CS202 Computer Organization Project 文档

开发者说明

项目成员	负责工作	贡献百分比
12310513 娄毅彬	指令译码模块、寄存器堆模块、仿真测试与验证、上板部署与调试	33%
12310519 方酉城	整体架构设计、数据存储器模块、仿真测试与验证、流水线架构	33%
12310520 芮煜涵	ALU模块、程序计数器与指令存储器、样例测试	33%

CPU架构设计说明

ISA

> **参考指令集**：本项目基于 RISC-V **RV32I Base Integer Instruction Set**

类型	指令	Opcode	功能说明
R	add, sub, and, or, xor, sll, srl, sra, slt, sltu	0110011	算术与逻辑运算
I	addi, andi, ori, xori, slli, srli, lw, jalr	0010011 / 0000011 / 1100111	立即数运算、访存、跳转
S	sw, sh, sb	0100011	存储指令
B	beq, bne, blt, bge, bltu, bgeu	1100011	条件跳转
U	lui, auipc	0110111 / 0010111	立即加载、PC偏移
J	jal	1101111	非条件跳转
SYS	ecall	1110011	系统调用，扩展IO

寄存器配置

- 32个通用寄存器，位宽32位；
- 保留x0始终为0；
- 使用x10 (a0)、x17 (a7) 作为`ecall`传参寄存器；
- x31作为栈指针，初始值为`0x2000`。

异常处理支持

- 基于简化系统调用判断：
    - 当执行 `ecall` 且 `a7 == 1` → 输出 `a0` 到数码管；
    - 当 `a7 == 5` → 读取开关值至 `a0`。
- 对于非法指令不会执行

CPU类型

本项目完成了两种 CPU 架构的实现：

① 单周期 CPU（已上板）

- 所有指令在一个时钟周期内完成，按最长路径（如访存指令）设计时钟周期；
- 各模块（取指、译码、运算、访存、写回）并行接入，按顺序执行；
- 系统主时钟由 `cpuclk` 模块生成，驱动所有模块同步工作；
- 理论 CPI 为 1，实现了稳定的上板部署；
- 已成功移植至 FPGA 平台，完成 LED、数码管、IO 交互验证。

② 五级流水线 CPU（仿真实现）

- 采用经典五级结构：IF（取指）→ ID（译码）→ EX（执行）→ MEM（访存）→ WB（写回），通过 `*_to_*` 模块进行阶段寄存器传递；
- 实现了**数据前递机制（Forwarding）**，通过 `Forwarding_EX` 和 `Forwarding_ALU` 模块解决 EX/MEM 和 MEM/WB 阶段的数据冒险；
- 实现了**数据冒险检测与暂停控制（Stall）**，在 load-use 情况下通过 `Hazard_Detection_Unit` 模块生成 stall 信号，暂停流水线推进；
- 实现了**基于 LRU 缓存的静态分支预测模块（PC_Prediction）**，根据历史跳转记录预测跳转地址，并在预测失败时通过 `wrong_prediction_flag` 清空错误指令；
- 指令流控制机制完善，支持基本的冲突恢复逻辑；
- 主时钟统一由 `cpuclk` 提供，各阶段按拍同步执行；
- 仿真过程中流水线运行稳定，CPI 理论值接近 1，吞吐率优于单周期架构。

总体说明：

- 两种架构均共用统一指令集、控制器与模块接口；
- 主时钟统一由 `cpuclk` 输出，保持系统一致性；
- 系统调用通过 `ecall` + `a7` + `a0` 实现简单功能调用模拟。

寻址空间设计

- **结构类型**：本设计采用**哈佛结构（Harvard Architecture）**，指令存储器（`prgrom`）与数据存储器（`RAM`）完全独立

- **寻址单位**：系统采用**字节寻址**，即地址每次加1表示移动一个字节；所有读写指令均以 **32 位 word 对齐**访问。
- **指令空间**：
    - 由 `prgrom` IP 核提供；
    - 地址宽度为 14 位（`pc[15:2]`），支持最多 **16K 条指令**（每条 32 位），即 **64KB 容量**；
    - 指令通过 UART 接口上传加载，支持按字对齐顺序读取。
- **数据空间**：
    - 由 `RAM` IP 核提供，容量同样为 **64KB**，支持完整的字节、半字、字读写；
    - 内部以 byte 数组存储，结合 load/store 类型完成数据拼接与符号扩展；
    - 支持内存映射 I/O 地址段（详见 IO 支持部分）。
- **栈空间设置**：
    - 初始栈指针寄存器 `x31` 被设置为 `0x00002000`，即 RAM 高地址处；
    - 栈按传统方式向低地址生长，当前实现未提供 `push/pop` 等专用指令扩展，但栈空间可用于函数调用与中间变量存储；
    - 在 `Decoder` 模块中通过寄存器初始化完成栈指针设置。

