
74          else begin
75              state <= next_state;
76              word_count <= next_word_count;
77          end
78      end
79
80      // state ctrl
81      always @ (*) begin
82          if (rst) begin
```

```
83              next_state = S_IDLE;
84              next_word_count = 2'b00;
85          end
86          else begin
87              case (state)
88                  S_IDLE: begin
89                      if (en_r || en_w) begin
90                          if (cache_hit)
91                              next_state = S_IDLE;
92                          else if (cache_valid && cache_dirty)
93                              next_state = S_PRE_BACK;
94                          else
95                              next_state = S_FILL;
96                      end
97                      next_word_count = 2'b00;
98                  end
99
100                 S_PRE_BACK: begin
101                     next_state = S_BACK;
102                     next_word_count = 2'b00;
103                 end
104
105                 S_BACK: begin
106                     if (mem_ack_i && word_count ==
    ↪ {ELEMENT_WORDS_WIDTH{1'b1}})    // 2'b11 in default case
107                         next_state = S_FILL;
108                     else
109                         next_state = S_BACK;
110
111                     if (mem_ack_i)
112                         next_word_count = word_count + 2'b01;
113                     else
114                         next_word_count = word_count;
115                 end
116
117                 S_FILL: begin
118                     if (mem_ack_i && word_count ==
    ↪ {ELEMENT_WORDS_WIDTH{1'b1}})
119                         next_state = S_WAIT;
120                     else
121                         next_state = S_FILL;
122
```

```
123                     if (mem_ack_i)
124                         next_word_count = word_count + 2'b01;
125                     else
126                         next_word_count = word_count;
127                 end
128 
129                 S_WAIT: begin
130                     next_state = S_IDLE;
131                     next_word_count = 2'b00;
132                 end
133             endcase
134         end
135     end
136 
137     // cache ctrl
138     always @ (*) begin
139         case(state)
140             S_IDLE, S_WAIT: begin
141                 cache_addr = addr_rw;
