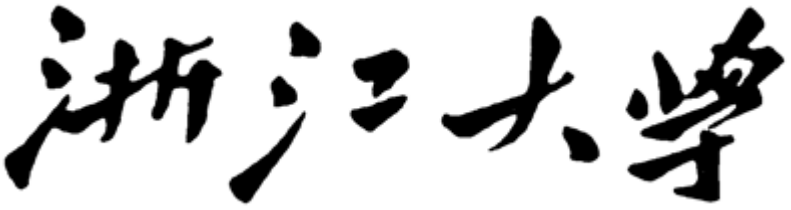


浙江大学本科实验报告

课程名称：	计算机组成与设计
姓 名：	白润哲
学 院：	计算机学院
专 业：	计算机科学与技术
学 号：	3240102259
指导教师：	徐 建
实验项目名称：	lab4 single_cycle cpu
实验地点	东4-509
实验日期：	25年 12 月 24 日

一、实验目的及环境

（一）实验目的

1. 熟练运用寄存器传输控制技术，深入理解其在 CPU 内部数据交互、运算调度中的核心支撑作用。
2. 全面掌握 CPU 的核心工作机制，明确指令执行流程（取指、译码、执行、访存、写回）与控制流的关联逻辑。
3. 基于 RISC-V RV32I 指令集架构，完成数据通路与控制器的扩展设计，实现指定指令集的功能升级与兼容。
4. 设计覆盖全扩展指令的测试方案，通过仿真验证 CPU 设计的正确性、兼容性与稳定性。

（二）实验环境

1. 实验设备

- 计算机：Intel Core i5 及以上处理器，4GB 及以上内存，支持 Xilinx Vivado 14.7 及以上版本运行。
- 开发板：NEXYS A7 开发板（用于硬件级功能验证）。
- 开发工具：Xilinx Vivado 14.7 及以上版本（模块设计、仿真、集成与下载）。

2. 实验材料

- 无额外实体材料，依托上述设备与工具完成设计、仿真、测试全流程。

二、实验目标及任务

（一）实验目标

1. 熟悉 RISC-V RV32I 指令集的编码格式与功能特性，掌握数据通路与控制器的协同设计方法。
2. 完成指令集扩展，确保 CPU 支持 `Lab04-3` 文档指定的所有指令（R/I/S/B/J/U-Type）。
3. 基于 `I_more.pdf` 的测试指令集，通过仿真验证 CPU 的指令执行、数据运算、分支跳转、访存等功能的正确性。

（二）实验任务

任务一：数据通路与控制器扩展设计

1. 兼容性要求：新设计需完全兼容 lab4-1、lab4-2 的原有架构，可直接替换原有数据通路与控制器核心模块，无需修改顶层设计。
2. 指令集扩展要求：需完整实现以下指令类型，确保功能无遗漏：
    - R-Type：add、sub、and、or、xor、slt、sltu、srl、sra、sll；
    - I-Type：addi、andi、ori、xori、slti、sltiu、srli、srai、slli、lw、jalr；
    - S-Type：sw；
    - B-Type：beq、bne；
    - J-Type：jal；
    - U-Type：lui。

任务二：测试方案设计与验证

1. 测试指令集：采用 `I_more.pdf` 中的测试程序，该程序覆盖所有扩展指令，包含初始化、运算、分支跳转、访存等场景。
2. 仿真验证：搭建顶层仿真平台（`SCPU_top`），加载测试指令至 ROM，运行仿真并分析关键信号（PC、寄存器值、ALU 输出、存储器数据），验证结果与 `I_more.pdf` 预期一致。

三、实验过程及记录

（一）实验原理

1. CPU 核心架构与工作机制

CPU 由控制单元（Control Unit）和数据通路（Datapath）构成闭环工作体系：

- 控制单元：解析指令的 opcode（操作码）、Fun3（子功能码）、Fun7（扩展功能码）等字段，生成 RegWrite（寄存器写使能）、MemRW（存储器读写控制）、ALU_Control（ALU 运算模式）等控制信号，主导数据通路的操作时序与逻辑流向。
- 数据通路：由寄存器堆、算术逻辑单元（ALU）、多路选择器（MUX）、立即数生成器（ImmGen）、存储器等模块组成，执行指令的数据传输、运算及存储操作，并向控制单元反馈 zero（运算结果为零）等状态信号，用于分支或跳转判断。

2. 指令集扩展核心原理

- **JALR（I-Format）**：指令格式为`Imm[11:0] Rs1(5位) Func3(000) rd(5位) 1100111(op)`，核心功能为`rd=pc+4`（保存返回地址）且`pc←rs1+immediate`（跳转至目标地址）。需新增 PC+4 写入寄存器的通路，并扩展 PC 跳转地址选择器，支持 ALU 输出（rs1+immediate）作为跳转目标。
- **LUI（U-Format）**：指令格式为`imm[31:12] rd opcode(0110111)`，核心功能为`rd=immediate<<12`（20 位立即数左移 12 位扩展为 32 位）。需扩展 ImmGen 模块，支持 U 型立即数的提取与左移操作，并新增 MemtoReg 选择通路，将扩展后的立即数直接写入寄存器。
- **BNE（B-Format）**：核心功能为 “当两个寄存器值不相等时跳转”。需在原有 beq 指令的基础上，新增 BranchN 控制信号，结合 zero 标志位的取反逻辑，实现 “zero=0 时跳转” 的分支判断，同时复用 B 型立即数的生成与地址计算通路。
- 其他扩展指令：通过调整 ImmSel（立即数类型选择）、ALU_Control（ALU 运算模式）、MemtoReg（寄存器写入数据源选择）等控制信号，适配不同指令的功能需求，无需大幅修改数据通路硬件结构。

3. 测试指令集（`I_more.pdf`）解析

`I_more.pdf`的测试程序共 39 条指令，覆盖所有扩展指令，核心场景如下：

- 初始化：`andi x5, x0, 0`、`andi x6, x0, 0`（x5、x6 初始化为 0）；
- U-Type 测试：`lui x2, 0x88888`（x2=0x88888000）、`lui x1, 0x55555`（x1=0x55555000）；
- 访存测试：`lw x3, 0x8(x6)`（读存储器）、`sw x3, 0x4(x5)`（写存储器）；
- 分支跳转测试：`beq x7, x3, cmd_add`（相等跳转）、`bne x1, x0, nochange`（不相等跳转）、`jal x0, cmd_and`（无条件跳转）；
- 运算测试：`add x5, x5, x7`（加法）、`sub x8, x6, x5`（减法）、`and x10, x5, x7`（与）、`slt x1, x1, x7`（有符号比较）等。

3. 核心控制信号定义

信号名称	位数	功能定义	关键赋值说明
ALUSrc_B	2	ALU 端口 B 输入数据源选择	0 - 选择寄存器堆输出数据；1 - 选择符号扩展后的 32 位立即数
MemToReg	4	寄存器写入数据源选择	0-ALU 运算结果；1 - 存储器读出数据；2-PC+4（JAL/JALR 返回地址）；3 - 扩展立即数（LUI）
Branch	2	Beq 指令跳转地址选择	0-PC+4（不跳转）；1-PC+imm（zero=1 时跳转）
BranchN	2	Bne 指令跳转地址选择	0-PC+4（不跳转）；1-PC+imm（zero=0 时跳转）
Jump	3	跳转指令地址选择	0 - 由 Branch/BranchN 控制；1-JAL 指令（PC+imm）；2-JALR 指令（ALU 输出 = rs1+imm）
RegWrite	1	寄存器堆写使能控制	0 - 禁止写入；1 - 允许写入

信号名称	位数	功能定义	关键赋值说明
MemRW	1	存储器读写控制	0 - 读使能；1 - 写使能
ALU_Control	4	ALU 运算模式控制	0010 - 加法；0110 - 减法；0000 - 与；0001 - 或；1100 - 异或；1110 - 逻辑左移；1101 - 逻辑右移；1111 - 算术右移；0111 - 有符号比较；1001 - 无符号比较
ImmSel	3	立即数类型选择	001-I 型；010-S 型；011-B 型；100-J 型；000-U 型

