中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929001　　姓名：谢嘉楠　　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

一、实验目的

1. 基于实验项目3已实现的微结构，设计支持 RISC-V 32-bit 整型指令集($RV32I$)的功能型处理器。
2. 通过功能和性能评估，对比 RISC-V/MIPS 指令集**译码器的实现开销**，理解 RISC-V 指令格式的设计思想。

二、实验过程

这次实验需要实现对 RISCV 中 37 条基本指令的支持。因为有了在实验 2 和实验 3 中涉及 MIPS 处理器的经验，再加上 RISCV 相比于 MIPS 具有更规整的指令格式、更清晰的指令分类，在整体设计和实现的过程中我都会更加熟练，花费的时间也更少。

我主要的设计过程还是先根据 RISCV 指令集手册，把每条指令在执行过程中，CPU 中各个模块的输入信号写下来，整理成一张译码表格。因为主要的数据通路与 MIPS 是一样的，所以很大一部分 verilog 代码框架都可以复用，所以接下来我就直接在 MIPS 处理器的框架上进行修改，主要修改的部分在指令的译码部分，以及各个模块输入的选择信号和使能信号，其他的部分，像状态机的状态跳转不需要太多修改，像性能计数器也不需要修改。最后就进行云平台仿真，根据波形找出错误。

1. 指令译码表的整理

第一步是先搞清楚每条指令要实现的操作，这个在 RISCV 指令集手册上可以看到，不过 RISCV 指令集手册是把指令一类一类地讲的，它不像 MIPS 手册一样是每一条指令单独一页，然后对指令操作的描述也没有用RTL代码表现出来，而是用英文描述的，这样看起来就没有那么方便直观，还是有一些不适应。

其实大部分指令的操作和 MIPS 是一模一样的（如 R_type 和 I_type 的计算指令，load 和 store 指令），有几条指令会和 MIPS 中的不同（如 branch 指令，jump 指令）。同时有些指令在 RISCV 中被去掉了（如 lwl,lwr,swl,swr 指令），还有一些指令是新增的（如 luipc 指令、I_type 的 shift 指令）。每一类指令的输入数据都整理下下面的表格中了。

load immediate（加载立即数）指令

指令类型		指令名	opcode	ALU			寄存器堆写		
指令格式	指令功能			A	B	ALUop	wen	RF_waddr	RF_wdata
U-type	load immediate	LUI	0110111				1	rd	{U_immediate,12'b0}
		LUIPC	0010111	PC	U_immediate	010			result1

calculate（计算）指令

指令类型		指令名	opcode	寄存器堆读		ALU			寄存器堆写		
指令格式	指令功能			RF_raddr1	RF_raddr2	A	B	ALUop	wen	RF_waddr	RF_wdata
R_type	calculate	ADD	0110011	rs1	rs2	rdata1	rdata2	010	1	rd	result1
		SUB						110			
		AND						000			
		OR						001			
		XOR						100			
		SLT						111			
		SLTU						011			
I_type	calculate	ADDI	0010011	rs1		rdata1	signextended(I_immediate)	010	1	rd	result1
		ANDI						000			
		ORI						001			
		XORI						100			
		SLTI						111			
		SLTIU						011			

shift（移位）指令

指令类型		指令名	opcode	寄存器堆读		shifter			寄存器堆写		
指令格式	指令功能			RF_raddr1	RF_raddr2	shift_data	shift_amount	shiftop	wen	RF_waddr	RF_wdata
R_type	shift	SLL	0110011	rs1	rs2	rdata1	rdata2[4:0]	00	1	rd	result2
		SRL						10			
		SRA						11			
I_type	shift	SLLI	0010011	rs1		rdata1	I_immediate[4:0]	00	1	rd	result2
		SRLI						10			
		SRAI						11			