对外设 IO 的支持

- **访问方式**：本设计采用**MMIO**方式访问外设，外设设备通过映射在数据存储空间的特定地址段上实现读写，无需使用专用 I/O 指令
- **地址映射表**：

地址（32位）	设备类型	访问类型	说明
`0x0000_2000`	LED	W	写入 8 位值控制板上 LED 灯状态
`0x0000_2001`	数码管段码显示	W	写入 32 位数值，控制段码内容显示
`0x0000_2002`	数码管进制标志	W	写入最低位控制十进制/十六进制切换
`0x0000_2010`	拨码开关（Switch）	R	读取 8 位输入值
`0x0000_2011`	班级编号（三位数）	R	读取编号输入
`0x0000_2012`	确认键（Confirm）	R	读取是否按下确认按钮

- **控制机制**：
    - 在 `Dmem` 模块中，通过对 `addr_out` 的判断，实现对以上地址的定向访问；
    - 读操作通过 `load_type` 类型判定是否为符号扩展（如 `lb`、`lbu`）；
    - 写操作直接写入对应设备模块，例如 `leds`、`segments`、`base`。
- **系统调用支持**：

    - 项目支持 `ecall` 指令模拟系统调用，通过寄存器 `a7` 指示调用类型，`a0` 传参或回传结果；
        - `a7 == 1`：将 `a0` 中值输出到数码管；
        - `a7 == 5`：将开关值读取到 `a0`；
- **访问方式选择**：**采用轮询方式（Polling）**访问 I/O，程序需主动发出访问指令读取或写入 I/O 地址；

CPU接口

本项目 CPU 通过顶层模块 CPU_TOP 与开发板和外设连接，接口设计覆盖时钟管理、通信传输、人机交互与显示控制等功能，具体如下：

时钟接口

- clkin（输入）：系统主时钟输入，由开发板提供
- cpuclk 模块对主时钟分频，分别输出：
    - clk：供主 CPU 使用的系统运行时钟
    - upg_clk_i：供 UART 下载模块使用的串口时钟

复位接口

- rst（输入）：高电平有效的复位信号
- 内部通过与 UART 下载状态联合判断生成 rst1，在程序加载完成后进入主运行状态

UART通信接口

- rx（输入）：UART 接收端，用于从 PC 下载程序（通过 uart_bmpg_0 模块）
- tx（输出）：UART 发送端，预留扩展使用
- kickOff（输出）：指示程序加载完成，启动 CPU 正式运行
- start_pg（输入）：用户通过按键启动程序加载过程

用户输入接口

- switch[7:0]（输入）：8 位拨码开关，用于输入数据或指令参数
- number[2:0]（输入）：3位编号输入，供数据读取选择
- confirm（输入）：确认按键，配合 Debug 模块用于切换显示内容或控制交互

输出显示接口

- leds[7:0]（输出）：8 位 LED 灯，显示数值或状态信息
- segment1[7:0]、segment2[7:0]（输出）：段码显示内容（两个数码管）
- anode[7:0]（输出）：段码位选，支持动态扫描显示
- base（输出）：段码进制控制信号，指示当前为十进制或十六进制显示模式

方案分析说明

1. 乘法运算：硬件 vs 软件

对比维度	硬件方案	软件方案
实现方式	使用硬件乘法器，单周期完成乘法运算	通过循环加法或移位运算模拟乘法，需多条指令完成
性能	速度快，硬件资源占用较多	性能较低，每次乘法延迟较大
资源占用	占用更多硬件资源（如乘法阵列）	占用少量 CPU 资源
实现复杂度	硬件实现复杂，增加了开发负担	实现简单，适合教学与实验
适用场景	高频计算或大规模运算	资源受限，性能要求不高的场景

最终选择：由于项目使用 RV32I 指令集且资源受限，选择了**软件方案**，这样可以节省硬件资源，且在功能验证阶段性能已足够。

---

2. I/O控制：中断 vs 轮询

对比维度	中断控制	轮询控制
响应速度	实时响应，适用于高频或突发数据处理	需主动查询，响应较慢
实现复杂度	需要设计中断控制器，优先级管理等复杂逻辑	实现简单，只需通过地址判断访问外设
性能	性能较高，CPU 空闲时可处理其他任务	性能较低，CPU 在无任务时浪费资源
适用场景	适用于高频、突发事件的场景	适用于低频、周期性任务或简单外设控制

