

中国科学院大学

《计算机组成原理(研讨课)》实验报告

姓名 韦欣池 学号 2022K8009907004 专业 计算机科学与技术
实验项目编号 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 推送到实验课 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(比如 Verilog HDL 关键代码段 {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)

• 支持 RV32I 的功能型处理器设计。

本次实验的任务是将 MIPS 指令集改成 RV32I, 并实现其中的 37 条指令, 指令数量比 MIPS 少。首先要做的是对 RV32I 中的这 37 条指令编写新的译码表:

指令类型	序号	指令名	指令码					指令含义				
R-Type			funct7(31:25)	rs2(24:20)	rs1(19:15)	funct3(14:12)	rd(11:7)	opcode(6:0)				
	1	add	0000000	rs2	rs1	000	rd	0110011	rd <- rs1 + rs2			
	2	sub	0100000			001			rd <- rs1 - rs2			
	3	and	0000000			111			rd <- rs1 & rs2 (逻辑与运算)			
	4	or				110			rd <- rs1 rs2 (逻辑或运算)			
	5	xor				100			rd <- rs1 ^ rs2 (逻辑异或运算)			
	6	sll				001			rd <- rs1 << rs2 (逻辑左移)			
	7	slt				010			rd <- (rs1 < rs2) (有符号比较, 小于[rd]置1, 大于等于[rd]置0)			
	8	sltu				011			rd <- (rs1 < rs2) (无符号比较, 小于[rd]置1, 大于等于[rd]置0)			
	9	srl				101			rd <- rs1 >> rs2 (逻辑右移)			
10	sra	0100000				rd <- rs1 >> rs2 (算数右移)						
指令类型	序号	指令名	指令码					指令含义				
I-Type			imm[11:0](31:20)		rs1(19:15)	funct3(14:12)	rd(11:7)	opcode(6:0)				
	11	addi	imm[11:0]	rs1	rd	000	0010011	rd <- rs1 + imm				
	12	slti				010		rd <- (rs1 < imm) (有符号比较, 小于[rd]置1, 大于等于[rd]置0)				
	13	sltiu				011		rd <- (rs1 < imm) (无符号比较, 小于[rd]置1, 大于等于[rd]置0)				
	14	xori				100		rd <- rs1 ^ imm				
	15	ori				110		rd <- rs1 imm				
	16	andi				111		rd <- rs1 & imm				
	17	lb				000		0000011	rd <- M[rs1 + imm][7:0] (符号位扩展)			
	18	lh				001			rd <- M[rs1 + imm][15:0] (符号位扩展)			
	19	lw				010			rd <- M[rs1 + imm][31:0] (符号位扩展)			
	20	lbu				100			rd <- M[rs1 + imm][7:0] (零扩展)			
	21	lhu				101			rd <- M[rs1 + imm][15:0] (零扩展)			
	22	jalr				000		1100111	rd <- PC + 4; PC <- rs1 + imm			
	序号	指令名				指令码					指令含义	
	23	slli				0000000	shaamt(24:20)	rs1(19:15)	funct3(14:12)	rd(11:7)	opcode(6:0)	
	24	srl				0100000	shaamt	rs1	001	rd	0010011	rd <- rs1 << imm[4:0] (逻辑左移)
		0100000						101			rd <- rs1 >> imm[4:0] (逻辑右移)	
								rd <- rs1 >> imm[4:0] (算数右移)				

