浙江大學
ZHEJIANG UNIVERSITY

Lab 2 Pipelined CPU Supporting Exception and Interrupt

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 3 月 16 日

Lab 2 Report

1 实验目的

- **理解异常与中断处理机制：**学习 RISC-V 特权等级、控制状态寄存器（CSR）、Trap 处理流程及其在流水线 CPU 中的实现。
- **掌握流水线 CPU 设计方法：**设计并实现 Exception Unit，管理 CSR 读写、中断异常处理及状态转换，实现支持异常和中断的 RISC-V 流水线 CPU。

2 实验内容与原理

本实验要求设计支持例外与中断处理的流水线 CPU，主要内容包括数据通路设计、CSR 寄存器与指令支持、以及异常与中断处理单元的实现。主要原理如下：

2.1 CSR 指令与 CSR 寄存器

CSR（Control Status Register）用于管理 CPU 状态和特权级。实验中涉及的主要寄存器及其地址包括：

- **mstatus**（地址 0x300）：保存中断使能、特权级、前一个中断使能（MPIE）等状态位。
- **mtvec**（地址 0x305）：Trap 入口地址，决定异常或中断发生后 CPU 跳转的位置。
- **mepc**（地址 0x341）：机器异常程序计数器，保存引起异常的指令地址（例外时）或下一条指令地址（中断时）。
- **mcause**（地址 0x342）：记录异常或中断的原因；中断时最高位为 1，例外时为 0。
- **mtval**（地址 0x343）：保存出错地址或非法指令。

常见 CSR 指令包括 `csrrw`、`csrrs`、`csrrc`，通过这些指令实现对 CSR 寄存器的读写和状态控制。

2.2 异常与中断处理流程

当 CPU 运行过程中检测到异常或中断时，将通过 Trap 机制进入异常处理流程。主要步骤如下：

1. **Trap 进入：**

- **保存状态**：将当前异常指令的 PC 保存至 `mepc`（例外）或下一条 PC（中断），并将异常原因写入 `mcause`。同时，更新 `mstatus`，将当前中断使能位 MIE 清零，并保存原始状态到 MPIE 字段；同时保存原特权级到 MPP 域。
- **跳转**：根据 `mtvec` 指定的 Trap 入口地址重定向 PC，开始异常处理程序。

2. **Trap 处理**：在 Trap Handler 中，根据 `mcause` 判断异常类型或中断类型，进行相应处理。

3. **Trap 退出**：执行 `mret` 指令恢复 `mstatus`，并将 `mepc` 中保存的地址送入 PC，继续程序执行。

异常与中断的区别如下：

1. **异常（Exception）**：由指令内部错误（如非法指令、访存错误、ECALL）同步触发，`mepc` 保存异常指令地址。

2. **中断（Interrupt）**：由外部设备或定时器异步触发，`mepc` 保存下一条指令地址。

实验中采用有限状态机实现异常与中断处理，其状态转移主要有：

1. **STATE_IDLE**：正常状态，检测异常或中断信号。

2. **STATE_MEPC**：保存 PC 到 `mepc`（根据异常或中断类型选择保存当前或下一条 PC）。

3. **STATE_MCAUSE**：保存异常/中断原因到 `mcause`。

此外，当检测到 `mret` 信号时，恢复 `mstatus` 和 PC（读取 `mepc`），同时刷新流水线寄存器，确保精确异常处理。

3 实验设计

本实验的设计包括以下几个部分：

3.1 输入与输出信号设计

3.1.1 输入信号

- **CSR 信号**：包含 CSR 的读写使能、读写地址、写数据（或立即数）以及写入模式（写、置位、清除）。
- **异常/中断检测信号**：如 `illegal_inst`（非法指令）、`l_access_fault`（加载访问错误）、`s_access_fault`（存储访问错误）、`ecall_m`（ECALL 异常）、以及 `interrupt` 信号。

- **恢复信号：**`mret` 指令，用于异常处理后退出 Trap。
- **PC 信息：**当前 PC (`epc_cur`) 和下一条 PC (`epc_next`)，用于记录异常或中断发生时的返回地址。

3.1.2 输出信号

- **PC 跳转控制：**包括跳转地址 (`PC_redirect`) 和多路选择信号 (`redirect_mux`)，用于选择 Trap 入口或异常返回地址。
- **流水线控制信号：**包括各级流水线寄存器 Flush 信号（`reg_FD_flush`、`reg_DE_flush`、`reg_EM_flush`、`reg_MW_flush`）以及取消写寄存器信号 (`RegWrite_cancel`)。
- **CSR 数据输出：**通过 `csr_r_data_out` 输出当前 CSR 数据。

3.2 状态机设计与控制逻辑

内部状态机采用三状态结构：

- **STATE_IDLE：**在此状态下，检测异常或中断信号，并根据检测结果决定是否触发 Trap 流程。同时处理普通 CSR 操作。
- **STATE_MEPC：**在该状态下，将选定的 PC（例外为当前 PC，中断为下一条 PC）保存到 `mepc` 寄存器，同时刷新流水线。
- **STATE_MCAUSE：**将组合好的异常或中断原因写入 `mcause` 寄存器，完成 Trap 进入前的必要更新。

3.3 流水线寄存器 Stall 与 Flush 控制

为保证精确异常处理：

- 异常或中断发生时，必须刷新流水线中未提交的指令，防止错误状态传播；
- 在 mret 指令触发时，也需要刷新部分流水线寄存器以恢复正常执行状态；
- 同时，在异常处理过程中可能需要取消 WB 阶段的写寄存器操作，防止错误数据写入。

