浙江大學
ZHEJIANG UNIVERSITY

Lab 1 Pipelined CPU Supporting RISC-V RV32I Instructions

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 2 月 27 日

Lab 1 Report

1 实验目的

- **理解 RISC-V RV32I 指令：**掌握 RISC-V RV32I 指令集的基本内容和结构，强化对 R、I、S、B、U、J 型指令的编码格式、解码方案、工作流程的理解。
- **掌握流水线 CPU 设计：**掌握经典五级流水线 CPU 的设计方法，能够实现执行 RISC-V RV32I 指令集的流水线处理。
- **掌握数据冒险的检测与处理机制：**设计数据冒险的判断单元 Hazard Detect Unit，并实现数据转发和旁路机制。
- **掌握分支预测与流水线停顿的处理：**以 Predict Not Taken 方案进行流水线预测，并实现预测未命中分支时的流水线停顿与刷新机制。

2 实验原理

2.1 控制单元的设计 —— CtrlUnit.v

控制单元根据输入指令的内容，判断指令类型和操作，从而生成流水线 CPU 所需的各种控制信号。这些信号帮助协调流水线各个阶段的操作，应对数据冒险和控制冒险的发生。

2.1.1 对指令类型的判断

在本实现中，对 R 型、I 型、S 型、B 型等指令的类型判断分为两级：首先根据 OPcode 判断指令的基本类型，随后根据 funct3 和 funct7 来判断具体的指令类型。而对于特殊指令（如 LUI、AUIPC、JALR），则直接通过一级判断完成。

```
wire BEQ = Bop & funct3_0;
...
wire LB =  Lop & funct3_0;
...
wire SB = Sop & funct3_0;
...
wire LUI   = opcode == 7'b0110111;
wire AUIPC = opcode == 7'b0010111;
wire JAL  = opcode == 7'b1101111;
assign JALR = opcode == 7'b1100111 && funct3_0
```

2.1.2 基本控制信号的生成

1. **cmp_ctrl 信号**
   该控制信号在 B 型指令下存在意义，其值将传递给分支指令比较单元 cmp_32（实现细节见下文），作用是为 cmp_32 指定比较规则。例如：若当前解码的指令为 BEQ 指令，则该控制信号的值将为 3'b001，cmp_32 接收到该信号值后，将以“是否相等”为规则，对指令中指定的两寄存器的值进行比较。

2. **Branch 信号**
   该控制信号在 B 型、JAL、JALR 指令下存在意义，其值将传递给 IF 阶段的一个二路选择器，作用是支持指令的跳转操作。该信号为 1 有两种情形：一是指令为 JAL 或 JALR；二是指令为 B 型且 cmp_32 传回的比较结果为真。

3. **ALUSrc_A 信号**
   该信号控制 ALU 的第一个操作数的来源。当该信号为 1 时，ALU 的第一个操作数将来自于当前指令的 PC 地址。对于 JAL、JALR 和 AUIPC 指令，由于它们需要将 rd 寄存器的值更新为 PC+4（或 PC+ 立即数），因此该信号为 1。

4. **ALUSrc_B 信号**
   该信号控制 ALU 的第二个操作数的来源。当该信号为 1 时，ALU 的第二个操作数将来自于立即数生成器。对于涉及立即数生成的 I 型、L 型、S 型和 LUI、AUIGC 指令，该信号为 1.

```
assign cmp_ctrl = B_valid ? (BEQ  ? 3'b001 :
                  BNE  ? 3'b010 :
                  BLT  ? 3'b011 :
                  BLTU ? 3'b100 :
                  BGE  ? 3'b101 :
                  BGEU ? 3'b110 : 3'b000) : 3'b000;
assign Branch = JAL | JALR | B_valid & cmp_res;
assign ALUSrc_A = JAL | JALR | AUIPC;
assign ALUSrc_B = I_valid | L_valid | S_valid | LUI | AUIPC;
```

2.1.3 冒险相关信号的生成

根据指令类型生成 hazard_optype 信号，其值将传递给 HazardDetectionUnit（实现细节见下文），以便其正确生成冒险的信号。

```
assign hazard_optype =
        (R_valid | I_valid | JAL | JALR | LUI | AUIPC)? 2'b01 :
         L_valid ? 2'b10 :
```

```
4          S_valid ? 2'b11 : 2'b00;
```

2.2 冒险判断单元的设计 —— HazardDetectionUnit.v

2.2.1 模块的输入与输出

输入信号：

1. Branch_ID：表示 ID 阶段指令是否为需要进行跳转的分支指令。若是，则由本模块向流水线寄存器 FD 发出清空信号。

2. rs1_ID、rs2_ID、rs1use_ID、rs2use_ID：表示 ID 阶段指令是否读取 rs1、rs2 寄存器，以及读取的寄存器号。

3. hazard_optype_ID：表示 ID 阶段指令的操作类型，区分 ALU 操作、LOAD 操作和 STORE 操作，用于与其它控制信号一同确定是否进行前递以及前递的数据来源。