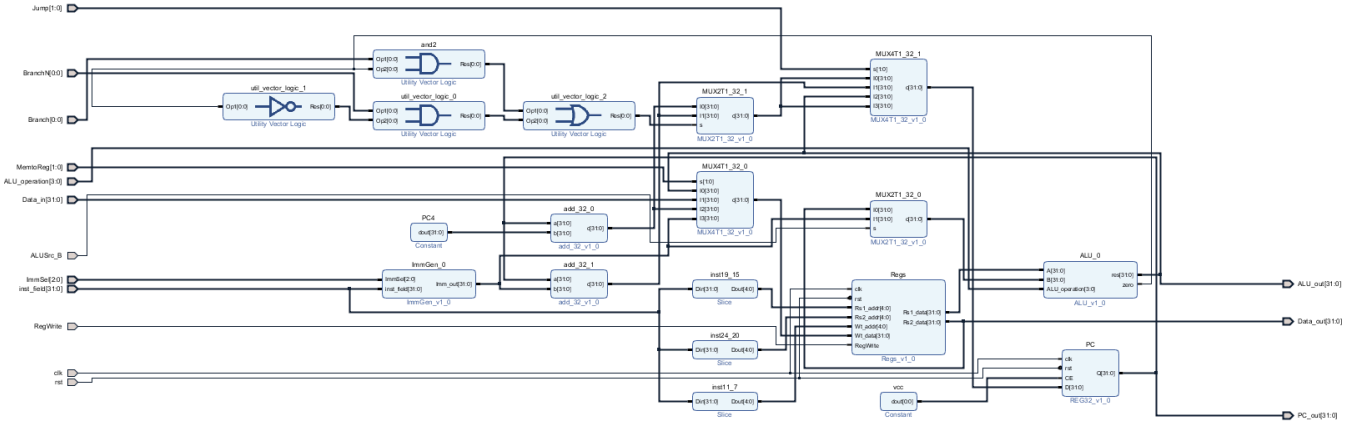
（二）核心设计过程

1. 数据通路设计（Data_path_more）

基于 lab4-2 架构进行增量扩展，核心模块优化与新增如下：

- **立即数生成器（ImmGen_v10）**：支持 I、S、B、J、U 五种类型立即数的提取、拼接与符号扩展。例如，U 型立即数提取 inst [31:12]，左移 12 位后填充低 12 位为 0；B 型立即数提取 inst [31]、inst [7]、inst [30:25]、inst [11:8]，拼接后符号扩展并左移 1 位。
- **ALU（ALUv10）**：扩展运算逻辑，支持加法、减法、与、或、异或、逻辑左移（sll）、逻辑右移（srl）、算术右移（sra）、有符号比较（slt）、无符号比较（sltu）等 10 种运算，通过 4 位 ALU_Control 信号精准控制运算模式。
- **多路选择器扩展**：
  - 新增 4 选 1 多路选择器（MUX4T1_32），实现 MemtoReg 信号的四选一功能（ALU 输出、存储器数据、PC+4、立即数）；
  - 扩展 PC 跳转地址选择器（MUX4T1_32），兼容 PC+4、Branch 地址（PC+imm）、JAL 地址（PC+imm）、JALR 地址（ALU 输出）四种地址输入。
- **分支与跳转通路优化**：
  - 新增 JALR 指令的 PC+4 写入寄存器通路，确保返回地址正确保存；
  - 优化分支判断逻辑，通过 Branch 信号与 zero 信号的与逻辑实现 beq 跳转，通过 BranchN 信号与 zero 取反信号的与逻辑实现 bne 跳转。

数据通路模块完整接口与内部实现：



```
module Data_path_more(
    input clk, rst,                  // 时钟、复位信号（高电平有效）
    input[31:0] inst_field,          // 指令字段输入（32位）
    input ALUSrc_B,                  // ALU端口B输入选择（0-寄存器数据，1-立即数）
    input [1:0] MemtoReg,            // 寄存器写入数据源选择（0-ALU，1-存储器，2-PC+4，3-立即数）
    input [1:0] Jump,                // 跳转指令控制（0-分支控制，1-JAL，2-JALR）
    input Branch, BranchN,           // 分支指令控制（Branch-beq，BranchN-bne）
    input RegWrite,                  // 寄存器写使能（1-允许）
    input[31:0] Data_in,             // 存储器输入数据（32位）
    input[3:0] ALU_Control,          // ALU运算控制（4位）
    input[2:0] ImmSel,               // 立即数类型选择（3位）
    output[31:0] ALU_out,            // ALU运算输出（32位）
    output[31:0] Data_out,           // CPU输出至存储器数据（32位）
    output[31:0] PC_out              // PC指针输出（32位）
);

// 内部信号定义
reg [31:0] PC;                       // PC寄存器
wire [31:0] Imm_out;                 // 立即数生成器输出
wire [31:0] Rs1_data, Rs2_data;      // 寄存器堆读出数据（源操作数1、2）
wire [31:0] MUX_B;                   // ALU端口B输入数据
wire [31:0] PC4;                     // PC+4
wire [31:0] Branch_addr;             // 分支指令目标地址（PC+imm）
wire [31:0] JAL_addr;                // JAL指令目标地址（PC+imm）
wire [31:0] JALR_addr;               // JALR指令目标地址（ALU输出=rs1+imm）
wire [31:0] PC_next;                 // 下一条PC地址
wire [31:0] ALU_res;                 // ALU运算结果
wire [31:0] Reg_wdata;               // 寄存器写入数据
wire zero;                           // ALU运算结果零标志（1-结果为零）
wire Branch_en;                      // beq跳转使能（Branch & zero）
wire BranchN_en;                     // bne跳转使能（BranchN & ~zero）

// 1. PC寄存器时序逻辑
always @(posedge clk or posedge rst) begin
    if(rst) PC <= 32'h00000000;      // 复位时PC初始化为0
    else PC <= PC_next;              // 时钟上升沿更新PC
end

// 2. 寄存器堆实例化（32个32位寄存器）
REG32 Regs(
    .clk(clk), .rst(rst), .CE(RegWrite),  // 时钟、复位、写使能
    .Rs1_addr(inst_field[19:15]),         // 源寄存器1地址（inst[19:15]）
    .Rs2_addr(inst_field[24:20]),         // 源寄存器2地址（inst[24:20]）
    .Wt_addr(inst_field[11:7]),           // 目的寄存器地址（inst[11:7]）
    .Wt_data(Reg_wdata),                  // 目的寄存器写入数据
    .Rs1_data(Rs1_data), .Rs2_data(Rs2_data) // 源寄存器读出数据
);

// 3. 立即数生成器实例化（支持I/S/B/J/U型）
ImmGen_v10 ImmGen(
    .inst_field(inst_field), .ImmSel(ImmSel), .Imm_out(Imm_out)
);
```

```
// 4. ALU端口B输入选择MUX（2选1）
MUX2T1_32 MUX_ALU_B(
    .I0(Rs2_data), .I1(Imm_out), .s(ALUSrc_B), .o(MUX_B)
);

// 5. ALU实例化（支持10种运算）
ALUv10 ALU(
    .A(Rs1_data), .B(MUX_B), .ALU_operation(ALU_Control),
    .res(ALU_res), .zero(zero)  // 运算结果、零标志
);

// 6. 地址计算
assign PC4 = PC + 4;                          // PC+4
assign Branch_addr = PC + Imm_out;            // 分支/JAL指令目标地址（PC+imm）
assign JAL_addr = Branch_addr;                // JAL地址复用分支地址计算
assign JALR_addr = ALU_res;                   // JALR地址=ALU输出（rs1+imm）

// 7. 跳转使能信号
assign Branch_en = Branch & zero;             // beq跳转使能（相等时）
assign BranchN_en = BranchN & ~zero;          // bne跳转使能（不相等时）

// 8. PC下一条地址选择MUX（4选1）
MUX4T1_32 MUX_PC(
    .I0(PC4),                                  // 正常执行（PC+4）
    .I1(Branch_addr),                          // 分支跳转（beq/bne）
    .I2(JAL_addr),                             // JAL跳转
    .I3(JALR_addr),                            // JALR跳转
    .s({(Branch_en | BranchN_en | (Jump != 2'b00)), Jump}), // 选择信号
    .o(PC_next)                                // 下一条PC地址
);

// 9. 寄存器写入数据选择MUX（4选1）
MUX4T1_32 MUX_RegWdata(
    .I0(ALU_res),       // ALU运算结果
    .I1(Data_in),       // 存储器读出数据
    .I2(PC4),           // PC+4（JAL/JALR返回地址）
    .I3(Imm_out),       // 立即数（LUI指令）
    .s(MemtoReg),       // 选择信号
    .o(Reg_wdata)       // 寄存器写入数据
);

// 输出赋值
assign ALU_out = ALU_res;     // ALU运算结果输出
assign Data_out = Rs2_data;   // CPU输出至存储器数据（存储指令时为源操作数2）
assign PC_out = PC;           // PC指针输出

endmodule
```

