浙江大学
ZHEJIANG UNIVERSITY


# 计算机组成 实验报告

| | |
|---|---|
| 姓　　名: | NoughtQ |
| 学　　号: | 1145141919810 |
| 专　　业: | 计算机科学与技术 |
| 课程名称: | 计算机组成 |
| 指导教师: | 赵莎 |
| 实验地点: | 东四 509 教室 |

2024 年 11 月 19 日

# 目录

# Lab4：单周期 CPU

## 一、 实验目的和要求

1. 复习寄存器传输控制技术
2. 掌握 CPU 的核心组成：数据通路与控制器
3. 设计数据通路的功能部件
4. 进一步了解计算机系统的基本结构
5. 熟练掌握 IP 核的使用方法
6. 学会测试方案和程序的设计

## 二、 实验内容和原理

### 2.1 单周期 CPU

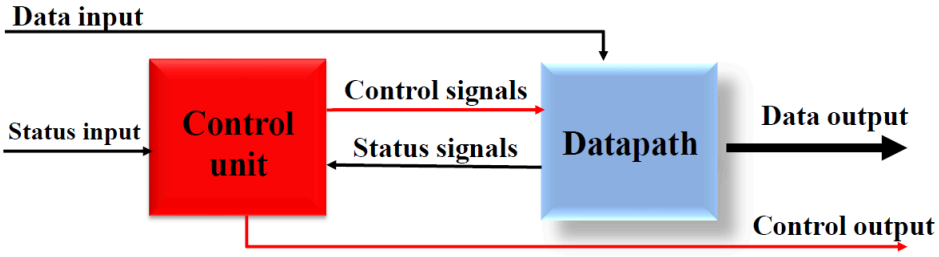
#### 2.1.1 实验目标及任务

目标：熟悉 SOC 系统的原理，掌握 IP 核集成设计 CPU 的方法。

任务:利用数据通路和控制器两个IP核集成设计CPU(根据原理图采用RTL代码方式)。

#### 2.1.2 实验原理

单周期 CPU 是最简单的一种 CPU，之所以称为“单周期”，是因为它在一个时钟周期内仅执行一条指令。它由**数据通路**和**控制器**两部分构成。




本次实验要设计的 SCPU 接口如下所示:

```
input clk,                  // 时钟信号
input rst,                  // 复位信号
input MIO_ready,            // Not used
input [31:0] inst_in,       // 指令输入总线
input [31:0] Data_in,       // 数据输入总线

output CPU_MIO,             // Not used
output MemRW,               // 内存读写信号
output [31:0] PC_out,       // PC 输出（当前指令）
```

```
output [31:0] Data_out,    // 数据输出
output [31:0] Addr_out     // 内存访问地址输出
```

## 2.2 数据通路

### 2.2.1 实验目标及任务

目标：熟悉 RISC-V RV32I 的指令特点，了解数据通路的原理，设计并测试数据通路。

任务：

1. 设计实现数据通路（采用 RTL 实现）

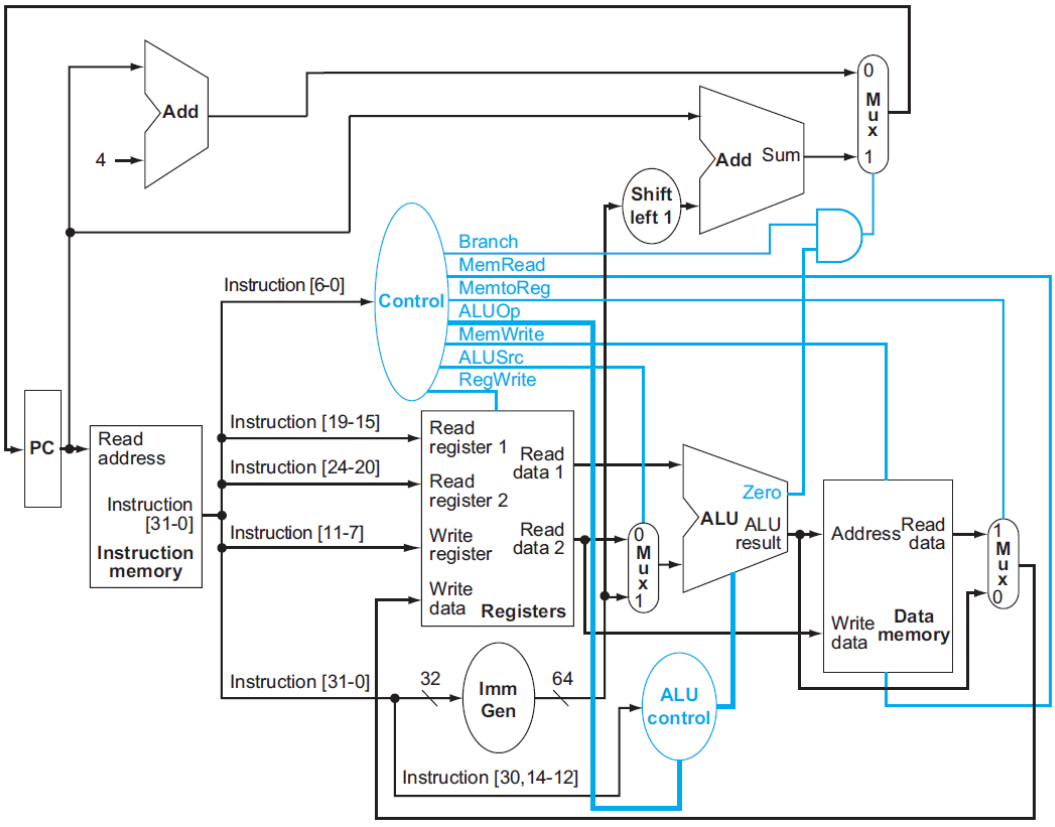
- ALU 和 Regs 调用 Exp01 设计的模块（可直接加 RTL）
- PC 寄存器设计及 PC 通路建立
- ImmGen 立即数生成模块设计
- 此实验在 Exp4-0 的基础上完成，替换 Exp4-0 的数据通路核

2. 设计数据通路测试方案并完成测试

- 通路测试：I-格式通路、R-格式通路
- 部件测试：ALU、Register Files

### 2.2.2 实验原理

下图展示的是一个单周期 CPU 的原理图，其中黑色部分代表的是数据通路（蓝色部分代表的是控制器，将在下一节介绍）。



该数据通路能够执行以下类型的指令：

- R 型指令：add、sub、and、or、xor、slt
- I 型指令：addi、andi、ori、xori、slti、lw
- S 型指令：sw
- SB 型指令：beq
- UJ 型指令：jal

先来看一下数据通路(DataPath)相关的接口：

```
// DataPath.v
input clk,                      // 时钟信号
input rst,                      // 复位信号
input [31:0] inst_field,        // 指令数据域[31:7]
input [31:0] Data_in,           // 内存输入
input [2:0] ALU_Control,        // ALU 操作控制
input [1:0] ImmSel,             // 立即数生成操作控制
input [1:0] MemtoReg,           // Regs 写入数据源控制
input ALUSrc_B,                 // ALU 端口 B 输入选择
input Jump,                     // J 指令
input Branch,                   // SB 指令
input RegWrite,                 // 寄存器写信号

output [31:0] PC_out,           // PC 指针输出
output [31:0] Data_out,         // CPU 数据输出（本实验用寄存器 2 的值作为该输出的值）
output [31:0] ALU_out           // ALU 运算输出
```

其中 MUX、ALU 等元件已在 Lab0 和 Lab1 中设计完成，只需使用 Verilog 代码模拟连线连接起来即可，故这里不再赘述。本次实验需要独立设计的有 PC 寄存器和立即数生成器。

**PC 寄存器**（实际上是一个 32 位的寄存器）的接口如下所示：

```
// REG32.v
input clk,
input rst,
input CE,               // 使能信号，本实验中其值始终为常数 1
input [31:0]D,          // 输入数据

output reg [31:0] Q     // 输出数据
```

PC 寄存器的作用：指示当前正在执行的指令地址，并和其他元件构成 PC 通路，来决定下一条指令取的是连续的下一条指令（PC + 4），还是指令中指定的指令（立即数 imm，对应指令 PC + imm）。

**立即数生成器**的接口如下所示：

```
// ImmGen.v
input [1:0] ImmSel,             // 立即数类型选择信号，有 4 个值对应 4 个指令类型，分别为：
// 00: I-type
// 01: S-type
// 10: SB-type
// 11: UJ-type
input [31:0] inst_field,        // 输入的指令

output reg [31:0] Imm_out       // 输出的立即数
```

立即数生成器的作用：根据不同的指令类型来正确地读取其中的立即数

## 2.3 控制器

### 2.3.1 实验目标及任务

目标：熟悉 RISC-V RV32I 的指令特点，了解控制器的原理，设计并测试控制器。

任务：

1. 用硬件描述语言设计实现控制器

- 根据 Exp04-1 数据通路及指令编码完成控制信号真值表
- 此实验在 Exp04-1 的基础上完成，替换 Exp04-1 的控制器核

2. 设计控制器测试方案并完成测试

- OP 译码测试：R-格式、访存指令、分支指令，转移指令
- 运算控制测试：Function 译码测试

### 2.3.2 实验原理

回顾“数据通路”一节中给出的原理图的蓝色部分，它对应的就是控制器部分。本实验的控制器包含以下控制信号：

| 信号 | 源数目 | 功能定义 | 赋值0时动作 | 赋值1时动作 | 赋值2时动作 |
|---|---|---|---|---|---|
| ALUSrc_B | 2 | ALU端口B输入选择 | 选择源操作数寄存器2数据 | 选择32位立即数（符号扩展后） | - |
| MemToReg | 3 | 寄存器写入数据选择 | 选择ALU输出 | 选择存储器数据 | 选择PC+4 |
| Branch | 2 | Beq指令目标地址选择 | 选择PC+4地址 | 选择转移目的地址 PC+imm（zero=1） | - |
| Jump | 3 | J指令目标地址选择 | 由Branch决定输出 | 选择跳转目标地址 PC+imm（JAL） | - |
| RegWrite | - | 寄存器写控制 | 禁止寄存器写 | 使能寄存器写 | - |
| MemRW | - | 存储器读写控制 | 存储器**读使能**，存储器写禁止 | 存储器**写使能**，存储器读禁止 | - |
| ALU_Control | 000-111 | 3位ALU操作控制 | 参考表ALU_Control | | |
| ImmSel | 000-111 | 3位立即数组合控制 | 参考表ImmSel | | |

