本科实验报告

课程名称：计算机体系结构

设计名称：Pipelined CPU

姓　　名：曾帅王异鸣

学　　号：3190105729 3190102780

学　　院：计算机学院

专　　业：计算机科学与技术

班　　级：图灵 1901

指导老师：姜晓红

2021 年 10 月 16 日

目录

1 实验目的

本次实验要求我们实现加入了 cache 的 CPU 流水线

1. 了解 Cache Management Unit 的运行规则以及其状态图
2. 掌握 CMU 的设计方法以及与 CPU 的交互方法
3. 掌握验证 CMU 正确性的方法，并把含有 Cache 的 CPU 与不含 Cache 的 CPU 进行比较。

2 实验内容

实验的基本要求是实现一个包含 cache 的 CPU 流水线
本实验已给出主要框架，需要完成的实验内容如下：

1. 设计 CMU 并将其整合进 CPU 中
2. 观察并分析仿真结果
3. 比较含有 Cache 与不含 Cache 的 CPU 的表现。

3 实验原理

3.1 Cache Management Unit

本实验中需要实现的主要模块就是 CMU，即 Cache Management Unit，CMU 中内含 cache 模块，其负责处理数据并将其传入或传出 cache，并完成 cache 与 CPU 和 memory 的交互。

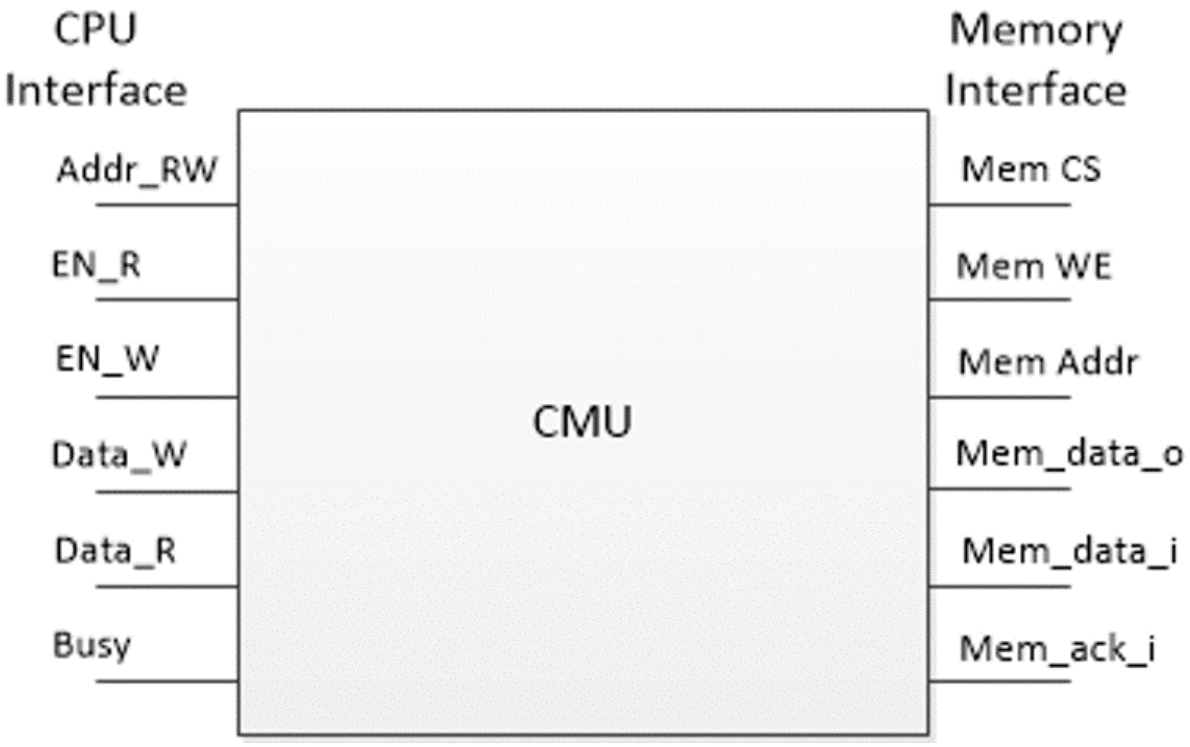

图 3.1: CMU 模块

CMU 模块安置在流水线的 MEM 阶段，其与 CPU 之间有一系列的接口，包括 Data_Read，Data_Write，Address 等，这提供了 CPU 与 cache 之间数据传输的通道。同时，CMU 还与 Memory 之间有一系列接口，以用于 cache 和 memory 之间进行数据传输。

