中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929034 姓名：侯汝垚 专业：计算机科学与技术

实验序号：4 实验名称：定制 RISC-V 功能型处理器设计

注 1：撰写此 Word 格式实验报告后以 PDF 格式保存 SERVE CloudIDE 的 /home/serve-ide/cod-lab/reports 目录下（注意：reports 全部小写）。文件命名规则：prjN.pdf，其中“prj”和后缀名“pdf”为小写，“N”为 1 至 4 的阿拉伯数字。例如：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：prj5-projectname.pdf，其中“-”为英文标点符号的短横线。文件命名举例：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 命令将实验报告 PDF 文件添加到本地仓库 master 分支，并通过 git push 推送到 SERVE GitLab 远程仓库 master 分支（具体命令详见实验报告）。

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

1. RTL 代码段

RISC-V 指令集的指令格式与 MIPS 类似，需要实现的命令大部分也都在上一个实验中有所涉及，因此主要看与 MIPS 处理器不相同的地方。

```
// IR*************************************
wire [ 6:0] func7, opcode;
wire [ 4:0] rs2, rs1, rd;
wire [ 2:0] func3;
wire [31:0] U_imm, B_imm, J_imm, S_imm, I_imm;
reg  [31:0] IR;
wire IRWrite;

always @(posedge clk) begin
        if (IRWrite) begin
                IR <= Instruction;
        end
        else begin
                IR <= IR;
        end
end
assign {func7, rs2, rs1, func3, rd, opcode} = IR;
assign I_imm = {{20{IR[31]}}, IR[31:20]};
assign S_imm = {{20{IR[31]}}, IR[31:25], IR[11:7]};
assign U_imm = {IR[31:12], 12'b0};
assign B_imm = {{20{IR[31]}}, IR[7], IR[30:25], IR[11:8], 1'b0};
assign J_imm = {{12{IR[31]}}, IR[19:12], IR[20], IR[30:21], 1'b0};
```

首先是对于指令个字段的分解，以分得最细的 R-type 指令为基础分解，然后将 I、S、B、U、J 五种格式的立即数分别拼接形成，这五种指令的立即数格式各不相同，而 MIPS 中立即数只有 J-type 和 I-type 两种。

2.

```
// PC*****************************************
reg  [31:0] PC_reg;
wire [31:0] PC_next, PC_jmp;
wire PCWrite;
wire branchValid = PCWriteCond && (Zero ^ func3[0] ^ func3[2]);

assign PC = PC_reg;
assign PC_next = {32{PCWriteCond && ~branchValid}} & PC_plus
               | {32{(branchValid || ~PCWriteCond) && !JALR}} & ALUOut
               | {32{(branchValid || ~PCWriteCond) &&  JALR}} & {ALUOut[31:1], 1'b0};
always @(posedge clk) begin
        if (rst) begin
                PC_reg <= 32'b0;
        end
        else if (PCWrite || branchValid) begin
                PC_reg <= PC_next;
        end
        else begin
                PC_reg <= PC_reg;
        end
end

reg  [31:0] PC_plus;
always @(posedge clk) begin
        if (current_state[1])
                PC_plus <= ALU_result;
        else
                PC_plus <= PC_plus;
end
```

另一个很大的不同在于 RICS-V 指令集中涉及 PC 参与运算的指令，PC 是当前指令的 PC 而非下一条指令的 PC，因此不能像 MIPS 处理器一样取完指令后直接将 PC 自增，但是如果不自增，对于 B-type 指令，译码阶段计算跳转地址，执行阶段计算跳转条件是否成立，这两个周期都无法通过 ALU 计算 PC + 4，一种解决办法办法是 PC 自增但是用一个寄存器当前指令的 PC，另一种方法是 PC 不自增而是用一个寄存器在取指阶段存储 PC + 4。参考金标准中 PC 在执行阶段更新，采用后者。寄存器 PC_plus 在取指阶段寄存 PC + 4。

```
    wire func30 = func3[0] || R_type && func7[5];
    wire [2:0] ALUEx;
    assign ALUEx[0] = ~func3[2] &&  func3[1] ||  func3[2] && (func3[1] ^ func30);
    assign ALUEx[1] = ~func3[2];
    assign ALUEx[2] =  func3[2] && ~func3[1] || ~func3[2] && (func3[1] ^ func30);
    wire [2:0] ALUop = {3{current_state[3] || I_type_load || S_type || current_state [1]}} & 3'b010
                     | {3{current_state[4] && (R_type || I_type_cal)}}           & ALUEx
                     | {3{current_state[4] && B_type && ~func3[2]}}              & 3'b110
                     | {3{current_state[4] && B_type &&  func3[2] && ~func3[1]}} & 3'b111
                     | {3{current_state[4] && B_type &&  func3[2] &&  func3[1]}} & 3'b011;
```