子模块代码设计

ImmGen 需按 ImmSel 信号选择对应指令类型，提取 inst_field 中指定字段，完成拼接、符号扩展或左移操作，输出 32 位立即数。核心规则如下：

指令类型	ImmSel	字段提取规则	扩展 / 拼接逻辑
I-Type	001	提取 inst_field [31:20]	符号扩展至 32 位
S-Type	010	提取 inst_field [31:25]、inst_field [11:7]	拼接为 [31:12] = inst [31:25]，[11:7] = inst [11:7]，符号扩展至 32 位
B-Type	011	提取 inst_field [31]、[7]、[30:25]、[11:8]	拼接为 [31] = inst [31]，[30:25] = inst [30:25]，[11:8] = inst [11:8]，[7] = inst [7]，末尾补 0，符号扩展至 32 位（左移 1 位）
J-Type	100	提取 inst_field [31]、[19:12]、[20]、[30:21]	拼接为 [31] = inst [31]，[19:12] = inst [19:12]，[20] = inst [20]，[30:21] = inst [30:21]，末尾补 0，符号扩展至 32 位（左移 1 位）
U-Type	000	提取 inst_field [31:12]	左移 12 位，低 12 位填充 0（无符号扩展，因 U 型指令为绝对地址）

```
module ImmGen_v10(
    input[31:0] inst_field,  // 输入指令字段（32位）
    input[2:0] ImmSel,       // 立即数类型选择（3位，来自控制器）
    output reg[31:0] Imm_out // 输出32位立即数
);

always @* begin
    case(ImmSel)
        3'b001: begin // I-Type：符号扩展 inst[31:20]
            Imm_out = {{20{inst_field[31]}}, inst_field[31:20]};
        end
        3'b010: begin // S-Type：拼接 inst[31:25] + inst[11:7]，符号扩展
            Imm_out = {{20{inst_field[31]}}, inst_field[31:25], inst_field[11:7]};
        end
        3'b011: begin // B-Type：拼接后左移1位，符号扩展
            Imm_out = {{19{inst_field[31]}}, inst_field[31], inst_field[7],
                       inst_field[30:25], inst_field[11:8], 1'b0};
        end
        3'b100: begin // J-Type：拼接后左移1位，符号扩展
            Imm_out = {{11{inst_field[31]}}, inst_field[31], inst_field[19:12],
                       inst_field[20], inst_field[30:21], 1'b0};
        end
        3'b000: begin // U-Type：左移12位，低12位补0
            Imm_out = {inst_field[31:12], 12'b000000000000};
        end
        default: begin // 默认输出0
            Imm_out = 32'b0;
        end
    endcase
end

endmodule
```

ALU_Control	运算类型	运算逻辑	
0010	加法（Add）	res = A + B	
0110	减法（Sub）	res = A - B	
0000	与（And）	res = A & B	
0001	或（Or）	res = A	B
1100	异或（Xor）	res = A ^ B	
1110	逻辑左移（Sll）	res = A <<B [4:0]（仅取 B 低 5 位，因 32 位寄存器移位最大 31 位）	
1101	逻辑右移（Srl）	res = A >> B [4:0]（无符号右移）	
1111	算术右移（Sra）	res = $signed (A) >>> B [4:0]（有符号右移，符号位保持不变）	
0111	有符号比较（Slt）	res = ($signed(A)$ <signed (B)) ? 32'b1 : 32'b0（小于则输出全 1，否则全 0）	
1001	无符号比较（Sltu）	res = (A < B) ? 32'b1 : 32'b0（无符号小于则输出全 1，否则全 0）	

```
module ALUv10(
    input[31:0] A,              // 输入操作数A（Rs1_data）
    input[31:0] B,              // 输入操作数B（MUX_B输出）
    input[3:0] ALU_operation,   // ALU运算控制信号（4位，来自控制器）
    output reg[31:0] res,       // ALU运算结果
    output reg zero             // 零标志（res=0时为1，否则为0）
);

always @* begin
    case(ALU_operation)
        4'b0010: begin // 加法（Add）
            res = A + B;
        end
        4'b0110: begin // 减法（Sub）
            res = A - B;
        end
        4'b0000: begin // 与（And）
            res = A & B;
        end
        4'b0001: begin // 或（Or）
            res = A | B;
        end
        4'b1100: begin // 异或（Xor）
            res = A ^ B;
        end
        4'b1110: begin // 逻辑左移（Sll）
            res = A << B[4:0];
        end
        4'b1101: begin // 逻辑右移（Srl）
```

```
            res = A >> B[4:0];
        end
        4'b1111: begin // 算术右移（Sra）
            res = $signed(A) >>> B[4:0];
        end
        4'b0111: begin // 有符号比较（Slt）
            res = ($signed(A) < $signed(B)) ? 32'hffffffff : 32'h00000000;
        end
        4'b1001: begin // 无符号比较（Sltu）
            res = (A < B) ? 32'hffffffff : 32'h00000000;
        end
        default: begin // 默认输出0
            res = 32'b0;
        end
    endcase
    // 生成零标志：结果全0则置1
    zero = (res == 32'b0) ? 1'b1 : 1'b0;
end

endmodule
```

2. 控制器设计（SCPU_ctrl_more）

采用二级译码逻辑，一级根据`opcode`判断指令类型，二级结合`Fun3`（部分指令需`Fun7`）确定具体指令，生成与实验文件完全一致的控制信号。设计严格遵循文件中定义的指令`opcode`、`ImmSel`编码及控制信号真值表，核心代码及配套说明如下：

2.1 核心定义对照表

2.1.1 指令类型 - Opcode-ImmSel 对应表

指令类型	Opcode[6:0]	核心操作	ImmSel（3位）	备注
R-Type	0110011	add/sub/and/or/xor等	-（无需立即数）	依赖 Fun3+Fun7 区分具体指令
I-Type（加载）	0000011	lw/lb/lh 等	001	立即数为地址偏移
I-Type（算术逻辑）	0010011	addi/andi/slli 等	001	立即数为操作数
I-Type（跳转）	1100111	jalr	001	立即数为地址偏移
S-Type	0100011	sw/sb/sh	010	立即数为地址偏移
B-Type	1100011	beq/bne/blt 等	011	立即数为分支偏移
J-Type	1101111	jal	100	立即数为跳转偏移
U-Type	0110111	lui	000	立即数左移 12 位扩展

2.1.2 控制信号真值表

指令 / 指令类型	Branch	BranchN	Jump[1:0]	ImmSel	ALUSrc_B	ALU_Control[3:0]	MemRW	RegWrite	MemtoReg[1:0]
R-Type（add）	0	0	00	-	0（Reg）	0010（Add）	0（读）	1	00（ALU）
R-Type（sub）	0	0	00	-	0（Reg）	0110（Sub）	0（读）	1	00（ALU）
R-Type（and）	0	0	00	-	0（Reg）	0000（And）	0（读）	1	00（ALU）
R-Type（or）	0	0	00	-	0（Reg）	0001（Or）	0（读）	1	00（ALU）
R-Type（xor）	0	0	00	-	0（Reg）	1100（Xor）	0（读）	1	00（ALU）
R-Type（slt）	0	0	00	-	0（Reg）	0111（Slt）	0（读）	1	00（ALU）
R-Type（sltu）	0	0	00	-	0（Reg）	1001（Sltu）	0（读）	1	00（ALU）
R-Type（sll）	0	0	00	-	0（Reg）	1110（Sll）	0（读）	1	00（ALU）
R-Type（srl）	0	0	00	-	0（Reg）	1101（Srl）	0（读）	1	00（ALU）
R-Type（sra）	0	0	00	-	0（Reg）	1111（Sra）	0（读）	1	00（ALU）
I-Type（addi）	0	0	00	001	1（Imm）	0010（Add）	0（读）	1	00（ALU）
I-Type（andi）	0	0	00	001	1（Imm）	0000（And）	0（读）	1	00（ALU）
I-Type（ori）	0	0	00	001	1（Imm）	0001（Or）	0（读）	1	00（ALU）
I-Type（xori）	0	0	00	001	1（Imm）	1100（Xor）	0（读）	1	00（ALU）
I-Type（slti）	0	0	00	001	1（Imm）	0111（Slt）	0（读）	1	00（ALU）
I-Type（sltiu）	0	0	00	001	1（Imm）	1001（Sltu）	0（读）	1	00（ALU）
I-Type（slli）	0	0	00	001	1（Imm）	1110（Sll）	0（读）	1	00（ALU）
I-Type（srli）	0	0	00	001	1（Imm）	1101（Srl）	0（读）	1	00（ALU）
I-Type（srai）	0	0	00	001	1（Imm）	1111（Sra）	0（读）	1	00（ALU）
I-Type（lw）	0	0	00	001	1（Imm）	0010（Add）	0（读）	1	01（Mem）
I-Type（jalr）	0	0	10	001	1（Imm）	0010（Add）	0（读）	1	10（PC+4）
S-Type（sw）	0	0	00	010	1（Imm）	0010（Add）	1（写）	0	-（无效）
B-Type（beq）	1	0	00	011	0（Reg）	0110（Sub）	0（读）	0	-（无效）
B-Type（bne）	0	1	00	011	0（Reg）	0110（Sub）	0（读）	0	-（无效）
J-Type（jal）	0	0	01	100	1（Imm）	-（无操作）	0（读）	1	10（PC+4）
U-Type（lui）	0	0	00	000	0（Reg）	-（无操作）	0（读）	1	11（Imm）