这张图列出了每个信号起到的作用。其中 ALU_Control 是 3 位信号，ImmSel 和 MemToReg 是 2 位信号，其余信号都是 1 位信号。

先来看 ImmSel 信号的参考表：

| Instruction type | Instruction opcode[6:0] | Instruction operation | (sign-extend)immediate | Imm Sel |
|---|---|---|---|---|
| I-type | 0000011 | Lw;lbu;lh;lb;lhu | **(sign-extend) instr[31:20]** | 00 |
| | 0010011 | Addi;slti;sltiu;xori;ori;andi; | | |
| | 1100111 | jalr | | |
| S-type | 0100011 | Sw;sb;sh | **(sign-extend) instr[31:25],[11:7]** | 01 |
| B-type | 1100011 | Beq;bne;blt;bge;bltu;bgeu | **(sign-extend) instr[31],[7],[30:25],[11:8], 1'b0** | 10 |
| J-type | 1101111 | jal | **(sign-extend) instr[31],[19:12],[20],[30:21],1'b0** | 11 |
| | | | | |

由于几乎每类指令都用到了 ALU，所以 ALU 信号的参考表过长，限于篇幅就不列出来了。

接下来看控制器对应的接口：

```
// SCPU_ctrl.v
input [4:0] OPcode,          // Opcode------inst[6:2]
input [2:0] Fun3,            // Function-----inst[14:12]
```

```
input Fun7,                         // Function-----inst[30]
input MIO_ready,                    // CPU Wait

output reg [1:0] ImmSel,            // 立即数选择控制
output reg ALUSrc_B,                // 源操作数 2 选择
output reg [1:0] MemtoReg,          // 写回数据选择控制
output reg Jump,                    // jal
output reg Branch,                  // beq
output reg RegWrite,                // 寄存器写使能
output reg MemRW,                   // 存储器读写使能
output reg [2:0] ALU_Control,       // alu 控制
output reg CPU_MIO                  // not use
```

- 这里 `OPcode` 只取高 5 位，是因为 RISC-V 指令的低 2 位都是 `11`，而高 5 位会根据指令类型的不同而发生变化，因此只需根据高 5 位便可以判断指令类型了。

## 2.4 指令扩展

### 2.4.1 实验目标及任务

目标: 熟悉 RISC-V RV32I 的指令特点，了解控制器和数据通路的原理，扩展实验 lab4-2 CPU 指令集，设计并测试 CPU。

任务:

1. 重新设计数据通路和控制器，在 lab4-2 的基础上完成

- 兼容 lab4-1、lab4-2 的数据通路和控制器
- 替换 lab4-1、lab4-2 的数据通路控制器核
- 扩展不少于下列指令
  - 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`

2. 设计指令集测试方案并完成测试

### 2.4.2 实验原理

在原先 SCPU 的基础上，需要增加以下指令:

- R 型指令: `sltu`、`sra`、`sll`
- I 型指令: `sltiu`、`srai`、`slli`、`jalr`
- B 型指令: `bne`
- U 型指令: `lui`

下面简要分析如何在原来的 SCPU 上进行修改：

- R 型指令：为 ALU 模块添加更多的功能，包括无符号小于比较 `sltu`、算术右移 `sra`、逻辑左移 `sll` 这三个子模块。这也意味着 ALU 的选择增多了，因此 ALU 控制信号需要增加一位，MUX 的位宽也随之扩大
- I 型指令：
  - `sltiu`、`srai`、`slli`：对应增加的 R 型指令的立即数版本，因此无需再添加新的内容
  - `jalr`：它的跳转方式不同于 `jal`，因此 `Jump` 字段需要再增 1 位，以应对这种新的情况
- B 型指令（`bne`）：基本上与 `beq` 类似，唯一区别在于前者要用 `BranchN` 信号来判断两数是否不等
- U 型指令（`lui`）：由于它的立即数字段不同于其他已存在的指令，因此立即数生成器的选择信号需要再增 1 位，且需要对这类情况进行单独的处理

总结一下，虽然扩展了不少指令，但是不需要对原来 SCPU 的内部结构进行大幅度的改动，可能改动最多的地方就是 ALU，以及一些信号的位宽增加，大部分结构都是保持不变的，这也印证了课上所说的“Simplicity favors regularity”的设计原则。

# 三、 实验设备和环境

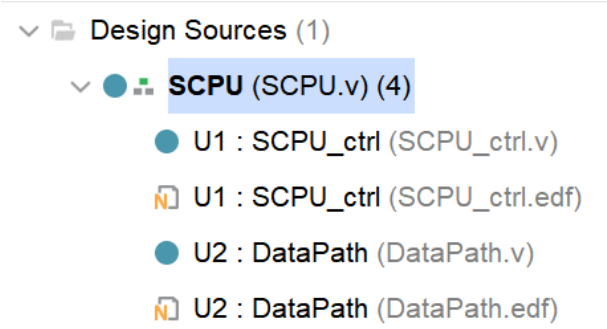
- 操作系统：Windows 11
- 开发工具：Xilinx VIVADO 2023.2
- NEXYS A7 开发板

# 四、 实验步骤

> 注意
>
> 因为 Typst 会将代码块中的 <= 自动转化为 ⩽，但我在代码中多次用到这类非阻塞赋值的符号，因此可能会影响到代码的阅读，请见谅~

## 4.1 搭建单周期 CPU

1. 在 Vivado 上新建工程 OExp04-IP2CPU
2. 在工程中新建文件 SCPU.v，作为顶层模块
3. 将已提供的两个 IP 核（数据通路和控制器）的封装文件添加到当前工程的 IP 库目录中，文件结构如下所示：




4. 按照给出的逻辑原理图，利用 Verilog 语言进行模块的调用和连接，代码如下所示：

```
`timescale 1ns / 1ps

module SCPU(
    input clk,
    input rst,
    input MIO_ready,
    input [31:0] inst_in,
    input [31:0] Data_in,

    output CPU_MIO,
    output MemRW,
    output [31:0] PC_out,
    output [31:0] Data_out,
    output [31:0] Addr_out
);

wire [1:0] ImmSel;
wire ALUSrc_B;
wire [1:0] MemtoReg;
wire Jump;
wire Branch;
wire RegWrite;
wire [2:0] ALU_Control;

SCPU_ctrl U1(
    .OPcode(inst_in[6:2]),
    .Fun3(inst_in[14:12]),
    .Fun7(inst_in[30]),
    .MIO_ready(MIO_ready),
    .ImmSel(ImmSel),
    .ALUSrc_B(ALUSrc_B),
    .MemtoReg(MemtoReg),
```

```
    .Jump(Jump),
    .Branch(Branch),
    .RegWrite(RegWrite),
    .MemRW(MemRW),
    .ALU_Control(ALU_Control),
    .CPU_MIO(CPU_MIO)
);

DataPath U2(
    .ALUSrc_B(ALUSrc_B),
    .ALU_Control(ALU_Control),
    .Branch(Branch),
    .Data_in(Data_in),
    .ImmSel(ImmSel),
    .Jump(Jump),
    .MemtoReg(MemtoReg),
    .RegWrite(RegWrite),
    .clk(clk),
    .inst_field(inst_in),
    .rst(rst),
    .ALU_out(Addr_out),
    .Data_out(Data_out),
    .PC_out(PC_out)
);

endmodule
```

## 4.2 数据通路设计

1. 在 Vivado 上新建工程 OExp04-Datapath
2. 导入 Lab0 和 Lab1 中已经设计好的模块，包括 ALU、Regs、and32、ADC32 等等
3. 新建子模块 REG32 和 ImmSel（分别作为 PC 寄存器和立即数生成器），代码分别如下所示：

```
// REG32.v
`timescale 1ns / 1ps

module REG32(
    input clk,
    input rst,
    input CE,
    input [31:0]D,

    output reg [31:0] Q
);

    always @(posedge clk or posedge rst) begin
        if (rst == 1)
            Q <= 32'b0;
        else if (CE == 1)
            Q <= D;
    end

endmodule
```

```
// ImmSel.v
`timescale 1ns / 1ps

module ImmGen(
    input [1:0] ImmSel,
    input [31:0] inst_field,
    output reg [31:0] Imm_out
);

    always @(*) begin
        case (ImmSel)
            // I-type
            2'b00: Imm_out = {{20{inst_field[31]}}, inst_field[31:20]};
            // S-type
          2'b01: Imm_out = {{20{inst_field[31]}}, inst_field[31:25],
```

```
inst_field[11:7]};
            // B-type
             2'b10: Imm_out = {{20{inst_field[31]}}, inst_field[7],
inst_field[30:25], inst_field[11:8], 1'b0};
            // J-type
            2'b11: Imm_out = {inst_field[31:12], 12'b0};
        endcase
    end

endmodule
```

