国科大计算机组成原理实验
prj4-RISCV指令集处理器实现

李国峰 2018K8009922027

2021年6月21日

实验设计

译码逻辑

主要是译码器的设计需要根据RISCV指令集的特点来进行，其余部分如状态机、访存接口设计等与上一实验完全一致，不再赘述。

与MIPS指令格式相比，RISCV指令中寄存器的源操作数域和目的操作数域是固定的，这对于译码器的实现提供了很大的便利，这意味着不再需要像MIPS一样根据具体的指令去选择将哪个域的地址接入到寄存器堆的地址端口，直接将RISCV指令码相应的域直接接入即可。

```
assign rs1    = inst_reg[19:15];
assign rs2    = inst_reg[24:20];
assign rd     = inst_reg[11: 7];
assign opcode = inst_reg[ 6: 0];
assign funct3 = inst_reg[14:12];
assign funct7 = inst_reg[31:25];

reg_file registers(
    .clk   (clk     ),
    .rst   (rst     ),
    .waddr (RF_waddr),
    .raddr1(rs1     ),
    .raddr2(rs2     ),
    .wen   (RF_wen  ),
    .wdata (RF_wdata),
    .rdata1(rs1_value),
    .rdata2(rs2_value)
);
```

RISCV指令类型所做的运算较为固定，大多数指令可以直接根据指令类型进行ALUop的生成，少部分指令例如算数逻辑运算指令需要根据`funct3`和`funct7`进一步进行译码。

```
assign r_type = opcode ==7'b0110011;
```

```
assign i_type =  opcode == 7'b1100111 ||
                 opcode == 7'b0000011 ||
                 opcode == 7'b0010011  ;

assign s_type = opcode == 7'b0100011;

assign b_type = opcode == 7'b1100011;

assign u_type = opcode == 7'b0110111 ||
                opcode == 7'b0010111  ;

assign j_type = opcode == 7'b1101111;

assign aluop[ 0] = r_type && funct3 == 3'b000 && funct7 == 7'b0000000 ||
                   i_type && opcode == 7'b1100111                     ||
                   i_type && opcode == 7'b0000011                     ||
                   i_type && opcode == 7'b0010011 && funct3 == 3'b000 ||
                   s_type                                             ||
                   j_type ;

assign aluop[ 1] = r_type && funct3 == 3'b000 && funct7 == 7'b0100000;

assign aluop[ 2] = r_type && funct3 == 3'b111 && funct7 == 7'b0000000 ||
                   i_type && funct3 == 3'b111 && opcode == 7'b0010011 ;

assign aluop[ 3] = r_type && funct3 == 3'b110 && funct7 == 7'b0000000 ||
                   i_type && funct3 == 3'b110 && opcode == 7'b0010011 ;

assign aluop[ 4] = u_type && opcode == 7'b0010111;

assign aluop[ 5] = r_type && funct3 == 3'b100 && funct7 == 7'b0000000 ||
                   i_type && funct3 == 3'b100 && opcode == 7'b0010011 ;

assign aluop[ 6] = r_type && funct3 == 3'b010 && funct7 == 7'b0000000 ||
                   i_type && funct3 == 3'b010 && opcode == 7'b0010011 ||
                   b_type && funct3 == 3'b100                         ||
                   b_type && funct3 == 3'b101 ;

assign aluop[ 7] = r_type && funct3 == 3'b011 && funct7 == 7'b0000000 ||
                   i_type && funct3 == 3'b011 && opcode == 7'b0010011 ||
                   b_type && funct3 == 3'b110                         ||
                   b_type && funct3 == 3'b111 ;

assign aluop[ 8] = r_type && funct3 == 3'b001 && funct7 == 7'b0000000 ||
                   i_type && funct3 == 3'b001 && funct7 == 7'b0000000 && opcode
== 7'b0010011 ;

assign aluop[ 9] = r_type && funct3 == 3'b101 && funct7 == 7'b0000000 ||
                   i_type && funct3 == 3'b101 && funct7 == 7'b0000000 && opcode
== 7'b0010011;

assign aluop[10] = r_type && funct3 == 3'b101 && funct7 == 7'b0100000 ||
                   i_type && funct3 == 3'b101 && funct7 == 7'b0100000 && opcode
== 7'b0010011 ;

assign aluop[11] = u_type && opcode == 7'b0110111;
```

