# 中国科学院大学
# 《计算机组成原理(研讨课)》实验报告

姓名 韩初晓 学号 2023K8009908002 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。 |

# 1 电路设计及说明

## 1.1 MIPS 指令集与 RISC-V 指令集的区别

通过观察 MIPS 与 RISC-V 指令集,我们可以看出,我们需要实现的指令名称大部分相似,但是,两个指令集的译码逻辑有显著差别。主要体现在以下几点:

- MIPS 指令集主要分为 R、I 和 J 三类。然而 RISC-V 的指令分类则更加丰富,有 R、I、S、B、U 和 J 类型。
- MIPS 指令集的 opcode 在高位,而 RISC-V 指令集的 opcode 在低七位。
- 与 MIPS 指令集 `rs`,`rt` 和 `rd` 的划分和 RISC-V 中 `rs1`,`rs2` 和 `rd` 的划分有显著差别。不过幸运的是,在 RISC-V 中,如果有寄存器堆的写回操作,那么目的寄存器一定是 `rd`。
- RISC-V 指令集中功能码的划分和 MIPS 指令集有显著区别。MIPS 指令集含有功能码的指令均为五位功能码,而 RISC-V 则有 `funct3` 和 `funct7` 之分。
- RISC-V 指令集的分支类指令比较精简,删去了 MIPS 指令集中小于等于则跳转和大于等于则跳转的指令 `blez` 和 `bgez` 两个指令,这更加接近 ALU 的工作方式。
- `jump` 类指令被缩减为两种。`jal` 作为唯一一个 J 型指令,他将 PC 和立即数相加得到跳转地址;`jalr` 作为一个跳转指令, 将 `rs1` 中的数据和立即数相加, 得到跳转地址。RISC-V 指令集在返回地址的存储上相比 MIPS 指令集更加简洁——这两个跳转指令都会将返回地址存储在 `rd` 目的寄存器中。
- 在跳转地址的计算过程中, RISC-V 指令的处理方式更加简洁: 直接将 PC 与偏移量相加, 得到跳转地址, 而无需像 MIPS 指令集一样,先将 PC 与 4 相加后再加上偏移量。
- RISC-V 指令集删去了 MIPS 指令中繁琐的非对齐访存指令 `lwr`、`lwl`、`swr`、`swl`,处理起来容易得多。

关于不同的指令格式,在手册中有如下的详细描述:

*The alignment constraint for base ISA instructions is relaxed to a two-byte boundary when instruction extensions with 16-bit lengths or other odd multiples of 16-bit lengths are added.*

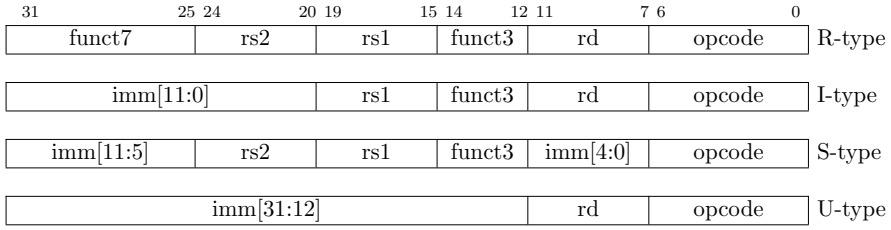


Figure 2.2: RISC-V base instruction formats. Each immediate subfield is labeled with the bit position (imm[$x$]) in the immediate value being produced, rather than the bit position within the instruction's immediate field as is usually done.

图 1: RISC-V 指令格式

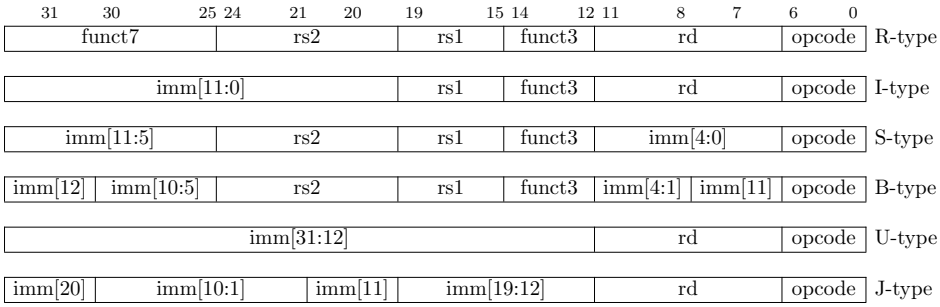


Figure 2.3: RISC-V base instruction formats showing immediate variants.

图 2: RISC-V 指令格式

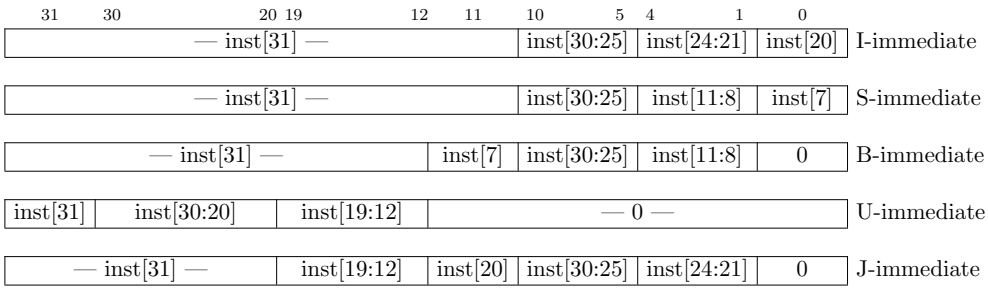


Figure 2.4: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses inst[31].

图 3: RISC-V 指令格式

根据指令集手册,我制作了 RISC-V 译码表如下:

| Instruction Category | Mnemonic | Format | Description | Opcode | Funct | rs Usage | rt Usage | rd Usage | shamt Usage | imm Usage | target Usage | RegReadAddr1 | RegReadAddr2 | ALUSrcA | ALUSrcB | ALUControl |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| U-Type Arithmetic | LUI | U | Load Upper Immediate (rt) imm -> rd | 0110111 | N/A | N/A | N/A | Dst Reg (rd) | N/A | U-Immediate (bits 31:12) | N/A | N/A | N/A | Zero | Imm (U-Type) | 010 |
| U-Type Arithmetic | AUIPC | U | ADD (Upper Immediate) to PC (PC + U-imm) -> rd | 0010111 | N/A | | | | | N/A | J-Immediate | | | PC | Imm (J-Type) | |
| J-Type Jump | JAL | J | Jump and Link (PC + J-imm -> PC, PC+4 -> rd) | 1101111 | | | | | | J-Immediate (Signed Offset) | | | | | JumpSE | |
| J-Type Jump | JALR | I | Jump and Link Register (rs1 + I-imm) & ~1 -> PC, PC+4 -> rd) | 1100111 | 000 | Base Addr Reg (rs1) | | | | | | | | | | |
| B-Type Branch | BEQ | B | Branch on Equal (if rs1 == rs2, PC + B-imm) | 1100011 | 000 | Compare Reg1 (rs1) | Compare Reg2 (rs2) | N/A | N/A | B-Immediate (Signed Offset) | | | rs2 | | RegData2 | 110 |
| B-Type Branch | BNE | | Branch on Not Equal (if rs1 != rs2, PC + B-imm) | | 001 | | | | | | | | | | | 111 |
| B-Type Branch | BLT | | Branch on Less Than (Signed) (if rs1 < rs2, PC + B-imm) | | 100 | | | | | | | | | | | |
| B-Type Branch | BGE | | Branch on Greater Than or Equal (Signed) (if rs1 >= rs2, PC + B-imm) | | 101 | | | | | | | | | | | 011 |
| B-Type Branch | BLTU | | Branch on Less Than (Unsigned) (if rs1 < rs2, PC + B-imm) | | 110 | | | | | | | | | | | |
| B-Type Branch | BGEU | | Branch on Greater Than or Equal (Unsigned) (if rs1 >= rs2, PC + B-imm) | | 111 | | | | | | | | | | | |
| I-Type Load | LB | I | Load Byte (Signed) (SignExt(Mem[rs1+I-imm])) -> rd) | 0000011 | 000 | Base Addr Reg (rs1) | N/A | Dst Reg (rd) | N/A | I-Immediate (Signed Offset) | N/A | rs1 | N/A | RegData1 | ImmSE | 010 |
| I-Type Load | LH | | Load Halfword (Signed) (SignExt(Mem[rs1+I-imm])) -> rd) | | 001 | | | | | | | | | | | |
| I-Type Load | LW | | Load Word (Mem[rs1+I-imm]) -> rd) | | 010 | | | | | | | | | | | |
| I-Type Load | LBU | | Load Byte (Unsigned) (ZeroExt(Mem[rs1+I-imm])) -> rd) | | 100 | | | | | | | | | | | |
| I-Type Load | LHU | | Load Halfword (Unsigned) (ZeroExt(Mem[rs1+I-imm])) -> rd) | | 101 | | | | | | | | | | | |
| S-Type Store | SB | S | Store Byte (rs2[7:0] -> Mem[rs1+S-imm]) | 0100011 | 000 | | Src Reg (rs2) | N/A | | S-Immediate (Signed Offset) | | | rs2 | | ImmSE | 010 |
| S-Type Store | SH | | Store Halfword (rs2[15:0] -> Mem[rs1+S-imm]) | | 001 | | | | | | | | | | | |
| S-Type Store | SW | | Store Word (rs2 -> Mem[rs1+S-imm]) | | 010 | | | | | | | | | | | |
| I-Type Arithmetic | ADDI | I | Add Immediate (rs1 + I-imm -> rd) | 0010011 | 000 | Src Reg (rs1) | N/A | | | I-Immediate (Signed) | | | | | | 111 |
| I-Type Compare | SLTI | | Set Less Than Immediate (Signed) (rs1 < I-imm) ? 1 : 0 -> rd) | | 010 | | | | | I-Immediate (Unsigned, effectively) | | | | | | 011 |
| I-Type Logical | XORI | | XOR Immediate (rs1 ^ I-imm -> rd) | | 100 | | | | | | | | | | | 100 |
| I-Type Logical | ORI | | OR Immediate (rs1 \| I-imm -> rd) | | 110 | | | | | I-Immediate (ZeroExt for logic) | | | | | ImmZE | 001 |
| I-Type Logical | ANDI | | AND Immediate (rs1 & I-imm -> rd) | | 111 | | | | | | | | | | | 000 |
| I-Type Shift | SLLI | | Shift Left Logical Immediate (rs1 << shamt -> rd) | | 001 | | | | Shift Amount (imm[4:0]) | I-Immediate (shamt in low 5 bits, funct7_like in upper 7) | | | | | N/A | 00 |
| I-Type Shift | SRLI | | Shift Right Logical Immediate (rs1 >> shamt -> rd) | | 101 | | | | | | | | | | | 10 |
| I-Type Shift | SRAI | | Shift Right Arithmetic Immediate (rs1 >>> shamt -> rd) | | 101 | | | | | | | | | | | 11 |
| R-Type Arithmetic | ADD | R | Add (rs1 + rs2 -> rd) | 0110011 | 000 | Src1 Reg (rs1) | Src2 Reg (rs2) | Dst Reg (rd) | N/A | N/A | N/A | rs1 | rs2 | RegData1 | RegData2 | 010 |
| R-Type Arithmetic | SUB | | Subtract (rs1 - rs2 -> rd) | | 000 | | | | | | | | | | | 110 |
| R-Type Shift | SLL | | Shift Left Logical (rs1 << rs2[4:0] -> rd) | | 001 | | Shift Amt Reg (rs2) | | | | | | | | RegData2[low 5 bits] | 00 |
| R-Type Compare | SLT | | Set Less Than (Signed) (rs1 < rs2) ? 1 : 0 -> rd) | | 010 | | Src2 Reg (rs2) | | | | | | | | RegData2 | 111 |
| R-Type Compare | SLTU | | XOR (rs1 ^ rs2 -> rd) | | 011 | | | | | | | | | | | 011 |
| R-Type Logical | XOR | | Shift Right Logical (rs1 >> rs2[4:0] -> rd) | | 100 | | Shift Amt Reg (rs2) | | | | | | | | RegData2[low 5 bits] | 10 |
| R-Type Shift | SRL | | Shift Right Arithmetic (rs1 >>> rs2[4:0] -> rd) | | 101 | | | | | | | | | | | 11 |
| R-Type Logical | OR | | OR (rs1 \| rs2 -> rd) | | 110 | | Src2 Reg (rs2) | | | | | | | | RegData2 | 001 |
| R-Type Logical | AND | | AND (rs1 & rs2 -> rd) | | 111 | | | | | | | | | | | 000 |