3.2 Cache Operation Flow

CMU 中的操作流程如下图所示

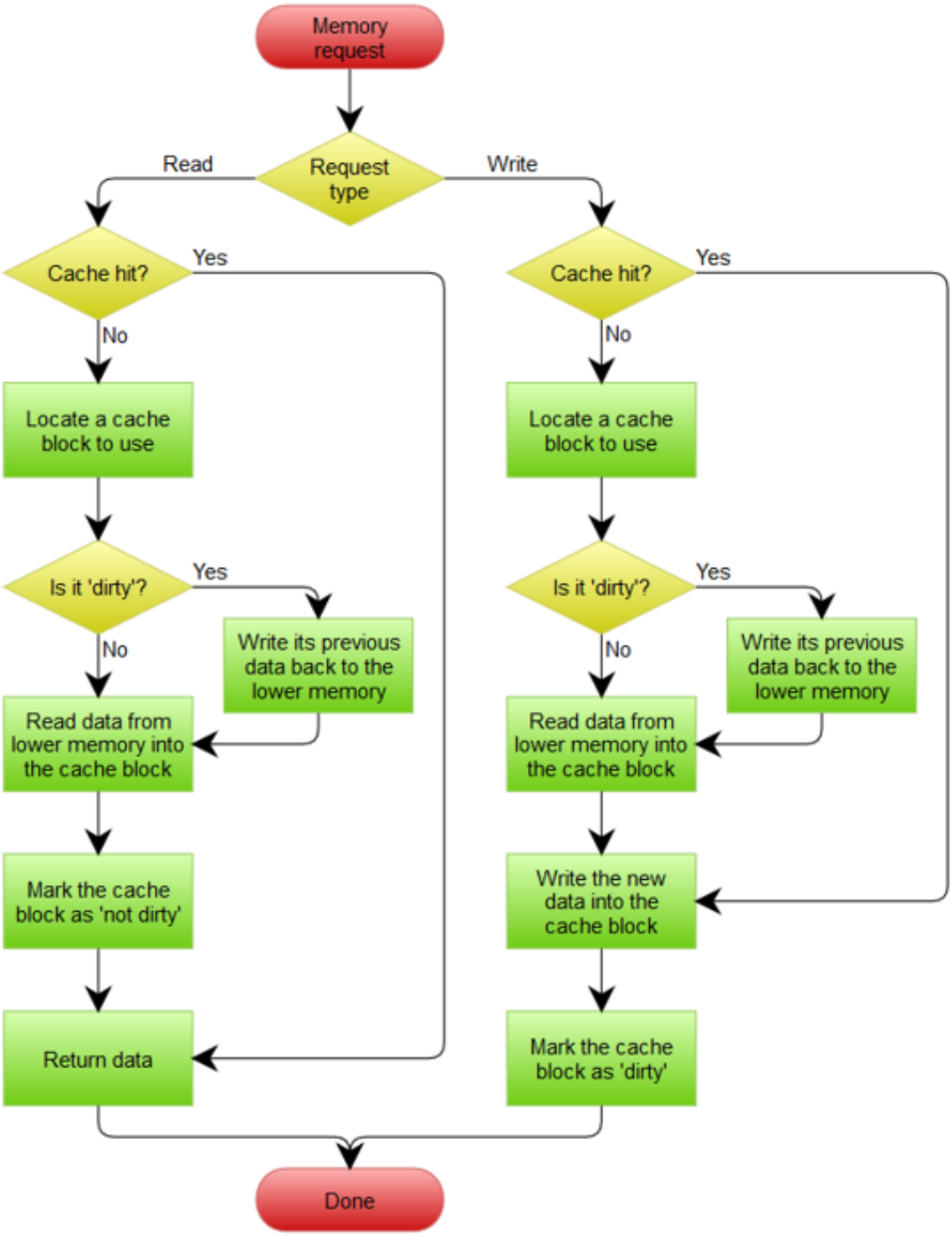

图 3.2: Cache Operation Flow

当 CPU 需要对数据进行操作时，会首先传入 read 或 write bit。CMU 接收到信号后首先查看地址是否 hit，若 hit 则直接读或写数据。否则检查需要置换的 block 是否 dirty，若 dirty 则需要先将当前块写回 memory。之后从 memory 中取数据存入 cache，

再进行读或写操作。

3.3 Cache Management State Machine

CMU 的内部结构实际上是一个有限状态机，其有 S_IDLE、S_PRE_BACK、S_BACK、S_FILL、S_WAIT 等 5 个状态。

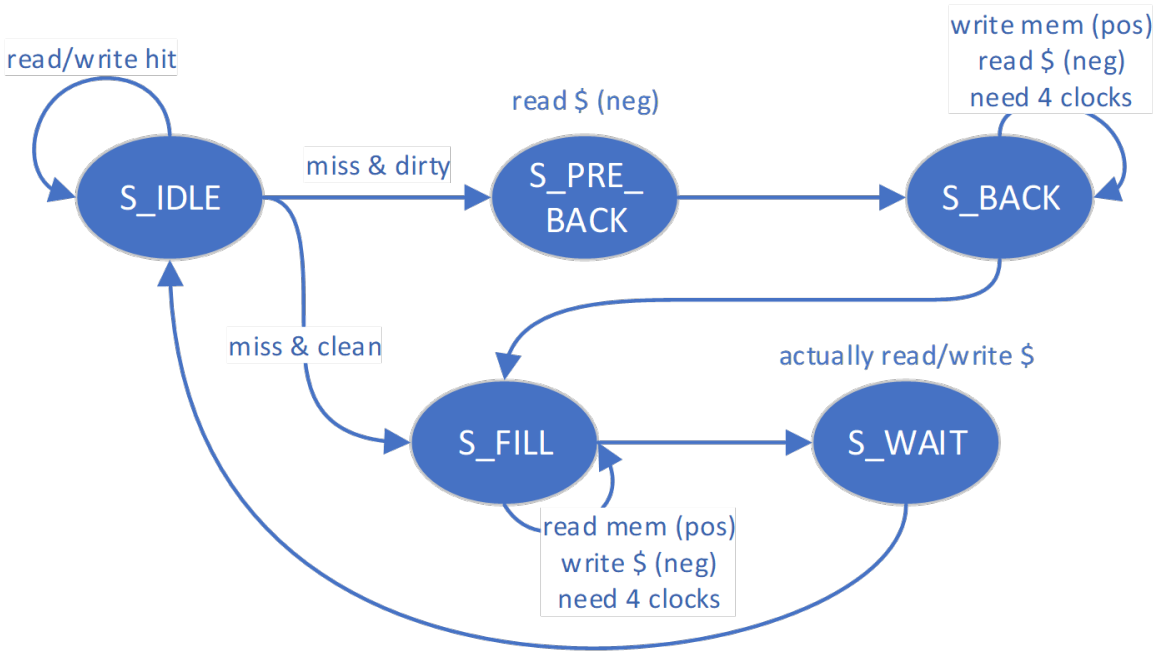

图 3.3: CMU 状态机

cache 操作均发生在当前 state 的下降沿，memory 操作均发生在当前 state 的上升沿。每个状态的具体描述如下：

1. S_IDLE：空闲状态，不进行 memory 操作，cache 操作 hit 的情况下一直处于这个状态。

2. S_PRE_BACK：为了写回，先进行一次读 cache。

3. S_BACK：上升沿将上个状态的数据写回到 memory，下降沿从 cache 读下次需要写回的数据，由计数器控制直到整个 cache line 全部写回。由于 memory 设置为 4 个周期完成读写操作，因此需要等待 memory 给出 ack 信号，才能进行状态的改变。