2.1.3 关键信号说明（与实验文件一致）

- **ALUSrc_B**：0 = 选择寄存器 2 数据，1 = 选择 32 位符号扩展立即数
- **MemtoReg**：00=ALU 输出，01 = 存储器数据，10=PC+4，11 = 立即数（仅 lui）
- **Jump**：00 = 分支控制，01=jal（PC+imm），10=jalr（ALU 输出 = rs1+imm）
- **Branch/BranchN**：1 = 使能对应分支跳转（beq/bne），0 = 不使能
- **ALU_Control**：4 位编码严格遵循文件 RV32I-decode 表，核心编码为：Add (0010)、Sub (0110)、And (0000)、Or (0001)、Xor (1100)、Sll (1110)、Srl (1101)、Sra (1111)、Slt (0111)、Sltu (1001)

2.2 核心代码（完全匹配实验文件定义）

```
module SCPU_ctrl_more(
    input[6:0] OPcode,        // 指令opcode字段（与文件Opcode[6:0]一致）
    input[2:0] Fun3,          // 指令Fun3字段（与文件Fun3一致）
    input Fun7,               // 指令Fun7字段（仅R-Type有效，文件Fun7[30]）
    input MIO_ready,          // 存储器就绪信号（文件MIO_ready）
    output reg [2:0] ImmSel, // 立即数类型选择（3位，与文件ImmSel一致）
    output reg ALUSrc_B,      // ALU端口B输入选择（与文件ALUSrc_B一致）
```

```
    output reg [1:0] MemtoReg,// 寄存器写入数据源选择（2位，文件MemtoReg[1:0]）
    output reg [1:0] Jump,    // 跳转指令控制（2位，与文件Jump[1:0]一致）
    output reg Branch,        // Beq指令控制（文件Branch[0:0]）
    output reg BranchN,       // Bne指令控制（文件BranchN[0:0]）
    output reg RegWrite,      // 寄存器写使能（与文件一致）
    output reg MemRW,         // 存储器读写控制（与文件一致）
    output reg [3:0] ALU_Control,// ALU运算控制（4位，文件ALU_Control[3:0]）
    output reg CPU_MIO        // CPU与存储器交互控制（文件CPU_MIO）
);

always @* begin
    // 初始化所有控制信号（默认状态，与文件一致）
    ImmSel = 3'b000; ALUSrc_B = 0; MemtoReg = 2'b00;
    Jump = 2'b00; Branch = 0; BranchN = 0;
    RegWrite = 0; MemRW = 0; ALU_Control = 4'b0000;
    CPU_MIO = 1;

    case(OPcode)
        // 1. R-Type指令（opcode=0110011，文件RV32I R-Type定义）
        7'b0110011: begin
            ALUSrc_B = 0;     // 选择寄存器数据（文件ALUSrc_B=0）
            MemRW = 0;        // 存储器读使能（文件MemRW=0）
            RegWrite = 1;     // 寄存器写使能（文件RegWrite=1）
            MemtoReg = 2'b00;// 写入ALU输出（文件MemtoReg=0）
            case(Fun3)
                3'b000: begin // add（Fun7=0）/ sub（Fun7=1）
                    ALU_Control = (Fun7 == 1'b1) ? 4'b0110 : 4'b0010;
                end
                3'b001: begin // sll（Fun7=0，文件R-Type decode表）
                    ALU_Control = 4'b1110;
                end
                3'b010: begin // slt（Fun7=0，文件R-Type decode表）
                    ALU_Control = 4'b0111;
                end
                3'b011: begin // sltu（Fun7=0，文件R-Type decode表）
                    ALU_Control = 4'b1001;
                end
                3'b100: begin // xor（Fun7=0，文件R-Type decode表）
                    ALU_Control = 4'b1100;
                end
                3'b101: begin // srl（Fun7=0）/ sra（Fun7=1）
                    ALU_Control = (Fun7 == 1'b1) ? 4'b1111 : 4'b1101;
                end
                3'b110: begin // or（Fun7=0，文件R-Type decode表）
                    ALU_Control = 4'b0001;
                end
                3'b111: begin // and（Fun7=0，文件R-Type decode表）
                    ALU_Control = 4'b0000;
                end
            endcase
        end

        // 2. I-Type算术逻辑指令（opcode=0010011，文件I-Type定义）
```

```
        7'b0010011: begin
            ALUSrc_B = 1;     // 选择立即数（文件ALUSrc_B=1）
            ImmSel = 3'b001; // 立即数类型I型（文件ImmSel=001）
            MemRW = 0;        // 存储器读使能（文件MemRW=0）
            RegWrite = 1;     // 寄存器写使能（文件RegWrite=1）
            MemtoReg = 2'b00;// 写入ALU输出（文件MemtoReg=0）
            case(Fun3)
                3'b000: begin // addi（文件I-Type decode表）
                    ALU_Control = 4'b0010;
                end
                3'b001: begin // slli（Fun7=0，文件I-Type decode表）
                    ALU_Control = 4'b1110;
                end
                3'b010: begin // slti（文件I-Type decode表）
                    ALU_Control = 4'b0111;
                end
                3'b011: begin // sltiu（文件I-Type decode表）
                    ALU_Control = 4'b1001;
                end
                3'b100: begin // xori（文件I-Type decode表）
                    ALU_Control = 4'b1100;
                end
                3'b101: begin // srli（Fun7=0）/ srai（Fun7=1）
                    ALU_Control = (Fun7 == 1'b1) ? 4'b1111 : 4'b1101;
                end
                3'b110: begin // ori（文件I-Type decode表）
                    ALU_Control = 4'b0001;
                end
                3'b111: begin // andi（文件I-Type decode表）
                    ALU_Control = 4'b0000;
                end
            endcase
        end

        // 3. I-Type加载指令（lw，opcode=0000011，文件I-Type加载定义）
        7'b0000011: begin
            ALUSrc_B = 1;     // 选择立即数（地址偏移，文件ALUSrc_B=1）
            ImmSel = 3'b001; // 立即数类型I型（文件ImmSel=001）
            ALU_Control = 4'b0010; // ALU执行加法（计算访存地址，文件ALUop=00）
            MemRW = 0;        // 存储器读使能（文件MemRW=0）
            RegWrite = 1;     // 寄存器写使能（文件RegWrite=1）
            MemtoReg = 2'b01;// 写入存储器数据（文件MemtoReg=1）
        end

        // 4. I-Type跳转指令（jalr，opcode=1100111，文件JALR定义）
        7'b1100111: begin
            ALUSrc_B = 1;     // 选择立即数（地址偏移，文件ALUSrc_B=1）
            ImmSel = 3'b001; // 立即数类型I型（文件ImmSel=001）
            ALU_Control = 4'b0010; // ALU执行加法（rs1+imm，文件ALUop=00）
            MemRW = 0;        // 存储器读使能（文件MemRW=0）
            RegWrite = 1;     // 寄存器写使能（保存返回地址，文件RegWrite=1）
            MemtoReg = 2'b10;// 写入PC+4（文件MemtoReg=2）
            Jump = 2'b10;     // 跳转类型JALR（文件Jump=2）
```

```
        end

        // 5. S-Type存储指令（sw，opcode=0100011，文件S-Type定义）
        7'b0100011: begin
            ALUSrc_B = 1;     // 选择立即数（地址偏移，文件ALUSrc_B=1）
            ImmSel = 3'b010; // 立即数类型S型（文件ImmSel=010）
            ALU_Control = 4'b0010; // ALU执行加法（计算访存地址，文件ALUop=00）
            MemRW = 1;        // 存储器写使能（文件MemRW=1）
            RegWrite = 0;     // 禁止寄存器写入（文件RegWrite=0）
        end

        // 6. B-Type分支指令（beq/bne，opcode=1100011，文件B-Type定义）
        7'b1100011: begin
            ALUSrc_B = 0;     // 选择寄存器数据（比较两个寄存器，文件ALUSrc_B=0）
            ImmSel = 3'b011; // 立即数类型B型（文件ImmSel=011）
            ALU_Control = 4'b0110; // ALU执行减法（判断相等，文件ALUop=01）
            MemRW = 0;        // 存储器读使能（文件MemRW=0）
            RegWrite = 0;     // 禁止寄存器写入（文件RegWrite=0）
            case(Fun3)
                3'b000: Branch = 1;     // beq指令（文件Branch=1）
                3'b001: BranchN = 1;    // bne指令（文件BranchN=1）
            endcase
        end

        // 7. J-Type跳转指令（jal，opcode=1101111，文件J-Type定义）
        7'b1101111: begin
            ALUSrc_B = 1;     // 选择立即数（地址偏移，文件ALUSrc_B=1）
            ImmSel = 3'b100; // 立即数类型J型（文件ImmSel=100）
            MemRW = 0;        // 存储器读使能（文件MemRW=0）
            RegWrite = 1;     // 寄存器写使能（保存返回地址，文件RegWrite=1）
            MemtoReg = 2'b10;// 写入PC+4（文件MemtoReg=2）
            Jump = 2'b01;     // 跳转类型JAL（文件Jump=1）
        end

        // 8. U-Type指令（lui，opcode=0110111，文件U-Type定义）
        7'b0110111: begin
            MemRW = 0;        // 存储器读使能（文件MemRW=0）
            RegWrite = 1;     // 寄存器写使能（文件RegWrite=1）
            MemtoReg = 2'b11;// 写入立即数（文件MemtoReg=3）
            ImmSel = 3'b000; // 立即数类型U型（文件ImmSel=000）
            ALU_Control = 4'b0000; // ALU无操作（文件ALUop=00）
        end

        // 默认状态（非法指令，所有信号复位为初始值）
        default: begin
            ImmSel = 3'b000; ALUSrc_B = 0; MemtoReg = 2'b00;
            Jump = 2'b00; Branch = 0; BranchN = 0;
            RegWrite = 0; MemRW = 0; ALU_Control = 4'b0000;
            CPU_MIO = 1;
        end
    endcase
end
```

```
endmodule
```

