中国科学院大学
《计算机体系结构》实验报告

组员 1 武弋 2022K8009929002

组员 2 王泽黎 2022K8009929011

组员 3 张家玮 2022K8009929010

实验项目编号 3 实验名称 添加用户态指令设计专题实验

一、 实验内容简介

本次实验实现了在流水线 CPU 中添加部分用户态指令的功能，其中包括算术逻辑类运算指令、乘除运算类指令、转移指令以及访存指令。本报告将结合 RTL 源码解释这些指令是如何实现的，并对实验进行的过程进行总结。

二、 RTL 源码及解释

- **添加算术逻辑类运算指令。**

本部分包括 slti, sltui, andi, ori, xori, sll, srl, sra, pcaddu12i 九个指令的实现。在添加新指令时，要搞明白指令在不同流水级中的执行过程，以及指令的控制信号和数据通路，然后在 CPU 的各个模块中添加对应的逻辑。和已经实现的指令一样，首先要参照指令集手册，对指令码译码得到这些指令的控制信号，代码如下：

```
// Arithmetic and logical operation instructions
assign inst_slti   = op_31_26_d[6'h00] & op_25_22_d[4'h8];
assign inst_sltui  = op_31_26_d[6'h00] & op_25_22_d[4'h9];
assign inst_andi   = op_31_26_d[6'h00] & op_25_22_d[4'hd];
assign inst_ori    = op_31_26_d[6'h00] & op_25_22_d[4'he];
assign inst_xori   = op_31_26_d[6'h00] & op_25_22_d[4'hf];
assign inst_sll_w  = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] & op_19_15_d[5'h0e];
assign inst_srl_w  = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] & op_19_15_d[5'h0f];
assign inst_sra_w  = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] & op_19_15_d[5'h10];
assign inst_pcaddu12i = op_31_26_d[6'h07] & ~inst_ID[25];
```

得到译码信号之后，要用这些信号来组成数据选择信号，控制数据通路，以读寄存器堆选择信号 2 为例，代码如下：

```
assign rf_using2 = inst_beq
|| inst_bne
|| inst_blt
|| inst_bge
|| inst_bltu
|| inst_bgeu // exp11: add branch instructions
|| inst_jirl // branch instructions without b/bl
// || inst_ld_w
// || inst_st_w // load/store instructions
|| inst_add_w
|| inst_sub_w
|| inst_slt
```

```
|| inst_sltu
|| inst_nor
|| inst_and
|| inst_or
|| inst_xor // ALU without imm instructions
|| mul_inst // multiplication instructions
|| div_inst // division instructions
|| inst_sll_w
|| inst_srl_w
|| inst_sra_w // shift instructions without imm instructions
// || inst_slli_w
// || inst_srli_w
// || inst_srai_w
// || inst_addi_w // imm instructions
```

其给出了需要使用寄存器堆读出数据 2 的情况，其他控制信号的组合逻辑与此类似，都是根据具体指令的实现方法，用译码信号通过或逻辑得到。

由于添加的指令需要用 alu 进行运算，所以要把这些指令添加到 alu 的控制信号 alu_op 中，代码如下：

```
assign alu_op[ 0] = inst_add_w | inst_addi_w | inst_ld_w | inst_ld_b | inst_ld_h | inst_ld_bu |
    inst_ld_hu | inst_st_w | inst_st_b | inst_st_h | inst_jirl | inst_bl | inst_pcaddu12i; //
    exp11: add branch instructions
assign alu_op[ 1] = inst_sub_w;
assign alu_op[ 2] = inst_slt | inst_slti;
assign alu_op[ 3] = inst_sltu | inst_sltui;
assign alu_op[ 4] = inst_and | inst_andi;
assign alu_op[ 5] = inst_nor;
assign alu_op[ 6] = inst_or | inst_ori;
assign alu_op[ 7] = inst_xor | inst_xori;
assign alu_op[ 8] = inst_slli_w | inst_sll_w;
assign alu_op[ 9] = inst_srli_w | inst_srl_w;
assign alu_op[10] = inst_srai_w | inst_sra_w;
assign alu_op[11] = inst_lu12i_w;
```

