实验十一 RV32I 单周期 CPU

姓名：徐沐杰

学号：211240088

班级：1班

邮箱：litrehinn@smail.nju.edu.cn

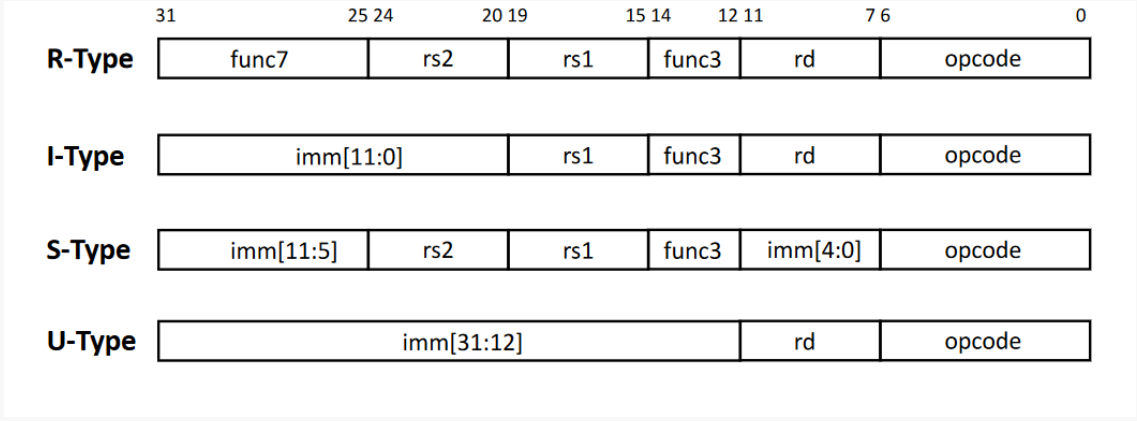
实验时间：2022.12.9

实验十一 RV32I 单周期 CPU

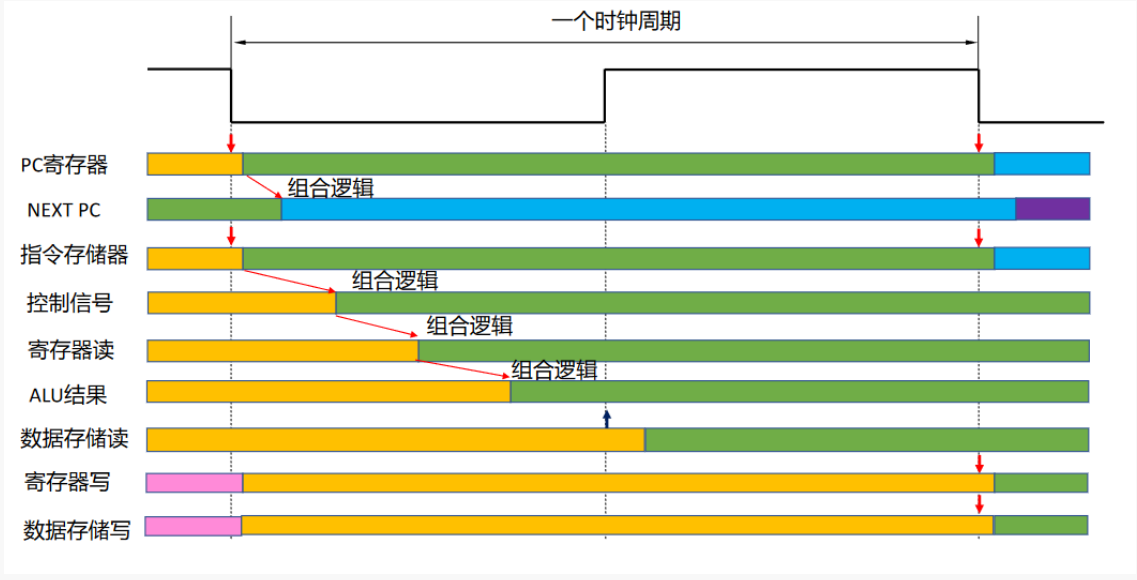
1.实验目的

利用 FPGA 实现 RV32I 指令集中除系统控制指令之外的其余指令。利用单周期方式实现 RV32I 的控制通路及数据通路，并能够顺利通过功能仿真。

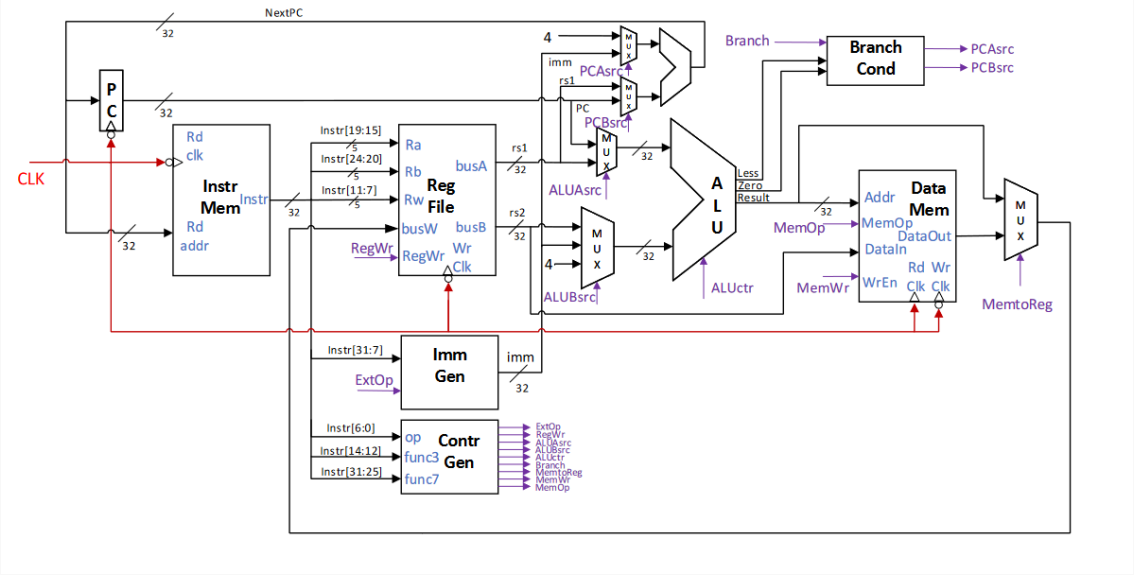
2.实验原理



RV32I 指令 的基本格式



单周期 CPU 原理图



单周期 CPU 设计图

3.实验环境/器材

在线仿真测试平台。

4.程序代码或流程图

oj 通关代码

```
module control_signal_generator (
    input  wire [31:0] instr,
    output wire [2:0]  ExtOp,
    output wire        RegWr,
    output wire        ALUAsrc,
    output wire [1:0]  ALUBsrc,
    output wire [3:0]  ALUctr,
    output wire [2:0]  Branch,
    output wire        MemtoReg,
    output wire        MemWr,
    output wire [2:0]  MemOp,
    output wire [(5)-1:0] rd,
    output wire [(5)-1:0] rs1,
    output wire [(5)-1:0] rs2
);
// ======== Decode ======
    wire [6:0] op    = instr[6:0];
    assign  rs1   = instr[19:15];
    assign  rs2   = instr[24:20];
    assign  rd    = instr[11:7];
    wire [2:0] func3 = instr[14:12];
    wire [6:0] func7 = instr[31:25];
// ======================

    reg [18:0] control_signal;
    assign {ExtOp, RegWr, Branch, MemtoReg, MemWr, MemOp, ALUAsrc, ALUBsrc,
ALUctr} = control_signal;

    always @(*) begin
```

```
        casex ({op[6:2], func3, func7[5]})
            //ARI
            9'b01101xxxx: control_signal = 19'b001_1_000_0_0_000_0_01_0011;
            9'b00101xxxx: control_signal = 19'b001_1_000_0_0_000_1_01_0000;
            9'b00100000x: control_signal = 19'b000_1_000_0_0_000_0_01_0000;
            9'b00100010x: control_signal = 19'b000_1_000_0_0_000_0_01_0010;
            9'b00100011x: control_signal = 19'b000_1_000_0_0_000_0_01_1010;
            9'b00100100x: control_signal = 19'b000_1_000_0_0_000_0_01_0100;
            9'b00100110x: control_signal = 19'b000_1_000_0_0_000_0_01_0110;
            9'b00100111x: control_signal = 19'b000_1_000_0_0_000_0_01_0111;
            9'b001000010: control_signal = 19'b000_1_000_0_0_000_0_01_0001;
            9'b001001010: control_signal = 19'b000_1_000_0_0_000_0_01_0101;
            9'b001001011: control_signal = 19'b000_1_000_0_0_000_0_01_1101;
            9'b011000000: control_signal = 19'b000_1_000_0_0_000_0_00_0000;
            9'b011000001: control_signal = 19'b000_1_000_0_0_000_0_00_1000;
            9'b011000010: control_signal = 19'b000_1_000_0_0_000_0_00_0001;
            9'b011000100: control_signal = 19'b000_1_000_0_0_000_0_00_0010;
            9'b011000110: control_signal = 19'b000_1_000_0_0_000_0_00_1010;
            9'b011001000: control_signal = 19'b000_1_000_0_0_000_0_00_0100;
            9'b011001010: control_signal = 19'b000_1_000_0_0_000_0_00_0101;
            9'b011001011: control_signal = 19'b000_1_000_0_0_000_0_00_1101;