4. 在工程中新建文件 DataPath.v，作为顶层模块，按照给出的逻辑原理图，利用 Verilog 语言进行模块的调用和连接，代码如下所示：

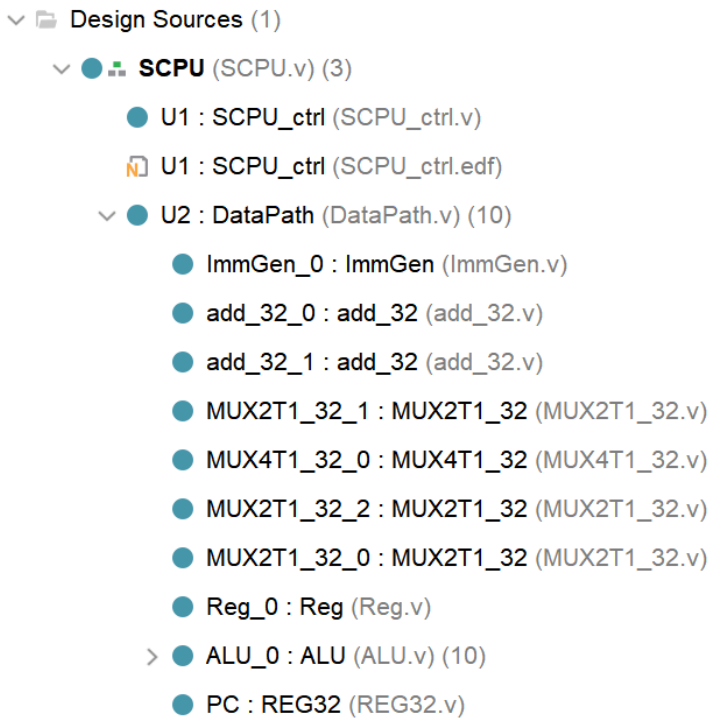
```
`timescale 1ns / 1ps

module DataPath(
    input clk,
    input rst,
    input [31:0] inst_field,
    input [31:0] Data_in,
    input [2:0] ALU_Control,
    input [1:0] ImmSel,
    input [1:0] MemtoReg,
    input ALUSrc_B,
    input Jump,
    input Branch,
    input RegWrite,

    output [31:0] PC_out,
    output [31:0] Data_out,
    output [31:0] ALU_out
);

    wire [31:0] ImmGen_0_out;
    wire [31:0] add_32_0_c;
    wire [31:0] add_32_1_c;
    wire MUX2T1_32_1_s;
    wire [31:0] MUX2T1_32_0_o;
    wire [31:0] MUX2T1_32_1_o;
    wire [31:0] MUX2T1_32_2_o;
    wire [31:0] MUX4T1_32_0_o;
    wire [31:0] PC_Q;
```

```
    wire [31:0] Reg_Rs1_data;
    wire [31:0] Reg_Rs2_data;
    wire ALU_zero;
    wire [31:0] ALU_res;

    ImmGen ImmGen_0(
        .ImmSel(ImmSel),
        .inst_field(inst_field),
        .Imm_out(ImmGen_0_out)
    );

    add_32 add_32_0(
        .a(PC_Q),
        .b(32'd4),
        .c(add_32_0_c)
    );

    add_32 add_32_1(
        .a(PC_Q),
        .b(ImmGen_0_out),
        .c(add_32_1_c)
    );

    MUX2T1_32 MUX2T1_32_1(
        .I0(add_32_0_c),
        .I1(add_32_1_c),
        .s(MUX2T1_32_1_s),
        .o(MUX2T1_32_1_o)
    );

    MUX4T1_32 MUX4T1_32_0(
        .s(MemtoReg),
        .I0(ALU_res),
        .I1(Data_in),
        .I2(add_32_0_c),
        .I3(add_32_0_c),
        .o(MUX4T1_32_0_o)
    );

    MUX2T1_32 MUX2T1_32_2(
        .I0(MUX2T1_32_1_o),
        .I1(add_32_1_c),
```

```
        .s(Jump),
        .o(MUX2T1_32_2_o)
    );

    MUX2T1_32 MUX2T1_32_0(
        .I0(Reg_Rs2_data),
        .I1(ImmGen_0_out),
        .s(ALUSrc_B),
        .o(MUX2T1_32_0_o)
    );

    Reg Reg_0(
        .clk(clk),
        .rst(rst),
        .Rs1_addr(inst_field[19:15]),
        .Rs2_addr(inst_field[24:20]),
        .Wt_addr(inst_field[11:7]),
        .Wt_data(MUX4T1_32_0_o),
        .RegWrite(RegWrite),
        .Rs1_data(Reg_Rs1_data),
        .Rs2_data(Reg_Rs2_data)
    );

    ALU ALU_0(
        .A(Reg_Rs1_data),
        .ALU_operation(ALU_Control),
        .B(MUX2T1_32_0_o),
        .res(ALU_res),
        .zero(ALU_zero)
    );

    REG32 PC(
        .clk(clk),
        .rst(rst),
        .CE(1'b1),
        .D(MUX2T1_32_2_o),
        .Q(PC_Q)
    );

    assign MUX2T1_32_1_s = Branch & ALU_zero;
    assign Data_out = Reg_Rs2_data;
    assign ALU_out = ALU_res;
```

```
    assign PC_out = PC_Q;

endmodule
```

5. 将该数据通路模块替换原来用到的 IP 核，此时 SCPU 工程文件结构如下所示:




## 4.3 控制器设计

1. 在 Vivado 上新建工程 OExp04-SCPU_ctrl
2. 创建顶层模块文件 SCPU_ctrl.v，通过 Verilog 语言实现控制器的功能，由主控制器部分和 ALU 控制部分构成，下面分别展示了它们对应的代码(接口在“实验原理”部分已给出，故不再赘述):

```
reg [1:0] ALUop;
wire[3:0] Fun;

// 主控制器
always @(*) begin
    ALUSrc_B = 0;
    MemtoReg = 0;
    RegWrite = 0;
    Branch = 0;
    Jump = 0;
    MemRW = 0;
    CPU_MIO = 0;
    ALUop = 2'b10;
```

```
    case(OPcode)
        5'b01100: begin     // ALU
            ALUSrc_B = 0;
            MemtoReg = 0;
            RegWrite = 1;
            MemRW = 0;
            Branch = 0;
            Jump = 0;
            ALUop = 2'b10;
            ImmSel = 2'b00;
        end

        5'b00100: begin    // ALU(addi, …)
            ALUSrc_B = 1;
            MemtoReg = 0;
            RegWrite = 1;
            MemRW = 0;
            Branch = 0;
            Jump = 0;
            ALUop = 2'b11;
            ImmSel = 2'b00;
        end

        5'b00000: begin     // load
            ALUSrc_B = 1;
            MemtoReg = 1;
            RegWrite = 1;
            MemRW = 0;
            Branch = 0;
            Jump = 0;
            ALUop = 2'b00;
            ImmSel = 2'b00;
        end

        5'b01000: begin    // store
            ALUSrc_B = 1;
            MemtoReg = 0;
            RegWrite = 0;
            MemRW = 1;
            Branch = 0;
            Jump = 0;
            ALUop = 2'b00;
```

```
            ImmSel = 2'b01;
        end

        5'b11000: begin   // beq
            ALUSrc_B = 0;
            MemtoReg = 0;
            RegWrite = 0;
            MemRW = 0;
            Branch = 1;
            Jump = 0;
            ALUop = 2'b01;
            ImmSel = 2'b10;
        end

        5'b11011: begin   // jump
            ALUSrc_B = 0;
            MemtoReg = 2'b10;
            RegWrite = 1;
            MemRW = 0;
            Branch = 0;
            Jump = 1;
            ALUop = 2'b00;
            ImmSel = 2'b11;
        end

        default: begin
            ALUSrc_B = 0;
            MemtoReg = 0;
            RegWrite = 0;
            MemRW = 0;
            Branch = 0;
            Jump = 0;
            ALUop = 2'b00;
            ImmSel = 2'b00;
        end
    endcase
end

// ALU 控制器
assign Fun = {Fun3, Fun7};
always @(*) begin
    case(ALUop)
```

```
        2'b00: ALU_Control = 3'b010;    // add 计算地址（load/store）
        2'b01: ALU_Control = 3'b110;    // sub 比较条件（beq）
        2'b10:      // R-Type
            case(Fun)
                4'b0000: ALU_Control = 3'b010;    // add
                4'b0001: ALU_Control = 3'b110;    // sub
                4'b1110: ALU_Control = 3'b000;    // and
                4'b1100: ALU_Control = 3'b001;    // or
                4'b0100: ALU_Control = 3'b111;    // slt
                4'b1010: ALU_Control = 3'b101;    // srl
                4'b1000: ALU_Control = 3'b011;    // xor
                default: ALU_Control = 3'bx;
            endcase
        2'b11:      // Immediate Operations
            case(Fun3)
                3'b000: ALU_Control = 3'b010;    // addi
                3'b111: ALU_Control = 3'b000;    // andi
                3'b110: ALU_Control = 3'b001;    // ori
                3'b010: ALU_Control = 3'b111;    // slti
                3'b101: ALU_Control = 3'b101;    // srli
                3'b100: ALU_Control = 3'b011;    // xori
                default: ALU_Control = 3'bx;
            endcase
    endcase
end
```