这样，通过控制信号将所需的三个 alu 输入信号连接到对应的数据通路上，即可实现新指令的功能。

- **添加乘除运算类指令**。

本部分添加的指令包括 mul.w, mulh.w, mulh.wu, div.w, mod.w, div.wu, mod.wu。类似地，首先也需要添加新指令的译码信号，代码如下：

```
// Multiplication and division instructions
assign inst_mul_w  = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] & op_19_15_d[5'h18];
assign inst_mulh_w = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] & op_19_15_d[5'h19];
assign inst_mulh_wu = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h1] &
    op_19_15_d[5'h1a];
assign inst_div_w  = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h2] & op_19_15_d[5'h00];
assign inst_mod_w  = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h2] & op_19_15_d[5'h01];
assign inst_div_wu = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h2] & op_19_15_d[5'h02];
assign inst_mod_wu = op_31_26_d[6'h00] & op_25_22_d[4'h0] & op_21_20_d[2'h2] & op_19_15_d[5'h03];

assign mul_inst = inst_mul_w | inst_mulh_w | inst_mulh_wu;
```

```
assign div_inst = inst_div_w | inst_mod_w | inst_div_wu | inst_mod_wu;
```

但是，alu 是不具备做乘除法运算的功能的，所以要添加乘法模块，代码如下：

```
multiplier u_multiplier(
    .mul_op    (mul_op_EX ),
    .mul_src1  (mul_src1),
    .mul_src2  (mul_src2),
    .mul_result (mul_result)
    );
```

共添加了三个模块，乘法，有符号除法和无符号除法，分别用于实现乘法和除法指令的功能。

乘法模块的输入输出方式和 alu 是非常类似的，也是通过译码信号控制数据通路，将待乘数据与乘法器 op 输入进乘法器。乘法运算的实现方式是直接调用 Xilinx 提供的乘法器 IP 核。乘法器模块代码如下：

```
module multiplier(
    input wire [ 2:0] mul_op,
    input wire [31:0] mul_src1,
    input wire [31:0] mul_src2,
    output wire [31:0] mul_result
);

wire op_mul_w; // multiply
wire op_mulh_w; // multiply high-bits
wire op_mulh_wu; // multiply high-bits unsigned

assign op_mul_w = mul_op[0];
assign op_mulh_w = mul_op[1];
assign op_mulh_wu = mul_op[2];

wire [32:0] mul_src1_ext;
wire [32:0] mul_src2_ext;

assign mul_src1_ext = {op_mulh_w & mul_src1[31], mul_src1}; //位数扩展
assign mul_src2_ext = {op_mulh_w & mul_src2[31], mul_src2}; //位数扩展

// 66-bit signed multiply
wire signed [65:0] mul_res_66;
assign mul_res_66 = $signed(mul_src1_ext) * $signed(mul_src2_ext); // 直接调用Xillinx
    IP实现乘法功能

wire [31:0] mull_result; // low 32 bits
wire [31:0] mulh_result; // high 32 bits
// wire [31:0] mulh_wu_result;
assign mull_result = mul_res_66[31: 0];
assign mulh_result = mul_res_66[63:32];

// final result mux
assign mul_result = ({32{op_mul_w       }} & mull_result) //选取低32位
                  | ({32{op_mulh_w | op_mulh_wu}} & mulh_result) //选取高32位
```

```
;

endmodule
```

除法器也通过调用 Xilinx 提供的 IP 核实现，但是出于时序考虑，不是直接使用除法符号，而是利用 Xillinx IP 核的除法模块进行运算，代码如下：

```
div_signed u_div_signed(
    .aclk               (clk          ),
    .s_axis_divisor_tdata (div_src2   ),
    .s_axis_divisor_tready (s_divisor_ready ),
    .s_axis_divisor_tvalid (s_divisor_valid ),
    .s_axis_dividend_tdata (div_src1  ),
    .s_axis_dividend_tready(s_dividend_ready),
    .s_axis_dividend_tvalid(s_dividend_valid),
    .m_axis_dout_tdata  (s_div_out    ),
    .m_axis_dout_tvalid (s_div_out_valid )
    );

div_unsigned u_div_unsigned(
    .aclk               (clk          ),
    .s_axis_divisor_tdata (div_src2   ),
    .s_axis_divisor_tready (u_divisor_ready ),
    .s_axis_divisor_tvalid (u_divisor_valid ),
    .s_axis_dividend_tdata (div_src1  ),
    .s_axis_dividend_tready(u_dividend_ready),
    .s_axis_dividend_tvalid(u_dividend_valid),
    .m_axis_dout_tdata  (u_div_out    ),
    .m_axis_dout_tvalid (u_div_out_valid )
    );
```