4 代码实现

```
module ExceptionUnit(
input clk, rst,

input csr_rw_in,                    //判断是否是CSR的读写指令
input[1:0] csr_wsc_mode_in,         //CSR的写入方式
input csr_w_imm_mux,                //CSR的写入数据来源
input[11:0] csr_rw_addr_in,         //CSR的读写地址
input[31:0] csr_w_data_reg,         //CSR的写入数据
input[4:0] csr_w_data_imm,          //CSR的写入立即数
output[31:0] csr_r_data_out,        //CSR的读出数据

input interrupt,                    //判断例外或中断的种类
input illegal_inst,
input l_access_fault,
input s_access_fault,
input ecall_m,

input mret,                         //判断是否是mret指令

input[31:0] epc_cur,                //当前PC和下一条指令的PC，用于写入CSR
input[31:0] epc_next,

output[31:0] PC_redirect,           //控制PC的跳转，包括多选器的控制信号和跳转地址
output redirect_mux,
output reg_FD_flush, reg_DE_flush, reg_EM_flush, reg_MW_flush,  //控制流水线寄存器的清空
output RegWrite_cancel              //对于例外，取消WB阶段的写寄存器
);

reg csr_w;
reg[11:0] csr_raddr;     //读地址
reg[11:0] csr_waddr;     //写地址
reg[31:0] csr_wdata;     //写数据
wire[31:0] mstatus;      //CSR寄存器中保存的状态信息
reg[1:0] csr_wsc;        //写入方式


CSRRegs csr(.clk(clk),.rst(rst),
```

```
.csr_w(csr_w),           //写使能
.raddr(csr_raddr),       //读地址
.waddr(csr_waddr),       //写地址
.wdata(csr_wdata),       //写数据
.rdata(csr_r_data_out), //读数据
.mstatus(mstatus),       //CSR寄存器中保存的状态信息
.csr_wsc_mode(csr_wsc)  //写入方式
);

//According to the diagram, design the Exception Unit
localparam STATE_IDLE = 2'b00;
localparam STATE_MEPC = 2'b01;
localparam STATE_MCAUSE = 2'b10;

reg[1:0]  cur_state, next_state;
reg[31:0] epc;
reg[31:0] mcause;

reg reg_FD_flush_control = 0;
reg reg_DE_flush_control = 0;
reg reg_EM_flush_control = 0;
reg reg_MW_flush_control = 0;
reg RegWrite_cancel_control = 0;

always @(posedge clk or posedge rst) begin
if (rst) begin
cur_state <= STATE_IDLE;
epc <= 32'd0;
mcause <= 32'd0;
csr_raddr<=0;     //读地址
csr_waddr<=0;     //写地址
csr_wdata<=0;     //写数据
csr_wsc<=0;
end else begin
cur_state <= next_state;
end
end

always @* begin
//默认值
```

```
78 csr_w = 0;
79 reg_FD_flush_control = 0;
80 reg_DE_flush_control = 0;
81 reg_EM_flush_control = 0;
82 reg_MW_flush_control = 0;
83 RegWrite_cancel_control = 0;
84
85 case(cur_state)
86 STATE_IDLE:begin
87 if(interrupt|illegal_inst|l_access_fault|s_access_fault|ecall_m)
88 begin                    //中断
89 csr_w = 1;               //更新mstatus寄存器
90 csr_waddr = 12'h300;
91 csr_wdata = {mstatus[31:8],mstatus[3],mstatus[6:4],1'b0,mstatus
       [2:0]};
92 // 刷新流水线
93 reg_FD_flush_control = 1;
94 reg_DE_flush_control = 1;
95 reg_EM_flush_control = 1;
96 reg_MW_flush_control = 1;
97 if(~interrupt)RegWrite_cancel_control = 1;
98 epc = (interrupt ? epc_next : epc_cur);
99 mcause= (interrupt ? 32'h80000000 : 0) |
100 (illegal_inst ? 32'h2 : 0) |
101 (l_access_fault ? 32'h5 : 0) |
102 (s_access_fault ? 32'h7 : 0) |
103 (ecall_m ? 32'hb : 0);
104
105 next_state = STATE_MEPC;
106
107 end
108 else if(mret) begin              //mret指令，用于从例外中恢复
109 csr_w = 1;                       //恢复mstatus寄存器
110 csr_waddr = 12'h300;
111 csr_wdata = {mstatus[31:8],1'b1,mstatus[6:4],mstatus[7],mstatus
       [2:0]};
112 csr_raddr = 12'h341;             //恢复PC
113 reg_EM_flush_control = 1;
114 reg_DE_flush_control = 1;
115 reg_FD_flush_control = 1;
```

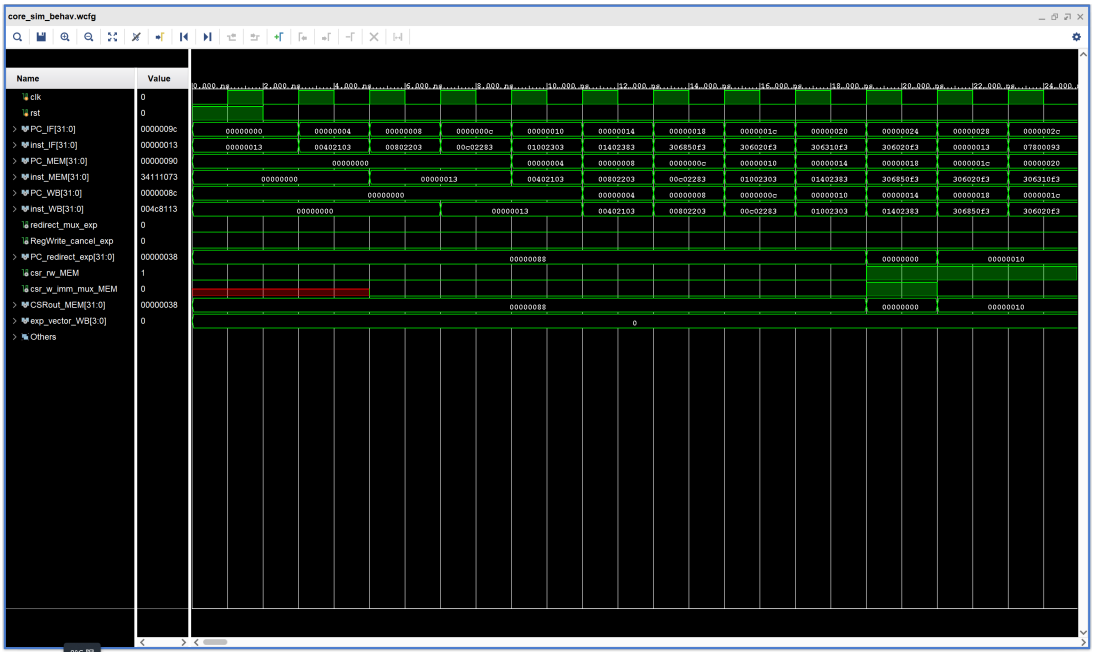
```
116 next_state = STATE_IDLE;
117 end
118 else if(csr_rw_in) begin    //CSR读写指令
119 csr_wsc = csr_wsc_mode_in;
120 csr_w = 1;
121 csr_raddr = csr_rw_addr_in;
122 csr_waddr = csr_rw_addr_in;
123 if(csr_w_imm_mux) begin
124 csr_wdata = csr_w_data_imm;
125 end
126 else begin
127 csr_wdata = csr_w_data_reg;
128 end
129 next_state = STATE_IDLE;
130 end
131 else begin                  //其他一般情况
132 csr_w = 0;
133 next_state = STATE_IDLE;
134 end
135 end
136 STATE_MEPC:begin
137 csr_w = 1;
138 csr_waddr = 12'h341;
139 csr_wdata = epc;
140 csr_raddr = 12'h305;
141 reg_FD_flush_control = 1;
142 next_state = STATE_MCAUSE;
143 end
144 STATE_MCAUSE:begin
145 csr_w = 1;
146 csr_waddr = 12'h342;
147 csr_wdata = mcause;
148 next_state = STATE_IDLE;
149 end
150 endcase
151 end
152
153 assign PC_redirect = csr_r_data_out;
154 assign redirect_mux = mret||(cur_state==STATE_MEPC);
155
```

```
156 assign reg_FD_flush = reg_FD_flush_control;
157 assign reg_DE_flush = reg_DE_flush_control;
158 assign reg_EM_flush = reg_EM_flush_control;
159 assign reg_MW_flush = reg_MW_flush_control;
160 assign RegWrite_cancel = RegWrite_cancel_control;
161
162 endmodule
```