4. rd_EXE、rd_MEM 等：表示其它阶段使用到的寄存器号。

输出信号：

1. PC_EN_IF：控制 PC 寄存器的使能。该值为 0 当且仅当仅当发生载入—使用型数据冒险（load_stall 为 1），需要插入停顿。

2. reg_FD_EN、reg_DE_EN 等：控制流水线寄存器的使能。

3. reg_FD_stall：如果发生载入—使用型数据冒险（load_stall 为 1），则需要暂停 FD 阶段的流水线操作，直至下个时钟周期。

4. reg_FD_flush、reg_DE_flush 等：清空对应的流水线寄存器。

5. forward_ctrl_ls：当 EXE 阶段的指令是 STORE，且 MEM 阶段的操作是 LOAD，并且 rs2_EXE 与 rd_MEM 匹配时，说明需要对 rs2 进行加载-存储的转发处理。

6. forward_ctrl_A、forward_ctrl_B：根据 rs1(rs2) 的状态，决定转发到 ALU 的 A(B) 输入的源。它是一个 2 位的控制信号，分别对应不同的源（不前递、来自 EXE 阶段 ALU 结果的前递、来自 MEM 阶段 ALU 结果的前递、来自 MEM 阶段内存读取值的前递）。

2.2.2 判断逻辑

```
module HazardDetectionUnit(
input clk,
input Branch_ID, rs1use_ID, rs2use_ID,
input [1:0] hazard_optype_ID,
input [4:0] rd_EXE, rd_MEM, rs1_ID, rs2_ID, rs2_EXE,
output PC_EN_IF,
output reg_FD_EN, reg_FD_stall, reg_FD_flush,
output reg_DE_EN, reg_DE_flush,
output reg_EM_EN, reg_EM_flush,
output reg_MW_EN,
output forward_ctrl_ls,
output [1:0] forward_ctrl_A, forward_ctrl_B
);

reg[1:0] hazard_optype_EXE, hazard_optype_MEM;
always@(posedge clk) begin
hazard_optype_MEM <= hazard_optype_EXE & {2{~reg_EM_flush}};
hazard_optype_EXE <= hazard_optype_ID & {2{~reg_DE_flush}};
end

localparam hazard_optype_ALU = 2'd1;
localparam hazard_optype_LOAD = 2'd2;
localparam hazard_optype_STORE = 2'd3;

wire rs1_forward_1 = rs1use_ID && rs1_ID == rd_EXE && rd_EXE &&
   hazard_optype_EXE == hazard_optype_ALU;
wire rs1_forward_stall = rs1use_ID && rs1_ID == rd_EXE && rd_EXE &&
   hazard_optype_EXE == hazard_optype_LOAD
&& hazard_optype_ID != hazard_optype_STORE;
wire rs1_forward_2 = rs1use_ID && rs1_ID == rd_MEM && rd_MEM &&
   hazard_optype_MEM == hazard_optype_ALU;
wire rs1_forward_3 = rs1use_ID && rs1_ID == rd_MEM && rd_MEM &&
   hazard_optype_MEM == hazard_optype_LOAD;

wire rs2_forward_1 = rs2use_ID && rs2_ID == rd_EXE && rd_EXE &&
   hazard_optype_EXE == hazard_optype_ALU;
wire rs2_forward_stall = rs2use_ID && rs2_ID == rd_EXE && rd_EXE &&
   hazard_optype_EXE == hazard_optype_LOAD
```

```
&& hazard_optype_ID != hazard_optype_STORE;
wire rs2_forward_2 = rs2use_ID && rs2_ID == rd_MEM && rd_MEM &&
   hazard_optype_MEM == hazard_optype_ALU;
wire rs2_forward_3 = rs2use_ID && rs2_ID == rd_MEM && rd_MEM &&
   hazard_optype_MEM == hazard_optype_LOAD;

wire load_stall = rs1_forward_stall | rs2_forward_stall;

assign PC_EN_IF = ~load_stall;
assign reg_FD_stall = load_stall;
assign reg_FD_flush = Branch_ID;
assign reg_DE_flush = load_stall;
assign reg_EM_flush = 1'b0;

assign reg_FD_EN=1'b1;
assign reg_DE_EN=1'b1;
assign reg_EM_EN=1'b1;
assign reg_MW_EN=1'b1;

assign forward_ctrl_A = {2{rs1_forward_1}} & 2'd1 |
{2{rs1_forward_2}} & 2'd2 |
{2{rs1_forward_3}} & 2'd3 ;

assign forward_ctrl_B = {2{rs2_forward_1}} & 2'd1 |
{2{rs2_forward_2}} & 2'd2 |
{2{rs2_forward_3}} & 2'd3 ;

assign forward_ctrl_ls = rs2_EXE == rd_MEM  &&
hazard_optype_EXE == hazard_optype_STORE    &&
hazard_optype_MEM == hazard_optype_LOAD;


endmodule
```