142                 cache_load = en_r;
143                 cache_edit = en_w;
144                 cache_store = 1'b0;
145                 cache_u_b_h_w = u_b_h_w;
146                 cache_din = data_w;
147             end
148             S_BACK, S_PRE_BACK: begin
149                 cache_addr = {addr_rw[ADDR_BITS-1:BLOCK_WIDTH],
    ↪ next_word_count, {ELEMENT_WORDS_WIDTH{1'b0}}};
150                 cache_load = 1'b0;
151                 cache_edit = 1'b0;
152                 cache_store = 1'b0;
153                 cache_u_b_h_w = 3'b010;
154                 cache_din = 32'b0;
155             end
156             S_FILL: begin
157                 cache_addr = {addr_rw[ADDR_BITS-1:BLOCK_WIDTH],
    ↪ word_count, {ELEMENT_WORDS_WIDTH{1'b0}}};
158                 cache_load = 1'b0;
159                 cache_edit = 1'b0;
160                 cache_store = mem_ack_i;
161                 cache_u_b_h_w = 3'b010;
162                 cache_din = mem_data_i;
```

```
163             end
164         endcase
165     end
166     assign data_r = cache_dout;
167 
168     // mem ctrl
169     always @ (*) begin
170         case (next_state)
171             S_IDLE, S_PRE_BACK, S_WAIT: begin
172                 mem_cs_o = 1'b0;
173                 mem_we_o = 1'b0;
174                 mem_addr_o = 32'b0;
175             end
176 
177             S_BACK: begin
178                 mem_cs_o = 1'b1;
179                 mem_we_o = 1'b1;
180                 mem_addr_o = {cache_tag,
    ↪ addr_rw[ADDR_BITS-TAG_BITS-1:BLOCK_WIDTH], next_word_count,
    ↪ {ELEMENT_WORDS_WIDTH{1'b0}}};
181             end
182 
183             S_FILL: begin
184                 mem_cs_o = 1'b1;
185                 mem_we_o = 1'b0;
186                 mem_addr_o = {addr_rw[ADDR_BITS-1:BLOCK_WIDTH],
    ↪ next_word_count, {ELEMENT_WORDS_WIDTH{1'b0}}};
187             end
188         endcase
189     end
190     assign mem_data_o = cache_dout;
191 
192     assign stall = (next_state != S_IDLE);
193 
194 endmodule
```

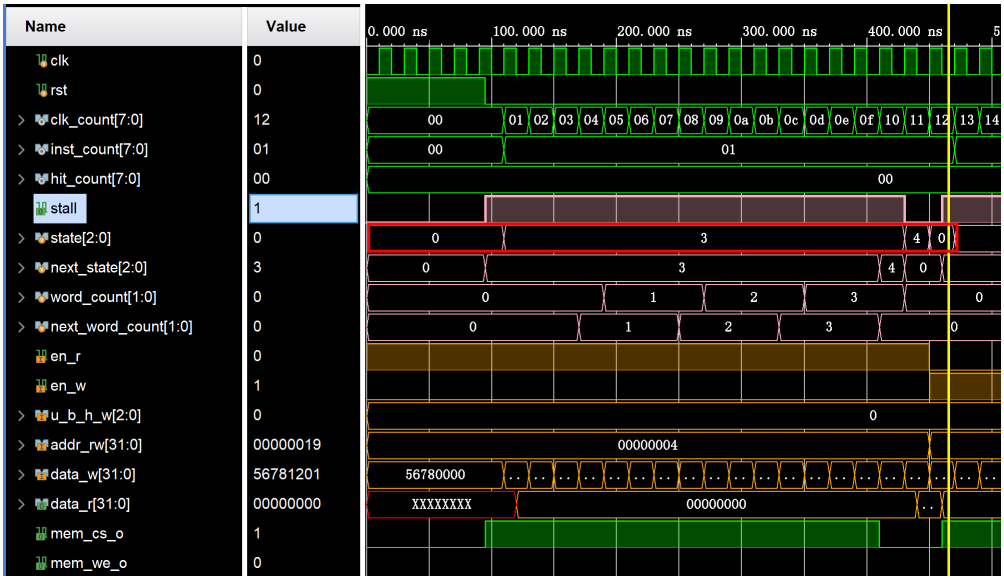
3.2 数据记录及结果分析

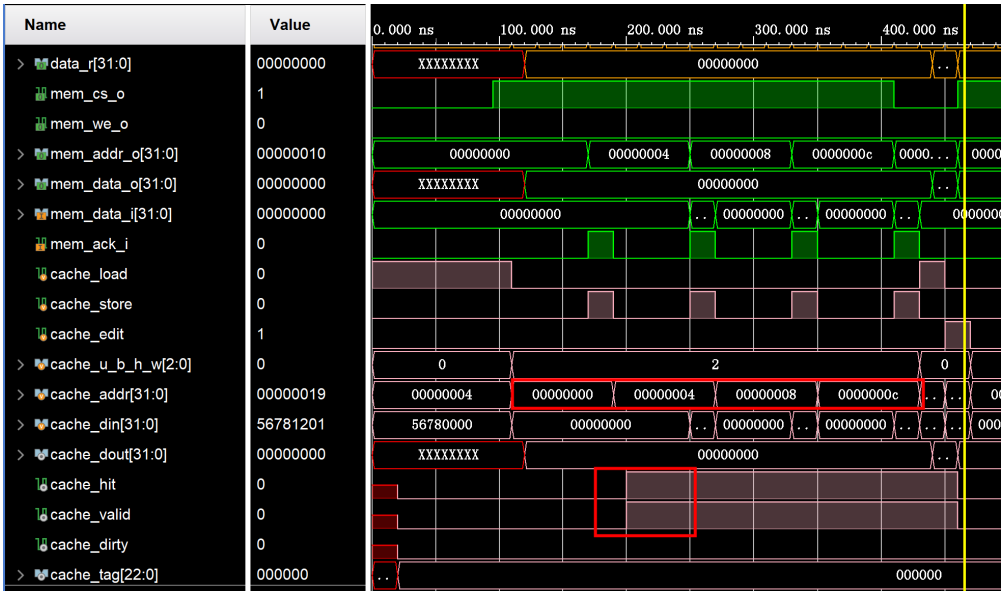
Since there are many cases in simulation wave, I only take some examples to show.For the convenience of explanation, all my explanations below are for the positions corresponding to the **yellow vertical lines** in the simulation diagram.