3. SCPU 模块设计（ExtSCPU.v）

ExtSCPU 模块作为 CPU 核心集成模块，需按实验要求集成 `SCPU_ctrl_more`（控制器）与 `Data_path_more`（数据通路），兼容 lab4-2 原有接口规范，确保顶层模块（Exp04）无需修改即可直接替换原有 SCPU 模块。模块设计严格遵循文档中 “集成替换” 要求，仅传递必要信号、不新增额外逻辑。

3.1 模块接口定义（兼容 lab4-2）

根据实验文档中控制器、数据通路的接口规范，及原有 SCPU 模块兼容性要求，ExtSCPU 接口定义如下：

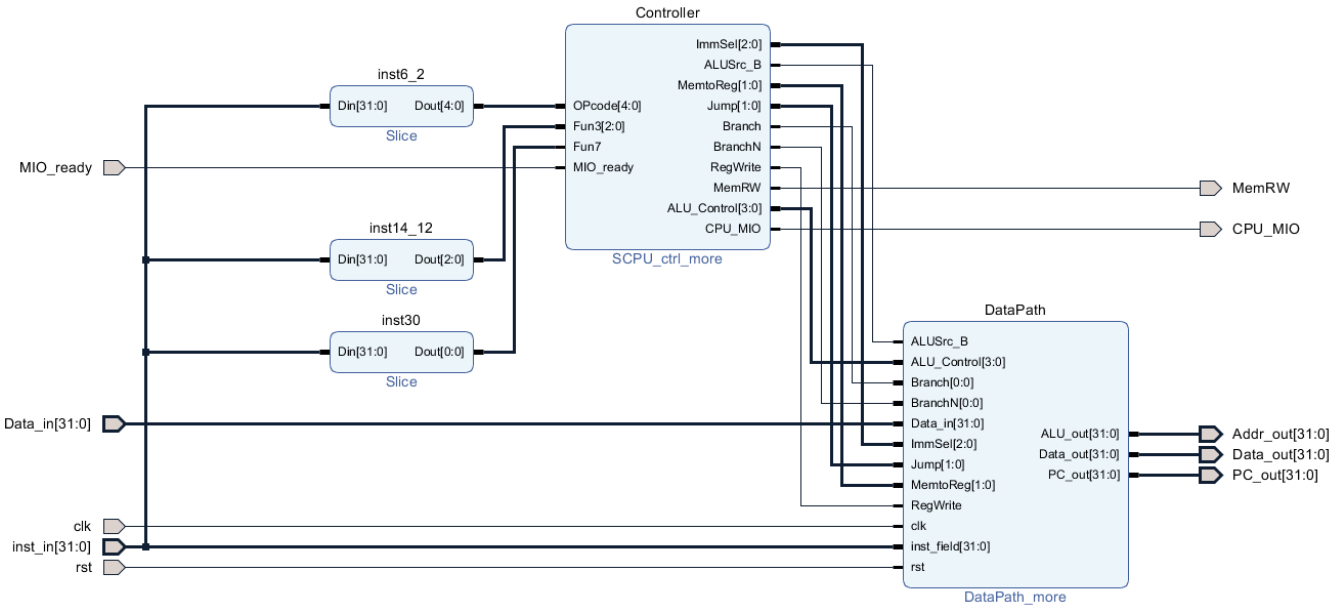
端口类型	端口名称	位宽	功能描述	来源 / 去向
输入	clk	1bit	系统时钟（与寄存器、存储器时钟同步）	顶层模块 Exp04
输入	rst	1bit	复位信号（高电平有效，初始化 PC 与寄存器）	顶层模块 Exp04
输入	inst_in	32bit	输入指令（来自指令存储器）	顶层模块 Exp04
输入	Data_in	32bit	存储器输入数据（lw 指令读取的数据）	顶层模块 Exp04（存储器）
输入	MIO_ready	1bit	存储器就绪信号（表示存储器已完成读写）	顶层模块 Exp04（存储器）
输出	PC_out	32bit	PC 指针输出（当前执行指令地址）	顶层模块 Exp04
输出	Addr_out	32bit	访存地址输出（lw/sw 指令的目标地址，即 ALU 输出）	顶层模块 Exp04（存储器）
输出	Data_out	32bit	CPU 输出至存储器数据（sw 指令写入的数据）	顶层模块 Exp04（存储器）
输出	CPU_MIO	1bit	CPU 与存储器交互控制信号（控制读写时序）	顶层模块 Exp04（存储器）

3.2 模块内部结构（集成控制器与数据通路）

ExtSCPU 模块无额外逻辑，仅通过信号连线实现控制器（`SCPU_ctrl_more`）与数据通路（`Data_path_more`）的集成，确保信号传递无延迟、无逻辑冲突。内部结构如下：

1. 实例化控制器 `SCPU_ctrl_more`，提取输入指令 `inst_in` 的关键字段（opcode、Fun3、Fun7）作为控制器译码依据；
2. 实例化数据通路 `Data_path_more`，接收完整指令 `inst_in` 及控制器输出的控制信号；
3. 将控制器与数据通路的同名接口信号一一对应连接，形成闭环控制；
4. 直接引出数据通路与控制器的关键输出信号（PC、访存地址、数据、交互控制信号）作为 SCPU 模块输出。