jump（跳转）指令

指令类型		指令名	opcode	寄存器堆读		ALU			跳转		寄存器堆写		
指令格式	指令功能			RF_raddr1	RF_raddr2	A	B	ALUop	跳转地址	更新条件	wen	RF_waddr	RF_wdata
J_type	jump	JAL	1101111			PC	signextended (J_immediate)	010	result1	1	1	rd	PC+4
I_type		JALR	1100111	rs1		rdata1	signextended (I_immediate)	010	{result1[31:1],1'b0}	1			

branch（分支）指令

指令类型		指令名	opcode	寄存器堆读		ALU			跳转	
指令格式	指令功能			RF_raddr1	RF_raddr2	A	B	ALUop	跳转地址	更新条件
B_type	branch	BEQ	1100011	rs1	rs2	rdata1	rdata2	110	PC+signextended (B_immediate)	zero
		BNE								~zero
		BLT						111		result1[0]
		BGE								~result1[0]
		BLTU						011		carryout
		BGEU								~carryout

load（加载）指令

指令类型		指令名	opcode	寄存器堆读	ALU			内存访问			寄存器堆写		
指令格式	指令功能			RF_raddr1	A	B	ALUop	Address	MemRead	MemWrite	wen	RF_waddr	RF_wdata
I_type	load	LB	0000011	rs1	rdata1	signextended (I_immediate)	010	{result1[31:2],2'b0}	1	0	1	rd	signextended(MDR[result1[1:0]||000:+8])
		LH											signextended(MDR[result1[1:0]||000:+16])
		LW											MDR
		LBU											zeroextended(MDR[result1[1:0]||000:+8])
		LHU											zeroextended(MDR[result1[1:0]||000:+16])

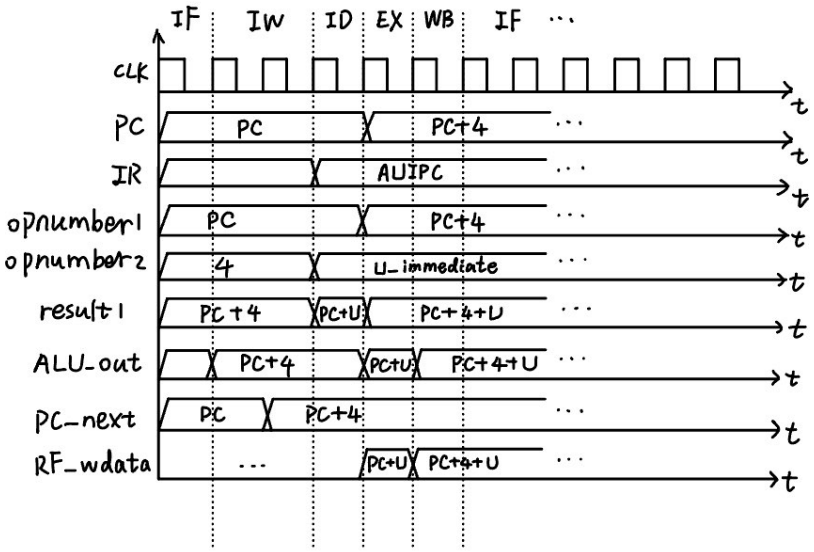
store（存数）指令

指令类型		指令名	opcode	寄存器堆读		ALU			shifter			内存访问				
指令格式	指令功能			RF_raddr1	RF_raddr2	A	B	ALUop	shift_data	shift_amount	shiftop	Address	MemRead	MemWrite	WriteData	Write_strb
S_type	store	SB	0100011	rs1	rs2	rdata1	signextended (S_immediate)	010	rdata2	{result1[1:0],3'b0}	00	{result1[31:2],2'b0}	0	1	result2	result1[1]?(result1[0]4'b1000:4'b0100):(result1[0]?4'b0010:4'b0001)
		SH														result1[1]?(4'b1100):(4'b0011)
		SW														4'b1111