图 1: R-Type 和 I-Type 类型指令

指令类型	序号	指令名	指令码						指令含义
S-Type	26	sb	imm[11:5] (31:25)	rs2(24:20)	rs1(19:15)	funct3(14:12)	imm[4:0] (11:7)	opcode(6:0)	$M[rs1 + imm[7:0]] \leftarrow rs2[7:0]$
	27	sh	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	$M[rs1 + imm[15:0]] \leftarrow rs2[15:0]$
	28	sw				010			$M[rs1 + imm[31:0]] \leftarrow rs2[31:0]$
指令类型	序号	指令名	指令码						指令含义
B-Type	29	beq	imm[12:10:5] (31:25)	rs2(24:20)	rs1(19:15)	funct3(14:12)	imm[4:1:11] (11:7)	opcode(6:0)	if $rs1 == rs2$, then $PC \leftarrow imm$
	30	bne				001			if $rs1 \neq rs2$, then $PC \leftarrow imm$
	31	blt				100			if $rs1 < rs2$, then $PC \leftarrow imm$
	32	bge				101			if $rs1 \geq rs2$, then $PC \leftarrow imm$
	33	bltu				110			if $rs1 < rs2$, then $PC \leftarrow imm$ (无符号比较)
	34	bgeu				111			if $rs1 \geq rs2$, then $PC \leftarrow imm$ (无符号比较)
指令类型	序号	指令名	指令码						指令含义
U-Type	35	lui	imm[31:12] (31:12)				rd(11:7)	opcode(6:0)	$rd \leftarrow imm \ll 12$
	36	auipc	imm[31:12]				rd		$rd \leftarrow PC + (imm \ll 12)$
指令类型	序号	指令名	指令码						指令含义
J-Type	37	jal	imm[20:10:111:19:12] (31:12)				rd(11:7)	opcode(6:0)	$rd \leftarrow PC + 4; PC \leftarrow imm$

图 2: S B U J Type 类型指令

然后我们需要做的就是将需要修改的代码进行修改。

第一个需要修改的地方是状态转移图的部分, 根据讲义要求, 我们需要分支指令在执行阶段(更新 PC 值)后立即进入下一条指令的取指阶段。并且 R-Type 类型指令在执行阶段后进入 WB 阶段。下图为修改后的状态转移图:

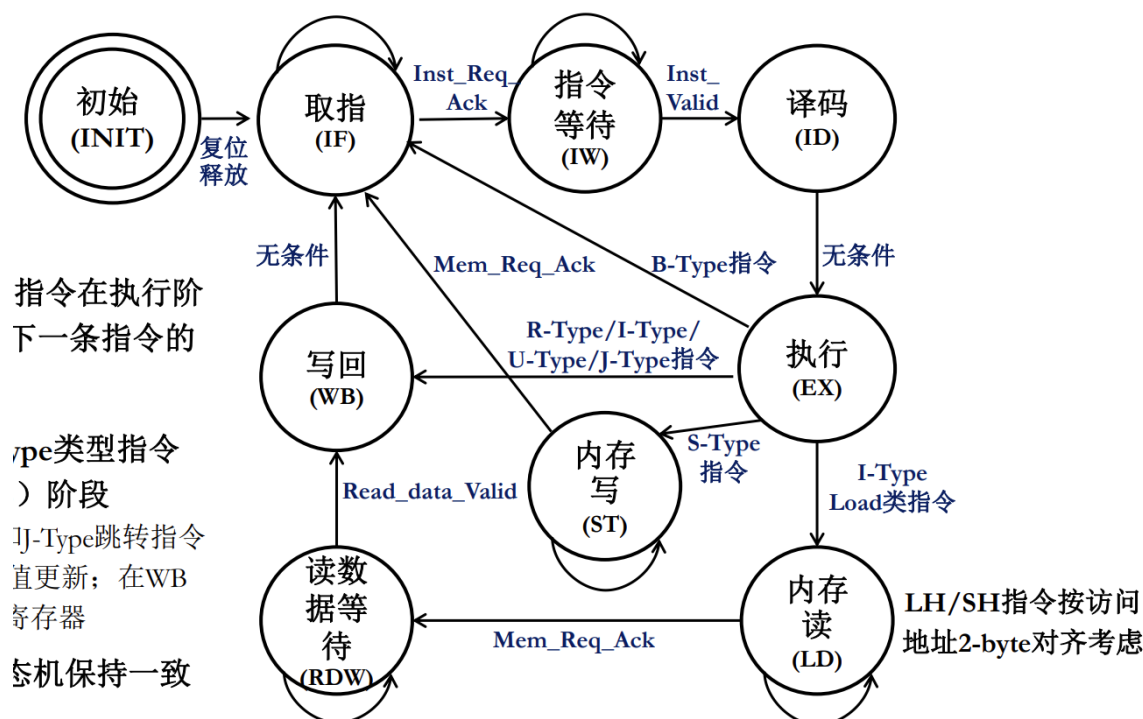


图 3: 状态转移图

对应的代码部分的修改为:

