中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2018K8009929046　　姓名：何咏哲　　专业：计算机科学与技术

实验序号：4　　实验名称：RISC-V 指令集处理器

注 1：请在实验项目个人本地仓库中创建顶层目录 doc。撰写此 Word 格式实验报告后以 PDF 格式保存在 doc 目录下。文件命名规则：学号-prjN.pdf，其中学号中的字母“K”为大写，“-”为英文连字符，“prj”和后缀名“pdf”为小写，“N”为 1 至 4 的阿拉伯数字。例如：2018K8009929000-prj1.pdf。PDF 文件大小应控制在 5MB 以内。此外，实验项目 5 包含多个选做内容，每个选做实验应提交各自的实验报告文件，文件命名规则：学号-prj5-projectname.pdf，例如：2018K8009929000-prj5-dma.pdf。具体要求详见实验项目 5 讲义。

注 2：使用 git add 及 git commit 命令将 doc 目录下的实验报告 PDF 文件添加到本地仓库，并通过 git push 推送提交。

注 3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

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）

```
121 always @(posedge clk) begin
122     if(rst) begin
123         PC <= 32'b0;
124     end else if(state==EX) begin
125         PC <= Branch_eff ? Branch_PC : PC_4;
126     end else begin
127         PC <= PC;
128     end
129 end
130
131 always @(posedge clk) begin
132     if(state==IF) begin
133         originPC <= PC;
134     end else begin
135         originPC <= originPC;
136     end
137 end
```

PC 在 EX 阶段更新，其中 Branch_eff 以及 Branch_PC 包含了 Branch 和 Jump 类指令。通过新增一个 originPC 来记录跳转之前的 PC 的值。

```
always @(*) begin
    case(state)
        RSTSTATE: next_state = IF;
        IF: if(Inst_Req_Ack & Inst_Req_Valid) begin
                next_state = IW;
            end else begin
                next_state = IF;
            end
        IW: if(Inst_Ack & Inst_Valid) begin
                next_state = ID;
            end else begin
                next_state = IW;
            end
        ID: next_state = EX;
        EX: case(type)
                B_type: next_state = IF;
                S_type: next_state = ST;
                I_type: if(opcode==`LOAD) begin
                            next_state = LD;
                        end else begin
                            next_state = WB;
                        end
                default: if(opcode==7'b0) begin //avoid the influence of empty instruction
                            next_state = IF; // although it might be uncessary
                        end else begin
                            next_state = WB;
                        end
            endcase
        // all the hand-shaking signals should be in the condition of "if"
        // or the signals may not become valid in time
        LD: if(Mem_Req_Ack & MemRead) begin
                next_state = RDW;
            end else begin
                next_state = LD;
```

```
                            next_state = WB;
                        end
            endcase
        // all the hand-shaking signals should be in the condition of "if"
        // or the signals may not become valid in time
        LD: if(Mem_Req_Ack & MemRead) begin
                next_state = RDW;
            end else begin
                next_state = LD;
            end
        ST: if(Mem_Req_Ack & MemWrite) begin
                next_state = IF;
            end else begin
                next_state = ST;
            end
        WB: next_state = IF;
        RDW: if(Read_data_Ack & Read_data_Valid) begin
                next_state = WB;
            end else begin
                next_state = RDW;
            end
        default: next_state = IF;
    endcase
end
```

状态机之间的转移，主要进行了三个方面的修改：第一，原来用于判断的握手信号只有输入方，现为了避免信号没有及时拉高的情况，将一对握手信号同时作为判断依据；第二，修改了 EX 阶段的跳转分支，以指令类型作为关键字，和 RISC-V 相适配；第三，增加了对空指令的特判，因为一开始在 debug 的时候发现，如果输入空指令（全 0）的话会被归入到 J_type 里面，导致向通用寄存器中写入奇怪的东西。后来发现 RISC-V 中并没有全 0 的指令，其使用的是 nop 来表示空指令，对应的是 addi x0, x0, 0，而 x0 代表的是零号寄存器，所以不会造成任何影响。

```
// MemRead and MemWrite used to be wire type
// However, the program always recieve bad trap at the begenning
// This problem was solved after changing these two variables to reg type
always @(posedge clk) begin
    MemRead <= (state == LD)?1:0;
end

always @(posedge clk) begin
    MemWrite <= (state == ST)?1:0;
end