Test bench is as follows:

```
initial begin
        data[0] = 40'h0_2_00000004;  // read miss                1+17
        data[1] = 40'h0_3_00000019;  // write miss               1+17
        data[2] = 40'h1_2_00000008;  // read hit                 1
        data[3] = 40'h1_3_00000014;  // write hit                1
        data[4] = 40'h2_2_00000204;  // read miss                1+17
        data[5] = 40'h2_3_00000218;  // write miss               1+17
        data[6] = 40'h0_3_00000208;  // write hit                1
        data[7] = 40'h4_2_00000414;  // read miss + dirty        1+17+17
        data[8] = 40'h1_3_00000404;  // write miss + clean       1+17
        data[9] = 40'h0;           // end                      total: 128
        end
```

- first read miss(clean)





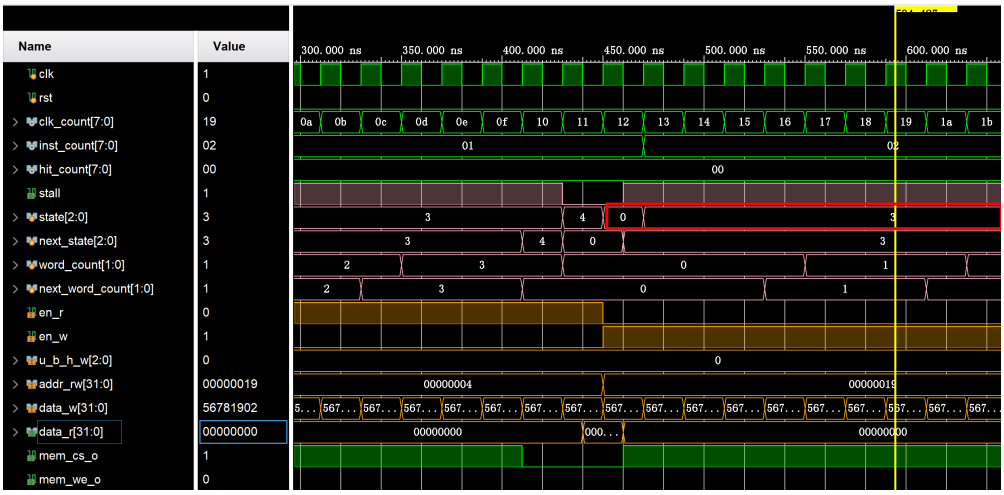
In the corresponding implementation:

```
data[0] = 40'h0_2_00000004; // read miss 1+17
```

Explanation: In the event of a read miss, and assuming there is an available and replaceable position in the cache, the state transitions are as follows: 0 (`S_IDLE`) → 3 (`S_FILL`) → 4 (`S_WAIT`) → 0 (`S_IDLE`). This corresponds to a normal read miss under the condition of a clean cache, where memory data is written into the cache.

Additionally, during the `S_FILL` process, because the CPU needs the data at address `0x4`, the cache reads the block from memory ranging from `0x0` to `0xC`. After reading the data at `0x4`, `cache_hit` and `cache_valid` become 1.

- first write miss(clean)

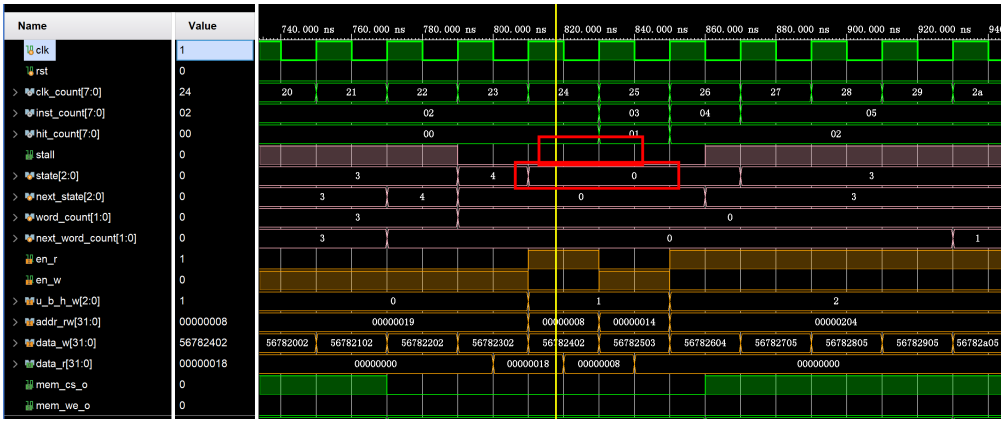


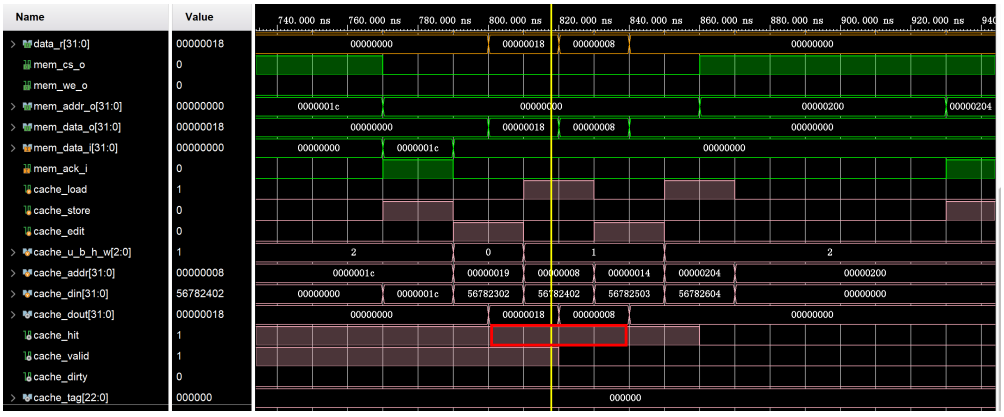
In the corresponding code:

```
data[1] = 40'h0_3_00000019; // write miss 1+17
```

Explanation: The state transitions remain as `S_IDLE` → `S_FILL` → `S_WAIT` → `S_IDLE`. This represents a write miss under the condition of a clean cache, writing the corresponding data from memory into the cache. Additionally, the write data address is `0x19`, so the block from `0x10` to `0x1C` is placed into the cache. `cache_hit` and `cache_valid` also become 1.

