

中国科学院大学计算机组成原理实验课

实 验 报 告

学号： 2020K8009970001 姓名： 金扬 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

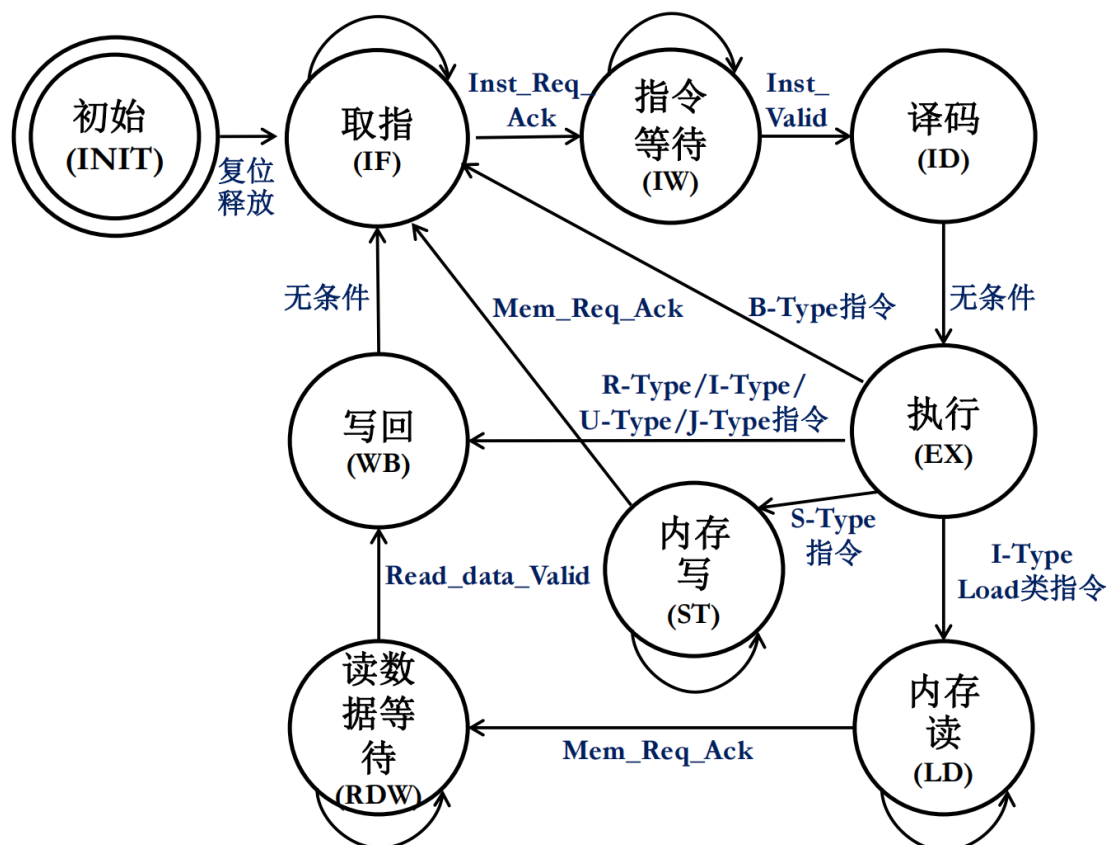
- 注 1：撰写此 Word 格式实验报告后以 PDF 格式保存在~/COD-Lab/reports 目录下。文件命名规则：prjN.pdf，其中“prj”和后缀名“pdf”为小写，“N”为 1 至 4 的阿拉伯数字。例如：prj1.pdf。PDF 文件大小应控制在 5MB 以内。此外，实验项目 5 包含多个选做内容，每个选做实验应提交各自的实验报告文件，文件命名规则：prj5-projectname.pdf，其中“-”为英文标点符号的短横线。文件命名举例：prj5-dma.pdf。具体要求详见实验项目 5 讲义。
- 注 2：使用 git add 及 git commit 命令将实验报告 PDF 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)