图 4: RISC-V 指令集译码表 (第一部分)

| Instruction Category | Mnemonic | Format | Description | ALU Operation Meaning | ShiftAmountSource | MemRead | MemWrite | MemAddressSource | MemWriteDataSource | MemWriteSize | BranchCondition | NextPCSrc | RegWriteEnable | RegWriteAddrSource | RegWriteDataSource |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| U-Type Arithmetic | LUI | U | Load Upper Immediate (rd = imm << 12) | Add (0 + U-imm) | | | | | | | N/A | PC+4 | | | ALUResult |
| U-Type Arithmetic | AUIPC | U | Add Upper Immediate to PC (PC + U-imm -> rd) | Add (PC + U-imm) | | | | | | | | | 1 | rd | |
| J-Type Jump | JAL | J | Jump and Link (PC + J-imm -> PC, PC+4 -> rd) | Add (for target addr calc) | | | | | | | Always Jump | JumpTargetAddr | | | PC+4 |
| I-Type Jump | JALR | I | Jump and Link Register (rs1 + I-imm) & ~1 -> PC, PC+4 -> rd) | Add (for target addr calc) | | | | | | | | JumpRegisterTarget(rs1+imm) | | | |
| B-Type Branch | BEQ | B | Branch on Equal (if rs1 == rs2, PC + B-imm) | Subtract (for compare) | | 0 | 0 | N/A | N/A | N/A | Zero == 1 | BranchTarget | 0 | N/A | N/A |
| B-Type Branch | BNE | | Branch on Not Equal (if rs1 != rs2, PC + B-imm) | | | | | | | | Zero == 0 | | | | |
| B-Type Branch | BLT | | Branch on Less Than (Signed) (if rs1 < rs2, PC + B-imm) | Set Less Than (Signed) | | | | | | | ALUResult == 1 | | | | |
| B-Type Branch | BGE | | Branch on Greater Than or Equal (Signed) (if rs1 >= rs2, PC + B-imm) | Set Less Than (Unsigned) | | | | | | | ALUResult == 0 | | | | |
| B-Type Branch | BLTU | | Branch on Less Than (Unsigned) (if rs1 >= rs2, PC + B-imm) | | | | | | | | ALUResult == 1 | | | | |
| B-Type Branch | BGEU | | Branch on Greater Than or Equal (Unsigned) (if rs1 >= rs2, PC + B-imm) | | | | | | | | ALUResult == 0 | | | | |
| I-Type Load | LB | I | Load Byte (Signed) (SignExt(Mem[rs1+I-imm])) -> rd) | Add (for address calc) | N/A | 1 | 0 | ALUResult | N/A | SignExt(MemData[7:0]) | N/A | PC+4 | 1 | rd | SignExt(MemReadData[7:0]) |
| I-Type Load | LH | | Load Halfword (Signed) (SignExt(Mem[rs1+I-imm])) -> rd) | | | | | | | SignExt(MemData[15:0]) | | | | | SignExt(MemReadData[15:0]) |
| I-Type Load | LW | | Load Word (Mem[rs1+I-imm]) -> rd) | | | | | | | MemData[31:0] | | | | | MemReadData |
| I-Type Load | LBU | | Load Byte (Unsigned) (ZeroExt(Mem[rs1+I-imm])) -> rd) | | | | | | | ZeroExt(MemData[7:0]) | | | | | ZeroExt(MemReadData[7:0]) |
| I-Type Load | LHU | | Load Halfword (Unsigned) (ZeroExt(Mem[rs1+I-imm])) -> rd) | | | | | | | ZeroExt(MemData[15:0]) | | | | | ZeroExt(MemReadData[15:0]) |
| S-Type Store | SB | S | Store Byte (rs2[7:0] -> Mem[rs1+S-imm]) | | | | 1 | | RegData2[low 8 bits] | Byte mask based on Addr[1:0] | | | 0 | N/A | N/A |
| S-Type Store | SH | | Store Halfword (rs2[15:0] -> Mem[rs1+S-imm]) | | | | | | RegData2[low 16 bits] | Halfword mask based on Addr[1] | | | | | |
| S-Type Store | SW | | Store Word (rs2 -> Mem[rs1+S-imm]) | | | | | | RegData2 | 1111 (All bytes) | | | | | |
| I-Type Arithmetic | ADDI | I | Add Immediate (rs1 + I-imm -> rd) | Add | | | | | | | | | | | |
| I-Type Compare | SLTI | | Set Less Than Immediate (Signed) (rs1 < I-imm) ? 1 : 0 -> rd) | Set Less Than (Signed) | | | | | | | | | | | ALUResult |
| I-Type Compare | SLTIU | | Set Less Than Immediate (Unsigned) (rs1 <u I-imm) ? 1 : 0 -> rd) | Set Less Than (Unsigned) | | | | | | | | | | | |
| I-Type Logical | XORI | I | XOR Immediate (rs1 ^ I-imm -> rd) | XOR | | | | | | | | | | | |
| I-Type Logical | ORI | | OR Immediate (rs1 \| I-imm -> rd) | OR | | | | | | | | | | | |
| I-Type Logical | ANDI | | AND Immediate (rs1 & I-imm -> rd) | AND | | | | | | | | | | | |
| I-Type Shift | SLLI | | Shift Left Logical Immediate (rs1 << shamt -> rd) | Shift Left Logical | shamt (imm[4:0]) | | | | | | | | | | ShifterResult |
| I-Type Shift | SRLI | | Shift Right Logical Immediate (rs1 >> shamt -> rd) | Shift Right Logical | | | | | | | | | | | |
| I-Type Shift | SRAI | | Shift Right Arithmetic Immediate (rs1 >>> shamt -> rd) | Shift Right Arithmetic | | | | | | | | | | | |
| R-Type Arithmetic | ADD | R | Add (rs1 + rs2 -> rd) | Add | N/A | | | | | | | | 1 | rd | ALUResult |
| R-Type Arithmetic | SUB | | Subtract (rs1 - rs2 -> rd) | Subtract | | | | | | | | | | | |
| R-Type Shift | SLL | | Shift Left Logical (rs1 << rs2[4:0] -> rd) | Shift Left Logical | rs2[low 5 bits] | | | | | | | | | | ShifterResult |
| R-Type Compare | SLT | | Set Less Than (Signed) (rs1 < rs2) ? 1 : 0 -> rd) | Set Less Than (Signed) | | | | | | | | | | | |
| R-Type Compare | SLTU | R | Set Less Than (Unsigned) (rs1 <u rs2) ? 1 : 0 -> rd) | Set Less Than (Unsigned) | N/A | | | | | | | | | | ALUResult |
| R-Type Logical | XOR | | XOR (rs1 ^ rs2 -> rd) | XOR | | | | | | | | | | | |
| R-Type Shift | SRL | | Shift Right Logical (rs1 >> rs2[4:0] -> rd) | Shift Right Logical | rs2[low 5 bits] | | | | | | | | | | ShifterResult |
| R-Type Shift | SRA | | Shift Right Arithmetic (rs1 >>> rs2[4:0] -> rd) | Shift Right Arithmetic | | | | | | | | | | | |
| R-Type Logical | OR | | OR (rs1 \| rs2 -> rd) | OR | N/A | | | | | | | | | | ALUResult |
| R-Type Logical | AND | | AND (rs1 & rs2 -> rd) | AND | | | | | | | | | | | |