4. S_FILL：上升沿从 memory 读取数据，下降沿向 cache 写入数据，由计数器控制直到整个 cache line 全部写入。与 S_BACK 类似，需要等待 ack 信号。

5. S_WAIT：执行之前由于 miss 而不能进行的 cache 操作。

3.4 源代码

```
1  module cmu (
2          // CPU side
3          input clk,
4          input rst,
5          input [31:0] addr_rw,
6          input en_r,
7          input en_w,
8          input [2:0] u_b_h_w,
9          input [31:0] data_w,
10         output [31:0] data_r,
11         output stall,
12
13         // mem side
14         output reg mem_cs_o = 0,
15         output reg mem_we_o = 0,
16         output reg [31:0] mem_addr_o = 0,
17         input [31:0] mem_data_i,
18         output [31:0] mem_data_o,
19         input mem_ack_i,
20
21         // debug info
22         output [2:0] cmu_state
23         );
24
25         `include "addr_define.vh"
26
27         reg [ADDR_BITS-1:0] cache_addr = 0;
28         reg cache_load = 0;
29         reg cache_store = 0;
30         reg cache_edit = 0;
31         reg [2:0] cache_u_b_h_w = 0;
32         reg [WORD_BITS-1:0] cache_din = 0;
33         wire cache_hit;
34         wire [WORD_BITS-1:0] cache_dout;
35         wire cache_valid;
36         wire cache_dirty;
37         wire [TAG_BITS-1:0] cache_tag;
38
39         cache CACHE (
40         .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         //important
193         assign stall = (next_state != S_IDLE);
194
195 endmodule
```

4 实验步骤与调试

4.1 仿真

根据已经写好的代码，进行仿真模拟

4.2 综合

选择左侧面板的 Run Synthesis 或者点击上方的绿色小三角，选择 Synthesis

4.3 实现

选择左侧面板的 Run Implementation 或者点击上方的绿色小三角，选择 Implementation。值得注意的是执行 implementation 之前应该确保引脚约束存在且正确，同时之前已经综合过最新的代码。

4.4 验证设计

选择左侧面板的 Open Elaborated Design，输出的结果如下，根据原理图来判断，基本没有问题

4.5 生成二进制文件

选择左侧面板的 Generate Bitstream 或者点击上上的绿色二进制标志。同时生成 Bitstream 前要确保: 之前已经综合、实现过最新的代码。如没有，直接运行会默认从综合、实现开始。此过程还要注意生成的 bit 文件默认存放在.runs 下相应的 implementation 文件夹中

4.6 烧写上板

点击左侧的 Open Hardware Manager → 点击 Open Target → Auto Connect → 点击 Program Device → 选择 bistream 路径，烧写。验证结果见实验结果部分。

5 实验结果与分析

5.1 Cache 仿真结果分析

仿真代码设计 为了实现 Cache 正确性的仿真验证,本次实验采用了对 Cache 和 CMU 模块的单独仿真。仿真代码的设计如下：

```
1    module cmu_sim (
2        input wire clk,
3        input wire rst,
4        output reg [7:0] clk_count = 0,
5        output reg [7:0] inst_count = 0,
6        output reg [7:0] hit_count = 0
```

```
7         );
8
9         // instruction
10        reg [3:0] index = 0;
11        wire valid;
12        wire write;
13        wire [31:0] addr;
14        wire [2:0] u_b_h_w;
15        wire stall;
16        inst INST (
17            .clk(clk),
18            .rst(rst),
19            .index(index),
20            .valid(valid),
21            .write(write),
22            .addr(addr),
23            .u_b_h_w(u_b_h_w)
24        );
25
26        always @(posedge clk) begin
27            if (rst)
28                index <= 0;
29            else if (valid && ~stall)
30                index <= index + 1'h1;
31        end
32        // ram
33        wire mem_cs;
34        wire mem_we;
35        wire [31:0] mem_addr;
36        wire [31:0] mem_din;
37        wire [31:0] mem_dout;
38        wire mem_ack;
39        data_ram RAM (
40            .clk(clk),
41            .rst(rst),
42            .addr({21'b0, mem_addr[10:0]}),
43            .cs(mem_cs),
44            .we(mem_we),
45            .din(mem_din),
46            .dout(mem_dout),
47            .stall(),
48            .ack(mem_ack),
```

```
49              .ram_state()
50          );
51
52          // cache
53          wire [31:0] data_r;
54          cmu CMU (
55              .clk(clk),
56              .rst(rst),
57              .addr_rw(addr),
58              .u_b_h_w(u_b_h_w),
59              .en_r(~write),
60              .data_r(data_r),
61              .en_w(write),
62              .data_w({16'h5678, clk_count, inst_count}),
63              .stall(stall),
64              .mem_cs_o(mem_cs),
65              .mem_we_o(mem_we),
66              .mem_addr_o(mem_addr),
67              .mem_data_i(mem_dout),
68              .mem_data_o(mem_din),
69              .mem_ack_i(mem_ack),
70              .cmu_state()
71          );
72
73          // counter
74          reg stall_prev;
75
76          always @(posedge clk) begin
77              if (rst)
78                  stall_prev <= 0;
79              else
80                  stall_prev <= stall;
81          end
82
83          always @(posedge clk) begin
84              if (rst) begin
85                  clk_count <= 0;    // 时钟计数
86                  inst_count <= 0;   // 指令计数
87                  hit_count <= 0;    // 命中计数
88              end
89              else if (valid) begin
90                  clk_count <= clk_count + 1'h1;
```

```
                inst_count <= index + 1'h1;
                if (~stall_prev && ~stall)
                    hit_count <= hit_count + 1'h1;
            end
        end

    endmodule
```

为了验证结果，本次实验还设计了一个单独的 Test Bench 以供测试：

```
module inst (
    input wire clk,
    input wire rst,
    input wire [3:0] index,  // instruction index
    output wire valid,  // stop running if valid is 0
    output wire write,  // write enable signal for cache
    output wire [31:0] addr,  // address for cache
    output wire [2:0] u_b_h_w
    );
    reg [39:0] data [0:9];
    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
    assign
        u_b_h_w = data[index][38:36],
        valid = data[index][33],
        write = data[index][32],
        addr = data[index][31:0];

endmodule
```

