浙江大學
ZHEJIANG UNIVERSITY

Lab 4 Pipelined CPU with Cache

2024-2025 春夏学期 计算机体系结构

课程实验报告

姓名 王浩雄

学号 3230106032

年级 2023 级

专业 混合班（计算机科学与技术）

班级 混合 2303 班

2025 年 4 月 15 日

Lab 4 Report

1 实验目的

本实验要求在 Lab3 实现的 Cache 基础上，进一步完成 Cache 控制单元（CMU）的设计，并将其整合进流水线 CPU 中。具体要求如下：

1. 理解 CMU 的基本原理以及其状态机的工作机制；
2. 掌握 CMU 状态机的设计思想与状态转换逻辑，重点理解其在缓存命中与未命中过程中的控制流程；
3. 掌握 CMU 的功能验证方法，通过仿真验证其正确性；
4. 掌握 CMU 的设计方法，将其完整集成到 CPU 系统中，实现结构清晰、功能完整的访存子系统。

2 CMU 模块接口说明

CMU 模块的接口，按照连接对象可分为以下三类：

1. **CPU 侧接口：**
   - `clk`：时钟信号。
   - `rst`：异步复位信号。
   - `addr_rw`：读/写地址。
   - `en_r`：读使能，表示 load 指令。
   - `en_w`：写使能，表示 store 指令。
   - `u_b_h_w`：数据类型控制信号（表示 byte/half/word 及符号信息等）。
   - `data_w`：写入 Memory 的数据。
   - `data_r`：从 Cache/Memory 读出的数据。
   - `stall`：当 CMU 正在处理访存时，通知流水线暂停操作。该信号传给 Hazard detection unit, 然后该 Unit 对流水线寄存器和 PC 寄存器进行控制，从而实现流水线停顿。
2. **内存侧接口：**
   - `mem_cs_o`：Memory 选择信号，高电平激活。

- `mem_we_o`：Memory 写使能，高电平表示写操作，低电平表示读操作。
- `mem_addr_o`：读写 Memory 的地址。
- `mem_data_o`：写入 Memory 的数据。
- `mem_data_i`：从 Memory 读回的数据。
- `mem_ack_i`：内存响应信号，表明本次操作完成。实验中 Cache 和 Memory 交互数据的时候是逐字发送的，每次读写成功一个字, Memory 都会给一个 ack 信号。

3. **调试输出接口：**

- `cmu_state`：输出当前状态机状态，用于调试工作。

3 状态机设计与切换逻辑

在 CMU 中，由于访存流程包含了多个相互依赖的操作。直接用组合逻辑难以清晰、可靠地描述各个操作步骤之间的时序关系和控制逻辑，且一些操作需要再多个时钟周期内完成。因此，采用有限状态机进行 CMU 设计。

该状态机的关键点在于**缓存操作与内存操作分布在不同时钟边沿上**。下面对逐个状态进行详细说明：

- **缓存操作**：发生在当前状态的下降沿。
- **内存操作**：发生在当前状态的上升沿。

3.1 S_IDLE（空闲状态）

- **含义**：系统处于空闲状态，不会有内存操作。
- **操作**：
  1. 如果缓存命中，则继续执行缓存操作，不需要访问内存，系统维持在该状态。
  2. 如果未命中，根据缓存是否有效及数据是否脏，决定进入下一状态。

Code Listing 1: S_IDLE 状态转移逻辑

```
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
```

3.2 S_PRE_BACK（准备回写状态）

- **含义**：进入此状态主要目的是为将脏数据回写到内存作准备。
- **操作**：在此状态的下降沿进行一次缓存读取操作，从缓存中取出需要回写的数据。

Code Listing 2: S_PRE_BACK 状态转移逻辑

```
S_PRE_BACK: begin
        next_state = S_BACK;
        next_word_count = 2'b00;
end
```

3.3 S_BACK（正在回写状态）

- **含义**：在该状态下，系统将脏数据写回到内存。
- **操作**：
  1. 在上升沿：将 S_PRE_BACK 状态中读取的数据写入内存。
  2. 在下降沿：从缓存中读取下一段要写回的数据，供下一次上升沿使用。
- **过程**：整个缓存行需要写回，因此该过程会持续多个周期。由于一次内存写操作需要 4 个周期，在继续下一步之前需要等待内存的确认信号 mem_ack_i。

Code Listing 3: S_BACK 状态转移逻辑

```
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
```

