中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2020K8009907032　姓名：唐嘉良　专业：计算机科学与技术

实验序号：04　　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）

本实验需要修改的地方并不是太多，最主要的就是指令译码、ALUop 与 RF_wdata 等信号的修改。先看译码！

RISC-V 指令格式已经足够工整，结合指令手册上的指令分类，可以把指令分为以下几类（这里通过定义宏来表示）：

```
11  //encode the instruction type
12  `define R 3'b000
13  `define I 3'b001
14  `define S 3'b010
15  `define B 3'b011
16  `define U 3'b100
17  `define J 3'b101
18
```

这在一定程度上减轻了写代码的负担。但是在硬件层面具有一些劣势，将在下面详细说明。

在指令分类之前，我们先解析指令：

```
//The instruction components
wire [6:0] opcode = Instruction_valid[6:0];
wire [4:0] rd = Instruction_valid[11:7];
wire [2:0] funct3 = Instruction_valid[14:12];
wire [4:0] rs1 = Instruction_valid[19:15];
wire [4:0] rs2 = Instruction_valid[24:20];
wire [4:0] shamt = rs2;
wire [6:0] funct7 = Instruction_valid[31:25];
wire [11:0] I_imm = Instruction_valid[31:20];
wire [11:0] S_imm = {Instruction_valid[31:25],Instruction_valid[11:7]};
wire [12:0] B_imm = {Instruction_valid[31],Instruction_valid[7],Instruction_valid[30:25],Instruction_valid[11:8],1'b0};//The branch addr is multiply of 2, so the LSB
wire [19:0] U_imm = Instruction_valid[31:12];
wire [20:0] J_imm = {Instruction_valid[31],Instruction_valid[19:12],Instruction_valid[20],Instruction_valid[30:21],1'b0};//The jump addr is multiply of 2, so the LSB
```

f 2, so the LSB is 1'b0

f 2, so the LSB is 1'b0

可以看到，我们利用_imm 后缀的几个变量提取并暂时存放指令中的立即数部分，并成功将所有可能的部分提取了出来。这里只要对照指令组成写即可，注意到 B 和 J 类型指令的 LSB 补 0。

```
//sign-extend of imm part of instruction
wire [31:0] I_ex = {{20{I_imm[11]}},I_imm};
wire [31:0] S_ex = {{20{S_imm[11]}},S_imm};
wire [31:0] B_ex = {{19{B_imm[12]}},B_imm};
wire [31:0] U_ex = {U_imm,{12{1'b0}}};//load upper imm, rest is all 0
wire [31:0] J_ex = {{11{J_imm[20]}},J_imm};

```

随后进行指令对应操作数的确定。这里注意：ISBJ 四大类指令都是符号扩展，而 U 类指令则是低 12 位填充全 0。

```
//Encode the type of instruction
wire [2:0] ins = ({opcode[5],opcode[4],opcode[2]} == 3'b110)? `R:
                                //(({opcode[6],opcode[5],opcode[2]} == 3'b000)? `I:
                                (({opcode[6],opcode[5],opcode[4]} == 3'b010)? `S:
                                (({opcode[6],opcode[2]} == 2'b10)? `B:
                                (({opcode[6],opcode[2]} == 2'b01)? `U:
                                ((opcode[3]==1'b1)? `J:`I))));    //trick:I is the most complex one
wire isshift = ((ins == `R) || (ins == `I && opcode[4] == 1'b1)) && (funct3[1:0] == 2'b01);
```

在指令译码部分，我们根据指令的 opcode 码特征进行了分类，并对指令是

否是移位指令进行了判断。这里用到了一个小技巧：三目运算符最后一个判断是前面条件之并取补，那么可以把译码最复杂的 I 类指令放在最后判断，减少了一定工作量和代码量，让整体代码风格更加简洁易读。