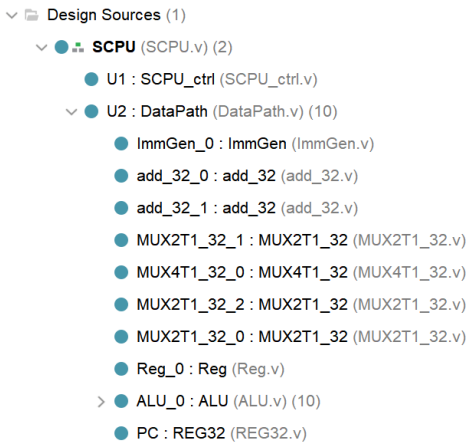
3. 用以下的仿真代码，对控制器部分进行仿真，将生成的波形图与实验原理部分控制器信号表进行比对，检验控制器的功能正确性（波形图及其分析将在“实验结果与分析”一节给出）。

```
`timescale 1ns / 1ps

module SCPU_ctrl_tb();
    reg [4:0] OPcode;
    reg [2:0] Fun3;
    reg Fun7;
    reg MIO_ready;
    wire [1:0] ImmSel;
    wire ALUSrc_B;
    wire [1:0] MemtoReg;
    wire Jump;
    wire Branch;
    wire RegWrite;
```

```
    wire MemRW;
    wire [2:0] ALU_Control;
    wire CPU_MIO;

    initial begin
        //Initialize Inputs
        OPcode = 0;
        // Fun = 0;
        Fun3 = 0;
        Fun7 = 0;
        MIO_ready = 0;
        #40;
        // Wait 40ns for global reset to finish
        // Add stimulus here
        // 检查输出信号和关键信号输出是否满足真值表
        OPcode = 5'b01100;     // ALU 指令, 检查 ALUop = 2'b10;RegWrite
= 1
        Fun3 = 3'b000; Fun7 = 1'b0;     // add, 检查 ALU_Control = 3'b010
        #20;
        Fun3 = 3'b000; Fun7 = 1'b1;     // sub, 检查 ALU_Control = 3'b110
        #20;
        Fun3 = 3'b111; Fun7 = 1'b0;     // and, 检查 ALU_Control = 3'b000
        #20;
        Fun3 = 3'b110; Fun7 = 1'b0;     // or, 检查 ALU_Control = 3'b001
        #20;
        Fun3 = 3'b010; Fun7 = 1'b0;     // slt, 检查 ALU_Control = 3'b111
        #20;
        Fun3 = 3'b101; Fun7 = 1'b0;     // srl, 检查 ALU_Control = 3'b101
        #20;
        Fun3 = 3'b100; Fun7 = 1'b0;     // xor, 检查 ALU_Control = 3'b011
        #20;
        Fun3 = 3'b111; Fun7 = 1'b1;     // 间隔
        #10;
        OPcode = 5'b00000;     // load 指令, 检查 ALUop = 2'b00,
        #20;     // ALUSrc_B = 1, MemtoReg = 1, RegWrite = 1
        OPcode = 5'b01000;
        #20;     // store 指令, 检查 ALUop = 2'b00, MemRW = 1,ALUSrc_B
= 1
         OPcode = 5'b11000;     // beq 指令, 检查 ALUop = 2'b01,Branch
= 1
        #20;
        OPcode = 5'b11011;     // jump 指令, 检查 Jump = 1
```

```
        #20;
        OPcode = 5'b00100;     // I 指令, 检查 ALUop = 2'b11; RegWrite
= 1
        Fun3 = 3'b000;     // addi, 检查 ALU_Control = 3'b010
        #20;
        Fun3 = 3'b111;     // andi, 检查 ALU_Control = 3'b000
        #20;
        Fun3 = 3'b110;     // ori, 检查 ALU_Control = 3'b001
        #20;
        Fun3 = 3'b010;     // slti, 检查 ALU_Control = 3'b111
        #20;
        Fun3 = 3'b101;     // srli, 检查 ALU_Control = 3'b101
        #20;
        Fun3 = 3'b100;     // xori, 检查 ALU_Control = 3'b011
        #20;
    end

    SCPU_ctrl SCPU_ctrl_U(
        OPcode,
        Fun3,
        Fun7,
        MIO_ready,
        ImmSel,
        ALUSrc_B,
        MemtoReg,
        Jump,
        Branch,
        RegWrite,
        MemRW,
        ALU_Control,
        CPU_MIO
    );

endmodule
```

4. 若控制器功能正确，则用刚刚设计好的控制器替换原来 SCPU 内的控制器 IP 核，此时的 SCPU 工程的文件结构应如下所示：




5. 根据 PPT 给出的逻辑原理图，利用 RAM（对应的 coe 文件用于初始化数据，在 Lab0 中已给出）和 ROM（用于存储 RISC-V 指令的机器码，可自行设计）搭建一个仿真平台，检验我们设计的 SCPU 的功能正确性。仿真平台的代码如下所示：

```
`timescale 1ns / 1ps

module soc_test_wrapper(
    input clk,
    input rst
);

    wire [31:0] RAM_B_douta;
    wire [31:0] SCPU_inst_in;
    wire [31:0] SCPU_Addr_out;
    wire SCPU_MemRW;
    wire [31:0] SCPU_Data_out;
    wire [31:0] SCPU_PC_out;
    wire MIO;
    wire [9:0] Addr_out_slice;
    wire [9:0] PC_out_slice;

    RAM_B RAM_B_0(
        .clka(~clk),
        .wea(SCPU_MemRW),
        .addra(Addr_out_slice),
        .dina(SCPU_Data_out),
        .douta(RAM_B_douta)
    );

    ROM ROM_0(
        .a(PC_out_slice),
        .spo(SCPU_inst_in)
```

```
    );

    SCPU SCPU_wrapper_0(
        .Data_in(RAM_B_douta),
        .MIO_ready(MIO),
        .clk(clk),
        .inst_in(SCPU_inst_in),
        .rst(rst),
        .Addr_out(SCPU_Addr_out),
        .CPU_MIO(MIO),
        .Data_out(SCPU_Data_out),
        .MemRW(SCPU_MemRW),
        .PC_out(SCPU_PC_out)
    );

    assign Addr_out_slice = SCPU_Addr_out[11:2];
    assign PC_out_slice = SCPU_PC_out[11:2];

endmodule
```

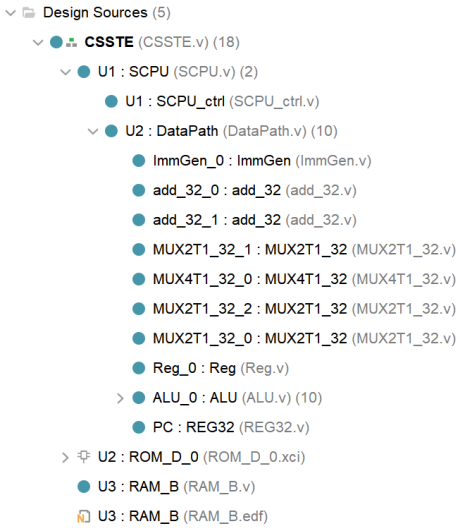
由于 RAM 和 ROM 的构建方法已在 Lab0 的 PPT 内详细介绍过，故这里不再赘述。

6. 接下来编写仿真代码（代码如下所示），对仿真平台进行仿真，观察波形图是否符合预期变化，从而判断 CPU 功能的正确与否（波形图及其分析将在“实验结果与分析”一节给出）
   - 由于外部端口只有 clk,rst，因此需要在 Scope 窗口添加观察信号（在 Scope 窗口下点击子模块，右键 Add to wave window 添加信号到波形窗口，可根据需要自行添加）

```
`timescale 1ns / 1ps

module tb();

    reg clk;
    reg rst;

    soc_test_wrapper u(
        .clk(clk),
        .rst(rst)
    );

    always #5 clk = ~clk;
```

```
    initial begin
        clk = 0;
        rst = 1;
        #1;
        rst = 0;
    end

endmodule
```

7. 用我们刚刚设计好的整个 SCPU 模块替换 Lab2 的 SOC 系统中预先提供给我们的 SCPU 模块，此时 SOC 系统工程的文件结构应如下所示（这里只截取一部分内容）：




8. 对 SOC 系统工程进行综合、实现，烧录为 bit 流文件，然后上板验证，通过 VGA 观察结果是否符合预期，并需要设计测试记录表格，记录程序的 VGA 输出。

## 4.4 指令扩展

1. 新建工程项目 OExp04-ExtSCPU，将原 SCPU 的模块整体搬到该工程文件中，以便接下来的修改和扩展
2. 先处理数据通路部分：数据通路中的两个子模块 ALU 和 ImmGen 需要进行修改，代码分别如下所示：

```
`timescale 1ns / 1ps

module ALU(
    input [31:0] A,
    input [3:0] ALU_operation,
    input [31:0] B,
    output [31:0] res,
    output zero
);
```

```
// 省略一些接线
wire [31:0]MUX16T1_32_0_o;

// 省略未修改的子模块

// 由于 ALU 功能的扩展，对应的控制信号也随之扩展，因此 MUX 也要做扩展
MUX16T1_32 MUX16T1_32_0(
    .I0(and32_0_res),
    .I1(or32_0_res),
    .I2(addc_32_0_S[31:0]),
    .I3(32'b0),
    .I4(32'b0),
    .I5(32'b0),
    .I6(addc_32_0_S[31:0]),
    .I7({31'b0,addc_32_0_S[32]}),
    .I8(32'b0),
    .I9(sltu32_0_res),
    .I10(32'b0),
    .I11(32'b0),
    .I12(xor32_0_res),
    .I13(srl32_0_res),
    .I14(sll32_0_res),
    .I15(sra32_0_res),
    .o(MUX16T1_32_0_o),
    .s(ALU_operation)
);

// 新增的子模块
sll32 sll32_0(
    .A(A),
    .B(B),
    .res(sll32_0_res)
);

sra32 sra32_0(
    .A(A),
    .B(B),
    .res(sra32_0_res)
);

sltu32 sltu32_0(
```

```
        .A(A),
        .B(B),
        .S(sltu32_0_res)
    );

    assign res = MUX16T1_32_0_o;
endmodule
```

```
`timescale 1ns / 1ps

module ImmGen(
    input [2:0] ImmSel,         // 信号扩宽 1 位
    input [31:0] inst_field,
    output reg [31:0] Imm_out
);

    always @(*) begin
        case (ImmSel)
            // I-type
            3'b001: Imm_out = {{20{inst_field[31]}}, inst_field[31:20]};
            // S-type
          3'b010: Imm_out = {{20{inst_field[31]}}, inst_field[31:25],
inst_field[11:7]};
            // B-type
            3'b011: Imm_out = {{20{inst_field[31]}}, inst_field[7],
inst_field[30:25], inst_field[11:8], 1'b0};
            // J-type
          3'b100: Imm_out = {{12{inst_field[31]}}, inst_field[19:12],
inst_field[20], inst_field[30:21], 1'b0};
            // U-type
            3'b000: Imm_out = {inst_field[31:12], 12'h000};
        endcase
    end

endmodule
```