- read hit and write hit



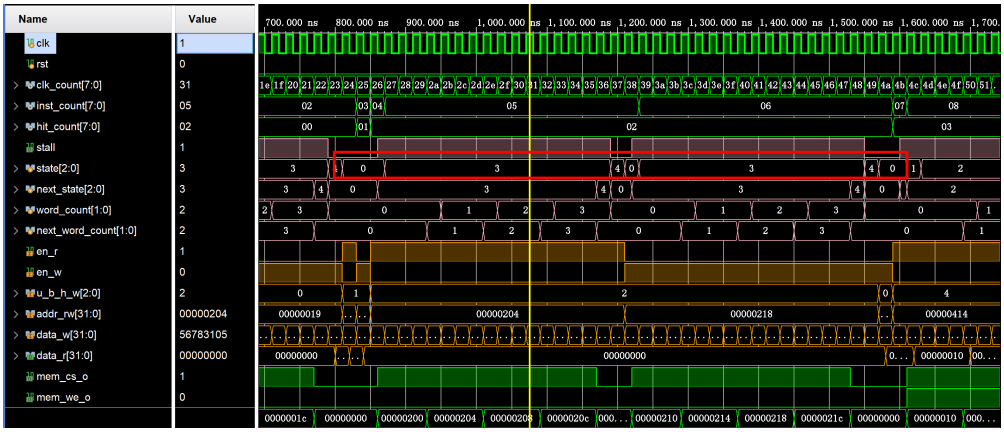


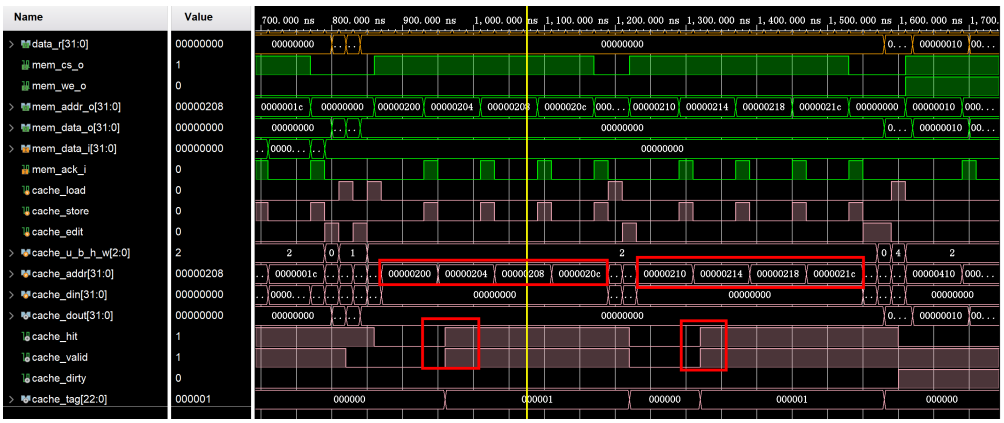
In the corresponding code:

```
data[2] = 40'h1_2_00000008; // read hit 1
data[3] = 40'h1_3_00000014; // write hit 1
```

Explanation: When a read hit and a write hit occur, the state machine remains in the `S_IDLE` state. The CPU does not experience any pause, so the third and fourth instructions execute directly and complete. During this process, `stall` is 0, and `cache_hit` is also 1.

