单周期 CPU 设计

李卓 pb19000064

实验目的

理解 CPU 的结构和工作原理

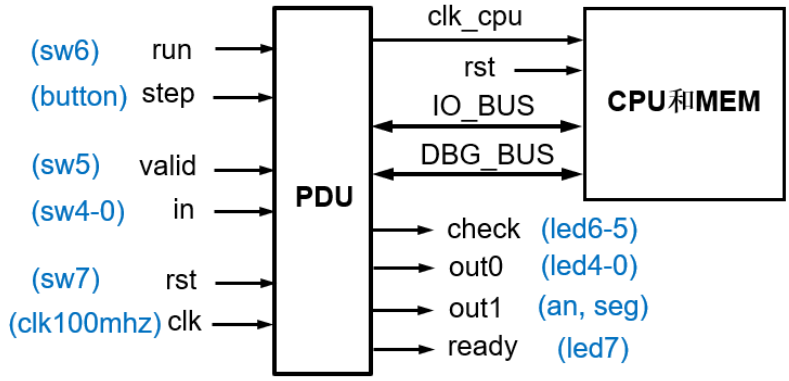
掌握单周期 CPU 的设计和调试方法

熟练掌握数据通路和控制器的设计和描述方法

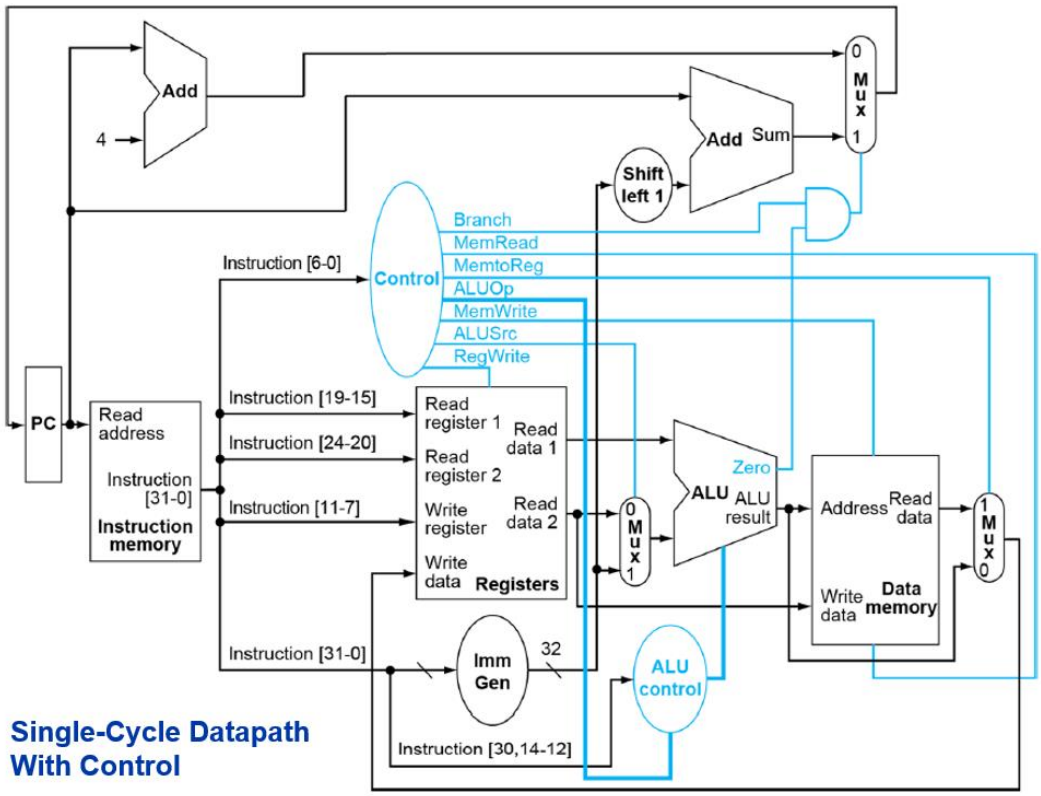
实验原理

设计实现单周期 RISC-V CPU，可执行以下 10 条指令

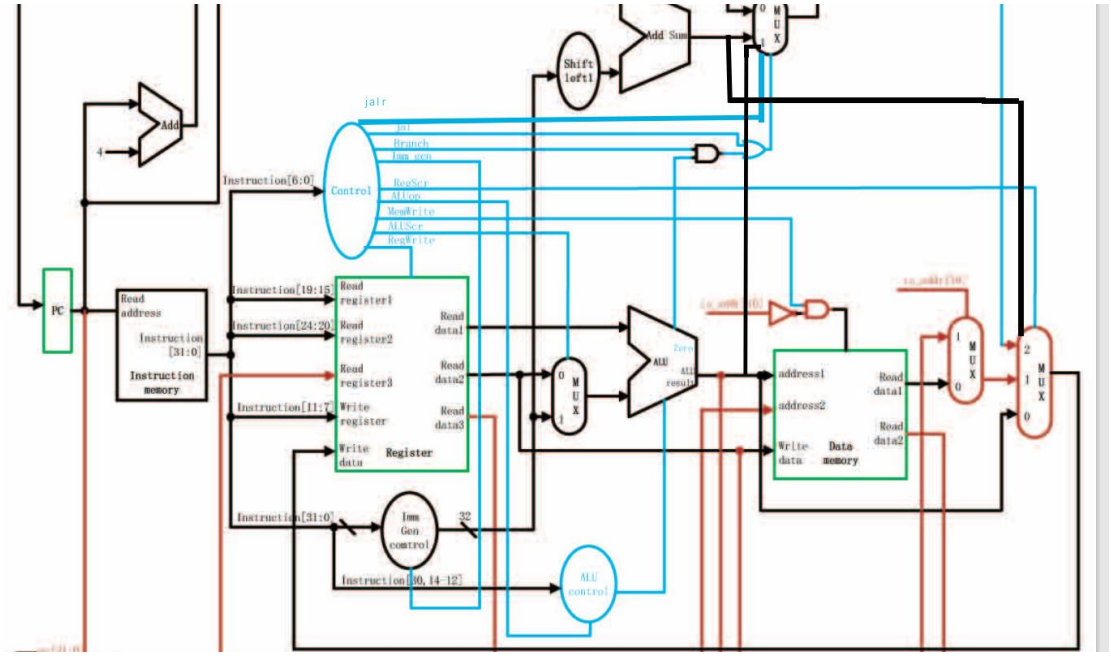
- add, addi, sub, auipc, lw, sw, beq, blt, jal, jalr



单周期 CPU 数据通路



该通路只能实现原有的六条指令，对于新加的四条指令(auipc, blt, jalr sum),需要修改通路。



sub：不需特殊处理

auipc：

连接 regscr 的 mux 和 pc_adder。执行时 pc_adder 将 pc 和立即数相加，RegSrc=11，mux 选择 pc adder 的结果 sum

blt:

根据 funct3 位判断 blt 指令。alu_control 向 alu 发送操作符 LT:101，alu 将两数相减 最高位为一则说明小于，zero 置为 1，其他与 beq 指令完全相同

jalr：

control unit 增加控制信号 jalr。连接 alu 的结果与 pc_mux。执行时，jalr 为 1，pc_mux 输入为 3，选择 alu 相加的结果写入 pc。RegSrc=10，原 pc+4 的结果写入寄存器中

io_bus 信号

CPU 运行时访问开关(sw)、指示灯(led)和数码管(an, seg)

- io_addr：I/O 外设的地址
- io_din：CPU 接收来自输入缓冲寄存器（IBR）的 sw 输入数据
- io_dout：CPU 向 led 和 seg 输出的数据
- io_we：CPU 向 led 和 seg 输出时的使能信号，利用该信号将 io_dout 存入输出缓冲寄存器（OBR），再经数码管显示电路将其显示在数码管（an，seg）