图 5: RISC-V 指令集译码表 (第二部分)

## 1.2 关键模块结构设计说明

### 1.2.1 逻辑电路图

本次实验的电路图和第三次实验所设计的电路图基本相同,具体形式如下:

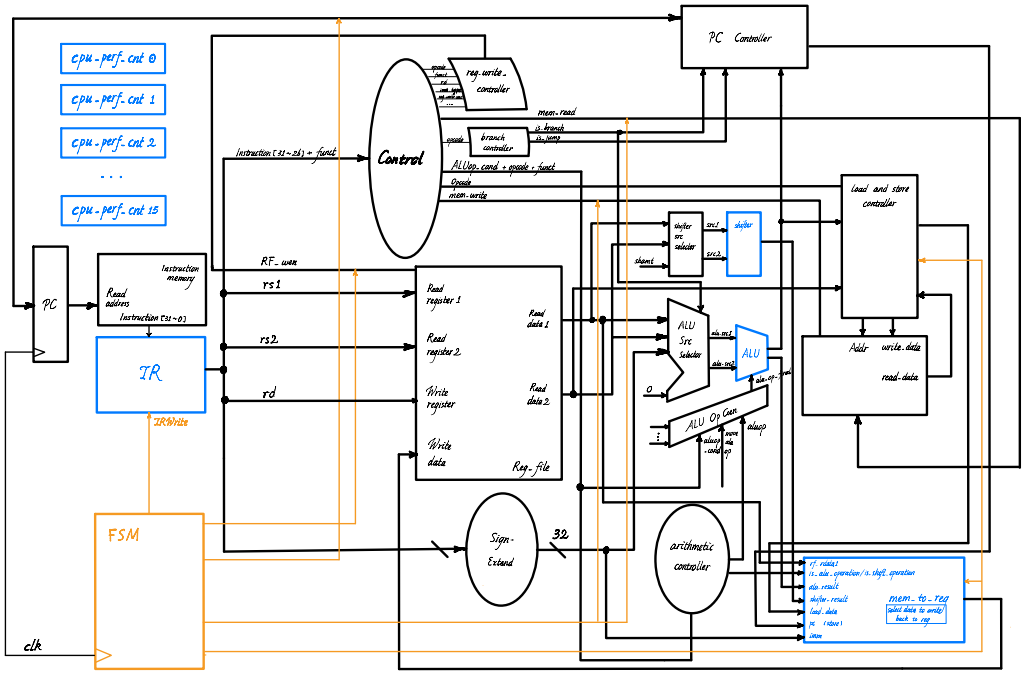


图 6: 定制 RISC-V 功能形处理器电路结构图

### 1.2.2 指令解码

相比 MIPS 指令集,RISC-V 指令集的编码格式比较整齐,因此可以针对不同的指令类型进行统一的解码:

```
//============================================================================
// PIPELINE STAGE 2: Instruction Decode / Register Fetch (ID)
//============================================================================

// -- Instruction Field Decoding --
assign    opcode  = IR[ 6: 0]; // Opcode field
assign    funct3  = IR[14:12]; // Funct3 field
assign    funct7  = IR[31:25]; // Funct7 field
assign    rs1     = IR[19:15]; // Source register 1
assign    rs2     = IR[24:20]; // Source register 2
assign    rd      = IR[11: 7]; // Destination register
assign    RF_waddr = rd;     // Register file write address

// -- Immediate type decoding --
assign    imm_I   = {{20{IR[31]}}, IR[31:20]};                  // Immediate value (I-type)
assign    imm_S   = {{20{IR[31]}}, IR[31:25], IR[11:7]};        // Immediate value (S-type)
assign    imm_B   = {{20{IR[31]}}, IR[7], IR[30:25], IR[11:8], 1'b0}; // Immediate value (B-type)
assign    imm_U   = {IR[31:12], 12'b0};                         // Immediate value (U-type)
assign    imm_J   = {{12{IR[31]}}, IR[19:12], IR[20], IR[30:21], 1'b0}; // Immediate value (J-type)
assign    imm     = (is_I) ? imm_I :
                    (is_S) ? imm_S :
                    (is_B) ? imm_B :
                    (is_U) ? imm_U :
                    (is_J) ? imm_J : 32'b0; // Default immediate value

// -- Instruction Type Decoding --
assign    is_OPIMM = (opcode == 7'b0010011); // OP-IMM instruction
assign    is_addi = (opcode == 7'b0010011) && (funct3 == 3'b000); // ADDI instruction
assign    is_ebreak = (opcode == 7'b1110011) && (funct3 == 3'b000) && (IR[31:20] ==
    12'b000000000001); // EBREAK instruction
assign    is_load = (opcode == 7'b0000011); // Load instruction
assign    is_jalr = (opcode == 7'b1100111); // JALR instruction
assign    is_lui  = (opcode == 7'b0110111); // LUI instruction
assign    is_auipc = (opcode == 7'b0010111); // AUIPC instruction

// -- Control Signals for Instruction Types --
assign    is_I    = is_OPIMM || is_load || is_jalr; // I-type instruction
assign    is_S    = (opcode == 7'b0100011);    // S-type instruction
assign    is_B    = (opcode == 7'b1100011);    // B-type instruction
assign    is_U    = is_lui || is_auipc;        // U-type instruction
assign    is_J    = (opcode == 7'b1101111);    // J-type instruction
assign    is_R    = (opcode == 7'b0110011);    // R-type instruction
```

### 1.2.3 ALU 和移位器源操作数选择逻辑

根据以上解码的结果,我们可以构建出 ALU 和移位器的操作数选择逻辑。相关代码如下:

```
// -- Decode ALU and shifter operations based on instruction type --
assign alu_src1 = (is_I || is_R || is_S || is_B) ? RF_rdata1 : // rs1 for most operations
                  (is_J)                         ? current_pc : // PC for JAL offset calculation
                  (is_lui)                       ? 32'b0      : // Zero for LUI (0 + imm)
                  (is_auipc)                     ? current_pc : // PC for AUIPC (PC + imm)
                                                   32'b0;     // Default to zero

assign alu_src2 = (is_I || is_J) ? imm : // Immediate for I-types (incl. JALR) and J-type (JAL)
                  (is_R)         ? RF_rdata2 : // rs2 for R-type operations
                  (is_S)         ? imm       : // Immediate for S-type (store address calculation)
                  (is_B)         ? RF_rdata2 : // rs2 for B-type (branch comparison)
                  (is_U)         ? imm       : // Immediate for U-type (LUI/AUIPC)
                                   32'b0;     // Default to zero

assign shifter_src1 = RF_rdata1;                 // Source for shift operations is always rs1
assign shifter_src2 = (is_R) ? RF_rdata2[4:0] : // R-type shifts use lower 5 bits of rs2
                      (is_I) ? imm[4:0]     :    // I-type shifts use lower 5 bits of immediate
                               5'b0;               // Default
```