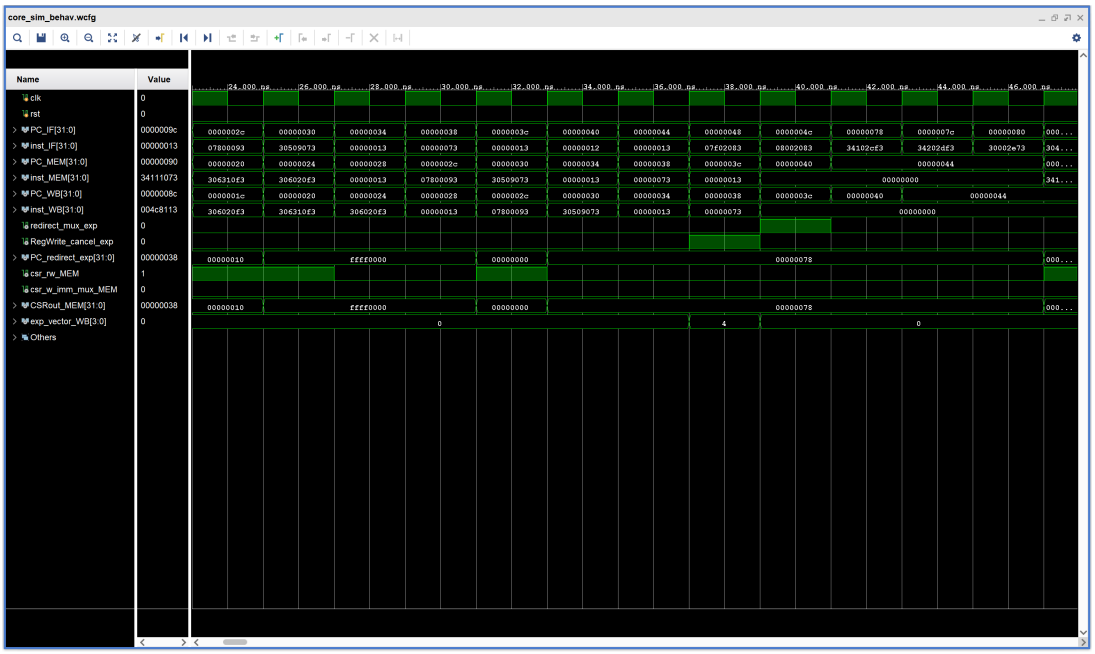
5 实验结果

5.1 仿真测试结果

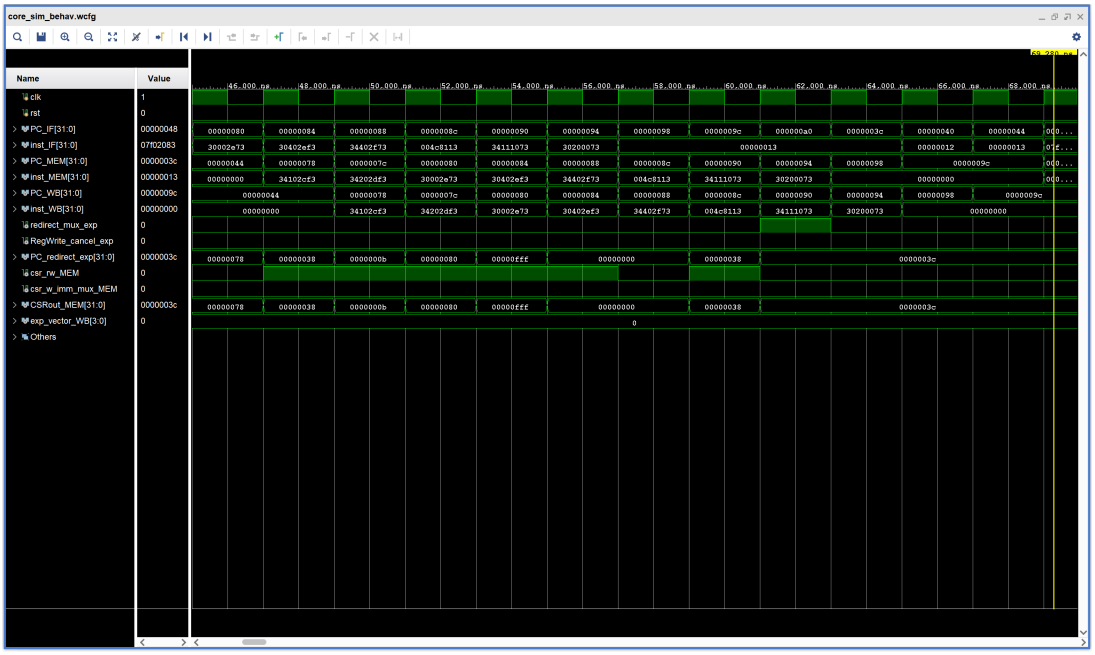
5.1.1 仿真波形



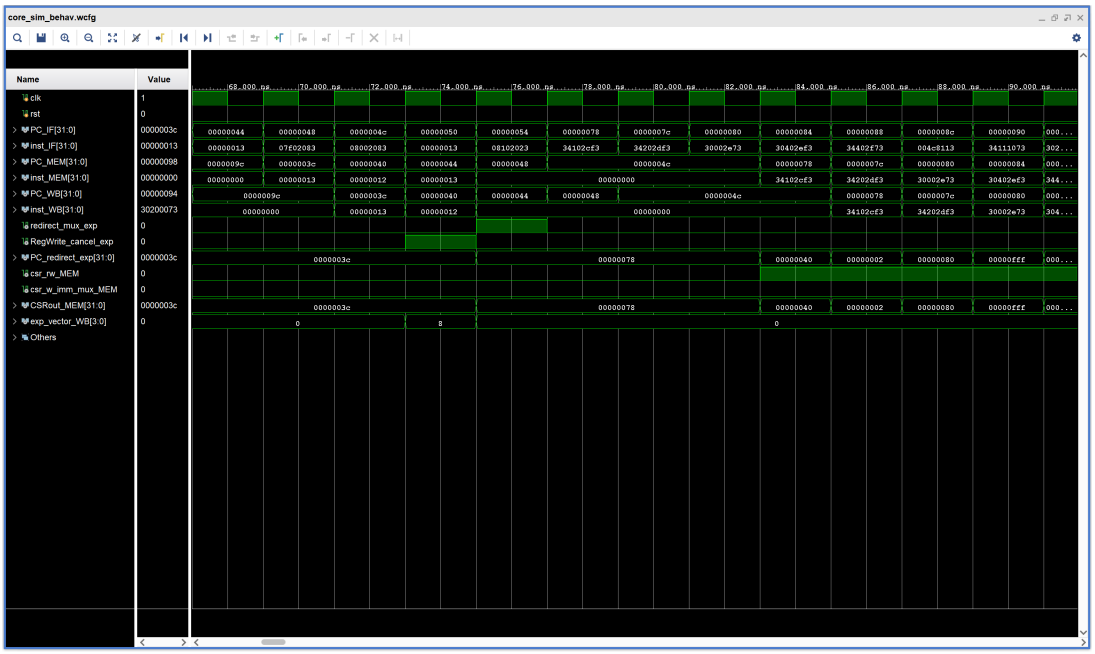
PC：0x0-0x2C



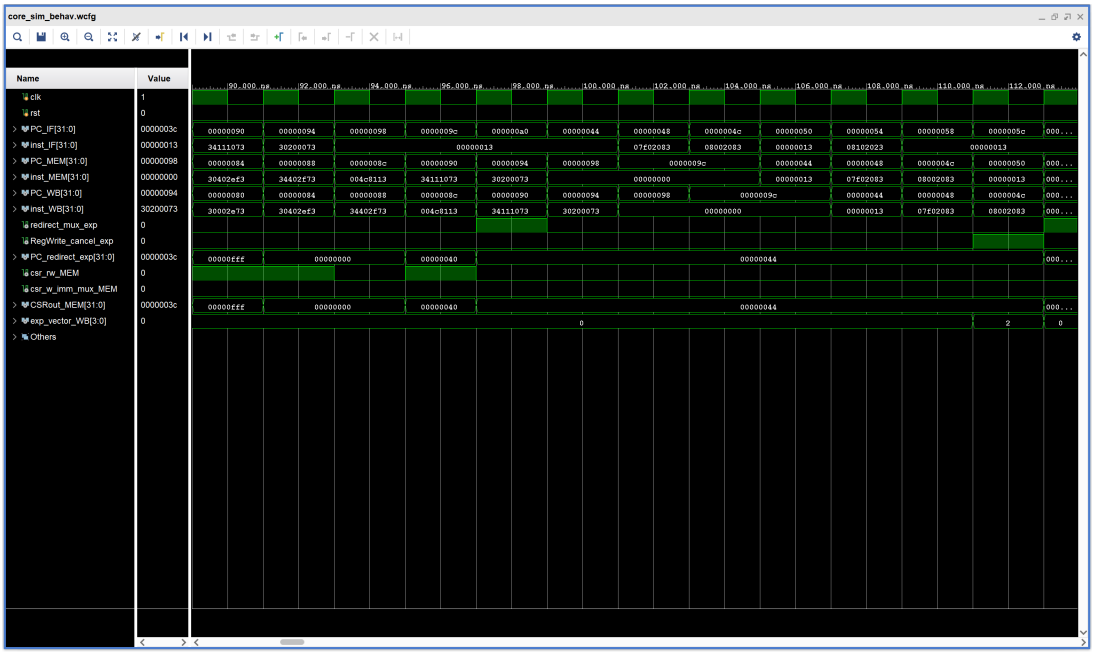
PC：0x2C-0x80



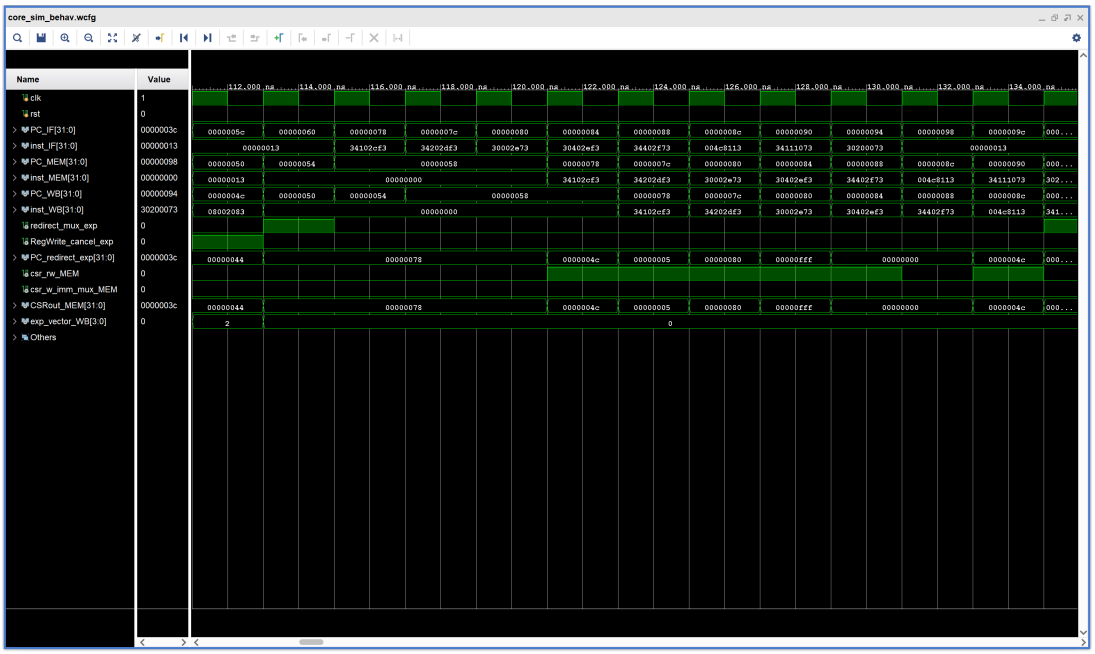
PC：0x80-0x44



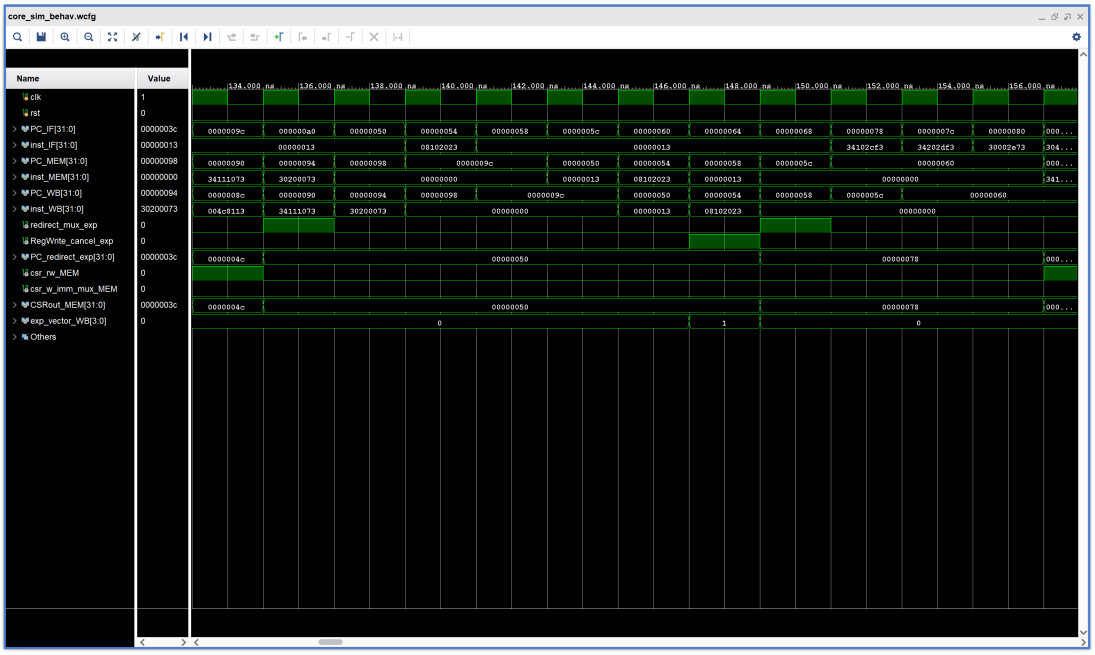
PC：0x44-0x90



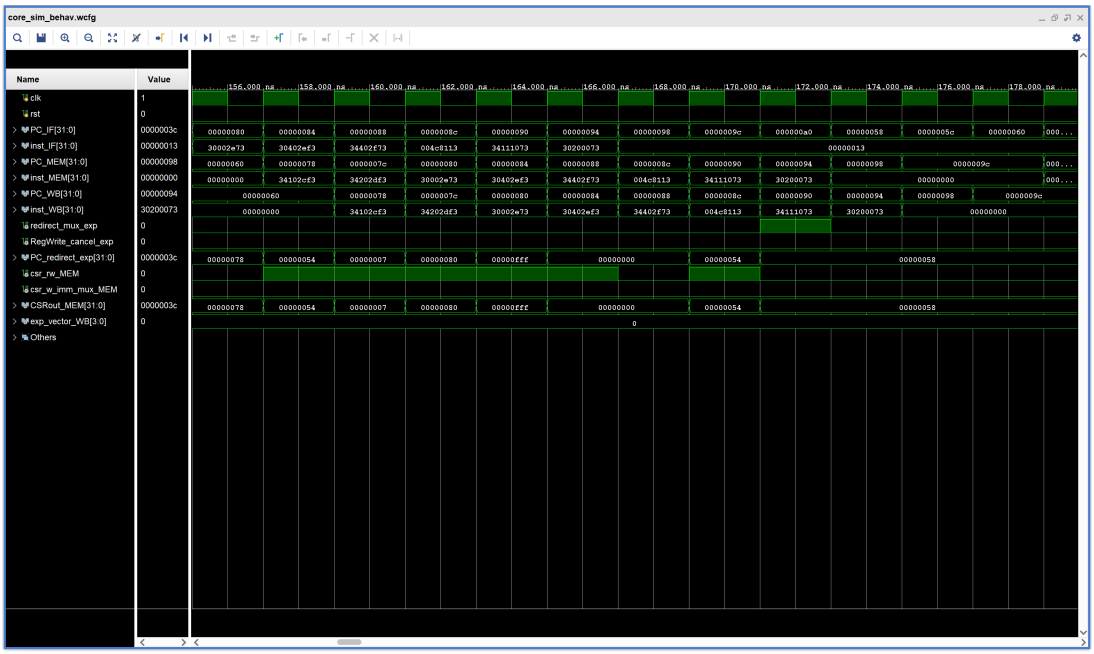
PC：0x90-0x5C



PC：0x5C-0x9C



PC：0x9C-0x80



PC：0x80-0x60

5.1.2 测试点 1

在 ROM 代码中，0x30 和 0x38 处的指令是异常处理的重要触发点，以下是详细分析：

```
0x30: csrw x1, 0x305, x1    # 更新 mtvec 寄存器的值
0x34: addi  x1, x1, 4       # x1 = x1 + 4
0x38: ecall                 # 触发 Environment Call
```

指令解析：

