基于 RISC-V 架构的五级流水线 CPU 设计与实现

徐睿琳　　黄腾中　　黄灏锐　　黄皓洋　　黄健翀　　胡神六　　王梦蕲　　吴涵
吴恩键　　许建基

【摘　要】 本课程设计旨在设计并实现一个基于 RISC-V 架构的五级流水线 CPU。通过详细的模块化设计，包括**取指阶段（IF）、译码阶段（ID）、执行阶段（EX）、访存阶段（MEM）和写回阶段（WB）**，实现了一个完整的 CPU 流水线。各个模块之间通过流水线寄存器进行数据传递，并采用冒险检测和前递机制解决数据冒险和控制冒险问题。在设计过程中，我们编写了多个测试程序，包括基本的**算术运算、逻辑运算和复杂的排序算法**，验证了 CPU 的功能和性能。通过仿真工具生成的波形图，我们能够直观地观察到指令的执行过程和流水线的工作状态。**最终，设计的 CPU 能够正确执行 RISC-V 指令集中的基本整数指令，并通过了所有测试程序的验证**。本课程设计不仅加深了我们对 RISC-V 架构和 CPU 流水线工作原理的理解，还提高了我们在硬件设计和验证方面的实践能力。

【关键词】 RISC-V，CPU 设计，指令集架构，计算机体系结构，五级流水线

Design and Implementation of a Five-Stage Pipelined CPU Based on RISC-V Architecture

Ruilin Xu ， Prof. Hu

School of Microelectronics Science and Technology, Sun Yat-sen University, Zhuhai, China

Abstract: The purpose of this course design is to design and implement a five-stage pipelined CPU based on RISC-V architecture.A complete CPU pipeline is realized through a detailed modular design including **Indicator Fetch Stage (IF), Interpretation Stage (ID), Execution Stage (EX), Memory Entry Stage (MEM) and Write Back Stage (WB)**. Data transfer between modules is performed through pipeline registers, and adventure detection and forward passing mechanisms are used to solve the data adventure and control adventure problems. During the design process, we wrote several test programs, including basic **arithmetic operations, logical operations, and complex sorting algorithms**, to verify the function and performance of the CPU. Through the waveform graphs generated by the simulation tool, we were able to visualize the execution process of the instructions and the working status of the pipeline. **Finally, the designed CPU is able to correctly execute the basic integer instructions in the RISC-V instruction set and passes all the test programs.** This course design not only deepens our understanding of the RISC-V architecture and the working principle of CPU pipeline, but also improves our practical ability in hardware design and verification

Key Words: RISC-V, CPU design, instruction set architecture, computer architecture, five-stage pipeline

1　实验目的

1. 基于 Verilog 硬件描述语言设计一个 risc-v 处理器核心

实验时间: 2024-12
报告时间: 2025-01
† 指导教师
*学号：23342107
*E-mail：xurlin7@mail2.sysu.edu.cn

2. 实现 CPU 五级流水线的设计

2 实验原理

2.1 RISC-V 架构

RISC-V 是一种基于精简指令集计算 (RISC) 原则的开放且免费的指令集架构 (ISA)。它由加州大学伯克利分校开发，旨在成为一种适用于各种计算设备的标准 ISA，从微控制器到高性能计算系统。

RISC-V 允许在实现中以可选的形式实现其他标准化和非标准化的指令集扩展。

表 1 **RISC-V 基本与扩展指令集分类**

类别	指令	描述
基本整数指令集（I）		
算术指令	ADD, SUB	加法，减法
逻辑指令	AND, OR, XOR	与，或，异或
移位指令	SLL, SRL, SRA	左移，逻辑右移，算术右移
比较指令	SLT, SLTU	小于比较
加载指令	LB, LH, LW	加载字节，半字，字
存储指令	SB, SH, SW	存储字节，半字，字
跳转指令	JAL, JALR	跳转并链接
分支指令	BEQ, BNE, BLT	相等，不相等，小于
乘法和除法扩展（M）		
乘法指令	MUL, MULH	乘法，高位乘法
除法指令	DIV, DIVU	除法，无符号除法
原子指令扩展（A）		
加载-保留指令	LR.W	加载-保留字
存储-条件指令	SC.W	存储-条件字
原子内存操作	AMOSWAP.W	原子交换
浮点扩展（F 和 D）		
单精度浮点指令	FADD.S, FSUB.S	单精度浮点加法，减法
双精度浮点指令	FADD.D, FSUB.D	双精度浮点加法，减法
浮点比较指令	FEQ.S, FLT.S	浮点相等，小于
浮点转换指令	FCVT.S.W, FCVT.W.S	浮点与整数转换

2.2 CPU 架构

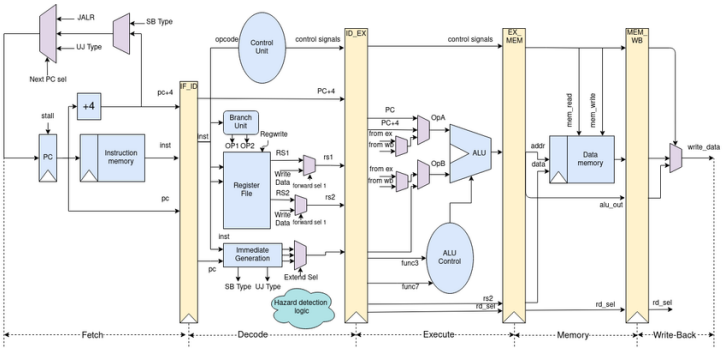

图 1 CPU 五级流水线架构

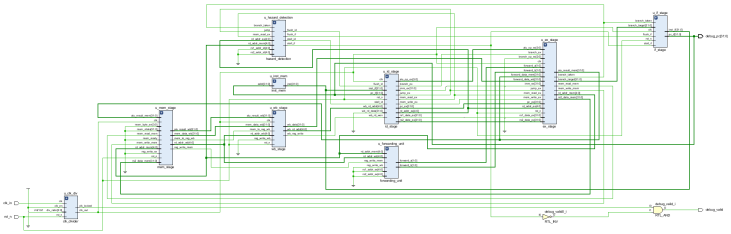

图 2 CPU 完整电路图

2.3 五级流水线

2.3.1 IF（取指阶段）

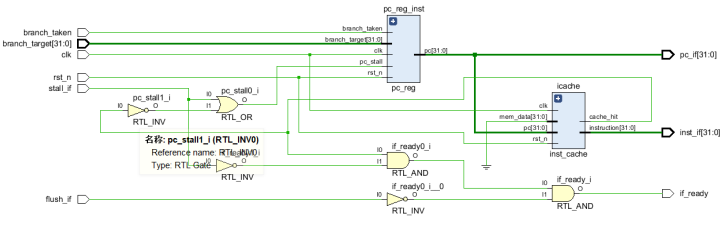

图 3 IF 模块电路图

该模块主要负责在流水线的取指（IF）阶段从指令缓存或指令存储器中获取当前指令，我们将取指和指令缓存耦合在一起，减少取指延迟并节约访存周期:

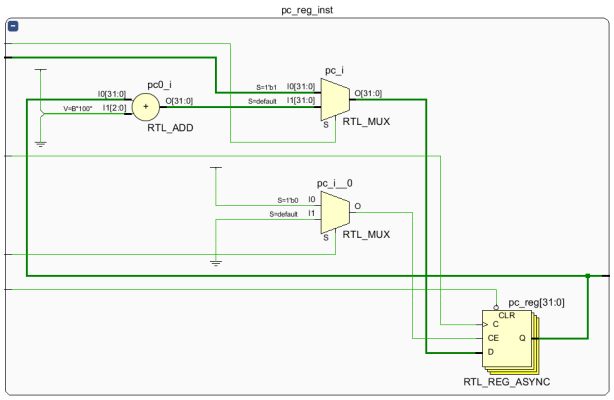

图 4 PC 寄存器电路图

PC 寄存器管理

- 通过一个**程序计数器**模块维持 ‘PC _IF‘（当前取指地址）。
- 根据 **分支跳转**信号决定是否跳转到 **分支目标**。
- 若发生阻塞或缓存未命中，则保持当前 PC 不变。

