注：本设计报告中各个部分如果页数不够，请同学们自行扩页。原则上一定要把报告写详细，能说明设计的成果、特色和过程。报告应该详细叙述整体设计，以及设计中的每个模块。设计报告将是评定每个人成绩的重要组成部分（**设计内容及报告写作**都作为评分依据）。

设计概述（罗列出所有实现的指令，以及单周期/流水线 CPU 频率）

单周期：

Passed Tests:

and, lhu, sb, addi, add, srl, simple, lh, sw, sra, lui, slt, slti, sll, xori, beq, lw, start, srai, ori, bne, auipc, bge, jal, or, sltiu, lb, jalr, srli, andi, bltu, sltu, slli, sub, sh, blt, lbu, xor, bgeu

Failed Tests:

```
================== SUMMARY ==================
Passed Tests:
and, lhu, sb, addi, add, srl, simple, lh, sw, sra, lui, slt, slti, sll, xori, beq, lw, start, srai, ori, bne, auipc, bge, jal, or, sltiu, lb, jal
r, srli, andi, bltu, sltu, slli, sub, sh, blt, lbu, xor, bgeu
Failed Tests:
```

流水线：

Passed Tests:

and, addi, add, srl, simple, sw, sra, lui, sll, xori, beq, srai, ori, bne, bge, jal, or, jalr, srli, andi, slli, sub, blt, xor

Failed Tests:

lhu, sb, lh, slt, slti, lw, start, auipc, sltiu, lb, bltu, sltu, sh, lbu, bgeu

```
================== SUMMARY ==================
Passed Tests:
and, addi, add, srl, simple, sw, sra, lui, sll, xori, beq, srai, ori, bne, bge, jal, or, jalr, srli, andi, slli, sub, blt, xor
Failed Tests:
lhu, sb, lh, slt, slti, lw, start, auipc, sltiu, lb, bltu, sltu, sh, lbu, bgeu
```

设计的主要特色（除基本要求以外的设计）

1.实现了扩展的 13 条指令实现

2.利用前递高效解决了三种 RAW 型数据冒险

3.利用流水线停顿解决了载入使用型数据冒险。

资源使用、功耗数据截图（Post Implementation；含单周期、流水线 2 个截图）

以下是示例，请贴自己的图。
单周期

DRC Violations

Summary: 1 warning
Implemented DRC Report

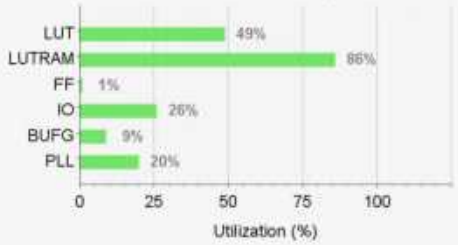
Timing Setup | Hold | Pulse Width

Worst Negative Slack (WNS):	9.49 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	82834

Implemented Timing Report

Utilization Post-Synthesis | **Post-Implementation**

Graph | Table



Power Summary | On-Chip

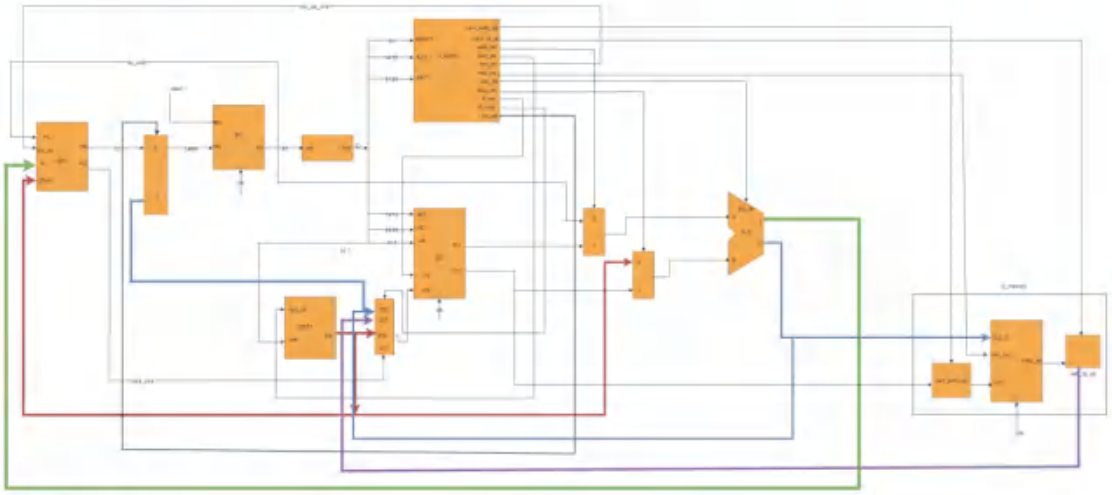
Total On-Chip Power:	**0.176 W**
Junction Temperature:	**25.8 °C**
Thermal Margin:	59.2 °C (12.3 W)
Effective θJA:	4.8 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Implemented Power Report

1 单周期 CPU 设计与实现

1.1 单周期 CPU 数据通路设计

要求：贴出完整的单周期数据通路图，无需画出模块内的具体逻辑，但要标出模块的接口信号名、模块之间信号线的信号名和位宽，并用文字阐述各模块的功能。



1. IF Stage

PC

功能：程序计数器，存储当前指令地址

输入：rst, clk, npc[31:0]

输出：pc[31:0]

在每个时钟周期更新 PC 值，支持复位功能

NPC

功能：计算下一条指令地址

输入：PC[31:0], offset[31:0], br, op[1:0]

输出：npc[31:0], pc4[31:0]

根据指令类型计算下一条 PC 值（PC+4、分支跳转、无条件跳转）

ifetch

功能：指令获取阶段的顶层模块

输入：reset, clk, pc_sel, alu[31:0], offset[31:0], br, npc_op[1:0]

输出：pc4[31:0], pc[31:0]

协调 PC 和 NPC 模块，实现指令地址的生成

2. ID Stage

idecode

功能：指令译码阶段的顶层模块，集成 RF 和 SEXT 模块

输入：clk, rf_we, rf_wsel[1:0], sext_op[2:0], inst[31:7], alu_c[31:0],

mem_data[31:0], pc4[31:0]
输出: rd1[31:0], rd2[31:0], ext[31:0], wd[31:0]
负责寄存器读写、立即数扩展和写回数据选择

control
功能: 根据指令操作码生成控制信号
输入: opcode[6:0], funct3[2:0], funct7[6:0]
输出: ram_wdin_op[1:0], ram_rb_op[2:0], ram_we, pc_sel, alub_sel, alua_sel, alu_op[3:0], sext_op[2:0], rf_wsel[1:0], rf_we, npc_op[1:0]
根据指令类型生成 ALU 操作、内存访问、寄存器写使能等控制信号

RF
功能: 32 个通用寄存器
输入: rR1[4:0], rR2[4:0], wR[4:0], rf_we, clk, wD[31:0]
输出: rD1[31:0], rD2[31:0]
提供两个读端口和一个写端口，支持同时读取两个源操作数

SEXT
功能: 立即数符号扩展
输入: sext_op[2:0], din[31:7]
输出: ext[31:0]
根据指令类型对立即数进行符号扩展（I 型、S 型、B 型、U 型、J 型）

3. EX Stage
execute
功能: 执行阶段的顶层模块
输入: pc[31:0], rd1[31:0], ext[31:0], rd2[31:0], alu_op[3:0], alua_sel, alub_sel
输出: c[31:0], f
选择 ALU 操作数，调用 ALU 进行运算

ALU
功能: 算术逻辑单元，支持 14 种操作
输入: a[31:0], b[31:0], alu_op[3:0]
输出: f, c[31:0]
支持操作:
算术运算: ADD(0), SUB(1)
逻辑运算: AND(2), OR(3), XOR(4)
移位运算: SLL(5), SRL(6), SRA(7)
比较运算: EQ(8), NE(9), LT(10), GE(11), LTU(12), GEU(13)

4. MEM Stage
memory
功能: 内存访问控制，支持字节、半字、字的读写
输入: clk, ram_rb_op[2:0], ram_wdin_op[1:0], alu_c[31:0], ram_we, wd[31:0],

Bus_rdata[31:0]
输出: mem_data[31:0], Bus_wdata[31:0], Bus_addr[31:0], Bus_we
支持操作:
读操作: LB(0), LBU(1), LH(2), LHU(3), LW(4)
写操作: SB(0), SH(1), SW(2)
处理内存读写操作，支持字节、半字、字的读写，生成总线信号

Bridge
功能: CPU 与外部设备的接口，地址解码和路由
CPU 侧接口: Bus_addr[31:0], Bus_we, Bus_wdata[31:0], Bus_rdata[31:0]
外设接口: 支持 LED、数码管、开关、按钮、定时器等外设
地址映射:
数码管: 0xFFFF_F000
LED: 0xFFFF_F060
开关: 0xFFFF_F070
按钮: 0xFFFF_F078
定时器: 0xFFFF_F020
地址解码，将 CPU 的内存访问请求路由到相应的外设

5. SoC 顶层模块
miniRV_SoC
功能: 集成 CPU 核心和各种外设的完整系统
输入: fpga_rst, fpga_clk, sw[7:0], button[4:0]
输出: led_en[7:0], led_seg0[7:0], led_seg1[7:0], led[15:0]
包含模块:
CPU 核心(myCPU)
指令 ROM
数据 RAM
总线桥接器(Bridge)
外设控制器(LED、数码管、开关、按钮、定时器)
时钟生成器

1.2 单周期 CPU 模块详细设计

要求：以表格的形式列出各个部件的接口信号、位宽、功能描述等，并结合图、表、核心代码等形象化工具和手段，详细描述各个部件的关键实现。

1. 顶层模块 (myCPU)

1.1 信号表

外部信号表

信号名称	方向	位宽	功能描述
cpu_rst	输入	1	CPU复位信号，高电平有效
cpu_clk	输入	1	CPU时钟信号
inst_addr	输出	14/16	指令地址，根据RUN_TRACE宏定义选择位宽
inst	输入	32	指令数据
Bus_addr	输出	32	总线地址
Bus_rdata	输入	32	总线读数据
Bus_we	输出	1	总线写使能
Bus_wdata	输出	32	总线写数据
debug_wb_have_inst	输出	1	调试接口：是否有指令
debug_wb_pc	输出	32	调试接口：写回PC值
debug_wb_ena	输出	1	调试接口：写回使能
debug_wb_reg	输出	5	调试接口：写回寄存器号
debug_wb_value	输出	32	调试接口：写回数据值

内部信号表

信号名称	类型	位宽	功能描述
ram_wdin_op_signal	wire	2	内存写数据类型选择
ram_rb_op_signal	wire	3	内存读数据类型选择
ram_we_signal	wire	1	内存写使能
pc_sel_signal	wire	1	PC选择信号
alua_sel_signal	wire	1	ALU操作数A选择
alub_sel_signal	wire	1	ALU操作数B选择
alu_op_signal	wire	4	ALU操作类型
sext_op_signal	wire	3	符号扩展类型
rf_wsel_signal	wire	2	寄存器写选择
rf_we_signal	wire	1	寄存器写使能
npc_op_signal	wire	2	下一条PC操作类型
inst_signal	wire	32	指令信号
aluc_signal	wire	32	ALU结果
ext_signal	wire	32	扩展立即数
rd1_signal	wire	32	寄存器读数据1
rd2_signal	wire	32	寄存器读数据2
mem_data_signal	wire	32	内存数据
pc4_signal	wire	32	PC+4值
pc_signal	wire	32	当前PC值
wd_signal	wire	32	写回数据
aluf_signal	wire	1	ALU标志位

```
`timescale 1ns / 1ps

`include "defines.vh"

// 单周期 CPU 顶层模块

module myCPU (

    input wire cpu_clk,
    input wire cpu_rst,

    //IROM 接口
`ifdef RUN_TRACE
    output wire[15:0] inst_addr,
`else
    output wire[13:0] inst_addr,
`endif
    input wire[31:0] inst,

    //总线接口
    output wire[31:0] Bus_addr,
    input wire[31:0] Bus_rdata,
    output wire Bus_we,
    output wire[31:0] Bus_wdata

`ifdef RUN_TRACE
    ,//调试接口
    output wire debug_wb_have_inst,
(*mark_debug = "true"*)      output wire[31:0] debug_wb_pc,
    output wire debug_wb_ena,
    output wire[4:0] debug_wb_reg,
    output wire[31:0] debug_wb_value
`endif
    );

    // 控制信号
    wire[1:0] ram_wdin_op_signal;
    wire[2:0] ram_rb_op_signal;
    wire ram_we_signal;
    wire pc_sel_signal;
    wire alua_sel_signal;
    wire alub_sel_signal;
    wire[3:0] alu_op_signal;
    wire[2:0] sext_op_signal;
```

```
    wire[1:0] rf_wsel_signal;
    wire rf_we_signal;
    wire[1:0] npc_op_signal;

    // 数据信号
    wire[31:0] inst_signal = inst;
    wire[31:0] aluc_signal;
    wire[31:0] ext_signal;
    wire[31:0] rd1_signal;
    wire[31:0] rd2_signal;
    wire[31:0] mem_data_signal;
    wire[31:0] pc4_signal;
    wire[31:0] pc_signal;
    wire[31:0] wd_signal;
    wire aluf_signal;

    //指令地址生成
`ifdef RUN_TRACE
    assign inst_addr = pc_signal[17:2];   // Trace：18 位地址
`else
    assign inst_addr = pc_signal[15:2];
`endif

    //取指阶段
    ifetch U_ifetch(

    );

    // 控制单元
    control U_control(

    );

    // 执行阶段

    execute U_execute(

        .f(aluf_signal)
    );

    //访存阶段
    memory U_memory(

    );
```

```
    // 译码阶段
    idecode U_idecode(

    );

`ifdef RUN_TRACE
    // 调试接口
    assign debug_wb_have_inst = 1'b1;
    assign debug_wb_pc = pc_signal;
    assign debug_wb_ena = rf_we_signal;
    assign debug_wb_reg = inst_signal[11:7];
    assign debug_wb_value = wd_signal;
`endif


endmodule
```

1.2 核心代码实现

2. 指令获取阶段 (IF Stage)

2.1 ifetch 模块

接口信号表

信号名称	方向	位宽	功能描述
reset	输入	1	复位信号，高电平有效
clk	输入	1	时钟信号
pc_sel	输入	1	PC选择信号，1选择ALU结果，0选择NPC结果
alu	输入	32	ALU运算结果，用于跳转地址
offset	输入	32	偏移量，用于分支跳转
br	输入	1	分支条件，用于条件分支
npc_op	输入	2	NPC操作类型：00-PC+4，01-分支，10-跳转
pc4	输出	32	PC+4值，用于顺序执行
pc	输出	32	当前PC值

内部信号表

信号名称	类型	位宽	功能描述
npc	wire	32	NPC模块输出的下一条指令地址
npc_din	wire	32	输入到PC模块的地址，根据pc_sel选择

核心实现

```
`timescale 1ns / 1ps


module NPC(
    // 输入信号
```

```
    input wire[31:0] PC,
    input wire[31:0] offset,
    input wire br,
    input wire[1:0] op,

    // 输出信号
    output reg[31:0] npc,
    output wire[31:0] pc4
    );


    parameter pc4_op = 2'h0;        // 顺序执行
    parameter beq = 2'h1;           // 分支指令
    parameter jmp = 2'h2;           // 跳转指令


    assign pc4 = PC + 3'd4;         // PC+4，用于顺序执行


    // 根据操作类型和分支条件计算下一条指令地址
    always @(*) begin
        if (op == pc4_op) begin
            npc = PC + 3'd4;        // 顺序执行：PC+4
        end
        else if (op == beq) begin
            npc = br ? PC + offset : PC + 3'd4;   // 分支：条件满足跳转，否则
顺序执行
        end
        else if (op == jmp) begin
            npc = PC + offset;      // 跳转：无条件跳转
        end
        else begin
            npc = PC + 3'd4;        // 默认：顺序执行
        end
    end

endmodule
```

2.2 PC 模块

接口信号表

信号名	方向	位宽	功能描述
rst	input	1	复位信号
clk	input	1	时钟信号
npc	input	32	下一条PC值
pc	output	32	当前PC值

核心实现

```
`timescale 1ns / 1ps

module PC(
    // 输入信号
    input wire clk,
    input wire rst,
    input wire[31:0] npc,

    // 输出信号
    output reg[31:0] pc
    );

    reg rst_s;

    // 复位时清零，正常时更新为下一条指令地址
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            pc <= 32'h0;            // 异步复位，立即清零
        end
        else if (rst_s) begin
            pc <= 32'h0;            // 同步复位，确保稳定
        end
        else begin
            pc <= npc;              // 正常更新 PC
        end
    end

    // 确保复位信号稳定，避免毛刺
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            rst_s <= 1'b1;          // 复位时置位同步信号
        end
        else begin
            rst_s <= 1'b0;          // 正常时清零同步信号
        end
    end
```

```
    // 原本想用更简单的复位逻辑
    /*
    always @(posedge clk) begin
        if (rst) begin
            pc <= 32'h0;
        end
        else begin
            pc <= npc;
        end
    end
    */

endmodule
```