有一些指令的源操作数1是`PC`，因此需要考虑`src1_is_pc`的生成。显然，`j_type`指令（即`jal`指令）的源操作数是`PC`；`i_type`指令中的`jalr`指令；以及`u_type`中的`auipc`。因此译码逻辑如下。

```
assign src1_is_pc = j_type ||
                    i_type && opcode == 7'b1100111 ||
                    u_type && opcode == 7'b0010111  ;
```

其他控制信号同理，不再赘述。

跳转指令

与MIPS的区别主要在于跳转条件不同。RISCV只保留了两条无条件跳转指令`jal`和`jalr`。其次是条件跳转指令，相比MIPS繁杂的跳转条件来说，RISCV仅有6条条件跳转指令。此外，RISCV没有转移指令延迟槽，大大降低了处理器设计的难度，让处理器设计思路更加自然协调。

立即数处理

需要注意的是，RISCV立即数只做符号扩展，没有零扩展。没有意识到这一点在实验后期debug给我造成了很大困扰。

其次，有趣的是，RISCV的`j_type`和`b_type`立即数是不连续的。原因如下：

1、这么做可以拥有更大的跳转范围——立即数字段不仅仅是一个常量，而是一个函数的输入。该函数通过一系列骚操作（位扩展与循环移位）把立即数原立即数映射到一个更多的数据范围，代价则是数据表示变得稀释，不过操作系统往往是把地址分成几大段，需要更大的跳转范围，稀疏的代价是可以接受的；

2、将指令信号的扇出和立即数多路复用的成本降低了近两倍，这也简化了低端实现中的数据通路逻辑。

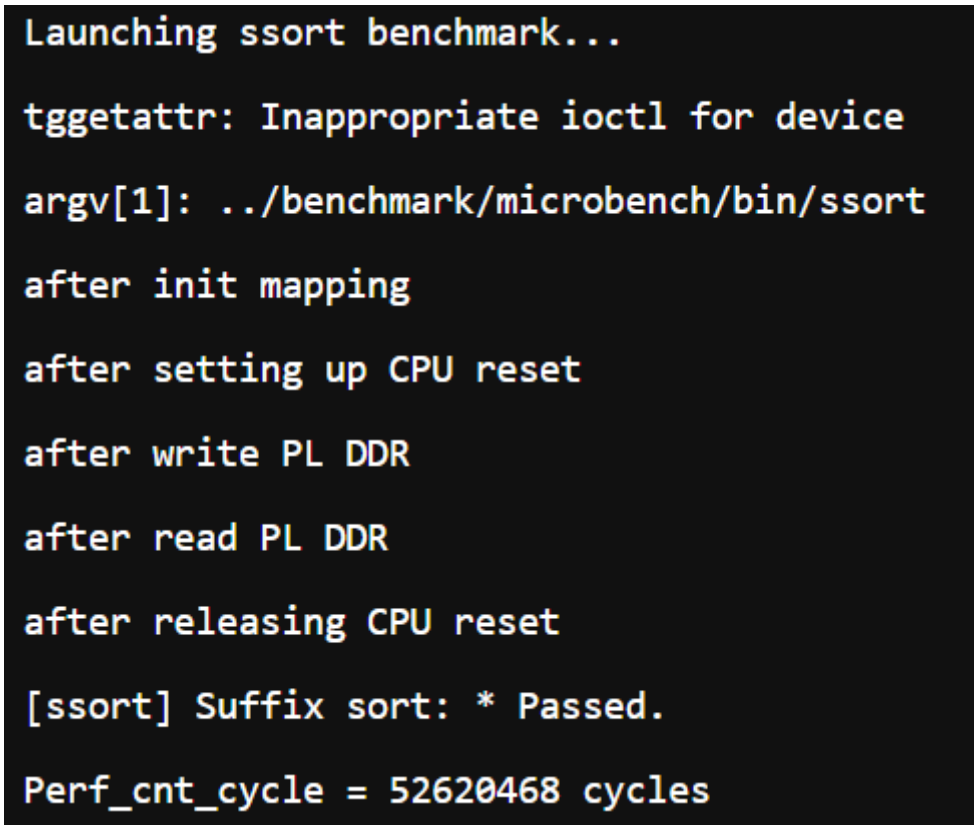
明确上述两点之后，根据指令类型来进行立即数处理，生成所有指令需要的立即数，再根据指令类型选出需要的即可。

```
assign i_type_imm = {{20{inst_reg[31]}}, inst_reg[31:20]};

assign s_type_imm = {{20{inst_reg[31]}}, inst_reg[31:25], inst_reg[11:7]};

assign b_type_imm = {{20{inst_reg[31]}}, inst_reg[7], inst_reg[30:25],
inst_reg[11:8], 1'b0};

assign u_type_imm = {{12{inst_reg[31]}}, inst_reg[31:12]};

assign j_type_imm = {{12{inst_reg[31]}}, inst_reg[19:12], inst_reg[20],
inst_reg[30:25], inst_reg[24:21], 1'b0};

assign final_imm = {32{i_type}} & i_type_imm |
                   {32{s_type}} & s_type_imm |
                   {32{b_type}} & b_type_imm |
                   {32{u_type}} & u_type_imm |
                   {32{j_type}} & j_type_imm ;
```