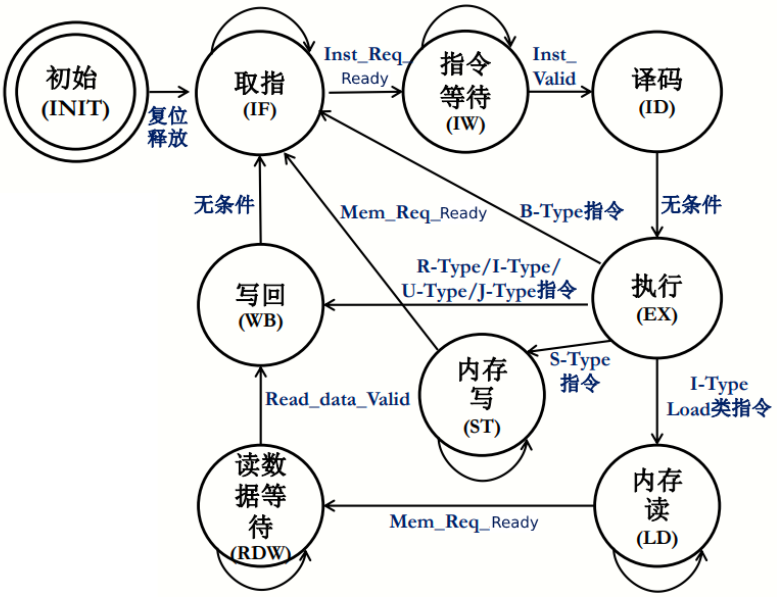
2. 多周期状态机的控制

更换到 RISCV 指令集后，CPU 的状态机几乎不需要改变。还是包括初始状态在内的 9 个状态。依旧使用 one-hot 码来编码 9 个状态。唯一需要注意的可能就是状态之间的跳转，可能会因为指令分类与功能上的变化而会有一点小变化。比如说，MIPS 中的 NOP 指令需要再译码状态之后直接跳回取指状态，但在 RISCV 中，NOP 指令通过加法指令$ADDI\ x_0\ x_0\ 0$来模拟实现（0 号寄存器中始终存着$32'b0$），这样它和所有I_type计算指令一样，要经过执行、写回状态，就不需要单独考虑，节省了译码的开销。

此外，还有新增的$auipc$指令，它和lui指令一样，经过执行、写回状态。lui写回寄存器的值就是$U_immediate$，而$auipc$写回寄存器的值则是在执行阶段计算出的$PC+4+U_immediate$。这条指令的一个明显的用处就是PC相对寻址，不过特别需要注意的是，实际上偏移量是相对于下一条指令（$PC+4$）而言的。下图是$auipc$指令执行过程中一些信号的波形变化示意图。



其他的指令相较于在$MIPS$中所经历的状态都没有变化，访存过程中的握手信号也没有变化，所以状态转化图（如下图）也几乎和$MIPS$是一样的。



3. 指令分类与译码

RISCV 指令集的指令格式更多，指令格式与功能的对应更加清晰。比如$rs1$、$rs2$字段代表寄存器堆的两个读地址，而rd字段代表寄存器堆的写地址，这 3 个字段在不同格式的指令中具有相同的位置，使得对应的端口的信号赋值非常方便。同时，RISCV 指令的 opcode 位于指令的低字段，而 funct3 和 funct7 也具有固定的位置，用于对同一类指令继续细分。

而 RISCV 指令与 MIPS 最大的不同可能就是立即数了，MIPS 的立即数都是连续分布在指令的一整段空间的，而 RISCV 的立即数有更多形式，而且会被拆分成几段分布在指令中。这应该是为了使得前面的$rs1$、$rs2$、$funct$等字段具有不变的位置，从而提高译码的效率。

不过可以看到，实际上我们所实现的 39 条指令还远没有充分利用指令的 opcode 和 funct 字段，这些还没有使用到的字段，其实是为 RISCV 指令集的扩展指令服务的。