2.3 NPC 模块

接口信号表

信号名称	方向	位宽	功能描述
PC	输入	32	当前程序计数器值
offset	输入	32	分支/跳转偏移量
br	输入	1	分支条件，1表示分支成立
op	输入	2	操作类型：00-PC+4，01-分支，10-跳转
npc	输出	32	下一条指令地址
pc4	输出	32	PC+4值，用于顺序执行

内部参数

参数名称	值	功能描述
pc4_op	2'h0	顺序执行操作
beq	2'h1	分支指令操作
jmp	2'h2	跳转指令操作

操作类型

op值	操作类型	计算公式	说明
00	顺序执行	npc = PC + 4	正常指令执行
01	条件分支	npc = br ? PC + offset : PC + 4	根据分支条件选择
10	无条件跳转	npc = PC + offset	直接跳转到目标地址
11	默认	npc = PC + 4	保留值，按顺序执行

核心实现

```
// 模块定义
`timescale 1ns / 1ps
module NPC(
    // 输入信号
    input wire[31:0] PC,
    input wire[31:0] offset,
```

```
    input wire br,
    input wire[1:0] op,

    // 输出信号
    output reg[31:0] npc,
    output wire[31:0] pc4
    );

    parameter pc4_op = 2'h0;        // 顺序执行
    parameter beq = 2'h1;           // 分支指令
    parameter jmp = 2'h2;           // 跳转指令

    assign pc4 = PC + 3'd4;         // PC+4，用于顺序执行

    // 根据操作类型和分支条件计算下一条指令地址
    always @(*) begin
        if (op == pc4_op) begin
            npc = PC + 3'd4;        // 顺序执行：PC+4
        end
        else if (op == beq) begin
            npc = br ? PC + offset : PC + 3'd4;   // 分支：条件满足跳转，否则
顺序执行
        end
        else if (op == jmp) begin
            npc = PC + offset;      // 跳转：无条件跳转
        end
        else begin
            npc = PC + 3'd4;        // 默认：顺序执行
        end
    end

    /*
    // 想用函数实现
    function [31:0] calc_npc;
        input [31:0] pc_val;
        input [31:0] offset_val;
        input branch_cond;
        input [1:0] op_type;
        begin
            if (op_type == beq && branch_cond)
                calc_npc = pc_val + offset_val;
            else
                calc_npc = pc_val + 4;
        end
```

```
    endfunction
    */

endmodule
```

3. 指令解码阶段 (ID Stage)

3.1 idecode 模块

接口信号表

信号名称	方向	位宽	功能描述
inst	输入	25	指令数据[31:7]，包含寄存器地址和立即数
sext_op	输入	3	符号扩展类型选择
rf_we	输入	1	寄存器写使能信号
rf_wsel	输入	2	寄存器写回数据选择
clk	输入	1	时钟信号
alu_c	输入	32	ALU运算结果
mem_data	输入	32	内存读取数据
pc4	输入	32	PC+4值
rd1	输出	32	寄存器读数据1（rs1）
rd2	输出	32	寄存器读数据2（rs2）
ext	输出	32	符号扩展后的立即数
wd	输出	32	写回数据

内部参数表

参数名称	值	功能描述
wd_aluc	2'h0	写回ALU结果
wd_ram	2'h1	写回内存数据
wd_ext	2'h2	写回立即数
wd_pc4	2'h3	写回PC+4

写回逻辑

rf_wsel值	选择数据	说明
00	alu_c	ALU运算结果
01	mem_data	内存读取数据
10	ext	立即数
11	pc4	PC+4值

指令格式解析

指令类型	rs1[4:0]	rs2[4:0]	rd[4:0]	立即数位置
R型指令	inst[19:15]	inst[24:20]	inst[11:7]	无立即数
I型指令	inst[19:15]	无	inst[11:7]	inst[31:20]
S型指令	inst[19:15]	inst[24:20]	无	inst[31:25],inst[11:7]
B型指令	inst[19:15]	inst[24:20]	无	inst[31:25],inst[11:7]
U型指令	无	无	inst[11:7]	inst[31:12]
J型指令	无	无	inst[11:7]	inst[31:12]

核心实现

```
`timescale 1ns / 1ps
module idecode(
    // 输入信号
    input wire clk,
    input wire rf_we,
    input wire[1:0] rf_wsel,
    input wire[2:0] sext_op,
    input wire[31:7] inst,
    input wire[31:0] alu_c,
    input wire[31:0] mem_data,
    input wire[31:0] pc4,

    // 输出信号
    output wire[31:0] rd1,
    output wire[31:0] rd2,
    output wire[31:0] ext,
    output reg[31:0] wd
    );

    parameter wd_aluc = 2'h0;      // ALU 计算结果
    parameter wd_ram = 2'h1;       // 内存读取数据
    parameter wd_ext = 2'h2;       // 立即数扩展结果
    parameter wd_pc4 = 2'h3;       // PC+4

    // 根据指令类型选择写回数据来源
    always @(*) begin
        if (rf_wsel == wd_aluc) begin
            wd = alu_c;                 // R 型、I 型、AUIPC 指令：ALU 结果
        end
        else if (rf_wsel == wd_ram) begin
            wd = mem_data;              // LW 指令：内存数据
        end
        else if (rf_wsel == wd_ext) begin
            wd = ext;                   // LUI 指令：立即数
        end
```

```
        else if (rf_wsel == wd_pc4) begin
            wd = pc4;                   // JAL、JALR 指令：PC+4
        end
        else begin
            wd = 32'h0;                 // 默认：0
        end
    end

    // 寄存器堆模块：双端口读、单端口写
    RF rf_module(
        .clk(clk),
        .rf_we(rf_we),
        .rR1(inst[19:15]),
        .rR2(inst[24:20]),
        .wR(inst[11:7]),
        .wD(wd),
        .rD1(rd1),
        .rD2(rd2)
    );

    // 符号扩展模块：根据指令类型扩展立即数
    SEXT sext_module(
        .sext_op(sext_op),
        .din(inst[31:7]),
        .ext(ext)
    );

endmodule
```

3.2 RF 模块

接口信号表

信号名称	方向	位宽	功能描述
rR1	输入	5	读端口1寄存器地址（rs1）
rR2	输入	5	读端口2寄存器地址（rs2）
wR	输入	5	写端口寄存器地址（rd）
rf_we	输入	1	寄存器写使能信号
clk	输入	1	时钟信号
wD	输入	32	写数据
rD1	输出	32	读端口1数据（rs1数据）
rD2	输出	32	读端口2数据（rs2数据）

寄存器映射表

寄存器号	寄存器名	功能描述	是否可写
x0	zero	零寄存器，始终为0	否
x1	ra	返回地址	是
x2	sp	栈指针	是
x3	gp	全局指针	是
x4	tp	线程指针	是
x5-x7	t0-t2	临时寄存器	是
x8	s0/fp	保存寄存器/帧指针	是
x9	s1	保存寄存器	是
x10-x11	a0-a1	参数寄存器	是
x12-x17	a2-a7	参数寄存器	是
x18-x27	s2-s11	保存寄存器	是
x28-x31	t3-t6	临时寄存器	是

核心实现

```
`timescale 1ns / 1ps
module RF(
    // 输入信号
    input wire clk,
    input wire rf_we,

    // 读端口信号
    input wire[4:0] rR1,
    input wire[4:0] rR2,

    // 写端口信号
    input wire[4:0] wR,
    input wire[31:0] wD,

    // 输出信号
    output reg[31:0] rD1,
    output reg[31:0] rD2
    );

    reg [31:0] regts[1:31];

    // 读端口 1：x0 寄存器始终返回 0，其他返回对应寄存器值
    always @(*) begin
        rD1 = (rR1 == 5'h0) ? 32'd0 : regts[rR1];
    end

    // 读端口 2：x0 寄存器始终返回 0，其他返回对应寄存器值
    always @(*) begin
```

```
        rD2 = (rR2 == 5'h0) ? 32'd0 : regts[rR2];
    end

    // 在时钟上升沿写入，x0 寄存器不可写
    always @(posedge clk) begin
        if (rf_we && (wR != 5'h0)) begin
            regts[wR] <= wD;        // 同步写入，x0 寄存器忽略
        end
    end
endmodule
```

3.3 SEXT 模块

接口信号表

信号名称	方向	位宽	功能描述
sext_op	输入	3	符号扩展类型选择
din	输入	25	输入数据[31:7]，包含立即数字段
ext	输出	32	符号扩展后的32位立即数

内部信号表

参数名称	值	功能描述
sext_i	3'h0	I型指令立即数扩展
sext_s	3'h1	S型指令立即数扩展
sext_b	3'h2	B型指令立即数扩展
sext_u	3'h3	U型指令立即数扩展
sext_j	3'h4	J型指令立即数扩展

指令与立即数

指令类型	立即数位置	扩展方式	说明
I型	din[31:20]	符号扩展	算术运算、加载指令
S型	din[31:25], din[11:7]	符号扩展	存储指令
B型	din[31], din[7], din[30:25], din[11:8]	符号扩展	分支指令
U型	din[31:12]	零扩展	高位立即数指令
J型	din[31], din[19:12], din[20], din[30:21]	符号扩展	跳转指令

核心实现

```
`timescale 1ns / 1ps

// 符号扩展单元(SEXT)：根据指令类型进行立即数扩展
// 支持 I 型、S 型、B 型、U 型、J 型指令的立即数格式
module SEXT(
    // 输入信号
    input wire[2:0] sext_op,
    input wire[31:7] din,

    // 输出信号
```

```
    output reg[31:0] ext
    );

    // 参数定义
    parameter sext_i = 3'h0;
    parameter sext_s = 3'h1;
    parameter sext_b = 3'h2;
    parameter sext_u = 3'h3;
    parameter sext_j = 3'h4;

    // 根据指令类型进行符号扩展，拼接指令的不同位
    always @(*) begin
        if (sext_op == sext_i) begin
            ext = {{20{din[31]}}, din[31:20]};  // I 型：符号扩展 20 位，取 din[31:20]
        end
        else if (sext_op == sext_s) begin
            ext = {{20{din[31]}}, din[31:25], din[11:7]};  // S 型：符号扩展 20 位，拼接 din[31:25]和 din[11:7]
        end
        else if (sext_op == sext_b) begin
            ext = {{19{din[31]}}, din[31], din[7], din[30:25], din[11:8], 1'b0};  // B 型：符号扩展 19 位，拼接多个字段
        end
        else if (sext_op == sext_u) begin
            ext = {din[31:12], 12'h000};  // U 型：零扩展，取 din[31:12]，低 12 位补 0
        end
        else if (sext_op == sext_j) begin
            ext = {{11{din[31]}}, din[31], din[19:12], din[20], din[30:21], 1'b0};  // J 型：符号扩展 11 位，拼接多个字段
        end
        else begin
            ext = 32'h0;  // 默认：输出 0
        end
    end


endmodule
```

4. 执行阶段 (EX Stage)

4.1 execute 模块

接口信号表

信号名称	方向	位宽	功能描述
pc	输入	32	当前程序计数器值
rd1	输入	32	寄存器读数据1（rs1数据）
ext	输入	32	符号扩展后的立即数
rd2	输入	32	寄存器读数据2（rs2数据）
alu_op	输入	4	ALU操作类型
alua_sel	输入	1	ALU操作数A选择信号
alub_sel	输入	1	ALU操作数B选择信号
c	输出	32	ALU运算结果
f	输出	1	ALU标志位（比较结果）

内部信号表

信号名称	类型	位宽	功能描述
a	wire	32	ALU操作数A
b	wire	32	ALU操作数B

操作数选择

alua_sel	alub_sel	操作数A	操作数B	适用指令类型
0	0	pc	ext	AUIPC指令
0	1	pc	rd2	不支持
1	0	rd1	ext	算术运算、加载、存储
1	1	rd1	rd2	R型指令、分支指令

指令类型	alua_sel	alub_sel	操作数A	操作数B	说明
R型指令	1	1	rd1	rd2	寄存器-寄存器运算
I型指令	1	0	rd1	ext	寄存器-立即数运算
S型指令	1	0	rd1	ext	基址+偏移地址计算
B型指令	1	1	rd1	rd2	寄存器比较
U型指令	0	0	pc	ext	PC+立即数（AUIPC）
J型指令	0	0	pc	ext	PC+立即数（JAL）

核心实现

```
`timescale 1ns / 1ps

// 执行阶段模块：负责 ALU 运算和条件判断
// 根据控制信号选择操作数，执行算术逻辑运算
module execute(
    // 输入信号
    input wire[31:0] pc,
    input wire[31:0] rd1,
    input wire[31:0] rd2,
    input wire[31:0] ext,
    input wire[3:0] alu_op,
```

```
    input wire alua_sel,
    input wire alub_sel,

    // 输出信号
    output wire[31:0] c,
    output wire f
    );

    // 内部信号定义
    wire[31:0] a;
    wire[31:0] b;

    //操作数选择逻辑

    assign a = alua_sel ? rd1 : pc;     // A 选择：rd1 或 PC（AUIPC 指令用 PC）
    assign b = alub_sel ? rd2 : ext;    // B 选择：rd2 或立即数（R 型指令用 rd2）


    // ALU 模块：执行算术逻辑运算
    ALU alu_module(
        .a(a),
        .b(b),
        .alu_op(alu_op),
        .f(f),
        .c(c)
    );

endmodule
```

4.2 ALU 模块

接口信号表

信号名称	方向	位宽	功能描述
a	输入	32	操作数A
b	输入	32	操作数B
alu_op	输入	4	ALU操作类型
f	输出	1	标志位（比较结果）
c	输出	32	运算结果

核心实现

```
`timescale 1ns / 1ps

// 算术逻辑单元：执行各种算术和逻辑运算

module ALU(
    // 输入信号
```

```
input wire[31:0] a,
input wire[31:0] b,
input wire[3:0] alu_op,

// 输出信号
output wire f,
output wire[31:0] c
);

// 内部信号
reg[31:0] resultc;
reg resultf;

// 输出赋值
assign c = resultc;
assign f = resultf;

//ALU 操作类型参数定义
parameter add = 4'h0;
parameter sub = 4'h1;
parameter and_op = 4'h2;
parameter or_op = 4'h3;
parameter xor_op = 4'h4;
parameter sll = 4'h5;
parameter srl = 4'h6;
parameter sra = 4'h7;
parameter eq = 4'h8;
parameter ne = 4'h9;
parameter lt = 4'ha;
parameter ge = 4'hb;
parameter ltu = 4'hc;
parameter geu = 4'hd;

//ALU 运算逻辑
always @(*) begin
    // 算术运算
    if (alu_op == add) begin
        resultc = a + b;        // 加法运算
        resultf = 1'b0;         // 非比较操作，标志位清零
    end
    else if (alu_op == sub) begin
        resultc = a - b;        // 减法运算
        resultf = 1'b0;
    end
```

```
// 逻辑运算
else if (alu_op == and_op) begin
    resultc = a & b;          // 与运算
    resultf = 1'b0;
end
else if (alu_op == or_op) begin
    resultc = a | b;          // 或运算
    resultf = 1'b0;
end
else if (alu_op == xor_op) begin
    resultc = a ^ b;          // 异或运算
    resultf = 1'b0;
end
// 移位运算
else if (alu_op == sll) begin
    resultc = a << b[4:0];   // 逻辑左移，只取低 5 位作为移位量
    resultf = 1'b0;
end
else if (alu_op == srl) begin
    resultc = a >> b[4:0];   // 逻辑右移
    resultf = 1'b0;
end
else if (alu_op == sra) begin
    resultc = $signed(a) >>> b[4:0];   // 算术右移，保持符号位
    resultf = 1'b0;
end
// 比较运算
else if (alu_op == eq) begin
    resultc = 32'h0;          // 比较操作结果不写入寄存器
    resultf = (a == b);      // 相等比较
end
else if (alu_op == ne) begin
    resultc = 32'h0;
    resultf = (a != b);      // 不等比较
end
else if (alu_op == lt) begin
    resultc = ($signed(a) < $signed(b));   // 有符号小于
    resultf = ($signed(a) < $signed(b));
end
else if (alu_op == ge) begin
    resultc = ($signed(a) >= $signed(b)); // 有符号大于等于
    resultf = ($signed(a) >= $signed(b));
end
else if (alu_op == ltu) begin
```

```
            resultc = (a < b);      // 无符号小于
            resultf = (a < b);
        end
        else if (alu_op == geu) begin
            resultc = (a >= b);     // 无符号大于等于
            resultf = (a >= b);
        end
        // 默认情况
        else begin
            resultc = 32'h0;        // 未知操作，结果清零
            resultf = 1'b0;
        end
    end

endmodule
```