1. 基于 RSICV32 的多周期状态机

状态转移图如课程课件所示:



共有九个状态，采用独热码对这九个状态逐个进行编码：

```
localparam INIT =9'b000000001;
localparam IF  =9'b000000010;
localparam IW  =9'b000000100;
localparam ID  =9'b000001000;
localparam EX  =9'b000010000;
localparam ST  =9'b000100000;
localparam LD  =9'b001000000;
localparam RDW =9'b010000000;
localparam WB  =9'b100000000;

localparam isINIT =0;
localparam isIF   =1;
localparam isIW   =2;
```

```

localparam isID    =3;
localparam isEX    =4;
localparam isST    =5;
localparam isLD    =6;
localparam isRDW   =7;
localparam isWB    =8;

```

采用“三段式”状态机描述方法：

- a. “第一段”用 always 时序逻辑，描述状态寄存器的同步状态跳转。

```

always @ (posedge clk) begin
    if (rst) begin
        current_state <= INIT;
    end else begin
        current_state <= next_state;
    end
end
end

```

- b. “第二段”用 always 组合逻辑，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。

```

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: next_state <= EX;
        EX: begin
            if (R_Type | I_Type & ~LOAD | U_Type | J_Type) begin
                next_state <= WB;
            end else if (LOAD) begin
                next_state <= LD;
            end
        end
    end
end

```

```

        end else if (S_Type) begin
            next_state <= ST;
        end else if (B_Type) begin
            next_state <= IF;
        end else begin
            next_state <= INIT;
        end
    end
end
ST: begin
    if (Mem_Req_Ready) begin
        next_state <= IF;
    end else begin
        next_state <= ST;
    end
end
LD: begin
    if (Mem_Req_Ready) begin
        next_state <= RDW;
    end else begin
        next_state <= LD;
    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

```

- c. “第三段” 用 always 时序逻辑或 assign 组合逻辑，根据状态机当前状态，描述不同控制线路及一些寄存器的同步变化。

2. 基于真实内存的多周期访存逻辑（与 prj3 相同）

```
assign Inst_Req_Valid = current_state[isIF];  
assign Inst_Ready    = current_state[isIW] ||  
current_state[isINIT];  
assign MemWrite      = current_state[isST];  
assign MemRead       = current_state[isLD];  
assign Read_data_Ready = current_state[isRDW] ||  
current_state[isINIT];
```

如代码所示，在 IF 状态拉高 Inst_Req_Valid，在接收到 Inst_Req_Ready 有效时进入 IW 状态等待指令。在 IW 状态拉高 Inst_Ready，在接收到 Inst_Valid 有效时进入 ID 状态。对于 Store 指令，在 ST 状态拉高 MemWrite，在接收到 Mem_Req_Valid 有效时返回 IF 状态。对于 Load 指令，在 LD 状态拉高 MemRead，在接收到 Mem_Req_Valid 有效时进入 RDW 等待读数，当 Read_data_Ready 有效时返回 IF 状态。

3. PC 寄存器、IR 寄存器的更新

```
//PC
assign Branch_or_not = (Zero ^ funct3[2] ^ funct3[0]) & Branch;
always @(posedge clk) begin
    if (rst) begin
        PC_reg <= 32'd0;
    end /*else if (current_state[isIW] && Inst_Valid) begin
        PC_reg <= PC_reg + 4;
    end */else if (current_state[isEX]) begin
        if (JAL) begin
            PC_reg <= PC_reg + imm;
        end else if (JALR) begin
            PC_reg <= (RF_rdata1 + imm) & {~31'b0, 1'b0};
        end else if (Branch_or_not) begin
            PC_reg <= PC_reg + imm;
        end else begin
            PC_reg <= PC_reg + 4;
        end
    end
end
assign PC = PC_reg;

always @(posedge clk) begin
    if (current_state[isID]) begin
        PC_normal <= PC_reg;
    end
end

//IR
always @(posedge clk) begin
    if (current_state[isIW] && Inst_Valid) begin
        IR <= Instruction;
    end
end
```