```

在上一个实验中，MemRead 和 MemWrite 都是 wire 类型，然而到了本次实验中，在我代码的基础上，继续使用 wire 类型的话会导致 medium 组仿真对而上板出错，将这两个握手信号修改为 reg 类型之后就能正确运行。具体原因我也不太清楚，猜测是信号的时序和延迟带来的影响。

```
assign opcode = valid_Instruction[6:0];
assign rd = valid_Instruction[11:7];
assign funct3 = valid_Instruction[14:12];
assign rs1 = valid_Instruction[19:15];
assign rs2 = valid_Instruction[24:20];
assign funct7 = valid_Instruction[31:25];
// the immediate/offset number in RISC-V has a common character
// that the highest bit of immediate/offset number is also at
// the highest position of RISC-V instruction
assign sign = valid_Instruction[31]?1'b1:1'b0;
assign PC_4 = PC + 4;
assign type = (opcode==`CALCU || (opcode==`IMMED && (funct3==`f3SLLI || funct3==`f3SRI)))? R_type:
              (opcode==`LOAD  || (opcode==`IMMED &&  funct3!=`f3SLLI && funct3!=`f3SRI)) ? I_type:
              (opcode==`STORE) ? S_type:
              (opcode==`BRANCH)? B_type:
              (opcode==`LUI || opcode==`AUIPC)? U_type : J_type; // JALR is put into J_type
```

译码逻辑部分，根据指令集中的描述，使用不同的信号来存储指令的不同部位，并根据指令格式将其划分为不同的类型。其中 JALR 指令的格式类型本来应该是 I_type,但为了后续执行的方便，我将 JALR 指令归入到了 J_type 中了。

另外 RISC-V 中立即数/offset 的最高位都被存储在了指令的最高位，这为译码带来了极大的方便。如图所示，我使用了 sign 信号来存储该数据。

```
// use to judge the shift instructions in R_type
//including SLLI, SRLI, SRAI
assign shift_imm = (opcode==`IMMED && (funct3==`f3SLLI || funct3==`f3SRI))?1'b1:1'b0;
```

立即数移位操作（SLLI，SRLI，SRAI，下同）在后续需要单独进行判断，故使用 shift_imm 信号来进行选择。

```
// B_type and J_type start from NO.1 bit, so the NO.0 bit should be assigned as 0
assign sign_extend = (type==I_type)? {{20{sign}},valid_Instruction[31:20]}:
                     (type==S_type)? {{20{sign}},valid_Instruction[31:25],valid_Instruction[11:7]}:
                     (type==B_type)? {{20{sign}},valid_Instruction[7],valid_Instruction[30:25],valid_Instruction[11:8],1'b0}:
                     (type==J_type)? {{12{sign}},valid_Instruction[19:12],valid_Instruction[20],valid_Instruction[30:21],1'b0}:
                     {valid_Instruction[31:12],12'b0};

//special judgement of JALR
assign JALR_extend = {{20{sign}},valid_Instruction[31:20]};
```

立即数/offset 有符号扩展，依照指令集给出的格式进行。其中 JALR 被归入到了 J_type，而其扩展应与 I_type 相同，故新增加了一个 JALR_extend 来进行特判。（其实可以通过判断 type==I_type || opcode==`JALR 来实现同样的功能，这样在后续 ALU 的操作中也能减少判断）。以及 B_type 和 J_type 指令的立即数只显式地表达到了第 1 位，这意味着第 0 位隐含为 0。

```
// assign the lowest 2 bits as 0
assign Address = ALU_result & 32'hfffffffc;
assign Inst_Ack = (state==RSTSTATE || state==IW)?1:0;
assign Inst_Req_Valid = (state == IF)?1:0;
assign Read_data_Ack = (state == RDW)?1:0;
```

Adress 的末两位应该为 0，因为内存中的数据是按照字节存储的。

```
reg_file reg_data(
    .clk(clk),
    .rst(rst),
    .waddr(rd),
    .raddr1(rs1),
    .raddr2(rs2),
    .wen(RF_wen),
    .wdata(RF_wdata),
    .rdata1(RF_rdata1),
    .rdata2(RF_rdata2)
);

alu res_cal(
    .A(ALU_num1),
    .B(ALU_num2),
    .ALUop(ALUop),
    .Result(ALU_result),
    .Overflow(),
    .CarryOut(),
    .Zero(Zero)
);