debug_bus 信号

调试时将存储器和寄存器堆内容，以及 CPU 数据通路状态信息导出显示

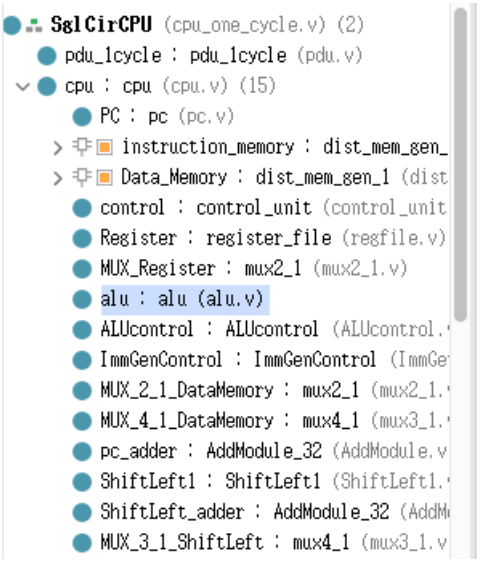
- m_rf_addr：存储器(MEM)或寄存器堆(RF)的调试读口地址
- rf_data：从 RF 读取的数据
- m_data：从 MEM 读取的数据
- pc：PC 的内容

pdu 运行方式：

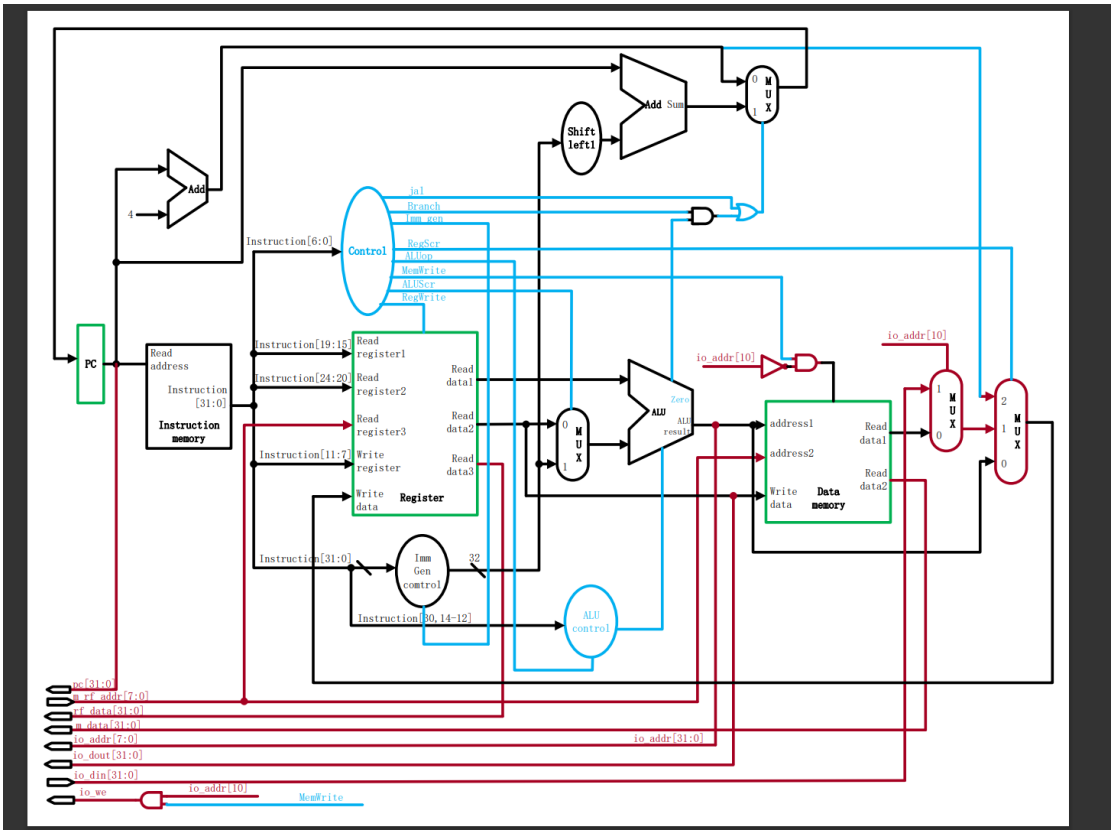
sw				button	led			an/seg	说明
7	6	5	4~0		7	6~5	4~0		
rst	run	valid	in	step	ready	check	out0	out1	
↑	-	-	-	-	1	00	0x1f	0x12…8	复位
x	1	valid	in	-	ready	00	out0	out1	连续运行
	0	valid	in	↑	ready	00	out0	out1	单步运行
		↑↓	addr_rf	x	0	01	addr_rf	data_rf	查看寄存器堆
			addr_m		0	10	addr_m	data_m	查看存储器
			-		0	11	0	PC	查看PC

