中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2020K8009926006 姓名：游昆霖 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 格式保存在~/COD-Lab/reports 目录下。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 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（比如关键 RTL 代码段{包含注释}及其对应的逻辑电路结构图、相应信号的仿真波形和信号变化的说明等）

说明：除 custom_cpu.v 外软硬件代码均可复用 prj3，同时逻辑电路图也与 mips 处理器的基本一致，以下只对 custom_cpu.v 中针对 RISC-V 指令集进行的改动进行说明。

（1）状态机第二部分

```
//Part II: Combinatorial logic
        always @ (*)
        begin
                case(current_state)
                        RST: begin
                                next_state = IF ;
                        end
                        IF:     begin
                                if (Inst_Req_Ready)
                                        next_state = IW;
                                else
                                        next_state = IF;
```

```
                end
                IW: begin
                        if (Inst_Valid)
                                next_state = ID;
                        else
                                next_state = IW;
                end
                //NOP: Instr = 32'b0
                ID: begin
                        next_state = EX ;  //no need to care NOP
                end
                //EX->IF:
                        //B_type
                //EX->WB
                        //R_type / I_calc / I_jalr / U_type / J_type
                //EX->LD
                        //I_load
                //EX->ST
                        //S_type
                EX: begin
                        if(B_type)
                                next_state = IF;
                        else if(R_type | I_calc | I_jalr | U_type | J_type )
                                next_state = WB;
                        else if(I_load)
                                next_state = LD;
                        else if(S_type)
                                next_state = ST;
                        else
                                next_state = RST;
                end
                LD: begin
                        if(Mem_Req_Ready)
                                next_state = RDW;
                        else
                                next_state = LD;
                end
                ST: begin
                        if(Mem_Req_Ready)
                                next_state = IF;
                        else
                                next_state = ST;
                end
                RDW: begin
```

```
                        if(Read_data_Valid)
                                next_state = WB;
                        else
                                next_state = RDW;
                end
                WB: begin
                        next_state = IF;
                end
                default: begin
                        next_state = RST;
                end
        endcase
end
```

状态机的状态与 prj3 相同，跳转条件有所区别，但对应的指令功能基本一致。值得注意的是，本实验中 RISC-V 指令集无条件跳转指令(Jal 和 Jalr)均需要进行写寄存器操作，且 ID 到 EX 状态转移不再需要考虑为非 NOP 指令（RISCV 中的空指令用 addi x0,x0,0 实现）。

（2）PC 更新逻辑

```
always @ (posedge clk) begin
        if(current_state[1])//IF
                PC_tmp <= PC;
end

always @ (posedge clk) begin
        if (rst) begin
                PC <= 32'b0;
        end
        else if(current_state[2] & Inst_Valid & ~rst) begin
                PC <= ALU_Result; //PC+4 IW, consider the cycle before ID
        end
        else if(current_state[4] & ~rst)begin
                if (br_en)
                        PC <= ALU_Result_tmp; //br_tar
                else if(j_en)
                        PC <= j_tar;
                else
                        PC <= PC;
```

```
                        //PC <= br_en ? br_tar : j_en ? j_tar : PC ;
                        //j or br OP refresh in next IF, judge by EX
                        //note that the default result is PC4
                end
        end
```

虽然 RISCV 不使用分支延迟槽，因此求 PC 条件跳转地址的时候使用的是原 PC 值，但为提高 ALU 复用率及操作简便性，仍在 IW 阶段将 PC 进行+4，后续求条件跳转地址时使用 PC 的寄存值 PC_tmp 作为操作数。