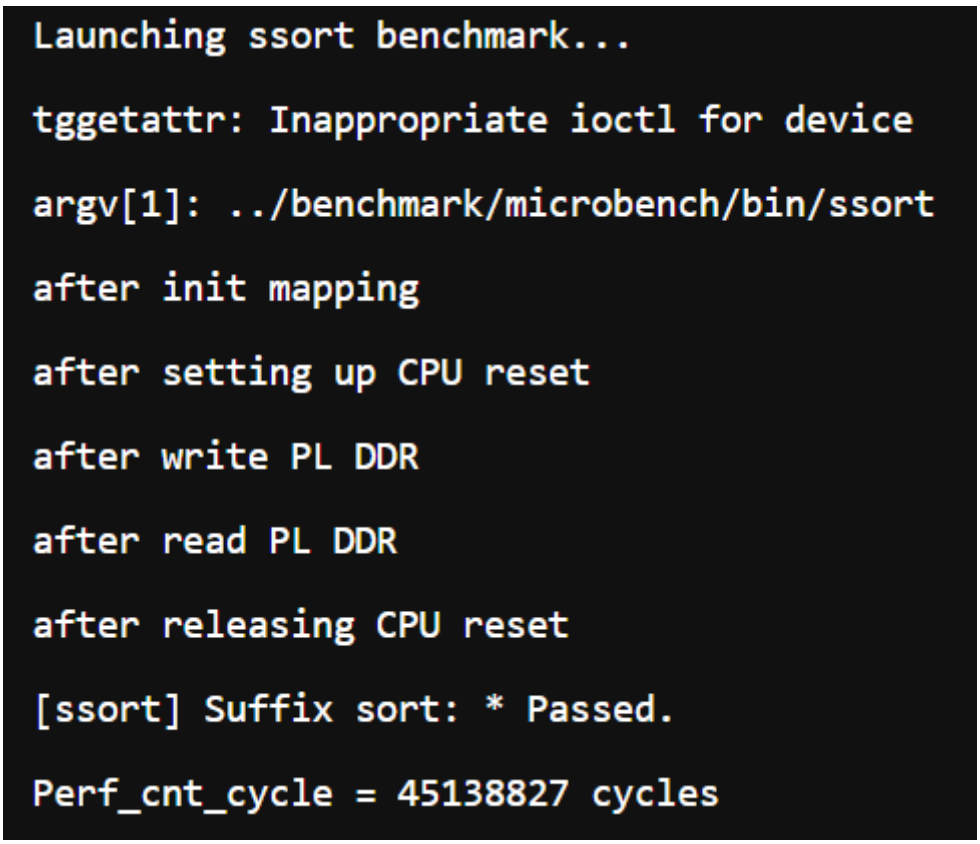
RISCV-MIPS对比

对比执行同一测试集所需的周期数（以microbench-ssort为例）。

MIPS：52620468



RISCV:45138827



对比可知，因为不需要考虑 `nop` 指令，以及RISCV比MIPS译码更加简洁等原因，RISCV比MIPS大约有14%的性能提升。

对于注释很少的解释（狡辩）

verilog的可综合部分语法主要是时序逻辑和组合逻辑两部分，其中时序逻辑的复杂度会比组合逻辑更高一些。在这些实验中，涉及到的时序逻辑代码并不复杂，并且PPT中以及给出了相关示例，因此我认为不需要注释；对于组合逻辑，说白了无非就是相关信号的连接和门电路的处理，在命名规范的条件下，要理解代码，只需要找到需要了解其功能的信号，然后查找它的生成逻辑，然后继续查找生成逻辑中的不理解的信号的生成逻辑，依此类推。在本实验中的组合逻辑都较为简单直观。此外，与软件设计语言相比，verilog（可综合语法）的可读性很强。因此不太想写注释。