中国科学院大学计算机组成原理实验课

实 验 报 告

学号：2019K8009915025　　姓名：马月骁　　专业：计算机科学与技术

实验序号：prj4　　实验名称：RISC-V 指令集处理器

一、 逻辑电路结构与 FPGA 运行结果的截图及说明

RISC-V 指令集处理器实验中共完成四个硬件文件（reg_file.v，alu.v，shifter.v 以及 custom_cpu.v）和两个软件文件的编写（printf.c 以及 bench.c）的设计与编写。

其中 reg_file.v，alu.v，shifter.v，printf.c 以及 bench.c 文件直接复用 prj3（内存及外设通路设计与处理器性能评估）中的对应文件，本次实验报告不进行重复说明。custom_cpu.v 文件中数据通路与 prj3 总体上一致，总体逻辑电路结构可参照 prj2（单周期处理器设计）以及 prj3 中的逻辑电路结构图。本次实验中 custom_cpu.v 文件中 RTL 代码相较 prj3 主要更改了指令译码部分，部分控制信号以及少量数据通路，其余部分（主要为状态机以及性能计数器相关代码）与 prj3 相应部分保持一致，本次实验报告不进行重复重复说明。

以下说明本次实验中相较 prj3 中有所改动的 RTL 代码（custom_cpu.v）：

（一）指令译码

1. 信号定义

```
66    wire [6:0]   opcode;
67    wire [4:0]   rs1;
68    wire [4:0]   rs2;
69    wire [4:0]   rd;
70    wire [6:0]   funct7;
71    wire [2:0]   funct3;
72    wire [11:0]  imm_Itype;
73    wire [4:0]   shamt;
74    wire [11:0]  imm_Stype;
75    wire [12:1]  imm_Btype;
76    wire [31:12] imm_Utype;
77    wire [20:1]  imm_Jtype;
```

以上为本次实验中所实现 RISC-V 指令集中 37 条基本指令各字段对应信号。

```
79    wire R_type;
80    wire I_type;
81    wire I_type_calc;
82    wire I_type_load;
83    wire I_type_jalr;
84    wire J_type;
85    wire S_type;
86    wire B_type;
87    wire U_type;
```

以上为对本次实验中所实现 RISC-V 指令集中 37 条基本指令进行分类的信号（便于之后控制信号赋值），包括 R-type，I-type，J-type，S-type，B-type 以及 U-type。其中 I-type 根据指令功能细分为 calculate、load、jalr 三类。

```
125    wire [31:0] sign_extension;
126    wire [31:0] zero_extension;
127    wire [31:0] imm_extension;
128    wire [31:0] Btype_extension;
129    wire [31:0] Utype_extension;
130    wire [31:0] Jtype_extension;
```