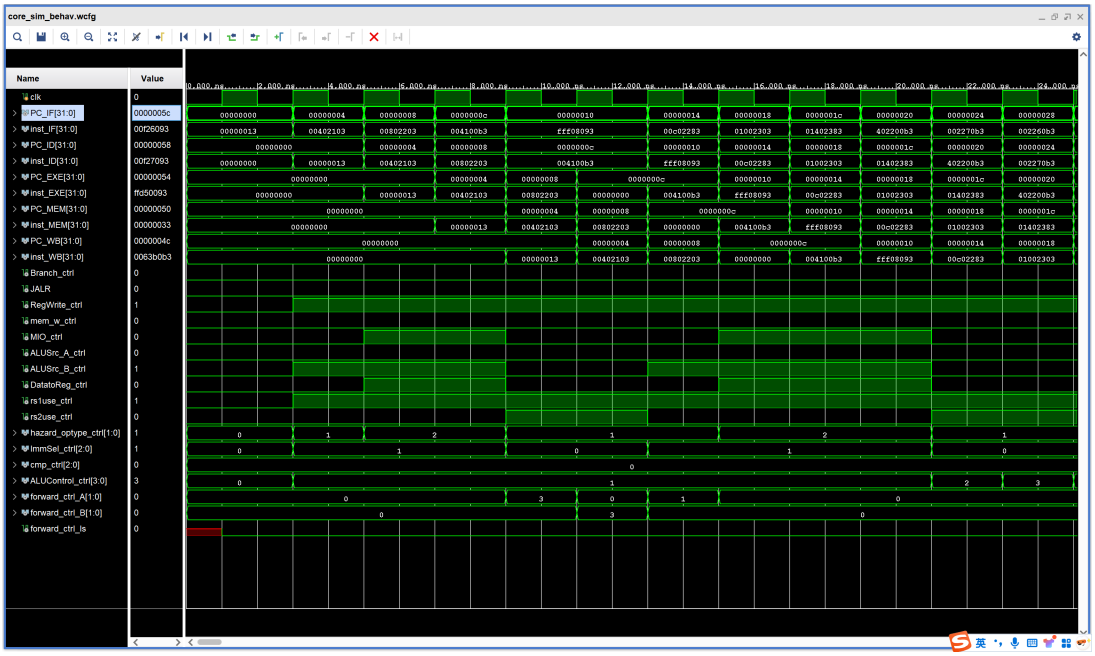
2.3 CPU 单元间的连线方案 —— RV32core.v

```
// IF
MUX2T1_32 mux_IF(.I0(PC_4_IF),.I1(jump_PC_ID),.s(Branch_ctrl),.o(
    next_PC_IF));

// ID
MUX4T1_32 mux_forward_A(.I0(rs1_data_reg),.I1(ALUout_EXE),.I2(
    ALUout_MEM),.I3(Datain_MEM),.s(forward_ctrl_A),.o(rs1_data_ID));

MUX4T1_32 mux_forward_B(.I0(rs2_data_reg),.I1(ALUout_EXE),.I2(
    ALUout_MEM),.I3(Datain_MEM),.s(forward_ctrl_B),.o(rs2_data_ID));

// EX
MUX2T1_32 mux_A_EXE(.I0(rs1_data_EXE),.I1(PC_EXE),.s(ALUSrc_A_EXE),.o
    (ALUA_EXE));

MUX2T1_32 mux_B_EXE(.I0(rs2_data_EXE),.I1(Imm_EXE),.s(ALUSrc_B_EXE),.
    o(ALUB_EXE));

MUX2T1_32 mux_forward_EXE(.I0(rs2_data_EXE),.I1(Datain_MEM),.s(
    forward_ctrl_ls),.o(Dataout_EXE));
```