大部分逻辑与基于 MIPS 指令集的实现类似，因此不再赘述。

需要注意的是，由于我实现的 RISC-V 处理器没有 MIPS 中分支延迟槽的设计，NOP 指令也有 EX 状态，因此选择在 EX 状态更新所有指令的 PC 值

4. 指令解码

```
//ID
assign {funct7, rs2, rs1, funct3, rd, opcode} = IR;
assign imm = {
    { //31:12
        U_Type | J_Type ?
        { //31:12
            { //31:20
                U_Type ? {IR[31],IR[30:20]} : {(12){IR[31]}}
            },
            IR[19:12] //19:12
        } : {(20){IR[31]}}
    },
    { //11:0
        U_Type ? 12'b0 :
        { //11:0
            { //11
                B_Type ? IR[7] : J_Type ? IR[20] : IR[31]
            },
            IR[30:25], //10:5
            { //4:1
                I_Type | J_Type ? IR[24:21] : IR[11:8]
            },
            { //0
                I_Type ? IR[20] : S_Type ? IR[7] : 1'b0
            }
        }
    }
};
```

依据字段将指令分割，并根据指令类型生成相应的立即数。

5. 控制信号生成

```
assign OP_IMM    = opcode[6:0] == 7'b0010011;
assign LUI       = opcode[6:0] == 7'b0110111;
assign AUIPC     = opcode[6:0] == 7'b0010111;
assign OP        = opcode[6:0] == 7'b0110011;
assign JAL       = opcode[6:0] == 7'b1101111;
assign JALR      = opcode[6:0] == 7'b1100111;
assign BRANCH    = opcode[6:0] == 7'b1100011;
assign LOAD      = opcode[6:0] == 7'b0000011;
assign STORE     = opcode[6:0] == 7'b0100011;

assign R_Type    = OP;
assign I_Type    = OP_IMM | JALR | LOAD;
assign S_Type    = STORE;
assign B_Type    = BRANCH;
assign U_Type    = LUI | AUIPC;
assign J_Type    = JAL;

assign Branch    = B_Type;
assign MemtoReg  = LOAD;
assign ALUEn     = (OP & ~funct7[0] | OP_IMM) & (funct3[1] | ~funct3[0]) | JALR | LOAD | S_Type | B_Type;
assign ShiftEn   = (OP & ~funct7[0] | OP_IMM) & (~funct3[1] & funct3[0]);
assign MULEn     = OP & funct7[0];
assign ALUSrc    = I_Type | S_Type;
assign ShiftSrc  = I_Type | S_Type;
assign RF_wen    = current_state[isWB] & (J_Type | I_Type | R_Type | U_Type);
```

先根据 opcode 识别当前指令，进而获得当前指令类型；再结合 funct3，funct7 以及当前状态信息，生成控制信号。

6. ALUOp 的生成

```
assign ALUOp = {(3){OP_IMM | OP}} & {  
    (funct3 == 3'b100) //xor  
    | (funct3 == 3'b010) //slt  
    | (funct3 == 3'b000 && funct7[5] && opcode[5]), //sub  
    ~funct3[2], //and, or, xor  
    funct3[1] & ~(funct3[0] & funct3[2]) //or, slt, sltu  
} | {(3){STORE | LOAD | JALR}} & 3'b010 //add 010  
    | {(3){BRANCH}} & {~funct3[1], 1'b1, funct3[2]};  
//sub 110 slt 111 sltu 011
```

根据当前指令类型，将 ALUOp 分为三类：

- OP_IMM 或 OP，此时对 ALUOp 根据 funct3, funct7, opcode 信息进行逐位编码。
- STORE 或 LOAD 或 JALR，此时需要对寄存器数和立即数做加法得到最终的地址，因此直接输入 010 (add)。
- BRANCH，此时需要判断两个寄存器数的相对大小，并根据结果选择分支，因此需要根据指令要求做减法、比较运算。