### 1.2.4 PC 及其更新模块

在设计 RISC-V 处理器的过程中，PC 的更新仍然是十分重要的一部分。为此，我们仍然需要计算出是否 branch_condition_satisfied 等信号。在我的设计中，我首先计算出了跳转指令和分支指令的目标地址，随后，计算分支条件是否满足的信号。最终，下一个 PC 通过上述信号选择得出。这一部分的相关代码如下：

```
// Branch Target Address Calculation
wire [31:0] branch_target = current_pc + imm_B - 4;

// Branch condition satisfied logic
assign branch_condition_satisfied = (current_state == EX) ?
                                    (is_B ?
                                    // Check funct3 to determine the specific branch condition
                                    ( (funct3 == 3'b000) ? alu_zero          :
                                      (funct3 == 3'b001) ? !alu_zero         :
                                      (funct3 == 3'b100) ? alu_result[0]     :
                                      (funct3 == 3'b101) ? !alu_result[0]    :
                                      (funct3 == 3'b110) ? alu_result[0]     :
                                      (funct3 == 3'b111) ? !alu_result[0]    :
                                                           1'b0
                                    ) :
                                    1'b0) :
                                    1'b0;

// Jump Target Address Calculation
wire [31:0] jump_target = (is_J) ? alu_result - 4            :
                          (is_jalr) ? ((alu_result) & 32'hFFFFFFFE) :
                          32'b0;
// -- PC Update Logic --
```

```
assign    next_pc   = ((is_J || is_jalr) && current_state == EX) ? jump_target :
                      (is_B && branch_condition_satisfied) ? branch_target :
                      current_pc + 4;

// -- PC Store Logic --
wire [31:0] pc_store;
assign    pc_store  = (is_J || is_jalr) ? current_pc :
                      32'b0;
```

### 1.2.5 执行阶段模块实例化

在执行 (EX) 阶段,我们需要对 ALU 和移位器进行实例化以及接线,同时,为了更好地适配多周期 CPU 的设计要求,我使用时序逻辑来锁存 ALU 和移位器的计算结果。相关的代码如下:

```
//=============================================================================
// PIPELINE STAGE 3: Execute (EX)
//=============================================================================

//-- ALU Execution --
wire [31:0] alu_out;           // ALU computation result

alu instance_alu (
    .A        (alu_src1),      // Input A from Src Sel
    .B        (alu_src2),      // Input B from Src Sel
    .ALUop    (alu_op),        // Input Opcode from Op Gen
    .Overflow (),
    .CarryOut (),
    .Zero     (alu_zero),      // Output: Zero flag -> To EX (PC Ctrl), WB
    .Result   (alu_out)        // Output: ALU computation result -> To EX (PC Ctrl), MEM, WB
);

always @(posedge clk) begin
    alu_result <= alu_out;     // Store ALU result for MEM stage
end

//-- Shifter Execution --
wire [31:0] shifter_out;          // Shifter computation result

shifter instance_shifter (
    .A        (shifter_src1),     // Input Data from Src Sel
    .B        (shifter_src2),     // Input Shift Amount from Src Sel
    .Shiftop  (shifter_op),       // Input Opcode from ID
    .Result   (shifter_out)       // Output: Shifter result -> To WB
);

always @(posedge clk) begin
    shift_result <= shifter_out;  // Store Shifter result for WB stage
end
```

### 1.2.6 访存阶段逻辑重构

RISC-V 指令集的访存指令相较于 MIPS 指令集存在较大差异。因此,需要对原先的 MIPS 多周期处理器的访存指令部分进行重构 (事实上,是适配 RISC-V 功能码并删除掉非对齐访存部分),相关代码如下:

```
//================================================================================
// PIPELINE STAGE 4: Memory Access (MEM)
//================================================================================

assign    Address    = (is_load || is_S) ? alu_result : 32'b0; // Address for load/store operations
wire [ 1:0] addr_offset = Address[1:0];                    // Address offset for load/store
    operations
wire      addr_0     = (addr_offset == 2'b00);
wire      addr_1     = (addr_offset == 2'b01);
wire      addr_2     = (addr_offset == 2'b10);
wire      addr_3     = (addr_offset == 2'b11);
wire      funct3_SB  = (funct3 == 3'b000);
wire      funct3_SH  = (funct3 == 3'b001);
wire      funct3_SW  = (funct3 == 3'b010);
wire      funct3_LB  = (funct3 == 3'b000);
wire      funct3_LH  = (funct3 == 3'b001);
wire      funct3_LW  = (funct3 == 3'b010);
wire      funct3_LBU = (funct3 == 3'b100);
wire      funct3_LHU = (funct3 == 3'b101);
wire      mem_write_internal = (current_state == ST) && is_S;
wire      mem_read_internal = (current_state == LD) && is_load;

// Memory Write Strobe Generation (Uses ID signals, EX result offset)
assign    Write_strb = {4{mem_write_internal}} & ( // Use is_store (mem_write_internal) from ID
                        (({4{funct3_SB}} & ((addr_0 ? 4'b0001 : 0) | (addr_1 ? 4'b0010 : 0) |
                            (addr_2 ? 4'b0100 : 0) | (addr_3 ? 4'b1000 : 0))) | // SB
                        ({4{funct3_SH}} & ((addr_0 ? 4'b0011 : 0) | (addr_2 ? 4'b1100 : 0))) // SH
                        ({4{funct3_SW}} & (addr_0 ? 4'b1111 : 4'b0000)) // SW
                       )); // -> To Memory Interface

assign    Write_data = {32{mem_write_internal}} & (
                        (({32{funct3_SB}} & {4{RF_rdata2[7:0]}}) | // SB
                        ({32{funct3_SH}} & {2{RF_rdata2[15:0]}}) | // SH
                        ({32{funct3_SW}} & RF_rdata2)        // SW
                      ));

assign    load_data       = (current_state == RDW && next_state == WB) ?
                            ( (funct3_LB || funct3_LBU) ? // LB/LBU
                              (addr_0 ? {{24{(~funct3[2] & Read_data [ 7])}}, Read_data [ 7: 0]} :
                               addr_1 ? {{24{(~funct3[2] & Read_data [15])}}, Read_data [15: 8]} :
                               addr_2 ? {{24{(~funct3[2] & Read_data [23])}}, Read_data [23:16]} :
                                        {{24{(~funct3[2] & Read_data [31])}}, Read_data [31:24]} )

                            : (funct3_LH || funct3_LHU) ? // LH/LHU
                              (addr_0 ? {{16{(~funct3[2] & Read_data [15])}}, Read_data [15: 0]} :
```

```
                                addr_2 ? {{16{(~funct3[2] & Read_data [31])}}, Read_data [31:16]} :
                                        32'b0 )

                              : (funct3_LW) ? Read_data // LW

                              : 32'b0 ) // Default unknown load
                              : 32'b0; // Default if not a load
```