修改完子模块后，再来修改数据通路模块，代码如下所示：

```
`timescale 1ns / 1ps

module DataPath_more(
    input clk,
    input rst,
```

```
    input [31:0] inst_field,
    input [31:0] Data_in,
    input [3:0] ALU_Control,    // 位数增宽
    input [2:0] ImmSel,         // 位数增宽
    input [1:0] MemtoReg,
    input ALUSrc_B,
    input [1:0] Jump,           // 位数增宽
    input Branch,
    input BranchN,              // 新增信号
    input RegWrite,

    output [31:0] PC_out,
    output [31:0] Data_out,
    output [31:0] ALU_out
);

    // 省略一些接线
    // 省略未改变的子模块

    MUX4T1_32 MUX4T1_32_0(
        .s(MemtoReg),
        .I0(ALU_res),
        .I1(Data_in),
        .I2(add_32_0_c),
        .I3(ImmGen_0_out),     // 这里做了修改
        .o(MUX4T1_32_0_o)
    );

    // 原来是一个 2-1 MUX, 由于选择变多扩展至 4-1 MUX
    MUX4T1_32 MUX4T1_32_1(
        .I0(MUX2T1_32_1_o),
        .I1(add_32_1_c),
        .I2(ALU_res),
        .I3(MUX2T1_32_1_o),
        .s(Jump),
        .o(MUX4T1_32_1_o)
    );

    // 多了对 bne 指令的判别
    assign MUX2T1_32_1_s = (Branch & ALU_zero) | (BranchN & ~ALU_zero);
    // 其余赋值语句略
endmodule
```

3. 接下来修改控制器模块，代码如下所示：

```
`timescale 1ns / 1ps

module SCPU_ctrl_more(
    input [4:0] OPcode,                 // Opcode------inst[6:2]
    input [2:0] Fun3,                   // Function-----inst[14:12]
    input Fun7,                         // Function-----inst[30]
    input MIO_ready,                    // CPU Wait
    output reg [2:0] ImmSel,            // 立即数选择控制（位数扩宽）
    output reg ALUSrc_B,                // 源操作数 2 选择
    output reg [1:0] MemtoReg,          // 写回数据选择控制
    output reg [1:0] Jump,              // jump（位数扩宽）
    output reg Branch,                  // beq
    output reg BranchN,                 // bne（新增）
    output reg RegWrite,                // 寄存器写使能
    output reg MemRW,                   // 存储器读写使能
    output reg [3:0] ALU_Control,       // alu 控制（位数扩宽）
    output reg CPU_MIO                  // not use
);

// 省略一些连线

// 主控制器
always @(*) begin
    // 省略初始化语句
    case(OPcode)
        // 省略对原有指令类型的处理语句
        5'b11000: begin    // beq & bne
            ALUSrc_B = 0;
            MemtoReg = 0;
            RegWrite = 0;
            MemRW = 0;
            Jump = 0;
            ALUop = 2'b01;
            ImmSel = 3'b011;
            if (Fun3 == 3'b000) begin
                Branch = 1;
                BranchN = 0;
            end else begin
                Branch = 0;
                BranchN = 1;
            end
```

```
        end

        5'b11001: begin    // jalr
            ALUSrc_B = 1;
            MemtoReg = 2'b10;
            RegWrite = 1;
            MemRW = 0;
            Branch = 0;
            BranchN = 0;
            Jump = 2;
            ALUop = 2'b00;
            ImmSel = 3'b001;
        end

        5'b01101: begin     // lui
            ALUSrc_B = 0;
            MemtoReg = 2'b11;
            RegWrite = 1;
            MemRW = 0;
            Branch = 0;
            BranchN = 0;
            Jump = 0;
            ALUop = 2'b00;
            ImmSel = 3'b000;
        end
    endcase
end

// ALU 控制器
always @(*) begin
    case(ALUop)
        // 省略未修改的部分
        2'b11:      // Immediate Operations
            case(Fun3)
                3'b000: ALU_Control = 4'b0010;     // addi
                3'b010: ALU_Control = 4'b0111;     // slti
                3'b011: ALU_Control = 4'b1001;     // sltiu
                3'b100: ALU_Control = 4'b1100;     // xori
                3'b110: ALU_Control = 4'b0001;     // ori
                3'b111: ALU_Control = 4'b0000;     // andi
                3'b001: ALU_Control = 4'b1110;     // slli
                3'b101:
```

```
                    if (Fun7 == 0)
                        ALU_Control = 4'b1101;    // srli
                    else
                        ALU_Control = 4'b1111;    // srai
                default: ALU_Control = 4'bx;
            endcase
    endcase
end

endmodule
```

4. 借鉴并适当修改之前设计控制器用到的仿真代码，对这里新得到的控制器模块进行正确性测试。因为仿真代码比较相似，故省略，但会在“实验结果与分析”一节给出波形图及简要分析。
5. 修改 SCPU 模块的内容。与原来的 SCPU 相比，输入和输出接口均为发生变化，发生变化的是部分连线位宽的增加，以及新增的 `BranchN` 接口，故这里不再列出具体的代码。
6. 将原来测试 SCPU 功能正确性的仿真平台移植到此工程文件中，基本上不需要做多大的改动，但可以将 ROM 的 coe 文件换成 PPT 中给出的 DEMO 程序，便于后面的检验（因为验收要求中已给出正确的结果，只要与正确结果比对即可判断正确性）。波形图及其分析放在“实验结果与分析”一节。
7. 若 SCPU 功能正确，将其放入 SOC 系统工程文件，替换原来的 SPCU，然后进行综合、烧录、上板验证，观察结果是否符合预期。

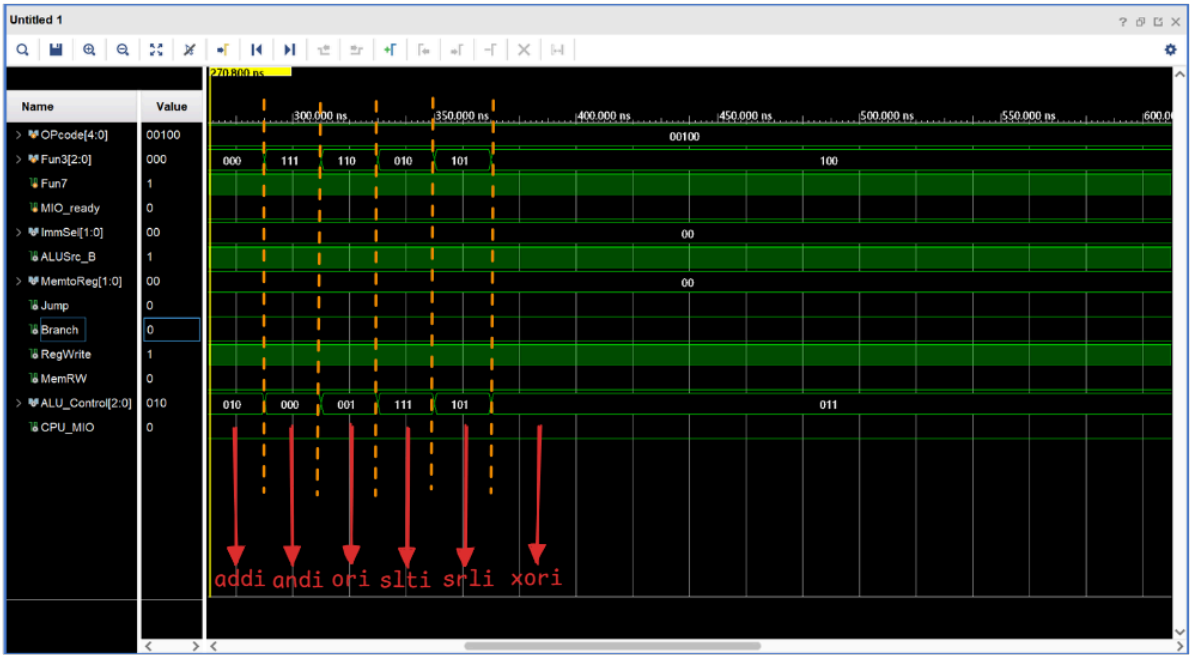
# 五、 实验结果与分析

## 5.1 控制器仿真结果

先分析 0-270ns 之间的波形：


- 0-40ns：初始化状态（复位）
- 40-180ns：执行 R 型指令，发现 `ALUop = 2'b10`，`RegWrite = 1`，符合预期
  - 40-60ns：执行 `add` 指令，发现 `ALU_Control = 3'b010`，符合预期
  - 60-80ns：执行 `sub` 指令，发现 `ALU_Control = 3'b110`，符合预期
  - 80-100ns：执行 `and` 指令，发现 `ALU_Control = 3'b000`，符合预期
  - 100-120ns：执行 `or` 指令，发现 `ALU_Control = 3'b001`，符合预期
  - 120-140ns：执行 `slt` 指令，发现 `ALU_Control = 3'b111`，符合预期
  - 140-160ns：执行 `srl` 指令，发现 `ALU_Control = 3'b101`，符合预期
  - 160-180ns：执行 `xor` 指令，发现 `ALU_Control = 3'b011`，符合预期
- 180-190ns：中场休息
- 190-210ns：执行 load 类指令，发现 `ALUop = 2'b00`，`ALUSrc_B = 1`，`MemtoReg = 1`，`RegWrite = 1`，符合预期
- 210-230ns：执行 store 类指令，发现 `ALUop = 2'b00`，`ALUSrc_B = 1`，`MemRW = 1`，符合预期
- 230-250ns：执行 `beq` 指令，发现 `ALUop = 2'b01`，`Branch = 1`，符合预期
- 250-270ns：执行 `jal` 指令，发现 `Jump = 1`，符合预期