以上为对本次实验中所实现 RISC-V 指令集中 37 条基本指令中指令码汇中立即数字段对应的完整立即数信号。

2. 信号赋值

```
254    assign opcode    = Instruction_reg[6:0];
255    assign rs1       = Instruction_reg[19:15];
256    assign rs2       = Instruction_reg[24:20];
257    assign rd        = Instruction_reg[11:7];
258    assign funct3    = Instruction_reg[14:12];
259    assign funct7    = Instruction_reg[31:25];
260    assign shamt     = rs2;
261    assign imm_Itype = Instruction_reg[31:20];
262    assign imm_Stype = {Instruction_reg[31:25], Instruction_reg[11:7]};
263    assign imm_Btype = {Instruction_reg[31], Instruction_reg[7], Instruction_reg[30:25], Instruction_reg[11:8]};
264    assign imm_Utype = Instruction_reg[31:12];
265    assign imm_Jtype = {Instruction_reg[31], Instruction_reg[19:12], Instruction_reg[20], Instruction_reg[30:21]};
266
267    assign R_type = (opcode == 7'b0110011);
268    assign I_type = I_type_calc | I_type_load | I_type_jalr;
269    assign I_type_calc = (opcode == 7'b0010011);
270    assign I_type_load = (opcode == 7'b0000011);
271    assign I_type_jalr = (opcode == 7'b1100111);
272    assign S_type = (opcode == 7'b0100011);
273    assign B_type = (opcode == 7'b1100011);
274    assign U_type = (opcode == 7'b0110111) | (opcode == 7'b0010111);
275    assign J_type = (opcode == 7'b1101111);
276
277    assign sign_extension = ({32{I_type}} & {{20{imm_Itype[11]}}, imm_Itype}) |
278                            ({32{S_type}} & {{20{imm_Stype[11]}}, imm_Stype});
279    assign zero_extension = ({32{I_type}} & {{20{1'b0}}, imm_Itype}) |
280                            ({32{S_type}} & {{20{1'b0}}, imm_Stype});
281    assign imm_extension = ({32{I_type_calc & (funct3 == 3'b011)}} & zero_extension) |
282                           ({32{~(I_type_calc & (funct3 == 3'b011))}} & sign_extension);
283    assign Btype_extension = {{19{imm_Btype[12]}}, imm_Btype, 1'b0};
284    assign Utype_extension = {imm_Utype, 12'b0};
285    assign Jtype_extension = {{11{imm_Jtype[20]}}, imm_Jtype, 1'b0};
```

上述信号根据译码表(依据 RSIC-V 指令手册制作)完成对各个信号的赋值，

实现对指令的译码。

（二）各部件控制信号以及部分数据通路

1. ALU

```
287     //ALUop:
288     //AND = 3'b000
289     //OR = 3'b001
290     //ADD = 3'b010   : R_type, I_type_calc, I_type_load/jalr, S_type
291     //SLTU = 3'b011
292     //XOR = 3'b100
293     //NOR = 3'b101
294     //SUB = 3'b110   : R_type, I_type_calc, B_type
295     //SLT = 3'b111
296     assign ALUop = ({3{R_type}} & (({3{~funct7[5] & (~|funct3)}} & 3'b010) |
297                                    ({3{funct7[5] & (~|funct3)}} & 3'b110) |
298                                    ({3{funct3 == 3'b010}} & 3'b111) |
299                                    ({3{(funct3 == 3'b011) | (funct3 == 3'b100)}} & funct3) |
300                                    ({3{(&funct3) | (funct3 == 3'b110)}} & ~funct3)
301                                   )
302                    ) |
303                    ({3{I_type_calc}} & (({3{~|funct3}} & 3'b010) |
304                                    ({3{funct3 == 3'b010}} & 3'b111) |
305                                    ({3{(funct3 == 3'b011) | (funct3 == 3'b100)}} & funct3) |
306                                    ({3{(&funct3) | (funct3 == 3'b001)}} & ~funct3)
307                                   )
308                    ) |
309                    ({3{I_type_load | S_type | I_type_jalr}} & 3'b010) |
310                    ({3{B_type}} & 3'b110) ;
311     assign ALU_A = RF_rdata1;
312     assign ALU_B = ({32{R_type | B_type}} & RF_rdata2) |
313                    ({32{I_type | S_type}} & imm_extension);
```

以上为 ALU 控制信号（ALUop）以及相关数据通路（ALU_A，ALU_B）的赋值与选择。

依据指令译码表，使用 ALU 的指令类型为 R-type，I-type，S-type 以及 B-type。根据各指令所需执行的操作类型选择对应的 ALUop。ALU_A 始终为 RF_rdata1；当为 R-type 或 B-type 类型时 ALU_B 选通 RF_rdata2，当为 I-type 或 S-type 类型时 ALU_B 选通立即数。

2. Shifter

```
314     //Shifterop:
315     //L = 2'b00
316     //AR = 2'b11
317     //LR = 2'b10
318     assign Shifterop = {funct3[2], funct7[5]};
319     assign Shifter_A = RF_rdata1;
320     assign Shifter_B = ({32{R_type}} & {27'b0, RF_rdata2[4:0]}) |
321                        ({32{I_type}} & {27'b0, shamt});
```