3.4 S_FILL（填充缓存状态）

- **含义**：当写回完成后，或者在未写回情况下，进入此状态从内存中读取数据填充缓存。
- **操作**：
  1. 在上升沿：从内存中读取数据。
  2. 在下降沿：将读取到的数据写入缓存。
- **过程**：与写回过程类似，为填满整个缓存行，该过程也需要多个周期，并等待内存确认信号 `mem_ack_i`。

Code Listing 4: S_FILL 状态转移逻辑

```
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
```

3.5 S_WAIT（等待状态）

- **含义**：针对之前 Cache miss 的情况，有些缓存操作可能未能完成，此状态用于执行这些剩余的操作。
- **操作**：仍然在下降沿执行缓存操作，待所有操作完成后返回空闲状态。

Code Listing 5: S_WAIT 状态转移逻辑

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

3.6 状态机状态更新逻辑

状态机状态和字计数器的更新在时钟上升沿进行。具体逻辑如下：

Code Listing 6: 状态更新逻辑

```
always @ (posedge clk) begin
        if (rst) begin
                state <= S_IDLE;
                word_count <= 2'b00;
        end
        else begin
                state <= next_state;
                word_count <= next_word_count;
        end
end
```

3.7 状态机切换逻辑总结

整个状态机切换流程可描述为：

1. **S_IDLE**： CPU 发出访存请求后，系统首先检测缓存命中情况：
   - 若命中，直接在 **S_IDLE** 状态完成操作。
   - 若未命中且缓存有效且数据脏，则进入 **S_PRE_BACK** 状态；
   - 否则直接进入 **S_FILL** 状态进行缓存填充。
2. **S_PRE_BACK**：准备回写，从缓存中读取出即将写回内存的数据，并初始化写回计数器，然后转入 **S_BACK** 状态。
3. **S_BACK**：在上升沿将从 **S_PRE_BACK** 获取的数据写回内存，在下降沿继续从缓存中读取下一段数据，整个过程持续多个周期，直至写回完毕后转入 **S_FILL** 状态。
4. **S_FILL**：在上升沿从内存中读取数据，下降沿将数据写入缓存，直到整个缓存行填充完毕后进入 **S_WAIT** 状态。

5. **S__WAIT**：处理因缓存未命中而留下的未完成缓存操作，最终返回 **S__IDLE** 状态等待下一个访存请求。

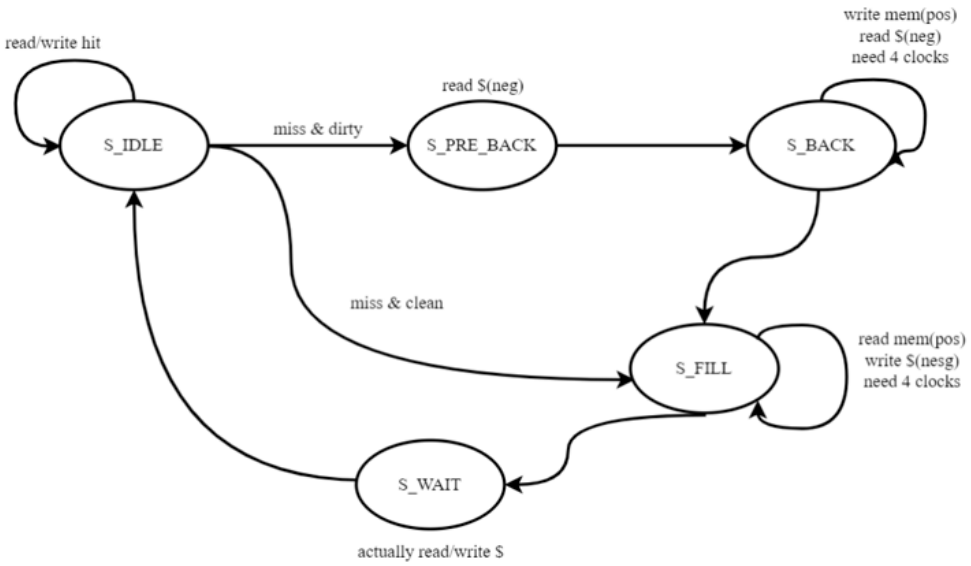


图 1: CMU 状态机切换流程图

4 实验结果

4.1 仿真测试点分析

通过仿真测试，我验证了 CMU 在不同操作要求、不同缓存状态下对缓存数据的获取和更新逻辑，确保了 CMU 控制逻辑正确实现。

使用的仿真激励文件如下：

```
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];
```

内存初始化内容如下：

```
00000000 00000004 00000008 0000000C
00000010 00000014 00000018 0000001C
00000020 00000024 00000028 0000002C
```