再分析 270ns 之后的波形：




- 270ns 之后：执行 I 型指令，发现 `ALUop = 2'b11`，`RegWrite = 1`，符合预期
  - 270-290ns：执行 `addi` 指令，发现 `ALU_Control = 3'b010`，符合预期
  - 290-310ns：执行 `andi` 指令，发现 `ALU_Control = 3'b000`，符合预期
  - 310-330ns：执行 `ori` 指令，发现 `ALU_Control = 3'b001`，符合预期
  - 330-350ns：执行 `slti` 指令，发现 `ALU_Control = 3'b111`，符合预期
  - 350-370ns：执行 `srli` 指令，发现 `ALU_Control = 3'b101`，符合预期
  - 370ns 之后：执行 `xori` 指令，发现 `ALU_Control = 3'b011`，符合预期

综上，控制器模块功能正确。

## 5.2 SCPU 仿真结果

我预先编写以下汇编代码作为 demo 程序：

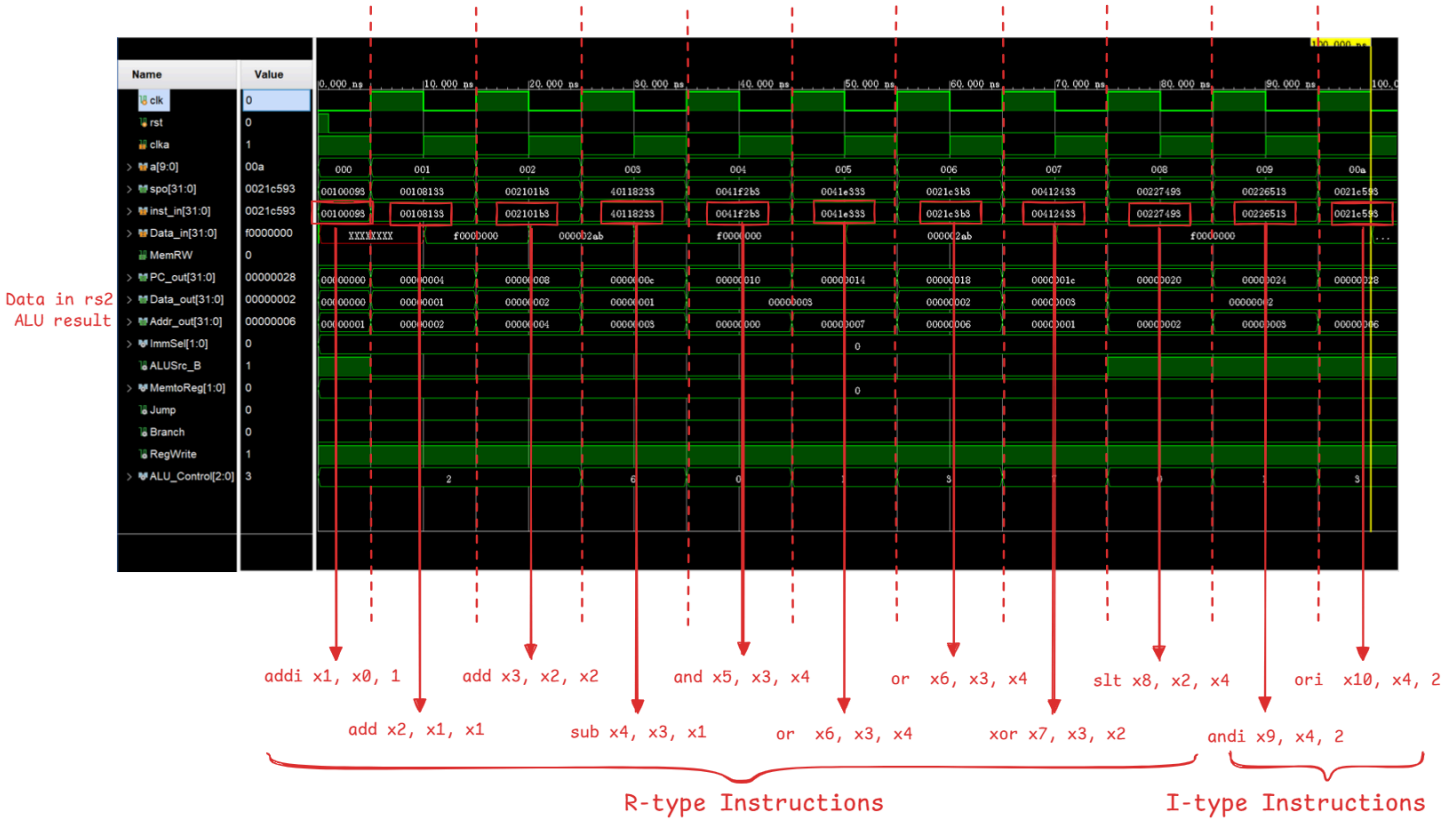
```
    addi x1, x0, 1      ; // x1 = 1
    add x2, x1, x1      ; // x2 = 2
    add x3, x2, x2      ; // x3 = 4
    sub x4, x3, x1      ; // x4 = 3
    and x5, x3, x4      ; // x5 = 0
    or  x6, x3, x4      ; // x6 = 7
    xor x7, x3, x2      ; // x7 = 6
    slt x8, x2, x4      ; // x8 = 1
    andi x9, x4, 2      ; // x9 = 2
    ori  x10, x4, 2     ; // x10 = 3
    xori x11, x3, 2     ; // x11 = 6
    slti x12, x3, 1     ; // x12 = 0
    beq x0, x5, branch  ; // Jump!
```

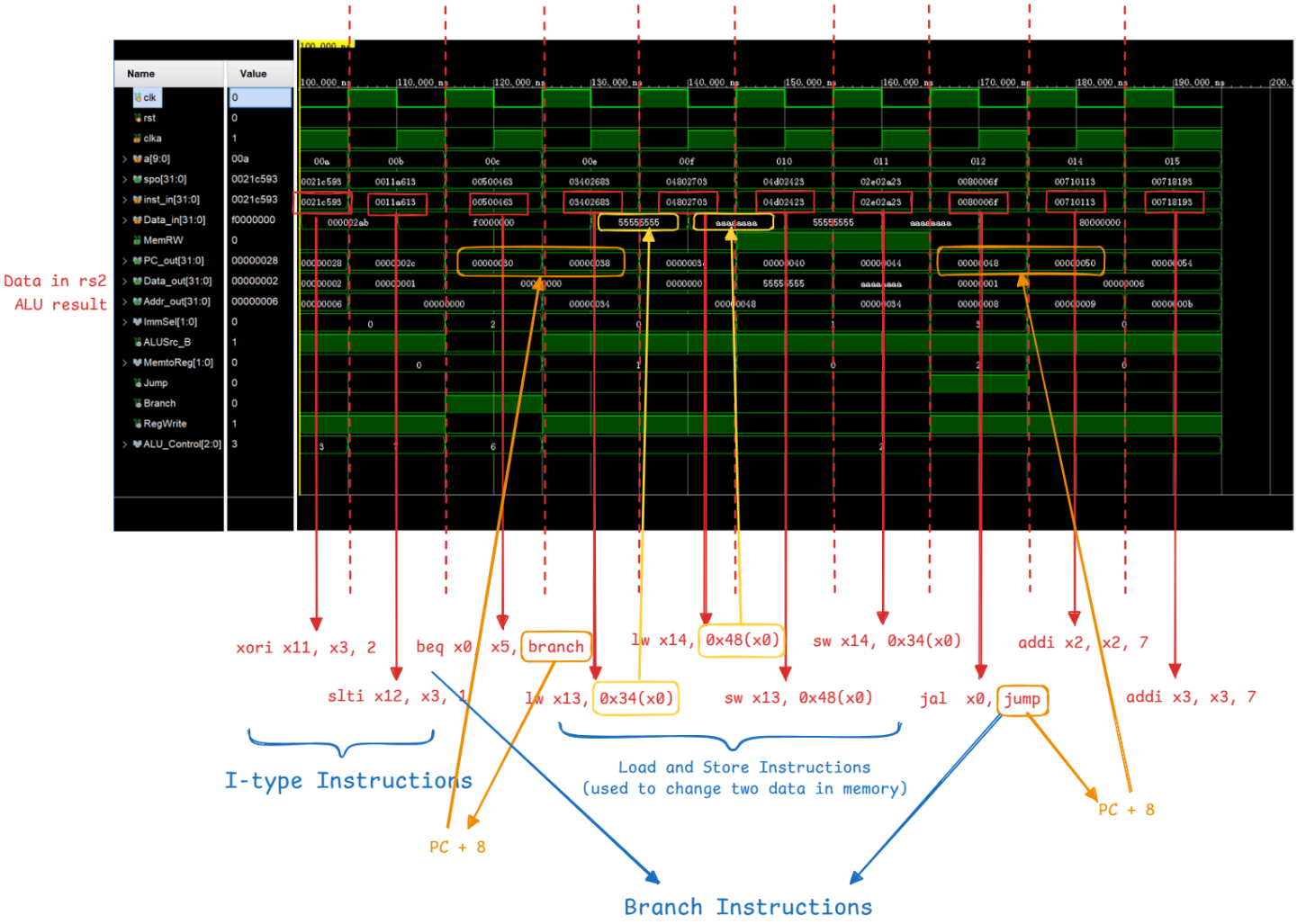
```
    addi x2, x2, 8
branch:
    lw   x13, 0x34(x0)  ; // x13 = 55555555
    lw   x14, 0x48(x0)  ; // x14 = AAAAAAAA
    sw   x13, 0x48(x0)
    sw   x14, 0x34(x0)
    jal  x0, jump       ; // Jump!
    addi x2, x2, 8
jump:
    addi x2, x2, 7      ; // x2 = 9
    addi x3, x3, 7
```

