# 中国科学院大学计算机组成原理实验课

# 实 验 报 告

学号：2018K8009929021 姓名：袁欣怡 专业：计算机科学与技术

实验序号：prj4 实验名称：RISC-V 指令集处理器实现

一、 逻辑电路结构与仿真波形的截图及说明（比如关键 RTL 代码段{包含注释}及其对应的逻辑电路结构、相应信号的仿真波形和信号变化的说明等）

1. Reg_file 部分

总思路和 mips 处理器一样，唯一的不同是 rst 信号拉高时需要将寄存器堆全部初始化为 0。

```
reg [31:0]r[31:0];

always@(posedge clk)
begin
        if (rst)
        begin
                r[0]<=0;r[1]<=0;r[2]<=0;r[3]<=0;
                r[4]<=0;r[5]<=0;r[6]<=0;r[7]<=0;
                r[8]<=0;r[9]<=0;r[10]<=0;r[11]<=0;
                r[12]<=0;r[13]<=0;r[14]<=0;r[15]<=0;
                r[16]<=0;r[17]<=0;r[18]<=0;r[19]<=0;
                r[20]<=0;r[21]<=0;r[22]<=0;r[23]<=0;
                r[24]<=0;r[25]<=0;r[26]<=0;r[27]<=0;
                r[28]<=0;r[29]<=0;r[30]<=0;r[31]<=0;
        end
        if (wen)
        begin
                if (waddr==0) r[0]<=0;
                else r[waddr]<=wdata;
        end
end

assign rdata1=r[raddr1];
assign rdata2=r[raddr2];
```

2. ALU 部分

和 mips 相同，设置了八种运算，分别是：

000:按位与；001:按位或；010:加法；011:按位异或；

100:右移；101:无符号数比较；110:减法；111:有符号数比较。

```
assign ResultAND=A & B;

assign ResultOR=A | B;

assign ResultXOR=A ^ B;

assign SIGNED=~{B[31],B}+33'd1;
assign SIGNED1=(ALUop==3'b010)?{B[31],B}:SIGNED;
assign UNSIGNED=~{1'b0,B}+33'd1;
assign UNSIGNED1=(ALUop==3'b010)?{1'b0,B}:UNSIGNED;

assign temp={32'hffffffff,A}>>B;
assign ResultRight=(A[31]==1)?temp[31:0]:A>>B;

assign calculate={A[31],A}+SIGNED1;
assign calculate1={1'b0,A}+UNSIGNED1;

assign Overflow=calculate[32]^calculate[31];
assign CarryOut=calculate1[32];

assign Result=(ALUop==3'b000)?ResultAND: // AND
              (ALUop==3'b001)?ResultOR: // OR
              (ALUop==3'b010)?calculate[31:0]: // ADD
              (ALUop==3'b011)?ResultXOR: // XOR
              (ALUop==3'b100)?ResultRight: // shift right
              (ALUop==3'b101)?CarryOut: // compare unsigned numbers
              (ALUop==3'b110)?calculate[31:0]: //Substract
              (ALUop==3'b111)?{31'b0,calculate[31]^Overflow}: // SLT
              0;

assign Zero=(Result==0)?1:0;
```

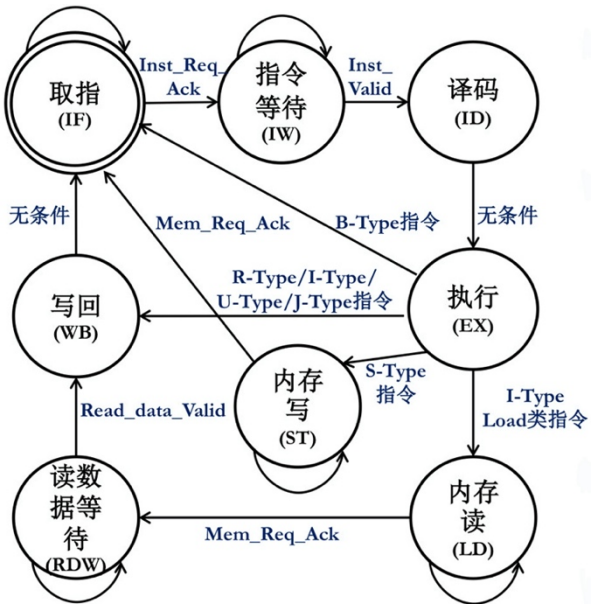
3. Custom_cpu 部分

（1） 三段式状态机：

第一段：时序逻辑控制状态跳转

```
//state machine
    always@(posedge clk)
    begin
        if(rst) state <= `N;
        else state <= nextstate;
    end
```