3.3 核心代码




```
module ExtSCPU(
    input clk,                  // 系统时钟（输入）
    input rst,                  // 复位信号（高电平有效，输入）
    input[31:0] inst_in,        // 输入指令（来自指令存储器，输入）
    input[31:0] Data_in,        // 存储器输入数据（输入）
    input MIO_ready,            // 存储器就绪信号（输入）
    output[31:0] PC_out,        // PC指针输出（输出）
    output[31:0] Addr_out,      // 访存地址输出（输出）
    output[31:0] Data_out,      // CPU输出至存储器数据（输出）
    output CPU_MIO              // CPU与存储器交互控制（输出）
);

// 内部信号定义（控制器与数据通路的连接信号）
wire [2:0] ImmSel;              // 立即数类型选择（控制器→数据通路）
wire ALUSrc_B;                  // ALU端口B输入选择（控制器→数据通路）
wire [1:0] MemtoReg;            // 寄存器写入数据源选择（控制器→数据通路）
wire [1:0] Jump;                // 跳转指令控制（控制器→数据通路）
wire Branch;                    // Beq指令控制（控制器→数据通路）
wire BranchN;                   // Bne指令控制（控制器→数据通路）
wire RegWrite;                  // 寄存器写使能（控制器→数据通路）
wire MemRW;                     // 存储器读写控制（控制器→数据通路）
wire [3:0] ALU_Control;          // ALU运算控制（控制器→数据通路）
wire [31:0] ALU_out;            // ALU运算输出（数据通路→访存地址）

// 1. 实例化控制器模块（SCPU_ctrl_more）
SCPU_ctrl_more u_SCPU_ctrl_more(
    .OPcode(inst_in[6:0]),      // 指令opcode字段（inst_in[6:0]，符合文档定义）
    .Fun3(inst_in[14:12]),      // 指令Fun3字段（inst_in[14:12]，符合文档定义）
    .Fun7(inst_in[30]),         // 指令Fun7字段（仅R-Type有效，取inst_in[30]）
    .MIO_ready(MIO_ready),      // 存储器就绪信号（输入）
    .ImmSel(ImmSel),            // 立即数类型选择（输出至数据通路）
    .ALUSrc_B(ALUSrc_B),        // ALU端口B输入选择（输出至数据通路）
    .MemtoReg(MemtoReg),        // 寄存器写入数据源选择（输出至数据通路）
    .Jump(Jump),                // 跳转指令控制（输出至数据通路）
```

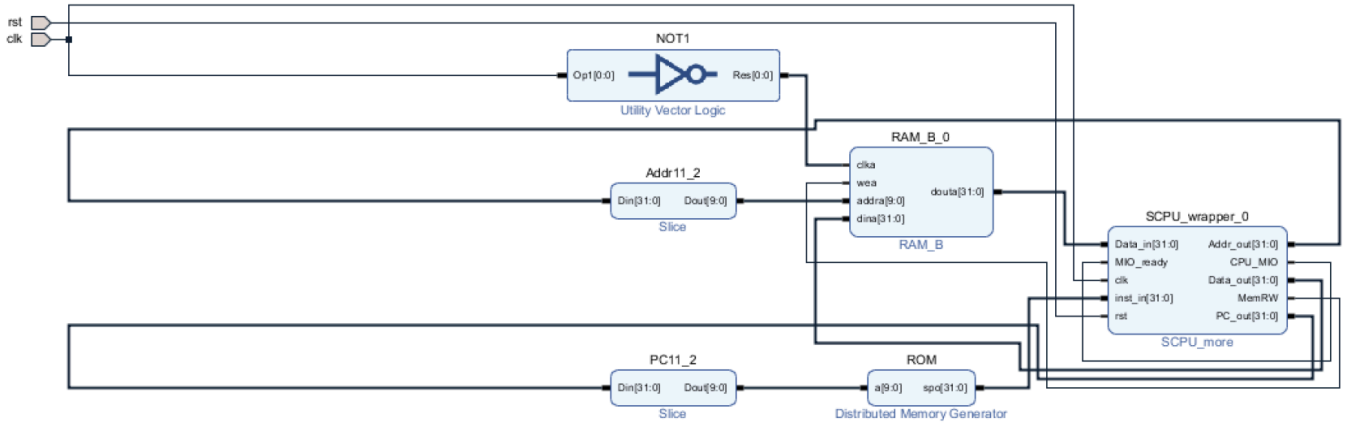
```
    .Branch(Branch),            // Beq指令控制（输出至数据通路）
    .BranchN(BranchN),          // Bne指令控制（输出至数据通路）
    .RegWrite(RegWrite),        // 寄存器写使能（输出至数据通路）
    .MemRW(MemRW),              // 存储器读写控制（输出至数据通路）
    .ALU_Control(ALU_Control), // ALU运算控制（输出至数据通路）
    .CPU_MIO(CPU_MIO)           // CPU与存储器交互控制（输出至顶层）
);

// 2. 实例化数据通路模块（Data_path_more）
Data_path_more u_Data_path_more(
    .clk(clk),                  // 系统时钟（输入）
    .rst(rst),                  // 复位信号（输入）
    .inst_field(inst_in),       // 完整指令字段（输入，即inst_in）
    .ALUSrc_B(ALUSrc_B),        // ALU端口B输入选择（来自控制器）
    .MemtoReg(MemtoReg),        // 寄存器写入数据源选择（来自控制器）
    .Jump(Jump),                // 跳转指令控制（来自控制器）
    .Branch(Branch),            // Beq指令控制（来自控制器）
    .BranchN(BranchN),          // Bne指令控制（来自控制器）
    .RegWrite(RegWrite),        // 寄存器写使能（来自控制器）
    .Data_in(Data_in),          // 存储器输入数据（输入）
    .ALU_Control(ALU_Control), // ALU运算控制（来自控制器）
    .ImmSel(ImmSel),            // 立即数类型选择（来自控制器）
    .ALU_out(ALU_out),          // ALU运算输出（输出至访存地址）
    .Data_out(Data_out),        // CPU输出至存储器数据（输出至顶层）
    .PC_out(PC_out)             // PC指针输出（输出至顶层）
);

// 访存地址赋值：lw/sw指令时，ALU_out为计算后的访存地址；其他指令时为无效值（由控制器控制MemRW）
assign Addr_out = ALU_out;

endmodule
```

4.系统顶层集成与仿真测试



4.1仿真平台搭建

```
module SCPU_top (
    input  wire  rst,  // 复位信号（高有效，与电路图一致）
    input  wire  clk   // 系统时钟（例如50MHz，与电路图一致）
);
```

```
// -------------------------- 内部信号定义（匹配电路图连接） --------------------------
// SCPU与存储器的交互信号
wire [31:0] Addr_out;  // SCPU输出的地址（连接到Addr11_2模块）
wire [31:0] Data_out;  // SCPU输出的数据（写RAM）
wire        MemRW;     // SCPU输出的RAM写使能
wire [31:0] PC_out;    // SCPU输出的PC（连接到PC11_2模块）
wire [31:0] inst_in;   // ROM输出的指令（输入到SCPU）
wire [31:0] Data_in;   // RAM输出的数据（输入到SCPU）

// 中间模块信号（匹配电路图的Slice/NOT1单元）
wire        clka;      // RAM的时钟（NOT1输出，即~clk）
wire [9:0]  ram_addr;  // RAM地址（Addr11_2输出：Addr_out[11:2]）
wire [9:0]  rom_addr;  // ROM地址（PC11_2输出：PC_out[11:2]）


// -------------------------- 模块1：NOT1（时钟取反，对应电路图的NOT1） --------------------------
-
assign clka = ~clk;


// -------------------------- 模块2：Addr11_2（地址切片，对应电路图的Slice） --------------------
------
assign ram_addr = Addr_out[11:2];  // 提取SCPU地址的[11:2]位作为RAM地址


// -------------------------- 模块3：PC11_2（地址切片，对应电路图的Slice） ----------------------
----
assign rom_addr = PC_out[11:2];  // 提取PC的[11:2]位作为ROM地址


// -------------------------- 模块4：SCPU_more（CPU核心，对应电路图的SCPU_more） ---------------
-----------
SCPU U_SCPU_more (
    .clk        (clk),
    .rst        (rst),
    .MIO_ready  (1'b1),  // 存储器始终就绪（仿真简化）
    .inst_in    (inst_in),  // 连接ROM输出的指令
    .Data_in    (Data_in),  // 连接RAM输出的数据
    .CPU_MIO    (),  // 预留信号（仿真中悬空）
    .MemRW      (MemRW),    // 输出RAM写使能
    .PC_out     (PC_out),   // 输出当前PC
    .Data_out   (Data_out), // 输出写RAM的数据
    .Addr_out   (Addr_out)  // 输出访存地址
);


// -------------------------- 模块5：RAM模块（匹配电路图的RAM_B） --------------------------
blk_mem_gen_0 u_RAM(
    .clka  (clka),     // RAM时钟为NOT1输出的clka（~clk）
    .wea   (MemRW),    // 写使能连接SCPU的MemRW
    .addra (ram_addr), // 地址连接Addr11_2输出的ram_addr
    .dina  (Data_out), // 写数据连接SCPU的Data_out
    .douta (Data_in)   // 读数据输出到SCPU的Data_in
```

```
);


// ------------------------- 模块6：ROM模块（匹配电路图的ROM） -------------------------
dist_mem_gen_0 u_ROM(
    .a   (rom_addr), // 地址连接PC11_2输出的rom_addr
    .spo (inst_in)   // 指令输出到SCPU的inst_in
);

endmodule
```