```

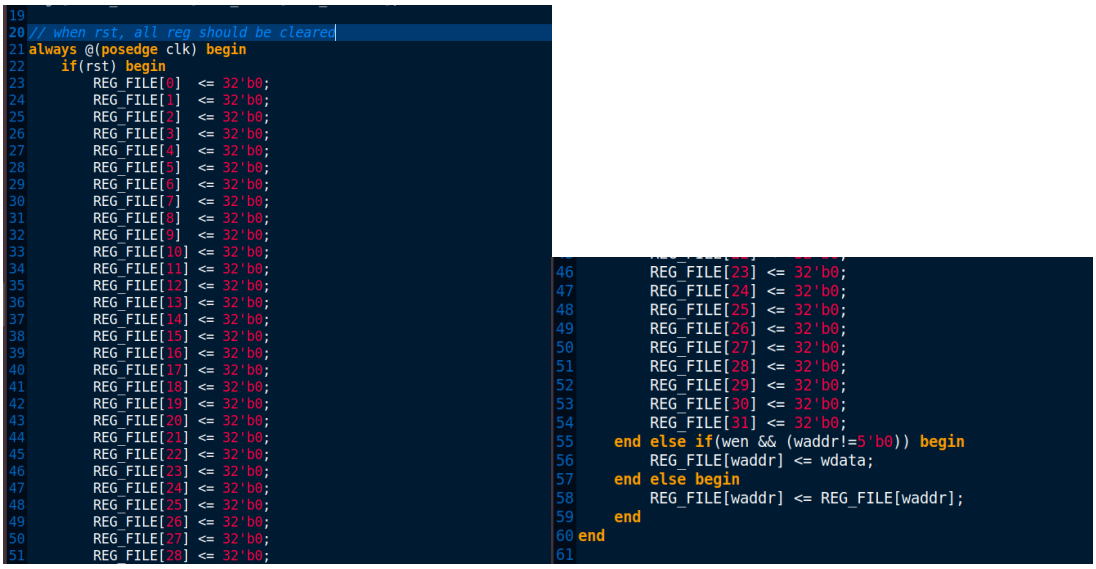
```
assign ALUop = (opcode==`LOAD || opcode==`STORE || type==J_type || (type==I_type && funct3==`f3ADDI) ||
               ( type==R_type && funct3==`f3A_S && funct7==`f7ADD)) ? `ADD :
               ((type==B_type && (funct3==`f3BEQ  || funct3==`f3BNE)) || (type==R_type && funct3==`f3A_S && funct7==`f7SUB)) ? `SUB :
               ((type==I_type && funct3==`f3ORI)  || (type==R_type && funct3==`f3OR))  ? `OR  :
               ((type==I_type && funct3==`f3ANDI) || (type==R_type && funct3==`f3AND)) ? `AND :
               ((type==B_type && (funct3==`f3BLT  || funct3==`f3BGE)) || (type==R_type && funct3==`f3SLT) || (type==I_type && funct3==`f3SLTI)) ? `SLT :
               ( type==R_type && (funct3==`f3SLLI || funct3==`f3SLL)) ? `SL  :
               ((type==R_type && funct3==`f3SRI && funct7==`f7SRLI) || (type==R_type && funct3==`f3SR && funct7==`f7SRL)) ? `SRL :
               ((type==R_type && funct3==`f3SRI && funct7==`f7SRAI) || (type==R_type && funct3==`f3SR && funct7==`f7SRA)) ? `SRA :
               ((type==I_type && funct3==`f3XORI) || (type==R_type && funct3==`f3XOR)) ? `XOR : `SLTU ;

assign ALU_num1 = opcode==`JAL ? originPC : RF_rdata1;
assign ALU_num2 = (type==B_type || (type==R_type && !shift_imm)) ? RF_rdata2 :
                  shift_imm ? {27'b0,rs2} :
                  opcode==`JALR ? JALR_extend : sign_extend;
```