### 1.2.7 写回数据源选择器逻辑重构

由于指令集上的区别，写回数据源选择器的逻辑需要根据译码表重构。类似的，由于写回操作是在一个指令周期结束后的第一个时钟上升沿发生的，因此，我同样利用时序逻辑锁存了选择出来的数据。相关代码如下：

```
//=============================================================================
// PIPELINE STAGE 5: Write Back (WB)
//=============================================================================

// -- Register Write Address and Enable Logic --
assign     RF_waddr         = rd;
assign     RF_wen           = Regwrite_fsm && (is_I || is_R || is_J || is_U) && (rd != 0);

//-- Write Back Data Selection (Uses ID signals, EX results, MEM results) --
wire [31:0] _RF_wdata;
assign     _RF_wdata        = (current_state == RDW && next_state == WB) ? load_data :
                              (is_J || is_jalr)                    ? pc_store   :
                              (is_U)                               ? alu_result :
                              ((is_R || is_I) && is_alu_operation)  ? alu_result :
                              ((is_R || is_I) && is_shifter_operation) ? shift_result :
                              32'b0;

always @(posedge clk) begin
    RF_wdata <= _RF_wdata;
end
```

### 1.2.8 状态机设计

控制 RISC-V 处理器运行的全局状态机的基本结构和 MIPS 处理器的情况基本类似。只不过，状态转移的条件根据指令集的不同有一些细微的变化。我完全参照了讲义上的状态转移图设计了这个状态机，相关代码如下：

```
// -- FSM Next State Logic --
always @(*) begin
  case (current_state)
    INIT : begin
      if (rst) begin
        next_state = INIT; // Reset state
      end
      else begin
        next_state = IF;
      end
```

```
    end
    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
    EX : begin
      if (is_B) begin
        next_state = IF;
      end
      else if (is_load) begin
        next_state = LD;
      end
      else if (is_R || is_I || is_U || is_J) begin
        next_state = WB;
      end
      else if (is_S) begin
        next_state = ST;
      end
      else if (is_ebreak) begin
        next_state = IF;
      end
      else begin
        next_state = IF;
      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
    ST : begin
      if (Mem_Req_Ready) begin
        next_state = IF;
      end
      else begin
        next_state = ST;
      end
    end
    WB : next_state = IF;
    default: next_state = IF;
  endcase
  end
```

### 1.2.9 性能计数器

在本次实验中,我一共实现了 16 个性能计数器,它们如下:

```
//------------------------------------------------------------------------
// Performance Counters
//------------------------------------------------------------------------
  reg [31:0] perf_cycle_count;       // cnt_0
  reg [31:0] perf_retired_inst_count; // cnt_1
  reg [31:0] perf_retired_load_count; // cnt_2
  reg [31:0] perf_retired_store_count; // cnt_3
  reg [31:0] perf_branch_executed_count; // cnt_4
  reg [31:0] perf_branch_taken_count; // cnt_5
  reg [31:0] perf_if_stall_count;    // cnt_6
  reg [31:0] perf_mem_access_stall_count; // cnt_7 (LD/ST stalls on Mem_Req_Ready)
  reg [31:0] perf_iw_stall_count;    // cnt_8
  reg [31:0] perf_rdw_stall_count;   // cnt_9
  reg [31:0] perf_jump_executed_count; // cnt_10
  reg [31:0] perf_alu_op_executed_count; // cnt_11
  reg [31:0] perf_shift_op_executed_count; // cnt_12
  reg [31:0] perf_nop_in_id_count;   // cnt_13
  reg [31:0] perf_total_mem_ops_count; // cnt_14
  reg [31:0] perf_reg_writes_count;  // cnt_15
```

以其中的一个为例,下面展现性能计数器的更新逻辑以及端口的连接:

首先,我定义了一个 `increment` 信号,用来判断计数器何时增加:

```
  assign increment_retired_load = (current_state == WB && RF_wen && !is_nop && is_load);
```

随后,在每个时钟上升沿执行判断逻辑,并更新性能计数器。最后,将我们自定义的 reg 变量接入处理器的输

出接口：

```
// cnt_2: Retired Load Instruction Count
always @(posedge clk) begin
  if (rst) begin
    perf_retired_load_count <= 32'd0;
  end else if (increment_retired_load) begin
    perf_retired_load_count <= perf_retired_load_count + 1;
  end
end
assign cpu_perf_cnt_2 = perf_retired_load_count;
```

## 2 实验过程中遇到的问题、对问题的思考过程及解决方法

由于本次的 RISC-V 多周期 CPU 是在 MIPS 多周期 CPU 的基础上经过重构得到的，所以 debug 的过程要轻松很多。我遇到的第一个错误便是在写完初稿之后，测试程序中移位器的计算总是出错。经过检查后发现在重构 `shifter_op` 的逻辑的过程中，我忘记定义 `shifter_op` 的 wire 信号，导致其被 implicit 定义为一个一位宽的信号。因此，移位器一直在按照输入的一位宽操作码进行错误的运算。因此，在进行 Verilog 代码的编写时，一定要检查位宽是否有问题。事实上，这些问题会在执行 Verilator 仿真命令的时候作为 WARNING 输出。因此，如果能够注意到这些 WARNING 并且试图完全修复它们，可以很大程度上修复位宽不匹配的问题。