实验过程

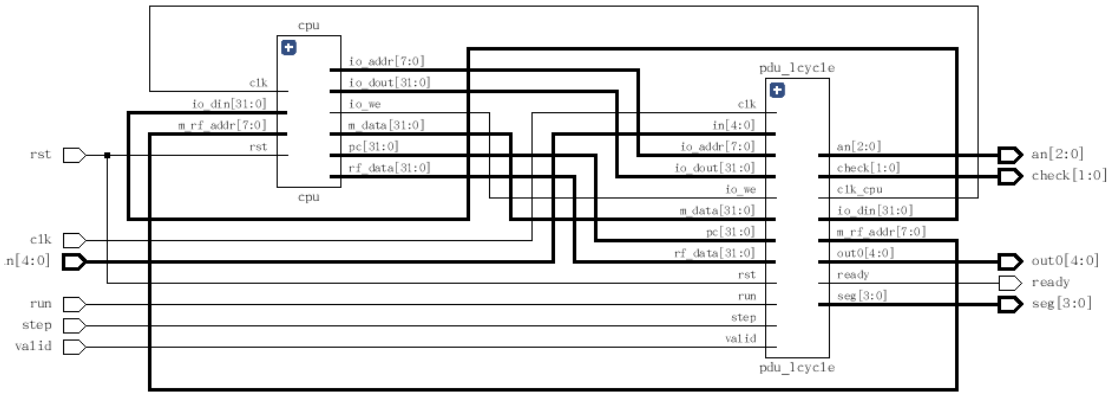
源文件结构：



各个模块定义如下图



电路



下载测试

FibFrMem.s

FibFrMem_IM.coe

FibFrMem_DM.coe

FibFrMem.bit

由 Mem 生成 fib 数列，分别为 汇编源文件，指令段 coe，数据段 coe，下载结果

FibFrIO.bit

由 IO 生成 fib 数列 的下载结果，指令段 coe 用老师给的

FibFrIO_IM.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
memory_initialization_radix  = 16;
memory_initialization_vector =
00100093
00000333
40102223
41002283
fe028ee3
40c02403
40802423
00832023
00430313
40002223
41002283
fe128ee3
40102223
41002283
fe028ee3
40c02483
40902423
00932023
00430313
40002223
41002283
fe128ee3
009402b3
40502423
00532023
00430313
00900433
005004b3
40102223
41002283
fe028ee3
40002223
```

FibFrMem.s*

```
.text
main:
# initial variables
lw s1 0(x0)
lw s2 1(x0)
# counter s4
li s4 1
# jump to fibo function
jal s10 fibo
# exit
exit:
add x0 x0 x0
jal t3,exit