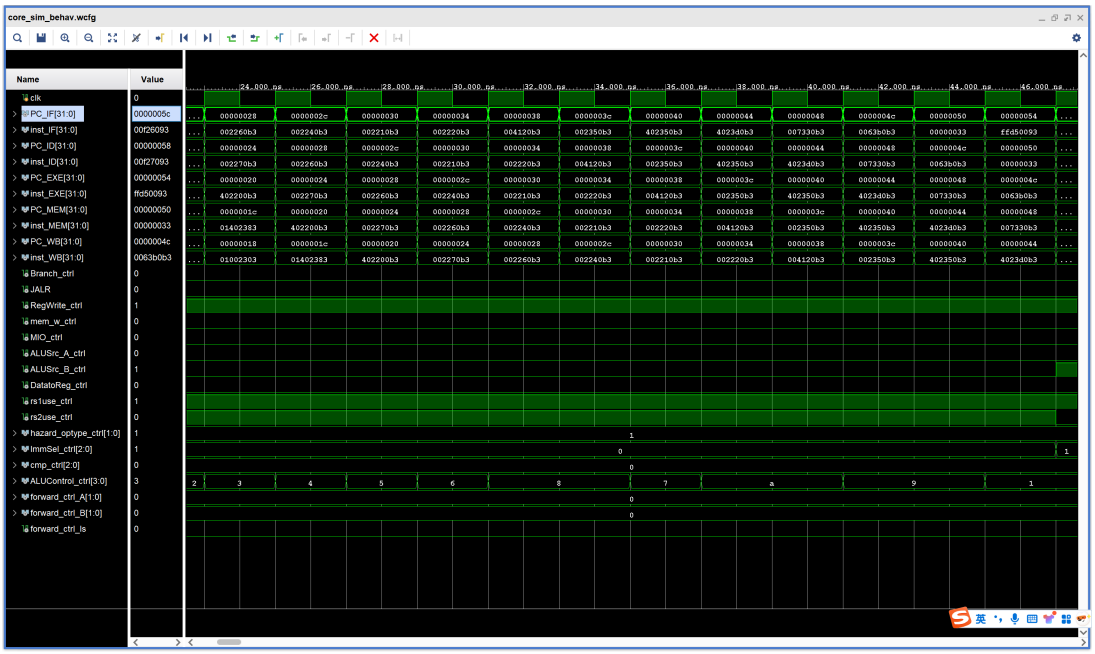
3 实验结果

3.1 仿真测试结果

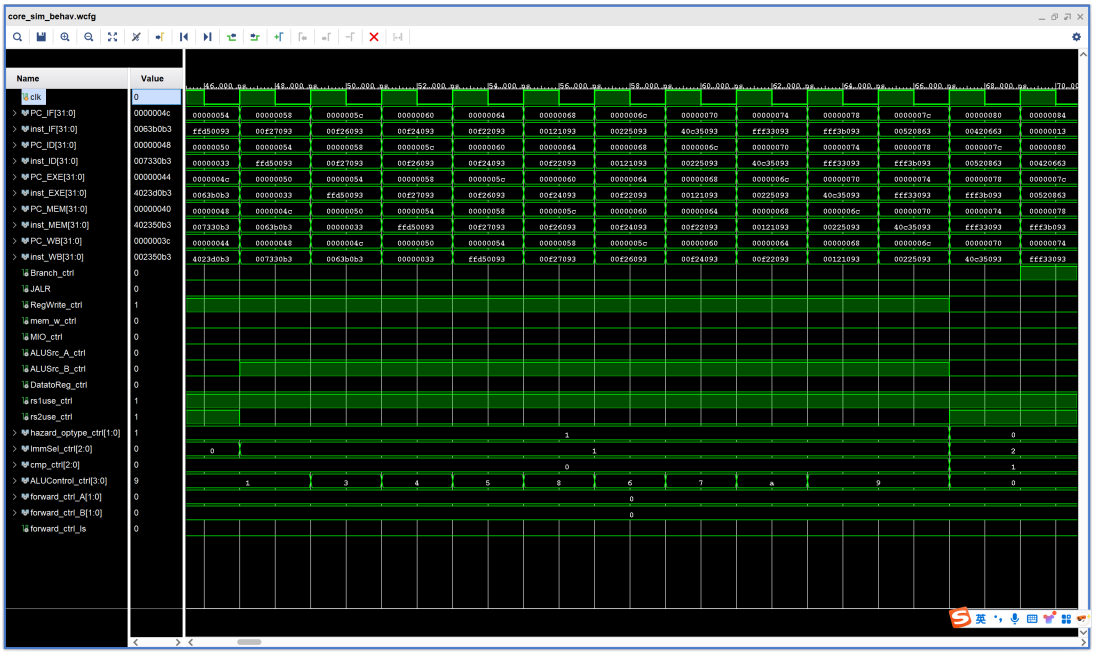
3.1.1 仿真波形



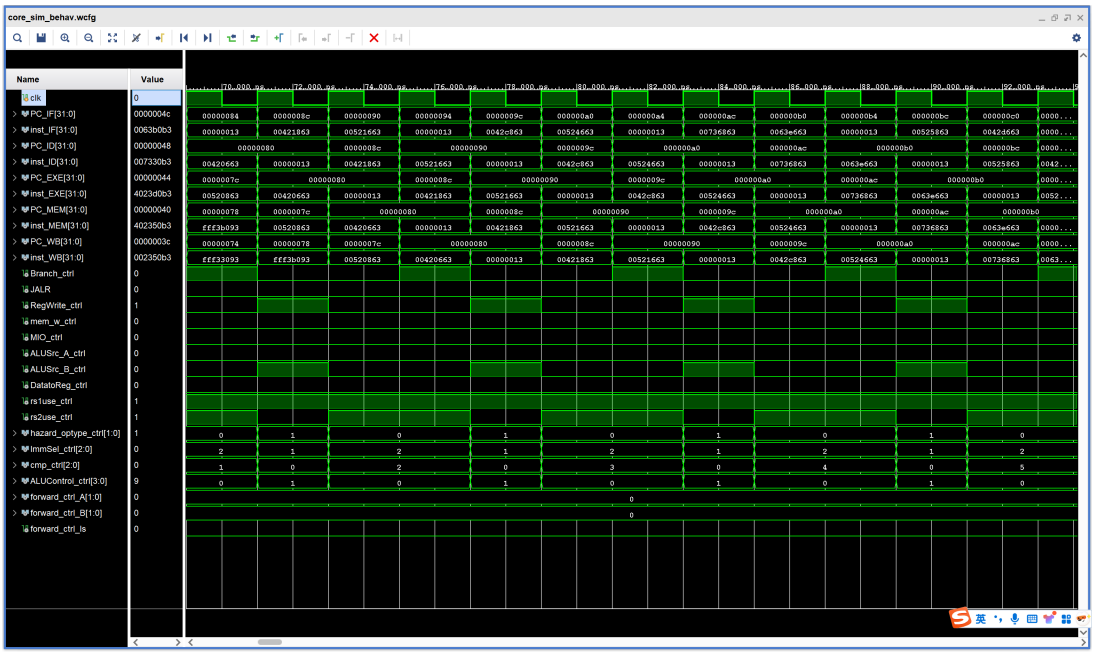
PC：0x0-0x28



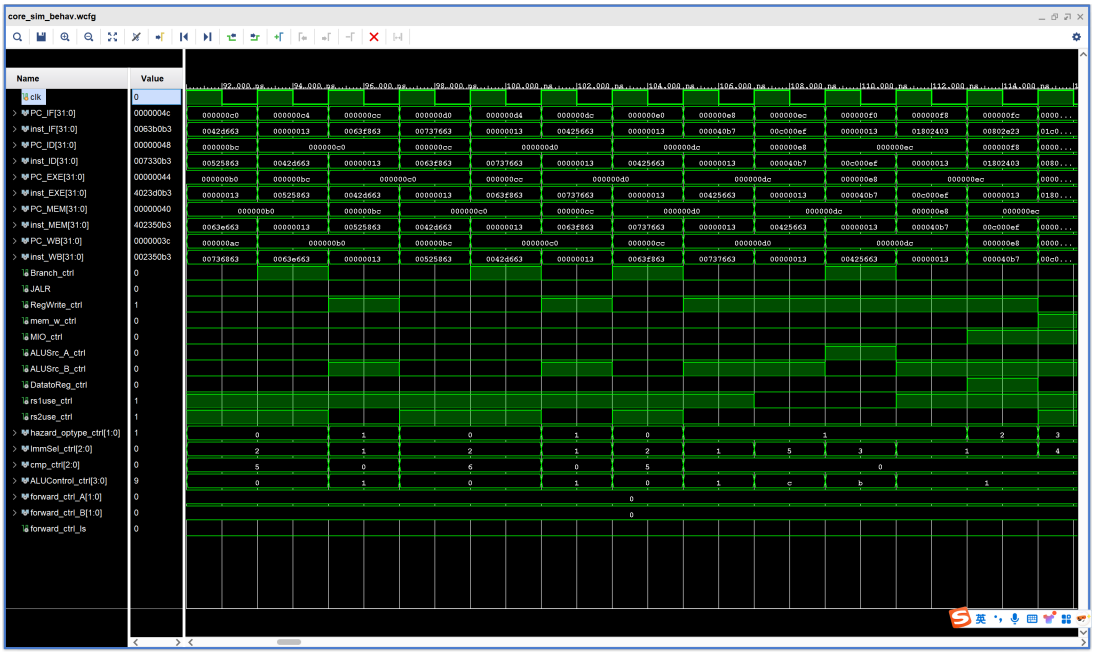
PC：0x2C-0x54



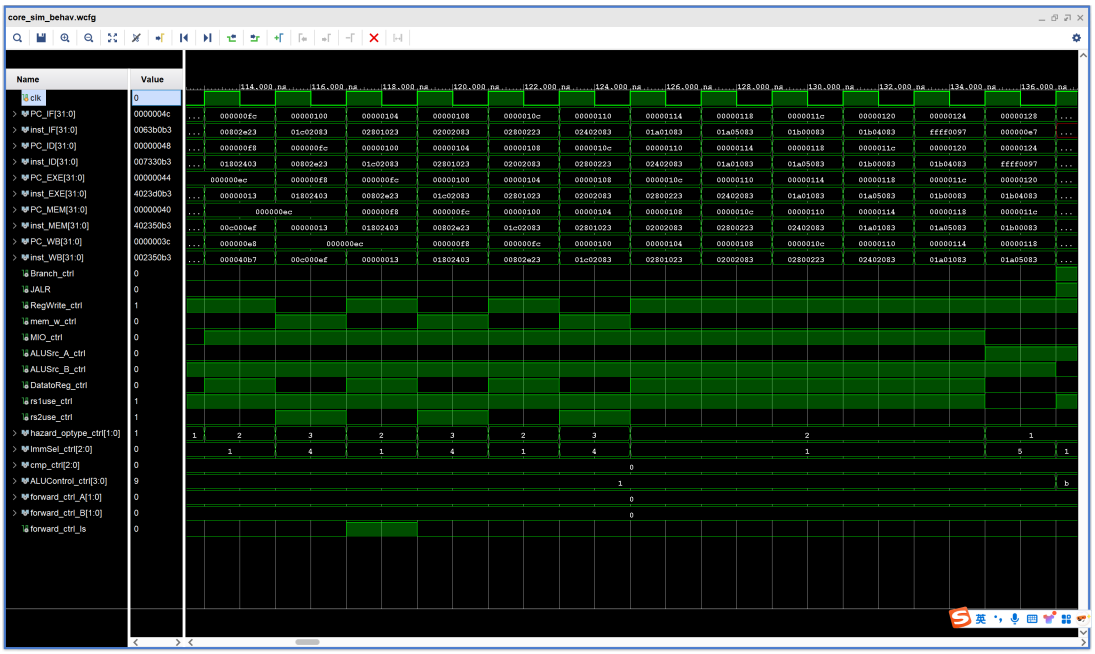
PC：0x58-0x84



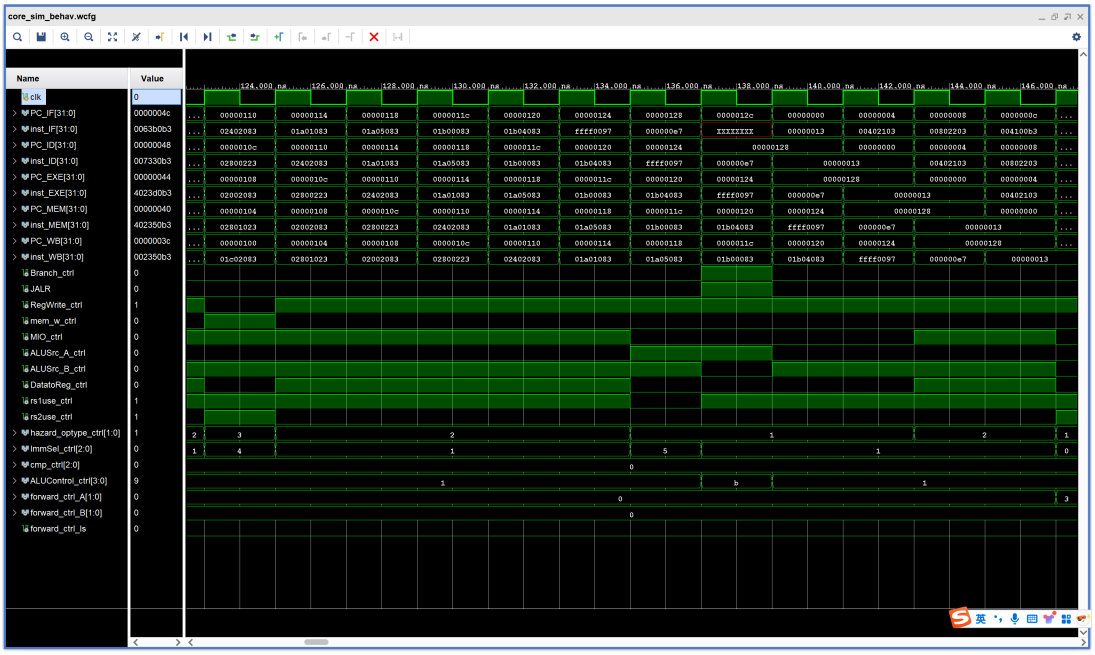