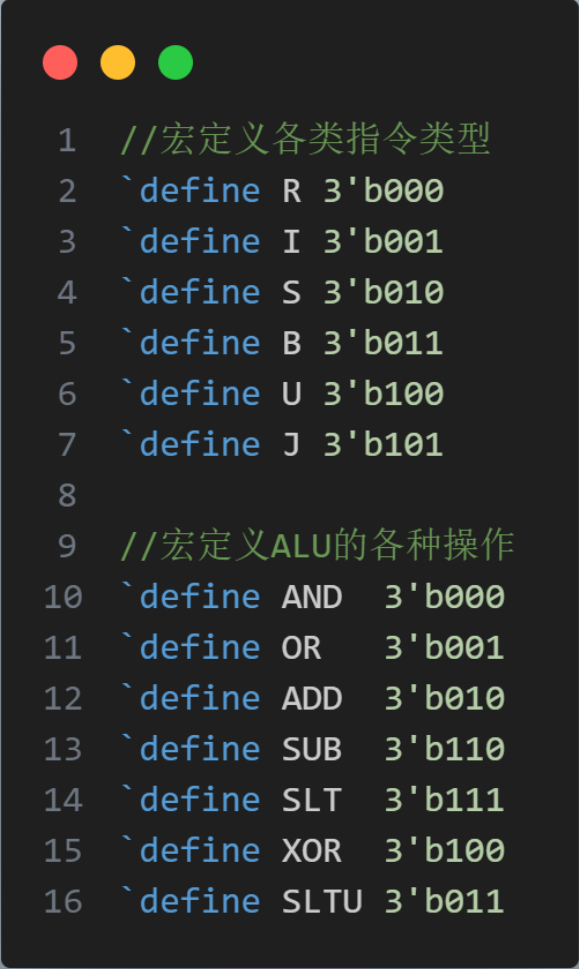
```

1 //第二段：根据状态机当前状态和输入信号，描述下一状态的计算逻辑
2 always@(*)
3 begin
4     case(current_state)
5     INIT: next_state = IF;
6     IF: begin
7         if(Inst_Req_Ready)
8             next_state = IW;
9         else
10            next_state = IF;
11        end
12    IW: begin
13        if(Inst_Valid)
14            next_state = ID;
15        else
16            next_state = IW;
17        end
18    ID: begin
19        next_state = EX;
20    end
21    EX: begin
22        if(ins == `B) //B-Type指令
23            next_state = IF;
24        else if( (ins == `R || ins == `U || ins == `J || (ins == `I && opcode[6:4] != 3'b0)) //R-Type指令, I-Type运算指令, JAL指令
25            next_state = WB;
26        else if( (ins == `S) //S-Type指令
27            next_state = ST;
28        else if( (ins == `I && opcode[6:4] == 3'b0) //I-Type Load指令
29            next_state = LD;
30        else
31            next_state = IF;
32        end
33    LD: begin
34        if(Mem_Req_Ready)
35            next_state = RDW;
36        else
37            next_state = LD;
38        end
39    ST: begin
40        if(Mem_Req_Ready)
41            next_state = IF;
42        else
43            next_state = ST;
44        end
45    RDW: begin
46        if(Read_data_Valid)
47            next_state = WB;
48        else
49            next_state = RDW;
50        end
51    WB: next_state = IF;
52    default:
53        next_state = current_state;
54    endcase
55 end

```

图 4: 状态转移图部分代码

然后是组合逻辑部分的代码修改,我增加了 ins 变量用于表示指令的种类,同时宏定义了 ALU 的各类操作:



```
1 //宏定义各类指令类型
2 `define R 3'b000
3 `define I 3'b001
4 `define S 3'b010
5 `define B 3'b011
6 `define U 3'b100
7 `define J 3'b101
8
9 //宏定义ALU的各种操作
10 `define AND 3'b000
11 `define OR 3'b001
12 `define ADD 3'b010
13 `define SUB 3'b110
14 `define SLT 3'b111
15 `define XOR 3'b100
16 `define SLTU 3'b011
```