对于 ALUop 的译码，R-type 计算指令和 I-type 计算指令可以直接根据 funct3

字段结合 funct7[5]判断，而 funct[7]只在区分加减时有用，其余情况均为 0，故可以用 funct[7]和 funct[3]的与来进行译码。列出真值表后进行化简，获得 R-type 和 I-type 计算指令时的 ALUop，然后对于跳转指令、访存指令则单独考虑，在几种情况中选择出最后的 ALUop。

```
wire [ 1:0] Shfter_op = {func3[2], func7[5]};
```

经过对比发现，shifter_op 的译码也很轻松，只与 func[3]和 func[7]相关。

```
// memory load and store module ****************************************
// load
wire [3:0] offset;
assign Address   = {ALUOut[31:2], 2'b0};
assign offset[0] = (ALUOut[1:0] == 2'b00);
assign offset[1] = (ALUOut[1:0] == 2'b01);
assign offset[2] = (ALUOut[1:0] == 2'b10);
assign offset[3] = (ALUOut[1:0] == 2'b11);

// generate mask by op[1:0]
wire [31:0] mask, offsetRdata, validRdata, MemRdata;
assign mask[31:8] = {24{func3[1]}} & 24'hFFFFFF
                  | {24{func3[0]}} & 24'h0000FF;
assign mask[7:0] = 8'hFF;

// get RdataSignal by op[0]
wire [31:0] RdataSignal = {32{~func3[1] && ~func3[0]}} & {{24{validRdata[ 7]}}, 8'b0}
                        | {32{ func3[0]}}              & {{16{validRdata[15]}}, 16'b0};

assign offsetRdata = {32{offset[0]}} & {Read_data}
                   | {32{offset[1]}} & { 8'b0, Read_data[31: 8]}
                   | {32{offset[2]}} & {16'b0, Read_data[31:16]}
                   | {32{offset[3]}} & {24'b0, Read_data[31:24]};
assign validRdata = offsetRdata & mask;
assign MemRdata = {32{ func3[2]}} &  validRdata
                | {32{~func3[2]}} & (validRdata | RdataSignal);

reg [31:0] MDR;
always @(posedge clk) begin
        if (current_state[7] && Read_data_Ready && Read_data_Valid)
                MDR <= MemRdata;
        else
                MDR <= MDR;
end

// Store
wire [31:0] validWdata, swl_data;
wire [3:0] WriteWidth;
assign WriteWidth[0] = 1'b1;
assign WriteWidth[1] = func3[0] || func3[1];
assign WriteWidth[3:2] = {2{func3[1]}};

assign Write_strb = {4{offset[0]}} & {WriteWidth}
                  | {4{offset[1]}} & {WriteWidth[2:0], 1'b0}
                  | {4{offset[2]}} & {WriteWidth[1:0], 2'b0}
                  | {4{offset[3]}} & {WriteWidth[0]  , 3'b0};

assign Write_data = {32{offset[0]}} & {RF_rdata2}
                  | {32{offset[1]}} & {RF_rdata2[23: 0],  8'b0}
                  | {32{offset[2]}} & {RF_rdata2[15: 0], 16'b0}
                  | {32{offset[3]}} & {RF_rdata2[ 7: 0], 24'b0};
```

访存部分与 MIPS 基本一致，但是不需要考虑非对齐情况，更加简单。性能计数器、软件部分与 MIPS 处理器保持一致，无需改变。

二、 实验过程中遇到的问题、对问题的思考过程及解决方法

1．如何解决 PC 和 PC + 4 存储的矛盾

对于 B-type 指令，取指周期如果 PC 不自增，那么译码阶段计算跳转地址，执行阶段计算跳转条件是否成立，将没有周期可以用于计算 PC + 4，除非利用内存等地周期计算，但假如使用的是理想内存也不可行，所以解决办法是加一个寄存器存 PC + 4。

2．RISC-V 和 MIPS 中相同名字的指令执行过程不同

实验实现的 RISC-V 和 MIPS 指令大部分都重合，但是部分指令执行过程有细微差别，因为上个实验留下的印象较深，很容易让自己觉得自己是对的而难以排查出真正 bug，而 risc-v 的手册不像 MIPS 一样仔细列出每条指令语义，因此 debug 过程中出现了很多这也的错误，比如 risc-v 中立即数位运算是符号拓展而非零拓展，risc-v 的 jal 指令并不默认 31 号寄存器存返回地址。所以实验前还是得仔细研读指令手册，不要偷懒照抄 mips。

3．MDR 什么时候改变?