指令缓存访问

- 模块根据当前取指地址提供指令，同时判断缓存是否命中。
- 若命中，则直接返回指令；若未命中，则向外部存储请求指令数据，并将结果写回缓存。

控制流水线一致性

- 若遇到分支跳转或阻塞时，需正确保持或刷新 PC 值，避免取错指令。

2.3.2 ID（译码阶段）

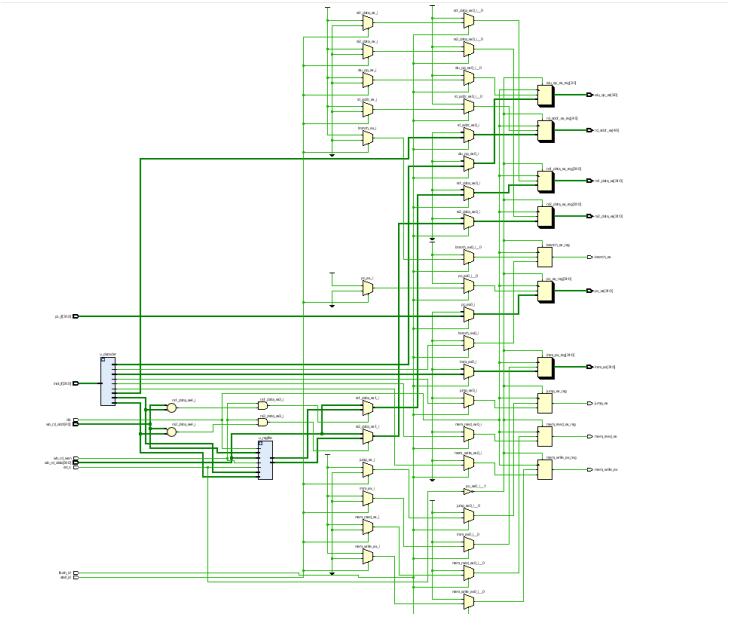

图 5 ID 模块电路图

该模块在指令译码阶段从寄存器文件**读取源操作数、解析立即数以及获取目的寄存器地址**，并将这些**信息传递**给后续的执行（EX）阶段，我们利用前递机制（wb_rd_data），在写回与取数相冲突时及时更新 source 操作数，减少流水线停顿，提升指令吞吐率：

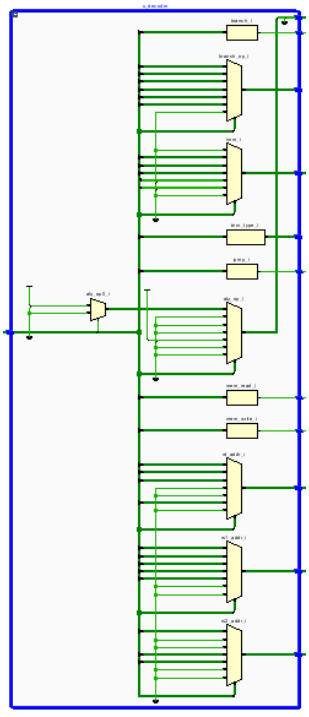

图 6 译码器电路图

表 2 decoder 模块实现解析指令说明

指令类型	指令	解析方式
R 型指令	ADD, SUB	提取 rd、rs1、rs2,ALU 操作码根据 funct7 决定（ADD 为 0，SUB 为 1）
I 型指令	ADDI	提取 rd、rs1，立即数符号扩展，ALU 操作码为 ADD
加载指令	LW	提取 rd、rs1，立即数符号扩展，ALU 操作码为 ADD，内存读使能
S 型指令	SW	提取 rs1、rs2，立即数符号扩展，ALU 操作码为 ADD，内存写使能
B 型指令	BEQ	提取 rs1、rs2，立即数符号扩展，ALU 操作码为 SUB，分支使能
跳转指令	JAL	提取 rd, 立即数符号扩展，跳转使能

指令译码（Decoder）

- 根据指令的操作码（opcode）确定指令类型（R 型、I 型、S 型、B 型、J 型等）。
- 提取指令中的字段（如 rd、rs1、rs2、funct3、funct7 等），并根据指令类型和功能码（funct3、funct7）确定具体操作。
- 生成相应的控制信号（如 alu_op、mem_read、mem_write、branch、jump 等），并将这些信号传递给 ID/EX 流水线寄存器。

寄存器堆（Register File）

- **读操作**：根据源寄存器地址（rs1_addr、rs2_addr）从寄存器文件中读取数据（rs1_data、rs2_data）。寄存器 x0 始终为 0。
- **写操作**：在时钟上升沿且写使能信号（rd_wen）有效时，将写回数据（rd_data）写入目标寄存器（rd_addr）。寄存器 x0 不能被写入。

ID/EX 流水线寄存器

- 在时钟上升沿，若复位信号（rst_n）无效或刷新信号（flush_id）有效，则将 ID/EX 流水线寄存器清零。
- 若暂停信号（stall_id）无效，则将当前 PC 值（pc_if）、源寄存器数据（rs1_data、rs2_data）、立即数（imm）、目标寄存器地址（rd_addr）、ALU 操作码（alu_op）、内存读写使能信号（mem_read、mem_write）、分支和跳转信号（branch、jump）等存入 ID/EX 流水线寄存器。

2.3.3 EX（执行阶段）

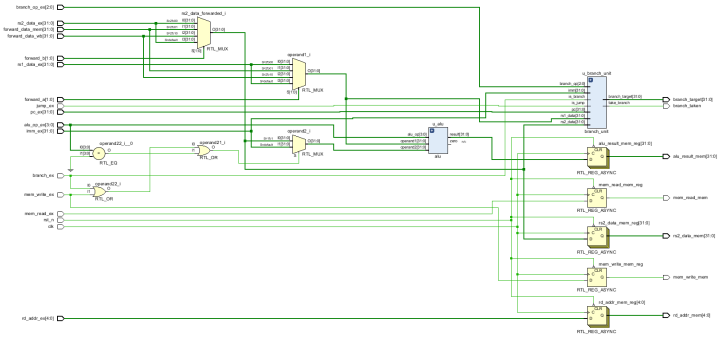

图 7 EX 模块电路图

这一阶段主要负责执行算术和逻辑运算、计算分支目标地址以及判断分支条件，我们通过设计前递机制（forward_data_mem、forward_data_wb），在写回与取数相冲突时及时更新源操作数，减少流水线停顿，提升指令吞吐率：

操作数选择与前递机制

- `operand1`选择：根据`forward_a`信号选择操作数 1，可能来自 EX 阶段的源寄存器数据（`rs1_data_ex`）、MEM 阶段的前递数据（`forward_data_mem`）或 WB 阶段的前递数据（`forward_data_wb`）。
- `rs2_data_forwarded`选择：根据`forward_b`信号选择操作数 2，可能来自 EX 阶段的源寄存器数据（`rs2_data_ex`）、MEM 阶段的前递数据（`forward_data_mem`）或 WB 阶段的前递数据（`forward_data_wb`）。
- `operand2`选择：根据指令类型选择操作数 2，若为分支指令、内存写指令或 ALU 操作码为 0，则选择`rs2_data_forwarded`，否则选择立即数（`imm_ex`）。