fibo:
# add 2 values to s6 reg ( s2+s1->s6)
add s6 s1 s2
# data transfer ( s2->s1, s6->s2)
sw s2 8(s7)
lw s1 8(s7)
sw s6 8(s7)
lw s2 8(s7)
sw s6, 0x408(x0)  #out1=f0
# add counter value
addi s4 s4 1
# check if the calculate cycles reach
li a5 2
jal s8,fibo
```

FibFrMem_IM.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(

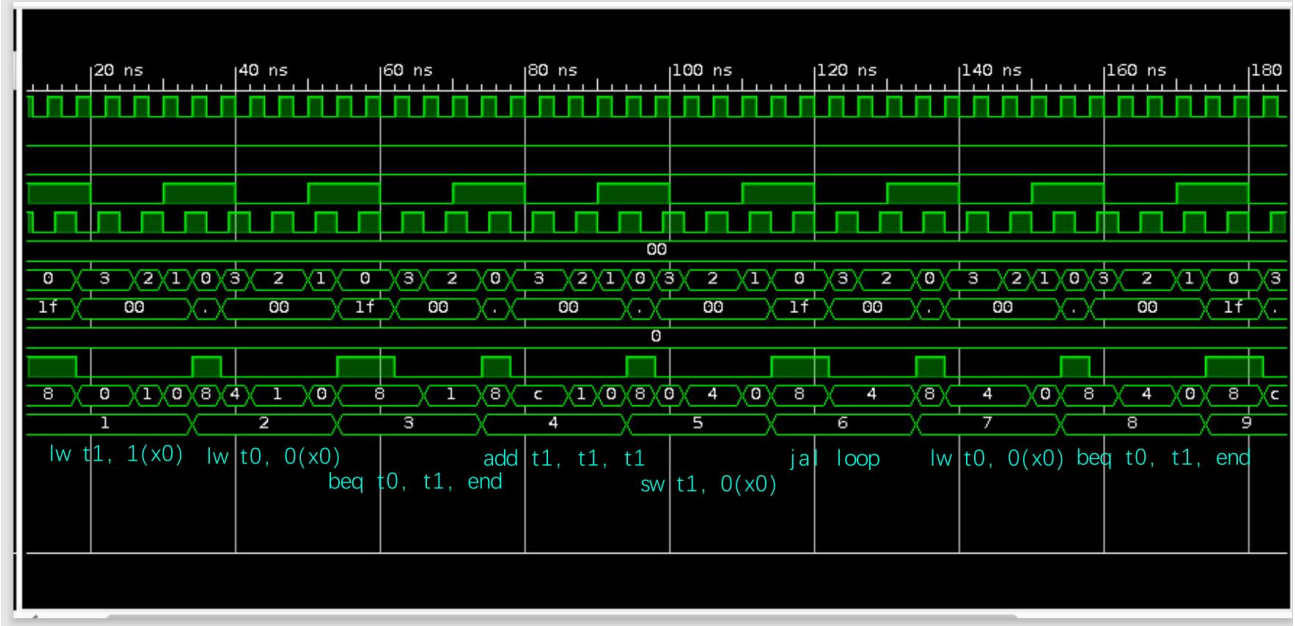
```
memory_initialization_radix = 16;
memory_initialization_vector =
00002483
00102903
00100a13
00c00d6f
00000033
ffdffe6f
01248b33
012ba423
008ba483
016ba423
008ba903
41602423
001a0a13
00200793
fe1ffc6f
```

仿真测试：

test1.s

```
lw t1, 4(x0)
loop:
lw t0, 0(x0)
beq t0, t1, end
add t1, t1, t1
sw t1, 0(x0)
jal loop
end:
```

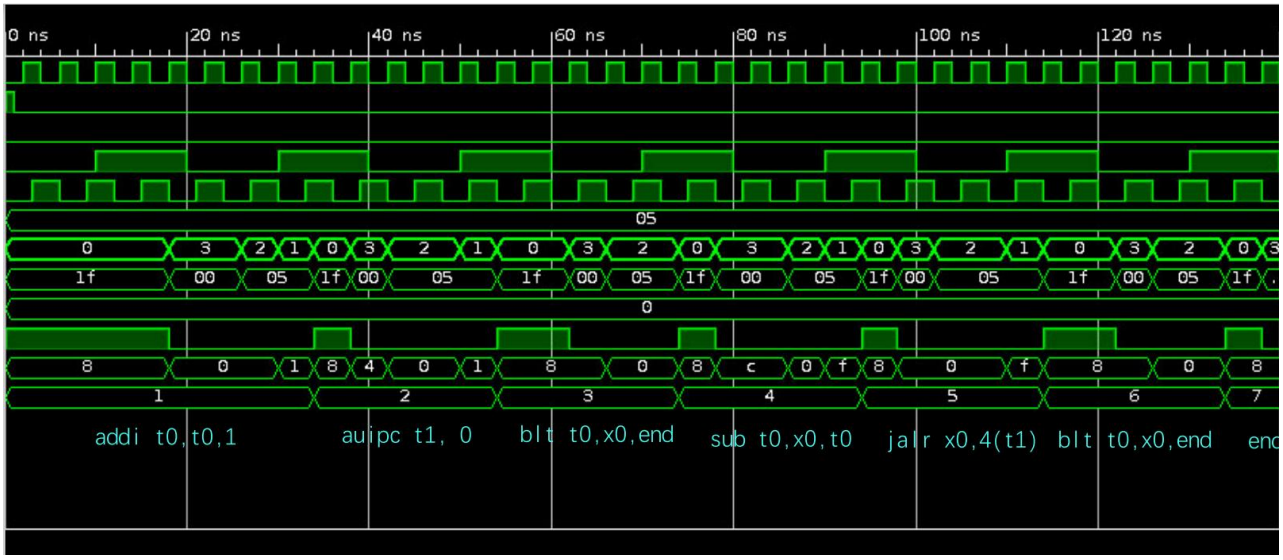
生成的波形图 *wave1.pdf*



test2.s