            9'b011001100: control_signal = 19'b000_1_000_0_0_000_0_00_0110;
            9'b011001110: control_signal = 19'b000_1_000_0_0_000_0_00_0111;
            //J
            9'b11011xxxx: control_signal = 19'b100_1_001_0_0_000_1_10_0000;
            9'b11001000x: control_signal = 19'b000_1_010_0_0_000_1_10_0000;
            9'b11000000x: control_signal = 19'b011_0_100_0_0_000_0_00_0010;
            9'b11000001x: control_signal = 19'b011_0_101_0_0_000_0_00_0010;
            9'b11000100x: control_signal = 19'b011_0_110_0_0_000_0_00_0010;
            9'b11000101x: control_signal = 19'b011_0_111_0_0_000_0_00_0010;
            9'b11000110x: control_signal = 19'b011_0_110_0_0_000_0_00_1010;
            9'b11000111x: control_signal = 19'b011_0_111_0_0_000_0_00_1010;
            //L&S
            9'b00000000x: control_signal = 19'b000_1_000_1_0_000_0_01_0000;
            9'b00000001x: control_signal = 19'b000_1_000_1_0_001_0_01_0000;
            9'b00000010x: control_signal = 19'b000_1_000_1_0_010_0_01_0000;
            9'b00000100x: control_signal = 19'b000_1_000_1_0_100_0_01_0000;
            9'b00000101x: control_signal = 19'b000_1_000_1_0_101_0_01_0000;
            9'b01000000x: control_signal = 19'b010_0_000_0_1_000_0_01_0000;
            9'b01000001x: control_signal = 19'b010_0_000_0_1_001_0_01_0000;
            9'b01000010x: control_signal = 19'b010_0_000_0_1_010_0_01_0000;
            default: begin control_signal = 19'b010_0_000_0_1_010_0_01_0000; end
        endcase
    end

endmodule

module imm_generator (
    input  wire [31:0] instr,
    input  wire [2:0]  ExtOp,
    output reg  [31:0] imm
);
    always @(*)
    case (ExtOp)
        3'b000: imm = {{20{instr[31]}}, instr[31:20]}; //I
        3'b001: imm = {instr[31:12], 12'b0}; //U
        3'b010: imm = {{20{instr[31]}}, instr[31:25], instr[11:7]}; //S
```

```
        3'b011: imm = {{20{instr[31]}}, instr[7], instr[30:25], instr[11:8],
1'b0}; //B
        3'b100: imm = {{12{instr[31]}}, instr[19:12], instr[20], instr[30:21],
1'b0}; //J
    endcase
endmodule

module pc_generator(
    input   wire    [(32)-1:0]    pc,             //pcbsrc
    input   wire                  pc_source_sel,  //pcasrc
    input   wire                  pc_offset_sel,
    input   wire    [(32)-1:0]    jalr_reg,
    input   wire    [(32)-1:0]    offset,
    output  wire    [(32)-1:0]    next_pc
);


    wire [(32)-1:0] pc_source = pc_source_sel ? jalr_reg : pc;
    wire [(32)-1:0] pc_offset = pc_offset_sel ? offset : 4;

    assign next_pc = pc_source + pc_offset;

endmodule

module inst_memory (
    input  wire clk,
    input  wire [31:0] addr,
    output reg [31:0] instr
);
    parameter instr_size = 1000;
    reg [31:0] instr_mem [instr_size:0];
    initial