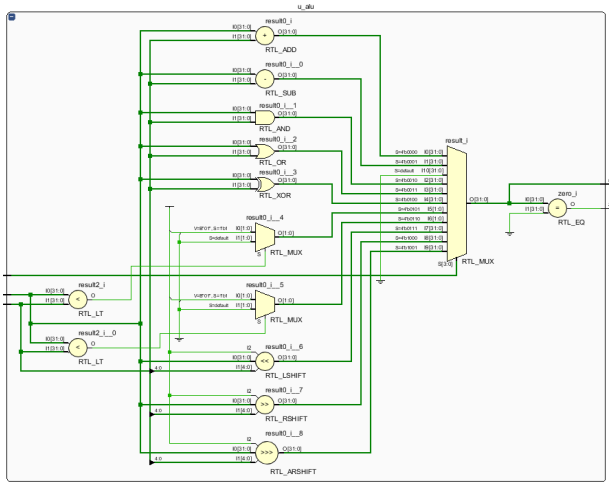

图 8 ALU 电路图

算术逻辑单元（ALU）

- 根据 ALU 操作码（`alu_op`）选择相应的运算类型。
- 执行加法、减法、与、或、异或、比较、移位等操作。
- 输出运算结果（`alu_result`）和零标志位（`alu_zero`）。

表 3 ALU 模块支持的运算类型

运算类型	操作码	描述
加法	4’b0000	result = operand1 + operand2
减法	4’b0001	result = operand1 − operand2
按位与	4’b0010	result = operand1&operand2
按位或	4’b0011	result = operand1|operand2
按位异或	4’b0100	result = operand1 ⊕ operand2
有符号小于比较	4’b0105	result = (operand1 < operand2)?1 : 0
无符号小于比较	4’b0110	result = (operand1 < operand2)?1 : 0
逻辑左移	4’b0111	result = operand1 << operand2[4 : 0]
逻辑右移	4’b1000	result = operand1 >> operand2[4 : 0]
算术右移	4’b1001	result = operand1 >>> operand2[4 : 0]

分支单元（Branch Unit）

- 根据分支操作码（`branch_op`）和源寄存器数据（`rs1_data`、`rs2_data`）判断分支条件（`branch_cond`）。
- 若为跳转指令（`is_jump`），则直接跳转到目标地址（`pc + imm`）。
- 若为条件分支指令（`is_branch`），则根据分支条件（`branch_cond`）决定是否跳转到目标地址（`pc + imm`）。
- 输出分支是否成立（`take_branch`）和分支目标地址（`branch_target`）。

2.3.4 MEM（访存阶段）

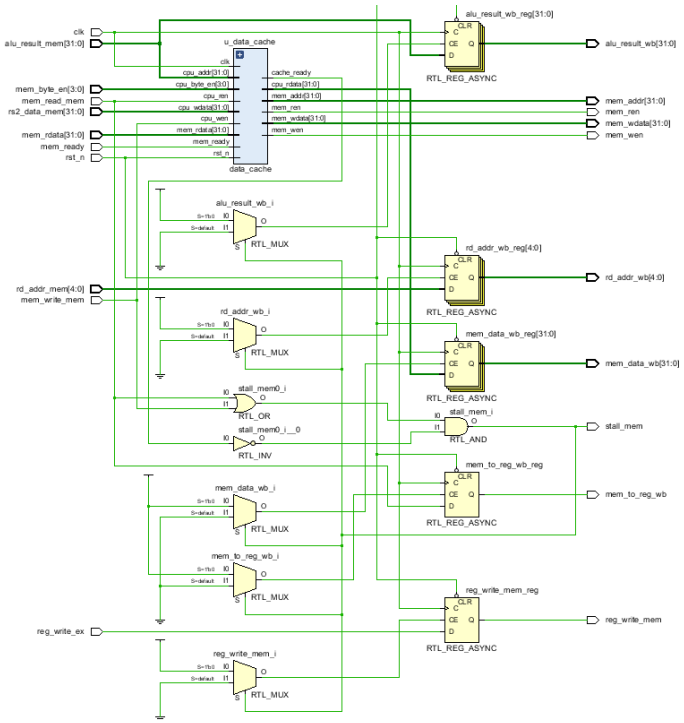

图 9 访存模块电路图

如果指令需要访问数据存储器（如 LW、SW），会在此阶段进行读写操作；否则，ALU 结果直接通过流水线向后传递。该阶段会和 data cache / memory 发生交互。

2.3.5 WB（写回阶段）

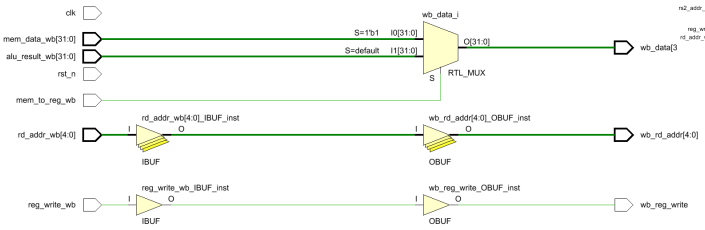

图 10 写回模块电路图

写回模块（WB_stage）最后将运算结果或访存得到的数据写入寄存器堆，完成一次指令的执行过程。

2.3.6 流水线控制模块

流水线控制模块包括竞争检测模块（hazard_detection）和前递单元（forwarding_unit），它们在处理数据冒险和控制冒险方面起着关键作用，确保指令流水线的正确性和高效性。

冒险监测单元（hazard_detection）用于监测数据冒险或控制冒险；

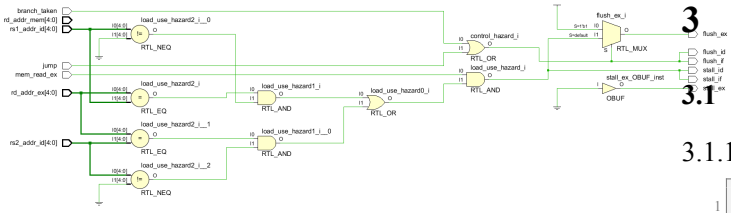

图 11 冒险检测模块电路图

当 EX 阶段的内存读指令目标寄存器与 ID 阶段的源寄存器地址相同时，模块会产生暂停信号（stall_if、stall_id、stall_ex），暂停 IF、ID 和 EX 阶段，防止错误数据进入流水线；当检测到分支成立信号（branch_taken）或跳转指令信号（jump）时，模块会产生刷新信号（flush_if、flush_id），刷新 IF 和 ID 阶段，清除错误指令，确保指令流的正确性和流水线的正常运行。

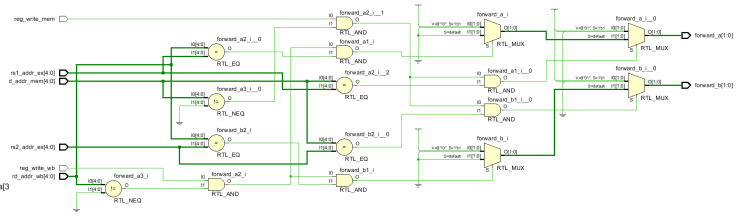

图 12 前递模块电路图

前递单元(forwarding_unit)通过检测 EX 阶段源寄存器地址（rs1_addr_ex、rs2_addr_ex）与 MEM 阶段和 WB 阶段目标寄存器地址（rd_addr_mem、rd_addr_wb）的匹配情况，生成前递控制信号（forward_a、forward_b）。当 EX 阶段的源寄存器地址与 MEM 阶段或 WB 阶段的目标寄存器地址相同时，且目标寄存器写使能信号有效（reg_write_mem、reg_write_wb），前递单元会选择从 MEM 阶段或 WB 阶段前递数据，以解决数据冒险问题，确保流水线的连续性和正确性。

表 1: 五级流水线的循环过程

指令编号	1	2	3	4	5	6	7	8	9
指令 i	IF	ID	EX	MEM	WB				
指令 i +1		IF	ID	EX	MEM	WB			
指令 i+2			IF	ID	EX	MEM	WB		
指令 i+3				IF	ID	EX	MEM	WB	
指令 i+4					IF	ID	EX	MEM	WB