在我最开始的设计中，大部分寄存器都是每个周期都变化，因为大部分数据都是上个周期获得然后在下个周期使用，但是验收过程中陈飞羽老师指出，对于 MDR，数据可能就在一个周期内有效，为了 CPU 的健壮性，寄存器变化条件应该严格设定，比如 ALUOut 只在得出计算结果时变化，MDR 只在握手成功时跳转，保证寄存器中的数据是正确的数据。

```
reg [31:0] MDR;
always @(posedge clk) begin
        if (current_state[7] && Read_data_Ready && Read_data_Valid)
                MDR <= MemRdata;
        else
                MDR <= MDR;
end
```

4. 为什么 risc-v 的指令数少于 mips?

在验收过程中我的回答是可能是一些 risc-v 中特有的指令比如 LBU、BGEU 降低了 risc-v 的指令数目，但是没有考虑 NOP 指令的影响，当时没有考虑到延迟槽问题，因为我们的 CPU 并没有设计流水线，但是结合 MIPS 中存储的返回地址是 PC + 8 可以知道，实际上程序里在分支指令处应该是插入了很多 NOP 指令，所以我认为这可能是主要原因之一。因此之后在性能计数器中加入了 NOP 指令的计数器。

三、 对讲义中思考题（如有）的理解和回答

MIPS 和 RISC-V 处理器的性能对比

列出表格统计 mips 和 risc-v 处理器的总周期数、总指令数和 NOP 指令数。

	A	B	C	D	E	F	G	H	I
1		RISC-V		MIPS					
2	测试程序	总周期	总指令数	总周期	总指令数	NOP指令数	非NOP指令数	指令数之比	排除NOP指令后的指令数比
3	15pz	548228821	5224490	526798618	5287737	16273	5271464	0.988038929	0.991089003
4	bf	38813882	452863	46343985	559075	54928	504147	0.810021911	0.898275701
5	dinic	1476055	16700	1704874	19352	1003	18349	0.862959901	0.910131342
6	fib	181089661	2549534	179409543	2525748	15627	2510121	1.009417408	1.015701634
7	md5	385733	4924	404891	5253	325	4928	0.937369122	0.999188312
8	qsort	783441	9489	705053	8365	433	7932	1.134369396	1.196293495
9	queen	6886401	81499	6838960	80882	4607	76275	1.007628397	1.06848902
10	sieve	745817	10204	1192801	16504	180	16324	0.618274358	0.625091889
11	ssort	44722782	619054	52486359	728315	44578	683737	0.849981121	0.905397836
12	总	823132593	8968757	815885084	9231231	137954	9093277	0.971566739	0.986306367
13									
14									

总体来看 RISC-V 指令的性能略优于 MIPS，即使排除 NOP 指令后依然有微小优势，而我们实现的 RISC-V 指令数目是少于 MIPS 的，RISC-V 用更少的指令数获得更好的性能。细看的话大部分程序差距在百分之以内，但是 sieve 程序出现明显较大差距，这些差距也会和编译器有关，尽管两者采用的指令大部分相同，但是编译器可能

会用不同方式优化。

四、 在课后，你花费了大约 10 小时完成此次实验。

五、 对于此次实验的心得、感受和建议

本次实验因为有之前的经验，所以进行得很顺利，基本上和设计 MIPS 处理采用相同思路即可，状态机基本一模一样，大部分控制信号也可以通用，主要处理对指令的译码，在这个过程中也可以看出 risc-v 指令集的优势，risc-v 指令种类划分非常清晰，除了特殊的 JALR 外基本上一种格式指令对应一类操作，而 mips 中的 R-type 则是包罗万象，既要考虑计算移位，又要考虑 mov 或者 jump。另外 risc-v 在 ALUop 译码方面也有较大优势，只需要通过指令种类和 func3 以及 func7 中的某一位即可译码。MIPS 的延迟槽可能也是一项过时的技术，因此 MIPS 的衰败和 RISC-V 的兴起是有原因的。