最终选择：考虑到开发周期与实现复杂度，选择了**轮询控制**，这种方式对我们实现的 I/O 操作（如 LED 显示、开关读取）足够且简单。

---

3. 分支预测：静态预测 vs 软件跳转

对比维度	静态分支预测（硬件）	软件跳转（软件）
性能	提前预测跳转地址，减少流水线空泡，提升吞吐率	每次跳转都暂停流水线，性能较低
实现方式	基于 LRU 缓存或其他历史记录进行预测	通过计算条件判断跳转与否
实现复杂度	需要设计预测逻辑和回滚机制	实现简单，直接执行跳转指令
性能影响	性能显著提升，减少控制冒险带来的延迟	性能影响较大，每次跳转需重启流水线

最终选择：为了优化性能，我们在流水线版本中实现了**静态分支预测**，使用 LRU 缓存记录跳转地址，减少了控制冒险带来的性能损失。

系统上板使用说明

在系统上板过程中，开发板的输入与输出操作主要通过以下设备与模块进行交互：

1. 复位操作

- **输入设备**：`rst` 引脚（高电平有效）
- **实现方式**：按下复位按钮时，`rst` 信号被置为高电平，系统将重启，所有模块会被初始化，程序计数器（PC）被清零，重新开始执行。

2. CPU工作模式切换

- **输入设备**：`start_pg` 按键（用于启动程序加载）
- **实现方式**：按下 `start_pg` 按键后，系统开始加载程序并切换到工作模式。此时，通过 UART 接口下载程序代码，系统准备好后，开始正式执行。此操作仅在加载阶段有效。

3. 输入信号（用户交互）

- **拨码开关（Switch）**：`switch[7:0]` 输入用于设置程序的不同参数或选择操作模式。
- **确认按键**：`confirm` 按键，按下后可切换输出内容，如查看不同的寄存器或 PC 值。

4. 输出信号（观测区域）

- **LED 输出**：`leds[7:0]` 控制开发板上的 8 个 LED 灯，用于显示程序执行状态或结果值。
- **数码管显示**：`segment1[7:0]`、`segment2[7:0]` 显示数值信息，`anode[7:0]` 控制位选，用于在数码管上动态显示当前寄存器值或程序计数器值。
- **进制显示**：`base` 信号控制数码管显示进制，十进制或十六进制。

自测试说明

测试方法	测试类型	测试用例描述	测试结果	测试结论
仿真	单元测试	测试 R-type 指令（如 add, sub）是否正确执行	通过	功能正常
仿真	单元测试	测试 I-type 指令（如 lw, addi）是否正确执行	通过	功能正常
仿真	单元测试	测试 J-type 指令（如 jal）是否正常工作	通过	功能正常
仿真	单元测试	测试 S-type 指令（如 sw, sh）是否正确执行	通过	功能正常
仿真	集成测试	基本测试场景 1	通过	功能正常
仿真	集成测试	基本测试场景 2	通过	功能正常
上板	集成测试	基本测试场景 1	通过	功能正常
上板	集成测试	基本测试场景 2	通过	功能正常

测试结论：

- 所有基本指令类型（R-type、I-type、J-type、S-type）均通过了仿真测试，并且在不同输入下功能正常，指令执行正确。
- 测试场景 1 和场景 2 均在仿真中通过
- 测试场景1、2在上板测试中通过，说明系统能够成功完成指令执行和数据处理。

开源及AI对于本次大作业的启发和帮助

Pipeline设计参考：

- 在设计五级流水线时，参考了网络上的开源项目(项目网址：https://github.com/Cypher-Bruce/RISC-V-CPU)，尤其是数据冒险和控制冒险的处理方式，如前递机制（Forwarding）和流水线暂停控制（Stall）。这些设计提供了有价值的思路。

AI代码生成：

- 利用AI（Chat-GPT4o）生成了一些辅助代码，特别是在调试和测试用例生成方面，加速了开发过程。AI帮助快速生成了框架和测试模板，验证了不同边界条件。

遇到的问题与解决方案：

- **数据冒险**：在流水线设计中遇到数据冒险问题，通过参考开源设计并引入前递和暂停机制解决了该问题。
- **AI生成的测试代码问题**：AI生成的测试代码与硬件实际行为不完全匹配，经过手动调整后，确保代码与硬件设计一致，增加了更多的边界测试。