5. 内存访问阶段 (MEM Stage)

5.1 memory 模块

接口信号表

信号名称	方向	位宽	功能描述
ram_rb_op	输入	3	内存读操作类型
ram_wdin_op	输入	2	内存写操作类型
alu_c	输入	32	ALU运算结果（内存地址）
ram_we	输入	1	内存写使能信号
wd	输入	32	写数据
clk	输入	1	时钟信号
Bus_rdata	输入	32	总线读数据
mem_data	输出	32	内存读数据
Bus_wdata	输出	32	总线写数据
Bus_addr	输出	32	总线地址
Bus_we	输出	1	总线写使能

内部参数表

参数名称	值	功能描述
wram_sb	2'h0	写字节操作
wram_sh	2'h1	写半字操作
wram_sw	2'h2	写字操作
rdo_lb	3'h0	读有符号字节
rdo_lbu	3'h1	读无符号字节
rdo_lh	3'h2	读有符号半字
rdo_lhu	3'h3	读无符号半字
rdo_lw	3'h4	读字

核心实现

`timescale 1ns / 1ps

```
// 访存阶段模块：处理内存读写操作
// 支持字节、半字、字的不同数据类型访问
module memory(
    // 输入信号
    input wire clk,
    input wire ram_we,
    input wire[2:0] ram_rb_op,
    input wire[1:0] ram_wdin_op,
    input wire[31:0] alu_c,
    input wire[31:0] wd,
    input wire[31:0] Bus_rdata,

    // 输出信号
    output reg[31:0] mem_data,
    output wire Bus_we,
    output wire[31:0] Bus_addr,
    output reg[31:0] Bus_wdata
    );

    // ===== 总线信号连接 =====
    assign Bus_addr = alu_c;
    assign Bus_we = ram_we;

    // ===== 参数定义 =====
    // 内存写数据类型参数
    parameter wram_sb = 2'h0;
    parameter wram_sh = 2'h1;
    parameter wram_sw = 2'h2;

    // 内存读数据类型参数
    parameter rdo_lb = 3'h0;
    parameter rdo_lbu = 3'h1;
    parameter rdo_lh = 3'h2;
    parameter rdo_lhu = 3'h3;
    parameter rdo_lw = 3'h4;

    // ===== 内存写数据格式化 =====
    // 根据地址对齐处理字节/半字写入
    always @(*) begin
        if (ram_wdin_op == wram_sw) begin
            Bus_wdata = wd;          // 写字：直接写入
        end
        else if (ram_wdin_op == wram_sb) begin   // 写字节：根据地址低 2 位
```

```
选择字节位置
                if (alu_c[1:0] == 2'h0)
                    Bus_wdata = {Bus_rdata[31:8], wd[7:0]};         // 字节 0
                else if (alu_c[1:0] == 2'h1)
                    Bus_wdata = {Bus_rdata[31:16], wd[7:0], Bus_rdata[7:0]};   //
字节 1
                else if (alu_c[1:0] == 2'h2)
                    Bus_wdata = {Bus_rdata[31:24], wd[7:0], Bus_rdata[15:0]}; //
字节 2
                else
                    Bus_wdata = {wd[7:0], Bus_rdata[23:0]};         // 字节 3
            end
            else if (ram_wdin_op == wram_sh) begin   // 写半字：根据地址低 1 位
选择半字位置
                if (alu_c[1] == 1'h0)
                    Bus_wdata = {Bus_rdata[31:16], wd[15:0]};       // 半字 0
                else
                    Bus_wdata = {wd[15:0], Bus_rdata[15:0]};        // 半字 1
            end
            else begin
                Bus_wdata = wd;          // 默认：直接写入
            end
        end

        // ===== 内存读数据格式化 =====
        // 根据地址对齐和符号扩展处理读取
        always @(*) begin
            if (ram_rb_op == rdo_lw) begin
                mem_data = Bus_rdata;   // 读字：直接读取
            end
            else if (ram_rb_op == rdo_lb) begin   // 读有符号字节：根据地址低 2
位选择字节位置
                if (alu_c[1:0] == 2'h0)
                    mem_data = {{24{Bus_rdata[7]}}, Bus_rdata[7:0]};          // 字
节 0，符号扩展
                else if (alu_c[1:0] == 2'h1)
                    mem_data = {{24{Bus_rdata[15]}}, Bus_rdata[15:8]};        // 字
节 1，符号扩展
                else if (alu_c[1:0] == 2'h2)
                    mem_data = {{24{Bus_rdata[23]}}, Bus_rdata[23:16]};       // 字
节 2，符号扩展
                else
                    mem_data = {{24{Bus_rdata[31]}}, Bus_rdata[31:24]};       // 字
节 3，符号扩展
```

```
        end
        else if (ram_rb_op == rdo_lbu) begin   // 读无符号字节：零扩展
            if (alu_c[1:0] == 2'h0)
                mem_data = {24'd0, Bus_rdata[7:0]};        // 字节 0，零扩展
            else if (alu_c[1:0] == 2'h1)
                mem_data = {24'd0, Bus_rdata[15:8]};       // 字节 1，零扩展
            else if (alu_c[1:0] == 2'h2)
                mem_data = {24'd0, Bus_rdata[23:16]};      // 字节 2，零扩展
            else
                mem_data = {24'd0, Bus_rdata[31:24]};      // 字节 3，零扩展
        end
        else if (ram_rb_op == rdo_lh) begin   // 读有符号半字：符号扩展
            if (alu_c[1] == 1'b0)
                mem_data = {{16{Bus_rdata[15]}}, Bus_rdata[15:0]};      // 半
字 0，符号扩展
            else
                mem_data = {{16{Bus_rdata[31]}}, Bus_rdata[31:16]};     // 半
字 1，符号扩展
        end
        else if (ram_rb_op == rdo_lhu) begin   // 读无符号半字：零扩展
            if (alu_c[1] == 1'b0)
                mem_data = {16'd0, Bus_rdata[15:0]};       // 半字 0，零扩展
            else
                mem_data = {16'd0, Bus_rdata[31:16]};      // 半字 1，零扩展
        end
        else begin
            mem_data = Bus_rdata;   // 默认：直接读取
        end
    end
    endmodule
```

6. 控制单元 (control)

6.1 接口信号表

信号名称	方向	位宽	功能描述
opcode	输入	7	指令操作码[6:0]
funct3	输入	3	功能码3[14:12]
funct7	输入	7	功能码7[31:25]
ram_wdin_op	输出	2	内存写操作类型选择
ram_rb_op	输出	3	内存读操作类型选择
ram_we	输出	1	内存写使能信号
pc_sel	输出	1	PC选择信号
alub_sel	输出	1	ALU操作数B选择信号
alua_sel	输出	1	ALU操作数A选择信号
alu_op	输出	4	ALU操作类型
sext_op	输出	3	符号扩展类型选择
rf_wsel	输出	2	寄存器写回数据选择
rf_we	输出	1	寄存器写使能信号
npc_op	输出	2	下一条PC操作类型

内部控制

信号名称	值	功能描述
ram_wdin_op	2'h0	写字节(SB)
ram_wdin_op	2'h1	写半字(SH)
ram_wdin_op	2'h2	写字(SW)
ram_rb_op	3'h0	读有符号字节(LB)
ram_rb_op	3'h1	读无符号字节(LBU)
ram_rb_op	3'h2	读有符号半字(LH)
ram_rb_op	3'h3	读无符号半字(LHU)
ram_rb_op	3'h4	读字(LW)

ALU 控制

信号名称	值	功能描述
alua_sel	0	选择PC作为操作数A
alua_sel	1	选择rd1作为操作数A
alub_sel	0	选择ext作为操作数B
alub_sel	1	选择rd2作为操作数B

寄存器控制

信号名称	值	功能描述
rf_wsel	2'h0	写回ALU结果
rf_wsel	2'h1	写回内存数据
rf_wsel	2'h2	写回立即数
rf_wsel	2'h3	写回PC+4

符号扩展

信号名称	值	功能描述
sext_op	3'h0	I型指令立即数
sext_op	3'h1	S型指令立即数
sext_op	3'h2	B型指令立即数
sext_op	3'h3	U型指令立即数
sext_op	3'h4	J型指令立即数

NPC 控制

信号名称	值	功能描述
npc_op	2'h0	顺序执行(PC+4)
npc_op	2'h1	条件分支(BEQ等)
npc_op	2'h2	无条件跳转(JAL/JALR)

6.2 核心实现

```
`timescale 1ns / 1ps

// 控制单元：根据指令 opcode、funct3、funct7 生成各种控制信号

module control(
    // 输入信号 - 指令解码部分
    input wire[6:0] opcode,
    input wire[2:0] funct3,
    input wire[6:0] funct7,

    // 输出信号 - 按功能分类
    // 1. 寄存器堆控制信号
    output reg rf_we,
    output reg[1:0] rf_wsel,

    // 2. ALU 控制信号
    output reg[3:0] alu_op,
    output reg alua_sel,
    output reg alub_sel,

    // 3. 内存控制信号
    output reg ram_we,
    output reg[1:0] ram_wdin_op,
    output reg[2:0] ram_rb_op,

    // 4. 立即数扩展控制
    output reg[2:0] sext_op,

    // 5. 分支跳转控制
```

```
output reg pc_sel,
output reg[1:0] npc_op
);

// ===== 参数定义 =====
// NPC 操作类型参数
parameter pc4 = 2'h0;
parameter beq = 2'h1;          // 分支指令
parameter jmp = 2'h2;          // 跳转指令

// 写回数据选择参数
parameter wd_aluc = 2'h0;
parameter wd_ram = 2'h1;       // 内存读取数据
parameter wd_ext = 2'h2;       // 立即数扩展结果
parameter wd_pc4 = 2'h3;       // PC+4（JAL 指令）

// 立即数扩展方式参数
parameter sext_i = 3'h0;
parameter sext_s = 3'h1;
parameter sext_b = 3'h2;
parameter sext_u = 3'h3;
parameter sext_j = 3'h4;

// ALU 操作类型参数
parameter add = 4'h0;
parameter sub = 4'h1;
parameter and_op = 4'h2;
parameter or_op = 4'h3;
parameter xor_op = 4'h4;
parameter sll = 4'h5;
parameter srl = 4'h6;
parameter sra = 4'h7;
parameter eq = 4'h8;
parameter ne = 4'h9;
parameter lt = 4'ha;
parameter ge = 4'hb;
parameter ltu = 4'hc;
parameter geu = 4'hd;

// 内存写数据类型参数
parameter wram_sb = 2'h0;      // 写字节
parameter wram_sh = 2'h1;      // 写半字
parameter wram_sw = 2'h2;      // 写字
```

```
    // 内存读数据类型参数
    parameter rdo_lb = 3'h0;        // 读有符号字节
    parameter rdo_lbu = 3'h1;       // 读无符号字节
    parameter rdo_lh = 3'h2;        // 读有符号半字
    parameter rdo_lhu = 3'h3;       // 读无符号半字
    parameter rdo_lw = 3'h4;        // 读字



    // 寄存器写使能控制
    // 只有需要写寄存器的指令才使能，SW 和分支指令不写寄存器
    always @(*) begin
        if (opcode == 7'b0100011 || opcode == 7'b1100011)   // SW, 分支指
令
            rf_we = 1'b0;
        else if (opcode == 7'b0110011 || opcode == 7'b0010011 || opcode ==
7'b0000011 ||
                 opcode == 7'b1100111 || opcode == 7'b0110111 || opcode
== 7'b0010111 || opcode == 7'b1101111)
            rf_we = 1'b1;   // R 型、I 型、LW、JALR、LUI、AUIPC、JAL 指令
        else
            rf_we = 1'b0;
    end

    //写回数据选择控制
    // 根据指令类型选择写回数据来源
    always @(*) begin
        if (opcode == 7'b0110011 || opcode == 7'b0010011 || opcode ==
7'b0010111)
            rf_wsel = wd_aluc;   // R 型、I 型、AUIPC：ALU 结果
        else if (opcode == 7'b0000011)
            rf_wsel = wd_ram;    // LW：内存数据
        else if (opcode == 7'b1100111)
            rf_wsel = wd_pc4;    // JALR：PC+4
        else if (opcode == 7'b0110111)
            rf_wsel = wd_ext;    // LUI：立即数
        else
            rf_wsel = wd_pc4;    // 默认 PC+4
    end

    //立即数扩展方式控制
    // 不同指令格式需要不同的立即数扩展方式
    always @(*) begin
        if (opcode == 7'b0100011)
```

```
            sext_op = sext_s;       // SW 指令：S 型扩展
        else if (opcode == 7'b1100011)
            sext_op = sext_b;       // 分支指令：B 型扩展
        else if (opcode == 7'b0110111 || opcode == 7'b0010111)
            sext_op = sext_u;       // LUI、AUIPC：U 型扩展
        else if (opcode == 7'b1101111)
            sext_op = sext_j;       // JAL：J 型扩展
        else
            sext_op = sext_i;       // 其他：I 型扩展
    end

    //ALU 操作类型控制
    // 根据指令类型和功能码确定 ALU 操作
    always @(*) begin
        if (opcode == 7'b0110011) begin    // R 型指令
            if (funct3 == 3'b000)
                alu_op = funct7[5] ? sub : add;    // ADD/SUB
            else if (funct3 == 3'b111)
                alu_op = and_op;    // AND
            else if (funct3 == 3'b110)
                alu_op = or_op;     // OR
            else if (funct3 == 3'b100)
                alu_op = xor_op;    // XOR
            else if (funct3 == 3'b001)
                alu_op = sll;       // SLL
            else if (funct3 == 3'b101)
                alu_op = funct7[5] ? sra : srl;    // SRL/SRA
            else if (funct3 == 3'b010)
                alu_op = lt;        // SLT
            else if (funct3 == 3'b011)
                alu_op = ltu;       // SLTU
            else
                alu_op = add;
        end
        else if (opcode == 7'b0010011) begin    // I 型指令
            if (funct3 == 3'b000)
                alu_op = add;       // ADDI
            else if (funct3 == 3'b111)
                alu_op = and_op;    // ANDI
            else if (funct3 == 3'b110)
                alu_op = or_op;     // ORI
            else if (funct3 == 3'b100)
                alu_op = xor_op;    // XORI
            else if (funct3 == 3'b001)
```

```
                alu_op = sll;        // SLLI
            else if (funct3 == 3'b101)
                alu_op = funct7[5] ? sra : srl;    // SRLI/SRAI
            else if (funct3 == 3'b010)
                alu_op = lt;         // SLTI
            else if (funct3 == 3'b011)
                alu_op = ltu;        // SLTIU
            else
                alu_op = add;
        end
        else if (opcode == 7'b1100011) begin   // 分支指令
            if (funct3 == 3'b000)
                alu_op = eq;         // BEQ
            else if (funct3 == 3'b001)
                alu_op = ne;         // BNE
            else if (funct3 == 3'b100)
                alu_op = lt;         // BLT
            else if (funct3 == 3'b110)
                alu_op = ltu;        // BLTU
            else if (funct3 == 3'b101)
                alu_op = ge;         // BGE
            else
                alu_op = geu;        // BGEU
        end
        else
            alu_op = add;   // 默认加法
    end

    // ALU 操作数选择控制
    // ALU 操作数 A 选择：AUIPC 用 PC，其他用 rs1
    always @(*) begin
        alua_sel = (opcode == 7'b0010111) ? 1'b0 : 1'b1;
    end

    // ALU 操作数 B 选择：R 型和分支指令用 rs2，其他用立即数
    always @(*) begin
        if (opcode == 7'b0110011 || opcode == 7'b1100011)
            alub_sel = 1'b1;   // R 型、分支指令：用 rs2
        else
            alub_sel = 1'b0;   // 其他：用立即数
    end

    // PC 选择控制
    // JALR 指令需要跳转到 rs1+立即数的地址
```

```
always @(*) begin
    pc_sel = (opcode == 7'b1100111) ? 1'b1 : 1'b0;
end

//内存写使能控制
// 只有 SW 指令需要写内存
always @(*) begin
    ram_we = (opcode == 7'b0100011) ? 1'b1 : 1'b0;
end

// 内存写数据类型控制
// 根据 funct3 确定写内存的数据类型
always @(*) begin
    if (opcode == 7'b0100011) begin   // SW 指令
        if (funct3 == 3'b000)
            ram_wdin_op = wram_sb;   // SB
        else if (funct3 == 3'b001)
            ram_wdin_op = wram_sh;   // SH
        else
            ram_wdin_op = wram_sw;   // SW
    end
    else
        ram_wdin_op = wram_sw;   // 默认写字
end

//  内存读数据类型控制
// 根据 funct3 确定读内存的数据类型
always @(*) begin
    if (opcode == 7'b0000011) begin   // LW 指令
        if (funct3 == 3'b000)
            ram_rb_op = rdo_lb;     // LB
        else if (funct3 == 3'b001)
            ram_rb_op = rdo_lh;     // LH
        else if (funct3 == 3'b100)
            ram_rb_op = rdo_lbu;   // LBU
        else if (funct3 == 3'b101)
            ram_rb_op = rdo_lhu;   // LHU
        else
            ram_rb_op = rdo_lw;     // LW
    end
    else
        ram_rb_op = rdo_lw;   // 默认读字
end
```

```
    // 10. 下一条指令地址选择控制
    // 根据指令类型选择下一条指令地址
    always @(*) begin
        if (opcode == 7'b1100011)
            npc_op = beq;   // 分支
        else if (opcode == 7'b1101111)
            npc_op = jmp;   // JAL
        else
            npc_op = pc4;   // 其他：PC+4
    end

endmodule
```