下面是对 RISCV 中指令字段的截取，以及对 RISCV 指令的分类。

```
// 截取指令中的不同字段，用于译码时使用
    wire [ 6:0] opcode      = IR[ 6: 0];
```

```
    wire [ 4:0] rd          = IR[11: 7];
    wire [ 2:0] funct3      = IR[14:12];
    wire [ 4:0] rs1         = IR[19:15];
    wire [ 4:0] rs2         = IR[24:20];
    wire [ 6:0] funct7      = IR[31:25];
    wire [11:0] I_immediate = IR[31:20];
    wire [11:0] S_immediate = {IR[31:25],IR[11:7]};
    wire [12:0] B_immediate = {IR[31],IR[7],IR[30:25],IR[11:8],1'b0};
    wire [31:0] U_immediate = {IR[31:12],12'b0};
    wire [20:0] J_immediate = {IR[31],IR[19:12],IR[20],IR[30:21],1'b0};
```

```
// 指令分类阶段
    // 对指令格式进行分类
    wire R_type            = opcode[5] && opcode[4] && ~opcode[2];
    wire I_type            = ~opcode[5] && opcode[4] && ~opcode[2];
    // 对指令功能进行分类
    wire R_calculate       = R_type && ~R_shift && ~multiply;
    wire R_shift           = R_type && (funct3[1:0]==2'b01);
    wire multiply          = R_type && (funct7[0]);
    wire I_calculate       = I_type && ~I_shift;
    wire I_shift           = I_type && (funct3[1:0]==2'b01);
    wire calculate         = R_calculate || I_calculate;
    wire shift             = R_shift || I_shift;
    wire jump              = opcode[6] && opcode[2];
    wire branch            = opcode[6] && ~opcode[2];
    wire load              = ~opcode[5] && ~opcode[4];
    wire store             = ~opcode[6] && opcode[5] && ~opcode[4];
    wire load_imm          = ~opcode[6] && opcode[2];
    wire jal               = jump && opcode[3];
    wire jalr              = jump && ~opcode[3];
```

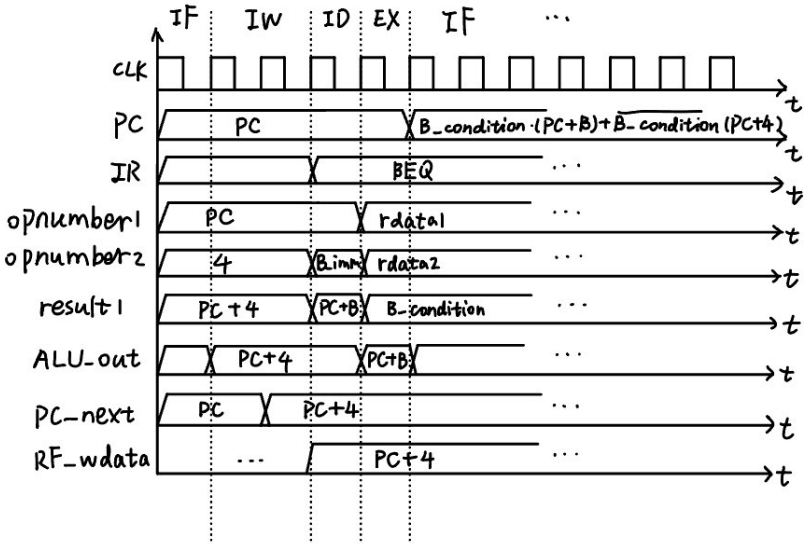
接下来，还需要对 CPU 内各个模块的输入端口进行赋值。这一步骤也是主要在 MIPS 处理器的基础上进行修改的，得益于 RISCV 与 MIPS 的相似性，其实修改的部分并不多。

寄存器堆部分，主要是写回数据的改动，需要增加$auipc$指令的写回数据，但因为 load 指令没有了lwl和lwr指令，写回数据的选择反而可以删掉一大块最复杂的部分，变得简单了一些。最后还是根据指令功能分类，将不同情况的写回数据与对应的使能信号“&”“一下，最后“|”起来。

```
    assign RF_wdata
= load_wdata | calculate_wdata | shift_wdata | jump_wdata | load_imm_wdata
```