问题及总结

问题及总结

1. **遇到的问题**：
    - **存储指令执行异常**：在测试过程中，发现部分存储指令（如 `sw`）未能正确执行，导致数据未按预期存储到内存中。经过调试，发现是由于内存映射部分的地址计算错误，重新调整了地址计算和存储模块的控制信号后问题解决。
    - **寄存器堆访问错误**：在多个指令访问寄存器堆时，发现读取寄存器值时出现错误，调试后发现是由于信号时序问题，修改了相关的时序控制，确保了寄存器堆的正常读取和写入。
2. **思考**：
    - 在硬件设计中，时序和信号同步至关重要，特别是当多个模块之间存在数据传递时，任何时序问题都会影响到最终的结果。
    - 对于存储和寄存器堆的访问，需要在设计时格外注意各个信号之间的依赖关系，避免由于控制信号的错误配置导致功能异常。
3. **总结**：
    - 通过调试存储和寄存器堆模块，解决了指令执行异常的问题，也增强了对硬件设计中时序控制的理解。
    - 这次开发经历让我们意识到硬件设计不仅仅是模块的拼接，更多的是对各个模块之间时序和信号的精准把控，未来会在这方面更加细致地设计与测试。

Bonus说明

Bonus 对应功能点的设计说明

1. **ECALL 指令功能扩展**
    - 实现了对 `ecall` 系统调用指令的支持，通过 `a7` 和 `a0` 寄存器配合，实现了输出数码管数据、读取拨码开关等功能。
    - 支持自定义扩展功能编号，例如 `a7=1` 为输出，`a7=5` 为输入。
2. **AUIPC 指令支持**
    - 实现了 RISC-V 中 `auipc` 指令的功能，指令执行效果为将 `pc + imm` 的结果写入寄存器。
3. **UART 程序下载功能**
    - 通过 `uart_bmpg_0` 模块实现 UART 下载程序功能，在 FPGA 上支持通过串口将机器码加载到 `prgrom` 中运行。
4. **Pipeline CPU 仿真实现**
    - 设计并实现了五级流水线结构，包括完整的 Forwarding、Stall 控制、Branch Prediction 等功能。
    - 实现调试模块 `Debug`，用于记录每个周期的 PC 和寄存器状态，辅助仿真与调试。

设计思路及与周边模块的关系

1. ECALL 指令功能扩展

- **设计思路**：
    - 通过在控制器中识别 `opcode == 1110011` 且 `inst[31:20] == 0x000` 判断为 `ecall`。
    - 使用寄存器 `a7` 指定功能号（如输出、输入），`a0` 传递参数或结果。
    - 在 `Decoder` 模块中读取 `a0` 和 `a7`；
    - 在 `Dmem` 模块中，根据 `ecall_flag` 和 `a7` 的值选择执行行为，如输出到段码、读取开关数据。
- **与模块关系**：
    - `Controller`：识别 ecall 指令，输出 `ecall_flag`；
    - `Decoder`：读取 `a0` 和 `a7`；
    - `Dmem`：实现系统调用具体逻辑，控制输出或读取。

2. AUIPC 指令支持

- **设计思路**：
    - 在 Controller 中识别 `opcode == 0010111`；
    - 设置 `auipc_flag`，在 ALU 中将 `pc + imm` 结果输出作为 ALU 结果；
    - 使用正常写回流程将结果写入目标寄存器。

- **与模块关系**：
    - `Controller`：检测 auipc 指令，输出 `auipc_flag`；
    - `ALU`：检测 `auipc_flag`，执行 `pc + imm32` 运算；
    - `Decoder`：提供 imm；
    - `CPU_TOP`：保持控制信号流畅传递。

3. UART 程序下载功能

- **设计思路**：
    - 在上电后等待用户按下 `start_pg`，启动 UART 模块；
    - 通过 UART 发送 `.coe` 文件的内容（或二进制指令流）；
    - 数据写入 `prgrom` 指令存储器，完成后启动 CPU 正常运行。
- **与模块关系**：
    - `uart_bmpg_0`：接收串口数据，输出地址、数据和写使能；
    - `IFetch`：判断是否为 UART 下载状态，选择 `upg_clk_i` 或主时钟；
    - `CPU_TOP`：控制 reset 信号，以及指示 `kickOff` 状态启动程序运行。