7. 访存与写回

```
//Write  
assign Address = Result & ~32'b11;  
assign Write_data = RF_rdata2 << {Result[1:0], 3'b0};  
assign Write_strb = { ( Result[1] | funct3[1]) & ( Result[0] | funct3[0] | funct3[1]),  
    ( Result[1] | funct3[1]) & (~Result[0] | funct3[0] | funct3[1]),  
    (~Result[1] | funct3[1]) & ( Result[0] | funct3[0] | funct3[1]),  
    (~Result[1] | funct3[1]) & (~Result[0] | funct3[0] | funct3[1])  
};  
  
//Read  
always @(posedge clk) begin  
    if (current_state[isRDW] && Read_data_Valid) begin  
        MDR <= Read_data;  
    end  
end  
  
//Write Back  
assign Read_data_shifted = MDR >> {Result[1:0], 3'b0};  
assign Read_data_masked = Read_data_shifted & {{(16){funct3[1]}}, {(8){funct3[0] | funct3[1]}}, {(8){1'b1}}}  
    | {(32){~funct3[2] & Read_data_sign_bit}} & ~{(16){funct3[1]}}, {(8){funct3[0] | funct3[1]}}, {(8){1'b1}}};  
assign Read_data_sign_bit = Read_data_shifted[funct3[1:0]==2'b01 ? 15 : 7];  
assign RF_wdata = {(32){LUI}} & imm  
    | {(32){AUIPC}} & PC_normal + imm  
    | {(32){JAL | JALR}} & PC_normal + 4  
    | {(32){LOAD}} & Read_data_masked  
    | {(32){OP | OP_IMM}} & Result;
```