- **0x30 处指令：**指令为 csrw 0x305, x1。该指令将寄存器 x1 的值写入 CSR，目标地址为 0x305，对应 mtvec 寄存器。这一步骤设置了 Trap 处理程序的入口地址，即当异常发生时，CPU 将从 mtvec 指定的地址开始执行 Trap Handler。
- **0x38 处指令：**指令为 ecall，该指令触发环境调用异常。在执行 ecall 后，CPU 将进入 Trap 处理流程，保存当前的异常上下文，并将 PC 跳转至 mtvec 指定的入口地址 0x78。

测试结论：

仿真结果表明，当 0x38 的 ecall 触发异常时，CPU 能正确跳转到由 mtvec（即 x1 在 0x30 处写入的值）指定的 Trap Handler 地址 0x78，符合设计预期。

5.1.3 测试点 2

在 ROM 代码中，0x94 处的 mret 指令用于退出 Trap 模式，恢复异常发生前的状态，并跳转回正常执行流。与 mret 指令配套的，还有前面几条恢复关键寄存器状态的

指令。下面对这些指令进行详细解析：

```
0x78: csrr x25, 0x341
0x8C: addi x2, x25, 4
0x90: csrw 0x341, x2    # 将 mepc 的值加 4
0x94: mret              # 从 Trap 模式退出，跳转到 mepc 存储的地址
```

指令解析：

- **0x90 处指令：**指令为 `csrw 0x341, x2`，与 0x78/0x8C 处的指令相配合，使得 `mepc` 寄存器的值自增 4。经过这一设计，由异常触发的 Trap 模式将跳转回错误指令的下一条指令处，而由中断触发的 Trap 模式将跳转回错误指令的下两条指令处。