4. Pipeline CPU 仿真实现

- **设计思路**：
    - 采用经典五级流水结构：IF → ID → EX → MEM → WB；
    - 每个阶段使用 `*_to_*` 模块作为流水线寄存器；
    - 实现数据冒险处理（Forwarding）、load-use hazard（Stall）控制；
    - 控制冒险使用分支预测模块 `PC_Prediction`；
    - 增加 Debug 模块记录每周期 PC 和寄存器状态，便于仿真调试。
- **与模块关系**：
    - `ID_to_EX`、`EX_to_MEM`、`MEM_to_WB` 等：连接各流水线阶段；
    - `Forwarding_EX`、`Forwarding_ALU`：处理数据前递；
    - `Hazard_Detection_Unit`：插入 NOP 保证正确执行；
    - `PC_Prediction`：基于缓存的静态预测实现；
    - `Debug`：输出每周期寄存器值与PC，配合确认按键切换。

核心代码及必要说明

1. ECALL 指令支持

控制器识别 ecall：

```
7'b1110011: begin // SYSTEM 指令
    if (inst[31:20] == 12'b000000000000) begin // 匹配 ecall
        ecall_flag = 1;                        // 设置系统调用标志
        MemorIOtoReg = 1;                      // 结果来自 IO
    end
end
```

Decoder 中访问 `a0` 与 `a7`：

```
always @(*) begin
    a0 = registers[10];  // x10, 作为 ecall 输入/输出
    a7 = registers[17];  // x17, 作为 ecall 调用号
end
```

Dmem 中实现系统调用行为：

```
// 输出到段码管（例如显示数值）
else if (a7 == 32'd1 && ecall_flag) begin
    leds <= 8'b0;
    segments <= a0;  // 将 a0 输出到段码显示
end

// 从拨码开关读取值
else if (a7 == 32'd5 && ecall_flag) begin
    loaded_data = {{24{switch_in[7]}}, switch_in[7:0]}; // lb 方式符号扩展
end
```

2. AUIPC 指令实现

控制器识别 AUIPC 指令：

```
7'b0010111: begin // AUIPC 指令识别
    ALUSrc = 1;
    RegWrite = 1;
    auipc_flag = 1; // 设置 AUIPC 标志
end
```

ALU 中执行 pc + imm：

```
else if (auipc_flag) begin
    ALU_result = pc + imm32; // AUIPC 实现：PC 加立即数
end
```

3. UART 下载模块实现

顶层模块 CPU_TOP 中连接 UART 接口：

```
uart_bmpg_0 uart_bmpg_0_inst (
    .upg_clk_i(upg_clk_i),
    .upg_rst_i(upg_rst),
    .upg_rx_i(rx),              // 串口接收数据
    .upg_clk_o(upg_clk_o),
    .upg_wen_o(upg_wen_o),      // 写使能
    .upg_done_o(upg_done_o),    // 下载完成标志
    .upg_adr_o(upg_adr_o),      // 下载地址
    .upg_dat_o(upg_dat_o)       // 下载数据
);
```

IFetch 中根据 UART 状态加载指令：

```
assign kickOff = upg_rst_i | (~upg_rst_i & upg_done_i); // 判断是否跳转到主时钟运行

prgrom urom(
    .clka(kickOff ? clk : upg_clk_i),        // 指令存储器时钟切换
    .wea(kickOff ? 1'b0 : upg_wen_i),        // 写使能控制
    .addra(kickOff ? prgrom_addr : upg_adr_i), // 地址选择
    .dina(kickOff ? 32'b0 : upg_dat_i),      // 写入数据
    .douta(inst)                             // 输出指令
);
```

4. Pipeline Forwarding、Stall 与预测核心模块

数据前递模块 Forwarding_EX：

```
// MEM/WB 阶段冲突识别
assign MEM_hazard_1_flag = (write_reg_flag_MEM && (write_reg_idx_MEM ==
read_reg_idx_1_EX) && ...);

// 前递标志控制
assign read_data_1_forwarding_flag = MEM_hazard_1_flag || WB_hazard_1_flag;

// 前递数据选择逻辑
if (read_data_1_forwarding_flag)
    if (MEM_hazard_1_flag)
        read_data_1_forwarding = ALU_result_MEM; // 直接从 MEM 阶段转发
    else
        read_data_1_forwarding = mem_to_reg_flag_WB ? read_data_WB : ALU_result_WB;
// 从 WB 阶段选择
```