图 5: 宏定义部分

按照 RV32I 的指令操作码的格式进行重新划分, 并且按照指令集讲义中的要求将各类指令的立即数进行了扩展:

```

1 //指令码
2 assign opcode = Instruction_current[6:0]; //opcode为Instruction_current的低7位
3 assign rd = Instruction_current[11:7]; //rd为Instruction_current的7-11位
4 assign funct3 = Instruction_current[14:12]; //funct3为Instruction_current的12-14位
5 assign rs1 = Instruction_current[19:15]; //rs1为Instruction_current的15-19位
6 assign rs2 = Instruction_current[24:20]; //rs2为Instruction_current的20-24位
7 assign funct7 = Instruction_current[31:25]; //funct7为Instruction_current的高7位
8
9 //各类指令编码
10 assign ins = (opcode == 7'b0110011) ? `R
11 : (opcode == 7'b0100011) ? `S
12 : (opcode == 7'b1100011) ? `B
13 : (~opcode[6] && opcode[2]) ? `U
14 : (opcode == 7'b1101111) ? `J
15 : `I;
16
17 //各类操作数扩展
18 assign I_extend = {{21{Instruction_current[31]}},Instruction_current[30:20]};
19 assign S_extend = {{21{Instruction_current[31]}},Instruction_current[30:25],Instruction_current[11:7]};
20 assign B_extend = {{20{Instruction_current[31]}},Instruction_current[7],Instruction_current[30:25],Instruction_current[11:8],1'b0};
21 assign U_extend = {Instruction_current[31:12],12'b0};
22 assign J_extend = {{12{Instruction_current[31]}},Instruction_current[19:12],Instruction_current[20],Instruction_current[30:21],1'b0};

```

图 6: 操作码的划分及立即数的扩展

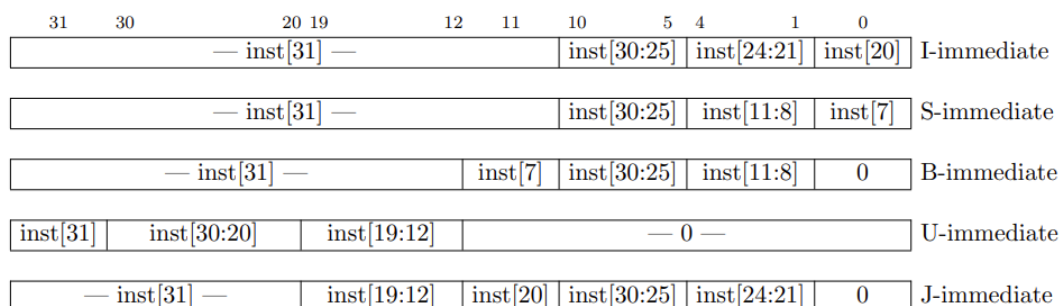


Figure 2.4: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses inst[31].