- **0x94 处指令：**指令为 `mret`，用于退出 Trap 模式。执行 `mret` 后，CPU 会从 `mepc` 寄存器中读取返回地址，并将该地址加载到 PC 中，使 CPU 跳转回异常发生前的正常指令流。

测试结论：

仿真和上板测试结果表明，当在 0x94 处执行 `mret` 指令时，CPU 能够正确退出 Trap 模式，并跳转到当时 `mepc` 中保存的返回地址。整个恢复过程确保了异常处理后系统状态精确恢复，并顺利继续执行后续指令，符合设计预期。

5.1.4 测试点 3

在 ROM 代码中，`0x40` 处的指令为非法指令，从而触发异常。以下是详细分析：

```
0x40: addi x0, x0, 0x12    # 指令编码错误，导致非法指令异常
0x78: csrr x25, 0x341      # Trap Handler 入口处
```

指令解析：

- **0x40 处指令：**这条指令为非法指令，从而触发异常。触发异常后，CPU 将当前 PC（即 0x40）存入 `mepc`，并设置 `mcause` 为对应的异常代码。
- **异常处理流程：**在检测到非法指令异常后，流水线中的所有寄存器将被 Flush，防止错误指令影响后续操作。随后，CPU 根据 `mtvec` 寄存器中设置的 Trap Handler 入口地址跳转至异常处理程序，即跳转到 0x78 处开始执行 Trap Handler。