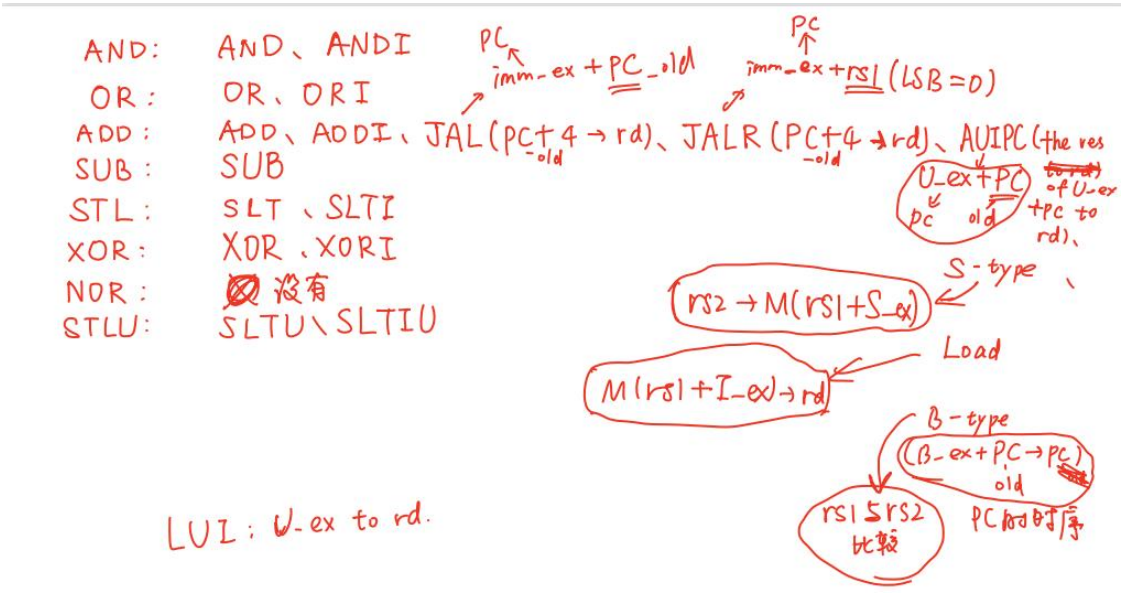
之所以要判断是否是移位指令，是因为移位指令比较繁，这里拿出来单独判断，方便后续工作。

```
224 //The control signal
225 wire [2:0] ALUop = ({3{(funct3 == 3'b111 && (ins == `I || ins == `R))}} & `AND) |//AND,ANDI
226                   ({3{(funct3 == 3'b110 && (ins == `I || ins == `R))}} & `OR) |//OR,ORI
227                                 ({3{(ins == `R && funct7[5] == 1'b0 && funct3 == 3'b000)//ADD
228                                  || (ins == `I && funct3 == 3'b000)//ADDI,JALR,LB
229                                  || (ins == `J)//JAL
230                                  || (ins == `U && opcode[5] == 1'b0)//AUIPC
231                                  || (ins == `S )//store
232                                || (opcode[6:4] == 3'b000)}} & `ADD) |//load
233                                  ({3{(ins == `R && funct7[5] == 1'b1 && funct3[0] == 1'b0) || (ins == `B && funct3[2:1] == 2'b00)}} & `SUB) |//SUB,BEQ,BNE
234                                  ({3{(funct3 == 3'b010 && ((ins == `I && opcode[4] == 1'b1) || ins == `R)) || (ins == `B && funct3[2:1] == 2'b10)}} & `STL) |//SLT,
235                                  ({3{(funct3 == 3'b100 && ((ins == `I && opcode[4] == 1'b1) || ins == `R))}} & `XOR) |//XOR,XORI
236                                  ({3{(funct3 == 3'b011 && ((ins == `I && opcode[4] == 1'b1) || ins == `R)) || (ins == `B && funct3[2:1] == 2'b11)}} & `STLU);//SLTU
237
```

```
UB) |//SUB,BEQ,BNE
== 2'b10)}} & `STL) |//SLT,SLTI,BLT,BGE

== 2'b11)}} & `STLU);//SLTU,SLTIU,BLTU,BGEU
```

这里（ALUop 的计算）是主要工作量所在，每个 ALUop 对应的指令已经在注释中指明，其中 ADD 最为复杂。为此，我总结了一张粗略的表如下：



借助这张表以及一些归纳，我将一些同类指令进行了合并判断，减少了一些判断逻辑和代码量。这里就能看到之前宏定义的缺陷：在 ins 判断这里每次都要进行三位判断！事实上，如果将 ins 归纳换为各类归纳，每次只需要进行一位判断。这里就可以看出对硬件并没有那么友好，仅仅是代码更清晰易读而已。

```
245  wire [31:0] aluA = ((ins == `U && opcode[5] == 1'b0) || ins == `J)? PC_old:rdata1;//JAL,AUIPC
246  wire [31:0] aluB = ({32{(ins == `U && opcode[5] == 1'b0)}} & U_ex) |//AUIPC
247                     ({32{(ins == `J)}} & J_ex) |
248                                        ({32{(ins == `I)}} & I_ex) |
249                                        ({32{(ins == `S)}} & S_ex) |
250                                        ({32{(ins == `R || ins == `B)}} & rdata2);//B-type is to compare rs1 and rs2
251  wire [31:0] ALU_res;
```