图 7: 立即数的扩展要求

由于指令的含义和数量与 MIPS 不同,因此我们需要重新确定各条指令是否需要 ALU 运算/Shift 运算以及需要具体用到哪个 ALU/Shift 操作:

```

1 //ALU操作
2 assign ALUOp = ({3{(ins == `I || ins == `R) && funct3 == 3'b111}} & `AND) | //AND/ANDI操作
3               ({3{(ins == `I || ins == `R) && funct3 == 3'b110}} & `OR) | //OR/ORI操作
4               ({3{(ins == `R && funct3 == 3'b0 && funct7[5] == 1'b0) || //ADD操作
5                 (ins == `I && funct3 == 3'b0) || //ADDI/LB/JALR操作
6                 (ins == `S) || (opcode == 7'b0000011) || //Store/Load操作
7                 (ins == `U && opcode[6:4] == 3'b001) || (ins == `J)}} & `ADD) | //AUIPC/JAL操作
8               ({3{(ins == `R && funct3 == 3'b0 && funct7[5]) || //SUB操作
9                 (ins == `B && funct3[2:1] == 2'b0)}} & `SUB) | //BEQ/BNE操作
10              ({3{(ins == `R && funct3 == 3'b010) || (ins == `I && funct3 == 3'b010 && opcode[4]) || //SLT/SLTI操作
11                (ins == `B && funct3[2:1] == 2'b10)}} & `SLT) | //BLT/BGE操作
12              ({3{(ins == `R && funct3 == 3'b100) || (ins == `I && funct3 == 3'b100 && opcode[4])}} & `XOR) | //XOR/XORI操作
13              ({3{(ins == `R && funct3 == 3'b011) || (ins == `I && funct3 == 3'b011 && opcode[4]) || //SLTU/SLTIU操作
14                (ins == `B && funct3[2:1] == 2'b11)}} & `SLTU); //BLTU/BGEU操作
15
16
17 assign ALU_A = (ins == `J || (ins == `U && opcode[5] == 1'b0)) ? PC_current
18               : RF_rdata1; //若为JAL/AUIPC操作, 则输入为PC_current, 否则正常输入
19 assign ALU_B = (ins == `R || ins == `B) ? RF_rdata2 //R-Type/B-Type操作
20               : (ins == `I) ? I_extend //I-Type操作
21               : (ins == `S) ? S_extend //S-Type操作
22               : (ins == `U && opcode[5] == 1'b0) ? U_extend //AUIPC操作
23               : (ins == `J) ? J_extend //J-Type操作
24               : 32'b0; //其他操作

```

图 8: ALU 操作部分代码

```

1 //Shifter操作
2 assign Shiftop = ((ins == `R || ins == `I) && funct3 == 3'b001 && opcode[4]) ? 2'b0 //SLL/SLLI操作
3               : ((ins == `R || ins == `I) && funct3 == 3'b010 && funct7[5] == 1'b0) ? 2'b10 //SRL/SRLI操作
4               : ((ins == `R || ins == `I) && funct3 == 3'b010 && funct7[5] == 1'b1) ? 2'b11 //SRA/SRAI操作
5               : 2'b01;
6 assign Shifter_B = (ins == `R) ? RF_rdata2[4:0] : I_extend[4:0]; //若为SLL/SRA/SRL, 则移位位数由rs2决定; 若为SLLI/SRAI/SRLI, 则由imm低五位决定

```

图 9: Shifter 操作部分代码

需要注意的是, 在 MIPS 指令集中的移位操作, 移位器的两个输入分别是 RF_data2 和 Shifter_B, 但是在 RV32I 中第一个输入应该是 RF_data1, 这是因为 RV32I 的移位操作是将 rs1 寄存器中的内容左移或右移 rs2 寄存器中的低五位或者立即数的低五位, 然后再存到 rd 寄存器中的。

```

1 shifter shifter_module(
2     .A(RF_rdata1),
3     .B(Shifter_B),
4     .Shiftop(Shiftop),
5     .Result(Shift_result)
6 );

```

图 10: Shifter 接口

除此之外,我们还需要修改跳转类指令,其中只需要修改 Jump 类型的两个指令即可,Branch 类型的指令无需修改,只需要进行变量名称的替换即可。

对于 JAL 和 JALR 两条指令,在指令集将一直的介绍如下图:

Unconditional Jumps

The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the pc to form the jump target address. Jumps can therefore target a ± 1 MiB range. JAL stores the address of the instruction following the jump (pc+4) into register rd. The standard software calling convention uses x1 as the return address register and x5 as an alternate link register.

Plain unconditional jumps (assembler pseudo-op J) are encoded as a JAL with rd=x0.

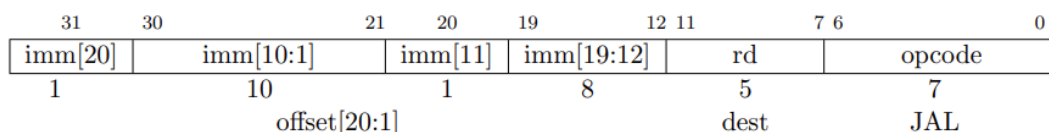


图 11: JAL 指令

The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the 12-bit signed I-immediate to the register rs1, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (pc+4) is written to register rd. Register x0 can be used as the destination if the result is not required.