下面选取部分关键测试点进行详细说明：

4.1.1 Read Miss (Clean)

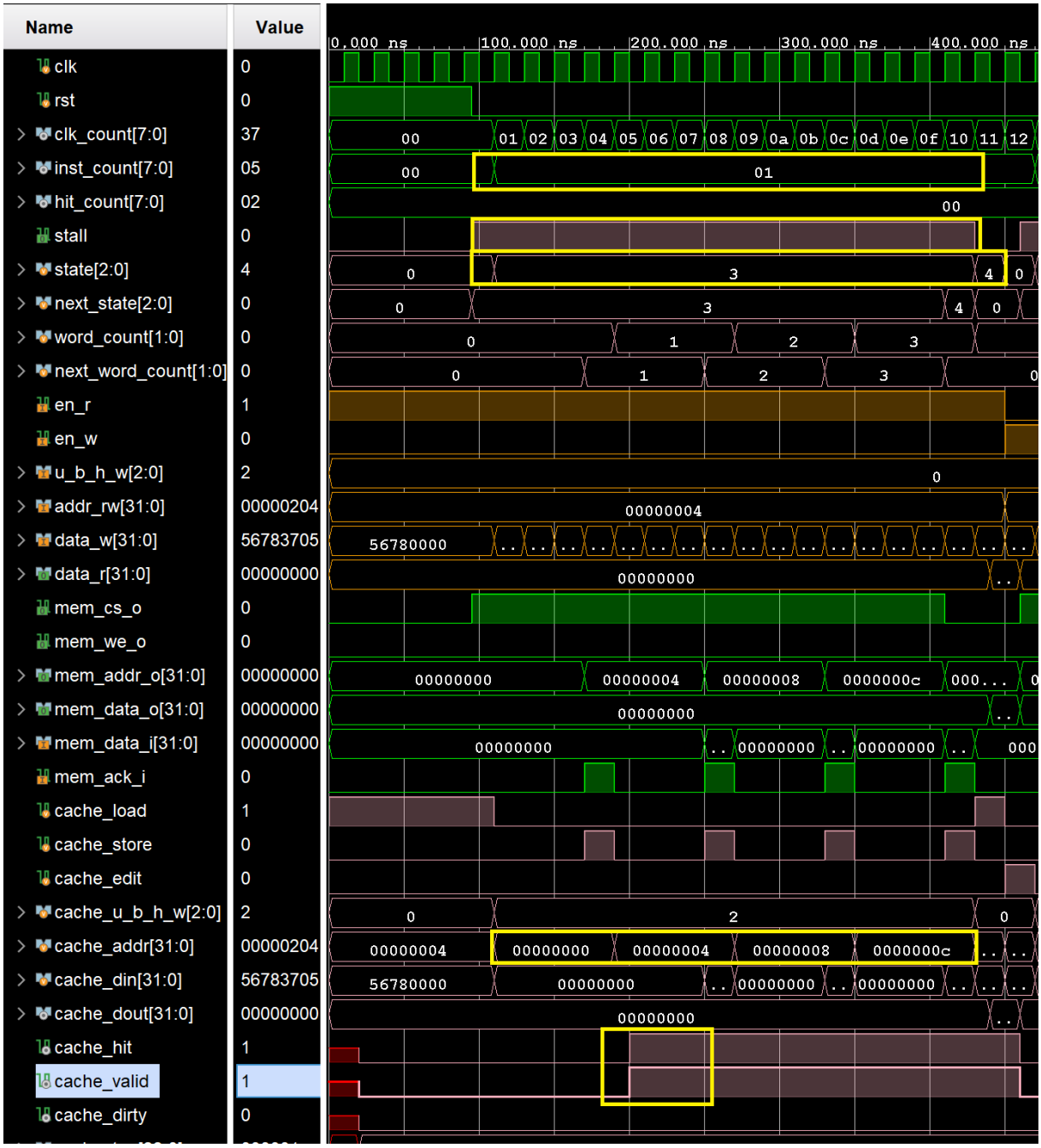


图 2: 指令号：0

指令：

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

解释：

1. 在发生读缺失的情况下，假设缓存中有一个可用且可替换的位置，状态转换如下：0 (S_IDLE) → 3 (S_FILL) → 4 (S_WAIT) → 0 (S_IDLE)。

2. 其中，在 S_FILL 阶段，由于内容是逐字读取的，因此可以观察到 `word_count` 由 0 自增到 3 的过程。

3. 此外，在 S_FILL 过程中，因为 CPU 需要地址 0x4 处的数据，缓存从内存中读取从 0x0 到 0xC 的块。在读取 0x4 处的数据后，`cache_hit` 和 `cache_valid` 变为 1。