仿真结果分析 根据上述仿真代码，实验的仿真结果如下，在此部分我组将对仿真结果进行逐一分析。首先同样是多周期的重置过程，以保证 cache 与内存内容的彻底重置。

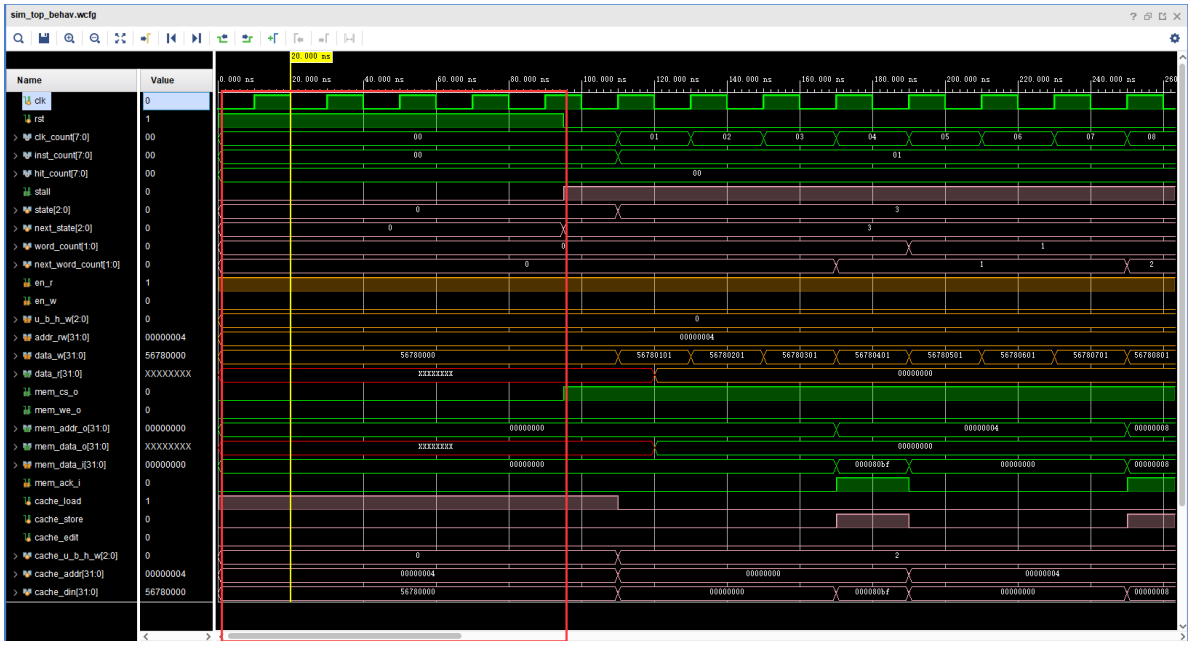

图 5.4: 重置过程

在 95ns 后，重置过程结束，此时将会进入第一个状态，对应 Test Bench 中第一条 cache 访问的执行，此时我们可以看到 mem_cs_o 信号已经升起，由于第一次内存访问是 read 操作并且是 read miss，所以可以看到 next_state 已经被置为了 3(S_FILL)，准备从 memory 中读取数据，并且此时 cmu 的 stall 信号升起，仿真结果正确。

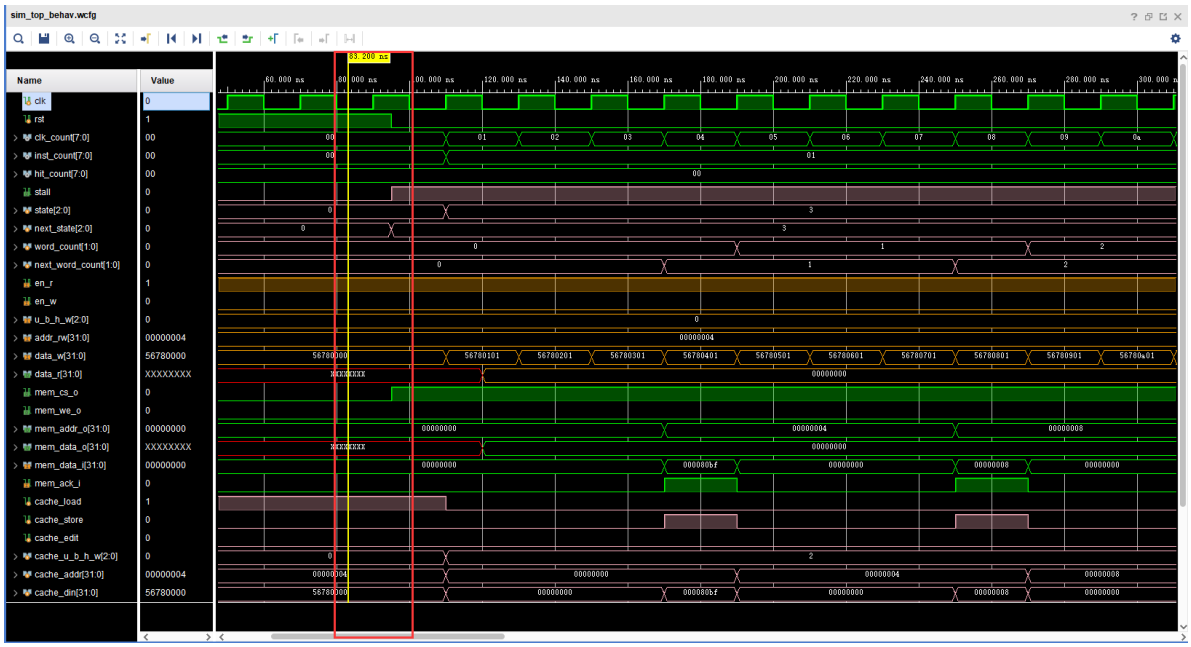

图 5.5: read miss

S_FILL 状态在上升沿从 memory 读取数据，下降沿向 cache 写入数据，由计数器控制直到整个 cache line 全部写入。由于每个 Word 的读取需要 4 个时钟周期，而一个 Data Line 中含有 4 个 Word，所以此过程需要 16 个时钟周期完成，我们可以从仿真图上看到 word_count 的值从 0 到 3，并且在每次读 Word 的最后一个周期 mem_ack 和 cache_store 信号都升起，说明已经从内存中得到一个 Word 并准备写入 cache，仿真结果正确。