## 3 实验结果

### 3.1 仿真结果

下面是仿真过程中的波形图，这些波形图能够展示上述模块工作的过程。

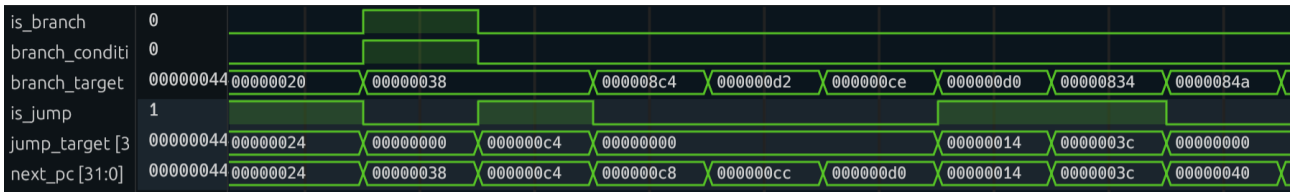


图 7: PC 及分支控制器仿真波形图

由图可见，处理器能正确的处理跳转和分支指令，生成正确的 PC。


图 8: 访存操作仿真波形图

由图可见，处理器能够正确的处理访存操作，并从内存获取或向内存写入数据。

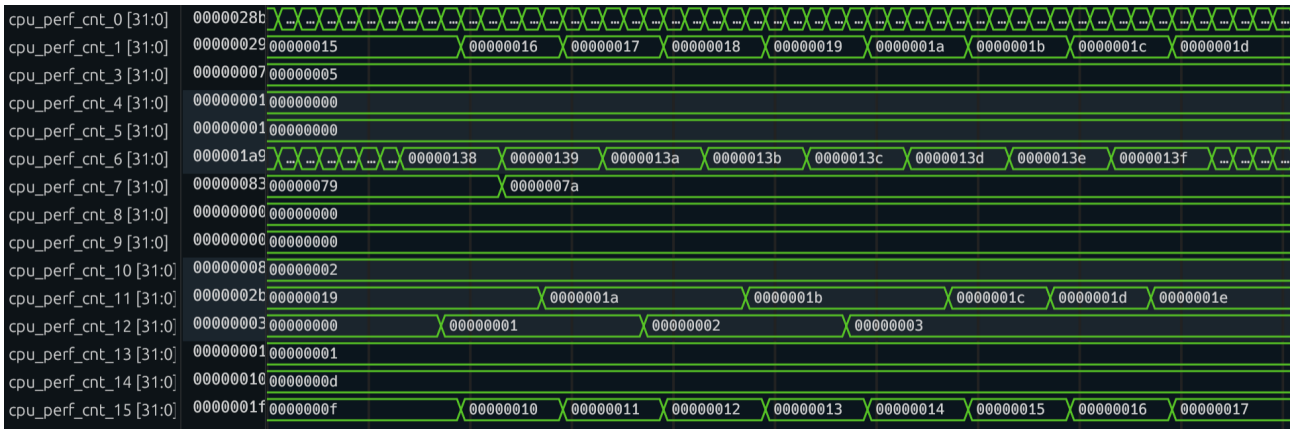


图 9: 性能计数器仿真波形图

由图可见,性能计数器能够正确的更新,统计出程序执行过程中的性能数据。

## 3.2 UART 控制器打印结果

hello 程序的终端打印结果如下:

```
$ bash ./fpga/design/ucas-cod/run/simple_cpu/fpga_run.sh $BENCH_SUITE $TARGET_DESIGN $CPU_ISA
RUNNER_CNT = 4
Completed FPGA configuration
Launching hello benchmark...
tggetattr: Inappropriate ioctl for device
reset: before MMIO access...
reset: MMIO accessed
axi_firewall_unblock: firewall error status: 00000000
main: before DDR accessing...
main: DDR accessed...
reset: before MMIO access...
reset: MMIO accessed
testing 1 2 0000003
faster and "cheaper"
deadf00d % DEADF00D
00000000100000000200000000300000000400000000 5
50 50 -50 4294967246
time 10413.29ms
reset: before MMIO access...
reset: MMIO accessed
./software/workload/ucas-cod/benchmark/simple_test/hello/riscv32/elf/hello passed
Hit good trap
pass 1 / 1
Job succeeded
```

可以看出,hello 程序的打印结果和预期相同,程序执行正常。

## 3.3 性能计数器统计结果分析

由于 microbench 基准测试程序集会打印性能计数器的值，因此，我们可以通过将 RISC-V 指令集和 MIPS 指令集的统计数据进行对比，得到两个指令集的一些差异。以 bf 基准测试为例，二者的输出结果确实呈现出了明显的差异：

Listing 1: RISC-V 统计结果

```
 --- Performance Counters for [bf] ---
   Cycles:                 38814005
   Retired Instructions:   404967
   Retired Loads:          94513
   Retired Stores:         5981
   Branches Executed:      91039
   Branches Taken:         36085
   IF Stalls:              0
   MEM Access Stalls:      112450
   IW Stalls:              29897078
   RDW Stalls:             6443174
   Jumps Executed:         47906
   ALU Ops Executed:       135350
   Shift Ops Executed:     78063
   NOPs in ID:             0
   Total MEM Ops Issued:   100490
   Register Writes:        307959
 --- End of Counters for [bf] ---
benchmark finished
time 741.36ms
```

Listing 2: MIPS 统计结果

```
 --- Performance Counters for [bf] ---
   Cycles:                 46345377
   Retired Instructions:   471921
   Retired Loads:          94513
   Retired Stores:         5977
   Branches Executed:      106688
   Branches Taken:         51760
   IF Stalls:              0
   MEM Access Stalls:      112437
   IW Stalls:              37019296
   RDW Stalls:             6446951
   Jumps Executed:         32234
   ALU Ops Executed:       187025
   Shift Ops Executed:     77707
   NOPs in ID:             54928
   Total MEM Ops Issued:   100484
   Register Writes:        359266
 --- End of Counters for [bf] ---
benchmark finished
time 823.23ms
```