第二段：组合逻辑控制下一状态




```
`define N 9'b000000001
`define IF 9'b000000010
`define IW 9'b000000100
`define ID 9'b000001000
`define EX 9'b000010000
`define WB 9'b000100000
`define ST 9'b001000000
`define LD 9'b010000000
`define RDW 9'b100000000
```

```
always@(*)
begin
    case(state)
    `N:
        nextstate = `IF;
    `IF:
        if(Inst_Req_Ack & Inst_Req_Valid) nextstate = `IW;
        else nextstate = `IF;
    `IW:
        if(Inst_Valid & Inst_Ack) nextstate = `ID;
        else nextstate = `IW;
    `ID:
        nextstate = `EX;
    `EX:
        if (Branch) nextstate = `IF;
        else if (Load) nextstate = `LD;
        else if (Store) nextstate = `ST;
        else nextstate = `WB;
    `WB:
        nextstate = `IF;
    `ST:
        if (Mem_Req_Ack & MemWrite) nextstate = `IF;
        else nextstate = `ST;
    `LD:
        if (Mem_Req_Ack & MemRead) nextstate = `RDW;
        else nextstate = `LD;
    `RDW:
        if (Read_data_Valid & Read_data_Ack) nextstate = `WB;
        else nextstate = `RDW;
    default:
        nextstate = `N;
    endcase
end
```

State 和 nextstate 信号的位宽均为 9 位，有 9 个不同状态。除了 PPT 上写的 8 个以外，还设计了一个初始状态 N。

使用 one-hot 的方式定义不同状态。

第三段：给信号赋值

```
always@(posedge clk)
begin
    if (rst) Inst_Req_Valid <= 0;
    else if (state==`IF)
    begin
        if (Inst_Req_Valid==0) Inst_Req_Valid <= 1;
        else if (Inst_Req_Valid & Inst_Req_Ack) Inst_Req_Valid <= 0;
    end
    else Inst_Req_Valid <= 0;
end

assign Inst_Ack = (state == `IW) | (state == `IF && Inst_Valid);
assign Read_data_Ack = (state == `RDW);

always@(posedge clk)
begin
    if (rst) MemRead <= 0;
    else if (state==`LD)
    begin
        if (MemRead==0) MemRead <= 1;
        else if (MemRead & Mem_Req_Ack) MemRead <= 0;
    end
    else MemRead <= 0;
end

always@(posedge clk)
begin
    if (rst) MemWrite <= 0;
    else if (state==`ST)
    begin
        if (MemWrite==0) MemWrite <= 1;
        else if (MemWrite & Mem_Req_Ack) MemWrite <= 0;
    end
    else MemWrite <= 0;
end
```

赋值的判断条件和 mips 一致。

(2) 处理 Instruction

```
always@(posedge clk)
if(rst) instruction <= 32'd0;
else if(Inst_Valid && Inst_Ack) instruction <= Instruction;
```

因为 Instruction 只出现一个时钟周期，所以用 instruction 储存指令。(对 Read_data 也做一样的处理。)

```
assign op = instruction[6:0];
assign rd = instruction[11:7];
assign func = instruction[14:12];
assign rs1 = instruction[19:15];
assign rs2 = instruction[24:20];
assign top7 = instruction[31:25];


 assign Lui = (op==7'b0110111);
 assign Auipc = (op == 7'b0010111);
 assign Jal = (op == 7'b1101111);
 assign Jalr = (op == 7'b1100111) && (func == 3'b000);
 assign Beq = (op == 7'b1100011) && (func == 3'b000);
 assign Bne = (op == 7'b1100011) && (func == 3'b001);
 assign Blt = (op == 7'b1100011) && (func == 3'b100);
 assign Bge =  (op == 7'b1100011) && (func == 3'b101);
 assign Bltu = (op == 7'b1100011) && (func == 3'b110);
 assign Bgeu = (op == 7'b1100011) && (func == 3'b111);
 assign Lb = (op == 7'b0000011) && (func == 3'b000);
 assign Lh = (op == 7'b0000011) && (func == 3'b001);
 assign Lw = (op == 7'b0000011) && (func == 3'b010);
 assign Lbu = (op == 7'b0000011) && (func == 3'b100);
 assign Lhu = (op == 7'b0000011) && (func == 3'b101);
 assign Sb = (op == 7'b0100011) && (func == 3'b000);
 assign Sh = (op == 7'b0100011) && (func == 3'b001);
 assign Sw = (op == 7'b0100011) && (func == 3'b010);
 assign Addi = (op == 7'b0010011) && (func == 3'b000);
 assign Slti = (op == 7'b0010011) && (func == 3'b010);
 assign Sltiu =  (op == 7'b0010011) && (func == 3'b011);
 assign Xori = (op == 7'b0010011) && (func == 3'b100);
 assign Ori = (op == 7'b0010011) && (func == 3'b110);
 assign Andi = (op == 7'b0010011) && (func == 3'b111);
 assign Slli = (op == 7'b0010011) && (func == 3'b001) && (top7 == 7'b0000000);
 assign Srli = (op == 7'b0010011) && (func == 3'b101) && (top7 == 7'b0000000);
 assign Srai = (op == 7'b0010011) && (func == 3'b101) && (top7 == 7'b0100000);
 assign Add = (op == 7'b0110011) && (func == 3'b000) && (top7 == 7'b0000000);
 assign Sub = (op == 7'b0110011) && (func == 3'b000) && (top7 == 7'b0100000);
 assign Sll = (op == 7'b0110011) && (func == 3'b001) && (top7 == 7'b0000000);
 assign Slt = (op == 7'b0110011) && (func == 3'b010) && (top7 == 7'b0000000);
 assign Sltu = (op == 7'b0110011) && (func == 3'b011) && (top7 == 7'b0000000);
 assign Xor = (op == 7'b0110011) && (func == 3'b100) && (top7 == 7'b0000000);
 assign Srl = (op == 7'b0110011) && (func == 3'b101) && (top7 == 7'b0000000);
 assign Sra = (op == 7'b0110011) && (func == 3'b101) && (top7 == 7'b0100000);
 assign Or = (op == 7'b0110011) && (func == 3'b110) && (top7 == 7'b0000000);
 assign And = (op == 7'b0110011) && (func == 3'b111) && (top7 == 7'b0000000);
```