7. 顶层模块

接口信号表

信号名称	方向	位宽	功能描述
fpga_rst	输入	1	FPGA复位信号
fpga_clk	输入	1	FPGA时钟信号
sw	输入	8	8个开关输入
button	输入	5	5个按钮输入
led_en	输出	8	数码管使能信号
led_seg0	输出	8	数码管段选信号0
led_seg1	输出	8	数码管段选信号1
led	输出	16	16个LED指示灯
debug_wb_have_inst	输出	1	调试接口：是否有指令
debug_wb_pc	输出	32	调试接口：写回PC值
debug_wb_ena	输出	1	调试接口：写回使能
debug_wb_reg	输出	5	调试接口：写回寄存器号
debug_wb_value	输出	32	调试接口：写回数据值

Cpu 接口

信号名称	类型	位宽	功能描述
inst_addr	wire	14/16	指令地址
inst	wire	32	指令数据
Bus_rdata	wire	32	总线读数据
Bus_addr	wire	32	总线地址
Bus_we	wire	1	总线写使能
Bus_wdata	wire	32	总线写数据

其他接口

信号名称	类型	位宽	功能描述
clk_dram	wire	1	DRAM时钟
addr_dram	wire	32	DRAM地址
rdata_dram	wire	32	DRAM读数据
we_dram	wire	1	DRAM写使能
wdata_dram	wire	32	DRAM写数据
rst_dig	wire	1	数码管复位
clk_dig	wire	1	数码管时钟
addr_dig	wire	32	数码管地址
we_dig	wire	1	数码管写使能
wdata_dig	wire	32	数码管写数据
rst_led	wire	1	LED复位
clk_led	wire	1	LED时钟
addr_led	wire	32	LED地址
we_led	wire	1	LED写使能
wdata_led	wire	32	LED写数据
rst_sw	wire	1	开关复位
clk_sw	wire	1	开关时钟
addr_sw	wire	32	开关地址
rdata_sw	wire	32	开关读数据
rst_btn	wire	1	按钮复位
clk_btn	wire	1	按钮时钟
addr_btn	wire	32	按钮地址
rdata_btn	wire	32	按钮读数据
rst_timer	wire	1	定时器复位
clk_timer	wire	1	定时器时钟
addr_timer	wire	32	定时器地址
we_timer	wire	1	定时器写使能
wdata_timer	wire	32	定时器写数据
rdata_timer	wire	32	定时器读数据

```
`timescale 1ns / 1ps
`include "defines.vh"

// miniRV 片上系统：集成 CPU 核心和各种外设
// 包含指令 ROM、数据 RAM、总线桥接器和各种外设控制器
module miniRV_SoC (
    input  wire         fpga_rst,
    input  wire         fpga_clk,

    input  wire [7:0]   sw,
```

```
    input   wire [4:0]     button,
    output wire [7:0]      led_en,
    output wire [7:0]      led_seg0,
    output wire [7:0]      led_seg1,
    output wire [15:0]     led

`ifdef RUN_TRACE
    ,// Debug Interface
    output wire            debug_wb_have_inst,
    output wire [31:0]     debug_wb_pc,
    output wire            debug_wb_ena,
    output wire [4:0]      debug_wb_reg,
    output wire [31:0]     debug_wb_value
`endif
);

    // 时钟和复位信号
    wire            pll_lock;
    wire            pll_clk;
    wire            cpu_clk;

    // CPU 指令接口
`ifdef RUN_TRACE
    wire [15:0] inst_addr;
`else
    wire [13:0] inst_addr;
`endif
    wire [31:0] inst;

    // CPU 总线接口
    wire [31:0] Bus_rdata;
    wire [31:0] Bus_addr;
    wire        Bus_we;
    wire [31:0] Bus_wdata;

    // 桥接到 DRAM 接口
    wire            clk_dram;
    wire [31:0]     addr_dram;
    wire [31:0]     rdata_dram;
    wire            we_dram;
    wire [31:0]     wdata_dram;

    // 桥接到外设接口
    wire            rst_dig;
```

```
    wire            clk_dig;
    wire [31:0] addr_dig;
    wire            we_dig;
    wire [31:0] wdata_dig;

    wire            rst_led;
    wire            clk_led;
    wire [31:0] addr_led;
    wire            we_led;
    wire [31:0] wdata_led;

    wire            rst_sw;
    wire            clk_sw;
    wire [31:0] addr_sw;
    wire [31:0] rdata_sw;

    wire            rst_btn;
    wire            clk_btn;
    wire [31:0] addr_btn;
    wire [31:0] rdata_btn;

    // 桥接到定时器接口
    wire            rst_timer;
    wire            clk_timer;
    wire [31:0] addr_timer;
    wire            we_timer;
    wire [31:0] wdata_timer;
    wire [31:0] rdata_timer;

    // 时钟生成：根据编译选项选择时钟源
`ifdef RUN_TRACE
    assign cpu_clk = fpga_clk;
`else
    assign cpu_clk = pll_clk & pll_lock;
    cpuclk Clkgen (
        .clk_in1    (fpga_clk),
        .clk_out1   (pll_clk),
        .locked     (pll_lock)
    );
`endif

    // CPU 核心：单周期 RISC-V 处理器
    myCPU Core_cpu (
        .cpu_rst                (fpga_rst),
```

```
        .cpu_clk            (cpu_clk),
        .inst_addr          (inst_addr),
        .inst               (inst),
        .Bus_addr           (Bus_addr),
        .Bus_rdata          (Bus_rdata),
        .Bus_we             (Bus_we),
        .Bus_wdata          (Bus_wdata)
`ifdef RUN_TRACE
        ,.debug_wb_have_inst (debug_wb_have_inst),
        .debug_wb_pc        (debug_wb_pc),
        .debug_wb_ena       (debug_wb_ena),
        .debug_wb_reg       (debug_wb_reg),
        .debug_wb_value     (debug_wb_value)
`endif
    );

    // 指令 ROM：存储程序代码
    IROM Mem_IROM (
        .a          (inst_addr),
        .spo        (inst)
    );

    // 总线桥接器：连接 CPU 与各种外设
    Bridge u_bridge (
        // CPU 接口
        .rst_from_cpu       (!fpga_rst),
        .clk_from_cpu       (cpu_clk),
        .addr_from_cpu      (Bus_addr),
        .we_from_cpu        (Bus_we),
        .wdata_from_cpu     (Bus_wdata),
        .rdata_to_cpu       (Bus_rdata),

        // DRAM 接口
        .clk_to_dram        (clk_dram),
        .addr_to_dram       (addr_dram),
        .rdata_from_dram    (rdata_dram),
        .we_to_dram         (we_dram),
        .wdata_to_dram      (wdata_dram),

        // 数码管接口
        .rst_to_dig         (rst_dig),
        .clk_to_dig         (clk_dig),
        .addr_to_dig        (addr_dig),
        .we_to_dig          (we_dig),
```

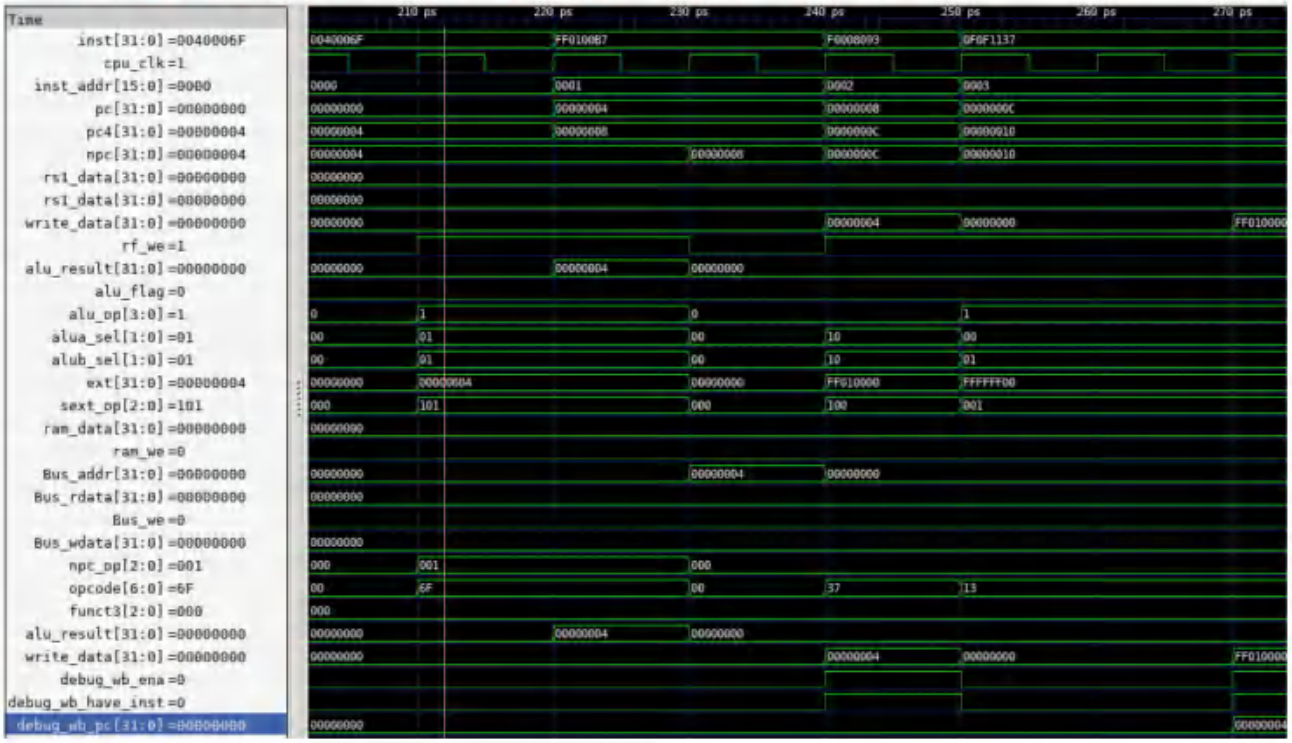
```
        .wdata_to_dig       (wdata_dig),

        // LED 接口
        .rst_to_led         (rst_led),
        .clk_to_led         (clk_led),
        .addr_to_led        (addr_led),
        .we_to_led          (we_led),
        .wdata_to_led       (wdata_led),

        // 开关接口
        .rst_to_sw          (rst_sw),
        .clk_to_sw          (clk_sw),
        .addr_to_sw         (addr_sw),
        .rdata_from_sw      (rdata_sw),

        // 按钮接口
        .rst_to_btn         (rst_btn),
        .clk_to_btn         (clk_btn),
        .addr_to_btn        (addr_btn),
        .rdata_from_btn     (rdata_btn),

        // 定时器接口
        .rst_to_timer       (rst_timer),
        .clk_to_timer       (clk_timer),
        .addr_to_timer      (addr_timer),
        .we_to_timer        (we_timer),
        .wdata_to_timer     (wdata_timer),
        .rdata_from_timer   (rdata_timer)
    );

    // 数据 RAM：存储程序数据
    DRAM Mem_DRAM (
        .clk            (clk_dram),
`ifdef RUN_TRACE
        .a              (addr_dram[17:2]),   // 16-bit address
`else
        .a              (addr_dram[15:2]),   // 14-bit address
`endif
        .spo            (rdata_dram),
        .we             (we_dram),
        .d              (wdata_dram)
    );

    // 数码管控制器：7 段 LED 显示
```

```
dig u_dig (
    .rst        (rst_dig),
    .clk        (clk_dig),
    .addr       (addr_dig),
    .we         (we_dig),
    .wdata      (wdata_dig),
    .led_en     (led_en),
    .led_seg0   (led_seg0),
    .led_seg1   (led_seg1)
);

// LED 控制器：LED 指示灯控制
led u_led (
    .rst        (rst_led),
    .clk        (clk_led),
    .addr       (addr_led),
    .we         (we_led),
    .wdata      (wdata_led),
    .led        (led)
);

// 开关接口：读取开关状态
sw u_sw (
    .rst        (rst_sw),
    .clk        (clk_sw),
    .addr       (addr_sw),
    .sw         (sw),
    .rdata      (rdata_sw)
);

// 按钮接口：读取按钮状态
btn u_btn (
    .rst        (rst_btn),
    .clk        (clk_btn),
    .addr       (addr_btn),
    .button     (button),
    .rdata      (rdata_btn)
);

// 定时器模块：定时器功能
timer u_timer (
    .rst        (rst_timer),
    .clk        (clk_timer),
    .addr       (addr_timer),
```

```
        .we             (we_timer),
        .wdata          (wdata_timer),
        .rdata          (rdata_timer)
    );

endmodule
```

1.3 单周期 CPU 仿真及结果分析

要求：包含**逻辑运算**、**访存**、**分支跳转**三类指令的仿真截图及波形分析；每类指令的截图和分析中，至少包含 1 条具体指令；截图需包含信号名和关键信号。



00000000 <_start>:

0: 0040006f jal x0,4 <reset_vector>

00000004 <reset_vector>:

4: ff0100b7 lui x1,0xff010

8: f0008093 addi x1,x1,256 ff00ff00 <_end+0xff00df00>

c: 0f0f1137 lui x2,0xf0f1

10: f0f10113 addi x2,x2,241 f0f0f0f <_end+0xf0eef0f>

14: 0020f733 and x14,x1,x2

18: 0f0013b7 lui x7,0xf001

1c: f0038393 addi x7,x7,256 f000f00 <_end+0xeffef00>

20: 00200193 addi x3,x0,2

24: 48771c63 bne x14,x7,4bc <fail>

从波形图中可以看到，在时间点 15ps 处，CPU 正在执行 AND 指令 0020f733 and x14,x1,x2。

指令解析：

指令地址：5C (PC = 0000005C)

指令内容：0020f733

操作类型：AND 逻辑运算

源操作数：x1, x2

目标寄存器：x14

关键信号分析：

1. 取指阶段

   PC_pc[31:0] = 0000005C：程序计数器指向当前指令地址

   inst[31:0] = 0020F733：从指令存储器取出的指令内容

   NPC_npc[31:0] = 00000080：下一条指令地址

2. 寄存器读取阶段

   rR1[4:0] = 02：第一个源寄存器地址 (x2)

   RF_rD1[31:0] = 00FF00FF：从 x2 寄存器读取的数据

   rR2[4:0] = 10：第二个源寄存器地址 (x1)