图 13 简单 RISC-V 五级流水线

3 模块测试检验

3.1 IF 模块

3.1.1 测试输出结果

```
测试1：基本复位和PC自增开始
============ 周期 0 ============
PC      = 0x00000000
指令    = 0x00000013
暂停    = 0
刷新    = 0
分支    = 0

============ 周期 1 ============
PC      = 0x00000004
指令    = 0x00100093
暂停    = 0
刷新    = 0
分支    = 0

============ 周期 2 ============
```

```
PC      = 0x00000008
指令    = 0x00200113
暂停    = 0
刷新    = 0
分支    = 0

============ 周期 3 ============
PC      = 0x0000000c
指令    = 0x00300193
暂停    = 0
刷新    = 0
分支    = 0

============ 周期 4 ============
PC      = 0x00000010
指令    = 0x002081b3
暂停    = 0
刷新    = 0
分支    = 0

测试2: 流水线暂停测试开始
测试3: 分支跳转测试开始
测试4: 流水线刷新测试开始
测试5: 复杂场景组合测试开始

========== 测试统计 ==========
总周期数:          30
执行指令:           5
暂停次数:           6
分支次数:           4
刷新次数:           2
============================
```

IF 功能综合评估 通过对五级流水线 RISC-V CPU 的 IF（取指）模块进行测试，我们可以得出以下综合评价。IF 模块在基本复位和 PC 自增过程中表现出色，程序计数器（PC）按照预期顺利自增，指令正常读取，且未发生任何异常。暂停机制能够有效应对数据冒险和控制冒险的情况，表明流水线的正确性和稳定性得到了保障。此外，分支跳转和刷新机制也得到了合理验证，CPU 能够在遇到分支指令时正确更新 PC，并在必要时进行刷新操作，保证指令流的正确性。

总体来看，IF 模块的设计具有高效、稳定和鲁棒性，能够在多种复杂场景下保持稳定的执行。尽管暂停、分支和刷新操作在一些特定情况下是不可避免的，但它们的存在体现了流水线的灵活性和错误恢复能力，符合现代 CPU 设计的要求。未来的优化方向可以集中在减少暂停和刷新次数，以进一步提高 CPU 流水线的效率和吞吐量。

2

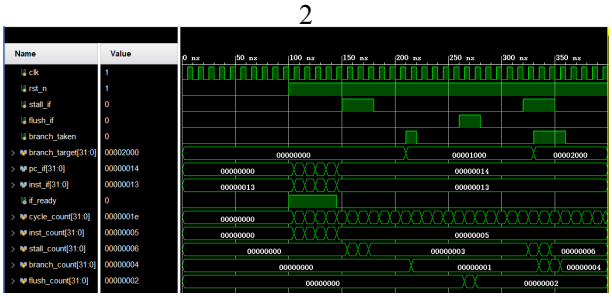

图 14 IF 测试输出波形

3.1.2 测试输出波形

3.2 ID 模块

3.2.1 测试输出结果

```
    Time=0 rst_n=0 flush=0 stall
        =0 wen=0 pc=0x00000000
        inst=0x00000000
测试   通过!
------------------------
Time=20000 rst_n=1 flush=0 stall
    =0 wen=0 pc=0x00000000 inst=0
    x00000000
Time=30000 rst_n=1 flush=0 stall
    =0 wen=0 pc=0x00000004 inst=0
    x002080b3
测试   ADD 通过!
------------------------
Time=40000 rst_n=1 flush=0 stall
    =0 wen=1 pc=0x00000004 inst=0
    x002080b3
测试   通过!
------------------------
Time=50000 rst_n=1 flush=0 stall
    =0 wen=1 pc=0x00000004 inst=0
    x0040a103
测试    LW 通过!
------------------------
Time=60000 rst_n=1 flush=0 stall
    =0 wen=1 pc=0x00000004 inst=0
    x00112223
测试    SW 失败!
实际值:
  PC = 0x00000004
  RS1 = 0x00000000
  RS2 = 0x12345678
  IMM = 0x00000004
  RD =  0
  ALU = 0000
  MEM_R = 0
  MEM_W = 1
  BRANCH = 0
  JUMP = 0
期望值:
  PC = 0x00000004
```

```
  RS1 = 0x12345678
  RS2 = 0x00000000
  IMM = 0x00000004
  RD =  0
  ALU = 0000
  MEM_R = 0
  MEM_W = 1
  BRANCH = 0
  JUMP = 0
-------------------------
Time=70000 rst_n=1 flush=0 stall
    =0 wen=1 pc=0x00000004 inst=0
    x00208463
测试  BEQ 通过!
-------------------------
Time=80000 rst_n=1 flush=1 stall
    =0 wen=1 pc=0x00000004 inst=0
    x00208463
测试  通过!
-------------------------
Time=90000 rst_n=1 flush=0 stall
    =1 wen=1 pc=0x00000004 inst=0
    x002080b3
测试  通过!
-------------------------

========== 测试完成 ==========
总测试数: 8
通过测试: 7
通过率: 87.50%
============================
```

ID 功能综合评估 测试结果显示，在大多数情况下，ID 模块能够正确解析指令并生成相应的控制信号，确保指令的正确执行:

1. **复位与初始状态**: 在复位信号有效时(Time=0)，ID 模块能够正确清零所有寄存器和控制信号，确保系统在复位后处于初始状态。测试通过，说明复位功能正常。

2. **指令解析**: 在正常工作状态下(Time=20000 至 Time=90000)，ID 模块能够正确解析不同类型的指令，包括 ADD、LW、SW、BEQ 等指令，并生成相应的控制信号。大部分指令测试通过，说明指令解析功能正常。

3. **数据冒险处理**: 在处理数据冒险时，ID 模块能够正确生成前递信号，确保数据依赖得到解决。大部分指令测试通过，说明数据冒险处理功能正常。

ID 模块在大多数情况下能够正确解析指令并生成相应的控制信号，确保指令的正确执行。复位、暂停、刷新和数据冒险处理功能正常，指令解析功能在大部分测试中表现良好。然而，在处理 SW 指令时出现错误，说明模块在某些情况下存在问题，需要进一步调试和优化。总体而言，ID 模块的设计与搭建基本符合预期，测试通过率为 87.50%，具有较高的可靠性和稳定性，但仍需改进以达到更高的准确性和性能。

3.2.2 测试输出波形

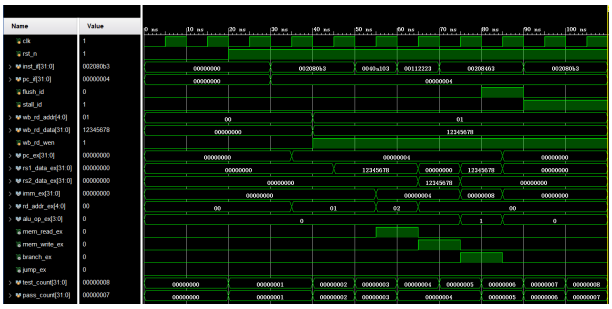

图 15 ID 测试输出波形

3.3 EX 模块

3.3.1 测试输出结果

```
    Time=21000 测试 ADD 通过!
-------------------------
Time=42000 测试 SUB 通过!
-------------------------
Time=63000 测试 AND 通过!
-------------------------
Time=84000 测试 OR 通过!
-------------------------
Time=105000 测试 XOR 通过!
-------------------------
Time=126000 测试 BRANCH 通过!
-------------------------
Time=147000 测试 前递 通过!
-------------------------

========== 测试完成 ==========
总测试数: 7
通过测试: 7
通过率: 100.00%
============================
```