复用了 ALU，上图为 ALU 输入的译码。

```
assign store_byte = sb_strb;
assign sb_result =  {32{store_byte[0]}} & {{24{1'b0}},RF_rdata2[7:0]} |
                    {32{store_byte[1]}} & {{16{1'b0}},RF_rdata2[7:0],{8{1'b0}}} |
                    {32{store_byte[2]}} & {{8{1'b0}},RF_rdata2[7:0],{16{1'b0}}} |
                    {32{store_byte[3]}} & {RF_rdata2[7:0],{24{1'b0}}};
assign sh_result =  {32{~ALU_result[1]}} & {{16{1'b0}},RF_rdata2[15:0]} |
                    {32{ ALU_result[1]}} & {RF_rdata2[15:0],{16{1'b0}}};
assign Store_extend = {32{funct3==`f3SB}}  &  sb_result |
                      {32{funct3==`f3SH}}  &  sh_result |
                      {32{funct3==`f3SW}}  &  RF_rdata2 ;
assign byte = ALU_result[1:0];
assign lb_data = ( byte[1] &  byte[0]) ? valid_Read_data[31:24] :
                 ((byte[1] & !byte[0]) ? valid_Read_data[23:16] :
                 ((!byte[1]&  byte[0]) ? valid_Read_data[15:8]  : valid_Read_data[7:0]));
assign lh_data = (!byte[1] & !byte[0]) ? valid_Read_data[15:0] : valid_Read_data[31:16];
assign Read_extend = {32{funct3==`f3LB}}  & {{24{lb_data[7]}},lb_data[7:0]}   |
                     {32{funct3==`f3LH}}  & {{16{lh_data[15]}},lh_data[15:0]} |
                     {32{funct3==`f3LW}}  & {valid_Read_data[31:0]}            |
                     {32{funct3==`f3LBU}} & {{24{1'b0}},lb_data[7:0]}          |
                     {32{funct3==`f3LHU}} & {{16{1'b0}},lh_data[15:0]};
assign Write_data = Store_extend;
assign sb_strb = 4'b1000 >> (~ALU_result[1:0]);
assign sh_strb = {ALU_result[1],ALU_result[1],!ALU_result[1],!ALU_result[1]};
assign sw_strb = 4'b1111;
assign Write_strb = funct3==`f3SB ? sb_strb :
                    funct3==`f3SH ? sh_strb :
                    sw_strb;
```

Write_strb 以及访存类操作的数据选择，内容与之前实验相同，只是更改了名字，在此不再赘述。



```
19
20 // when rst, all reg should be cleared
21 always @(posedge clk) begin
22     if(rst) begin
23         REG_FILE[0]  <= 32'b0;
24         REG_FILE[1]  <= 32'b0;
25         REG_FILE[2]  <= 32'b0;
26         REG_FILE[3]  <= 32'b0;
27         REG_FILE[4]  <= 32'b0;
28         REG_FILE[5]  <= 32'b0;
29         REG_FILE[6]  <= 32'b0;
30         REG_FILE[7]  <= 32'b0;
31         REG_FILE[8]  <= 32'b0;
32         REG_FILE[9]  <= 32'b0;
33         REG_FILE[10] <= 32'b0;
34         REG_FILE[11] <= 32'b0;
35         REG_FILE[12] <= 32'b0;
36         REG_FILE[13] <= 32'b0;
37         REG_FILE[14] <= 32'b0;
38         REG_FILE[15] <= 32'b0;
39         REG_FILE[16] <= 32'b0;
40         REG_FILE[17] <= 32'b0;
41         REG_FILE[18] <= 32'b0;
42         REG_FILE[19] <= 32'b0;
43         REG_FILE[20] <= 32'b0;
44         REG_FILE[21] <= 32'b0;
45         REG_FILE[22] <= 32'b0;
46         REG_FILE[23] <= 32'b0;
47         REG_FILE[24] <= 32'b0;
48         REG_FILE[25] <= 32'b0;
49         REG_FILE[26] <= 32'b0;
50         REG_FILE[27] <= 32'b0;
51         REG_FILE[28] <= 32'b0;
46         REG_FILE[23] <= 32'b0;
47         REG_FILE[24] <= 32'b0;
48         REG_FILE[25] <= 32'b0;
49         REG_FILE[26] <= 32'b0;
50         REG_FILE[27] <= 32'b0;
51         REG_FILE[28] <= 32'b0;
52         REG_FILE[29] <= 32'b0;
53         REG_FILE[30] <= 32'b0;
54         REG_FILE[31] <= 32'b0;
55     end else if(wen && (waddr!=5'b0)) begin
56         REG_FILE[waddr] <= wdata;
57     end else begin
58         REG_FILE[waddr] <= REG_FILE[waddr];
59     end
60 end
61
```