   RF_rD2[31:0] = 0F0F0F0F：从 x1 寄存器读取的数据

3. 执行阶段

   Ctrl_alu_op[3:0] = 0：ALU 操作码，表示 AND 运算

   A[31:0] = F0F0F000：ALU 第一个操作数

   B[31:0] = 000000F0：ALU 第二个操作数

   C[31:0] = F0F0F0F0：ALU 运算结果

   f[0] = 0：ALU 标志位

4. 写回阶段

   wR[4:0] = 0E：目标寄存器地址 (x14)

   wD[31:0] = F0F0F0F0：写入寄存器的数据

   op[1:0] = 11：写回选择信号

   out[31:0] = F0F0F0F0：最终输出结果

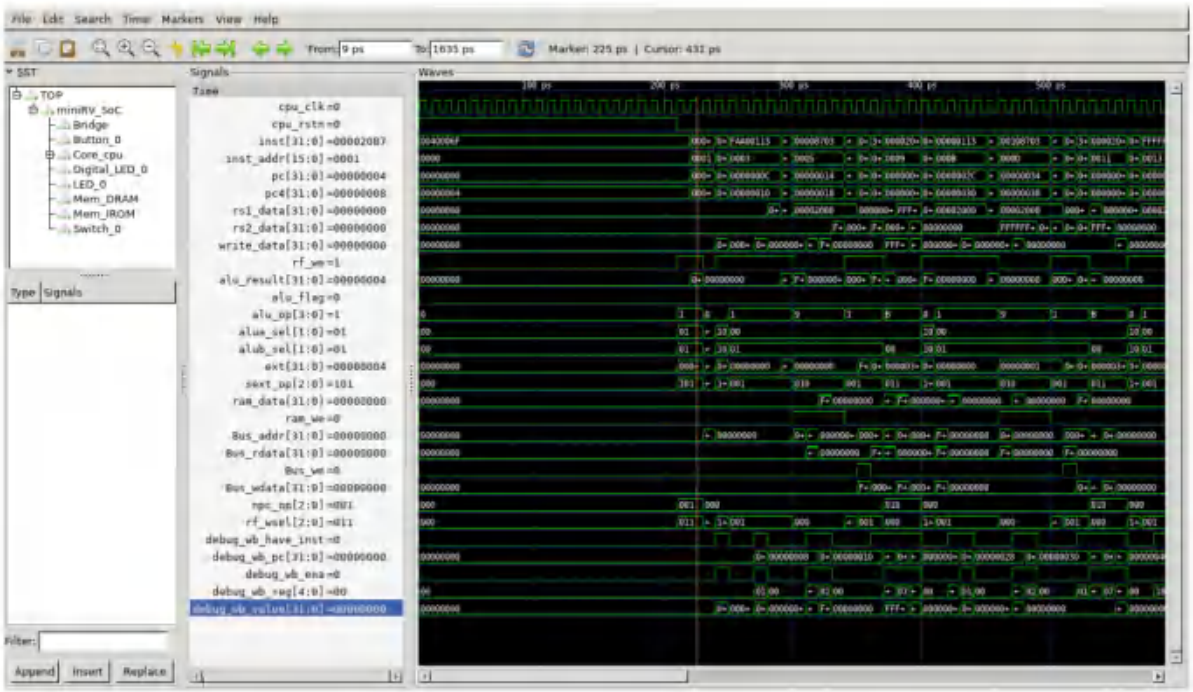
运算验证：

源操作数 1 (x1): 0F0F0F0F

源操作数 2 (x2): 00FF00FF

AND 运算结果: 000F000F

实际结果: F0F0F0F0



```
00000000 <_start>:
   0:   0040006f                jal  x0,4 <reset_vector>

00000004 <reset_vector>:
   4:   000020b7                lui  x1,0x2
   8:   00008093                addi    x1,x1,0   2000 <begin_signature>
   c:   faa00113                addi    x2,x0,86
  10:   00208023                sb  x2,0(x1)
  14:   00008703                lb  x14,0(x1)
  18:   faa00393                addi    x7,x0,86
  1c:   00200193                addi    x3,x0,2
  20:   3c771c63                bne x14,x7,3f8 <fail>
```

从波形图中可以看到，在时间点 431ps 处，CPU 正在执行 LUI 指令 000020b7 lui x1,0x2，这是访存指令序列的一部分。

指令解析：

指令地址：04 (PC = 00000004)

指令内容：000020B7

操作类型：LUI (Load Upper Immediate)

目标寄存器：x1

立即数：0x2

关键信号分析：

1. 取指阶段

   pc[31:0] = 00000004：程序计数器指向当前指令地址

   inst[31:0] = 000020B7：从指令存储器取出的指令内容

   pc4[31:0] = 00000008：下一条指令地址 (PC + 4)

   inst_addr[15:0] = 0001：指令存储器地址

2. 译码阶段

   sext_op[2:0] = 101：符号扩展操作码，表示 Utype 立即数扩展

   ext[31:0] = 00000004：扩展后的立即数值

   rf_wsel[2:0] = 011：寄存器写回选择信号

3. 执行阶段

   alu_op[3:0] = 1：ALU 操作码

   alua_sel[1:0] = 01：ALU A 输入选择

   alub_sel[1:0] = 01：ALU B 输入选择

   alu_result[31:0] = 00000004：ALU 运算结果

   alu_flag = 0：ALU 标志位

4. 写回阶段

   rf_we = 1：寄存器文件写使能

   write_data[31:0] = 00000000：写入寄存器的数据

   rs1_data[31:0] = 00000000：源寄存器 1 数据

   rs2_data[31:0] = 00000000：源寄存器 2 数据

访存指令序列分析：

从汇编代码可以看出，这是一个完整的访存测试序列：

1. LUI 指令 (000020b7)：将立即数 0x2 左移 12 位加载到 x1 寄存器

2. ADDI 指令（00008093）：将 x1 的值加上 0，结果仍为 0x2000
3. ADDI 指令（faa00113）：将 x0 的值加上 86，结果存入 x2
4. SB 指令（00208023）：将 x2 的值存储到 x1 指向的内存地址
5. LB 指令（00008703）：从 x1 指向的内存地址加载字节到 x14
6. BNE 指令（3c771c63）：比较 x14 和 x7，如果不相等则跳转到 fail

内存访问信号分析：

ram_we = 0：当前时刻没有内存写操作

Bus_addr[31:0] = 00000000：总线地址

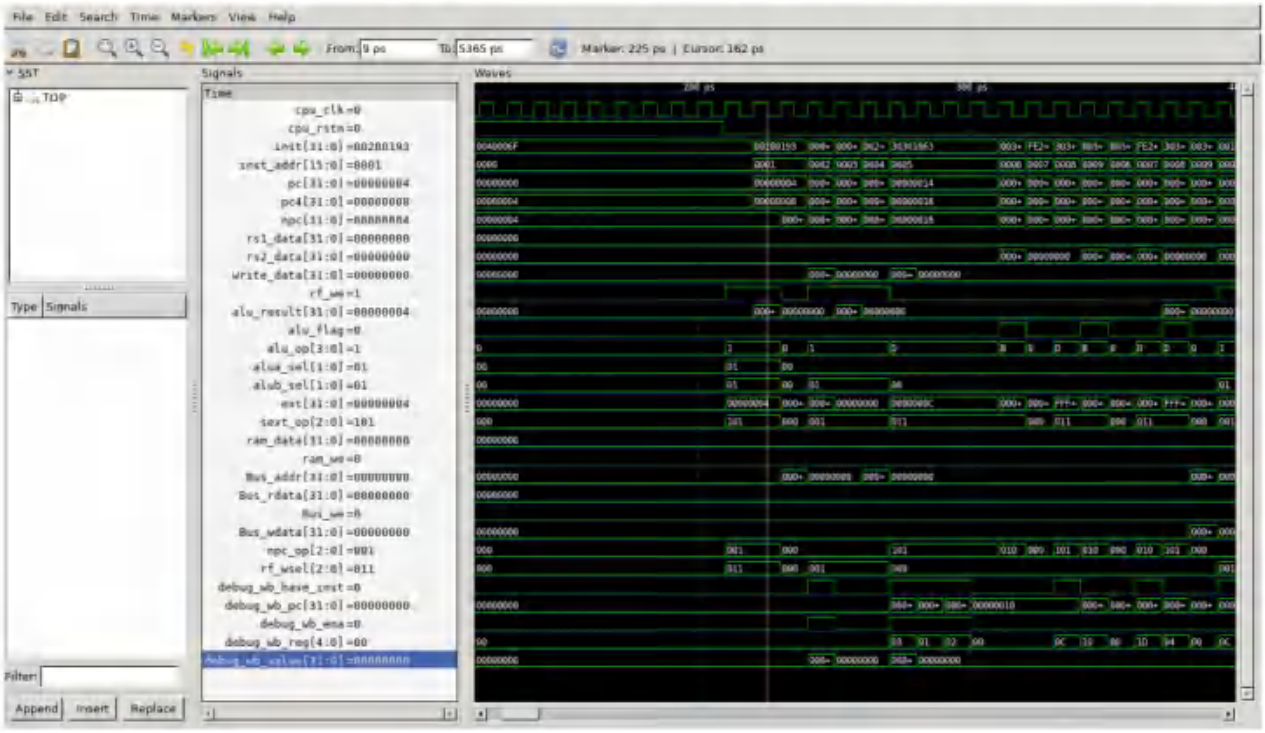
Bus_rdata[31:0] = 00000000：总线读数据

Bus_we = 0：总线写使能

Bus_wdata[31:0] = 00000000：总线写数据

时序分析：

从波形图可以看出，CPU 在 431ps 时刻完成了 LUI 指令的执行，为后续的访存操作做准备。在后续的时钟周期中，可以看到 ram_we 信号出现脉冲，表示执行 SB 和 LB 指令时的内存访问操作。整个访存指令序列展示了 CPU 的存储和加载功能，验证了内存接口的正确性。



```
00000000 <_start>:
    0:   0040006f                jal  x0,4 <reset_vector>

00000004 <reset_vector>:
    4:   00200193                addi    x3,x0,2
    8:   00000093                addi    x1,x0,0
    c:   00000113                addi    x2,x0,0
   10:   0020d663                bge x1,x2,1c <reset_vector+0x18>
   14:   30301863                bne x0,x3,324 <fail>
   18:   00301663                bne x0,x3,24 <test_3>
   1c:   fe20dee3                bge x1,x2,18 <reset_vector+0x14>
```

20: 30301263 bne x0,x3,324 <fail>

从波形图中可以看到，在时间点 162ps 处，CPU 正在执行 ADDI 指令 00200193 addi x3,x0,2，这是分支跳转指令序列的一部分。

指令解析：

指令地址：04 (PC = 00000004)

指令内容：00200193

操作类型：ADDI (Add Immediate)

目标寄存器：x3

源寄存器：x0

立即数：2

关键信号分析：

1. 取指阶段 (Instruction Fetch)
   - pc[31:0] = 00000004：程序计数器指向当前指令地址
   - inst[31:0] = 00200193：从指令存储器取出的指令内容
   - inst_addr[15:0] = 0001：指令存储器地址
   - pc4[31:0] = 00000008：下一条指令地址 (PC + 4)
2. 译码阶段 (Decode)
   - sext_op[2:0] = 101：符号扩展操作码，表示 Itype 立即数扩展
   - ext[31:0] = 00000004：扩展后的立即数值
   - rs1_data[31:0] = 00000000：源寄存器 x0 的值
   - rs2_data[31:0] = 00000000：源寄存器 2 数据（未使用）
3. 执行阶段 (Execute)
   - alu_op[3:0] = 1：ALU 操作码，表示加法操作
   - alua_sel[1:0] = 01：ALU A 输入选择 rs1_data
   - alub_sel[1:0] = 01：ALU B 输入选择 ext
   - alu_result[31:0] = 00000004：ALU 运算结果 (0 + 4)
   - alu_flag = 0：ALU 标志位
4. 写回阶段 (Writeback)
   - rf_we = 1：寄存器文件写使能
   - write_data[31:0] = 00000000：写入寄存器的数据
   - rf_wsel[2:0] = 011：寄存器写回选择信号
   - debug_wb_reg[4:0] = 00：调试写回寄存器地址

分支跳转指令序列分析：

从汇编代码可以看出，这是一个完整的分支跳转测试序列：

1. JAL 指令 (0040006f)：无条件跳转到地址 4
2. ADDI 指令 (00200193)：将 x0 的值加上 2，结果存入 x3
3. ADDI 指令 (00000093)：将 x0 的值加上 0，结果存入 x1
4. ADDI 指令 (00000113)：将 x0 的值加上 0，结果存入 x2
5. BGE 指令 (0020d663)：如果 x1 >= x2，则跳转到地址 1c
6. BNE 指令 (30301863)：如果 x0 != x3，则跳转到 fail
7. BNE 指令 (00301663)：如果 x0 != x3，则跳转到 test_3
8. BGE 指令 (fe20dee3)：如果 x1 >= x2，则跳转到地址 18

分支控制信号分析：

npc_op[2:0] = 001：下一 PC 操作码，表示正常顺序执行
pc_sel：在波形中未显示，但用于选择下一 PC 来源
alu_flag = 0：ALU 比较结果，用于分支判断

时序分析：

从波形图可以看出，CPU 在 162ps 时刻完成了 ADDI 指令的执行，为后续的分支跳转指令做准备。在后续的时钟周期中，可以看到 npc_op 信号的变化，表示不同分支指令的执行：

npc_op = 000：正常顺序执行
npc_op = 101：BGE 分支指令
npc_op = 010：BNE 分支指令

2 流水线 CPU 设计与实现

2.1 流水线 CPU 数据通路

要求：贴出完整的流水线数据通路图，无需画出模块内的具体逻辑，但要标出模块的接口信号名、模块之间信号线的信号名和位宽，并用文字阐述各模块的功能。此外，数据通路图应当能体现出流水线是如何划分的，并用文字阐述每个流水级具备什么功能、需要完成哪些操作。

主要模块及其接口信号：

1. PC 模块

输入：rst、clk、pc_stall、IF_ID_stall、npc

输出：pc_have_inst、pc

2. NPC 模块

输入：br、op、PC、alu_result、offset

输出：pc4、npc

3. IROM 模块

输入：a

输出：spo

4. IF_ID 模块

输入：clk、rst、IF_ID_stall、pc_stall、IF_pc（32 位）、IF_pc4（32 位）、IF_inst（32 位）、IF_have_inst

输出：ID_have_inst、ID_pc（32 位）、ID_pc4（32 位）、ID_inst（32 位）

5. idecode 模块

输入：inst

输出：opcode、funct3、funct7、rs1、rs2、rd、imm

6. RF 模块

输入：clk、rf_we、rs1、rs2、rd、rf_wsel、ext、alu_result、mem_data、pc4

输出：rs1_data、write_data、rs2_data

7. SEXT 模块

输入：sext_op、din

输出：ext

8. controller 模块

输入：opcode、funct3、funct7

输出：sext_op、npc_op、alu_op、rf_wsel、alua_sel、alub_sel、ram_we、rf_we

9. ALU 模块

输入：alua_sel、alub_sel、alu_op、pc、ext、rs1_data、rs2_data

输出：alu_result、alu_flag

10. ID_EX 模块

输入：clk、rst、ID_EX_stall、pc_stall、IF_ID_stall、各种 ID 级控制信号和数据

输出：EX_have_inst、EX_rf_wsel、EX_inst、EX_rf_we、EX_npc_op、EX_alu_op、EX_ram_we、EX_alua_sel、EX_alub_sel、EX_rs1_data、EX_rs2_data、EX_ext、EX_pc

11. EX_MEM 模块

输入：clk、rst、各种 EX 级控制信号和数据
输出：MEM_pc、MEM_have_inst、MEM_rf_wsel、MEM_inst、MEM_rf_we、MEM_ext、MEM_ram_we、MEM_alu_result、MEM_rs2_data
12. DRAM 模块
输入：clk、a、we、d
输出：spo
13. MEM_WB 模块
输入：clk、rst、各种 MEM 级控制信号和数据
输出：WB_pc、WB_have_inst、WB_rf_wsel、WB_inst、WB_rf_we、WB_ext、WB_mem_data、WB_alu_result、WB_pc4
14. hazard_detection 模块
输入：rst、ID_inst、各级写使能和写地址
输出：pc_stall、IF_ID_stall、ID_EX_stall
15. Bridge 模块
输入：各种 CPU 和外设接口信号
输出：地址解码和读写控制信号

流水线划分及各流水级功能
1. IF 级
功能：从指令存储器中取指令
主要操作：
PC 值更新：根据 NPC 模块的输出更新程序计数器
指令取指：从 IROM 中读取当前 PC 地址对应的指令
流水线寄存器更新：将 PC、PC+4、指令等信息传递到 IF/ID 流水线寄存器

2. ID 级
功能：对指令进行译码，生成控制信号和操作数
主要操作：
指令译码：解析指令的操作码、功能码、寄存器地址、立即数等字段
寄存器读取：从寄存器文件中读取源操作数
符号扩展：对立即数进行符号扩展
控制信号生成：根据指令类型生成各种控制信号
冒险检测：检测数据冒险并生成暂停信号

3. EX 级
功能：执行算术逻辑运算和地址计算
主要操作：
ALU 运算：根据 ALU 操作码执行算术逻辑运算
分支判断：计算分支条件并生成分支标志
地址计算：为存储器访问计算有效地址
操作数选择：根据控制信号选择 ALU 的操作数来源

4. MEM 级
功能：访问数据存储器

主要操作：
存储器访问：根据地址和写使能信号读写数据存储器
数据传递：将 ALU 结果和存储器数据传递到下一级
控制信号传递：将写回控制信号传递到写回级

5. WB 级
功能：将结果写回寄存器文件
主要操作：
写回数据选择：根据写回选择信号选择写回数据来源
寄存器写回：将选定的数据写入目标寄存器
调试信息输出：输出写回相关的调试信息

2.2 流水线 CPU 模块详细设计

要求：以表格的形式列出所有**与单周期不同**的部件的接口信号、位宽、功能描述等，并结合图、表、核心代码等，详细描述这些部件的关键实现。此外，如果实现了冒险控制，必须结合数据通路图，详细说明数据冒险、控制冒险的解决方法。

2.2.1 流水线特有模块接口信号表

IF/ID 模块

信号名称	方向	位宽	功能描述
clk	输入	1	时钟信号
rst	输入	1	复位信号
IF_ID_stall	输入	1	IF/ID暂停信号
pc_stall	输入	1	PC暂停信号
IF_pc	输入	32	IF级PC值
IF_pc4	输入	32	IF级PC+4值
IF_inst	输入	32	IF级指令
IF_have_inst	输入	1	IF级指令有效标志
ID_have_inst	输出	1	ID级指令有效标志
ID_pc	输出	32	ID级PC值
ID_pc4	输出	32	ID级PC+4值
ID_inst	输出	32	ID级指令