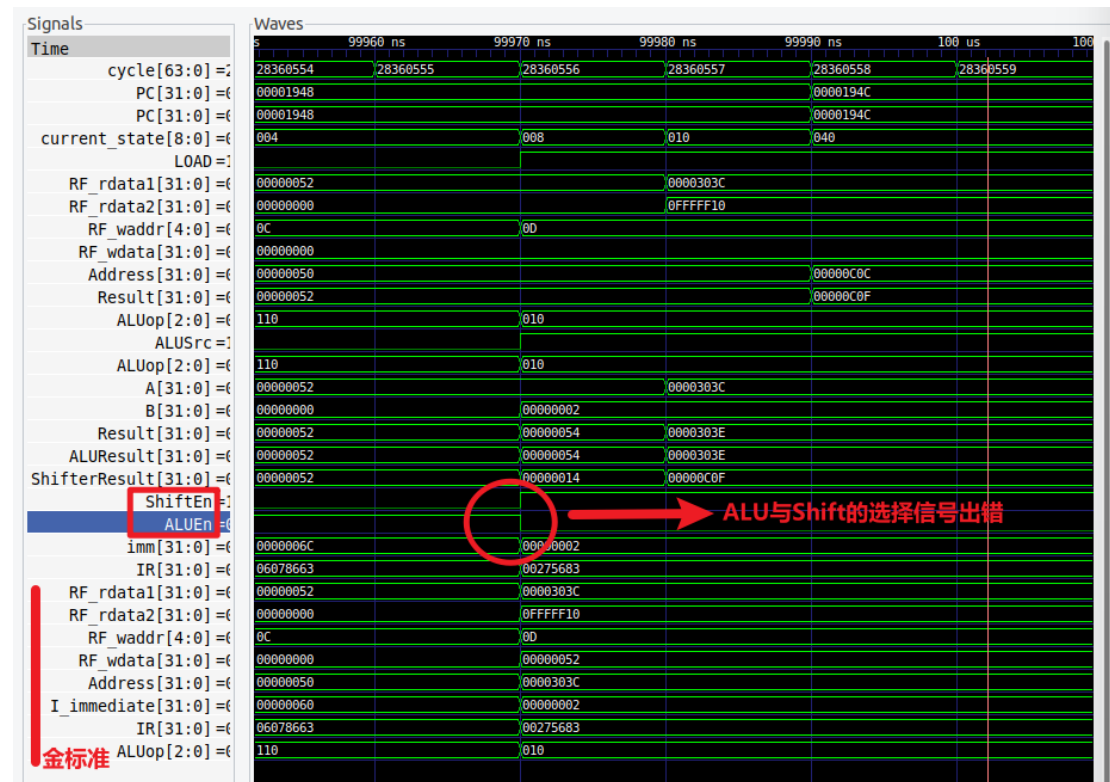
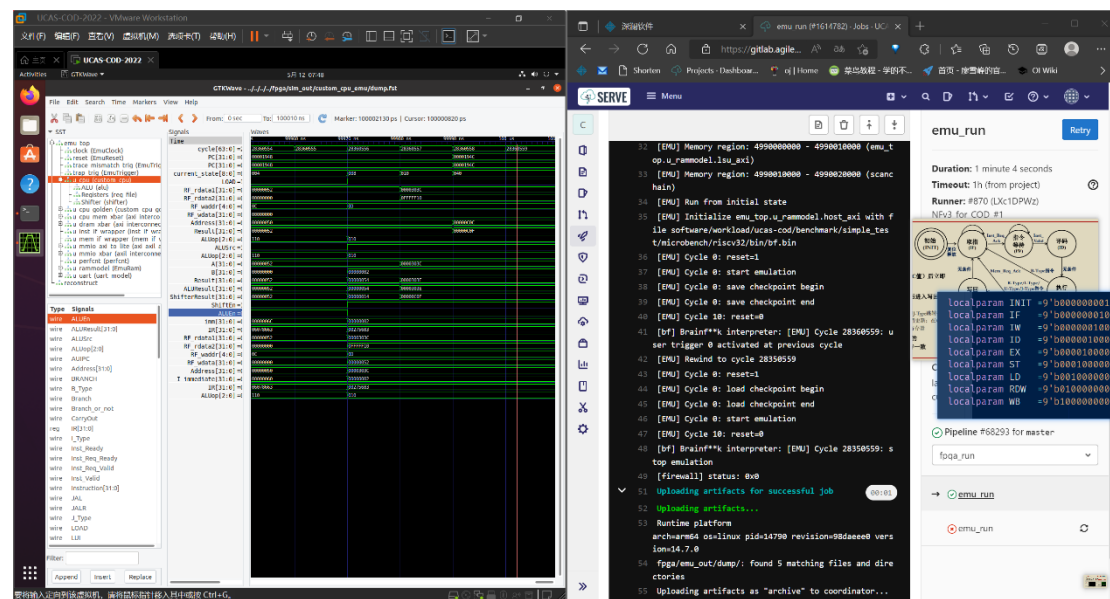
访存、写回与基于 MIPS 的实现类似，因此不再赘述。

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）

问题描述：Bhv Sim 通过但 Fpga Run 未通过。

在 FPGA 上运行时，microbench 组的 Brainf**k 编译器 Failed

原因分析：在运行仿真加速后，显示第 28360559 周期中断。



在对比金标准后，确定出错发生在第 28360556 周期。根据汇编文件可知当时运行的为 LH 指令。经过仔细观察波形图后，确定出错信号为 ALUEn 与 ShiftEn 信号。

这两个互斥的信号用于从 ALU 的计算结果和移位器的计算结果中选择最终计算结果。在出错的电路设计文件中，ALUEn 与 ShiftEn 由 LOAD 指令所属的 I_Type 类型生成，而非直接由 LOAD 指令生成；funct3 被用来判断 I_Type 类型的指令经由 ALU 还是移位器。因此，当 LOAD 指令对应的 funct3 字段满足 ShiftEn 的条件时，将错误选择移位器的计算结果，而非 ALU 的计算结果。