（3）译码部分

```
//Analyse Instruction code
        //imm_extension:
                //I_type: unsigned: SLTIU(011)  signed: other OP  shamt: shift   note that I_jalr: {rs1+signed(offset)}[31:1],0
                //S_type: signed
                //B_type: signed  in multiples of 2
                //U_type: fill low 12 bit with 0
                //J_type: signed  in multiples of 2
        assign opcode   = Instruction_tmp[6:0];
        assign rd       = Instruction_tmp[11:7];
        assign rs1      = Instruction_tmp[19:15];
        assign rs2      = Instruction_tmp[24:20];
        assign shamt    = Instruction_tmp[24:20];
        assign funct3   = Instruction_tmp[14:12];
        assign funct7   = Instruction_tmp[31:25];
        assign I_imm[11:0]  = Instruction_tmp[31:20];
        assign I_imm[31:12] = funct3 == 3'b011 ? {20'b0} : {20{Instruction_tmp[31]}};
        assign S_imm    = { {20{Instruction_tmp[31]}}, Instruction_tmp[31:25], Instruction_tmp[11:7]};
        assign B_imm    = { {19{Instruction_tmp[31]}}, Instruction_tmp[31], Instruction_tmp[7], Instruction_tmp[30:25], Instruction_tmp[11:8], 1'b0};
        assign U_imm    = { Instruction_tmp[31:12],12'b0};
        assign J_imm    = { {11{Instruction_tmp[31]}}, Instruction_tmp[31], Instruction_tmp[19:12], Instruction_tmp[20], Instruction_tmp[30:21], 1'b0};

//Differ type by one-bit signals
        assign  R_type  = opcode == 7'b0110011;
        assign  I_calc  = opcode == 7'b0010011;
        assign  I_load  = opcode == 7'b0000011;
```

```
        assign  I_jalr  = opcode == 7'b1100111;
        assign  I_type  = I_calc | I_load | I_jalr;
        assign  S_type  = opcode == 7'b0100011;
        assign  B_type  = opcode == 7'b1100011;
        assign  U_lui   = opcode == 7'b0110111;
        assign  U_auipc = opcode == 7'b0010111;
        assign  U_type  = U_lui | U_auipc;
        assign  J_type  = opcode == 7'b1101111;
```

由于 RISCV 指令集更为简洁齐整，操作数位置较为固定，且分类可固定由最低 7 位决定。因此在译码部分获得指令类别，操作数内容，以及不同类别指令对应的立即数拓展。