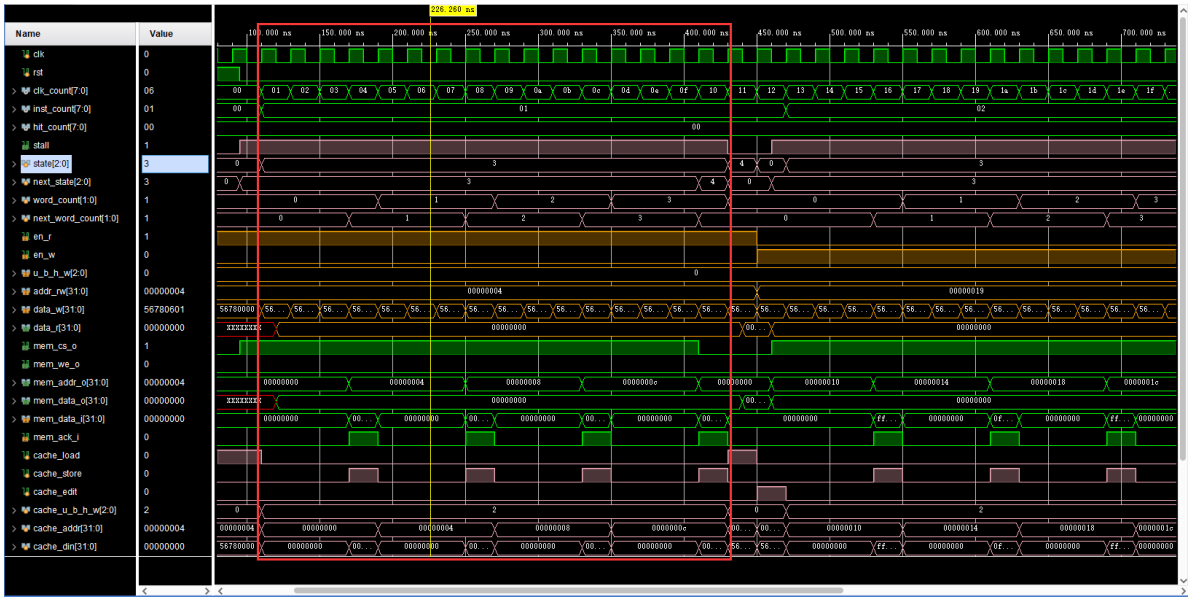

图 5.6: S_FILL

接下来进入 S_WAIT 状态，执行之前由于 miss 而不能进行的 cache 操作。此时 state 为 4(S_WAIT)，并且 cache_load 信号重新升起，并且 stall 信号降下仿真结果正确。