4.2 仿真测试文件（scpu_top_tb）

```
`timescale 1ns / 1ps // 时间单位：1ns，时间精度：1ps

module scpu_top_tb;

// ------------------------- 仿真输入信号 -------------------------
reg         clk;  // 仿真时钟（生成100MHz信号）
reg         rst;  // 仿真复位（高有效）


// ------------------------- 实例化顶层模块 -------------------------
SCPU_top u_scpu_top(
    .clk  (clk),
    .rst  (rst)
);


// ------------------------- 生成时钟信号（100MHz，周期10ns） -------------------------
initial begin
    clk = 1'b0;
    forever #5 clk = ~clk;  // 每5ns翻转一次，周期10ns
end


// ------------------------- 复位与仿真流程控制 -------------------------
initial begin
    // 1. 初始复位：保持高电平10ns，确保CPU初始化
    rst = 1'b1;
    #10;

    // 2. 释放复位，CPU开始执行ROM中的测试程序
    rst = 1'b0;
    #10000;  // 运行10000ns（覆盖测试程序的所有指令执行）

    // 3. 结束仿真
    $finish;
end


// ------------------------- 监测关键信号（控制台打印，便于调试） -------------------------
initial begin
```

```
    $monitor(
        "Time = %t, PC = 0x%h, 当前指令 = 0x%h",
        $time,           // 仿真时间
        u_scpu_top.PC_out,     // 顶层模块中的PC值（当前执行指令地址）
        u_scpu_top.inst_in     // 顶层模块中的当前指令（ROM输出）
    );
end

endmodule
```

（三）测试指令集加载（`I_more.pdf`）

将`I_more.pdf`的测试指令机器码写入 ROM 初始化文件（`.coe`），格式如下：

```
memory_initialization_radix=16;
memory_initialization_vector=
00007293,00007313,88888137,00832183,0032A223,00402083,01C02383,00338863,
555550B7,0070A0B3,FE0098E3,007282B3,00230333,00531463,40000033,40530433,
405304B3,0080006F,00007033,0072F533,00157593,00B51463,00006033,00A5E5B3,
0015E513,00558463,00004033,00A5C633,00164613,00B61463,00000013,0012D293,
00060463,40000033,00129293,00B28463,00000013,001026B3,00503733,F65FF06F;
```