### 3.3.1 性能计数器数据摘要

表 1: 性能计数器数据摘要

| 基准测试 | 指令集 | 时钟周期 | 完成指令数 | Load 指令数 | Store 指令数 | 分支执行数 | 分支成功数 | 跳转执行数 | ALU 操作数 | 移位操作数 | ID 阶段 NOP 数 | CPI |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **15pz** | RISC-V | 522,417,119 | 5,223,425 | 2,109,998 | 1,118,774 | 662,796 | 624,442 | 1,271 | 728,807 | 77,933 | 0 | 100.01 |
| | MIPS | 528,333,109 | 5,235,004 | 2,112,798 | 1,119,015 | 647,736 | 631,463 | 397 | 742,434 | 87,939 | 16,273 | 100.92 |
| **bf** | RISC-V | 38,814,005 | 404,967 | 94,513 | 5,981 | 91,039 | 36,085 | 47,906 | 135,350 | 78,063 | 0 | 95.84 |
| | MIPS | 46,345,377 | 471,921 | 94,513 | 5,977 | 106,688 | 51,760 | 32,234 | 187,025 | 77,707 | 54,928 | 98.20 |
| **dinic** | RISC-V | 1,477,047 | 16,479 | 4,042 | 1,534 | 2,137 | 950 | 327 | 6,663 | 1,696 | 0 | 89.63 |
| | MIPS | 1,704,899 | 18,331 | 4,640 | 1,895 | 2,169 | 1,166 | 24 | 7,194 | 2,121 | 1,003 | 92.99 |
| **fib** | RISC-V | 181,095,473 | 2,540,029 | 4,326 | 987 | 577,540 | 465,961 | 12,122 | 1,145,997 | 805,923 | 0 | 71.30 |
| | MIPS | 179,807,706 | 2,110,667 | 4,416 | 983 | 398,164 | 382,537 | 5,220 | 965,964 | 735,919 | 15,627 | 85.19 |
| **md5** | RISC-V | 387,519 | 4,865 | 545 | 107 | 394 | 291 | 68 | 3,306 | 497 | 0 | 79.65 |
| | MIPS | 407,136 | 4,930 | 484 | 103 | 588 | 263 | 4 | 3,346 | 401 | 325 | 82.58 |
| **qsort** | RISC-V | 787,703 | 9,393 | 1,524 | 950 | 1,678 | 1,245 | 170 | 4,246 | 922 | 0 | 83.86 |
| | MIPS | 704,799 | 7,869 | 1,526 | 947 | 1,713 | 1,280 | 135 | 3,165 | 452 | 433 | 89.56 |
| **queen** | RISC-V | 6,951,385 | 79,447 | 14,420 | 12,362 | 6,078 | 1,471 | 4,118 | 38,354 | 4,112 | 0 | 87.50 |
| | MIPS | 6,826,223 | 74,220 | 14,422 | 12,359 | 6,722 | 2,115 | 4,118 | 32,492 | 4,112 | 4,607 | 91.97 |
| **sieve** | RISC-V | 747,876 | 10,167 | 345 | 135 | 2,496 | 1,355 | 46 | 4,537 | 2,645 | 0 | 73.56 |
| | MIPS | 1,194,088 | 14,406 | 349 | 133 | 1,847 | 1,667 | 4 | 7,252 | 4,824 | 180 | 82.89 |
| **ssort** | RISC-V | 44,780,563 | 618,538 | 11,291 | 7,433 | 132,631 | 66,423 | 869 | 283,903 | 181,542 | 0 | 72.40 |
| | MIPS | 52,491,709 | 644,949 | 11,600 | 7,715 | 106,130 | 61,552 | 693 | 323,542 | 194,231 | 44,578 | 81.39 |

上述表格中，我们可以看出，所有测试的 CPI 值都非常高（例如 70-100）。这主要是由于大量 IW Stalls 和

RDW Stalls 造成的,这些停顿主导了 Cycles 计数。这些停顿更多地反映了内存延迟,而非根本的指令集架构差异。为了比较指令集架构的特性,我们将更关注指令数量和类型。

#### 3.3.2 RISC-V 指令集和 MIPS 指令集的差异比较

1. **分支延迟槽:**

   - **MIPS:** MIPS 的重要特性它有是分支延迟槽。这意味着在分支指令之后的那条指令总是会被执行,无论分支是否跳转。编译器会尝试用一条有用的指令来填充这个延迟槽;如果无法找到,则会插入一条 NOP (空操作) 指令。
   - **RISC-V:** RISC-V 没有分支延迟槽。这简化了后续流水线设计的工作,可以产生更简洁的代码。
   - **数据证据:** 在表格中,NOP 指令的计数体现了这一点。
     - **NOPs in ID:**
       * RISC-V: 在所有基准测试中始终为 **0**。
       * MIPS: 在每个基准测试中都显示出大量的 NOPs in ID (例如,15pz 为 16,273,bf 为 54,928,ssort 为 44,578)。这些几乎可以肯定是为分支延迟槽插入的 NOP 指令。

2. **指令数量:**

   - 通常情况下,对于相同的任务,MIPS 倾向于有略高的 Retired Instructions 数量 (例如 bf, dinic, ssort)。这种增加的很大一部分都是因为分支延迟槽插入了 NOP 指令导致的。
   - 例如,在 bf 测试中:
     - RISC-V: 404,967 条完成指令。
     - MIPS: 471,921 条完成指令。差值为 66,954,而 MIPS 有 54,928 条 NOP 指令。这占据了两个程序指令数量差异的很大一部分。
   - 也存在例外情况,如 fib 和 qsort,其中 MIPS 的完成指令数较少。这可能是指令集的特性导致的、针对 MIPS 对这些算法的特定编译器优化,导致 MIPS 的指令数目比 RISC-V 稍微少一些。

3. **跳转和分支:**

   - **Jumps Executed:** RISC-V 通常显示出比 MIPS 更高的 Jumps Executed 数量。MIPS 在 dinic、md5、sieve 这些程序中跳转的数量非常少,表明它可能非常依赖分支指令。

# 4 对讲义中思考题(如有)的理解和回答

未见讲义中的思考题。

# 5 实验所耗时间

在课后,你花费了大约 30 小时完成此次实验。

在本次实验中,我成功完成了定制 RISC-V 功能型处理器的设计与实现。这一过程基于对先前 MIPS 多周期处理器的重构,使我对不同指令集架构(ISA)在设计理念与实现细节上的差异有了更为深刻的理解。我重点设计并验证了 RISC-V 处理器的核心模块,包括指令译码逻辑、ALU 及移位器操作数选择机制、PC 更新与控制流管理、访存单元以及写回数据通路,并通过一个多周期状态机对处理器进行精确控制。此外,实验中集成的 16 个性能计数器,为后续的性能对比分析提供了关键数据支持。

通过对 RISC-V 与 MIPS 处理器在相同基准测试下性能数据的对比,我观察到了两者在指令执行效率上的一些显著差异。一个突出的发现是 MIPS 指令集的分支延迟槽特性。在执行如 bf 这种控制流密集型程序时,

MIPS 处理器由于需要填充延迟槽而引入了大量 NOP 指令，这直接导致了其总执行指令数和时钟周期的增加。相比之下，RISC-V 由于取消了分支延迟槽，在这些场景下展现出更高的指令执行效率。这一对比让我直观地认识到 ISA 设计决策对最终处理器性能的实际影响。

在解决实验过程中遇到的诸如 Verilog 代码位宽不匹配等具体问题时，我不仅提升了硬件调试的技能，也进一步体会到硬件设计的严谨性和关注编译器警告的重要性。

总体而言，本次实验不仅是对 Verilog HDL 编程和硬件调试能力的有效锻炼，更重要的是，通过从 MIPS 到 RISC-V 的设计迁移和深入的对比分析，我对计算机组成原理的核心概念，如指令集架构的权衡、数据通路的设计优化、控制逻辑的实现复杂度以及性能评估方法等，形成了更为具体和实践性的认识。我相信这些经验将为我后续深入学习更高级的处理器设计奠定坚实的基础。

# 6 致谢

感谢宁彦祯同学和宋俊仪同学，他们的真知灼见给了我很大的启发；感谢朱徐塬学长及各位助教，您们的帮助对我有非常大的指导意义。