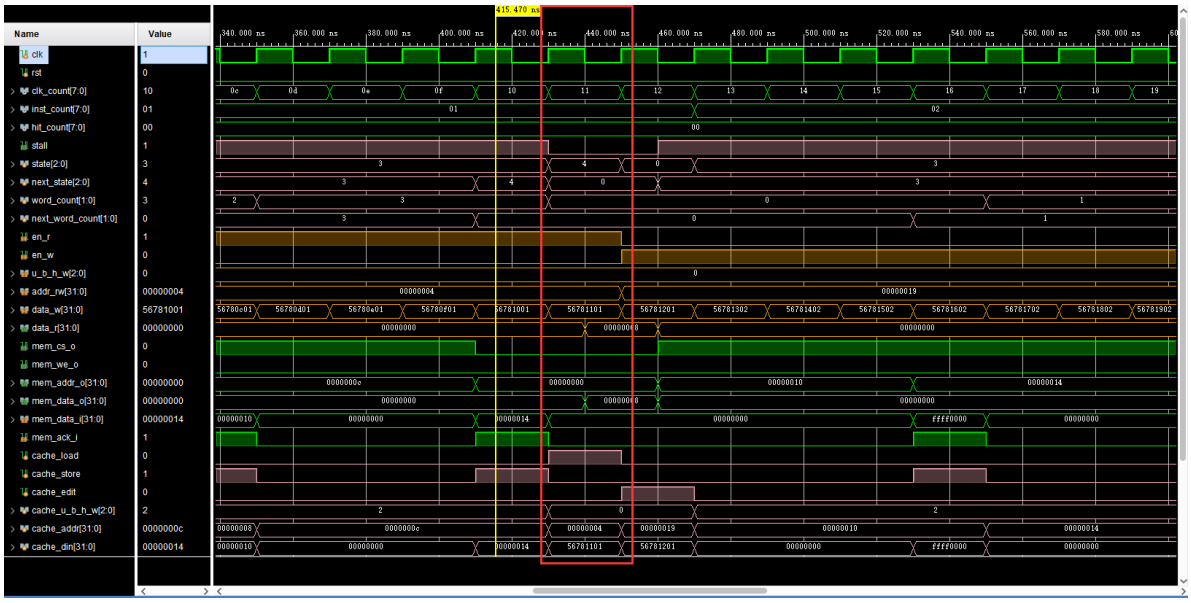

图 5.7: S_WAIT

接下来仿真 Write Miss 的情形，可以看到此时 en_w 和 cache_load 信号升起，cache addr 也变为 0x19

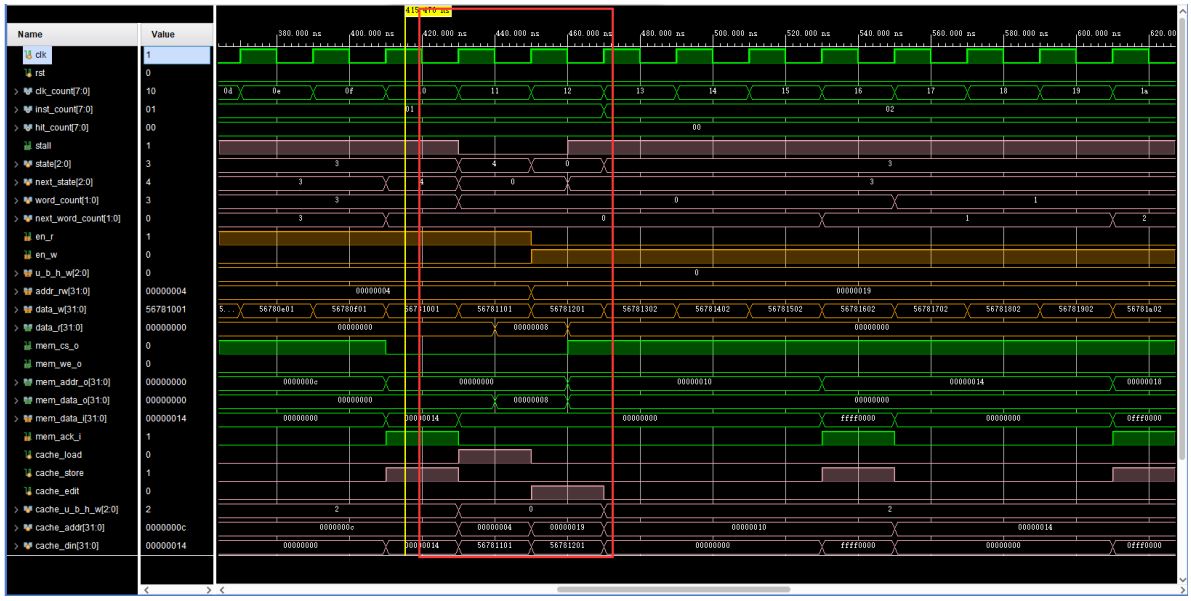

图 5.8: write miss

之后的同样进入 S_FILL 阶段，此阶段与前类似，在此不再赘述。此后进入 S_WAIT 状态，此时 cache_edit 信号升起，并且 cache_addr 也被修改为对应地址，仿真结果

正确

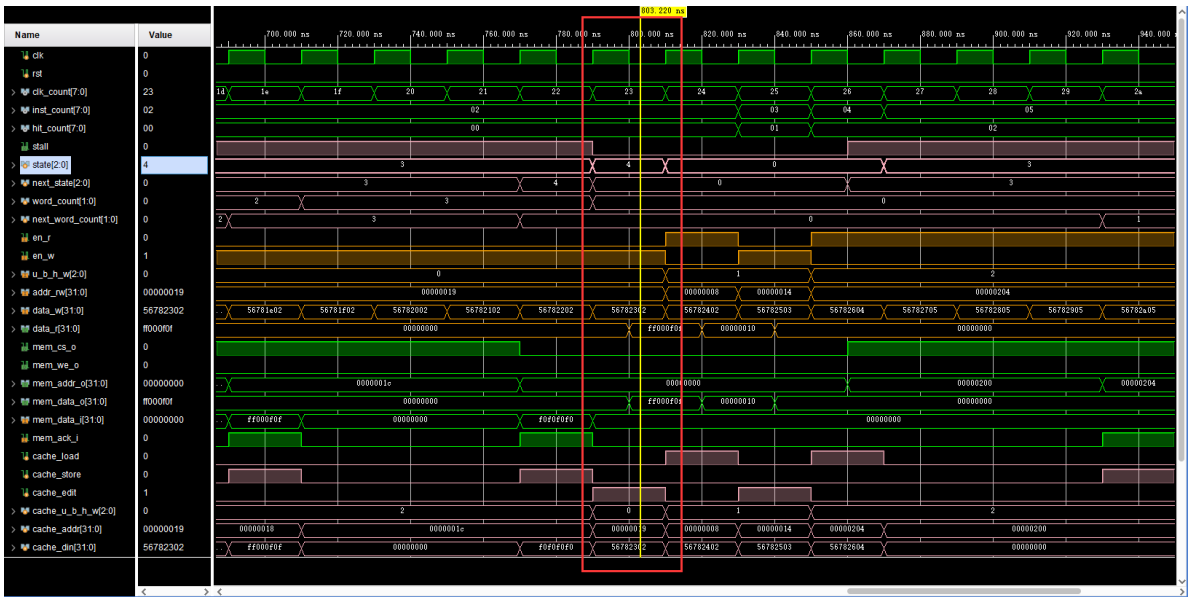

图 5.9: S_WAIT

read hit 与 write hit 的情形仅需一个时钟周期就可以完成，对应的使能信号升起，cache 的 addr 和 din/dout 也被修改为对应值，从图中可以看到仿真结果正确。

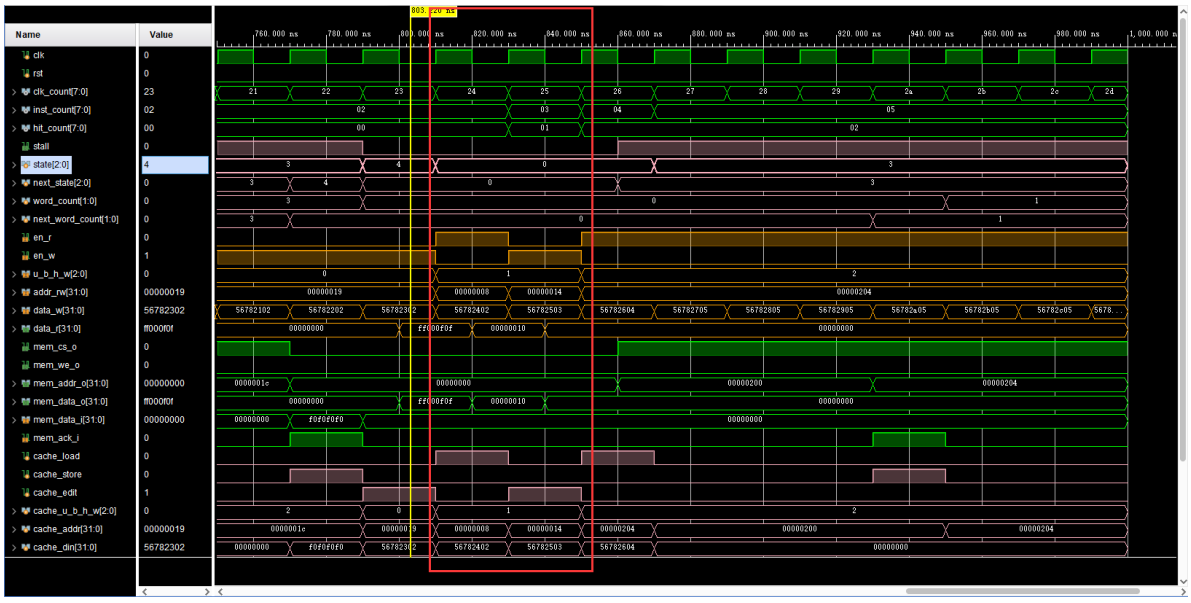

图 5.10: read hit

下面再来检查 dirty 情形 cache 的正确性，经过一系列操作之后再读取脏块，可以看到 state 下一周期被置为了 2(S_BACK)，S_BACK 在上升沿将上个状态的数据写回到 memory，下降沿从 cache 读下次需要写回的数据（因此最后一次读无意义），由