（4）ALU 部分

```
//Channel to ALU
        //IW stage: use add to save adder of PC+4
        //ID stage: use add to save adder of PC_tmp+B_imm
        //WB stage: use add to save adder of PC_reg+4 by I_jalr and J_type
                //note that ALU is also used in EX_stage to get PC+imm by J_type or rs1+imm by I_jalr
        //Operations related to ALU:
        //TYPE                          op1     op2     OP
        //R(exclude shift)              rs1     rs2     ...
        //I_calc(exclude shift)         rs1     I_imm   ...
        //I_load I_jalr                 rs1     I_imm   ADD
        //S_type                        rs1     S_imm   ADD
        //B_type
                //BEQ,BNE               rs1     rs2     SUB
                //other                 rs1     rs2     SLT/SLTU
        //U_auipc                       PC_tmp  U_imm   ADD
        //J_type                        PC_tmp  J_imm   ADD


        assign ALU_op_origin =  ({3{R_type & ~(funct3==3'b001 | funct3 == 3'b101)}} & ( {3{funct3
== 3'b000}}     & {funct7[5],2'b10}   |
                                                                                        {3{funct3[
2:1] == 2'b01}} & {~funct3[0],2'b11}  |
                                                                                        {3{funct3
== 3'b100}}     & funct3             |
                                                                                        {3{funct3[
2:1] == 2'b11}} & ~funct3            )
                                ) |
```

```
                                ({3{I_calc & ~(funct3==3'b001 | funct3 == 3'b101)}} & ( {3{funct3
== 3'b000}}     & 3'b010              |
                                                                                {3{funct3[
2:1] == 2'b01}} & {~funct3[0],2'b11}  |
                                                                                {3{funct3
== 3'b100}}     & funct3              |
                                                                                {3{funct3[
2:1] == 2'b11}} & ~funct3             )
                                ) |
                                ({3{I_load | I_jalr | S_type | U_auipc | J_type }}  &  3'b010 ) |
                                ({3{B_type}}                                        &
( funct3[2:1]==2'b00 ? 3'b110 : {~funct3[1], 2'b11}) );
        always @(posedge clk) begin
                //if(current_state[3]) //ID
                        ALU_op_tmp <= ALU_op_origin;
                // else if(current_state[4] & (I_jalr | J_type)) //EX before WB
                //         ALU_op_tmp <= 3'b010;
        end
        assign ALU_op_final =   current_state[2] | current_state[3] ? 3'b010 :
                                current_state[8] & (I_jalr | J_type) ? 3'b010 :
                                ALU_op_tmp;

        assign ALU_A_origin =   U_auipc | J_type ? PC_tmp :
                                RF_rdata1;
        always @(posedge clk) begin
                //if(current_state[3]) //ID
                        ALU_A_tmp <= ALU_A_origin;
                // else if(current_state[4] & (I_jalr | J_type)) //EX before WB
                //         ALU_A_tmp <= PC_tmp;
        end
        assign ALU_A_final =    current_state[2]  ? PC :
                                current_state[3]  ? PC_tmp :
                                current_state[8] & (I_jalr | J_type) ? PC_tmp :
                                ALU_A_tmp;

        assign ALU_B_origin =   {32{(I_calc & ~(funct3==3'b001 | funct3 == 3'b101)) | I_load |
I_jalr}}  & I_imm        |
                                {32{S_type}}
       & S_imm        |
                                {32{U_auipc}}
       & U_imm        |
                                {32{J_type}}
       & J_imm        |
                                {32{R_type & ~(funct3==3'b001 | funct3 == 3'b101) |
```

```
B_type}}            & RF_rdata2    ;
        always @(posedge clk) begin
                //if(current_state[3]) //ID
                        ALU_B_tmp <= ALU_B_origin;
                // else if(current_state[4] & (I_jalr | J_type)) //EX before WB
                //         ALU_B_tmp <= 32'd4;
        end
        assign ALU_B_final =    current_state[2] ? 32'd4 :
                                current_state[3] ? B_imm :
                                current_state[8] & (I_jalr | J_type) ? 32'd4 :
                                ALU_B_tmp;

        always @(posedge clk) begin
                        ALU_Result_tmp <= ALU_Result;
        end

        alu alu_inst(
           .A(ALU_A_final),
           .B(ALU_B_final),
           .ALUop(ALU_op_final),
           .Overflow(ALU_Overflow),
           .CarryOut(ALU_CarryOut),
           .Zero(ALU_Zero),
           .Result(ALU_Result)
        );
```

EX 阶段各类型指令对应操作数及操作类型如注释所示，即得 ALU_XX_origin，将其寄存得到了 ALU_XX_tmp。由于在 IW 阶段复用 ALU 得到 PC+4，ID 阶段得到 PC_tmp+B_imm，WB 阶段得到 PC_tmp+4。设计数据旁路，并以状态进行选择寄存值和旁路数据，从而得到与 ALU 相连的信号 ALU_XX_final。

（5）Shifter 部分

```
//Channel to Shifter
        //Operations related to Shifter
        //R_type shift  rs1 rs2
        //I_calc shift  rs1 shamt
```

```
        assign Shifter_op = (R_type | I_calc) & (funct3==3'b001 | funct3 == 3'b101) ?
{funct3[2],funct7[5]} : 0;
        assign Shifter_A  = RF_rdata1;
        assign Shifter_B  = R_type ? RF_rdata2[4:0] : shamt;
        always @(posedge clk)begin
                //if(current_state[3]) //ID
                        Shifter_op_tmp <= Shifter_op;
                        Shifter_A_tmp  <= Shifter_A;
                        Shifter_B_tmp  <= Shifter_B;
        end

        always @(posedge clk)begin
                //if(current_state[4])//EX
                        Shifter_Result_tmp <= Shifter_Result;
        end

        shifter shifter_inst(
           .A(Shifter_A_tmp),
           .B(Shifter_B_tmp),
           .Shiftop(Shifter_op_tmp),
           .Result(Shifter_Result)
        );
```