ALU 部分的改动应该是最大的。需要使用 ALU 的指令有 calculate、branch/jump、load/store 以及 auipc 指令，几乎包括了所以指令。calculate、auipc 指令对操作数的运算、load/store 指令对访存地址的计算都在执行阶段进行，而 branch/jump 指令对目标地址的计算则需要在译码阶段进行，因为在执行阶段 branch 指令需要用 ALU 进行对跳转条件的判断。这样，可以将 PC+4 的计算放在取指和指令等待(IF/IW)阶段，然后对于非 branch/jump 指令，可以在译码阶段执行阶段(ID→EX)的上升沿将 PC+4 的结果弹入 PC 中，而对于 branch(/jump)指令，则需要等到跳转条件被计算出来，并且目标地址被存到 ALUout 寄存器中后(EX→WB)再更新 PC 值。

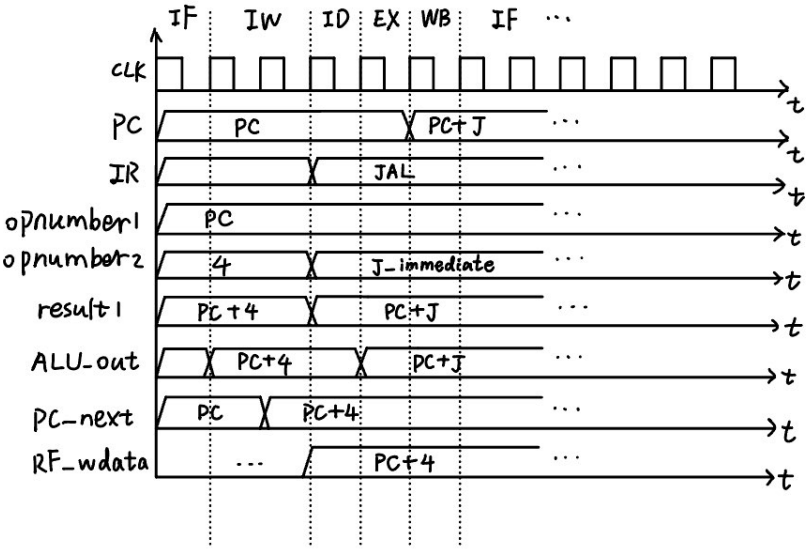
下面是 branch 指令（以 beq 为例）执行过程中的相关信号波形变化的示意图。



ALU 各种情况的使能信号比较多，所以我将各个使能信号绑在一起形成一串位宽的 src 信号，这样在使用时写起来会方便很多，也清晰很多。

```
    // ALU 的 opnumber 的赋值使能信号
    wire [2:0] ALUsrc_state =
{(current_state[3:1]==3'b000),current_state[3],current_state[2]||current_state[1]};
    wire [1:0] ALUsrcA      = {jalr,load_imm};
    wire [2:0] ALUsrcB1     = {branch,jal,jalr};
    wire [3:0] ALUsrcB2     = {load_imm,branch || R_calculate || multiply,load ||
I_calculate,store};
```

关于 jump 指令，RISCV 与 MIPS 有很大的不同。MIPS 中的 jump 指令，即无条件跳转指令的目标地址生成不需要使用 ALU（不需要加法），而是通过直接对立即数进行扩展，或者从寄存器中直接读取得到的，而其写回寄存器的返回地址 PC+4 是在 ALU 上算出来的，所以跳转地址的生成与返回地址的生成是互不干扰的。但是在 RISCV 中，jump 指令的目标地址也需要用 ALU 计算得到（其中 jal 是 PC+J_immediate，jalr 是 rdata1+I_immediate），这样两次计算就会相互冲突。因为返回地址 PC+4 是先计算的，如果将它存在 ALUout 中，等到写回阶段时会发现结果早就被覆盖掉了；如果将它存在 PC 中，则会影响目标地址 PC+J_immediate 的计算，而且同样地，到写回阶段 PC 值已经更新为目标地址了。**所以，我们需要一个额外的寄存器 PC_next，专门用来存放预先计算出的 PC+4 值。**下面是执行 jal 指令时相关信号波形变化的示意图。