这个错误仅在 Fpga Run 中出现，猜测是因为 Bhv Sim 中未对所有的 LOAD 类型指令进行仿真。该问题仅会在 funct3[1]==1'b0 且 funct3[0]==1'b1 时触发，因此只有 LH 指令出错，具有一定隐蔽性。

```
assign R_Type = OP;
assign I_Type = OP_IMM | JALR | LOAD;
assign S_Type = STORE;
assign B_Type = BRANCH;
assign U_Type = LUI | AUIPC;
assign J_Type = JAL;
assign Branch = B_Type;
assign MemtoReg = LOAD;
assign ALUEn = (R_Type | I_Type) & (funct3[1] | ~funct3[0]) | S_Type | B_Type;
assign ShiftEn = (R_Type | I_Type) & (~funct3[1] & funct3[0]);
assign ALUSrc = I_Type | S_Type;
assign ShiftSrc = I_Type | S_Type;
assign RF_wen = current_state[isWB] & (J_Type | I_Type | R_Type | U_Type);
```

错误原因： LOAD时错误选择了Shifter而非ALU的计算结果

解决方法：

如下图所示，将 I_Type 拆分，根据 I_Type 类型中的具体指令内容选择计算结果。

```
assign R_Type = OP; // hints encoded in register specifiers used in the instruc-
assign I_Type = OP_IMM | JALR | LOAD;
assign S_Type = STORE;
assign B_Type = BRANCH; // these indirect-pump instructions to guide the
assign U_Type = LUI | AUIPC; // tied to register numbers and the
assign J_Type = JAL; // (x1 and x5) are given as rs1 and rd, then the RAS
// is both pushed and popped to support continuities. If rs1 and rd are the same link regis-
// ter (either x1 or x5), the RAS is only pushed to enable macro-op fusion of the sequences:
assign Branch_rs = B_Type; // jpe ra, imm20; jalr ra, ra, imm12
assign MemtoReg = LOAD;
assign ALUEn = (OP | OP_IMM & (funct3[1] | ~funct3[0]) | JALR | LOAD | S_Type | B_Type;
assign ShiftEn = (OP | OP_IMM & (~funct3[1] & funct3[0]));
assign ALUSrc = I_Type | S_Type; // but B-immediate encodes signed
assign ShiftSrc = I_Type | S_Type; // target address. The conditional
assign RF_wen = current_state[isWB] & (J_Type | I_Type | R_Type | U_Type);
```