```
addi t0, x0, 1
auipc t1, 0
blt t0, x0, end
sub t0, x0, t0
jalr x0, 4(t1)
end:
```

生成的波形图 *wave2.pdf*



实验代码

SglCirCpu.v

```
module SglCirCPU(
        input run,    //sw6
        input step, //button
        input valid,//sw5
        input [4:0]in,//sw4-0
        input rst,    //sw7
        input clk,    //clk100mhz
        output [1:0]check, //led6-5
        output [4:0] out0,       //led4-0
        output [2:0] an,         //8 个数码管
        output [3:0] seg,
        output ready                 //led7
        );
        wire clk_cpu,io_we;
        wire[7:0]io_addr,m_rf_addr;
        wire[31:0]io_dout,io_din,rf_data,m_data,pc;

pdu_1cycle pdu_1cycle(
    .clk(clk),
    .rst(rst),
    //选择 CPU 工作方式;
    .run(run),
    .step(step),
    .clk_cpu(clk_cpu),
    //输入 switch 的端口
    .valid(valid),
    .in(in),
    //输出 led 和 seg 的端口
    .check(check),    //led6-5:查看类型
    .out0(out0),       //led4-0
    .an(an),         //8 个数码管
    .seg(seg),
    .ready(ready),                 //led7
    //IO_BUS
    .io_addr(io_addr),
    .io_dout(io_dout),
    .io_we(io_we),
    .io_din(io_din),

    //Debug_BUS
    .m_rf_addr(m_rf_addr),
    .rf_data(rf_data),
    .m_data(m_data),
    .pc(pc)
);
cpu cpu (
          .clk(clk_cpu),
          .rst(rst),
          //IO_BUS
          .io_addr(io_addr),          //led 和 seg 的地址
          .io_dout(io_dout),        //输出 led 和 seg 的
数据
          .io_we(io_we),                          //输出 led
和 seg 数据时的使能信号
          .io_din(io_din),                //来自 sw 的输入
```

pdu.v

```
module    pdu_1cycle(
    input clk,
    input rst,

    //选择 CPU 工作方式;
    input run,
    input step,
    output clk_cpu,

    //输入 switch 的端口
    input valid,
    input [4:0] in,

    //输出 led 和 seg 的端口
    output [1:0] check,    //led6-5:查看类型
    output [4:0] out0,       //led4-0
    output [2:0] an,         //8 个数码管
    output [3:0] seg,
    output ready,                  //led7

    //IO_BUS
    input [7:0] io_addr,
    input [31:0] io_dout,
    input io_we,
    output [31:0] io_din,

    //Debug_BUS
    output [7:0] m_rf_addr,
    input [31:0] rf_data,
    input [31:0] m_data,
    input [31:0] pc
);

reg [4:0] in_r;       //同步外部输入用
reg run_r, step_r, step_2r, valid_r, valid_2r;
wire step_p, valid_pn;    //取边沿信号

reg clk_cpu_r;          //寄存器输出 CPU 时钟
reg [4:0] out0_r;     //输出外设端口
reg [31:0] out1_r;
reg ready_r;
reg [19:0] cnt;         //刷新计数器，刷新频率约为
95Hz
reg [1:0] check_r;   //查看信息类型, 00-运行结果,
01-寄存器堆，10-存储器，11-PC

reg [7:0] io_din_a; //_a 表示为满足组合 always 描述
要求定义的，下同
reg ready_a;
reg [4:0] out0_a;
reg [31:0] out1_a;
reg [3:0] seg_a;

assign clk_cpu = clk_cpu_r;
assign io_din = io_din_a;
assign check = check_r;
assign out0 = out0_a;
assign ready = ready_a;
assign seg = seg_a;
assign an = cnt[19:17];
assign step_p = step_r & ~step_2r;        //取上升沿
assign valid_pn = valid_r ^ valid_2r;   //取上升沿或
下降沿
assign m_rf_addr = {{3{1'b0}}, in_r};
```

cpu.v