由以上代码可见，按照教材的提示，除法器不但需要输入数据与 op 信号，还需要输入 valid 和 ready 信号，用于握手控制除法器。因此，在 CPU 中定义了 div_executing 和 div_valid 两个 reg 信号用来形成握手信号，代码如下：

```
reg div_executing; //除法是否正在执行
always @(posedge clk) begin
    if (reset) begin
        div_executing <= 1'b0;
    end
    else if(div_inst_EX && ((s_div_in_EX && s_divisor_ready && s_dividend_ready) || (u_div_in_EX
         && u_divisor_ready && u_dividend_ready))) begin
        div_executing <= 1'b1;
    end
    else if((s_div_in_EX && s_div_out_valid) || (u_div_in_EX && u_div_out_valid)) begin
        div_executing <= 1'b0;
    end
end

reg div_valid; //是否可以进行除法
```

```
always @(posedge clk) begin
    if(reset) begin
        div_valid <= 1'b0;
    end
    else if(div_inst_EX && !div_executing) begin
        div_valid <= 1'b1;
    end
    else if(div_inst_EX && ((s_div_in_EX && s_divisor_ready && s_dividend_ready) || (u_div_in_EX
         && u_divisor_ready && u_dividend_ready))) begin
        div_valid <= 1'b0;
    end
end

assign s_divisor_valid = div_inst_EX && div_valid;
assign u_divisor_valid = div_inst_EX && div_valid;
assign s_dividend_valid = div_inst_EX && div_valid;
assign u_dividend_valid = div_inst_EX && div_valid;
    //当执行阶段的指令是除法指令，且判断除法指令可以执行时，拉高握手信号
```

- **添加转移指令**。

本部分中，需要添加的指令是 blt、bge、bltu 和 bgeu。这些指令与 beq、bne 的唯一区别在于是否跳转的判断条件不同，因此除了添加译码信号与数据通路信号外，需要添加新的转移条件判断信号，代码如下：

```
assign rj_eq_rd = (rj_value_EX == rkd_value_EX);
assign rj_l_rd = ($signed(rj_value_EX) < $signed(rkd_value_EX));
assign rj_lu_rd = (rj_value_EX < rkd_value_EX);
assign rj_geq_rd = !rj_l_rd;
assign rj_gequ_rd = !rj_lu_rd;
```

通过这些信号的组合，可以实现新的转移指令的功能。

- **添加访存指令**。

本部分中，需要添加的指令是 ld.b, ld.h, ld.bu, ld.hu, st.b, st.h。

对于 load 类指令，其余数据通路都是相同的，只需要对于 mem_result 信号根据不同的指令，选择不同位数与扩展符号的数据即可，代码如下：

```
assign mem_result = inst_ld_b_MEM ? data_sram_addr_MEM[1:0] == 2'h0 ? {{24{data_sram_rdata[
    7]}}, data_sram_rdata[ 7: 0]} :
                    data_sram_addr_MEM[1:0] == 2'h1 ? {{24{data_sram_rdata[15]}},
                        data_sram_rdata[15: 8]} :
                    data_sram_addr_MEM[1:0] == 2'h2 ? {{24{data_sram_rdata[23]}},
                        data_sram_rdata[23:16]} :
                    {{24{data_sram_rdata[31]}}, data_sram_rdata[31:24]} :
                inst_ld_h_MEM ? data_sram_addr_MEM[1:0] == 2'h0 ? {{16{data_sram_rdata[15]}},
                     data_sram_rdata[15: 0]} :
                    {{16{data_sram_rdata[31]}}, data_sram_rdata[31:16]} :
                inst_ld_bu_MEM ? data_sram_addr_MEM[1:0] == 2'h0 ? {{24'b0, data_sram_rdata[
                     7: 0]}} :
                    data_sram_addr_MEM[1:0] == 2'h1 ? {{24'b0, data_sram_rdata[15: 8]}} :
                    data_sram_addr_MEM[1:0] == 2'h2 ? {{24'b0, data_sram_rdata[23:16]}} :
                    {{24'b0, data_sram_rdata[31:24]}} :
```

```
                    inst_ld_hu_MEM ? data_sram_addr_MEM[1:0] == 2'h0 ? {{16'b0,
                        data_sram_rdata[15: 0]}} :
                       {{16'b0, data_sram_rdata[31:16]}} :
                       data_sram_rdata;
```