然后将其转化为 ROM 的 coe 文件，如下所示：

```
memory_initialization_radix=16;
memory_initialization_vector=
00100093,00108133,002101B3,40118233,0041F2B3,0041E333,
0021C3B3,00412433,00227493,00226513,0021C593,0011A613,
00500463,00810113,03402683,04802703,04D02423,02E02A23,
0080006F,00810113,00710113,00718193;
```

仿真结果（以及对波形的分析）如下面两张图片所示：





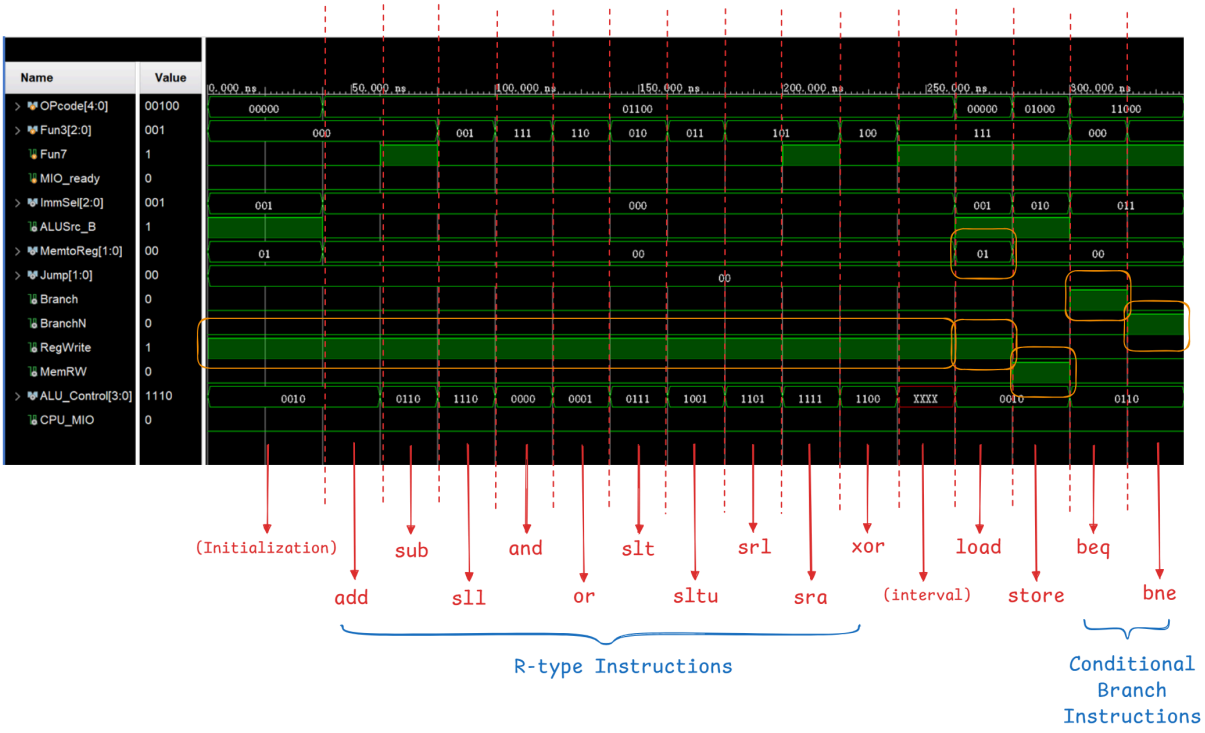

## 5.3 SOC 上板验证

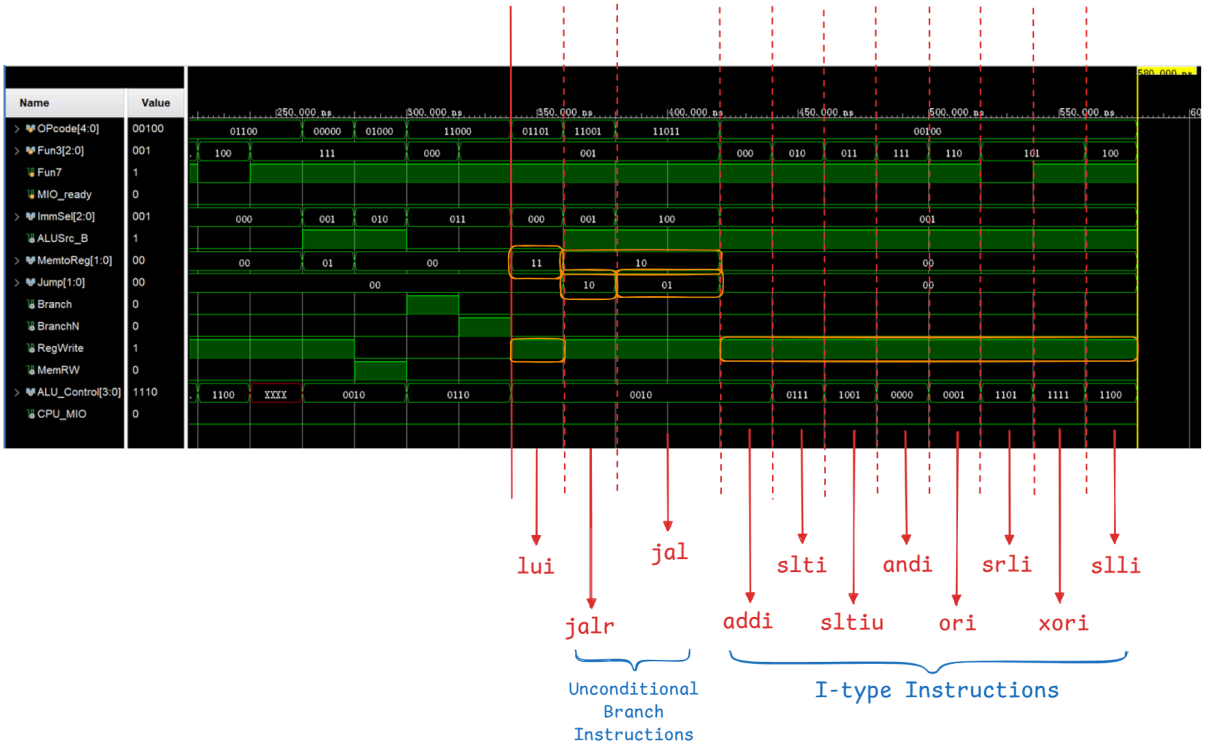
将烧录好的 bit 流文件传入 NEXYS 板上，通过 VGA 显示器观察结果，与波形图的信号进行比对，发现均符合预期，且经助教验收通过。