```
module    cpu (
    input clk,
    input rst,

    //IO_BUS
    output  [7:0] io_addr,          //led 和 seg 的
地址
    output  [31:0] io_dout,         //输出  led  和
seg 的数据
    output  io_we,                          //输出
led 和 seg 数据时的使能信号
    input [31:0] io_din,                //来自 sw 的
输入数据

    //Debug_BUS
    input [7:0] m_rf_addr,     //存储器(MEM)或
寄存器堆(RF)的调试读口地址
    output [31:0] rf_data,      //从 RF 读取的数
据
    output [31:0] m_data,       //从 MEM 读取的
数据
    output [31:0] pc                    //PC 的内容
);
    wire        [31:0]        PC_in,        PC_out,
PC_plus_4_new,PC_plus_4,        PC_NotJump,
WriteData,        MemData,        MDR;        //
PC_plus_4=PC+4;    PC_out  当前地址  PC_in
下一个指令
    wire zf, RegWrite, ALUSrc, PC_en, RegDst,
Jump,    Branch,    MemRead,    MemtoReg,
MemWrite,ALU_zero;
    wire                                    [31:0]
ReadData1_Data_Memory,ShiftLeft_Output,S
hiftLeft_plus_pc,MUX_3_1_out_DataMemory,
MUXout_DataMemory;
    wire                                    [31:0]
INS,ImmgenControl_Output,JumpAddr,
BranchAddr,        ALU_result,        ReadData1,
ReadData2, ALUsrcB, Imm,ReadData3; // INS
指令  wire [4:0] WriteReg; wire [2:0] ALUop;
    wire[1:0]ALUop;
    wire[2:0]ALUfunc;
    wire jal,jalr;
    wire Imm_gen;
    wire [1:0]RegSrc;
    assign PC_plus_4_new = pc + 32'd4;
    // wire MemWrite;
    // wire ALUSrc;
    // wire RegWrite;
    assign                  io_we                  =
MemWrite&&(~ALU_result[10]);
```

control_unit.v

```
module control_unit(
    input [6:0] instruction, // aka. op code
    // output reg jal, Branch, reg Imm_gen,
reg [1:0]RegSrc, reg MemWrite, reg
ALUSrc,reg RegWrite,
    // remove imm_gen
    output reg jalr,jal, Branch, reg
[1:0]RegSrc, reg MemWrite, reg ALUSrc,reg
RegWrite,
    output reg [1:0] ALUop
    );
    always@(*)
    case (instruction)
        7'b0110011://add sub
        begin
            jalr=0;
            jal=0;
            Branch=0;
            RegSrc=2'b00;
            ALUop=2'b00;
            MemWrite=0;
            ALUSrc=0;
            RegWrite=1;
        end
        7'b0010011://addi
        begin
            jalr=0;
            jal=0;
            Branch=0;
            RegSrc=2'b00;
            ALUop=2'b00;
            MemWrite=0;
            ALUSrc=1;
            RegWrite=1;
        end
        7'b1101111://jal
        begin
            jalr=0;
            jal=1;
            Branch=0;
            RegSrc=2'b10;
```

register_file.v

```
module register_file #(parameter WIDTH =
32)(
    input clk,
    input [4:0]ra0,
    output[WIDTH-1:0]rd0,
    input[4:0]ra1,
    output[WIDTH-1:0]rd1,
    input[4:0]wa,
    input we,
    input[WIDTH-1:0]wd,
    input[7:0]ra2,// didn't connect anything
    output[WIDTH-1:0]rd_debug
);

// reg [3:0] regfile[0:7];
reg [WIDTH-1:0] regfile[31:0];

assign rd0 = regfile[ra0],
            rd1 = regfile[ra1];

assign rd_debug = regfile[ra2];


always  @ (posedge  clk) begin
    if (we)  begin
        if(wa == 0)begin
            regfile[wa]<=0;
        end
        else begin
            regfile[wa]  <=  wd;
        end
    end
end

endmodule
```