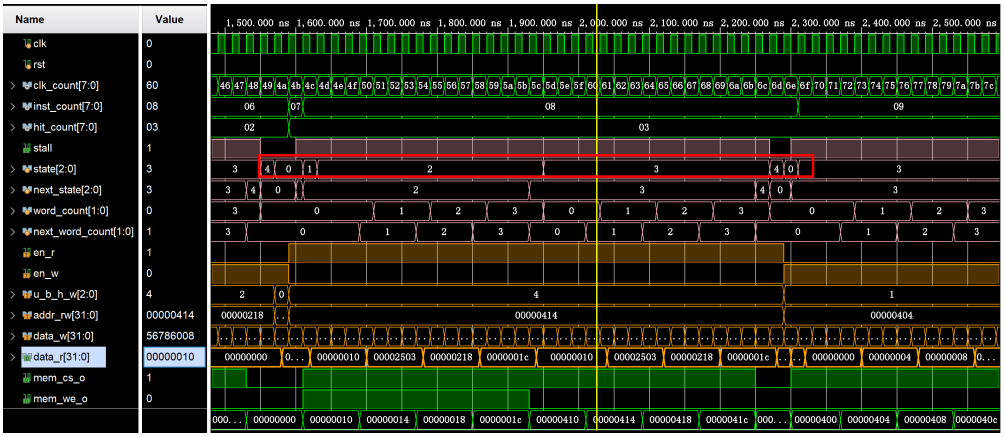
- dirty miss replace

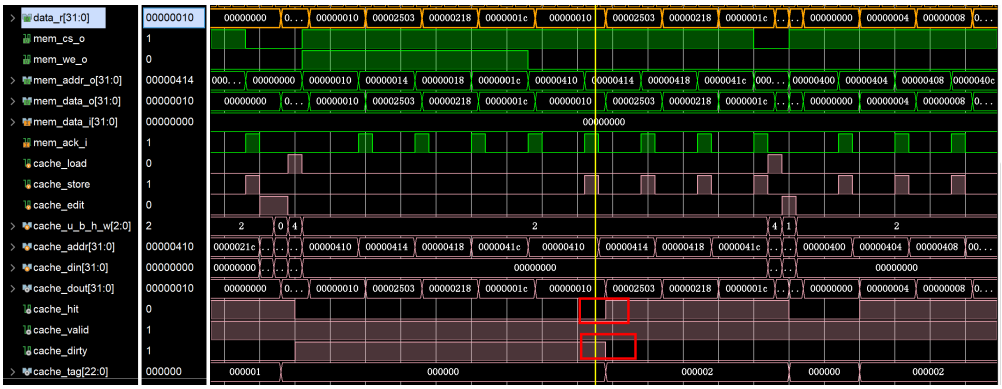





```
initial begin
data[4] = 40'h2_2_00000204; // read miss 1+17
data[5] = 40'h2_3_00000218; // write miss 1+17
data[6] = 40'h0_3_00000208; // write hit 1
```

Explanation: In these instructions, the normal replacement process occurs, going through the states `S_IDLE` → `S_FILL` → `S_WAIT` → `S_IDLE`. The data is loaded into the cache, and `cache_hit` is set to 1 when the corresponding event occurs. Then, for the `write hit`, there is no pause, and `stall` remains 0.



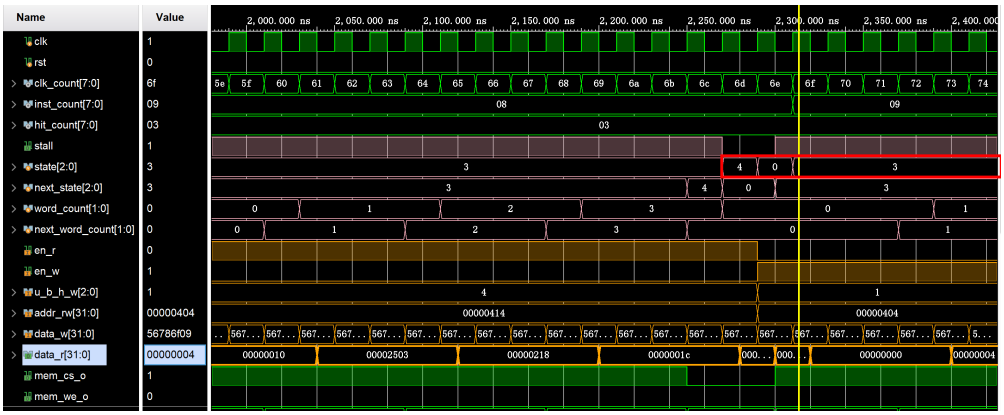


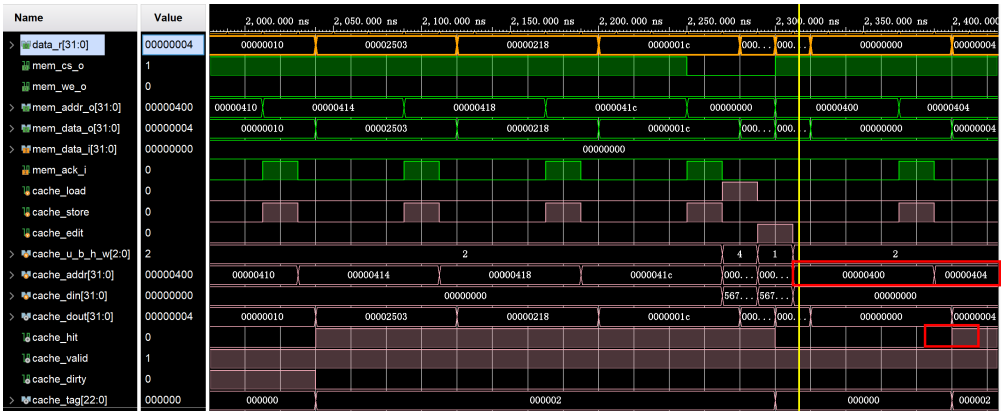

```
data[7] = 40'h4_2_00000414; // read miss + dirty 1+17+17
```

Explanation: To achieve a dirty replacement in a 2-way set-associative cache, it is necessary to fill the cache with the data for the desired operation and modify one of the data blocks to set it as dirty. During the write, this dirty data block is replaced.