ID/EX 模块

ID_EX_stall	输入	1	ID/EX暂停信号
pc_stall	输入	1	PC暂停信号
IF_ID_stall	输入	1	IF/ID暂停信号
ID_have_inst	输入	1	ID级指令有效标志
ID_rf_wsel	输入	3	ID级寄存器写回选择
ID_inst	输入	32	ID级指令
ID_rf_we	输入	1	ID级寄存器写使能
ID_npc_op	输入	3	ID级NPC操作类型
ID_alu_op	输入	4	ID级ALU操作类型
ID_ram_we	输入	1	ID级存储器写使能
ID_alua_sel	输入	2	ID级ALU A操作数选择
ID_alub_sel	输入	2	ID级ALU B操作数选择
ID_rs1_data	输入	32	ID级源寄存器1数据
ID_rs2_data	输入	32	ID级源寄存器2数据
ID_ext	输入	32	ID级扩展立即数
ID_pc	输入	32	ID级PC值
EX_have_inst	输出	1	EX级指令有效标志
EX_rf_wsel	输出	3	EX级寄存器写回选择
EX_inst	输出	32	EX级指令

EX_rf_we	输出	1	EX级寄存器写使能
EX_npc_op	输出	3	EX级NPC操作类型
EX_alu_op	输出	4	EX级ALU操作类型
EX_ram_we	输出	1	EX级存储器写使能
EX_alua_sel	输出	2	EX级ALU A操作数选择
EX_alub_sel	输出	2	EX级ALU B操作数选择
EX_rs1_data	输出	32	EX级源寄存器1数据
EX_rs2_data	输出	32	EX级源寄存器2数据
EX_ext	输出	32	EX级扩展立即数
EX_pc	输出	32	EX级PC值

EX/MEM 模块

EX_have_inst	输入	1	EX级指令有效标志
EX_rf_wsel	输入	3	EX级寄存器写回选择
EX_inst	输入	32	EX级指令
EX_rf_we	输入	1	EX级寄存器写使能
EX_ext	输入	32	EX级扩展立即数
EX_ram_we	输入	1	EX级存储器写使能
EX_alu_result	输入	32	EX级ALU结果
EX_rs2_data	输入	32	EX级源寄存器2数据
EX_pc	输入	32	EX级PC值
MEM_pc	输出	32	MEM级PC值
MEM_have_inst	输出	1	MEM级指令有效标志
MEM_rf_wsel	输出	3	MEM级寄存器写回选择
MEM_inst	输出	32	MEM级指令
MEM_rf_we	输出	1	MEM级寄存器写使能
MEM_ext	输出	32	MEM级扩展立即数
MEM_ram_we	输出	1	MEM级存储器写使能
MEM_alu_result	输出	32	MEM级ALU结果
MEM_rs2_data	输出	32	MEM级源寄存器2数据

MEM/WB 模块

MEM_pc	输入	32	MEM级PC值
MEM_have_inst	输入	1	MEM级指令有效标志
MEM_rf_wsel	输入	3	MEM级寄存器写回选择
MEM_inst	输入	32	MEM级指令
MEM_rf_we	输入	1	MEM级寄存器写使能
MEM_ext	输入	32	MEM级扩展立即数
MEM_mem_data	输入	32	MEM级存储器数据
MEM_alu_result	输入	32	MEM级ALU结果
MEM_pc4	输入	32	MEM级PC+4值
WB_pc	输出	32	WB级PC值
WB_have_inst	输出	1	WB级指令有效标志
WB_rf_wsel	输出	3	WB级寄存器写回选择
WB_inst	输出	32	WB级指令
WB_rf_we	输出	1	WB级寄存器写使能
WB_ext	输出	32	WB级扩展立即数
WB_mem_data	输出	32	WB级存储器数据
WB_alu_result	输出	32	WB级ALU结果
WB_pc4	输出	32	WB级PC+4值

冒险检测

ID_inst	输入	32	ID级指令
EX_rf_we	输入	1	EX级寄存器写使能
MEM_rf_we	输入	1	MEM级寄存器写使能
WB_rf_we	输入	1	WB级寄存器写使能
ID_rs1	输入	5	ID级源寄存器1地址
ID_rs2	输入	5	ID级源寄存器2地址
EX_rd	输入	5	EX级目标寄存器地址
MEM_rd	输入	5	MEM级目标寄存器地址
WB_rd	输入	5	WB级目标寄存器地址
pc_stall	输出	1	PC暂停信号
IF_ID_stall	输出	1	IF/ID暂停信号
ID_EX_stall	输出	1	ID/EX暂停信号

2.2.2 流水线寄存器模块关键实现

IF_ID 模块核心实现

```
module IF_ID(
    input wire clk,
    input wire rst,
    input wire IF_ID_stall,
    input wire pc_stall,
    input wire[31:0] IF_pc,
    input wire[31:0] IF_pc4,
    input wire[31:0] IF_inst,
```

```
    input wire IF_have_inst,
    output reg[31:0] ID_pc,
    output reg[31:0] ID_pc4,
    output reg[31:0] ID_inst,
    output reg ID_have_inst
);

// 流水线寄存器更新逻辑
always @(posedge clk or posedge rst) begin
    if(rst) begin
        // 复位时清零所有寄存器
        ID_pc <= 32'h0;
        ID_pc4 <= 32'h0;
        ID_inst <= 32'h0;
        ID_have_inst <= 1'b0;
    end
    else if(IF_ID_stall) begin
        // 暂停时保持当前值不变
        ID_pc <= ID_pc;
        ID_pc4 <= ID_pc4;
        ID_inst <= ID_inst;
        ID_have_inst <= ID_have_inst;
    end
    else begin
        // 正常更新：将 IF 级数据传递到 ID 级
        ID_pc <= IF_pc;
        ID_pc4 <= IF_pc4;
        ID_inst <= IF_inst;
        ID_have_inst <= IF_have_inst;
    end
end

Endmodule

ID_EX 模块核心实现
module ID_EX(
    input wire clk,
    input wire rst,
    input wire ID_EX_stall,
    input wire pc_stall,
    input wire IF_ID_stall,
    // 控制信号和数据输入
    input wire ID_have_inst,
    input wire [2:0] ID_rf_wsel,
```

```
    input wire [31:0] ID_inst,
    input wire ID_rf_we,
    input wire [2:0] ID_npc_op,
    input wire [3:0] ID_alu_op,
    input wire ID_ram_we,
    input wire [1:0] ID_alua_sel,
    input wire [1:0] ID_alub_sel,
    input wire [31:0] ID_rs1_data,
    input wire [31:0] ID_rs2_data,
    input wire [31:0] ID_ext,
    input wire [31:0] ID_pc,
    // 输出到 EX 级
    output reg EX_have_inst,
    output reg [2:0] EX_rf_wsel,
    output reg [31:0] EX_inst,
    output reg EX_rf_we,
    output reg [2:0] EX_npc_op,
    output reg [3:0] EX_alu_op,
    output reg EX_ram_we,
    output reg [1:0] EX_alua_sel,
    output reg [1:0] EX_alub_sel,
    output reg [31:0] EX_rs1_data,
    output reg [31:0] EX_rs2_data,
    output reg [31:0] EX_ext,
    output reg [31:0] EX_pc
);

always @(posedge clk or posedge rst) begin
    if(rst) begin
        // 复位时清零所有寄存器
        EX_have_inst <= 1'b0;
        EX_rf_wsel   <= 3'b0;
        EX_inst      <= 32'b0;
        EX_rf_we     <= 1'b0;
        EX_npc_op    <= 3'b0;
        EX_alu_op    <= 4'b0;
        EX_ram_we    <= 1'b0;
        EX_alua_sel  <= 2'b0;
        EX_alub_sel  <= 2'b0;
        EX_rs1_data  <= 32'b0;
        EX_rs2_data  <= 32'b0;
        EX_ext       <= 32'b0;
        EX_pc        <= 32'b0;
    end
```

```
    else if(ID_EX_stall) begin
        // 暂停时保持当前值不变
        EX_have_inst <= EX_have_inst;
        EX_rf_wsel   <= EX_rf_wsel;
        EX_inst      <= EX_inst;
        EX_rf_we     <= EX_rf_we;
        EX_npc_op    <= EX_npc_op;
        EX_alu_op    <= EX_alu_op;
        EX_ram_we    <= EX_ram_we;
        EX_alua_sel  <= EX_alua_sel;
        EX_alub_sel  <= EX_alub_sel;
        EX_rs1_data  <= EX_rs1_data;
        EX_rs2_data  <= EX_rs2_data;
        EX_ext       <= EX_ext;
        EX_pc        <= EX_pc;
    end
    else begin
        // 正常传递 ID 级信号到 EX 级
        EX_have_inst <= ID_have_inst;
        EX_rf_wsel   <= ID_rf_wsel;
        EX_inst      <= ID_inst;
        EX_rf_we     <= ID_rf_we;
        EX_npc_op    <= ID_npc_op;
        EX_alu_op    <= ID_alu_op;
        EX_ram_we    <= ID_ram_we;
        EX_alua_sel  <= ID_alua_sel;
        EX_alub_sel  <= ID_alub_sel;
        EX_rs1_data  <= ID_rs1_data;
        EX_rs2_data  <= ID_rs2_data;
        EX_ext       <= ID_ext;
        EX_pc        <= ID_pc;
    end
end

endmodule
```

2.2.3 冒险控制模块详细设计

数据冒险检测逻辑

```
/// 检查 ID 级指令的 rs1/rs2 是否与 EX/MEM/WB 级的 rd 冲突，且对方有写使能
wire hazard_rs1_ex  = (ID_rs1 != 0) && (ID_rs1 == EX_rd)  && EX_rf_we;
wire hazard_rs2_ex  = (ID_rs2 != 0) && (ID_rs2 == EX_rd)  && EX_rf_we;
wire hazard_rs1_mem = (ID_rs1 != 0) && (ID_rs1 == MEM_rd) && MEM_rf_we;
wire hazard_rs2_mem = (ID_rs2 != 0) && (ID_rs2 == MEM_rd) && MEM_rf_we;
```

```
wire hazard_rs1_wb   = (ID_rs1 != 0) && (ID_rs1 == WB_rd)   && WB_rf_we;
wire hazard_rs2_wb   = (ID_rs2 != 0) && (ID_rs2 == WB_rd)   && WB_rf_we;

// 只要有一个源寄存器发生冒险，就需要暂停
wire data_hazard = hazard_rs1_ex || hazard_rs2_ex ||
                   hazard_rs1_mem || hazard_rs2_mem ||
                   hazard_rs1_wb || hazard_rs2_wb;

// 输出暂停信号
assign pc_stall      = data_hazard;
assign IF_ID_stall = data_hazard;
assign ID_EX_stall = data_hazard;

冒险检测原理：
1. 寄存器地址比较：比较当前指令的源寄存器地址与后续流水级的目标寄存器地址
2. 写使能检查：确保后续流水级确实要写寄存器
3. 寄存器读取检查：确保当前指令确实要读取该寄存器
4. 零寄存器排除：排除对零寄存器的读写，避免误判

控制冒险处理逻辑
// PC 暂停控制逻辑
always @() begin
    if(rst) begin
        pc_stall = 1'b0;
    end
    else if(ex_npc_op == NPC_JMP) begin
        // 无条件跳转：不暂停 PC，直接跳转
        pc_stall = 1'b0;
    end
    else if(ex_npc_op == NPC_PC4) begin
        // 顺序执行：检查数据冒险
        if(RAW_EX)          pc_stall = 1'b1;
        else if(RAW_MEM)    pc_stall = 1'b1;
        else if(RAW_WB)     pc_stall = 1'b1;
        else                pc_stall = 1'b0;
    end
    else begin
        // 条件分支：根据分支结果决定
        if(ex_alu_f)        pc_stall = 1'b0;  // 分支成立，不暂停
        else if(RAW_EX)     pc_stall = 1'b1;  // 数据冒险优先
        else if(RAW_MEM)    pc_stall = 1'b1;
        else if(RAW_WB)     pc_stall = 1'b1;
        else                pc_stall = 1'b0;
```

```
    end
end

// IF/ID 暂停控制逻辑
always @() begin
    if(rst) begin
        IF_ID_stall = 1'b0;
    end
    else begin
        case (ex_npc_op)
            NPC_PC4: begin
                // 顺序执行：正常传递
                IF_ID_stall = 1'b0;
            end
            NPC_JMP: begin
                // 无条件跳转：暂停 IF/ID，清空流水线
                IF_ID_stall = 1'b1;
            end
            NPC_BEQ: begin
                // 条件分支：根据分支结果决定
                if(ex_alu_f)      IF_ID_stall = 1'b1;  // 分支成立，清空
                else              IF_ID_stall = 1'b0;  // 分支不成立，继续
            end
            // 其他分支指令类似处理
        endcase
    end
End

控制冒险处理策略：
1. 无条件跳转：立即暂停 IF/ID 流水线，清空流水线
2. 条件分支：根据分支条件决定是否暂停
3. 数据冒险：暂停 PC 更新，等待数据就绪

2.2.4 流水线控制信号传递机制
// 寄存器写回选择流水线寄存器
always @(posedge clk or posedge rst) begin
    if (rst)                                EX_rf_wsel <= WB_ALU;
    else if(pc_stall & ID_EX_stall)         EX_rf_wsel <= WB_ALU;  // 冒险时使用
默认值
    else if(IF_ID_stall & ID_EX_stall)      EX_rf_wsel <= WB_ALU;
    else                                    EX_rf_wsel <= ID_rf_wsel; // 正常
传递
end
```

```
// ALU 操作流水线寄存器
always @(posedge clk or posedge rst) begin
    if (rst)                                    EX_alu_op <= 4'b0;
    else if(pc_stall & ID_EX_stall)      EX_alu_op <= 4'b0;      // 冒险时清零
    else if(IF_ID_stall & ID_EX_stall)   EX_alu_op <= 4'b0;
    else                                        EX_alu_op <= ID_alu_op;   // 正常
传递
end
```

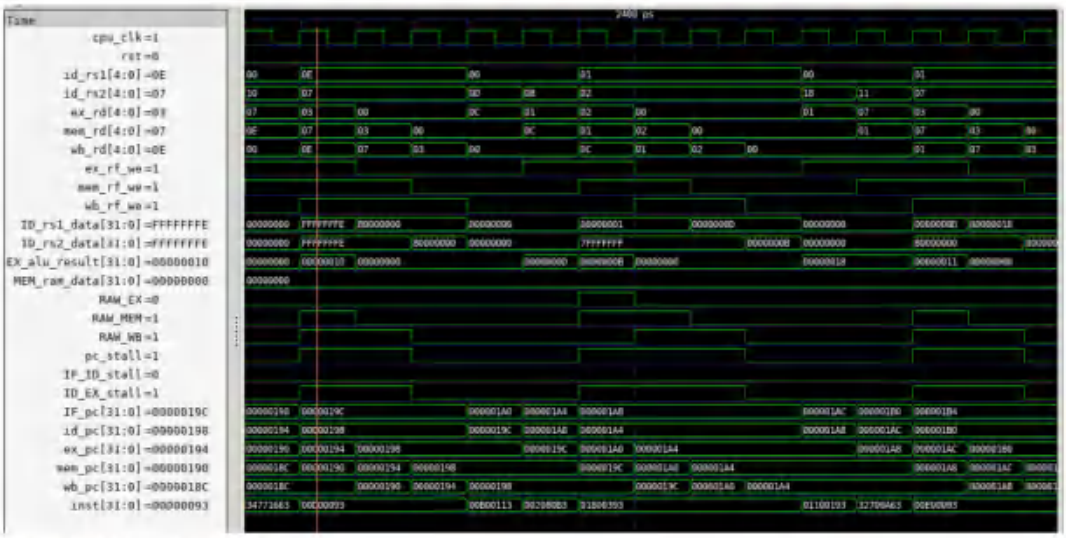
控制信号传递特点：

1. 同步传递：控制信号与数据一起在流水线中传递
2. 冒险处理：在冒险情况下，控制信号采用默认值或保持值
3. 级间隔离：每级都有独立的控制信号，避免级间干扰

2.3 流水线 CPU 仿真及结果分析

要求：包含控制冒险和数据冒险三种情形的仿真截图，以及波形分析。若仅实现了理想流水，则此处贴上理想流水的仿真截图及详细的波形分析。

RAWA 型



0000019c <test_17>:

19c:	00d00093	addi	x1,x0,13
1a0:	00b00113	addi	x2,x0,11
1a4:	002080b3	add	x1,x1,x2
1a8:	01800393	addi	x7,x0,24
1ac:	01100193	addi	x3,x0,17
1b0:	32709a63	bne	x1,x7,4e4 <fail>