    begin
        $readmemh("", instr_mem);
    end
    always @(negedge clk)
        instr <= instr_mem[addr];
endmodule

module data_memory(
    input  [31:0] addr,
    output reg [31:0] dataout,
    input  [31:0] datain,
    input  rdclk,
    input  wrclk,
    input [2:0] memop,
    input we
);
//add your code here
    wire [31:0] unit;
    wire [1:0] offset;
    wire [31:0] data;
    reg [31:0] mem [1000:0];
    assign unit = addr / 4;
    assign offset = addr % 4;
    assign data = mem[unit];
    integer i;
    always @(posedge rdclk)
```

```
    begin
        if(we == 1'b0)
        begin
            case(memop)
            3'b000:
                begin
                    if(data[offset * 8 + 7] == 1'b1)
                        dataout[31:8] = {24{1'b1}};
                    else
                        dataout[31:8] = {24{1'b0}};
                    case(offset)
                        2'b00: dataout[7:0] = data[7:0];
                        2'b01: dataout[7:0] = data[15:8];
                        2'b10: dataout[7:0] = data[23:16];
                        2'b11: dataout[7:0] = data[31:24];
                    endcase
                end
            3'b001:
                begin
                    if(data[offset * 8 + 15] == 1'b1)
                        dataout[31:16] = {16{1'b1}};
                    else
                        dataout[31:16] = {16{1'b0}};
                    case(offset)
                        2'b00: dataout[15:0] = data[15:0];
                        2'b10: dataout[15:0] = data[31:16];
                    endcase
                end
            3'b010:
                begin
                    dataout[31:0] = data;
                end
            3'b100:
                begin
                    dataout[31:8] = {24{1'b0}};
                    case(offset)
                        2'b00: dataout[7:0] = data[7:0];
                        2'b01: dataout[7:0] = data[15:8];
                        2'b10: dataout[7:0] = data[23:16];
                        2'b11: dataout[7:0] = data[31:24];
                    endcase
                end
            3'b101:
                begin
                    dataout[31:16] = {16{1'b0}};
                    case(offset)
                        2'b00: dataout[15:0] = data[15:0];
                        2'b10: dataout[15:0] = data[31:16];
                    endcase
                end
            endcase
        end
    end
    always @(posedge wrclk)
    begin
        if(we == 1'b1)
        begin
            case(memop)
```

```
            3'b000:
                begin
                     case(offset)
                        2'b00: mem[unit][7:0] = datain[7:0];
                        2'b01: mem[unit][15:8] = datain[7:0];