这个问题我一开始在设计时并没有关注到，其实是在查看仿真波形时，对照金标准信号时才发现有一个 PC_next 寄存器，才意识到为了支持 jump 指令的实现，确实需要增加一个 PC 寄存器。现在回想起来，

它应该算是这个实验中我遇到的最核心的困难。

剩下的两个模块，移位器 shifter 和内存读写控制信号，需要改动的地方也比较少，主要修改的是 shiftop 和 write_strb 的赋值，但它们的思路和 MIPS 中是一样的，只是因为 RISCV 指令中 funct 编码的变化，需要重新做一次译码。

到这里就基本完成了从 MIPS 处理器到 RISCV 处理器的修改。

三、实验总结

1. RISCV 与 MIPS 指令集**译码开销的定性对比**

我觉得$RISCV$和$MIPS$之间的差别主要其实在译码的开销上。$RISCV$用于控制和译码的组合逻辑开销要比 MIPS 更少。

首先是指令数量上，$RISCV$指令数量(在本实验中)更少，减少了译码的分类情况。

其次 RISCV 指令分类更加清晰。RISCV 基本上是一种指令格式对应一类功能的指令，这样在指令分类时就不需要将格式分类与功能分类混杂在一起。每一类功能的指令所使用的立即数字段、地址字段基本就是固定的，不像$MIPS$中这样，每一类功能的指令可能使用很多种指令格式。这样每一类指令的译码就更加方便，每一个模块的输入信号就只要按照指令功能分类即可，在编写时也更不容易出错。有一个直观的体现就是在$verilog$代码中，$MIPS$的CPU组合逻辑部分，有一些模块的输入需要很长的选择与控制逻辑，(例如寄存器堆的写数据RF_wdata，但是 RISCV 中就要好很多，尤其是$RISCV$中没有lwl和lwr指令)，导致一行会比较长，但是在$RISCV$中每一行对应的一类指令基本上就只有一种情况，所以每一行都比较短，看起来也更加清晰。

虽然看不到实际$FPGA$上用了多少个晶体管/LUT，不过在整理译码表和写$verilog$代码时确实可以感受到 RISCV 的设计使得译码更加精简，更加方便，同时这也减少了控制单元的一些延迟，有利于提高时钟频率，加快处理器速度。

2. RISCV 与 MIPS 指令集**运行性能的定量对比**

我在两个 CPU 中都加入了 3 个性能计数器，分别统计不同测试程序运行时的**时钟周期数(clk cycle)、指令周期数(instruction cycle)以及访存次数(load/store 指令数)**。我整理了 FPGA_run 阶段中的 9 组 microbench 测试结果，将它们列在了表格中。

首先是 RISCV 处理器的统计结果。

FPGA测试样例	RISCV				
	时钟周期数(clk cycle)	指令周期数(inst cycle)	**CPI(cycle per instruction)**	访存次数(load/store指令数)	**比例访存指令**
15-puzzle search	526452394	5224463	**100.77**	3228745	**61.80%**
Brainf**k interpreter	38813360	452836	**85.71**	100467	**22.19%**
Dinic's maxflow algorithm	1474195	16673	**88.42**	5549	**33.28%**
Fibonacci number	181088608	2549507	**71.03**	5286	**0.21%**
MD5 digest	384769	4897	**78.57**	625	**12.76%**
Quick sort	783051	9462	**82.76**	2447	**25.86%**
Queen placement	6894571	81472	**84.63**	26755	**32.84%**
Eratosthenes sieve	744622	10177	**73.17**	453	**4.45%**
Suffix sort	44731164	619027	**72.26**	18697	**3.02%**
AVERAGE			**81.92**		