4.1.2 Read and Write Hit

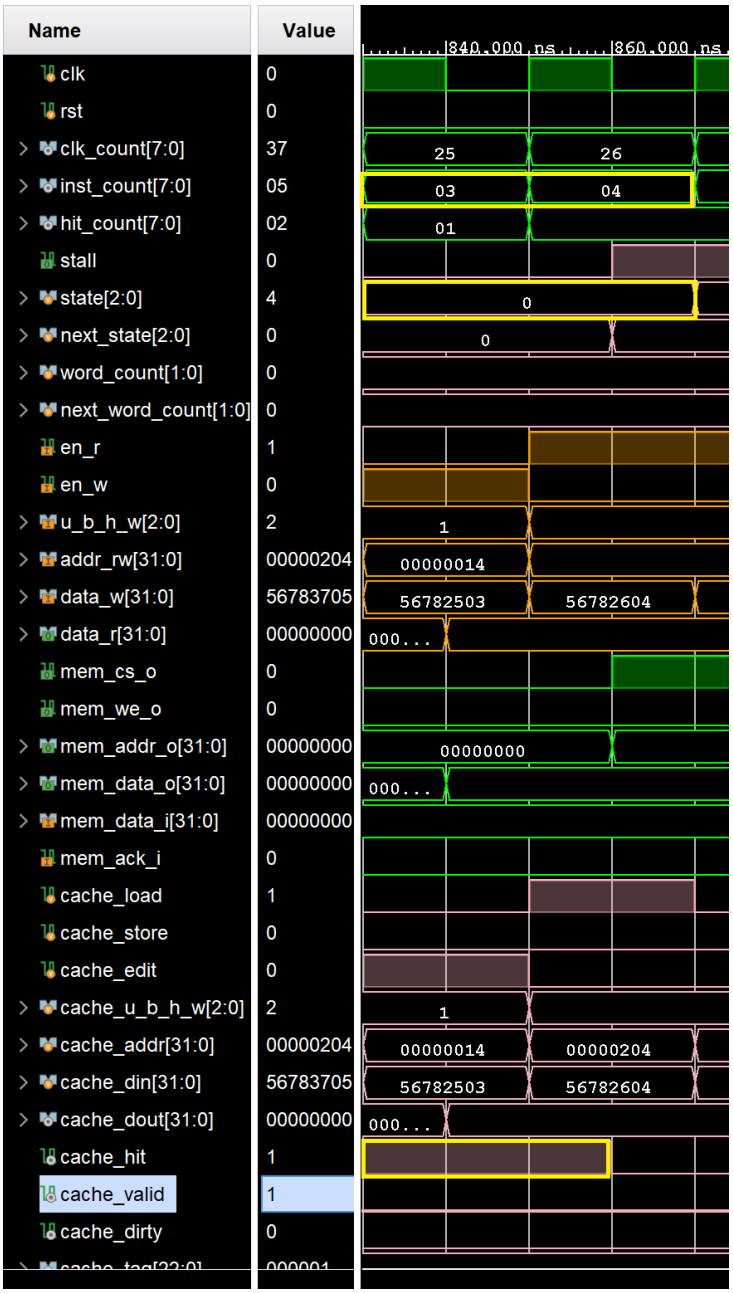


图 3: 指令号：2-3

指令：

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

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

解释：

在发生读命中和写命中时，状态机保持在 S_IDLE 状态，CPU 不会暂停。在此过程中，`stall` 为 0，`cache_hit` 为 1。

4.1.3 Read Miss (Dirty)

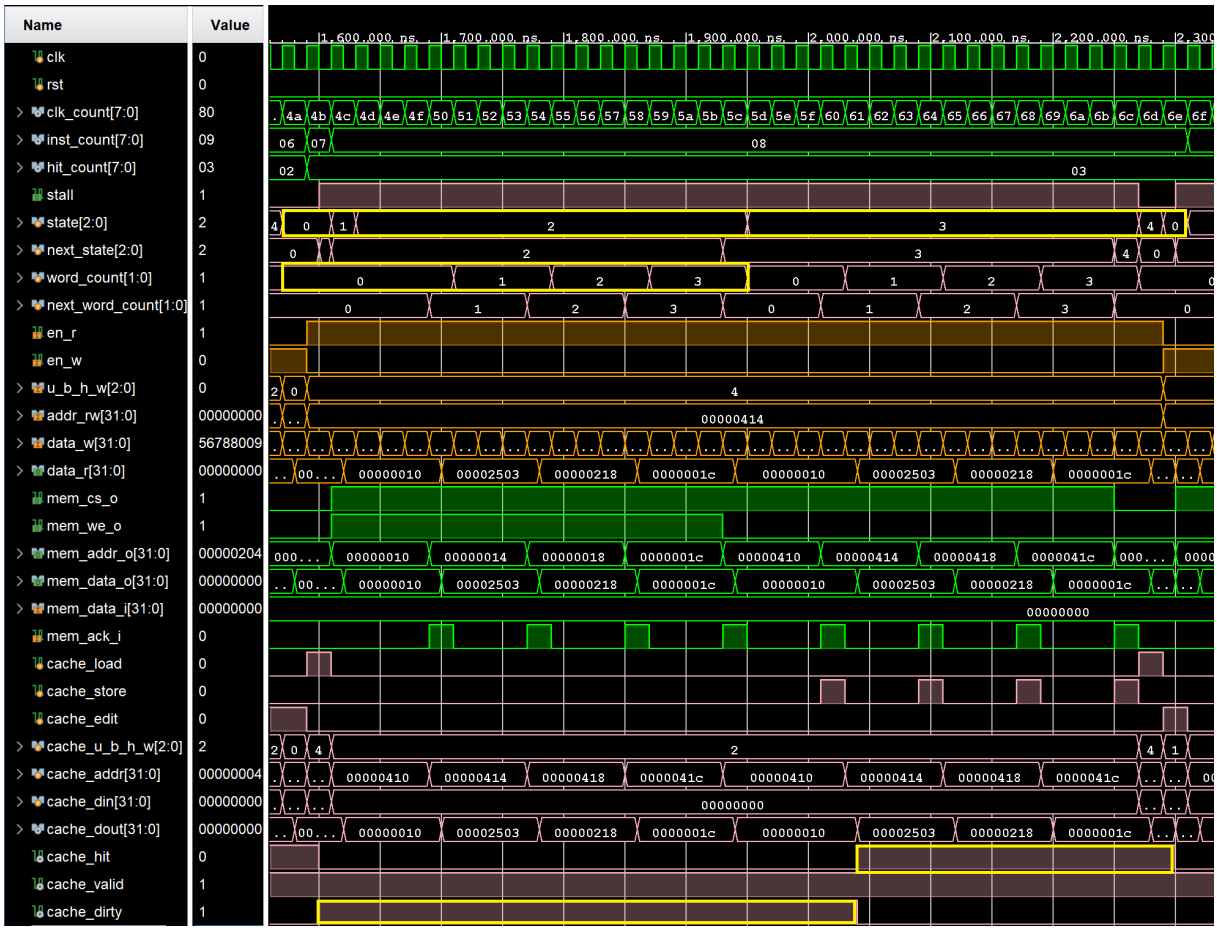


图 4: 指令号：7

指令：

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

解释：

1. 在发生读缺失且需要进行脏块替换时,状态转换如下:0 (S_IDLE) → 1 (S_PRE_BACK) → 2 (S_BACK) → 3 (S_FILL) → 4 (S_WAIT) → 0 (S_IDLE)。

2. 其中，在 S_BACK 和 S_FILL 两个阶段，由于内容是逐字读写的，因此可以观察到 `word_count` 由 0 自增到 3 的过程。

3. 在 S_BACK 阶段结束后，观察到 `cache_dirty` 信号变为 0，`cache_hit` 信号变为 1，这与设计逻辑相符。

4.2 上板测试结果分析

通过上板测试，我验证了 CMU 在不同操作要求、不同缓存状态下对缓存数据的获取和更新逻辑，确保了 CMU 控制逻辑正确实现。

ROM 初始化文件如下：

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
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	