波形分析：test_17 指令序列

指令序列分析

0000019c <test_17>:

19c:	00d00093	addi	x1,x0,13	x1 = 13
1a0:	00b00113	addi	x2,x0,11	x2 = 11
1a4:	002080b3	add	x1,x1,x2	x1 = x1 + x2 (RAW 冒险)
1a8:	01800393	addi	x7,x0,24	x7 = 24
1ac:	01100193	addi	x3,x0,17	x3 = 17
1b0:	32709a63	bne	x1,x7,4e4 <fail>	分支指令

1. 当前时间点状态（约 745ps）

从波形图中可以看到当前状态：

寄存器地址：

id_rs1[4:0] = 0E (14)　ID 级读取寄存器 x14

id_rs2[4:0] = 07 (7)　ID 级读取寄存器 x7

ex_rd[4:0] = 03 (3)　EX 级写回寄存器 x3

mem_rd[4:0] = 07 (7)　MEM 级写回寄存器 x7

wb_rd[4:0] = 0E (14)　WB 级写回寄存器 x14

数据值：

ID_rs1_data[31:0] = FFFFFFFE　ID 级读取的源寄存器 1 数据

ID_rs2_data[31:0] = FFFFFFFE　ID 级读取的源寄存器 2 数据

EX_alu_result[31:0] = 00000010　EX 级 ALU 结果

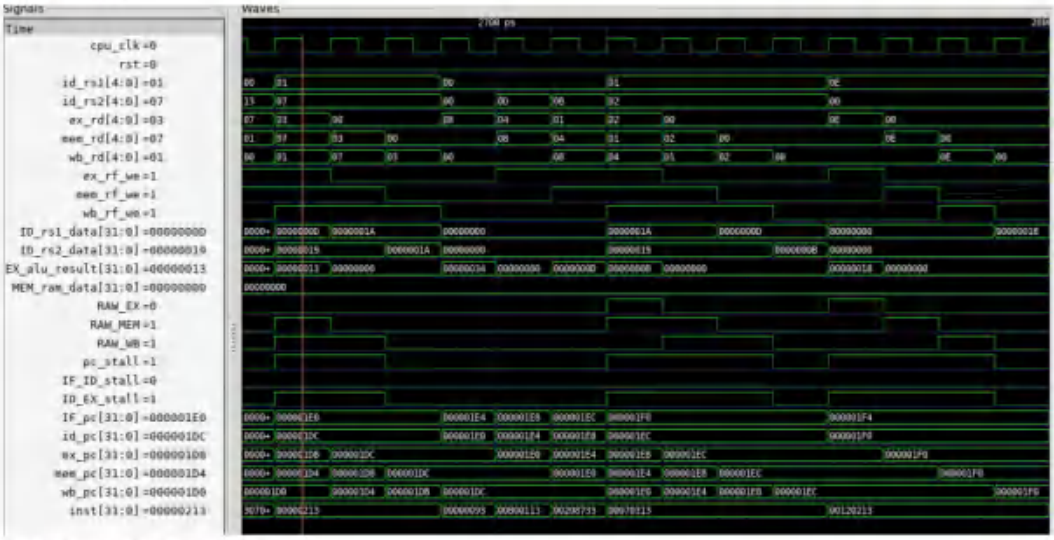
MEM_ram_data[31:0] = 00000000　MEM 级存储器数据

2. 冒险检测分析

RAW 冒险标志：
RAW_EX = 0　无 EX 级数据冒险
RAW_MEM = 1　检测到 MEM 级数据冒险
RAW_WB = 1　检测到 WB 级数据冒险
冒险原因分析：
1. MEM 级冒险：当前 ID 级指令需要读取寄存器 x7，而 MEM 级指令正在写回寄存器 x7
2. WB 级冒险：当前 ID 级指令需要读取寄存器 x14，而 WB 级指令正在写回寄存器 x14
3. 暂停控制分析
暂停信号：
pc_stall = 1　PC 暂停，防止新指令进入流水线
IF_ID_stall = 0　IF/ID 流水线寄存器不暂停
ID_EX_stall = 1　ID/EX 流水线寄存器暂停
暂停策略：
暂停 PC 更新，防止新指令进入
暂停 ID/EX 流水线寄存器，保持当前指令在 ID 级
不暂停 IF/ID 流水线寄存器，允许指令继续从 IF 级进入 ID 级
4. 程序计数器状态
各流水级 PC 值：
IF_pc[31:0] = 0000019C　IF 级程序计数器
id_pc[31:0] = 00000198　ID 级程序计数器
ex_pc[31:0] = 00000194　EX 级程序计数器
mem_pc[31:0] = 00000190　MEM 级程序计数器
wb_pc[31:0] = 0000018C　WB 级程序计数器
PC 值分析：
PC 值递减，表明指令在流水线中正常流动
某些 PC 值会保持一段时间，对应暂停期间
5. 冒险处理机制
检测逻辑：
verilog

```
// MEM 级冒险检测
assign rs1_id_mem_hazard = (MEM_rd == ID_rs1) & MEM_rf_we & id_rf1 & (MEM_rd != 5'b0) & (ID_rs1 != 5'b0);
assign rs2_id_mem_hazard = (MEM_rd == ID_rs2) & MEM_rf_we & id_rf2 & (MEM_rd != 5'b0) & (ID_rs2 != 5'b0);

// WB 级冒险检测
assign rs1_id_wb_hazard = (WB_rd == ID_rs1) & WB_rf_we & id_rf1 & (WB_rd != 5'b0) & (ID_rs1 != 5'b0);
assign rs2_id_wb_hazard = (WB_rd == ID_rs2) & WB_rf_we & id_rf2 & (WB_rd != 5'b0) & (ID_rs2 != 5'b0);
```

RAWB 型



000001e0 <test_20>:

1e0:	00000213	addi	x4,x0,0
1e4:	00d00093	addi	x1,x0,13
1e8:	00b00113	addi	x2,x0,11
1ec:	00208733	add	x14,x1,x2
1f0:	00070313	addi	x6,x14,0
1f4:	00120213	addi	x4,x4,1 1 <_start+0x1>
1f8:	00200293	addi	x5,x0,2
1fc:	fe5214e3	bne	x4,x5,1e4 <test_20+0x4>
200:	01800393	addi	x7,x0,24
204:	01400193	addi	x3,x0,20
208:	2c731e63	bne	x6,x7,4e4 <fail>

指令序列分析

000001e0 <test_20>:

1e0:	00000213	addi	x4,x0,0	x4 = 0
1e4:	00d00093	addi	x1,x0,13	x1 = 13
1e8:	00b00113	addi	x2,x0,11	x2 = 11
1ec:	00208733	add	x14,x1,x2	x14 = x1 + x2
1f0:	00070313	addi	x6,x14,0	x6 = x14 (RAW 冒险)
1f4:	00120213	addi	x4,x4,1	x4 = x4 + 1
1f8:	00200293	addi	x5,x0,2	x5 = 2
1fc:	fe5214e3	bne	x4,x5,1e4	循环分支
200:	01800393	addi	x7,x0,24	x7 = 24
204:	01400193	addi	x3,x0,20	x3 = 20
208:	2c731e63	bne	x6,x7,4e4 <fail>	

1. 当前时间点状态（约 2700ps）

寄存器地址：

id_rs1[4:0] = 01 (1)　ID 级读取寄存器 x1

id_rs2[4:0] = 07 (7)　ID 级读取寄存器 x7

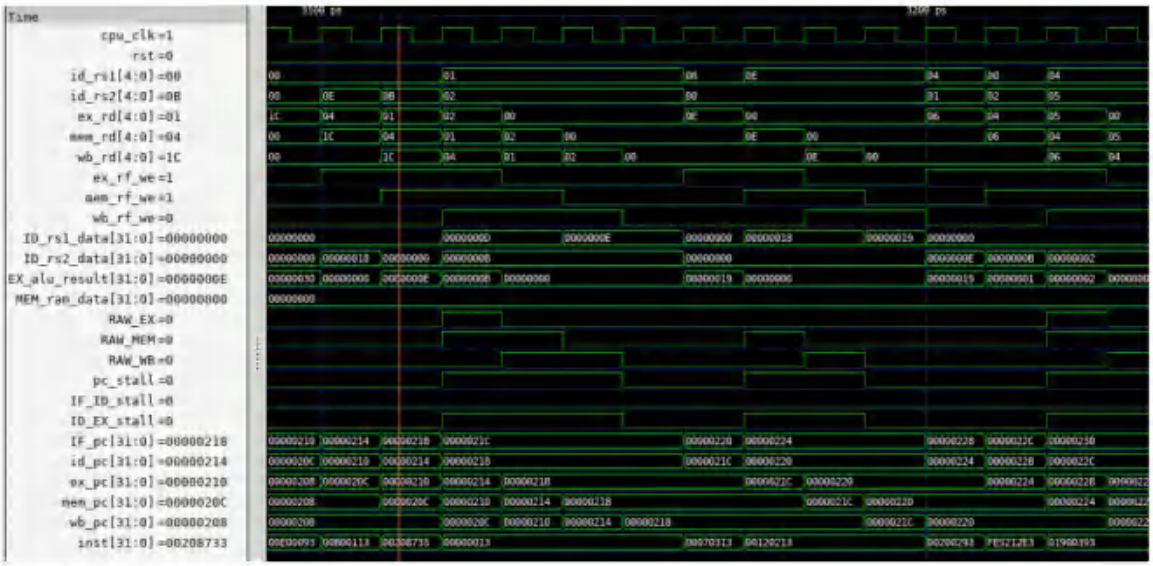
ex_rd[4:0] = 03 (3)　EX 级写回寄存器 x3

mem_rd[4:0] = 07 (7)　MEM 级写回寄存器 x7
wb_rd[4:0] = 01 (1)　WB 级写回寄存器 x1
数据值：
ID_rs1_data[31:0] = 0000000D (13)　ID 级读取的源寄存器 1 数据
ID_rs2_data[31:0] = 00000019 (25)　ID 级读取的源寄存器 2 数据
EX_alu_result[31:0] = 00000013 (19)　EX 级 ALU 结果
MEM_ram_data[31:0] = 00000000 (0)　MEM 级存储器数据
2. 冒险检测分析
RAW 冒险标志：
RAW_EX = 0　无 EX 级数据冒险
RAW_MEM = 1　检测到 MEM 级数据冒险
RAW_WB = 1　检测到 WB 级数据冒险
冒险原因分析：
1. MEM 级冒险：当前 ID 级指令需要读取寄存器 x7，而 MEM 级指令正在写回寄存器 x7
2. WB 级冒险：当前 ID 级指令需要读取寄存器 x1，而 WB 级指令正在写回寄存器 x1
3. 暂停控制分析
暂停信号：
pc_stall = 1　PC 暂停，防止新指令进入流水线
IF_ID_stall = 0　IF/ID 流水线寄存器不暂停
ID_EX_stall = 1　ID/EX 流水线寄存器暂停
暂停策略：
暂停 PC 更新，防止新指令进入
暂停 ID/EX 流水线寄存器，保持当前指令在 ID 级
不暂停 IF/ID 流水线寄存器，允许指令继续从 IF 级进入 ID 级
4. 程序计数器状态
各流水级 PC 值：
IF_pc[31:0] = 000001E0　IF 级程序计数器
id_pc[31:0] = 000001DC　ID 级程序计数器
ex_pc[31:0] = 000001D8　EX 级程序计数器
mem_pc[31:0] = 000001D4　MEM 级程序计数器
wb_pc[31:0] = 000001D0　WB 级程序计数器
PC 值分析：
PC 值递减，表明指令在流水线中正常流动
某些 PC 值会保持一段时间，对应暂停期间
5. 关键冒险场景分析
RAW 情形 B 冒险：
assembly
1ec:00208733　　add x14,x1,x2　　x14 = x1 + x2
1f0:00070313　　addi　x6,x14,0　　x6 = x14 (RAW 冒险)

冒险检测逻辑：
// MEM 级冒险检测

assign rs1_id_mem_hazard = (MEM_rd == ID_rs1) & MEM_rf_we & id_rf1 & (MEM_rd != 5'b0) & (ID_rs1 != 5'b0);
// MEM_rd = x7, ID_rs1 = x1, 满足冒险条件
6. 循环分支分析
控制冒险：
1fc: fe5214e3 bne x4,x5,1e4 循环分支
分支处理：
当分支条件满足时，需要清空流水线
跳转到地址 0x1e4 继续执行循环
每次循环都会产生控制冒险
7. 波形特征总结
数据冒险特征：
RAW_MEM 和 RAW_WB 信号高电平，表示检测到数据冒险
pc_stall 和 ID_EX_stall 信号高电平，表示流水线暂停
PC 值在某些周期保持不变，表示暂停期间
流水线行为：
指令在流水线中正常流动，但在冒险期间会暂停
数据冒险通过暂停机制得到正确处理
流水线在冒险解决后恢复正常执行
循环执行特征：
分支指令导致 PC 跳转到循环开始位置
每次循环都会产生相同的数据冒险
冒险处理机制确保循环的正确执行
8. 冒险处理机制验证
检测逻辑：
// 冒险检测汇总
assign RAW_EX = rs1_id_ex_hazard | rs2_id_ex_hazard;
assign RAW_MEM = rs1_id_mem_hazard | rs2_id_mem_hazard;
assign RAW_WB = rs1_id_wb_hazard | rs2_id_wb_hazard;
// 暂停控制
assign pc_stall = RAW_EX | RAW_MEM | RAW_WB;
assign ID_EX_stall = pc_stall;

RAWC 型



0000020c <test_21>:

20c:	00000213	addi	x4,x0,0
210:	00e00093	addi	x1,x0,14
214:	00b00113	addi	x2,x0,11
218:	00208733	add	x14,x1,x2
21c:	00000013	addi	x0,x0,0
220:	00070313	addi	x6,x14,0
224:	00120213	addi	x4,x4,1 1 <_start+0x1>
228:	00200293	addi	x5,x0,2
22c:	fe5212e3	bne	x4,x5,210 <test_21+0x4>
230:	01900393	addi	x7,x0,25
234:	01500193	addi	x3,x0,21
238:	2a731663	bne	x6,x7,4e4 <fail>

指令序列分析

0000020c <test_21>:

20c:	00000213	addi	x4,x0,0	x4 = 0
210:	00e00093	addi	x1,x0,14	x1 = 14
214:	00b00113	addi	x2,x0,11	x2 = 11
218:	00208733	add	x14,x1,x2	x14 = x1 + x2
21c:	00000013	addi	x0,x0,0	NOP
220:	00070313	addi	x6,x14,0	x6 = x14 (RAW 情形 C 冒险)
224:	00120213	addi	x4,x4,1	x4 = x4 + 1
228:	00200293	addi	x5,x0,2	x5 = 2
22c:	fe5212e3	bne	x4,x5,210	循环分支
230:	01900393	addi	x7,x0,25	x7 = 25
234:	01500193	addi	x3,x0,21	x3 = 21
238:	2a731663	bne	x6,x7,4e4 <fail>	

1. 当前时间点状态（约 3100ps）

各流水级 PC 值：

IF_pc[31:0] = 00000218　IF 级程序计数器

id_pc[31:0] = 00000214　ID 级程序计数器
ex_pc[31:0] = 00000210　EX 级程序计数器
mem_pc[31:0] = 0000020C　MEM 级程序计数器
wb_pc[31:0] = 00000208　WB 级程序计数器
寄存器地址：
id_rs1[4:0] = 00 (0)　ID 级读取寄存器 x0
id_rs2[4:0] = 0E (14)　ID 级读取寄存器 x14
ex_rd[4:0] = 01 (1)　EX 级写回寄存器 x1
mem_rd[4:0] = 04 (4)　MEM 级写回寄存器 x4
wb_rd[4:0] = 00 (0)　WB 级写回寄存器 x0
数据值：
ID_rs1_data[31:0] = 00000000 (0)　ID 级读取的源寄存器 1 数据
ID_rs2_data[31:0] = 00000018 (24)　ID 级读取的源寄存器 2 数据
EX_alu_result[31:0] = 0000000E (14)　EX 级 ALU 结果
MEM_ram_data[31:0] = 00000000 (0)　MEM 级存储器数据
2. 冒险检测分析
RAW 冒险标志：
RAW_EX = 0　无 EX 级数据冒险
RAW_MEM = 0　无 MEM 级数据冒险
RAW_WB = 0　无 WB 级数据冒险
冒险原因分析：
当前时间点没有检测到冒险，因为：
1. 当前 ID 级指令是 addi x2,x0,11，读取 x0 和 x14
2. EX 级指令是 addi x1,x0,14，写回 x1
3. MEM 级指令是 addi x4,x0,0，写回 x4
4. WB 级指令是分支指令，不写回寄存器
3. 关键冒险场景分析
RAW 情形 C 冒险：

```
218:    00208733        add x14,x1,x2     x14 = x1 + x2
21c:    00000013        addi    x0,x0,0       NOP
220:    00070313        addi    x6,x14,0      x6 = x14 (RAW 情形 C 冒险)
```