下面是 MIPS 处理器的统计结果。

FPGA测试样例	MIPS				
	时钟周期数(clk cycle)	指令周期数(inst cycle)	**CPI(cycle per instruction)**	访存次数(load/store指令数)	**比例访存指令**
15-puzzle search	527094576	5287715	**99.68**	3231787	**61.12%**
Brainf**k interpreter	46342672	559053	**82.89**	100464	**17.97%**
Dinic's maxflow algorithm	1703294	19330	**88.12**	6509	**33.67%**
Fibonacci number	179408372	2525726	**71.03**	5373	**0.21%**
MD5 digest	404158	5231	**77.26**	561	**10.72%**
Quick sort	704021	8343	**84.38**	2447	**29.33%**
Queen placement	6849670	80860	**84.71**	26755	**33.09%**
Eratosthenes sieve	1191840	16482	**72.31**	456	**2.77%**
Suffix sort	52507202	728293	**72.10**	19289	**2.65%**
AVERAGE			**81.39**		

对比两表的数据，RISCV 的时钟周期数总体来说比 MIPS 要少，如果在时钟周期相同的情况下，RICSV 会比 MIPS 更快一些。这可能是因为有几条指令，RISCV 把原本 MIPS 中多条指令的操作用一条指令就完成了(例如 LUIPC)，不过毕竟 RISCV 是精简指令集，它的指令也不会很复杂，并不会合并 MIPS 中的很多操作，所以提升并不明显。

其他性能测试得到的数据，如 CPI 和访存指令的比例，两个指令集都比较接近，没有明显的差距。这说明各个程序在编译时，编译器得到的两个指令集的机器码数量大致相同，而且访存指令所占的比例也大致相同。

3. MIPS 指令集如今已不再被被使用，而 RISCV 指令集的使用却越来越广泛，那么同为精简指令集，RISCV 指令集相较于 MIPS 指令集，还有哪些发展与提升呢？课后查阅过一些资料，我觉得主要可能有下面的三点吧。

开放性：RISC-V 是一种开放的指令集架构，可以自由地使用、修改和分发，而 MIPS 则是专有的指令集架构，需要获得许可才能使用。

可扩展性：RISC-V 指令集可以灵活地扩展，可以添加自定义指令和扩展指令集，而 MIPS 指令集则比较固定，不易扩展。

简洁性：RISC-V 指令集相对于 MIPS 指令集来说更为简洁，指令数目更少，编码方式也更为简单，这使得编译器和硬件实现更容易。

4. RISCV 指令集为什么去掉了 lwl 和 lwr 指令？

RISC-V 指令集没有 lwl 和 lwr 指令，是因为这两个指令在处理器实现上非常复杂，而且对于大多数应用程序也不是必需的。

lwl 和 lwr 指令用于在一个字中加载部分字节，具体来说，lwl 指令在一个字的左侧加载部分字节，而 lwr 指令在一个字的右侧加载部分字节。在处理器实现上，这两个指令需要进行字节对齐、字节顺序转换等复杂的操作，增加了处理器的复杂度和成本。

在 RISC-V 中，为了简化指令集和降低处理器实现的复杂度和成本，采用了一种简单的加载方式，即通过 lw 指令加载整个字，然后通过位运算和移位操作来获取需要的字节。

虽然这种方式可能会增加一些指令数量和运算复杂度，但是对于大多数应用程序来说，这种方式已经足够高效和灵活了。同时，RISC-V 也提供了一些扩展指令集，如 RV32M 和 RV64M，其中包含了一些更高级的指令，如位操作指令和乘除指令，可以进一步提高处理器的性能和功能。

5. 心得

总之，通过这次实验，我能够明显地感觉到自己对于简单 CPU 设计和实现的流程更加熟悉了，查看波形定位错误的环节真的缩短了很多。然后也通过实践，对 RISCV 指令集的一些优点和特色有了一定的了解，还是很有成就感的。