                        2'b10: mem[unit][23:16] = datain[7:0];
                        2'b11: mem[unit][31:24] = datain[7:0];
                    endcase
                end
            3'b001:
                begin
                    case(offset)
                        2'b00: mem[unit][15:0] = datain[15:0];
                        2'b10: mem[unit][31:16] = datain[15:0];
                    endcase
                end
            3'b010:
                begin
                    mem[unit][31:0] = datain[31:0];
                end
            endcase
        end
    end
endmodule

module regheap (
    input  wire WrClk,
    input  wire [4:0] Ra,
    input  wire [4:0] Rb,
    input  wire [4:0] Rw,
    input  wire        RegWr,
    input  wire [31:0] busW,
    output wire [31:0] busA,
    output wire [31:0] busB
);
    integer i;

    reg [31:0] regs [31:0];
    initial begin
        for (i = 0; i < 32; i = i + 1)
            regs[i] = 0;
    end
    assign busA = regs[Ra];
    assign busB = regs[Rb];
    always @(negedge WrClk)
    begin
        if(RegWr && Rw)
            regs[Rw] <= busW;
    end

endmodule

module ALU(
    input [31:0] dataa,
    input [31:0] datab,
    input [3:0]  ALUctr,
    output reg less,
    output reg zero,
```

```
    output reg [31:0] aluresult
);
//add your code here
    reg [31:0] reverse;
    reg [31:0] temp;
    reg carry;
    reg of;
    wire [4:0] offset;
    assign offset = datab[4:0];
    always @(*)
    begin
        case(ALUctr[2:0])
        3'b000:
            begin
                if(ALUctr[3] == 1'b0)
                begin
                    {carry, aluresult} = dataa + datab;
                    zero = (!aluresult);
                end
                else
                begin
                    {carry, aluresult} = dataa + ~datab + 1;
                    zero = (!aluresult);
                end
            end
        3'b001:
            begin
                aluresult = dataa << datab[4:0];
                zero = (!aluresult);
            end
        3'b010:
            begin
                if(ALUctr[3] == 1'b0)
                begin
                    reverse = ~datab;
                    {carry, aluresult} = dataa + reverse + 1;
                    of = (dataa[31] == reverse[31]) && (aluresult [31] !=
dataa[31]);
                    less = (aluresult[31] ^ of) ? 1'b1 : 1'b0;
                    aluresult = less;
                end
                else
                begin
                    reverse = ~datab;
                    {carry, aluresult} = dataa + reverse + 1;
                    less = (carry ^ 1) ? 1'b1 : 1'b0;
                    aluresult = less;
                end
                zero = (dataa == datab);
            end
        3'b011:
            begin
                aluresult = datab;