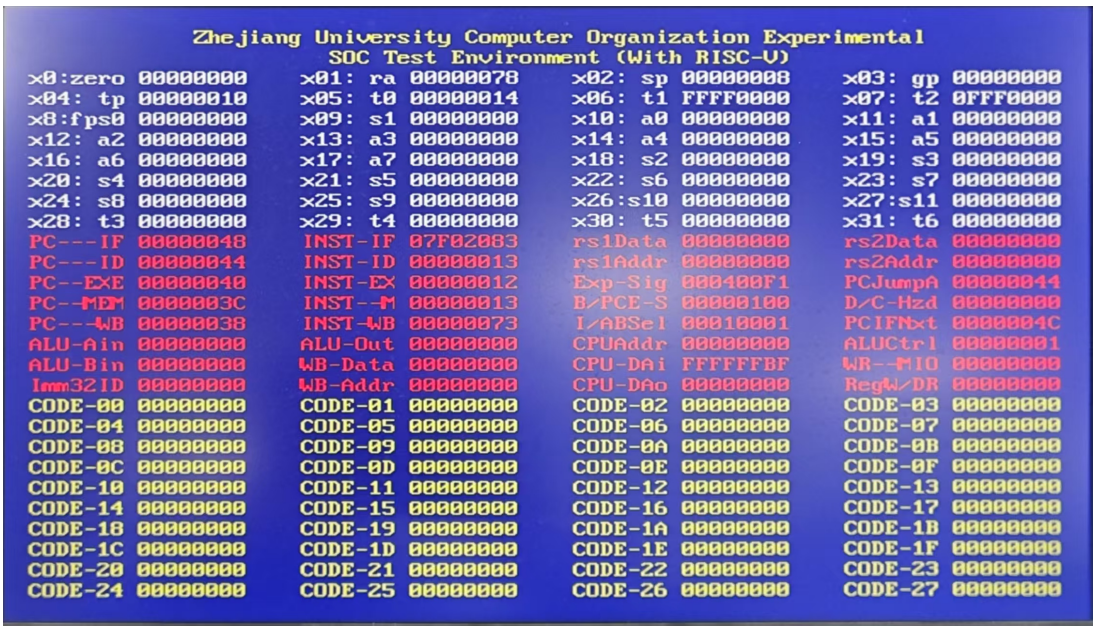
测试结论：

仿真和上板测试结果表明，当 0x40 处出现非法指令时，CPU 能正确检测到异常，并跳转到 Trap Handler 地址 0x78，符合设计预期。

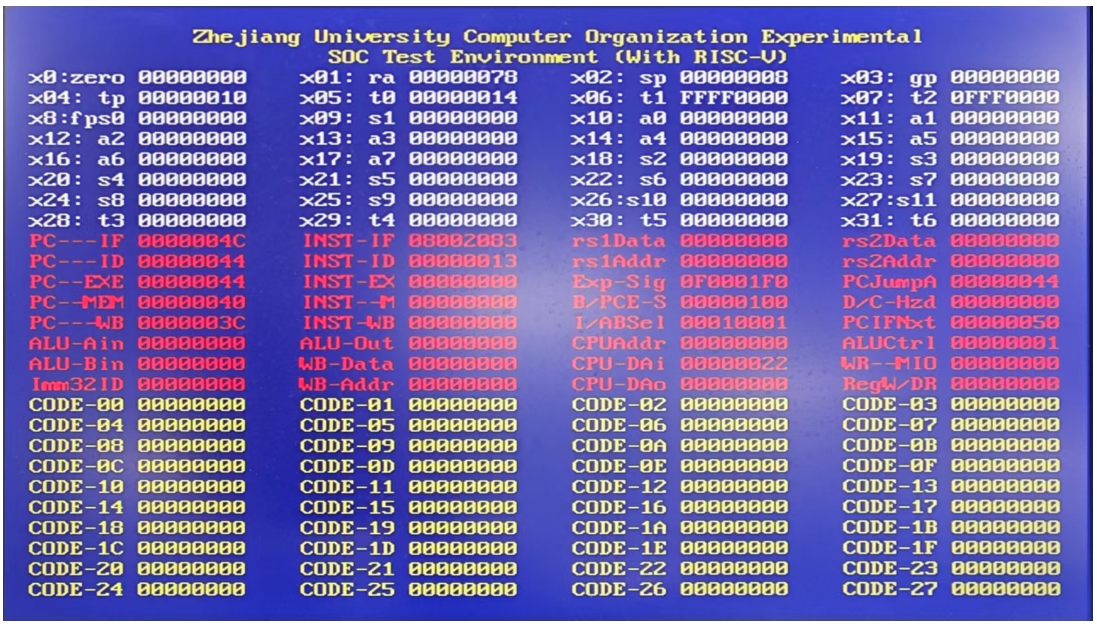
5.2 上板测试结果

5.2.1 测试点 1

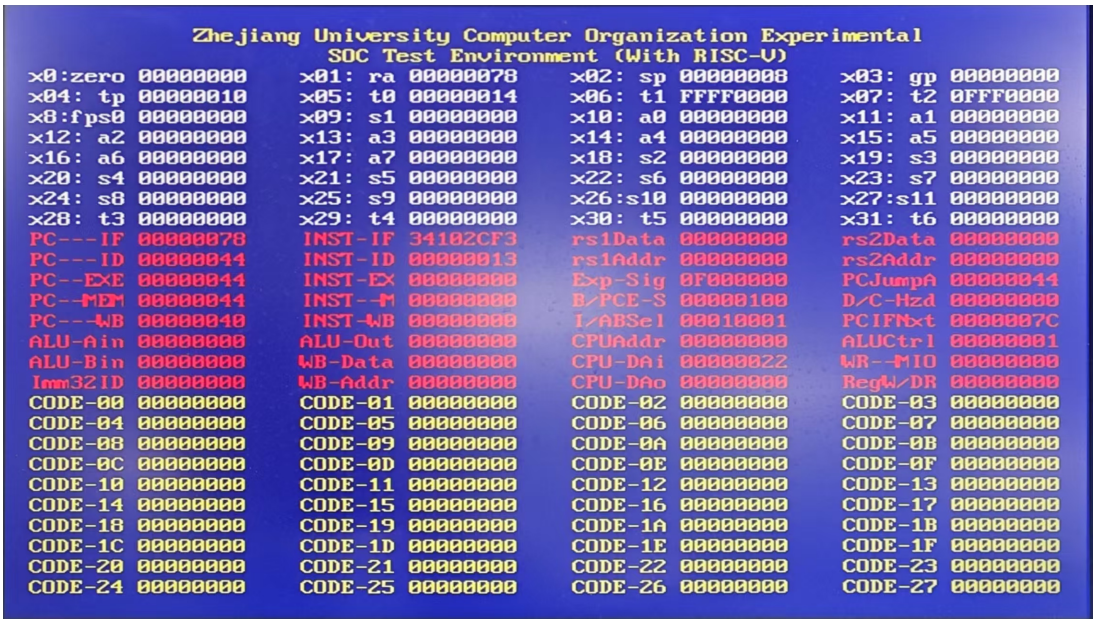
下述照片节选了指令 0x48-0x78 的上板表现。结合前文分析，上板测试结果与预期和仿真结果相符。