The preceding accesses are to create the conditions for this scenario. The state transitions at this point are 0 (`S_IDLE`) → 1 (`S_PRE_BACK`) → 2 (`S_BACK`) → 3 (`S_FILL`) → 4 (`S_WAIT`) → 0 (`S_IDLE`). This sequence corresponds to a dirty replacement in the case of a cache miss, fitting the conditions when cache dirty=1.

- clean miss replace






```
data[8] = 40'h1_3_00000404; // write miss + clean 1+17
```

Explanation: This instruction represents the replacement of a clean block, going through the state transitions `S_IDLE` → `S_FILL` → `S_WAIT` → `S_IDLE`. This process corresponds to a clean replacement in the case of a cache miss.

Now I'll show the figures about the result on sword. Firstly, I'll show data memory and instruction memory.

NO.	Data	Addr.	Comment
0	000080BF	0	
1	00000008	4	
2	00000010	8	
3	00000014	C	
4	FFFF0000	10	
5	0FFF0000	14	
6	FF000F0F	18	
7	F0F0F0F0	1C	
8	00000000	20	
9	00000000	24	
10	00000000	28	
11	00000000	2C	
12	00000000	30	
13	00000000	34	
14	00000000	38	
15	00000000	3C	

NO.	Instruction	Addr.	Comment
16	00000000	40	
17	00000000	44	
18	00000000	48	
19	00000000	4C	
20	A3000000	50	
21	27000000	54	
22	79000000	58	
23	15100000	5C	
24	00000000	60	
25	00000000	64	
26	00000000	68	
27	00000000	6C	
28	00000000	70	
29	00000000	74	
30	00000000	78	
31	00000000	7C	

图 1: data memory

NO.	Instruction	Addr.	Label	ASM	Comment
0	00000013	0	__start:	addi x0, x0, 0	
1	01c00083	4		lb x1, 0x01C(x0)	# F0F0F0F0 in 0x1C # miss, read 0x010~0x01C to set 1 line 0
2	01c01103	8		lh x2, 0x01C(x0)	# FFFFF0F0 hit
3	01c02183	C		lw x3, 0x01C(x0)	# F0F0F0F0 hit
4	01c04203	10		lbu x4, 0x01C(x0)	# 000000F0 hit
5	01c05283	14		lhu x5, 0x01C(x0)	# 0000F0F0 hit
6	21002003	18		lw x0, 0x210(x0)	# miss, read 0x210~0x21C to cache set 1 line 1
7	abcde0b7	1C		lw x7, 20(x0)	
8	402200b3	20		lui x1 0xABCDE	
9	71c08093	24		addi x1, x1, 0x71C	# x1 = 0xABCDE71C
10	00100023	28		sb x1, 0x0(x0)	# miss, read 0x000~0x00C to cache set 0 line 0
11	00101223	2C		sh x1, 0x4(x0)	# hit
12	00102423	30		sw x1, 0x8(x0)	# hit

图 2: instruction memory 1

NO.	Instruction	Addr.	Label	ASM	Comment
13	20002303	34		lw x6, 0x200(x0)	# miss, read 0x200~0x20C to cache set 0 line 1
14	40002383	38		lw x7, 0x400(x0)	# miss, write 0x000~0x00C back to ram, then read 0x400~40C to cache set 0 line 0
15	41002403	3C		lw x8, 0x410(x0)	# miss, no write back because of clean, read 0x410~41C to cache set 1 line 0
16	0ed06813	40	loop:	ori x16, x0, 0xED	# end
17	ffdff06f	44		jal x0, loop	

图 3: instruction memory 2

Then I'll show the results.