以上为 Shifter 控制信号（Shifterop）以及相关数据通路（Shifter_A，Shifter_B）的赋值与选择。

依据指令译码表，使用 Shifter 的指令类型为 R-type 与 I-type 中 calculate 类型。根据各指令所需执行的操作类型选择对应的 Shifterop。Shifter_A 始终为 RF_rdata1；当为 R-type 类型时 Shifter_B 选通 RF_rdata2，当为 I-type 中 calculate 类型时 Shifter_B 选通立即数。

3. Register File

```
323     assign RF_raddr1 = rs1;
324     assign RF_raddr2 = rs2;
325     assign RF_waddr = rd;
326    //R-type, I-type-calc: ALU_result | Shifter_result
327    //I-type-load: Read_data;
328    //I-type-jalr: pc4
329    //U-type: LUI: Utype_extension, AUIPC: AUIPC_result
330    //J-type: pc4
331     assign RF_wdata = ({32{R_type | I_type_calc}} & (({32{(funct3 == 3'b001) | (funct3 == 3'b101)}} & Shifter_result) |
332                                                      ({32{~((funct3 == 3'b001) | (funct3 == 3'b101))}} & ALU_result)
333                                                     )
334                     ) |
335                     ({32{I_type_load}} & load_data) |
336                     ({32{I_type_jalr | J_type}} & PC_reg4) |
337                     ({32{U_type}} & (({32{opcode == 7'b0110111}} & Utype_extension) |
338                                      ({32{opcode == 7'b0010111}} & AUIPC_result)
339                                     )
340                     );
341    //R-type, I-type, U-type, J-type
342     assign RF_wen = current_state[5];
```

以上为 Register File 的控制信号以及数据通路的赋值与选择。

其中 RF_wen 与 prj3 保持一致。RF_raddr1，RF_raddr2，RF_waddr 根据指令译码表始终分别为 rs1，rs2，rd。根据指令译码表，当指令为 R-type 或 I-type 中 calculate 类型时，RF_wdata 选通 ALU_result 或 Shifter_result；当指令为 I-type 中 load 类型时，RF_wdata 选通 load_data，当指令为 I-type 中 jalr 或 J-type 类型时，RF_wdata 选通 PC_reg4；当指令为 U-type 类型时，RF_wdata 根据具体指令选通对应结果。

4. Memory

除由于 RISC-V 指令集中没有 lwl，lwr 指令，本次实验 custom_cpu.v 文件中没有与之相关代码外，其余与 Memory 相关控制信号与数据通路与 prj3 中对应部分完全一致。详见 prj3

5. PC

PC 相关寄存器描述位于状态机描述部分，与 prj3 保持一致。

```
344     //BEQ(==): funct3 = 000
345     //BNE(!=): funct3 = 001
346     //BLT(<) : funct3 = 100
347     //BGE(>=): funct3 = 101
348     //BLTU   : funct3 = 110
349     //BGEU   : funct3 = 111
350     assign branch = B_type & (((~|funct3) & ALU_zero) |
351                               ((funct3 == 3'b001) & ~ALU_zero) |
352                               ((funct3 == 3'b100) & (ALU_Overflow ^ ALU_result[31])) |
353                               ((funct3 == 3'b101) & ~(ALU_Overflow ^ ALU_result[31])) |
354                               ((funct3 == 3'b110) & ALU_CarryOut) |
355                               ((&funct3) & ~ALU_CarryOut)
356                              );
357     assign branch_result = PC_reg + Btype_extension;
358     assign jump = J_type | I_type_jalr;
359     assign jump_result = ({32{J_type}} & (PC_reg + Jtype_extension)) |
360                          ({32{I_type_jalr}} & {ALU_result[31:1], 1'b0});
```

以上为条件跳转指令（B-type）与无条件跳转指令（I-type-jalr 与 J-type）部分控制信号与数据通路的赋值与选择。控制信号赋值与数据通路选择与 prj3 基本保持一致，仅仅根据译码方式转变进行微调。