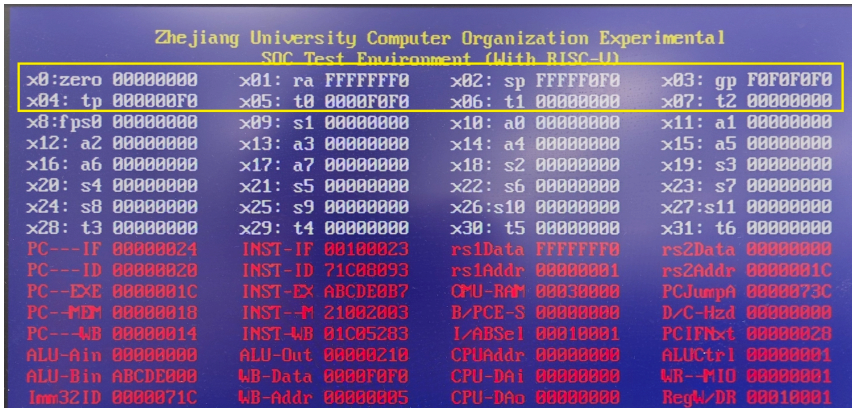
RAM 初始化内容如下：

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	

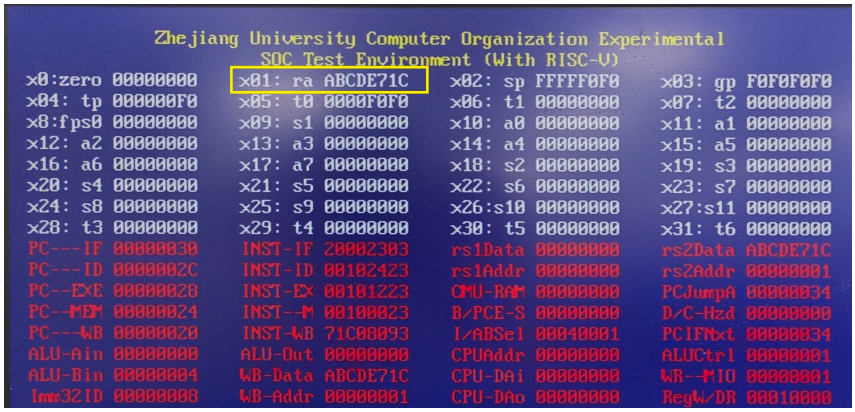
NO.	Instruction	Addr.	Comment
16	00000000	40	
17	00000000	44	
18	00000000	48	
19	00000000	4C	
20	A3000000	50	
21	27000000	54	
22	79000000	58	
23	15100000	5C	
24	00000000	60	
25	00000000	64	
26	00000000	68	
27	00000000	6C	
28	00000000	70	
29	00000000	74	
30	00000000	78	
31	00000000	7C	