Stall 检测模块 Hazard_Detection_Unit：

```
assign stall = (write_reg_flag_EX && mem_to_reg_flag_EX &&
                write_reg_idx_EX != 0 &&
                (write_reg_idx_EX == read_reg_idx_1_ID || write_reg_idx_EX ==
read_reg_idx_2_ID));
// Load-use 数据冒险触发暂停信号
```

分支预测模块 PC_Prediction：

```
// 如果命中历史缓存，使用预测值；否则顺序执行
if (current_pc_in_cache_flag && current_pc_in_cache_idx < LRU_capacity)
    program_counter_prediction = LRU_cache[current_pc_in_cache_idx][63:32];
else
    program_counter_prediction = program_counter + 4;
```

调试模块 Debug（记录每周期PC与寄存器）：

```
if (cycle_index < MAX_CYCLE) begin
    pc_log[cycle_index] <= current_pc; // 记录当前周期的 PC
    for (i = 0; i < 32; i = i + 1)
        reg_log[cycle_index][i] <= current_regfile[i]; // 记录寄存器状态
    cycle_index <= cycle_index + 1;
end

// 用户通过开关与确认信号选择查看某个寄存器值
if (confirm && switch != 0) begin
    display_data <= reg_log[view_index][switch];
end
```

测试说明

测试功能	测试场景描述	测试用例说明	测试结果	说明
ECALL 指令	通过 `ecall`+`a7/a0` 实现数码管输出或读取开关	`a7=1,a0=1234` 显示到段码；`a7=5` 从开关读取写回 `a0`	通过	数码管输出正常，开关值正确加载到 `a0` 寄存器
AUIPC 指令	执行 AUIPC 指令并与手动计算的 PC+imm 对比	指令 `auipc x5, 0x10`，应得结果为当前 PC+0x1000	通过	ALU 中成功将 PC 与 imm 相加并写入寄存器
UART 下载	使用 PC 工具通过串口向板端写入 `.coe` 格式的程序指令	下载一个包含多条指令的程序，观察 CPU 是否开始运行	通过	UART 成功写入 `prgrom`，KickOff 拉高，CPU 开始执行
Forwarding	仿真场景中连续依赖的加法指令，无暂停判断转发正确性	add x1,x0,x0 → add x2,x1,x1，x2 应为 x0 值的两倍	通过	Forwarding 模块成功转发数据，无数据冒险延迟
Load-Use Stall	仿真场景中 addi 紧跟 lw 指令，验证是否自动插入暂停	lw x1,0(x0) → addi x2,x1,1，观察流水线暂停一个周期	通过	Hazard_Detection_Unit 检测到冒险并插入 NOP 防止错误读取
分支预测	构造已命中分支和未命中的情况，观察是否提前取指并回退	循环跳转场景：beq、jal 指令配合 IF/ID 查看清空行为	通过	命中时直接跳转，未命中时预测错误可清空并修正，逻辑正常
调试（Debug）模块	使用拨码开关选择寄存器，按下确认键切换视图	选择开关为 `0~31`，观察数码管是否显示对应寄存器值	通过	PC 与寄存器值正确记录与切换显示

问题及总结

在 Bonus 部分的实现过程中，我们面临了以下几个具有挑战性的问题，并从中得到了不少经验总结：

1. Forwarding 与冲突判断逻辑易错

刚开始在写 Forwarding 模块时，逻辑条件设置得过于简单，没有完整考虑 `MEM/WB` 两阶段的转发优先级，导致部分指令结果出现延迟写入的问题。后来我们通过逐周期观察寄存器状态并加入 `Debug` 模块辅助排查，最终采用更清晰的优先级判断条件修复该问题。

2. Stall 判定初期误判频繁

在 load-use hazard 检测中，我们一开始只考虑了 `rs1` 寄存器，忽略了 `rs2` 也可能依赖前一条 `lw` 指令，导致部分 `beq`、`sw` 指令数据未准备好。通过观察波形和修改 `Hazard_Detection_Unit`，我们完善了 `rs1` 和 `rs2` 的判断。

3. 分支预测缓存替换策略设计复杂

我们使用的是 4 路 LRU 缓存进行 PC 跳转预测，刚开始在设计更新策略时顺序不对，导致预测更新混乱。调整之后，我们采用每次命中就移到最前、未命中就插入最前的策略，简单但有效。

最终，我们完成了一个带有数据前递、冲突暂停、基本预测机制的五级流水线模型，从设计、调试到验证的过程都大大提升了我们对 CPU 微结构的理解。