                zero = (!aluresult);
            end
        3'b100:
            begin
                aluresult = dataa ^ datab;
```

```
                zero = (!aluresult);
            end
        3'b101:
            begin
                if(ALUctr[3] == 1'b0)
                begin
                    temp = offset[0] ? {32'b0, dataa[31:1]} : dataa;
                    temp = offset[1] ? {32'b0, temp[31:2]} : temp;
                    temp = offset[2] ? {32'b0, temp[31:4]} : temp;
                    temp = offset[3] ? {32'b0, temp[31:8]} : temp;
                    temp = offset[4] ? {32'b0, temp[31:16]} : temp;
                end
                else
                begin
                    temp = offset[0] ? {dataa[31], dataa[31:1]} : dataa;
                    temp = offset[1] ? {{2{temp[31]}}, temp[31:2]} : temp;
                    temp = offset[2] ? {{4{temp[31]}}, temp[31:4]} : temp;
                    temp = offset[3] ? {{8{temp[31]}}, temp[31:8]} : temp;
                    temp = offset[4] ? {{16{temp[31]}}, temp[31:16]} : temp;
                end
                aluresult = temp;
                zero = (!aluresult);
            end
        3'b110:
            begin
                aluresult = dataa | datab;
                zero = (!aluresult);
            end
        3'b111:
            begin
                aluresult = dataa & datab;
                zero = (!aluresult);
            end
        endcase
    end
endmodule

module jump_control(
    input   wire    [2:0]   branch,
    input   wire            zero,
    input   wire            less,
    output  wire            pc_source_sel,
    output  wire            pc_offset_sel
);

    reg [1:0] pc_sel;
    assign {pc_source_sel, pc_offset_sel} = pc_sel;

    always @(*)
        casex ({branch, zero, less})
            5'b000xx: pc_sel = 2'b00;
            5'b001xx: pc_sel = 2'b01;
            5'b010xx: pc_sel = 2'b11;
            5'b1000x: pc_sel = 2'b00;
            5'b1001x: pc_sel = 2'b01;
            5'b1010x: pc_sel = 2'b01;
            5'b1011x: pc_sel = 2'b00;
            //to-do
```

```
            5'b110x0: pc_sel = 2'b00;
            5'b110x1: pc_sel = 2'b01;
            5'b111x0: pc_sel = 2'b01;
            5'b111x1: pc_sel = 2'b00;
        endcase

endmodule

module CPU(
    input   wire                    clk,
    input   wire                    rst,
    output  wire    [(32)-1:0]    inst_read_addr,
    input   wire    [(32)-1:0]    inst_read_data,
    output  wire                    inst_read_en,
    output  wire    [(32)-1:0]    data_read_addr,
    output  wire    [(32)-1:0]    data_write_addr,
    input   wire    [(32)-1:0]    data_read_data,
    output  wire    [(32)-1:0]    data_write_data,
    output  wire                    data_read_en,