计数器控制直到整个 cache line 全部写回。由于 memory 设置为 4 个周期完成读写操作，因此需要等待 memory 给出 ack 信号，才能进行状态的改变。由于一个 Data line 有 4 个 Word 而一个 Word 同样需要 4 个周期写回，所以此过程需要额外经历一个 S_PRE_BACK 状态和 16 个时钟周期的 S_BACK 状态，然后再类似与上述情形经历 16 周期 S_FILL 状态和 S_WAIT 状态，仿真结果正确。

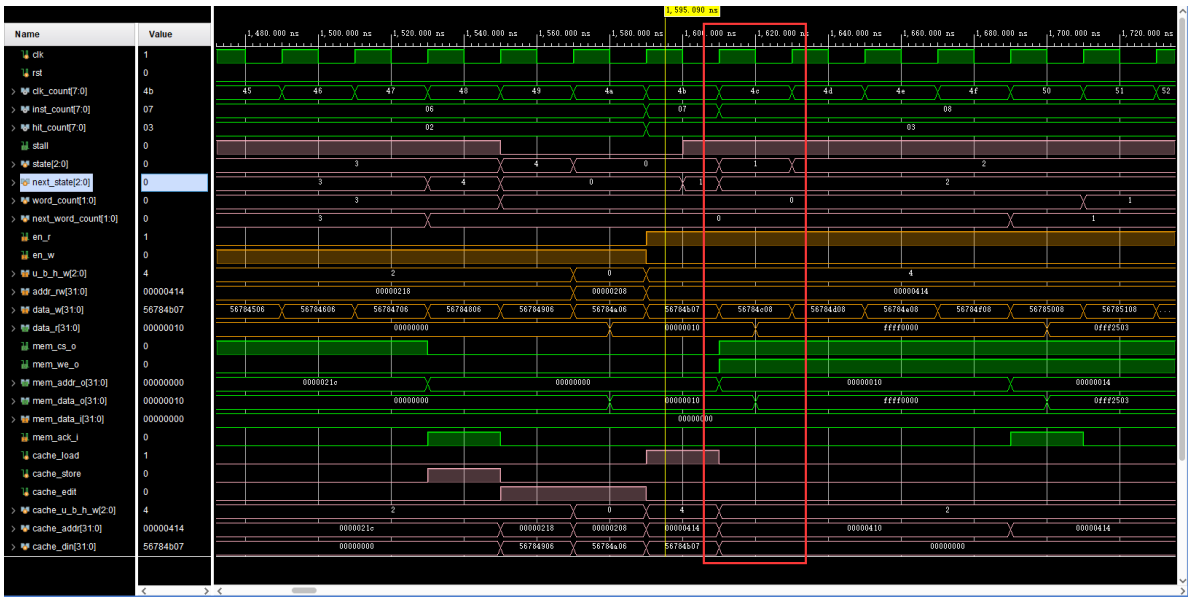

图 5.11: S_PRE_BACK

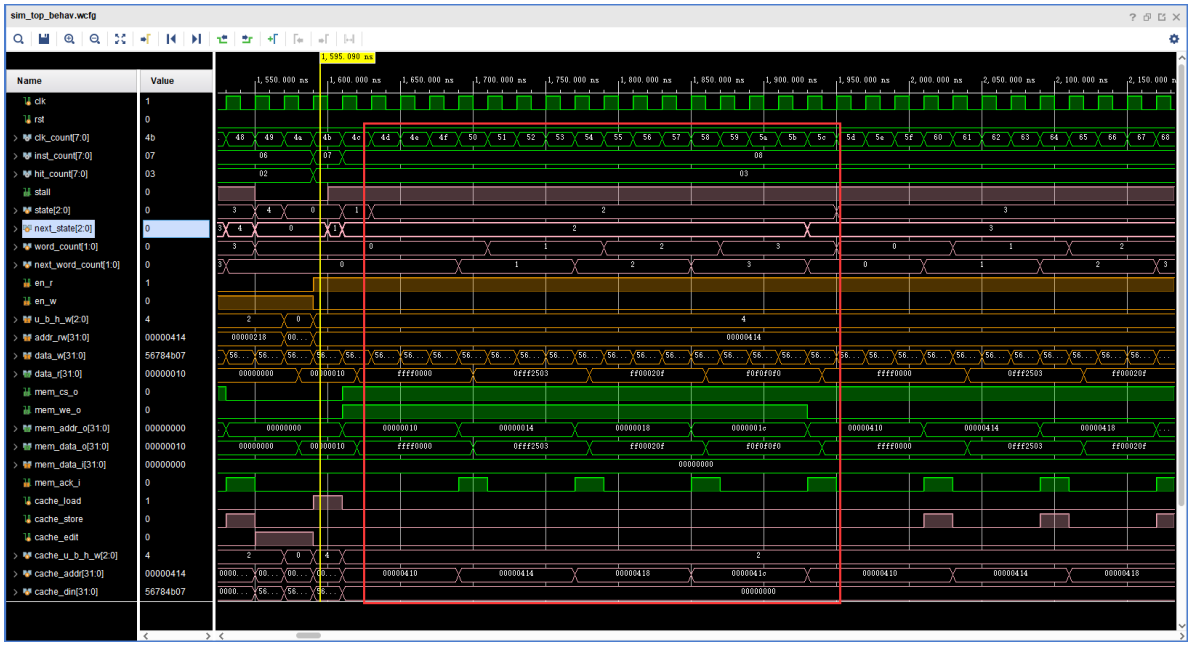

图 5.12: S_BACK

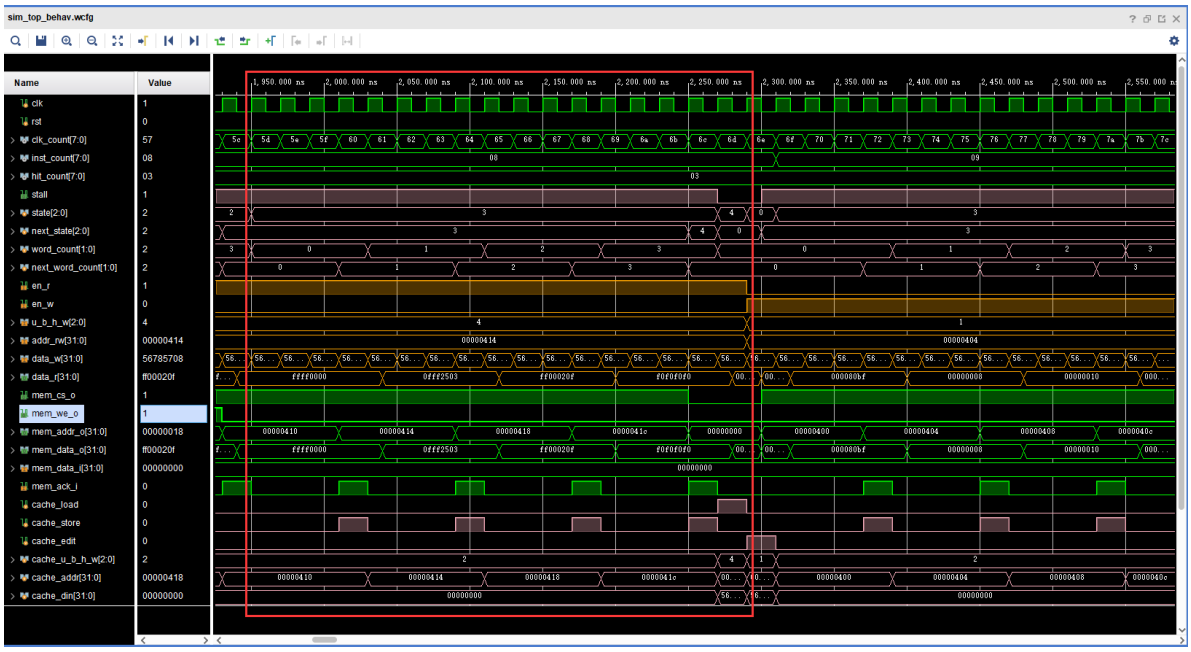

图 5.13: S_FILL and S_WAIT

5.2 上板结果分析

仿真结果正确已经基本验证了实验的正确性，下面我组再简单的通过几张结果截图分析正确性。