PC_IF：0x48



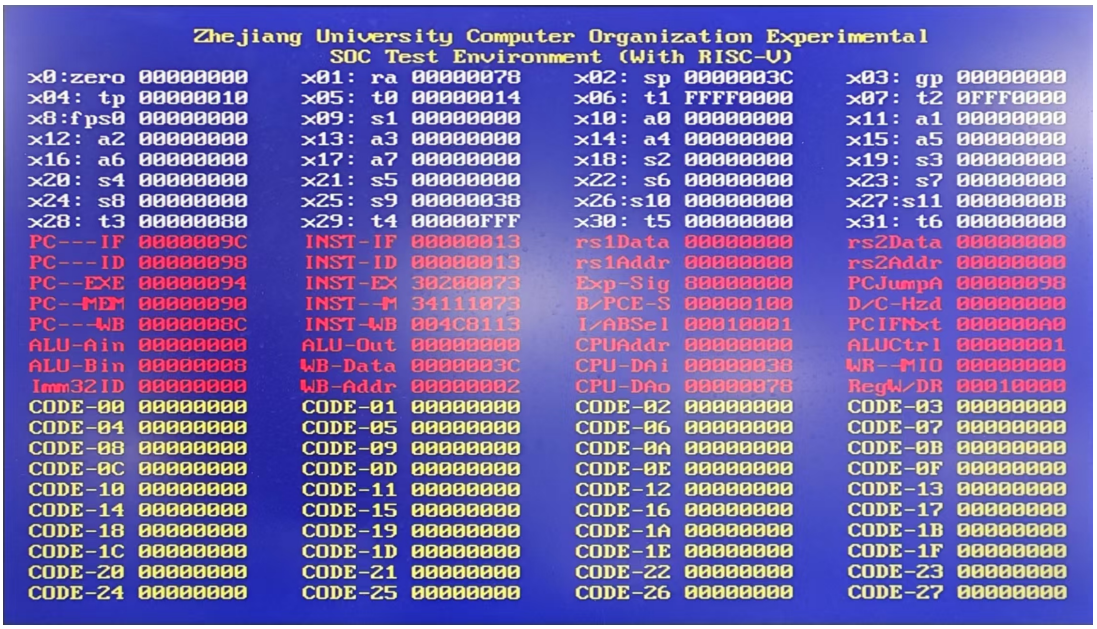
PC_IF：0x4C



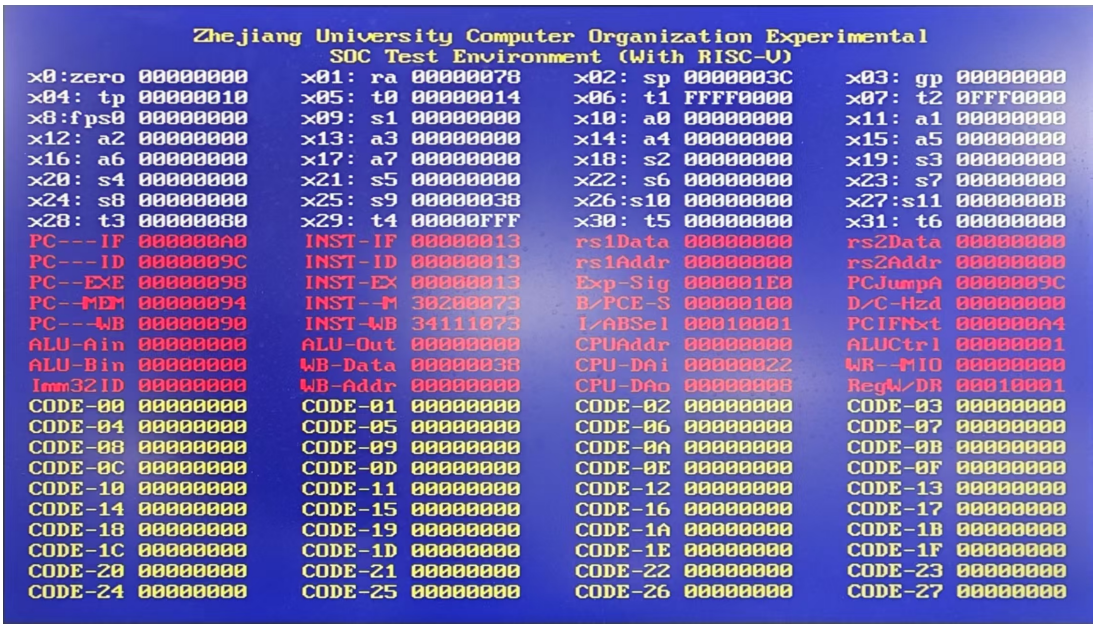
PC_IF：0x78

5.2.2 测试点 2

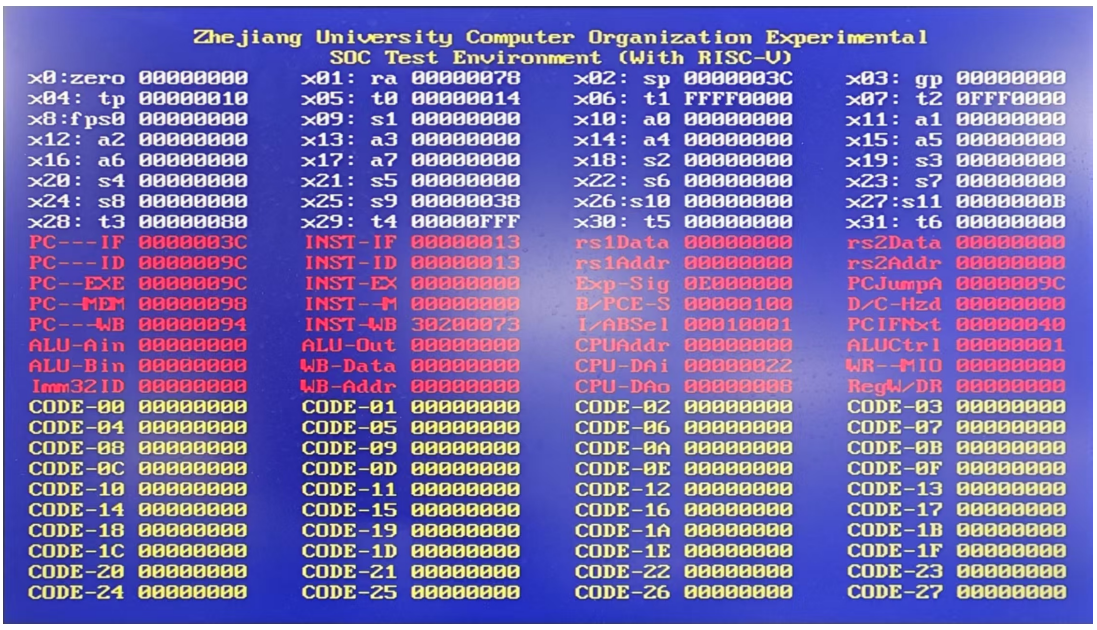
下述照片节选了指令 0x9C-0x3C 的上板表现。结合前文分析，上板测试结果与预期和仿真结果相符。