解决方案：
把I_Type拆分

三、 对讲义中思考题（如有）的理解和回答

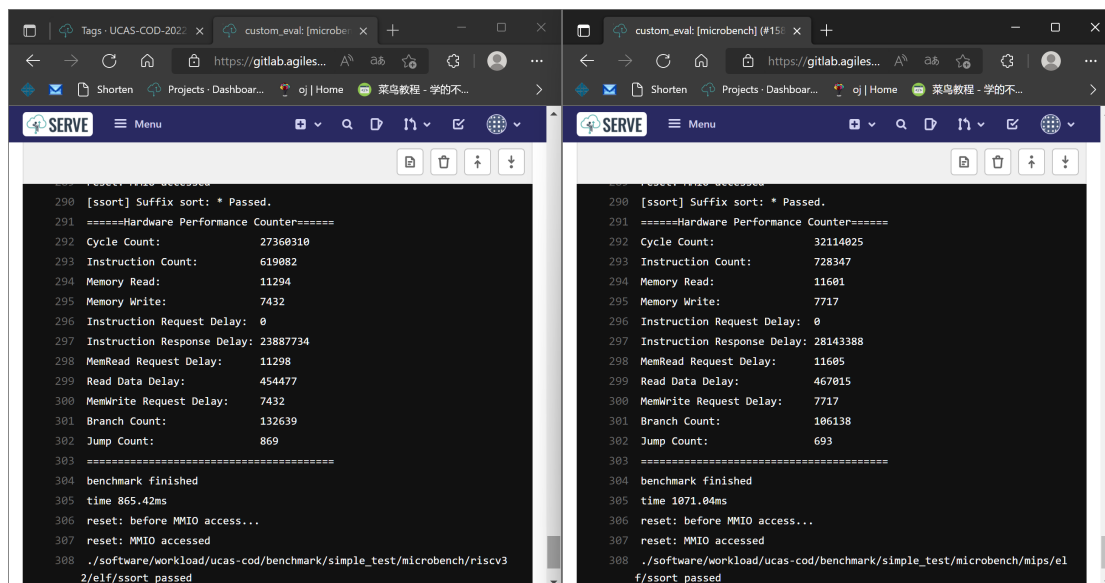
RISC-V/MIPS 指令集性能分析对比

1. RISC-V 指令集较 MIPS 指令集更规整

例如 RISC-V 指令中立即数的生成复用了各类型指令共有的部分,相较 MIPS 少用了一些数据选择器,成本更低。由此带来了更为简单优美的硬件电路设计。

2. RISC-V 指令集取消了分支延迟槽的设计,使得总指令数一般较 MIPS 集少

分支延迟槽主要用于提高多周期 CPU 流水线的性能。考虑到我们目前实现的多周期 CPU 都并不是流水结构,分支延迟槽并不能带来性能的提升,反而造成了汇编代码的冗长,带来不必要的指令开销。



```
290 [ssort] Suffix sort: * Passed.
291 =====Hardware Performance Counter=====
292 Cycle Count: 27360310
293 Instruction Count: 619082
294 Memory Read: 11294
295 Memory Write: 7432
296 Instruction Request Delay: 0
297 Instruction Response Delay: 23887734
298 MemRead Request Delay: 11298
299 Read Data Delay: 454477
300 MemWrite Request Delay: 7432
301 Branch Count: 132639
302 Jump Count: 869
303 =====
304 benchmark finished
305 time 865.42ms
306 reset: before MMIO access...
307 reset: MMIO accessed
308 ./software/workload/ucas-cod/benchmark/simple_test/microbench/riscv3
2/elf/ssort passed

290 [ssort] Suffix sort: * Passed.
291 =====Hardware Performance Counter=====
292 Cycle Count: 32114025
293 Instruction Count: 728347
294 Memory Read: 11601
295 Memory Write: 7717
296 Instruction Request Delay: 0
297 Instruction Response Delay: 28143388
298 MemRead Request Delay: 11605
299 Read Data Delay: 467015
300 MemWrite Request Delay: 7717
301 Branch Count: 106138
302 Jump Count: 693
303 =====
304 benchmark finished
305 time 1071.04ms
306 reset: before MMIO access...
307 reset: MMIO accessed
308 ./software/workload/ucas-cod/benchmark/simple_test/microbench/mips/el
f/ssort passed
```

这一点在上图的程序计数器中得以体现。左图基于 RISC-V 指令集，右图基于 MIPS 指令集。可以看到，对于 `ssort` 程序，左图总指令数较右图更少。相应地，总周期数也较右图更少。此外，还应注意到，Branch 指令数在总指令数中的占比不可忽略。综合以上信息，猜测是由于分支延迟槽的取消使得 Branch 指令后紧跟的延迟指令无需执行，由此带来了总指令数的减少，且减少量恰约为 Branch 指令总数。

汇编代码验证了这一猜测。对比 RISC-V 与 MIPS 的汇编代码，可以发现，MIPS 中有较多的 `nop` 指令，大都紧跟 Branch 指令。但 RISC-V 明显少了很多。这应该也是 RISC-V 在某些程序上性能明显好于 MIPS 的主要原因。

3. RISC-V32 指令集未包含非对齐访存指令，使得对编码空间的利用更高效

MIPS 指令集包含了非对齐访存指令。硬件电路较为复杂，性能较差。RISC-V 对非对齐访问的实现较 MIPS 指令集灵活，其编码的正交性更高。

四、 在课后，你花费了大约 15 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）

由于之前写过基于 MIPS 的 CPU，加上 RISC-V 指令集本身易于实现的特性，本次实验并不复杂。

但 RISC-V 指令手册的可读性比起 MIPS 较差。指令手册没有给出公式化的指令实现描述。需要投入较多精力研究 RISC-V 标准的具体含义。