与 load 类指令在 data_sram 的输出端处理数据相应，store 类指令在 data_sram 的输入端处理数据，代码如下：

```
assign data_sram_we = (inst_st_b_EX ? (4'h1 << data_sram_addr[1:0]) :
                      inst_st_h_EX ? (4'h3 << data_sram_addr[1:0]) :
                      4'hf) & {4{mem_we_EX && pipe_valid[2]}};

assign data_sram_wdata = inst_st_b_EX ? {4{rkd_value_EX[ 7:0]}} :
                      inst_st_h_EX ? {2{rkd_value_EX[15:0]}} :
                      rkd_value_EX;
```

根据指令将数据进行相应的位处理，然后写入到 data_sram 中即可。

三、 调试过程中遇到的重点问题

- **完成 exp10 时遇到的问题**。

遇到的第一个问题是在将信号从 EX 阶段传递到 MEM 阶段时漏了一些新添加的指令，导致数据通路出现问题。观察波形发现一些 MEM 阶段的信号异常处于不定态，从而返回查看代码发现问题，并改正。

第二个问题是调用 Xilinx IP 核实现除法时，除法器的实例化遇到了问题。由于是第一次使用该功能，且教材中关于实例化除法器的具体方法不是十分详细，所以遇到了一些困难，除法器无法被正确例化调用。经过小组同学讨论与上网查阅资料，找到了正确的例化方法。

第三个问题是本实验中遇到的最令人印象深刻的问题，也是在添加除法指令时遇到的。在编写代码过程中，有符号和无符号除法指令没有被完全隔离开，导致两个除法器在另一个执行除法运算时也执行了除法运算，导致产生了错误的数据。这个问题的调试过程是比较困难的，因为难以通过发生错误的指令附近的波形看出除法器发生错误的原因。于是只能对除法器产生的错误数据进行追踪，最终在若干个周期前发现是冗余的除法运算覆盖了正确的数据，从而发现并解决了代码中的问题。

- **完成 exp11 时遇到的问题**。

编写代码过程中遇到了一个与数据前递相关的问题。由于 exp9 的 testbench 不包含读内存数据后立即转移的情况，在之前的实验中漏掉了相应的数据前递信号。经过研究反汇编文件，添加上了这部分内容，解决了数据前递的问题。

仿真通过后，张家玮同学的设备在综合时出现了问题，经过上网查阅资料，发现是操作系统不支持。在另一台设备上可以正常执行并生成 bit 文件。

四、 实验总结

本次实验是第一个小组合作实验，经过讨论，达成了一致同意的分工方案，每个人都完成了自己的任务。在遇到困难时，同学之间的讨论和共同探索起到了很大的作用，使得实验能够顺利完成。本次实验也是我们第一次尝试调用 Xilinx IP 核，对于 IP 核的使用方法和调用方式有了更深入的了解。在实验中，我们也发现了一些教材中没有提到的问题，通过查阅资料和讨论，解决了这些问题，对于流水线 CPU 的理解也更加深入。

五、 实验分工

exp10 代码编写：王泽黎

exp11 代码编写：张家玮

prj3 实验报告撰写：武弋

除完成各自的任务外，小组成员之间在遇到问题时相互讨论，共同解决了实验中遇到的问题。