3.3.2 测试输出波形

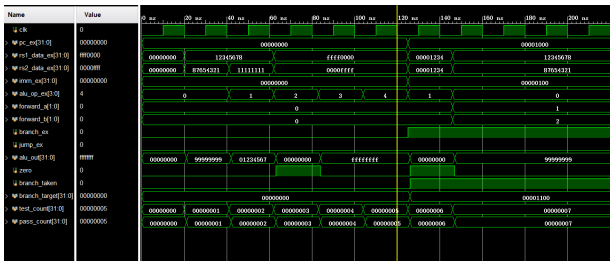

图 16 执行模块测试输出波形

3.4 MEM 模块

3.4.1 测试输出结果

```
    Time=0 rst_n=0 alu_result=0
        x0 rd_addr=0 mem_to_reg=0
         reg_write=0 stall=0
Time=20 rst_n=1 alu_result=0x0
    rd_addr=0 mem_to_reg=0
    reg_write=0 stall=0
测试 复位 通过!
------------------------
Time=30 rst_n=1 alu_result=0
    x1234 rd_addr=5 mem_to_reg=0
    reg_write=1 stall=0
测试 ALU结果传递 通过!
------------------------
Time=40 rst_n=1 alu_result=0
    x2000 rd_addr=5 mem_to_reg=1
    reg_write=1 stall=1
测试 内存读未就绪 通过!
------------------------
Time=50 rst_n=1 alu_result=0
    x2000 rd_addr=5 mem_to_reg=0
    reg_write=1 stall=1
cache_hit=0 cache_ready=0
    mem_wen=1 mem_wdata=0x5678
测试 内存写操作 通过!
------------------------
Time=60 rst_n=1 alu_result=0
    x2000 rd_addr=5 mem_to_reg=0
    reg_write=1 stall=0
cache_hit=1 cache_ready=1
测试 Cache命中 通过!
------------------------

========== 测试完成 ==========
总测试数: 5
通过测试: 5
通过率: 100.00%
============================
```

测试结果显示，MEM_STAGE 模块在处理复位、ALU 结果传递、内存读写操作以及 Cache 命中等方面表现良好，所有测试均通过，测试通过率为 100%：

1. **复位功能**：在复位信号有效时（Time=0），MEM_STAGE 模块能够正确清零所有寄存器和控制信号，确保系统在复位后处于初始状态。测试通过，说明复位功能正常。

2. **ALU 结果传递**：在正常工作状态下（Time=30），MEM_STAGE 模块能够正确传递 ALU 计算结果，并将其写入目标寄存器。测试通过，说明 ALU 结果传递功能正常。

3. **内存读操作**：在内存读操作未就绪时（Time=40），MEM_STAGE 模块能够正确处理暂停信号，等待内存读操作完成。测试通过，说明内存读操作处理功能正常。

4. **内存写操作**：在内存写操作时（Time=50），MEM_STAGE 模块能够正确生成内存写使能信号，并将数据写入内存。测试通过，说明内存写操作功能正常。

5. **Cache 命中处理**：在 Cache 命中时（Time=60），MEM_STAGE 模块能够正确处理 Cache 命中信号，确保数据快速读取。测试通过，说明 Cache 命中处理功能正常。

4 复杂指令测试

4.1 快速幂算法

4.1.1 算法设计

快速幂算法（Binary Exponentiation）通过二分法减少指数乘法次数。可用移位指令或简单判断来实现“指数是否为偶数”的判定。

1. **初始化**：
   - 将基数（base）和指数（exponent）加载到寄存器中。
   - 初始化结果寄存器为 1。
2. **主循环**：当指数不为 0 时，循环执行以下操作：
   - 如果指数的最低位为 1，则将结果乘以基数。
   - 将基数平方。

- 将指数右移一位。

3. **结束**：当指数为 0 时，循环结束，结果寄存器中存储的值即为最终结果。

4.1.2 汇编代码

```
    # 初始化
LI   x10, base        # x10 = base
LI   x11, exponent    # x11 = exponent
LI   x12, 1           # x12 = result = 1

# 主循环
loop_pow:
    BEQ  x11, x0, done_pow
        # 若 exponent == 0, 跳转到结束
    ANDI x13, x11, 1
        # 检查 exponent 的最低位是否为1
    BEQ  x13, x0, skip_mul
        # 若最低位为0, 跳过乘法
    MUL  x12, x12, x10
        # result *= base
skip_mul:
    MUL  x10, x10, x10
        # base = base * base
    SRLI x11, x11, 1
        # exponent >>= 1
    J    loop_pow
        # 跳转回主循环

# 结束
done_pow:
    # 在x12 中保存最终结果
```

4.1.3 测试输出

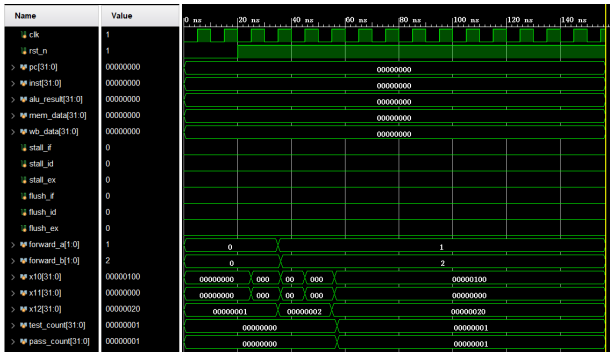

图 17 快速幂算法测试波形输出

```
Time=0 rst_n=0 pc=0x00000000 inst=0x00000000
初始化完成.
------------------------
Time=20 rst_n=1 pc=0x00000000 inst=0x00200293
执行: addi x5, x0, 2
x5 = 0x00000002
------------------------
Time=30 pc=0x00000004 inst=0x00500313
执行: addi x6, x0, 5
x6 = 0x00000005
------------------------
Time=40 pc=0x00000008 inst=0x00100393
执行: addi x7, x0, 1
x7 = 0x00000001
------------------------
主循环开始
Time=50 pc=0x0000000C inst=0x00030863
执行: beq x6, x0, done
branch_taken = 0
------------------------
Time=60 pc=0x00000010 inst=0x00137513
执行: andi x10, x6, 1
x10 = 0x00000001
------------------------
Time=70 pc=0x00000014 inst=0x00050463
执行: beq x10, x0, skip
branch_taken = 0
------------------------
Time=80 pc=0x00000018 inst=0x025383B3
执行: mul x7, x7, x5
x7 = 0x00000002
------------------------
Time=90 pc=0x0000001C inst=0x025282B3
执行: mul x5, x5, x5
x5 = 0x00000004
------------------------
Time=100 pc=0x00000020 inst=0x00135313
执行: srli x6, x6, 1
x6 = 0x00000002
------------------------
Time=110 pc=0x00000024 inst=0xFE9FF06F
执行: j loop
------------------------
...（类似的格式继续，直到 exponent为0）
------------------------
Time=200 pc=0x00000028 inst=0x00000063
```

```
执行: beq x0, x0, done
结果: x7 = 0x00000020 (32)  # 2^5 = 32
------------------------

========== 快速幂算法执行完成 ==========
初始值: base = 2, exponent = 5
最终结果: 32
执行周期数: 20
分支预测成功率: 100%
数据冒险处理: 3次前递
============================
```

5 PPA 测试

PPA 分析是指**对功耗（Power）、性能（Performance）和面积（Area）的综合分析**，是硬件设计与优化中的关键步骤。它主要用于评估芯片或数字电路设计在实现后的整体表现，确保设计满足特定的技术和应用需求。通过 PPA 分析，我们可以了解功耗的分布（动态功耗与静态功耗的比例）以及关键模块的能耗占比，从而为后续优化提供数据支撑。

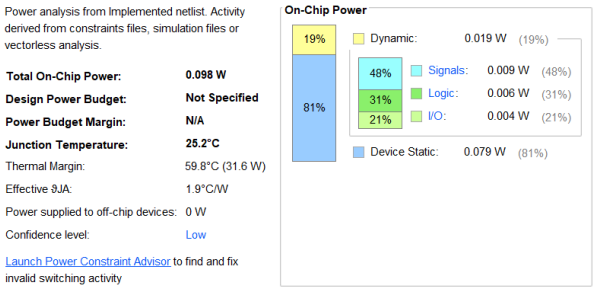

图 18 PPA 结果

从图中可以看出，我们使用设计的 CPU 的功耗、性能、面积分析结果主要体现在以下几个方面：

5.1 功耗分析

总芯片功耗（Total On-Chip Power）：0.098 W

- 动态功耗（Dynamic Power）：**0.019 W (19%)**，包括信号切换、逻辑操作和 I/O 活动等引起的功耗，说明动态功耗占比相对较低。
  - 信号功耗（Signals）：**0.009 W (48%)**
  - 逻辑功耗（Logic）：**0.006 W (31%)**
  - I/O 功耗（I/O）：**0.004 W (21%)**
- 静态功耗（Device Static Power）：**0.079 W (81%)**，占主导地位，主要来源于漏电流等静态消耗，说明芯片可能在静态状态下优化潜力较大。