（三）FPGA 运行结果分析

本次实验设计的处理器基于 RISC-V 指令集，prj3 中设计的处理器基于 MIPS 指令集。以下根据 FPGA 上 9 个 testbench 运行结果中性能计数器统计结果对 RISC-V 指令集与 MIPS 指令集性能进行对比分析。

基于 MIPS 指令集设计的处理器与基于 RISC-V 指令集设计的处理器在不同 testbench 上的表现各有优劣。

Testbench：40_ssort：

```
[ssort] Suffix sort: * Passed.
The number of cycle counts: 52755151
The number of reading memory counts: 54122
The number of instruction counts: 728408
The number of branch operation counts: 106130
The number of successful branch operation counts: 61552
The number of jump operation counts: 707
```

MIPS 指令集处理器

```
[ssort] Suffix sort: * Passed.
The number of cycle counts: 49172528
The number of reading memory counts: 52368
The number of instruction counts: 619137
The number of branch operation counts: 132631
The number of successful branch operation counts: 66423
The number of jump operation counts: 883
```

RISC-V 指令集处理器

由上图可见，在 testbench 40_ssort 中，RISC-V 处理器运行时所用的周期总数 49172528 与指令总数 619137 均小于 MIPS 处理器运行时所用的周期总数 52755151 和指令总数 728408，但 RISC-V 处理器运行时的 CPI=79.42 大于 MIPS 处理器运行时的 CPI=72.43；RISC-V 处理器运行时内存访问次数 52368 小于 MIPS 处理器运行时内存访问次数 54122。

Testbench：37_qsort：

```
[qsort] Quick sort: * Passed.
The number of cycle counts: 720524
The number of reading memory counts: 6902
The number of instruction counts: 8458
The number of branch operation counts: 1713
The number of successful branch operation counts: 1280
The number of jump operation counts: 149
```

MIPS 指令集处理器

```
[qsort] Quick sort: * Passed.
The number of cycle counts: 813526
The number of reading memory counts: 6902
The number of instruction counts: 9572
The number of branch operation counts: 1678
The number of successful branch operation counts: 1245
The number of jump operation counts: 184
```

RISC-V 指令集处理器

由上图可见，在 testbench 37_qsort 中，RISC-V 处理器运行时所用的周期总数 813526 与指令总数 9572 均大于 MIPS 处理器运行时所用的周期总数 720524 和指令总数 8458，但 RISC-V 处理器运行时的 CPI=84.99 略小于 MIPS 处理器运行时的 CPI=85.19；RISC-V 处理器运行时内存访问次数 6902 等于 MIPS 处理器运行时内存访问次数 6902。

Testbench：36_md5：

```
[md5] MD5 digest: * Passed.
The number of cycle counts: 420737
The number of reading memory counts: 1442
The number of instruction counts: 5346
The number of branch operation counts: 588
The number of successful branch operation counts: 263
The number of jump operation counts: 18
```

MIPS 指令集处理器

```
[md5] MD5 digest: * Passed.
The number of cycle counts: 403005
The number of reading memory counts: 1572
The number of instruction counts: 5007
The number of branch operation counts: 394
The number of successful branch operation counts: 291
The number of jump operation counts: 82
```

RISC-V 指令集处理器

由上图可见，在 testbench 36_md5 中，RISC-V 处理器运行时所用的周期总数 403005 与指令总数 5007 均小于 MIPS 处理器运行时所用的周期总数 420737

和指令总数 5346，但 RISC-V 处理器运行时的 CPI=80.49 大于 MIPS 处理器运行时的 CPI=78.70；RISC-V 处理器运行时内存访问次数 1572 大于 MIPS 处理器运行时内存访问次数 1442。

Testbench：34_dinic：

```
[dinic] Dinic's maxflow algorithm: * Passed.
The number of cycle counts: 1776165
The number of reading memory counts: 16922
The number of instruction counts: 19445
The number of branch operation counts: 2169
The number of successful branch operation counts: 1166
The number of jump operation counts: 38
```