PC：0x8C-0xC0



PC：0xC4-0xFC



PC：0x100-0x128



PC：0x12C-0x0

3.1.2 结果解释

我们以下面的代码段进行分析：

Instruction	PC	ASM
00402103	4	lw x2, 4(x0)
00802203	8	lw x4, 8(x0)
004100b3	C	add x1, x2, x4
fff08093	10	addi x1, x1, -1

当 PC=0xC 的指令进入 ID 阶段时：

- 由于 PC=0xC 的指令同时使用 rs1 和 rs2 寄存器，因此 rs1use_ctrl 和 rs2use_ctrl 均为真。
- 由于 PC=0xC 的指令为 R 型指令，因此 hazard_optype_control 为 1。
- 由于 PC=0xC 的指令使用的 rs1 寄存器与 PC=0x4 的指令使用的 rd 寄存器相同，故需要发生前递，forward_ctrl_A 为 3，表示将来自 MEM 阶段的内存读取结果前递给 ALU 作为其第一个操作数。
- 由于 PC=0xC 的指令使用的 rs2 寄存器与 PC=0x8 的指令使用的 rd 寄存器相同，检测到载入—使用型数据冒险，于是发生流水线停顿。待流水线停顿结束后，需要发生前递，forward_ctrl_B 为 3，表示将来自 MEM 阶段的内存读取结果前递给 ALU 作为其第二个操作数。