上板测试结果如下，均与预期相符：

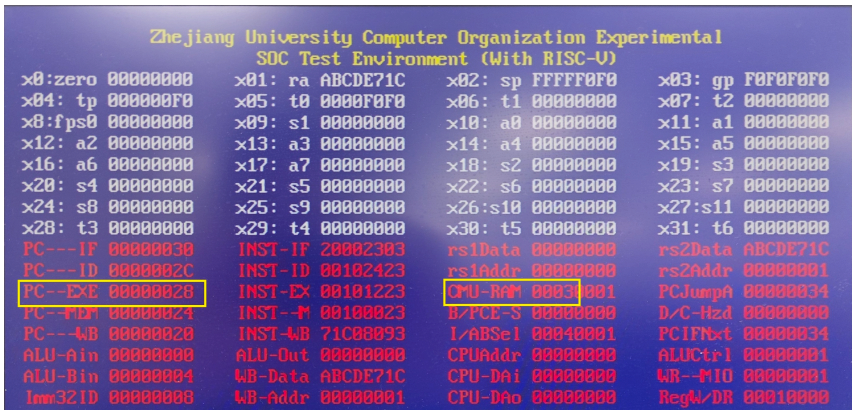
测试点 1：x1-x5 寄存器数据加载正确



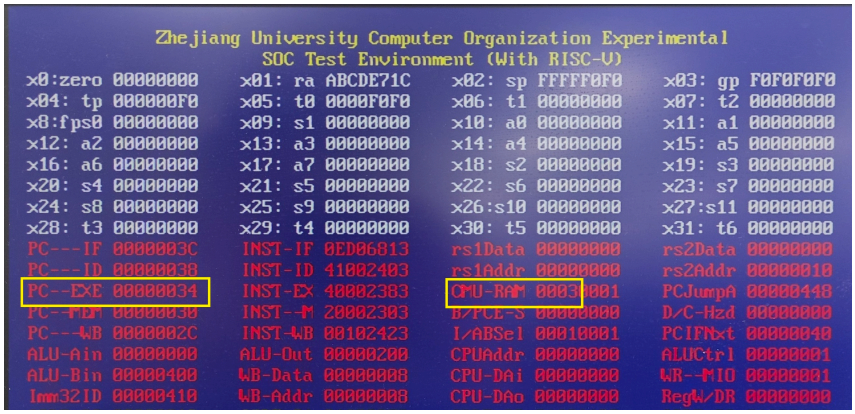
测试点 2：x1 寄存器的值计算正确 0xABCDE71C



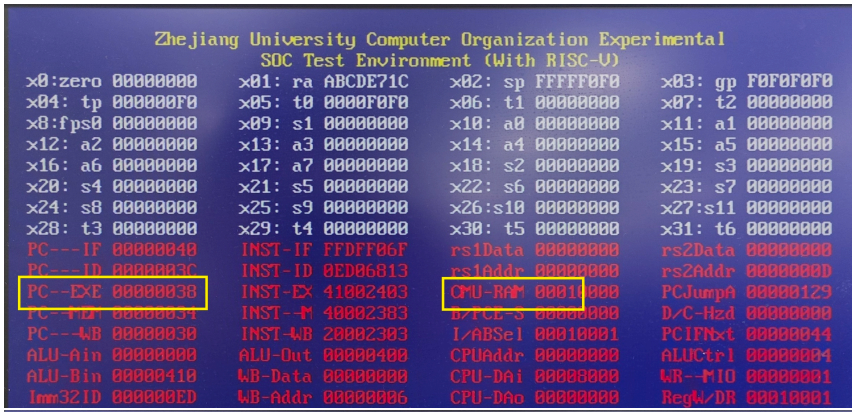
测试点 3：0x28 指令发生 Write Miss，进入状态机的 S_FILL 状态

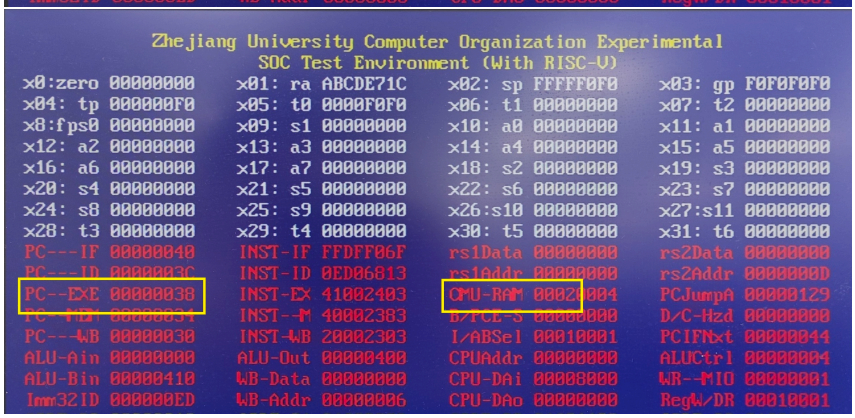


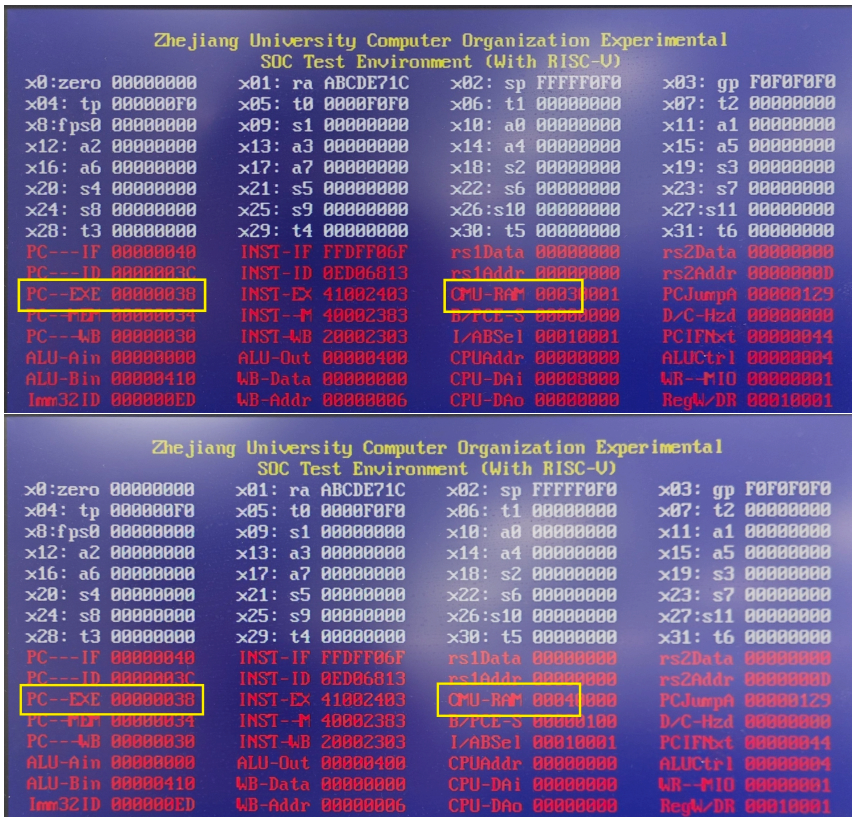
测试点 4：0x34 指令发生 Read Miss，进入状态机的 S_FILL 状态