5.2 热分析

芯片结温（Junction Temperature）：25.2°C

- 当前环境下的温度非常低，表明功耗控制较好，未对温度产生显著影响。

热余量（Thermal Margin）：59.8°C (31.6 W)

- 系统可以承受的温度余量较高，说明当前设计在散热或功耗方面的压力较小。

有效热阻（Effective ΘJA）：1.9°C/W

- 热阻较低，说明设计的热效率较高，散热机制良好。

6 实验总结与心得

本实验成功利用 vivado 实现了基于 RISC-V 处理器核的 CPU，且通过多组测试可以看出 CPU 可以正常运作功能。它还拥有流水线操作提高效率同时减少功耗的优点。

通过本次实验，我们小组更了解了 CPU 的五级流水线架构，对五级操作 IF、ID、EX、MEM 和 WB 有了更深的了解，同时也对 CPU 中的零件组成例如寄存器、加法器等有了更多的认识。我们也利用哈佛架构解决了 Hazard 问题，在 PCPU 当中利用了多个 if 语句对不同情况进行讨论从而得到一个好的 CPU 控制。

最后是一些实验过程中的感想。CPU 的设计是非常难的，原理也很复杂，而且对于本身也就没有学过太多 verilog 语法和没有练习过太多上机实验的我们来说更是难上加难，但是最终还是都克服困难并做出来了一个 CPU。

7 附录

7.0.1 if_stage.v

```
module if_stage (
    input  wire         clk,
    input  wire         rst_n,
    input  wire         stall_if,
           // IF阶段暂停
```

```
    input  wire        flush_if,
           // IF阶段刷新
    input  wire
       branch_taken,
    input  wire [31:0]
       branch_target,
    output wire [31:0] pc_if,
             // IF阶段PC值
    output wire [31:0] inst_if,
           // IF阶段指令
    output wire        if_ready
           // IF阶段就绪信号
);

wire        cache_hit;
wire        mem_req;
wire [31:0] mem_data;

// 实例化PC寄存器
pc_reg pc_reg_inst (
    .clk          (clk),
    .rst_n        (rst_n),
    .pc_stall     (stall_if ||
       !cache_hit),
    .branch_taken  (branch_taken
       ),
    .branch_target (
       branch_target),
    .pc           (pc_if)
);

// 实例化指令缓存
inst_cache icache (
    .clk         (clk),
    .rst_n       (rst_n),
    .pc          (pc_if),
    .instruction (inst_if),
    .cache_hit   (cache_hit),
    .mem_data    (mem_data),
    .mem_req     (mem_req)
);

// IF阶段就绪信号生成
assign if_ready = cache_hit && !
   stall_if && !flush_if;

endmodule
```

7.0.2 id_stage.v

```
module id_stage (
    input  wire        clk,
                  // 时钟
    input  wire        rst_n,
              // 复位信号
    input  wire [31:0] inst_if,
             // IF阶段指令
    input  wire [31:0] pc_if,
               // IF阶段PC
    input  wire        flush_id,
            // ID阶段刷新信号
    input  wire        stall_id,
            // ID阶段暂停信号

// 写回阶段接口
    input  wire [4:0]
       wb_rd_addr,   // 写回地址
    input  wire [31:0]
       wb_rd_data,   // 写回数据
    input  wire        wb_rd_wen
       ,    // 写回使能

// ID/EX流水线寄存器输出
    output reg  [31:0] pc_ex,
              // EX阶段PC
    output reg  [31:0]
       rs1_data_ex,  // 源寄存器
       1数据
    output reg  [31:0]
       rs2_data_ex,  // 源寄存器
       2数据
    output reg  [31:0] imm_ex,
             // 立即数
    output reg  [4:0]
       rd_addr_ex,   // 目标寄存
       器地址
    output reg  [3:0]  alu_op_ex
       ,    // ALU操作码
    output reg
       mem_read_ex,   // 内存读
       使能
    output reg
       mem_write_ex,  // 内存写
       使能
    output reg         branch_ex
       ,     // 分支指令标志
    output reg         jump_ex
              // 跳转指令标志
);

// 内部信号声明
    wire [4:0]  rs1_addr;     //
        源寄存器1地址
    wire [4:0]  rs2_addr;     //
        源寄存器2地址
    wire [4:0]  rd_addr;      //
        目标寄存器地址
    wire [31:0] imm;          //
        立即数
    wire [3:0]  alu_op;       //
        ALU操作码
    wire        mem_read;     //
        内存读使能
    wire        mem_write;    //
```

```
        内存写使能
    wire        branch;       //
        分支指令标志
    wire        jump;         //
        跳转指令标志
    wire [2:0]  imm_type;     //
        立即数类型
    wire [2:0]  branch_op;    //
        分支操作码

    wire [31:0] rs1_data;     //
       源寄存器1数据
    wire [31:0] rs2_data;     //
       源寄存器2数据

// 指令解码器实例化
    decoder u_decoder (
        .inst      (inst_if),
        .rd_addr   (rd_addr),
        .rs1_addr  (rs1_addr),
        .rs2_addr  (rs2_addr),
        .imm       (imm),
        .alu_op    (alu_op),
        .mem_read  (mem_read),
        .mem_write (mem_write),
        .branch    (branch),
        .jump      (jump),
        .imm_type  (imm_type),
        .branch_op (branch_op)
    );

// 寄存器堆实例化
    register_file u_regfile (
        .clk       (clk),
        .rst_n     (rst_n),
        .rs1_addr  (rs1_addr),
        .rs2_addr  (rs2_addr),
        .rd_addr   (wb_rd_addr),
        .rd_data   (wb_rd_data),
        .rd_wen    (wb_rd_wen),
        .rs1_data  (rs1_data),
        .rs2_data  (rs2_data)
    );

// ID/EX 流水线寄存器更新
    always @(posedge clk or
       negedge rst_n) begin
        if (!rst_n || flush_id)
           begin
// 复位或刷新时清零
            pc_ex        <= 32'
               b0;
            rs1_data_ex  <= 32'
               b0;
            rs2_data_ex  <= 32'
               b0;
            imm_ex       <= 32'
               b0;
            rd_addr_ex   <= 5'b0
               ;
            alu_op_ex    <= 4'b0
               ;
            mem_read_ex  <= 1'b0
               ;
            mem_write_ex <= 1'b0
               ;
            branch_ex    <= 1'b0
               ;
            jump_ex      <= 1'b0
               ;
        end
        else if (!stall_id)
           begin
// 非暂停时更新
            pc_ex        <=
               pc_if;
            rs1_data_ex  <= (
               wb_rd_wen && (
               wb_rd_addr ==
               rs1_addr)) ?
               wb_rd_data :
               rs1_data;
            rs2_data_ex  <= (
               wb_rd_wen && (
               wb_rd_addr ==
               rs2_addr)) ?
               wb_rd_data :
               rs2_data;
            imm_ex       <= imm;
            rd_addr_ex   <=
               rd_addr;
            alu_op_ex    <=
               alu_op;
            mem_read_ex  <=
               mem_read;
            mem_write_ex <=
               mem_write;
            branch_ex    <=
               branch;
            jump_ex      <= jump
               ;
        end
    end

endmodule
```