MIPS 指令集处理器

```
[dinic] Dinic's maxflow algorithm: * Passed.
The number of cycle counts: 1526821
The number of reading memory counts: 14274
The number of instruction counts: 16783
The number of branch operation counts: 2137
The number of successful branch operation counts: 950
The number of jump operation counts: 341
```

RISC-V 指令集处理器

由上图可见，在 testbench34_dinic 中，RISC-V 处理器运行时所用的周期总数 1526821 与指令总数 16783 均小于 MIPS 处理器运行时所用的周期总数 1776165 和指令总数 19445，且 RISC-V 处理器运行时的 CPI=90.97 略小于 MIPS 处理器运行时的CPI=91.34;RISC-V处理器运行时内存访问次数14274小于MIPS处理器运行时内存访问次数 16922。

综上 RISC-V 指令集处理器与 MIPS 指令集处理器在不同 testbench 上运行表现各有优劣。

两种处理器各自在 FPGA 版上运行 testbench 性能计数器统计结果如下：

testbench	周期数	指令总数	内存访问次数	条件跳转指令数	条件跳转指令跳转次数	非条件跳转指令数	CPI
32_15pz	535642110	5287830	8701718	647736	631463	411	101.30
33_bf	47168963	559168	212996	106688	51760	32248	84.36
34_dinic	1776165	19445	16922	2169	1166	38	91.34
35_fib	179511183	2525841	12826	398164	382537	5234	71.07
36_md5	420737	5346	1442	588	263	18	78.70
37_qsort	720524	8458	6902	1713	1280	149	85.19
38_queen	6916348	80975	78342	6722	2115	4132	85.41
39_sieve	1206034	16597	1292	1847	1667	18	72.67
40_ssort	52755151	728408	54122	106130	61552	707	72.43

MIPS 指令集处理器

testbench	周期数	指令总数	内存访问次数	条件跳转指令数	条件跳转指令跳转次数	非条件跳转指令数	CPI
32_15pz	570658298	5224573	8695146	622796	624442	1285	109.23
33_bf	42701054	452946	213004	91039	36085	47920	94.27
34_dinic	1526821	16783	14274	2137	950	341	90.97
35_fib	198544257	2549617	12654	577540	465961	12136	77.87
36_md5	403005	5007	1572	394	291	82	80.49
37_qsort	813526	9572	6902	1678	1245	184	84.99
38_queen	7077208	81582	78342	6078	1471	4132	86.75
39_sieve	829272	10287	1284	2496	1355	60	80.61
40_ssort	49172528	619137	52368	132631	66423	883	79.42

RISC-V 指令集处理器

	MIPS	RISC-V
平均周期数	91790801.67	96858441
平均指令数	1025785.333	996611.5556
平均CPI	89.4834413	97.18775631
平均内存访问次数	1009618	1008394
平均条件跳转指令成功率	0.891524875	0.833958918
平均跳转指令总数	146079.1111	167090.2222

MIPS 处理器与 RISC-V 处理器性能计数器数据处理表

由上表可知，总体上，RISC-V 处理器的平均周期数与平均 CPI 大于 MIPS 处理器，而平均指令数小于 MIPS 处理器；RISC-V 处理器的平均内存访问次数小于 MIPS 处理器；RISC-V 处理器的跳转指令数多于 MIPS 处理器，而条件跳转指令成功跳转比例低于 MIPS 处理器。