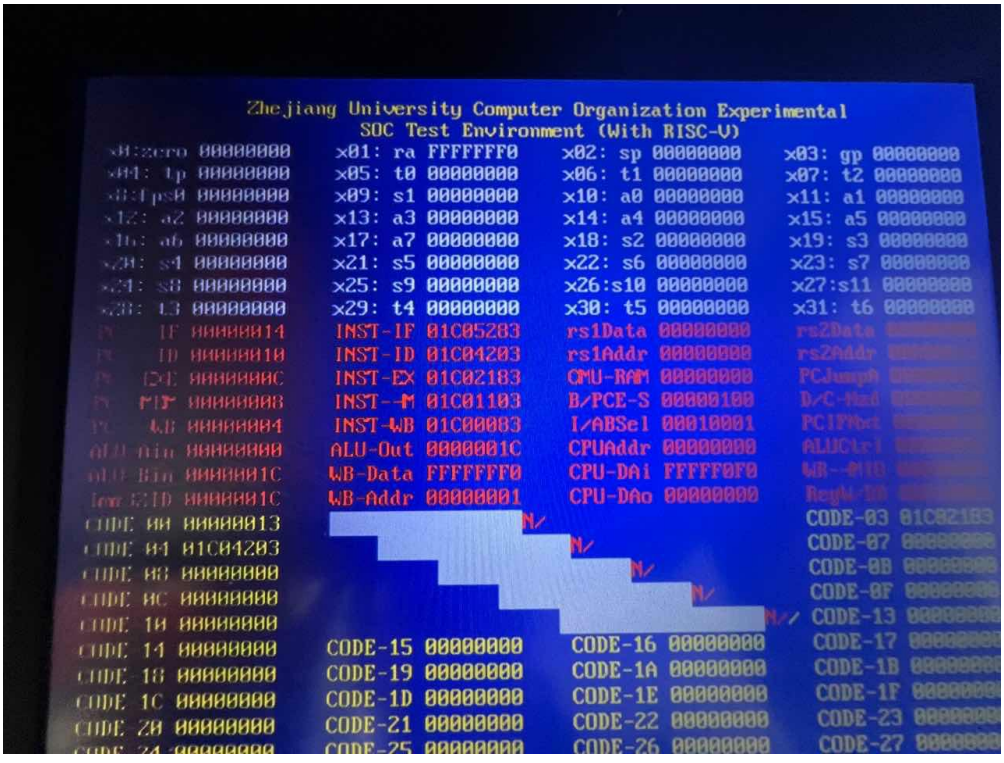


图 4: x1 load correct

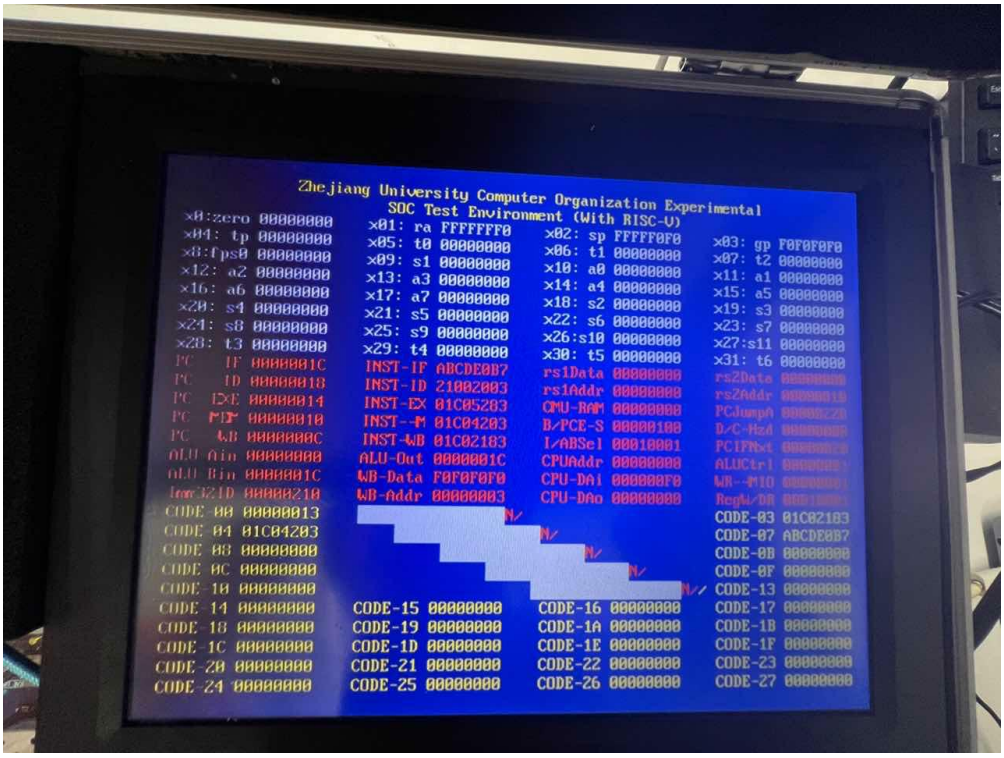


图 5: x2 and x3 load correct

图 6: x4 and x5 load correct

图 7: x1 set to correct value

图 8: x1 add to correct value

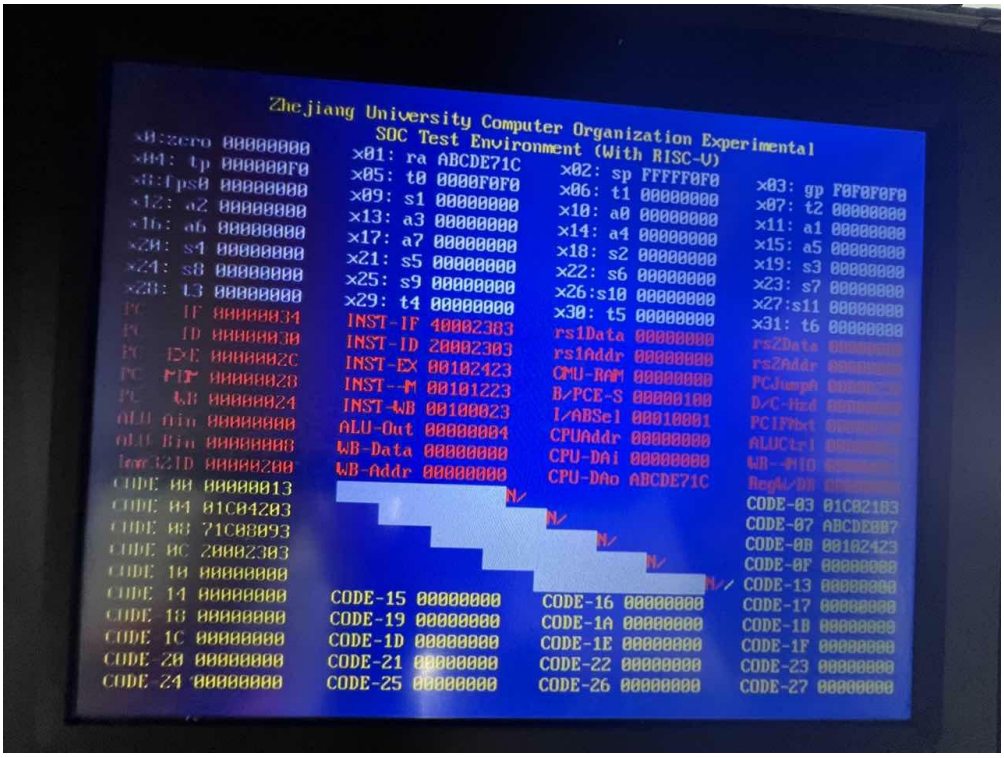


图 9: miss and stall