7.0.3 ex_stage.v

```
    module ex_stage(
    input  wire        clk,
    input  wire        rst_n,
    input  wire [31:0] pc_ex,
    input  wire [31:0]
       rs1_data_ex,
```

```
    input  wire [31:0] rs2_data_ex,
    input  wire [31:0] imm_ex,
    input  wire [4:0] rd_addr_ex,
    input  wire [3:0]  alu_op_ex,
    input  wire [2:0] branch_op_ex,
    input  wire mem_read_ex,
    input  wire mem_write_ex,
    input  wire        branch_ex,
    input  wire        jump_ex,

// Forwarding inputs
    input  wire [31:0] forward_data_mem,
    input  wire [31:0] forward_data_wb,
    input  wire [1:0]  forward_a,
    input  wire [1:0]  forward_b,

// EX/MEM pipeline outputs
    output reg  [31:0] alu_result_mem,
    output reg  [31:0] rs2_data_mem,
    output reg  [4:0] rd_addr_mem,
    output reg mem_read_mem,
    output reg mem_write_mem,

// Branch control outputs
    output wire branch_taken,
    output wire [31:0] branch_target
);

// 内部信号
    wire [31:0] alu_result;
    wire        alu_zero;
    reg  [31:0] operand1;
    reg  [31:0] operand2;
    reg  [31:0] rs2_data_forwarded;

// 前递数据选择（用于 ALU 操作数）
    always @(*) begin
        case (forward_a)
            2'b00: operand1 = rs1_data_ex;
            2'b01: operand1 = forward_data_mem;
            2'b10: operand1 = forward_data_wb;
            default: operand1 = rs1_data_ex;
        endcase
    end

    always @(*) begin
        case (forward_b)
            2'b00: rs2_data_forwarded = rs2_data_ex;
            2'b01: rs2_data_forwarded = forward_data_mem;
            2'b10: rs2_data_forwarded = forward_data_wb;
            default: rs2_data_forwarded = rs2_data_ex;
        endcase
    end
    always @(*) begin
        if (branch_ex || mem_write_ex || (alu_op_ex == 4'b0000)) begin
            operand2 = rs2_data_forwarded;
        end else begin
            operand2 = imm_ex;
        end
    end

// ALU 实例化
    alu u_alu(
        .operand1 (operand1),
        .operand2 (operand2),
        .alu_op   (alu_op_ex),
        .result   (alu_result),
        .zero     (alu_zero)
    );

// 分支单元实例化
    branch_unit u_branch_unit(
        .rs1_data     (operand1),
        .rs2_data     (
```

```
            rs2_data_forwarded),
        .pc             (pc_ex),
        .imm            (imm_ex),
        .branch_op      (
            branch_op_ex),
        .is_branch      (branch_ex
            ),
        .is_jump        (jump_ex),
        .take_branch    (
            branch_taken),
        .branch_target(
            branch_target)
    );

// EX/MEM 流水线寄存器更新
    always @(posedge clk or
        negedge rst_n) begin
        if (!rst_n) begin
            alu_result_mem <=
                32'b0;
            rs2_data_mem    <=
                32'b0;
            rd_addr_mem     <= 5'
                b0;
            mem_read_mem    <= 1'
                b0;
            mem_write_mem   <= 1'
                b0;
        end else begin
            alu_result_mem <=
                alu_result;
            rs2_data_mem    <=
                rs2_data_forwarded
                ;
            rd_addr_mem     <=
                rd_addr_ex;
            mem_read_mem    <=
                mem_read_ex;
            mem_write_mem   <=
                mem_write_ex;
        end
    end

endmodule
```

7.0.4 mem_stage

```
    module mem_stage #(
    parameter ADDR_WIDTH = 32
)(
    input  wire
                          clk,
    input  wire
                          rst_n,
    input  wire [ADDR_WIDTH-1:0]
         alu_result_mem,   // ALU
        计算结果
    input  wire [31:0]
                  rs2_data_mem,
            // 源操作数2
    input  wire [4:0]
                   rd_addr_mem,
             // 目标寄存器地址
    input  wire

       mem_read_mem,      // 内存
       读使能
    input  wire

       mem_write_mem,     // 内存
       写使能
    input  wire

       reg_write_ex,      // 寄存
       器写使能
    input  wire [3:0]
                    mem_byte_en,
            // 字节使能
    output reg  [31:0]
                  mem_data_wb,
            // 内存读数据
    output reg  [31:0]
                  alu_result_wb,
          // ALU结果
    output reg  [4:0]
                   rd_addr_wb,
            // 写回寄存器地址
    output reg

       mem_to_reg_wb,     // 选择
       内存数据写回
    output reg

       reg_write_mem,     // 寄存
       器写使能输出
    output wire

       stall_mem,         // 内存
       访问暂停
    output wire [ADDR_WIDTH-1:0]
        mem_addr,          // 内
       存地址
    output wire [31:0]
                  mem_wdata,
              // 写数据
    output wire
                         mem_wen,
                // 写使能
    output wire
                         mem_ren,
                // 读使能
    input  wire [31:0]
                  mem_rdata,
              // 读数据
```

```
    input  wire
        mem_ready          // 内存就绪
);

// Cache接口信号
    wire        cache_hit;
    wire        cache_ready;
    wire [31:0] cache_rdata;

// 实例化数据缓存
    data_cache #(
        .ADDR_WIDTH  (ADDR_WIDTH),
        .WAYS        (2),
        .CACHE_SIZE  (4096),
        .LINE_SIZE   (32)
    ) u_data_cache (
        .clk         (clk),
        .rst_n       (rst_n),
        // CPU接口
        .cpu_addr    (alu_result_mem),
        .cpu_wdata   (rs2_data_mem),
        .cpu_wen     (mem_write_mem),
        .cpu_ren     (mem_read_mem),
        .cpu_byte_en (mem_byte_en),
        .cpu_rdata   (cache_rdata),
        .cache_hit   (cache_hit),
        .cache_ready (cache_ready),
        // 内存接口
        .mem_addr    (mem_addr),
        .mem_wdata   (mem_wdata),
        .mem_wen     (mem_wen),
        .mem_ren     (mem_ren),
        .mem_rdata   (mem_rdata),
        .mem_ready   (mem_ready)
    );

// 暂停信号生成: 当访存操作未完成时暂停流水线
    assign stall_mem = (mem_read_mem || mem_write_mem) && !cache_ready;

// MEM/WB 流水线寄存器更新
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            mem_data_wb   <= 32'h0;
            alu_result_wb <= 32'h0;
            rd_addr_wb    <= 5'h0;
            mem_to_reg_wb <= 1'b0;
            reg_write_mem <= 1'b0;
        end else if (!stall_mem) begin
            mem_data_wb   <= cache_rdata;
            alu_result_wb <= alu_result_mem;
            rd_addr_wb    <= rd_addr_mem;
            mem_to_reg_wb <= mem_read_mem;
            reg_write_mem <= reg_write_ex;
        end
    end
endmodule
```

7.0.5 wb_stage

```
    module wb_stage (
    input  wire        clk,
    input  wire        rst_n,

    // MEM/WB流水线寄存器输入
    input  wire [31:0] mem_data_wb,
    input  wire [31:0] alu_result_wb,
    input  wire [ 4:0] rd_addr_wb,
    input  wire        mem_to_reg_wb,
    input  wire        reg_write_wb,

    // 写回阶段输出
    output wire [31:0] wb_data,      // 改为 wire
    output wire [ 4:0] wb_rd_addr,   // 改为 wire
    output wire        wb_reg_write  // 改为 wire
);
```

```
17
18      // 直接赋值
19      assign wb_data      =
            mem_to_reg_wb ?
            mem_data_wb :
            alu_result_wb;
20      assign wb_rd_addr   =
            rd_addr_wb;
21      assign wb_reg_write =
            reg_write_wb;
22
23      // Debug信息
24      `ifdef DEBUG
25      always @(posedge clk) begin
26          if (wb_reg_write &&
              wb_rd_addr != 5'h0)
              begin
27              $display("WB Stage:
                  Writing %h to x%d
                  ", wb_data,
                  wb_rd_addr);
28          end
29      end
30      `endif
31
32  endmodule
```