    output  wire                    data_write_en,
    output  wire    [2:0]           data_mode
);

    reg [(32)-1:0] pc;
    wire [(32)-1:0] next_pc, jalr_reg, offset;
    wire pc_offset_sel, pc_source_sel;
    wire [(32)-1:0] inst = inst_read_data;
    wire RegWr, ALUAsrc, MemtoReg, MemWr;
    wire [2:0] Branch, MemOp, ExtOp;
    wire [1:0] ALUBsrc;
    wire [3:0] ALUctr;
    wire [(5)-1:0] Ranum, Rbnum, Rwnum;
    wire [(32)-1:0] Ra, Rb;
    wire [(32)-1:0] imm;

    pc_generator PG (
        .next_pc(next_pc),
        .pc_offset_sel(pc_offset_sel),
        .pc_source_sel(pc_source_sel),
        .pc(pc),
        .jalr_reg(Ra),
        .offset(imm)
    );

    always @(negedge clk, posedge rst)
        if (rst) pc <= 0;
        else pc <= next_pc;

    assign inst_read_addr = next_pc;

    control_signal_generator CSG(
        .instr(inst),
        .ALUAsrc(ALUAsrc),
        .ALUBsrc(ALUBsrc),
        .ALUctr(ALUctr),
        .Branch(Branch),
        .MemtoReg(MemtoReg),
        .MemWr(MemWr),
```

```
        .MemOp(MemOp),
        .ExtOp(ExtOp),
        .RegWr(RegWr),
        .rs1(Ranum),
        .rs2(Rbnum),
        .rd(Rwnum)
    );

    imm_generator IG(
        .instr(inst),
        .ExtOp(ExtOp),
        .imm(imm)
    );

    wire [(32)-1:0] aluresult;

    wire [(32)-1:0] dataa = ALUAsrc ? pc : Ra;
    reg [(32)-1:0] datab;

    regheap RH(
        .WrClk(clk),
        .RegWr(RegWr),
        .Ra(Ranum),
        .Rb(Rbnum),
        .Rw(Rwnum),
        .busW(MemtoReg ? data_read_data : aluresult),
        .busA(Ra),
        .busB(Rb)
    );

    always @(*)
        case (ALUBsrc)
            2'b01: datab = imm;
            2'b10: datab = 4;
            2'b00: datab = Rb;
        endcase

    wire less, zero;

    ALU alu(
        .dataa(dataa),
        .datab(datab),
        .ALUctr(ALUctr),
        .less(less),
        .zero(zero),
        .aluresult(aluresult)
    );

    jump_control JC(
        .branch(Branch),
        .less(less),
        .zero(zero),
        .pc_offset_sel(pc_offset_sel),
        .pc_source_sel(pc_source_sel)
    );

    assign data_write_en = MemWr;
    assign data_read_en = MemtoReg;
```

```
    assign inst_read_en = 1;

    assign data_mode = MemOp;
    assign data_write_addr = aluresult;
    assign data_write_data = Rb;

endmodule



module rv32is(
    input   clock,
    input   reset,
    output [31:0] imemaddr,
    input  [31:0] imemdataout,
    output  imemclk,
    output [31:0] dmemaddr,
    input  [31:0] dmemdataout,
    output [31:0] dmemdatain,
    output  dmemrdclk,
    output  dmemwrclk,
    output [2:0] dmemop,
    output  dmemwe,
    output [31:0] dbgdata
);

    assign imemclk   = ~clock;
    assign dmemrdclk = clock;
    assign dmemwrclk = ~clock;