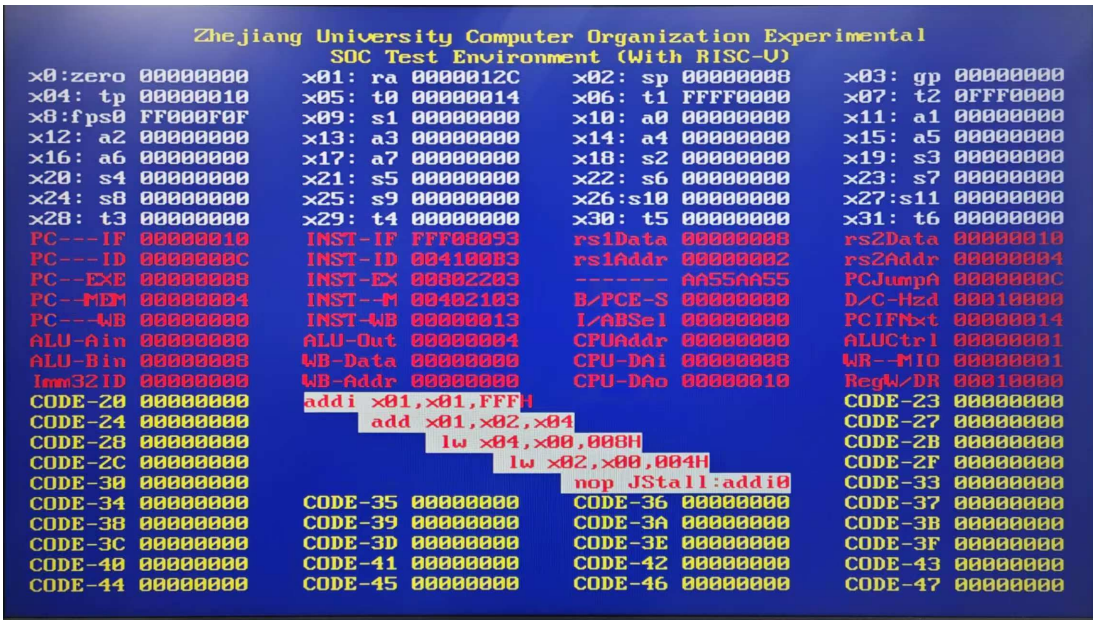
当 PC=0x10 的指令进入 ID 阶段时：

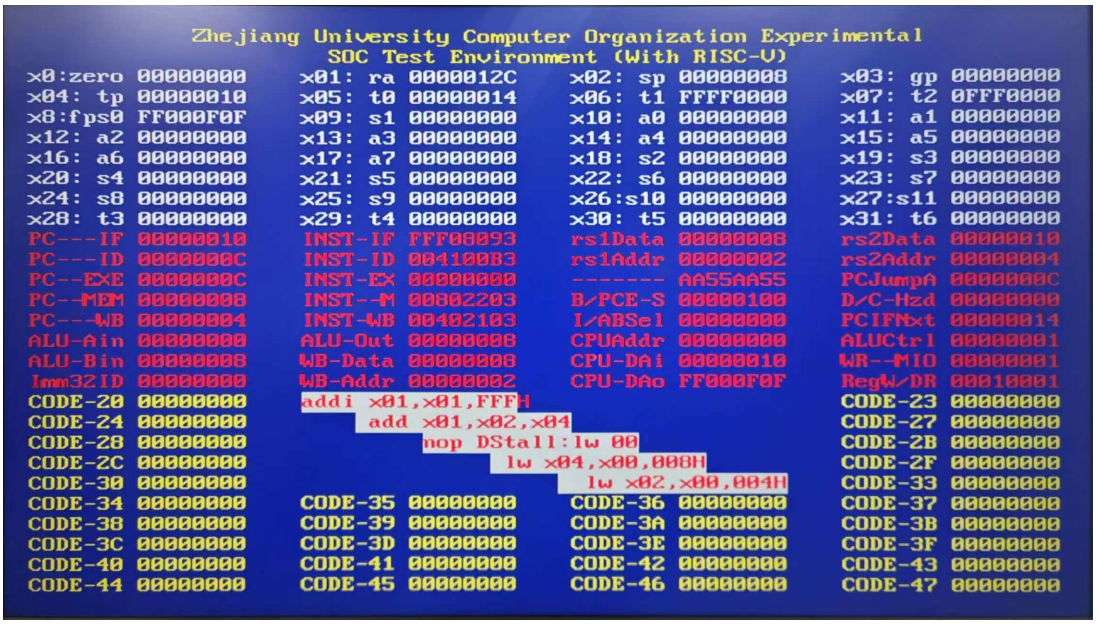
- 由于该指令需要将立即数生成器的输出作为 ALU 的第二个操作数,因此 ALUSrc_B 为 1。
- 由于 PC=0xC 的指令为 R 型指令，因此 hazard_optype_control 为 1。
- 由于 PC=0x10 的指令使用的 rs1 寄存器与 PC=0xC 的指令使用的 rd 寄存器相同，发生数据冒险，需要将来自 EXE 阶段的 ALU 运算结果前递给 ALU 作为其第一个操作数，于是 forward_ctrl_A 为 1。

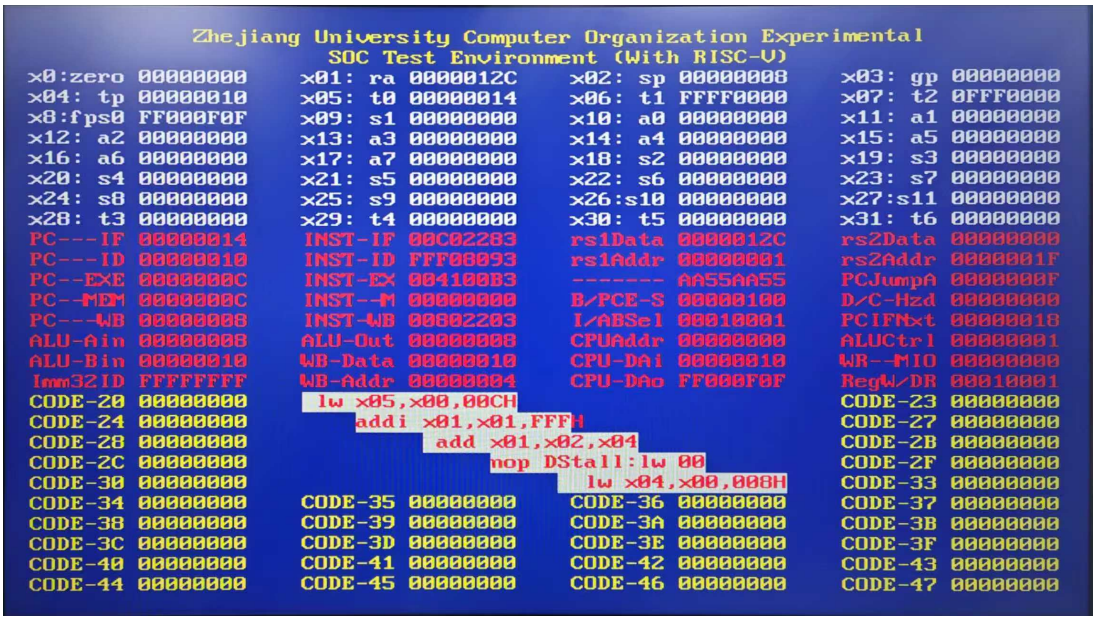
受限于篇幅和精力，不对其它指令的运行情况进行分析。经过校验，上述仿真结果与实验文档中提供的样例结果相同。