7.0.6 riscv_core_top

```
1       module riscv_core_top (
2       input  wire        clk_in,
3       input  wire        rst_n,
4       output wire [31:0] debug_pc,
5       output wire
            debug_valid
6   );
7
8   // 内部时钟管理
9       wire        core_clk;
10      wire        clk_locked;
11
12  // 流水线控制信号
13      wire        stall_if,
            stall_id, stall_ex,
            stall_mem;
14      wire        flush_if,
            flush_id, flush_ex,
            flush_mem;
15      wire [1:0]  forward_a,
            forward_b;
16
17  // 流水线数据通路信号
18      wire [31:0] pc_if, pc_id,
            pc_ex, pc_mem, pc_wb;
19      wire [31:0] inst_if, inst_id
            ;
20      wire [31:0] rs1_data_id,
            rs2_data_id;
21      wire [31:0] imm_id;
22      wire [31:0] alu_result_ex,
            alu_result_mem,
            alu_result_wb;
23      wire [31:0] mem_data_wb;
24      wire [4:0]  rs1_addr_id,
            rs2_addr_id;
25      wire [4:0]  rd_addr_id,
            rd_addr_ex, rd_addr_mem,
            rd_addr_wb;
26      wire [3:0]  alu_op_id,
            alu_op_ex;
27      wire [2:0]  branch_op_id,
            branch_op_ex;
28      wire        reg_write_id,
            reg_write_ex,
            reg_write_mem,
            reg_write_wb;
29      wire        mem_read_id,
            mem_read_ex, mem_read_mem
            ;
30      wire        mem_write_id,
            mem_write_ex,
            mem_write_mem;
31      wire        branch_id,
            branch_ex;
32      wire        jump_id, jump_ex
            ;
33      wire        branch_taken;
34      wire [31:0] branch_target;
35      wire        mem_to_reg_wb;
36      wire [31:0] wb_data;
37      wire [4:0]  wb_rd_addr;
38      wire        wb_reg_write;
39      wire        if_ready;
40      wire [31:0] rs2_data_ex,
            rs2_data_mem;
41
42  // 分频时钟(clk_divider)实例化
43      clk_divider u_clk_div (
44          .clk        (clk_in),
45          .rst_n      (rst_n),
46          .clk_en     (1'b1),
47          .div_ratio  (8'd4),
48          .clk_out    (core_clk),
49          .clk_locked (clk_locked)
50      );
51
52  // 指令存储器（inst_mem）实例化
53      inst_mem u_inst_mem (
54          .addr       (pc_if),
55          .inst       (inst_if)
56      );
57
58  // 取指模块（if_stage）实例化
59      if_stage u_if_stage (
```

```
        .clk            (core_clk),
        .rst_n          (rst_n),
        .stall_if       (stall_if),
        .flush_if       (flush_if),
        .branch_taken   (branch_taken),
        .branch_target  (branch_target),
        .pc_if          (pc_if),
        .inst_if        (inst_if),
        .if_ready       (if_ready)
    );

// 解码模块（id_stage）实例化
    id_stage u_id_stage (
        .clk            (core_clk),
        .rst_n          (rst_n),
        .pc_if          (pc_if),
        .inst_if        (inst_if),
        .wb_rd_addr     (wb_rd_addr),
        .wb_rd_data     (wb_rd_data),
        .wb_rd_wen      (wb_reg_write),
        .stall_id       (stall_id),
        .flush_id       (flush_id),
        .pc_ex          (pc_ex),
        .rs1_data_ex    (rs1_data_ex),
        .rs2_data_ex    (rs2_data_ex),
        .imm_ex         (imm_ex),
        .rd_addr_ex     (rd_addr_ex),
        .alu_op_ex      (alu_op_ex),
        .mem_read_ex    (mem_read_ex),
        .mem_write_ex   (mem_write_ex),
        .branch_ex      (branch_ex),
        .jump_ex        (jump_ex)
    );

// 执行模块（ex_stage）实例化
    ex_stage u_ex_stage (
        .clk              (core_clk),
        .rst_n            (rst_n),
        .pc_ex            (pc_ex),
        .rs1_data_ex      (rs1_data_ex),
        .rs2_data_ex      (rs2_data_ex),
        .imm_ex           (imm_ex),
        .rd_addr_ex       (rd_addr_ex),
        .alu_op_ex        (alu_op_ex),
        .branch_op_ex     (branch_op_ex),
        .mem_read_ex      (mem_read_ex),
        .mem_write_ex     (mem_write_ex),
        .branch_ex        (branch_ex),
        .jump_ex          (jump_ex),
        .forward_data_mem (alu_result_mem),
        .forward_data_wb  (wb_data),
        .forward_a        (forward_a),
        .forward_b        (forward_b),
        .alu_result_mem   (alu_result_mem),
        .rs2_data_mem     (rs2_data_mem),
        .rd_addr_mem      (rd_addr_mem),
        .mem_read_mem     (mem_read_mem),
        .mem_write_mem    (mem_write_mem),
        .branch_taken     (branch_taken),
        .branch_target    (branch_target)
    );

// 访存模块（mem_stage）实例化
    mem_stage #(
        .ADDR_WIDTH (32)
    ) u_mem_stage (
        .clk            (core_clk),
        .rst_n          (rst_n),
```

```
        .alu_result_mem (alu_result_mem),
        .rs2_data_mem   (rs2_data_mem),
        .rd_addr_mem    (rd_addr_mem),
        .mem_read_mem   (mem_read_mem),
        .mem_write_mem  (mem_write_mem),
        .reg_write_ex   (reg_write_ex),
        .mem_byte_en    (mem_byte_en),
        .mem_data_wb    (mem_data_wb),
        .alu_result_wb  (alu_result_wb),
        .rd_addr_wb     (rd_addr_wb),
        .mem_to_reg_wb  (mem_to_reg_wb),
        .reg_write_mem  (reg_write_mem),
        .stall_mem      (stall_mem),
        .mem_addr       (data_addr),
        .mem_wdata      (data_wdata),
        .mem_wen        (data_wen),
        .mem_ren        (data_ren),
        .mem_rdata      (data_rdata),
        .mem_ready      (data_ready)
    );

// 写回模块（wb_stage）实例化
    wb_stage u_wb_stage (
        .clk            (core_clk),
        .rst_n          (rst_n),
        .mem_data_wb    (mem_data_wb),
        .alu_result_wb  (alu_result_wb),
        .rd_addr_wb     (rd_addr_wb),
        .mem_to_reg_wb  (mem_to_reg_wb),
        .reg_write_wb   (reg_write_wb),
        .wb_data        (wb_data),
        .wb_rd_addr     (wb_rd_addr),
        .wb_reg_write   (wb_reg_write)
    );

// 冒险检测单元（hazard_detection）实例化
    hazard_detection u_hazard_detection (
        .rs1_addr_id    (rs1_addr_id),
        .rs2_addr_id    (rs2_addr_id),
        .rd_addr_ex     (rd_addr_ex),
        .rd_addr_mem    (rd_addr_mem),
        .mem_read_ex    (mem_read_ex),
        .branch_taken   (branch_taken),
        .jump           (jump_ex),
        .stall_if       (stall_if),
        .stall_id       (stall_id),
        .stall_ex       (stall_ex),
        .flush_if       (flush_if),
        .flush_id       (flush_id),
        .flush_ex       (flush_ex)
    );

// 前递单元（forwarding_unit）实例化
    forwarding_unit u_forwarding_unit (
        .rs1_addr_ex    (rs1_addr_id),
        .rs2_addr_ex    (rs2_addr_id),
        .rd_addr_mem    (rd_addr_mem),
        .reg_write_mem  (reg_write_mem),
        .rd_addr_wb     (rd_addr_wb),
        .reg_write_wb   (reg_write_wb),
        .forward_a      (forward_a),
        .forward_b      (
```

```
            forward_b)
190     );
191 // 调试信号连接
192     assign debug_pc     = pc_if;
193     assign debug_valid =
            clk_locked && !stall_if;
194 endmodule
```

参考文献