二、 实验过程中遇到的问题、对问题的思考过程及解决方法

（一）代码逻辑错误

1. custom_cpu.v

（1）缺少信号声明

问题（原因）：编写代码时遗漏信号申明，导致运行时信号值为 z，无法正常运行。

解决方案：根据波形图，依据逻辑回溯错误信号，定位出错时间点和出错信号，再加以修改。

（2）信号赋值出错

问题：编写代码时由于笔误设置信号宽度出错、选取信号位数出错、按位赋值时选取错误信号位数。

解决方案：根据波形图，定位出错信号位置，逐行排查逻辑，加以修改；编写代码时小心谨慎，防止书写错误。

（3）译码出错

问题：编写译码表时，对指令手册内容理解出现偏差，导致译码表编写出错，进而导致译码错误。

解决方案：根据波形图，定位出错信号位置，确定出错指令，比对译码表，若与译码表一致，查询指令手册中对应指令详细说明。

（二）调试过程难点

1. 波形查看

问题（原因）：DUMP_TIME 区间设置不当（最大值超出已经仿真最大时间），导致波形文件无法打开。

解决方案：根据仿真报错中已仿真的最大时间设置 DUMP_TIME 上界。

2. 性能计数器结果

问题：仿真阶段性能计数器输出结果始终为 0，上版结果正常。

该问题 prj3 中已经出现，未能解决，疑似平台问题。

3. 调试时间成本时间过高

问题 1：平台不稳定，导致仿真出错或 pipeline 处于 stuck 状态

解决方案：向助教反映，等待平台恢复。

问题 2：提交人数较多，pipeline 处于 pending 状态或时仿真时间过长。

解决方案：等待。若因超时出错，及时 retry。

问题 3：在全自动开发流程中，当仿真出错时，需要修改.gitlab-ci.yml 脚本（增加 DUMP_TIME），重新 push，重新仿真才能看到波形。同时由于 DUMP_TIME 要求于仿真运行时间保持一致（少于仿真运行时间看不到错误点，超出仿真运行时间波形文件无法生成），在一定情况下增加了仿真次数，使得 debug 周期变长。

解决方案：提交前检查代码，确认代码正确性，减少潜在 bug。

4. 操作不当导致无妨触发仿真

问题：查看波形后，在 flow 文件夹内误操作进行了 commit 和 push 操作，导致本地仓库与 github 公共仓库中 flow 文件 commit 号不一致，无法触发流程。

解决方案：通过 reset 命令撤销 commit，回退状态。

三、 对讲义中思考题（如有）的理解和回答

本次实验无思考题

四、 在课后，你花费了大约 10 小时完成此次实验。

平台上运行时间（等待结果时间）为 24+小时。

五、 对于此次实验的心得、感受和建议

（一）实验心得、感受

1. 代码编写

在已有代码的基础上进行修改可使代码编写速度大大加快。如本次实验中 custom_cpu.v 即是在 prj3 中 mips_cpu.v 的基础上修改所得。

在修改时需要检查所有需要修改的地方，防止遗漏。如变量名修改，若存在遗漏。会导致相关信号出错。

参考以往编写代码经验，如提前编写指令译码表等，可以缩短代码编写周期。

2. debug

开始实验项目前应保证头脑清醒，前一天休息不足容易导致编写代码时低级错误频发（如笔误写错信号位宽等）。清醒的头脑可以减少无谓的错误，进而可使代码编写事半功倍。

在 push 代码，trigger pipeline 前应当再检查一遍代码，减少潜在错误。看代码时间远远小于仿真时间，有利于减少 debug 时间。

感谢王嵩岳助教及时反馈我关于云平台的问题。

（二）实验建议

1. 仿真与上版

本次实验在平台上仿真时间远远大于上版运行时间，当进行实验第二、第三部分（外设控制器访问、性能计数器访问）的代码编写和修改时，依旧需要跑完仿真，时间成本高。同时由于上版时对仿真结果存在依赖，无法使上版与仿真同

时运行。

希望可以在代码已经保证可以正常仿真时，允许仿真与上版同步运行。

2. 仿真波形获得

本次实验中仿真波形主要使用于 debug 阶段。

在全自动开发流程中，当仿真出错时，需要修改.gitlab-ci.yml 脚本（增加 DUMP_TIME），重新 push，重新仿真才能看到波形。同时由于 DUMP_TIME 要求于仿真运行时间保持一致（少于仿真运行时间看不到错误点，超出仿真运行时间波形文件无法生成），在一定情况下增加了仿真次数，使得 debug 周期变长。

希望可以通过脚本文件，自动根据仿真时间确定合适 DUMP_TIME 并生成对应波形文件。