与此共同需要改写的还有 ALU 的两个端口：aluA 和 aluB。根据指令类型和各类指令的功能，我们确定究竟是 PC_old 还是 rdata1 加上 U/J/I/S_ex 还是 rdata2。

```
259  assign RF_wen = (current_state == WB);//must be and only be in state of WB
260  assign RF_waddr = rd;
261  assign RF_wdata = ({32{(ins == `U && opcode[5] == 1'b1)}} & U_ex) | //LUI
262                    ({32{(opcode[6:4] == 3'b000)}} & load_res) | //load
263                                    ({32{(ins == `J || (ins == `I && opcode[6] == 1'b1))}} & (PC_old + 4)) | //JAL,JALR
264                                    ({32{isshift}} & shifter_res) | //shift
265                                    ({32{(ins == `R || (ins == `I && opcode[4] == 1'b1) || (ins == `U && opcode[5] == 1'b0)) && (~isshift)}} & ALU_res);
```

当且仅当目前状态处于 WB 时拉高 RF_wen。此处无需赘述，主要改变在于 RF_wdata。由于 LUI 是将 U_ex 写入 reg，load 类指令是将计算出来的 load_res（与 mips 时计算方法一样）写入 reg，JAL 和 JALR 是将旧 PC 的值+4 写入 reg，shift 类型指令则是将移位器结果写入 reg 中。除此以外，其余写回寄存器堆的指令均是将 ALU_res 写入寄存器。

关于 PC 的跳转，实际上与 mips 差别不大，在于删去了 nop 指令的特判:

```
//Clk posedge behavior:get the next PC
always @(posedge clk) begin
                if(rst) begin
                        PC <= 32'b0;//reset PC
                end
        else if(current_state == EX) begin
            PC <= ((ins == `J || (ins == `I && opcode[6] == 1'b1))? J_addr: (PCSrc? PC_BRANCH: PC_next));//prepare to fetch next instruction
                end
        end
```

关于状态机的修改，差别也并不大，注意没有 nop 指令，按照课件所提及的进行改变即可:

```
//The calculate of next_state
always @(*) begin
    case(current_state)
            INIT:
                    next_state = IF;
                IF:
                    begin
                                if(Inst_Req_Ready) begin
                                        next_state = IW;
                                end
                                else begin
                                        next_state = IF;
                                end
                            end
                IW:
            begin
                                if(Inst_Valid) begin
                                        next_state = ID;
                                end
                                else begin
                                        next_state = IW;
                                end
                            end
                ID:
                    begin
                                next_state = EX;
                            end

                    end
            EX:
                begin
                            if(opcode[6:4] == 3'b000) begin
                                    next_state = LD;
                            end
                            else if(ins == `S) begin
                                    next_state = ST;
                            end
                            else if(ins == `B) begin
                                    next_state = IF;
                            end
                            else begin
                                    next_state = WB;
                            end
                    end
```

```
                                else begin
                                        next_state = WB;
                                end
                        end
                LD:
                    begin
                                if(Mem_Req_Ready) begin
                                        next_state = RDW;
                                end
                                else begin
                                        next_state = LD;
                                end
                        end
                ST:
                    begin
                                if(Mem_Req_Ready) begin
                                        next_state = IF;
                                end
                                else begin
                                        next_state = ST;
                                end
                        end
                RDW:
                    begin
                                if(Read_data_Valid) begin
                                        next_state = WB;
                                end
                                else begin
                                        next_state = RDW;
                                end
                        end
                WB:
                    next_state = IF;
                default:
                    next_state = INIT;
        endcase
end
```

改变主要在于 EX 状态的条件判断部分，这里的修改是平凡的，不再赘述。

```
418 //The number of mem_visit
419 reg [31:0] mem_cnt;
420 always @(posedge clk) begin
421         if(rst) begin
422                 mem_cnt <= 32'd0;
423         end
424         else if((Inst_Ready && Inst_Valid) || (Read_data_Ready && Read_data_Valid)) begin
425                 mem_cnt <= mem_cnt + 1;
426         end
427         else begin
428                 mem_cnt <= mem_cnt;
429         end
430 end
431         assign cpu_perf_cnt_2 = mem_cnt;
432 ////////////////////////////////////////////
433 endmodule
```

最后我增加了一个内存访问数的性能计数器，用以比较 RISC-V 和 MIPS 两大指令集，详见五。

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）

遇到了一个问题：因为 RF_wdata 判断错误而导致各种信号发生错误。这个 bug 极其难 de，因为寄存器堆是会保存错误的，尽管在写入的时候不会产生错误信号。于是需要从出错点往前寻找很多个时钟周期才能定位到当时的写入错误。