    wire clk = clock;
    wire rst = reset;
    wire [31:0] inst_read_data = imemdataout, data_read_data = dmemdataout;
    wire [31:0] inst_read_addr, data_read_addr, data_write_addr,
data_write_data;
    reg [31:0] pc = 0;
    wire data_write_en;
    wire [2:0] data_mode;

    wire [(32)-1:0] wire_next_pc, jalr_reg, offset;
    wire pc_offset_sel, pc_source_sel;
    wire [(32)-1:0] inst = inst_read_data;
    wire RegWr, ALUAsrc, MemtoReg, MemWr;
    wire [2:0] Branch, MemOp, ExtOp;
    wire [1:0] ALUBsrc;
    wire [3:0] ALUctr;
    wire [(5)-1:0] Ranum, Rbnum, Rwnum;
    wire [(32)-1:0] Ra, Rb;
    wire [(32)-1:0] imm;

    wire [(32)-1:0] next_pc;

    pc_generator PG (
        .next_pc(wire_next_pc),
        .pc_offset_sel(pc_offset_sel),
        .pc_source_sel(pc_source_sel),
        .pc(pc),
        .jalr_reg(Ra),
```

```
        .offset(imm)
    );

    assign next_pc = rst ? 0 : wire_next_pc;

    always @(negedge clk, posedge rst)
        if (rst) pc <= 0;
        else pc <= next_pc;

    assign inst_read_addr = next_pc;

    control_signal_generator CSG(
        .instr(inst),
        .ALUAsrc(ALUAsrc),
        .ALUBsrc(ALUBsrc),
        .ALUctr(ALUctr),
        .Branch(Branch),
        .MemtoReg(MemtoReg),
        .MemWr(MemWr),
        .MemOp(MemOp),
        .ExtOp(ExtOp),
        .RegWr(RegWr),
        .rs1(Ranum),
        .rs2(Rbnum),
        .rd(Rwnum)
    );

    imm_generator IG(
        .instr(inst),
        .ExtOp(ExtOp),
        .imm(imm)
    );

    wire [(32)-1:0] aluresult;

    wire [(32)-1:0] dataa = ALUAsrc ? pc : Ra;
    reg [(32)-1:0] datab;

    regheap myregfile(
        .WrClk(clk),
        .RegWr(RegWr),
        .Ra(Ranum),
        .Rb(Rbnum),
        .Rw(Rwnum),
        .busW(MemtoReg ? data_read_data : aluresult),
        .busA(Ra),
        .busB(Rb)
    );

    always @(*)
        case (ALUBsrc)
            2'b01: datab = imm;
            2'b10: datab = 4;
            2'b00: datab = Rb;
        endcase

    wire less, zero;
```

```
    ALU alu(
        .dataa(dataa),
        .datab(datab),
        .ALUctr(ALUctr),
        .less(less),
        .zero(zero),
        .aluresult(aluresult)
    );

    jump_control JC(
        .branch(Branch),
        .less(less),
        .zero(zero),
        .pc_offset_sel(pc_offset_sel),
        .pc_source_sel(pc_source_sel)
    );

    assign data_write_en = MemWr;

    assign data_mode = MemOp;
    assign data_read_addr = aluresult;
    assign data_write_addr = aluresult;
    assign data_write_data = Rb;


    assign dmemwe = data_write_en;
    assign dmemop = data_mode;
    assign dbgdata = pc;
    assign imemaddr = inst_read_addr;
    assign dmemaddr = MemWr ? data_write_addr : data_read_addr;
    assign dmemdatain = data_write_data;
endmodule
```