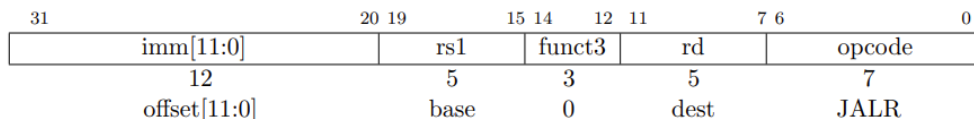


图 12: JALR 指令

这里需要注意的一点是, JALR 指令要求将跳转地址的最后一位变为 0, 因此我这里首先求出 RF_data1 和 I_extend 的和, 然后再将其最低位置为 0:

```

1 //Jump操作
2 assign Jump = (ins == `J || (ins == `I && opcode == 7'b1100111)) ? 1 : 0; //J-Type操作/I-Type JALR操作
3 assign JALR_addr = RF_data1+I_extend;
4 assign Jump_addr = (ins == `I && opcode[6]) ? {JALR_addr[31:1],1'b0} //若为I-Type JALR操作, 则JALR操作跳转地址为rs1+imm的值
5 : (PC_current + J_extend); //若为J-Type JAL操作, 则跳转地址为PC_current+imm的值

```

图 13: Jump 类指令部分代码

剩下的内存读写操作,由于 RV32I 指令数量比 MIPS 少了几条,相同的指令没有变化,因此只需要将少的那几条指令删掉即可。

值得注意的是, RV32I 新增的 U-Type 指令中除了 MIPS 的 LUI 指令之外,还增加了 AUIPC 指令,其含义是将当前 PC 的值加上左移 12 位的立即数的和存放到 rd 寄存器中,与 LUI 类似,只是多了加上 PC 的值这一内容。

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)

• 支持 RV32I 的功能型处理器设计。

本实验的内容较少,只需要仿照实验二的步骤,重新编写 RV32I 的指令译码表,然后将实验三的 custom_cpu 代码进行复用并修改即可。

不过在实验过程中还是遇到了一些 bug,下面将具体说明。

主要的问题是没有仔细看 RISC-V 指令集讲义而产生的错误

对于移位器的部分,我发现了需要将 Shifter_A 的接口改为 RF_data1,但是没有发现 Shifter_B 的值应该是 rs2 存储的值的第五位,而是理所当然的将五位地址号 rs2 作为偏移量。因为在 MIPS 指令集的移位器操作数 B 是操作码中的 sa 字段,其位置与 RV32I 的 rs2 字段相同。然后导致了写入 rd 寄存器的内容错误,从而导致了后面读写操作的错误,由于测试过程是具体的例子,因此只在具体的操作才会比较自己的结果和参考结果,这样就导致了我以为是 S-Type 内存写操作出了问题,但是我在检查之后发现 S-Type 类型操作没有问题,然后经过仔细查看波形以及对应的结果,终于发现了是 SLL 操作出了问题:

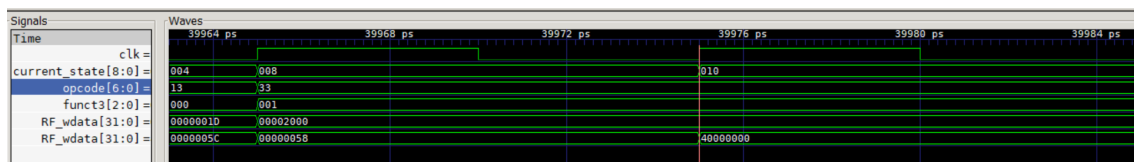


图 14: SLL 操作波形

从图中可以看到,当 opcode 为 7'b0110011, funct3 为 3'b001 时,对应的指令为 SLL,当 current_state 为 3'h010 时,即处于 EX 阶段时,写入 rd 寄存器的数据应该是 8'h40000000,但我的结果却是 8'h00002000,在定位到是 SLL 指令错误之后,我才去仔细阅读了对应的 SLL 操作的部分,发现了应当是偏移 rs2 寄存器中的值的后五位。