指令流进 MEM 阶段之前不做处理

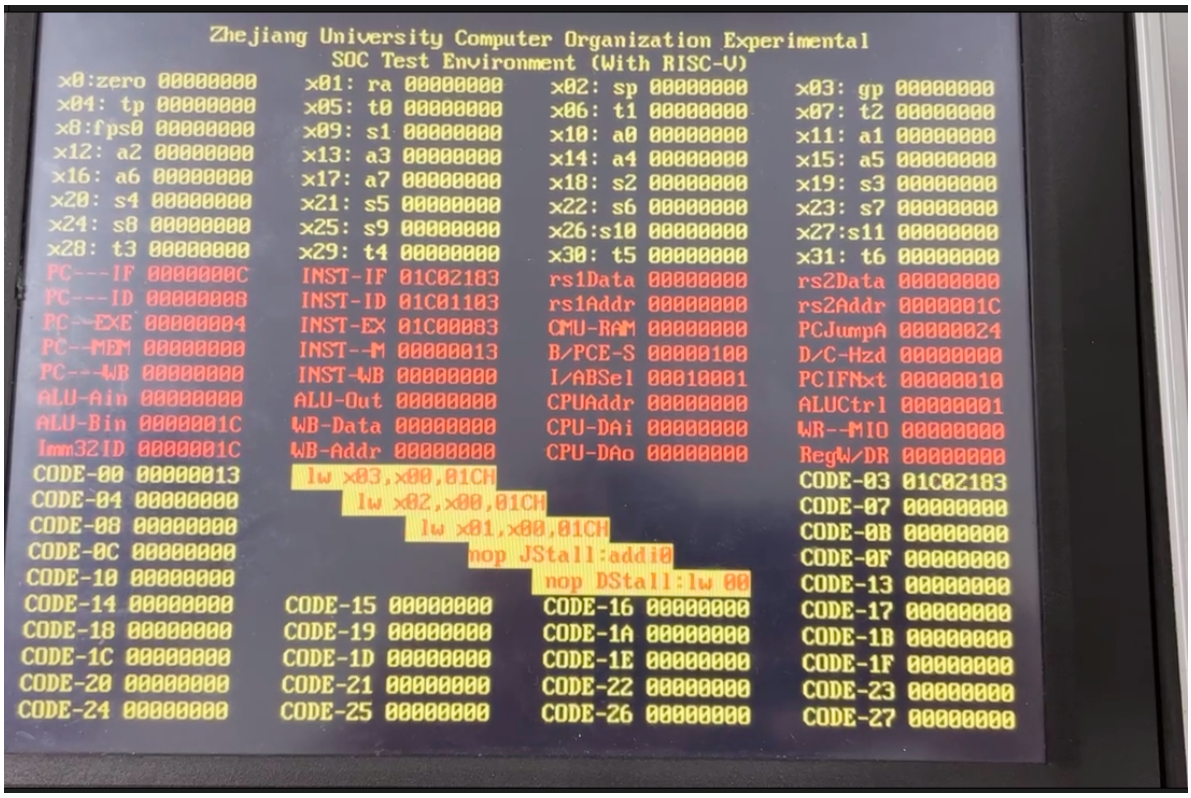

图 5.14: res1

第一条 load 指令进入 MEM 阶段后会花 16 个周期进行 S_FILL 阶段，从 CMU_RAM 的值可以看到每次读一个 Word 都是从 00030001 增加到 00030003，其中第四位是 state，最后一位是 word_count

图 5.15: res2

后续的流程类似，均符合上部分仿真结果，具体流程会在验收时展示。

6 讨论与心得

本实验要求在将上一次实验实现的 cache 模块加入流水线，本次实验主要需要理解整个 CMU 的控制流程，在理解此流程图的前提下实现过程是比较清晰的。此次实验不仅让我们进一步加深了对 cache 的理解，还让我们更好地理解了 cache 的控制流程，对 cache 在流水线中的使用有了更加清晰的认识。在实验过程中，我们也遇到了一些问题，如 stall 信号的设置条件等，但仔细看书和理解整个结构的设计之后，我们均找到了正确的答案。