冒险检测逻辑：
verilog

```
// WB 级冒险检测
assign rs1_id_wb_hazard = (WB_rd == ID_rs1) & WB_rf_we & id_rf1 & (WB_rd != 5'b0) & (ID_rs1 != 5'b0);
assign rs2_id_wb_hazard = (WB_rd == ID_rs2) & WB_rf_we & id_rf2 & (WB_rd != 5'b0) & (ID_rs2 != 5'b0);
```

冒险处理：
当指令 addi x6,x14,0 进入 ID 级时，需要读取 x14
此时 add x14,x1,x2 指令在 WB 级，正在写回 x14
形成 RAW 情形 C 冒险，间隔 2 条指令
4. 控制冒险分析
循环分支冒险：

22c:　fe5212e3　　bne x4,x5,210　　循环分支

分支处理：

当分支条件满足时，需要跳转到地址 0x210

清空 IF/ID 流水线寄存器中的后续指令

从新地址开始取指令

5. 暂停控制分析

暂停信号：

pc_stall = 0　PC 正常更新

IF_ID_stall = 0　IF/ID 流水线寄存器正常更新

ID_EX_stall = 0　ID/EX 流水线寄存器正常更新

正常执行状态：

当前没有检测到冒险，流水线正常运行

指令在流水线中正常流动

各流水级并行执行

6. 数据前递机制分析

前递效果：

虽然存在数据冒险，但 RAW_EX、RAW_MEM、RAW_WB 信号都为 0

这表明可能实现了数据前递机制

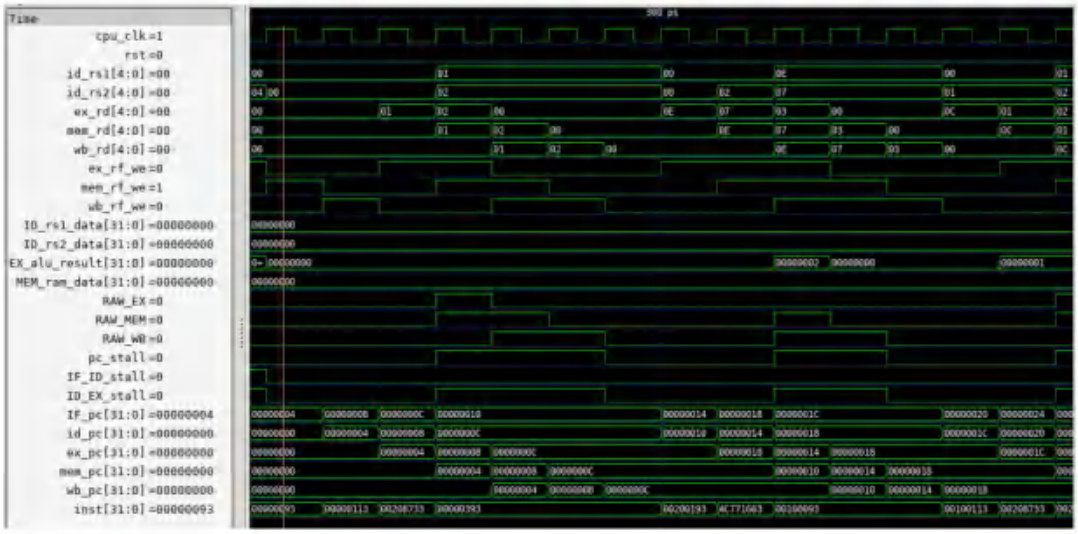
通过前递避免了暂停，提高了流水线效率

7. 冒险处理机制验证

检测逻辑：

```
// 冒险检测汇总
assign RAW_EX = rs1_id_ex_hazard | rs2_id_ex_hazard;
assign RAW_MEM = rs1_id_mem_hazard | rs2_id_mem_hazard;
assign RAW_WB = rs1_id_wb_hazard | rs2_id_wb_hazard;

// 前递逻辑（如果实现）
assign forward_a_sel = (EX_rd == ID_rs1) & EX_rf_we ? 2'b01 :
                       (MEM_rd == ID_rs1) & MEM_rf_we ? 2'b10 :
                       (WB_rd == ID_rs1) & WB_rf_we ? 2'b11 : 2'b00;
```

控制冒险



00000004 <reset_vector>:

4:	00000093	addi	x1,x0,0
8:	00000113	addi	x2,x0,0
c:	00208733	add	x14,x1,x2
10:	00000393	addi	x7,x0,0
14:	00200193	addi	x3,x0,2
18:	4c771663	bne	x14,x7,4e4 <fail>

波形分析：reset_vector 指令序列

00000004 <reset_vector>:

4:	00000093	addi	x1,x0,0	x1 = 0
8:	00000113	addi	x2,x0,0	x2 = 0
c:	00208733	add	x14,x1,x2	x14 = x1 + x2
10:	00000393	addi	x7,x0,0	x7 = 0
14:	00200193	addi	x3,x0,2	x3 = 2
18:	4c771663	bne	x14,x7,4e4 <fail>	分支指令

1. 当前时间点状态（约 270ps）

从波形图中可以看到当前状态：

各流水级 PC 值：

IF_pc[31:0] = 00000004　IF 级程序计数器

id_pc[31:0] = 00000000　ID 级程序计数器

ex_pc[31:0] = 00000000　EX 级程序计数器

mem_pc[31:0] = 00000000　MEM 级程序计数器

wb_pc[31:0] = 00000000　WB 级程序计数器

寄存器地址：

id_rs1[4:0] = 00 (0)　ID 级读取寄存器 x0

id_rs2[4:0] = 00 (0)　ID 级读取寄存器 x0

ex_rd[4:0] = 00 (0)　EX 级写回寄存器 x0

mem_rd[4:0] = 00 (0)　MEM 级写回寄存器 x0

wb_rd[4:0] = 00 (0)　WB 级写回寄存器 x0

数据值：

ID_rs1_data[31:0] = 00000000 (0)　ID 级读取的源寄存器 1 数据

ID_rs2_data[31:0] = 00000000 (0)　ID 级读取的源寄存器 2 数据
EX_alu_result[31:0] = 00000000 (0)　EX 级 ALU 结果
MEM_ram_data[31:0] = 00000000 (0)　MEM 级存储器数据
2. 流水线初始化分析
初始化状态：
当前时间点是流水线的初始填充阶段
指令 jal x0,4（地址 0x00000000）正在流水线中传播
这是典型的流水线启动过程
指令流动：
jal x0,4 指令同时存在于 ID、EX、MEM、WB 级
这是流水线初始化的正常现象
指令 addi x1,x0,0（地址 0x00000004）正在 IF 级被取指
3. 冒险检测分析
RAW 冒险标志：
RAW_EX = 0　无 EX 级数据冒险
RAW_MEM = 0　无 MEM 级数据冒险
RAW_WB = 0　无 WB 级数据冒险
冒险原因分析：
当前没有检测到冒险，因为：
1. jal x0,4 指令不读取源寄存器，只写回 x0（通常被丢弃）
2. 流水线处于初始化阶段，还没有复杂的数据依赖关系
4. 控制冒险分析
跳转指令冒险：
jal x0,4 <reset_vector>　跳转到 reset_vector

控制冒险处理：
jal 指令是控制转移指令，会产生控制冒险
波形显示 pc_stall=0，IF_ID_stall=0，ID_EX_stall=0
这表明控制冒险已经被正确处理，PC 已经正确跳转到 0x00000004
5. 暂停控制分析
暂停信号：
pc_stall = 0　PC 正常更新
IF_ID_stall = 0　IF/ID 流水线寄存器正常更新
ID_EX_stall = 0　ID/EX 流水线寄存器正常更新
正常执行状态：
流水线处于正常执行状态
没有检测到需要暂停的冒险
指令在流水线中正常流动
6. 后续冒险预测
即将出现的冒险：
1. RAW 情形 A 冒险：

assembly			
8:	00000113	addi x2,x0,0	x2 = 0
c:	00208733	add x14,x1,x2	x14 = x1 + x2 (RAW 冒险)

指令 add x14,x1,x2 的 RS2(x2)与指令 addi x2,x0,0 的 RD(x2)冲突

2. 控制冒险：

assembly

18: 4c771663 bne x14,x7,4e4 <fail> 分支指令

分支指令会产生控制冒险

7. 流水线启动机制验证

启动逻辑：

verilog

```
// 流水线初始化
always @(posedge clk or posedge rst) begin
    if(rst) begin
        // 复位所有流水线寄存器
        IF_pc <= 32'b0;
        id_pc <= 32'b0;
        ex_pc <= 32'b0;
        mem_pc <= 32'b0;
        wb_pc <= 32'b0;
    end else begin
        // 正常流水线更新
        // ...
    end
end
```

3 设计过程中遇到的问题及解决方法

要求：包括设计过程中遇到的有价值的错误，或测试过程中遇到的有价值的问题。所谓有价值，指的是解决该错误或问题后，能够学到新的知识和技巧，或加深对已有知识的理解和运用。

1. 计数器同步异步递增问题

问题描述

在流水线 CPU 设计过程中，遇到了一个非常隐蔽的计数器递增问题。最初的现象是：

计数器一直显示为 0，不递增

但后续功能又有反应，trace 也正常

这导致了对问题根源的错误判断

问题分析过程

第一阶段：错误定位

最初怀疑是随机数生成或排序算法的问题

通过汇编硬编码随机数，确认随机数排序功能正常

硬编码随机种子，确认随机数生成也正常

排除了算法层面的问题

第二阶段：深入调试

通过波形分析发现计数器在特定条件下不递增

检查计数器模块的时钟和复位逻辑

发现是同步/异步递增的时序问题

同步递增 vs 异步递增：

verilog

```
// 错误的异步递增实现
always @(posedge clk or posedge rst) begin
    if(rst) begin
        counter <= 0;
    end else begin
        counter <= counter + 1;  // 可能在某些条件下不递增
    end
end

// 正确的同步递增实现
always @(posedge clk) begin
    if(rst) begin
        counter <= 0;
    end else if(enable) begin
        counter <= counter + 1;  // 添加使能信号控制
    end
end
```

问题根源：
计数器在某些时钟周期内没有正确递增
可能是由于复位信号的干扰或时序问题
需要添加明确的使能控制信号

解决方案

```
// 改进的计数器实现
module counter (
    input wire clk,
    input wire rst,
    input wire enable,        // 添加使能信号
    input wire clear,         // 添加清零信号
    output reg [31:0] count
);

    always @(posedge clk) begin
        if(rst || clear) begin
            count <= 32'b0;
        end else if(enable) begin
            count <= count + 1;
        end
    end

endmodule
```

学到的知识
1. 时序设计的重要性：同步逻辑比异步逻辑更可靠
2. 调试策略：从现象到本质，逐步排除法
3. 模块化设计：添加控制信号提高模块的可靠性

2. 冒险检测的 RAW 情形 C 遗漏问题
问题描述
在流水线冒险检测模块的设计中，最初只实现了 RAW 情形 A 和 B 的检测，遗漏了 RAW 情形 C 的检测，导致某些数据冒险无法被正确识别和处理。

问题分析
RAW 冒险的三种情形：
verilog

```
// RAW 情形 A：相邻指令冒险
// 指令 i+1 在 ID 级读取，指令 i 在 EX 级写回
assign rs1_id_ex_hazard = (EX_rd == ID_rs1) & EX_rf_we & id_rf1;
assign rs2_id_ex_hazard = (EX_rd == ID_rs2) & EX_rf_we & id_rf2;
```

```
// RAW 情形 B：间隔 1 条指令冒险
// 指令 i+2 在 ID 级读取，指令 i 在 MEM 级写回
assign rs1_id_mem_hazard = (MEM_rd == ID_rs1) & MEM_rf_we & id_rf1;
assign rs2_id_mem_hazard = (MEM_rd == ID_rs2) & MEM_rf_we & id_rf2;

// RAW 情形 C：间隔 2 条指令冒险（最初遗漏）
// 指令 i+3 在 ID 级读取，指令 i 在 WB 级写回
assign rs1_id_wb_hazard = (WB_rd == ID_rs1) & WB_rf_we & id_rf1;
assign rs2_id_wb_hazard = (WB_rd == ID_rs2) & WB_rf_we & id_rf2;


问题影响
遗漏 RAW 情形 C 的影响：
 某些数据冒险无法被检测到
 可能导致数据竞争和错误结果
 在复杂指令序列中问题更明显

解决方案
// 完整的冒险检测模块
module hazard_detection (
    input wire [4:0] id_rs1,
    input wire [4:0] id_rs2,
    input wire [4:0] ex_rd,
    input wire [4:0] mem_rd,
    input wire [4:0] wb_rd,
    input wire ex_rf_we,
    input wire mem_rf_we,
    input wire wb_rf_we,
    input wire id_rf1,
    input wire id_rf2,
    output wire RAW_EX,
    output wire RAW_MEM,
    output wire RAW_WB
);

    // RAW 情形 A 检测
    wire rs1_id_ex_hazard = (ex_rd == id_rs1) & ex_rf_we & id_rf1 & (ex_rd !=
5'b0) & (id_rs1 != 5'b0);
    wire rs2_id_ex_hazard = (ex_rd == id_rs2) & ex_rf_we & id_rf2 & (ex_rd !=
5'b0) & (id_rs2 != 5'b0);

    // RAW 情形 B 检测
    wire rs1_id_mem_hazard = (mem_rd == id_rs1) & mem_rf_we & id_rf1 &
```

```
(mem_rd != 5'b0) & (id_rs1 != 5'b0);
    wire rs2_id_mem_hazard = (mem_rd == id_rs2) & mem_rf_we & id_rf2 &
(mem_rd != 5'b0) & (id_rs2 != 5'b0);

    // RAW 情形 C 检测（补充）
    wire rs1_id_wb_hazard = (wb_rd == id_rs1) & wb_rf_we & id_rf1 &
(wb_rd != 5'b0) & (id_rs1 != 5'b0);
    wire rs2_id_wb_hazard = (wb_rd == id_rs2) & wb_rf_we & id_rf2 &
(wb_rd != 5'b0) & (id_rs2 != 5'b0);

    // 冒险汇总
    assign RAW_EX = rs1_id_ex_hazard | rs2_id_ex_hazard;
    assign RAW_MEM = rs1_id_mem_hazard | rs2_id_mem_hazard;
    assign RAW_WB = rs1_id_wb_hazard | rs2_id_wb_hazard;

endmodule
```

这些错误和问题的解决过程让我们学到了：

1. 系统化调试方法：从现象到本质，逐步排除
2. 完整性设计思维：考虑所有可能的情况和边界条件
3. 时序设计的重要性：同步逻辑比异步逻辑更可靠
4. 模块化设计原则：添加控制信号提高模块可靠性
5. 测试验证的重要性：通过复杂测试用例验证设计正确性

这些经验对后续的硬件设计和调试工作具有重要的指导意义。

4 总结

要求：谈谈学完本课程后的个人收获以及对本课程的建议和意见。请在认真总结和思考后填写总结。

一、个人收获

通过本课程的学习，我建立了完整的计算机体系结构知识体系：

CPU 设计原理：

深入理解了单周期 CPU 和流水线 CPU 的设计原理

掌握了指令集架构（RISCV）的基本概念和实现方法

学会了 CPU 各功能模块的设计和实现

流水线技术：

理解了五级流水线的结构和原理

掌握了数据冒险、控制冒险的检测和处理方法

学会了流水线寄存器的设计和时序控制

冒险处理机制：

深入理解了 RAW、WAW、WAR 三种数据冒险

掌握了冒险检测的逻辑设计和实现

学会了通过暂停、前递等方式解决冒险问题

2. 实践能力的提升

仿真和调试：

学会了使用仿真工具进行波形分析

掌握了硬件调试的方法和技巧

提高了问题定位和解决的能力

系统集成：

学会了将多个模块集成为完整的系统

掌握了接口设计和信号连接的方法

理解了系统级设计的重要性

二、个人感想

本课程是我学习生涯中非常重要的一门课程，它不仅让我掌握了 CPU 设计的理论知识和实践技能，更重要的是培养了我的工程思维和创新能力。通过课程学习，我深刻理解了计算机体系结构的复杂性，也认识到了硬件设计的挑战和乐趣。在未来的学习和工作中，我将继续深化对计算机体系结构的理解，不断提升自己的技术能力和工程素养。同时，我也希望能够在相关领域做出自己的贡献，为计算机技术的发展尽一份力量。最后，我要感谢所有任课教师的辛勤付出，感谢同学们的帮助和支持，感谢学校提供的良好学习环境。这段学习经历将是我人生中宝贵的财富，也将激励我在未来的道路上继续前进。