由于 RISCV 中无 swl 和 swr 指令，不需复用 Shifter 得到写内存的数，考虑运算类型指令即可。

（6）Reg_file 部分

与 MIPS 逻辑基本一致，特别的，无条件跳转指令也需写寄存器。

（7）访存部分

减少了 swl 和 swr 指令，其余逻辑基本一致。

（8）性能计数器部分

与 prj3 使用的性能计数器相同，在更新条件处针对 RISCV 指令略作更改即可。

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，逻辑仿真和 FPGA 调试过程中的难点等）

问题 1：性能计数结果不符合预期

取值次数统计 inst_cnt 和访存次数统计 mem_cnt 远大于预期值，检查代码发现应当考虑内存延迟导致的状态停滞。添加对应 ready 信号使的仅统计该状态的最后一个周期，修改后性能计数结果符合预期。

以下问题均通过硬件仿真工具发现：

问题 2：硬件仿真工具第一阶段报错

查看报错日志，显示 next_state 产生了 latch。将状态机补充 default，if 补充 else，使得状态机覆盖所有情形即可。

问题 3：ALU_B_origin 遗漏情形

仿真通过，但上板错误，使用硬件仿真加速工具得到产生错误的时钟周期，结合硬件仿真的金标准信号以及出错 benchmark 的反汇编文件，发现 ALU_B_origin 遗漏了 I_jalr 和 I_load 的情形。

三、 对讲义中思考题（如有）的理解和回答

MIPS 和 RISC-V 性能对比：

	指令集	时钟周期数	指令总数	访存次数	内存延迟周期数	跳转次数	顺序更新次数	CPI
15pz	mips	330924220	5287727	3231803	91736634	631860	4639604	62.58
	riscv	324238832	5224477	3228762	90004486	625713	4598777	62.06
bf	mips	28358029	559065	100480	3987321	83994	420153	50.72
	riscv	23775610	452850	100484	3956925	83991	368872	52.50

dinic	mips	1047512	19342	6525	198150	1190	17159	54.16
	riscv	906059	16687	5566	171982	1277	15423	54.30
fib	mips	109644831	2525738	5389	188194	387757	2122364	43.41
	riscv	110676382	2549521	5303	183886	478083	2071451	43.41
md5	mips	249557	5243	577	20371	267	4661	47.60
	riscv	236203	4911	642	22927	359	4565	48.10
qsort	mips	437705	8355	2463	66179	1415	6517	52.39
	riscv	483070	9476	2464	65405	1415	8074	50.98
queen	mips	4240606	80872	26771	627369	6233	70042	52.44
	riscv	4284106	81486	26772	637557	5589	75910	52.57
sieve	mips	737234	16494	472	14654	1671	14653	44.70
	riscv	456038	10191	470	14335	1401	8803	44.75
ssort	mips	32122266	728305	19305	506651	62245	621492	44.11
	riscv	27431823	619041	18714	514622	67292	551762	44.31
平均结果	mips	56417995.56	1025682.33	377087.22	10816169.22	130736.89	879627.22	50.23
	riscv	54720902.56	996515.56	376575.22	10619125.00	140568.89	855959.67	50.33

由上表可知，从性能计数器统计结果上看，对于同一 benchmark，RISCV 指令集下，时钟周期数和指令数均略小于 MIPS 指令集，而 CPI 略大于 MIPS 指令集。

从代码实现和电路资源配置角度上看，RISCV 指令集分类更加清晰简洁，相同的控制信号在不同类型指令中位置的重复率也较高，因此可以用更少的电路资源完成译码部分的电路配置。

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（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）

心得感受:

在充分了解 prj3 中实现的 MIPS 型处理器的设计逻辑后，本次实验所需完

成的 RISCV 型处理器难度不大，且由于 RISCV 指令集更加简洁规整的特点，在译码和相关控制信号上分类也更加清晰，容易减少潜在 bug 的产生。本次实验的重点还是在于体会不同指令集对应处理设计的异同，以及进行不同指令集的性能评估比较。

建议：

在 prj4 实验中可以提高根据反汇编文件 debug 的能力要求，鼓励同学们减少对于金标准文件的依赖，增加自主检验错误能力。

致谢：

感谢陈欲晓助教和芦溶民助教在仿真加速工具上的帮助，感谢常老师对于代码规范和优化的建议。