Reg_file 的修改，在复位时所有寄存器均清零。

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）

问题 1：不知道还需要对访存类指令进行数据选择，事实上至今也没有在指令集中找到相关描述。

解决：请同学帮忙 debug 的过程中告诉我，和之前实验一样需要进行数据选择。于是将之前写的内容 paste 了过来，并做了简要的修改。

问题 2：JALR 指令的分类混乱，造成了立即数扩展以及 ALU 操作数的混乱。

解决：为了避免带来更多的 bug，将 JALR 在立即数扩展以及 ALU 操作数中特殊化处理了，如第一部分所示。虽然这种处理可以进一步优化。

问题 3：立即数移位指令操作不当，导致了 medium 组上板只能过一半。

解决：问题来源于指令归类问题，立即数移位指令应该是 R_type 的，但是在后续 ALUop 的译码中当作了 I_type 来处理。将其修改为对应的 R_type 后得以解决。

三、 对讲义中思考题（如有）的理解和回答

MIPS 和 RISC-V 同属于 RISC。与 MIPS 指令集相比，RISC-V 更加简洁优雅，例如 opcode 统一在指令的末 7 位，立即数的最高位都在指令的最高位，编码也更适合进行符号扩展，以及操作数的位置相对固定，这些都为译码带来了更多的便利。此外，MIPS 具有固定的 16 位和 32 位编码，而对于 RISC-V 而言，其被设置为模块化的，尽管以 32 位为基准，但仍有 16 位、64 位的变种，同时人们正在寻找 128 位（用于百亿分之一的计算）的扩展，这位 RISC-V 带来了更多的可能性与更广阔的未来。就我个人而言，我更加喜欢 RISC-V。

四、 在课后，你花费了大约___10___小时完成此次实验。

写代码 3h + debug 7h = 总共 10h

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）

实验难度较小，困难的地方主要在于 debug 时很难发现错误，不过 RISC-V 的译码较为简洁，很多错误都是可以通过认真仔细写代码来提前避免的。相比于上一个实验，我在看波形以及对照反汇编指令方面积累了更多的经验，丢掉拐棍一开始确实很折磨，但是这也是成长的过程中必不可少的一个过程。不过尽管如此，我还是希望在后续的教学过程中，老师们能逐步调整这个丢掉拐棍的过程，给同学们一个过渡阶段，例如在实验二的拐棍基础上减少自动比对功能，而是将正确执行的结果（包括指令、寄存器的值、内存读写情况等）以文本形式提供给同学，让同学们自行比对。不然的话一下子丢掉拐棍会让很多同学感觉到难度骤升，在茫然 debug 的过程中丧失了对实验的热情。

在帮助同学 debug 的过程中，我发现了与上一个实验类似的问题，即 ALU 和 regfile 的可综合性不强导致了实验结果错误，例如 regfile 中 if-else 条件没有写全，alu 中使用了>>>、$signed()之类的操作。而这些不易发现的错误也使得这位同学在 debug 过程中浪费了很多时间去做无用的工作。不过我觉得这种浪费是值得的，否则可能无法深入理解代码规范要求的意义所在。

希望在下一届的实验中，老师们能对 strb 以及访存类数据的选择进行一点点提示，我之前以为 RISC-V 不用考虑数据选择，从而使得自认为数据读写都没有错，然而实际上并不符合预期的正确结果。

另外在状态转移部分，我对空指令（全 0）进行了特判，防止其对我的逻辑造成干扰。当时并没有求证空指令是否真的会带来影响，在验收的过程中陈欲晓助教让我可以去了解一下 RISC-V 中的空指令，经过了解发现，RISC-V 中的空指令（nop）是通过 addi 向 0 号寄存器中写入数据来实现的，并不是全 0，因此应该被归入到 I_type 类型，而非向我原本以为的会被归入到 J_type，并且这一操作不会对寄存器带来影响。

最后，感谢**王嵩岳**同学在 debug 过程中对我的大力帮助！