## 5.4 指令扩展

### 5.4.1 控制器仿真结果

下面两张图展示了指令扩展后控制器的仿真波形图及其分析：

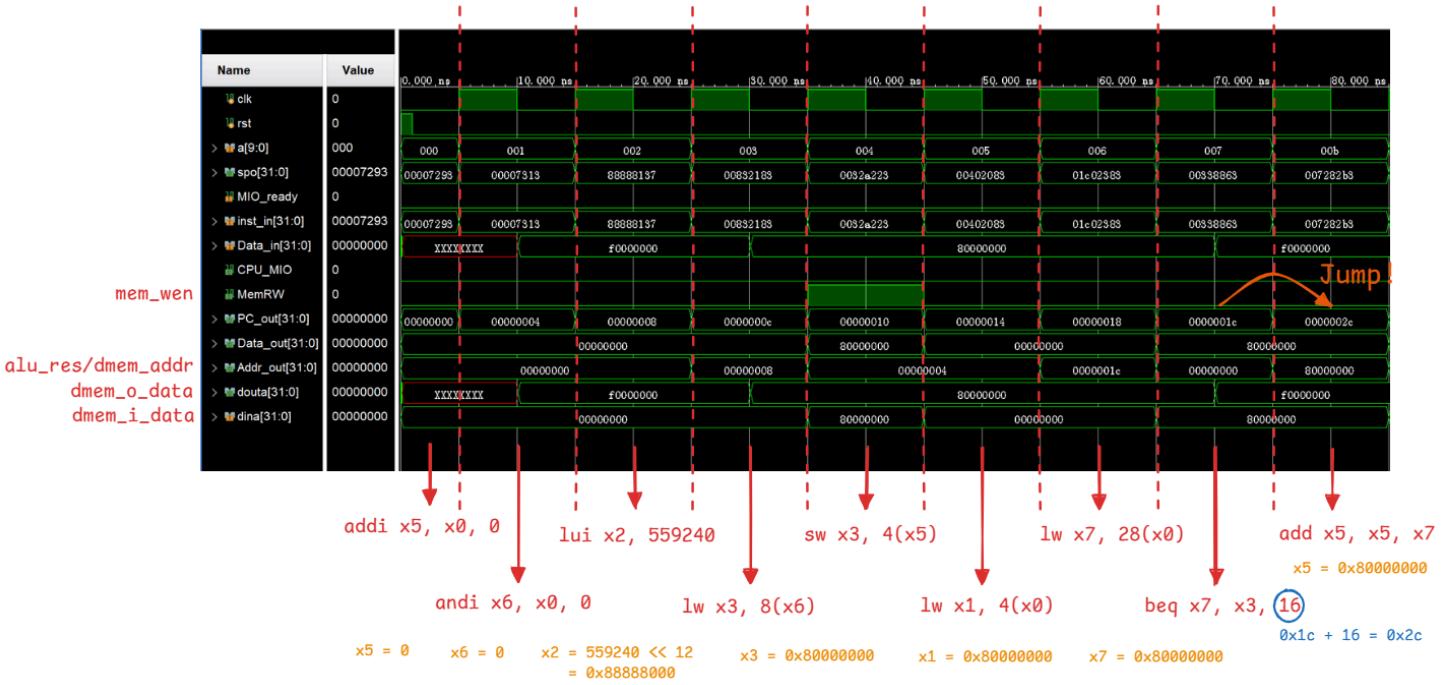


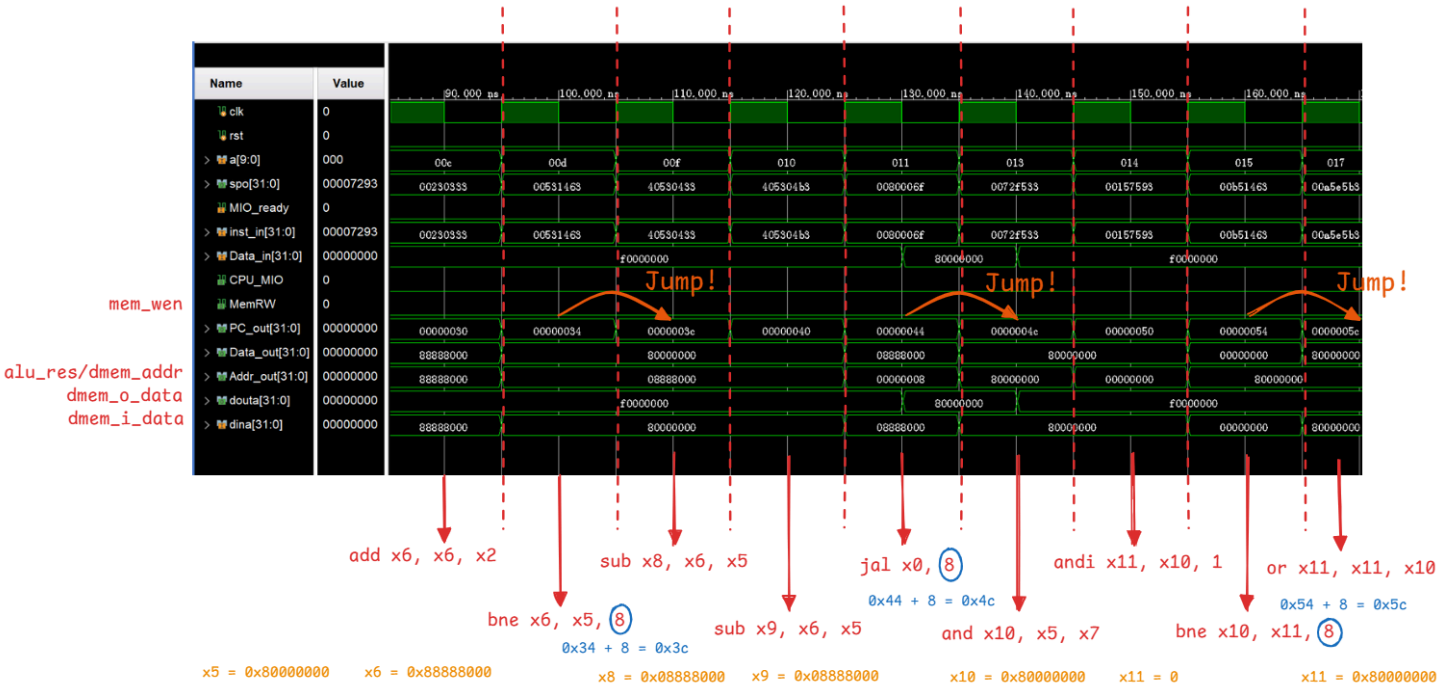



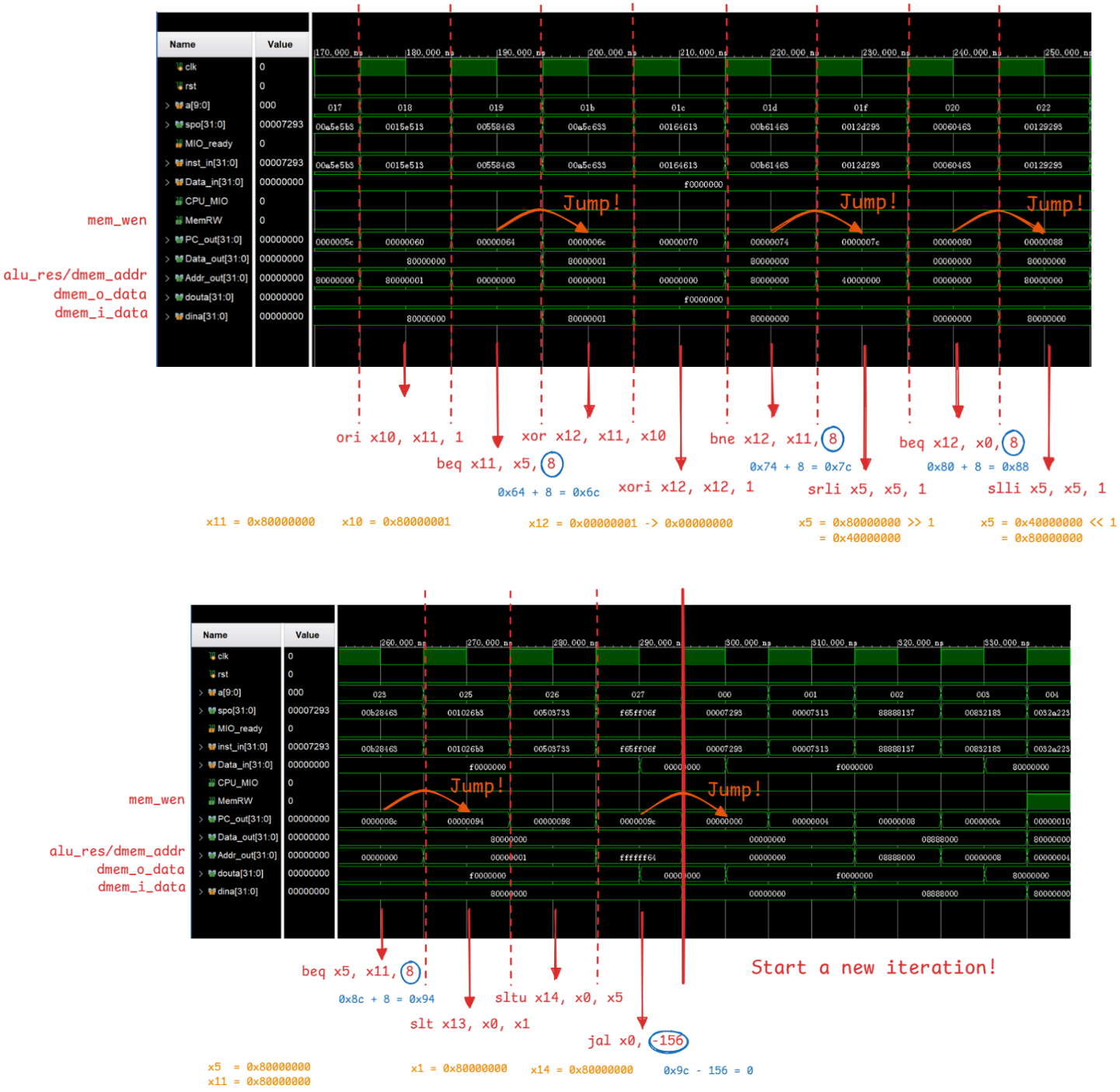

可以看到，波形结果符合预期。

### 5.4.2 SCPU 仿真结果

这里直接使用 `I_more.pdf` 给出的 DEMO 程序对指令扩展版本的 SCPU 进行仿真测试，结果如下所示：


mem_wen
alu_res/dmem_addr
dmem_o_data
dmem_i_data
Jump!
addi x5, x0, 0
lui x2, 559240
sw x3, 4(x5)
lw x7, 28(x0)
add x5, x5, x7
x5 = 0x80000000
andi x6, x0, 0
lw x3, 8(x6)
lw x1, 4(x0)
beq x7, x3, 16
0x1c + 16 = 0x2c
x5 = 0
x6 = 0
x2 = 559240 << 12
= 0x88888000
x3 = 0x80000000
x1 = 0x80000000
x7 = 0x80000000


mem_wen
alu_res/dmem_addr
dmem_o_data
dmem_i_data
Jump!
Jump!
Jump!
add x6, x6, x2
sub x8, x6, x5
jal x0, 8
andi x11, x10, 1
or x11, x11, x10
0x54 + 8 = 0x5c
bne x6, x5, 8
0x34 + 8 = 0x3c
sub x9, x6, x5
0x44 + 8 = 0x4c
and x10, x5, x7
bne x10, x11, 8
x5 = 0x80000000
x6 = 0x88888000
x8 = 0x08888000
x9 = 0x08888000
x10 = 0x80000000
x11 = 0
x11 = 0x80000000




可以看到，波形结果符合预期。

### 5.4.3 SOC 测试数据分析

将烧录好的 bit 流文件传入 NEXYS 板上，通过 VGA 显示器观察结果，与波形图的信号进行比对，发现均符合预期，且经助教验收通过。

# 六、 实验心得

这次实验相比前几次实验难度提升了不少，因为对应的理论知识相比前几次更加复杂，且对我们提出了更高的硬件设计要求。最痛苦的地方还是在于调试：比如对控制器进行仿真后，总会发现某些指令的某几个控制信号和预期的不一样，这时就需要仔细检查源代码的漏洞，可能找了半天才发现原来是错在一个很简单且很不起眼的地方；又比如即使仿真结果正确了，上板验证时发现 VGA 显示的结果不符合预期，经过我反复的排查，最终发现原来是在给 ROM 扩展位宽（之前用的电脑 VGA 显示不了，所以扩了位宽）后输入部分给改错了，导致 ROM 读取错误的内容，于是我还是改回原来的位宽，结果能够正确显示了。由于 Vivado 的综合和实现速度太慢，因此每次修改错误后再生成 bit 流需要耗费不少的时间成本，因此在今后的实验中，我得先熟悉理论知识，然后在编写代码的时候需要小心再小心，确保不要犯过于低级的错误，这样可以使我更高效率地完成实验。