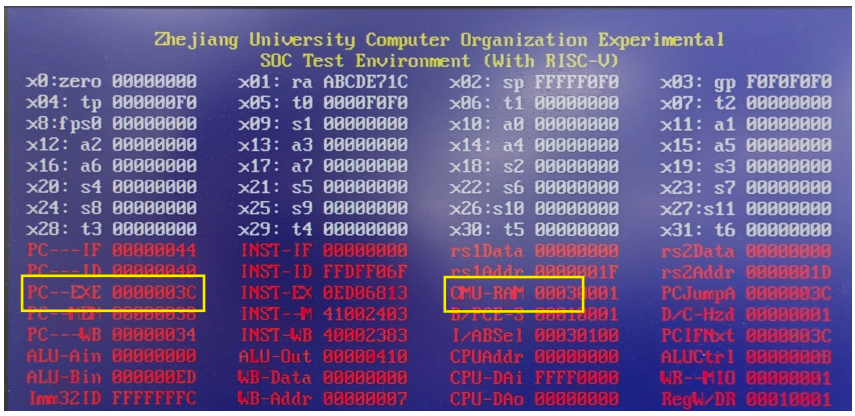
测试点 5：0x38 指令发生 Read Miss（Dirty），需先写回脏块再读取新块，依次经历状态机的所有状态



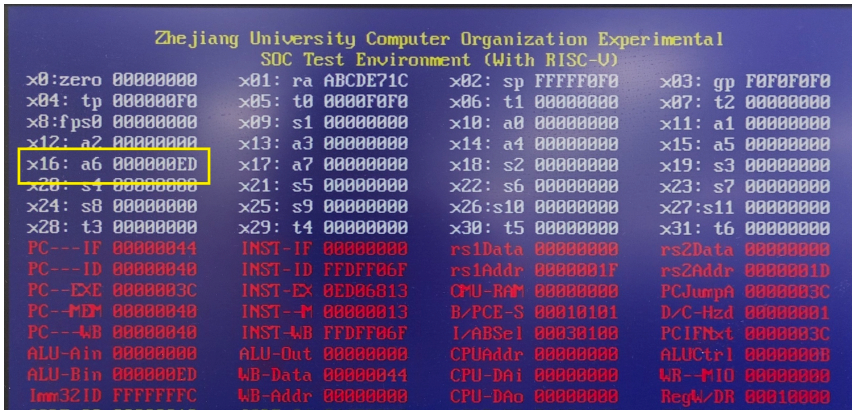




测试点 6：0x3C 指令发生 Read Miss，进入状态机的 S_FILL 状态



测试点 7：完成所有指令，x16 寄存器被设置为 0xED


Zhejiang University Computer Organization Experimental
SOC Test Environment (With RISC-V)
x0:zero 00000000 x01: ra ABCDE71C x02: sp FFFFF0F0 x03: gp F0F0F0F0
x04: tp 000000F0 x05: t0 0000F0F0 x06: t1 00000000 x07: t2 00000000
x08:fps0 00000000 x09: s1 00000000 x10: a0 00000000 x11: a1 00000000
x12: a2 00000000 x13: a3 00000000 x14: a4 00000000 x15: a5 00000000
x16: a6 000000ED x17: a7 00000000 x18: s2 00000000 x19: s3 00000000
x20: s4 00000000 x21: s5 00000000 x22: s6 00000000 x23: s7 00000000
x24: s8 00000000 x25: s9 00000000 x26:s10 00000000 x27:s11 00000000
x28: t3 00000000 x29: t4 00000000 x30: t5 00000000 x31: t6 00000000
PC---IF 00000044 INST-IF 00000000 rs1Data 00000000 rs2Data 00000000
PC---ID 00000040 INST-ID FFDFF06F rs1Addr 0000001F rs2Addr 0000001D
PC--EXE 0000003C INST-EX 0ED06813 CMU-RAM 00000000 PCJumpA 0000003C
PC--MEM 00000040 INST--M 00000013 B/PCE-S 00010101 D/C-Hzd 00000001
PC---WB 00000040 INST-WB FFDFF06F I/ABSel 00030100 PCIFNxt 0000003C
ALU-Ain 00000000 ALU-Out 00000000 CPUAddr 00000000 ALUCtrl 0000000B
ALU-Bin 000000ED WB-Data 00000044 CPU-DAi 00000000 WR--MIO 00000000
Imm32ID FFFFFFFC WB-Addr 00000000 CPU-DAo 00000000 RegW/DR 00010000