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
0000000	src2	src1	ADD/SLT/SLTU	dest	OP	
0000000	src2	src1	AND/OR/XOR	dest	OP	
0000000	src2	src1	SLL/SRL	dest	OP	
0100000	src2	src1	SUB/SRA	dest	OP	

ADD and SUB perform addition and subtraction respectively. Overflows are ignored and the low XLEN bits of results are written to the destination. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to rd if rs1 < rs2, 0 otherwise. Note, SLTU rd, x0, rs2 sets rd to 1 if rs2 is not equal to zero, otherwise sets rd to zero (assembler pseudo-op SNEZ rd, rs). AND, OR, and XOR perform bitwise logical operations.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.

图 15: R-Type 指令

三、 实验所耗时间

在课后,你花费了大约 8 小时完成此次实验。

四、实验思考与心得

• RISC-V/MIPS 指令集性能分析对比。

对于两种指令集的性能分析对比,我们通过性能计数器的结果进行比较:

MIPS	15pz	bf	dinic	fib	md5	qsort	queen	sieve	ssort
时钟周期数	527325394	46342677	1715247	179418637	404451	712898	6827222	1191704	52490821
指令数	5287711	559049	19326	2525722	5227	8339	80856	16478	728289
RISCV	15pz	bf	dinic	fib	md5	qsort	queen	sieve	ssort
时钟周期数	522802366	38811687	1479194	181088157	384627	786885	6885618	744437	44725243
指令数	5224458	452831	16668	2549502	4892	9457	81467	10172	619022
时钟周期数提升	0.86%	16.25%	13.76%	-0.93%	4.90%	-10.38%	-0.86%	37.53%	14.79%
指令数提升	1.20%	19.00%	13.75%	-0.94%	6.41%	-13.41%	-0.76%	38.27%	15.00%
平均时钟周期数提升	8.44%								
平均指令数提升	8.72%								

图 16: 两种指令集性能计数器比较

从图中可以看到,对于这九个 microbench,有六个测试 RISC-V 指令集比 MIPS 指令集效率更高,而弱于 MIPS 指令集的只有三个测试,其中 Fibonacci 和 Queen 这两个测试仅仅相差不到 1%,只有 qsort 测试 RISC-V 的效率明显低于 MIPS 指令集,然后从平均时钟周期和平均指令数的提升来看,达到了 8.5% 左右,性能提升还是较为明显的。

对于其中的原因,可能是由于 RISC-V 的指令更加规整,且更加简单,所以相较于 MIPS 更容易进行译码。RISC-V 支持多种立即数格式,使得不同类型的操作能够更高效地利用立即数,减少了额外的指令开销。此外,在修改指令的过程中,我发现 RISC-V 的加载和存储指令设计更加简单和高效,减少了地址计算和寄存器访问的开销。

而对于极少数的操作效率低于 MIPS,可能是某些特定操作在 MIPS 中有更优化的指令实现,而在 RISC-V 中可能需要多个指令组合才能完成,从而导致效率略低。

1 Instruction set

One of the most obvious differences between RISC-V and MIPS is the size and complexity of their instruction sets. RISC-V has a modular and scalable instruction set, which means that it consists of a small and simple core set of instructions (called the base ISA) and several optional extensions that add more functionality and features (such as floating-point, vector, or cryptography operations). MIPS, on the other hand, has a larger and more fixed instruction set, which includes some complex and irregular instructions that are not present in RISC-V (such as load-linked/store-conditional, branch-likely, or multiply-accumulate). While RISC-V aims to be minimal and flexible, MIPS aims to be comprehensive and stable.

图 17: 两种指令集指令格式的区别

此外, RISC-V 指令集的每个字段表示的内容固定, 有利于简化解码器的设计。例如, 立即数的高位和低位可能需要放在不同的位置, 以便与其他字段(如操作码、寄存器编号等)分开, 从而简化硬件电路。这也是为什么不同类型指令的立即数要分块放置的原因。