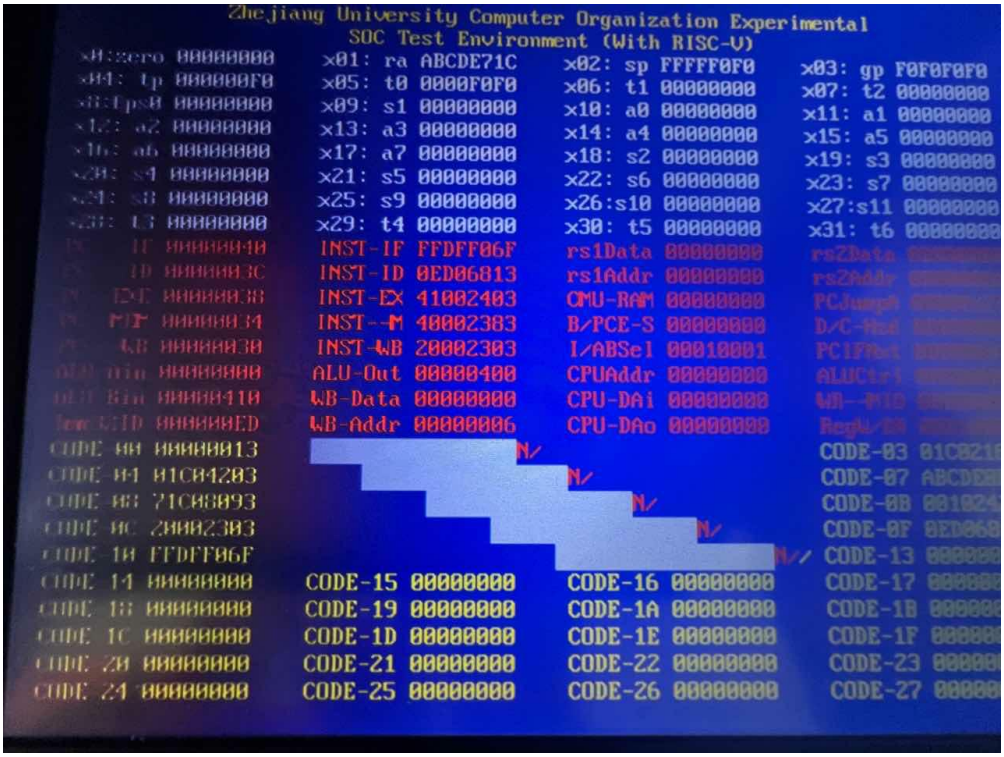


图 10: miss and stall 2

4 讨论与心得

The code need to write for the experiment does not exceed In addition, this experiment requires no more than 10 lines of code written by myself, and the state machine is also very clear. No major difficulties were encountered during the experiment.