PC_IF：0x9C



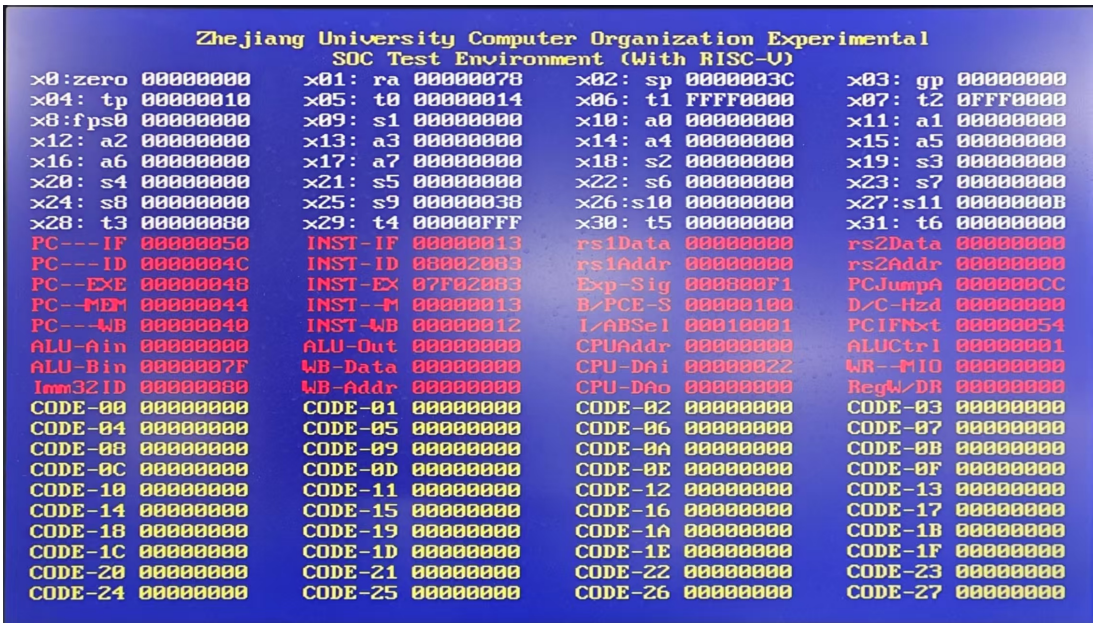
PC_IF：0xA0



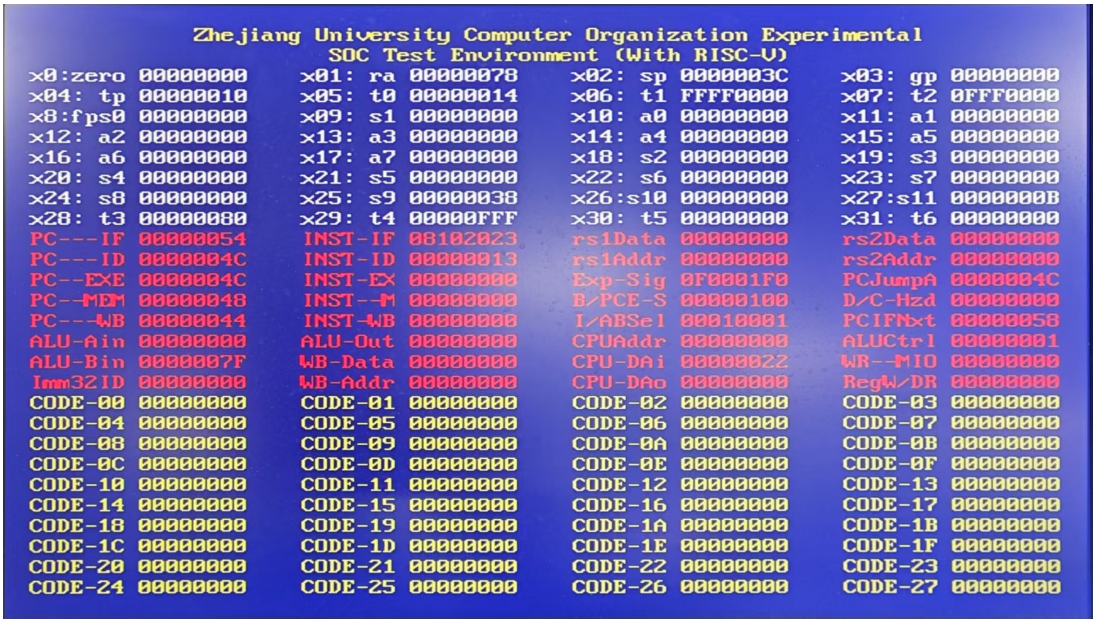
PC_IF：0x3C

5.2.3 测试点 3

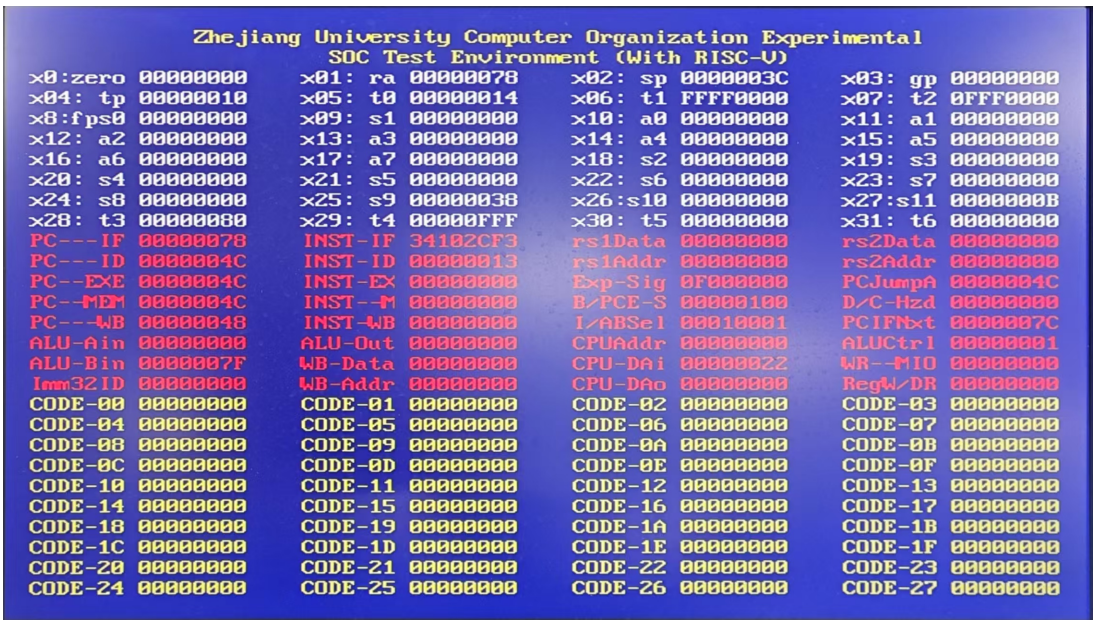
下述照片节选了指令 0x40-0x78 的上板表现。结合前文分析，上板测试结果与预期和仿真结果相符。



PC_IF：0x50



PC_IF：0x54


Zhejiang University Computer Organization Experimental
SOC Test Environment (With RISC-V)
x0:zero 00000000 x01: ra 00000078 x02: sp 0000003C x03: gp 00000000
x04: tp 00000010 x05: t0 00000014 x06: t1 FFFF0000 x07: t2 0FFF0000
x8: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 00000000 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 00000038 x26:s10 00000000 x27:s11 0000000B
x28: t3 00000080 x29: t4 00000FFF x30: t5 00000000 x31: t6 00000000
PC---IF 00000078 INST-IF 34102CF3 rs1Data 00000000 rs2Data 00000000
PC---ID 0000004C INST-ID 00000013 rs1Addr 00000000 rs2Addr 00000000
PC--EXE 0000004C INST-EX 00000000 Exp-Sig 0F000000 PCJumpA 0000004C
PC--MEM 0000004C INST--M 00000000 B/PCE-S 00000100 D/C-Hzd 00000000
PC---WB 00000048 INST-WB 00000000 I/ABSel 00010001 PCIFNxt 0000007C
ALU-Ain 00000000 ALU-Out 00000000 CPUAddr 00000000 ALUCtrl 00000001
ALU-Bin 0000007F WB-Data 00000000 CPU-DAi 00000022 WR--MIO 00000000
Imm32ID 00000000 WB-Addr 00000000 CPU-DAo 00000000 RegW/DR 00000000
CODE-00 00000000 CODE-01 00000000 CODE-02 00000000 CODE-03 00000000
CODE-04 00000000 CODE-05 00000000 CODE-06 00000000 CODE-07 00000000
CODE-08 00000000 CODE-09 00000000 CODE-0A 00000000 CODE-0B 00000000
CODE-0C 00000000 CODE-0D 00000000 CODE-0E 00000000 CODE-0F 00000000
CODE-10 00000000 CODE-11 00000000 CODE-12 00000000 CODE-13 00000000
CODE-14 00000000 CODE-15 00000000 CODE-16 00000000 CODE-17 00000000
CODE-18 00000000 CODE-19 00000000 CODE-1A 00000000 CODE-1B 00000000
CODE-1C 00000000 CODE-1D 00000000 CODE-1E 00000000 CODE-1F 00000000
CODE-20 00000000 CODE-21 00000000 CODE-22 00000000 CODE-23 00000000
CODE-24 00000000 CODE-25 00000000 CODE-26 00000000 CODE-27 00000000

PC_IF：0x78