3.2 上板测试结果

3.2.1 测试照片


Zhejiang University Computer Organization Experimental
SOC Test Environment (With RISC-V)
x0:zero 00000000 x01: ra 0000012C x02: sp 00000008 x03: gp 00000000
x04: tp 00000010 x05: t0 00000014 x06: t1 FFFF0000 x07: t2 0FFF0000
x8:fps0 FF000F0F 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 00000000 x26:s10 00000000 x27:s11 00000000
x28: t3 00000000 x29: t4 00000000 x30: t5 00000000 x31: t6 00000000
PC---IF 00000010 INST-IF FFF08093 rs1Data 00000008 rs2Data 00000010
PC---ID 0000000C INST-ID 004100B3 rs1Addr 00000002 rs2Addr 00000004
PC--EXE 00000008 INST-EX 00802203 ------- AA55AA55 PCJumpA 0000000C
PC--MEM 00000004 INST--M 00402103 B/PCE-S 00000000 D/C-Hzd 00010000
PC---WB 00000000 INST-WB 00000013 I/ABSel 00000000 PCIFNxt 00000014
ALU-Ain 00000000 ALU-Out 00000004 CPUAddr 00000000 ALUCtrl 00000001
ALU-Bin 00000008 WB-Data 00000000 CPU-DAi 00000008 WR--MIO 00000001
Imm32ID 00000000 WB-Addr 00000000 CPU-DAo 00000010 RegW/DR 00010000
CODE-20 00000000 addi x01,x01,FFFH CODE-23 00000000
CODE-24 00000000 add x01,x02,x04 CODE-27 00000000
CODE-28 00000000 lw x04,x00,008H CODE-2B 00000000
CODE-2C 00000000 lw x02,x00,004H CODE-2F 00000000
CODE-30 00000000 nop JStall:addi0 CODE-33 00000000
CODE-34 00000000 CODE-35 00000000 CODE-36 00000000 CODE-37 00000000
CODE-38 00000000 CODE-39 00000000 CODE-3A 00000000 CODE-3B 00000000
CODE-3C 00000000 CODE-3D 00000000 CODE-3E 00000000 CODE-3F 00000000
CODE-40 00000000 CODE-41 00000000 CODE-42 00000000 CODE-43 00000000
CODE-44 00000000 CODE-45 00000000 CODE-46 00000000 CODE-47 00000000


Zhejiang University Computer Organization Experimental
SOC Test Environment (With RISC-V)
x0:zero 00000000 x01: ra 0000012C x02: sp 00000008 x03: gp 00000000
x04: tp 00000010 x05: t0 00000014 x06: t1 FFFF0000 x07: t2 0FFF0000
x8:fps0 FF000F0F 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 00000000 x26:s10 00000000 x27:s11 00000000
x28: t3 00000000 x29: t4 00000000 x30: t5 00000000 x31: t6 00000000
PC---IF 00000010 INST-IF FFF08093 rs1Data 00000008 rs2Data 00000010
PC---ID 0000000C INST-ID 004100B3 rs1Addr 00000002 rs2Addr 00000004
PC--EXE 0000000C INST-EX 00000000 ------- AA55AA55 PCJumpA 0000000C
PC--MEM 00000008 INST--M 00802203 B/PCE-S 00000100 D/C-Hzd 00000000
PC---WB 00000004 INST-WB 00402103 I/ABSel 00000000 PCIFNxt 00000014
ALU-Ain 00000000 ALU-Out 00000008 CPUAddr 00000000 ALUCtrl 00000001
ALU-Bin 00000008 WB-Data 00000008 CPU-DAi 00000010 WR--MIO 00000001
Imm32ID 00000000 WB-Addr 00000002 CPU-DAo FF000F0F RegW/DR 00010001
CODE-20 00000000 addi x01,x01,FFFH CODE-23 00000000
CODE-24 00000000 add x01,x02,x04 CODE-27 00000000
CODE-28 00000000 nop DStall:lw 00 CODE-2B 00000000
CODE-2C 00000000 lw x04,x00,008H CODE-2F 00000000
CODE-30 00000000 lw x02,x00,004H CODE-33 00000000
CODE-34 00000000 CODE-35 00000000 CODE-36 00000000 CODE-37 00000000
CODE-38 00000000 CODE-39 00000000 CODE-3A 00000000 CODE-3B 00000000
CODE-3C 00000000 CODE-3D 00000000 CODE-3E 00000000 CODE-3F 00000000
CODE-40 00000000 CODE-41 00000000 CODE-42 00000000 CODE-43 00000000
CODE-44 00000000 CODE-45 00000000 CODE-46 00000000 CODE-47 00000000



3.2.2 结果解释

上述照片节选了 PC=0xC-0x10 时的流水线停顿的上板表现。结合前文分析，上板测试结果与预期和仿真结果相符。其余指令的上板测试结果不再赘述。