先对 instruction 进行拆分，再确定当前执行的是哪一条指令。

（3） PC 时序逻辑

```
always@(posedge clk)
begin
    if (rst) PC <= 32'b0;
    else if (state == `EX) PC <= PC2;
end

always@(posedge clk)
begin
    if(rst) PC1 <= 32'd0;
    else if(state == `IF) PC1 <= PC;
end

assign PC3 = PC + 32'd4;
assign PCjump = (Jal || Jalr)?{Result[31:1],1'b0}:(PC+extend);
assign PC2 = (Jal || Jalr || ((Beq || Bge || Bgeu) && Zero)||((Bne || Blt || Bltu ) && !Zero))?PCjump:PC3;
```

在 EX 状态是给 PC 赋新的值。

PC1 的作用是为了储存前一个 PC，在 Jal 和 Jalr 指令的时候需要用到。

（4） RF_wen，RF_waddr 和 RF_wdata

```
assign RF_wen = (state == `WB);
assign RF_waddr = rd;
assign RF_wdata = (Slli)?(A<<rs2):
                  (Sll)?(A<<{B[4:0]}):
                  (Srli)?(A>>rs2):
                  (Srl)?(A>>{B[4:0]}):
                  (Lb || Lbu || Lh || Lhu)?Data_load:
                  (Jal || Jalr)? PC1+32'd4:
                  (Lui)?{instruction[31:12],12'd0}:
                  (Load)?Read_data1:
                  Result;

assign Data_load = (Lb)?
                       (Result[1:0]==2'b00)?{{24{Read_data1[7]}},Read_data1[7:0]}:
                       (Result[1:0]==2'b01)?{{24{Read_data1[15]}},Read_data1[15:8]}:
                       (Result[1:0]==2'b10)?{{24{Read_data1[23]}},Read_data1[23:16]}:
                       (Result[1:0]==2'b11)?{{24{Read_data1[31]}},Read_data1[31:24]}:
                       0:
                   (Lbu)?
                       (Result[1:0]==2'b00)?$unsigned(Read_data1[7:0]):
                       (Result[1:0]==2'b01)?$unsigned(Read_data1[15:8]):
                       (Result[1:0]==2'b10)?$unsigned(Read_data1[23:16]):
                       (Result[1:0]==2'b11)?$unsigned(Read_data1[31:24]):
                       0:
                   (Lh)?
                       (Result[1]==1'b0)?{{16{Read_data1[15]}},Read_data1[15:0]}:
                       (Result[1]==1'b1)?{{16{Read_data1[31]}},Read_data1[31:16]}:
                       0:
                   (Lhu)?
                       (Result[1]==1'b0)?$unsigned(Read_data1[15:0]):
                       (Result[1]==1'b1)?$unsigned(Read_data1[31:16]):
                       0:
                   0;
```

WB 阶段时往寄存器内写入数据，因此在 WB 阶段时把 RF_wen 赋为 1.

写入数据的寄存器地址一直为 rd，比 mips 简单。

二、 实验过程中遇到的问题、对问题的思考过程及解决方法（比如 RTL 代码中出现的逻辑 bug，仿真、本地上板及云平台调试过程中的难点等）

Risc-v 对于立即数的存储很特别，写的时候需要耐心一点把位数写对。因为这个错误在 rtl 代码检查的时候不会报错，所以出错的时候需要多花一点时间。

总的来说在已经写过 mips 处理器之后再写 risc-v 处理器就显得非常简单，指令格式更加工整，而且也有了 debug 的经验，处理起来得心应手。

三、 对讲义中思考题（如有）的理解和回答

无

四、 在课后，你花费了大约 12 小时完成此次实验。

五、 对于此次实验的心得、感受和建议（比如实验是否过于简单或复杂，是否缺少了某些你认为重要的信息或参考资料，对实验项目的建议，对提供帮助的同学的感谢，以及其他想与任课老师交流的内容等）

感觉这个实验比 mips 简单多了，可以考虑先写 risc-v 处理器再写 mips 处理器，应该会更容易上手。

感谢蒋卓伦同学在 sep 讨论区里提出的关于状态跳转时判定信号如何书写的问题，很有帮助。