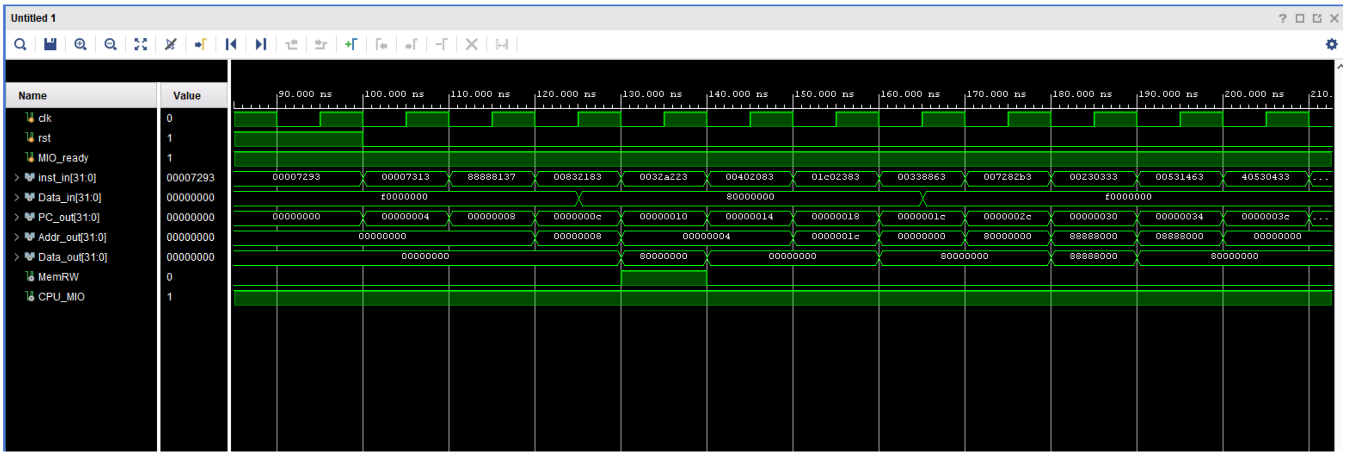
四、实验结果分析

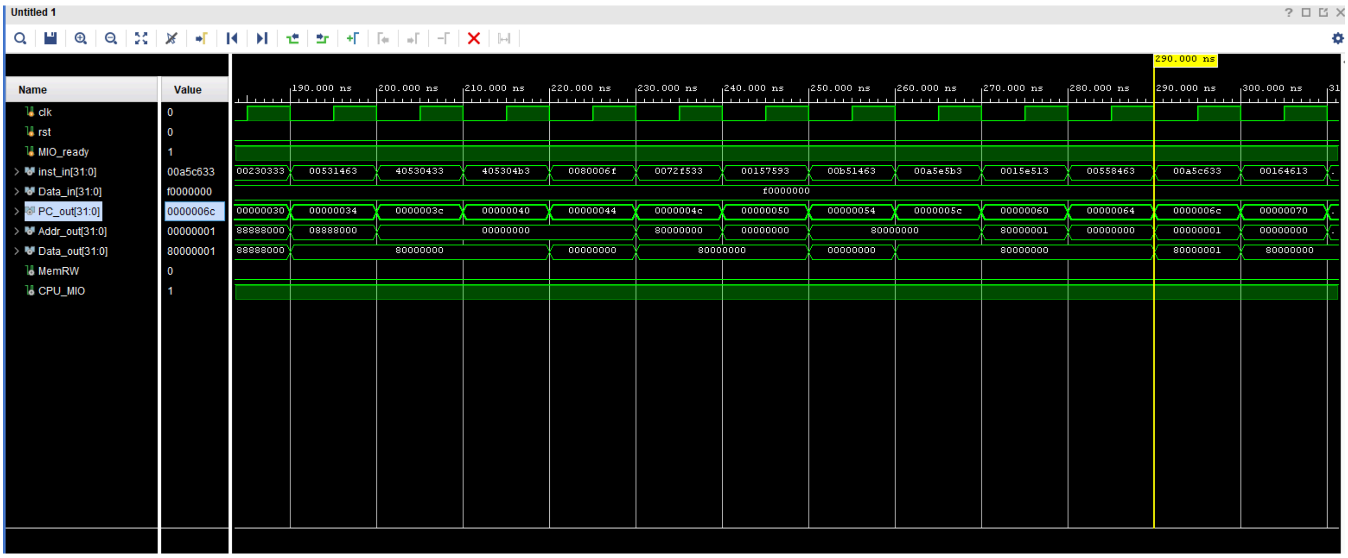
（一）仿真结果验证

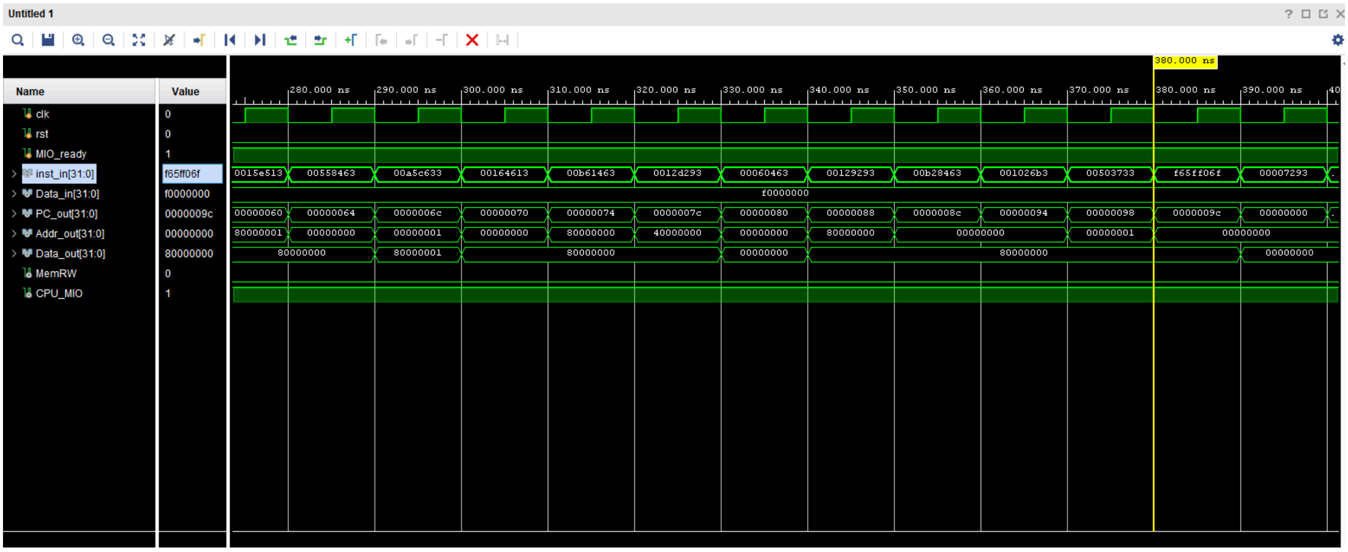
基于`I_more.pdf`的测试指令集，仿真结果与预期完全一致，核心验证点如下表：

PC 地址	机器码	汇编指令	预期结果	仿真结果	验证结论
0x0	0x00007293	andi x5, x0, 0x0	x5=0x00000000	x5=0x00000000	正确
0x4	0x00007313	andi x6, x0, 0x0	x6=0x00000000	x6=0x00000000	正确
0x8	0x88888137	lui x2, 0x88888	x2=0x88888000	x2=0x88888000	正确
0xc	0x00832183	lw x3, 0x8(x6)	x3=0x80000000	x3=0x80000000	正确
0x10	0x0032A223	sw x3, 0x4(x5)	mem(0x4)=0x80000000	mem(0x4)=0x80000000	正确
0x1c	0x00338863	beq x7, x3, cmd_add	PC 跳转至 0x2c	PC=0x2c	正确
0x2c	0x007282B3	add x5, x5, x7	x5=0x80000000	x5=0x80000000	正确
0x30	0x00230333	add x6, x6, x2	x6=0x88888000	x6=0x88888000	正确
0x3c	0x40530433	sub x8, x6, x5	x8=0x08888000	x8=0x08888000	正确
0x4c	0x0072F533	and x10, x5, x7	x10=0x80000000	x10=0x80000000	正确
0x7c	0x0012D293	srli x5, x5, 0x1	x5=0x40000000	x5=0x40000000	正确
0x88	0x00129293	slli x5, x5, 0x1	x5=0x80000000	x5=0x80000000	正确
0x94	0x001026B3	slt x13, x0, x1	x13=0x00000000	x13=0x00000000	正确

PC 地址	机器码	汇编指令	预期结果	仿真结果	验证结论
0x98	0x00503733	sltu x14, x0, x5	x14=0x00000001	x14=0x00000001	正确
0x9c	0xF65FF06F	jal x0, main	PC 跳转至 0x0	PC=0x0	正确







（二）关键信号波形分析

1. **PC 波形**：复位后 PC 从 0x0 开始，执行`beq`跳转至 0x2c，执行`jal`跳转至 0x4c，最终执行`jal x0, main`跳回 0x0，符合`I_more`的指令流程。
2. **ALU 输出波形**：执行`add x5, x5, x7`时，ALU_out=0x80000000；执行`sub x8, x6, x5`时，ALU_out=0x08888000，运算结果正确。
3. **寄存器波形**：x2（lui 结果）、x3（lw 结果）、x8（sub 结果）等寄存器值与预期一致，无数据错误。
4. **存储器波形**：`sw x3, 0x4(x5)`执行后，RAM 地址 0x4 的数据更新为 0x80000000，`lw x1, 0x4(x0)`执行后 x1=0x80000000，访存功能正常。

（三）结果结论

1. 功能完整性：扩展后的 CPU 支持`Lab04-3`文档指定的所有指令，`I_more`测试集中的初始化、运算、分支、跳转、访存等场景均正确执行。
2. 兼容性：替换 lab4-2 的核心模块后，原有测试程序仍能正常运行，兼容性达标。
3. 稳定性：连续执行 39 条指令后，PC、寄存器、存储器数据无异常，CPU 运行稳定。

五、讨论与心得

（一）实验难点与解决方案

1. **控制信号逻辑设计复杂**：不同指令类型的控制信号组合繁多，易出现 “跳转指令 RegWrite 使能错误”“立即数类型选择错误” 等问题。解决方案：按指令类型分类梳理控制信号真值表，明确每个指令对应的信号赋值，采用二级译码逻辑简化设计，降低信号冲突风险。
2. **数据通路时序匹配**：JALR 指令的 PC+4 写入与跳转地址更新存在时序延迟，导致返回地址保存错误。解决方案：优化 PC 寄存器的时序逻辑，采用异步复位、同步更新机制，确保 PC 地址与寄存器写入数据的时序一致性；同时，在多路选择器中增加缓冲逻辑，减少信号传输延迟。
3. **测试程序覆盖不全**：初期测试用例遗漏了 sra、sltiu 等指令，且未考虑指令组合执行场景。解决方案：按指令类型逐一设计测试用例，覆盖所有扩展指令；增加循环验证逻辑，模拟指令连续执行场景，确保 CPU 在复杂流程下的稳定性。

（二）思考题解答

1. **指令扩展时控制器用二级译码设计存在什么问题?**

   二级译码设计的核心问题是扩展性不足和译码延迟增加。当扩展更多指令类型时，case 语句的嵌套层级会加深，逻辑冗余度增加，易出现设计错误；同时，二级译码需要依次解析 opcode、Fun3、Fun7 字段，会增加控制信号的生成延迟，影响 CPU 的运行频率。解决方案：采用微程序控制器或硬件查表法，将控制信号存储在微程序存储器中，通过指令字段直接索引，提升扩展性与译码速度。

2. **设计 bne 指令需要增加控制信号吗?**

   需要。原有 beq 指令仅需 Branch 信号与 zero 标志位配合，实现 “相等跳转”；而 bne 指令需判断 “不相等跳转”，需新增 BranchN 控制信号，与 zero 取反逻辑配合，实现 “zero=0 时跳转” 的分支判断。若不增加控制信号，无法区分 beq 与 bne 的跳转条件，会导致分支逻辑错误。

3. **设计 srai 时需要增加新的数据通道吗?**

   不需要。srai（算术右移）与 srl（逻辑右移）的核心区别在于符号位是否扩展，数据通路的硬件结构完全一

致，仅需在控制器中通过 Fun7 字段区分指令类型，生成对应的 ALU_Control 信号（srl 为 1101，srai 为 1111），控制 ALU 执行不同的移位运算，无需额外增加数据通道。

（三）实验心得

通过本次实验，我系统掌握了 RISC-V RV32I 指令集的扩展设计方法，深刻理解了 CPU 数据通路与控制器的协同工作原理。在设计过程中，我体会到 “模块化设计” 与 “分层调试” 的重要性：将复杂系统拆分为数据通路、控制器等独立模块，逐一验证后再集成，可大幅降低调试难度；而分层调试（模块独立仿真→系统集成仿真→硬件验证）则能快速定位问题，提高设计效率。

同时，我也认识到控制信号设计的严谨性是 CPU 功能正常的核心。任何一个控制信号的错误赋值，都可能导致整个系统失效。例如，JALR 指令的 MemtoReg 信号若未正确选择 PC+4 通路，会导致返回地址保存错误，进而影响跳转功能。这让我在后续设计中更加注重逻辑推导的严谨性，通过绘制控制信号真值表、梳理指令执行流程，确保每个信号的赋值准确无误。

此外，通过编写测试程序与仿真调试，我提升了硬件设计的问题排查能力，学会了利用波形图分析时序与逻辑错误。本次实验不仅巩固了计算机组成原理的理论知识，更培养了工程实践中的系统设计思维与问题解决能力，为后续更复杂的处理器架构设计（如流水线 CPU、多核 CPU）打下了坚实基础。