5.实验步骤/过程

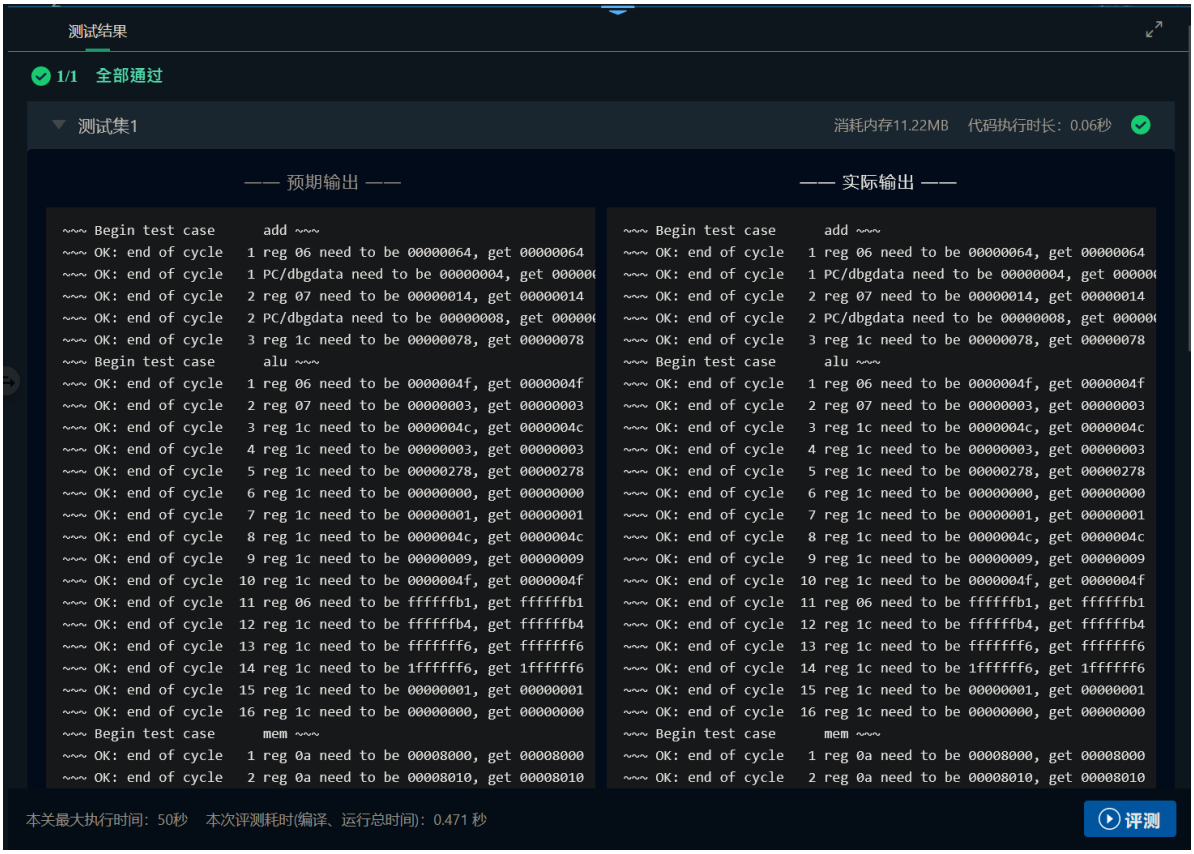
在本地分模块编写代码，用仿真编译器预编译命令编译合成后进行在线仿真测试。

6.测试方法

在线仿真测试。

7.实验结果

我通过了在线仿真测试。



8.实验中遇到的问题及解决办法

- pc 早期一直是 0，后来是发现在复位过程中没有给 next_pc 赋初值。
- 跳转指令执行异常，因为在给ALU的输入端赋值时应该使用 pc 而非 next_pc。

9.实验得到的启示

需要注意在给不同的部件赋值时，信号的时序关系已经不同了。

10.意见和建议

无