三、对讲义中思考题（如有）的理解和回答

无思考题。至于 RISC-V 和 MIPS 指令集的区别，详见五。

四、在课后，你花费了大约___6___小时完成此次实验。

要修改的地方不多，但是不得不说也有一定复杂度。

代码编写和 debug 等实验主体过程总计耗时 6h（忽视 serve 云平台运行时间）。

总共耗时 9h（算上 serve 云平台运行时间）。

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）

对 RISC-V 和 MIPS 指令集区别的思考和分析：

	A	B	C	D	E	F	G	H	I	J
1	MIPS	15pz	bf	dinic	fib	md5	qsort	queen	sieve	ssort
2	时钟周期数/次	325609339	28347022	1051701	109642732	247347	432722	4252762	728517	32131826
3	指令数/次	5287713	559051	19328	2525724	5229	8341	80858	16480	728291
4	访存次数/次	7400500	653553	23957	2530129	5702	9856	95269	16818	739880
5										
6	RISC-V	15pz	bf	dinic	fib	md5	qsort	queen	sieve	ssort
7	时钟周期数/次	331030067	23761272	915840	110785115	235220	480434	4269350	455516	27364819
8	指令数/次	5224461	452834	16671	2549505	4895	9460	81470	10175	619025
9	访存次数/次	7334450	547338	20704	2553822	5431	10975	95881	10511	630317
10										
11										
12										
13	RISC-V相对于MIPS的性能比较	15pz	bf	dinic	fib	md5	qsort	queen	sieve	ssort
14	时钟周期数相对浮动	1.64%	-19.30%	-14.83%	1.03%	-5.16%	9.93%	0.39%	-59.93%	-17.42%
15	指令数相对浮动	-1.21%	-23.46%	-15.94%	0.93%	-6.82%	11.83%	0.75%	-61.97%	-17.65%
16	访存次数相对浮动	-0.90%	-19.41%	-15.71%	0.93%	-4.99%	10.20%	0.64%	-60.00%	-17.38%
17										

将 9 个样例的 3 个性能计数器的数据统计成表，可以得到一些结论。

一般来说，RISC-V 指令集下的 CPU 性能比 MIPS 强，完成一个操作所需指令较少。

从表格中我们可以看到，完成同一个程序，RISC-V 所需指令数基本上都比 MIPS 要少，并且有几个样例甚至少很多。可见完成一个操作平均所需指令数更少。在 sieve 中甚至优化了 60%左右。为此，我在网上查询相关信息，却只得到了“MIPS 指令集垂直性太好导致实现同一操作需要更多指令”的回答。

当然，不乏例外情况：fib、qsort 和 queen 三个样例出现反常。可见 RISC-V 指令集的性能并非严格优于 MIPS；然而，即便是这三个反常样例，所多出的比例也不高。最多的是 ssort，也仅仅劣化了 17%，相比 sieve 的 60%并不算高。

为了验证我的猜想，我对九个样例的性能表现求了平均，得到平均性能提升率，如下：

平均优化率	
平均时钟周期数相对浮动	-11.52%
平均指令数相对浮动	-12.61%
平均访存次数相对浮动	-11.85%

猜想得到验证。进一步地，我们可以看出 **RISC-V 指令集下的 CPU 在平均性能（时钟周期数、指令数、访存次数）上比 MIPS 优化了约 12%**。由于实验课样例很有限，该结论并不能保证正确性，但是大致趋势是可以看出来的。

此外，RISC-V 有个明显的优点是**指令格式规整，所以译码成本比 MIPS 要低；另外，基于 RISC-V 指令集的 CPU 数据通路也更加简单**。事实上在代码编写过程中我就意识到了这一点。

一个思考是：RISC-V 到底有没有“NOP 指令”？事实上，RISC-V 中也有类似 MIPS 的 NOP 指令的指令，只不过本实验并没有留给我们编写。

另一个思考是：为什么 RISC-V 的指令里立即数会被划分为好几块?

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

与MIPS相比，RISC-V具有更加规整的指令格式

第一个想到的是，在 R-type 指令中，指令已经被预先确定为了这样的六段，在设计其它指令的时候由于没有 func7 等因素，为了保持指令规整，必须将立即数拆分开来填入不同的部分。此外，助教学长还提示我：这样拆分会让立即数的各个部分可能取值减少，但我对此存疑。

在上个实验的实验报告中，我忘记了提一个建议。**在此，强烈建议：删除矩阵乘法和水仙花数！删除矩阵乘法和水仙花数！删除矩阵乘法和水仙花数！**这两个样例实在太耗时了，严重影响同学们的云平台测试体验（例如，推到 serve 上跑，然后睡觉，结果第二天睡醒一看，发现 failed）。建议寻找其它的具有相

同或者更好的测试能力但是耗时较短的测试样例作为替代。

本次实验帮助我深刻体会到了 RISC-V 指令集的特点——格式规整、精简、性能优越等，并借助基于 RISC-V 的 CPU 的实现，对比 MIPS，分析总结了他们之间的异同和优缺点。

致谢：感谢孙维铭同学，帮助我 de 出了**一个极其隐蔽的 bug! 该 bug 横跨几千纳秒的波形图!**