lab5 report

实验内容

设计多周期流水线 CPU，主要实现以下内容：

- 用于流水级间传递数据的段间寄存器 `SEG_REG` 模块
- 用于冒险处理和控制流水线的 `Hazard` 模块
- 按照多周期 CPU 数据通路接线
- 实现 jal 指令控制前移到 id 段（**选做**）
- 实现 ebreak 指令的断点功能 （**选做**）

实验原理

SEG_REG

段间寄存器，接收来自某一级的数据，在时钟上升沿传递给下一级，以此实现多周期处理指令。为了实现流水线功能，还需要加入两个额外输入 `flush` 和 `stall`，分别用于冲刷流水线和停顿。

代码如下：

```
always @(posedge clk) begin
    if (flush) begin
        // 清空
        pc_cur_out <= 0;
        inst_out <= 0;
        ......
        br_type_out <= 3'b010;  // 我的不跳转编码是 010，找了半天 bug 原来在这。。。
        br_out <= 0;
        ......
        ebreak_out <= 0;
    end
    else if (!stall) begin
        // 传递
        pc_cur_out <= pc_cur_in;
        inst_out <= inst_in;
        ......
        br_type_out <= br_type_in;
        br_out <= br_in;
        ......
```

```
        ebreak_out <= ebreak_in;
    end
    else begin
        // 停顿
    end
end
```

如果接收到 flush 信号，就把输出清空；如果接收到 stall 信号，输出就不变；否则输出就等于输入

Hazard

冒险处理模块，处理指令的数据冒险或控制冒险，控制数据前递，控制流水线的冲刷或停顿。

代码如下：

```
// stall
always @(*) begin
    if ((rf_re0_ex && rf_wa_mem && (rf_ra0_ex == rf_wa_mem) && rf_wd_sel_mem == 2'b10)
    || (rf_re1_ex && rf_wa_mem && (rf_ra1_ex == rf_wa_mem) && rf_wd_sel_mem == 2'b10)) begin
        stall_if = 1'b1;
        stall_id = 1'b1;
        stall_ex = 1'b1;
        flush_mem = 1'b1;
    end
    else begin
        stall_if = 1'b0;
        stall_id = 1'b0;
        stall_ex = 1'b0;
        flush_mem = 1'b0;
    end
end
// flush
always @(*) begin
    if (ebreak_ex || pc_sel_ex == 2'b01 || pc_sel_ex == 2'b11) begin
        flush_id = 1'b1;
        flush_ex = 1'b1;
    end
    else if (pc_sel_ex == 2'b10) begin
        flush_id = 1'b1;
        flush_ex = 1'b0;
    end
    else begin
        flush_id = 1'b0;
        flush_ex = 1'b0;
```

```
    end
end
// rs1
always @(*) begin
    if (rf_re0_ex && rf_wa_mem && (rf_ra0_ex == rf_wa_mem)) begin
        rf_rd0_fe = 1'b1;
        case (rf_wd_sel_mem)
            3'b00: rf_rd0_fd = alu_ans_mem;
            3'b01: rf_rd0_fd = pc_add4_mem;
            3'b11: rf_rd0_fd = imm_mem;
            default: rf_rd0_fd = 32'h0;
        endcase
    end
    else if (rf_re0_ex && rf_wa_wb && (rf_ra0_ex == rf_wa_wb)) begin
        rf_rd0_fe = 1'b1;
        rf_rd0_fd = rf_wd_wb;
    end
    else begin
        rf_rd0_fe = 1'b0;
        rf_rd0_fd = 32'h0;
    end
end
// rs2
always @(*) begin
    if (rf_re1_ex && rf_wa_mem && (rf_ra1_ex == rf_wa_mem)) begin
        rf_rd1_fe = 1'b1;
        case (rf_wd_sel_mem)
            3'b00: rf_rd1_fd = alu_ans_mem;
            3'b01: rf_rd1_fd = pc_add4_mem;
            3'b11: rf_rd1_fd = imm_mem;
            default: rf_rd1_fd = 32'h0;
        endcase
    end
    else if (rf_re1_ex && rf_wa_wb && (rf_ra1_ex == rf_wa_wb)) begin
        rf_rd1_fe = 1'b1;
        rf_rd1_fd = rf_wd_wb;
    end
    else begin
        rf_rd1_fe = 1'b0;
        rf_rd1_fd = 32'h0;
    end
end
```

冒险主要分为以下几类处理:

- 读取-使用冒险：例如 add x1, x1, x1 指令在 EX 阶段需要使用 x1 的值，但上一个指令 lw x1, 0(x2) 刚到 MEM 阶段，还未读出内存，所以需要先停顿一周期，等 lw 指令运行到 WB 阶段，此时 rf_wd_wb 就是正确的值，但还未写入，所以需要数据前递。
- 普通数据冒险：EX 阶段要用的寄存器值刚到 MEM 段或者 WB 段，还没有写入，但是其实已经计算出来了，直接数据前递即可。（MEM 段的前递优先级更高，因为更新）
- 控制冒险：beq，jal 等指令默认不跳转，所以当判断出需要跳转时，要冲刷流水线，把放进来的错误指令冲刷掉，再跳转到正确位置继续执行。

数据通路

除了上面说的模块外，其他类似单周期，图片就不贴了，放 figs 文件夹里了

jal 控制前移

首先在 id 段加一个 ADD 模块用于计算跳转地址，代码如下：

```
ADD jal_add(
    .a(pc_cur_id),
    .b(imm_id),
    .sum(pc_jal_id)
);
```

然后更改 pc 选择器如下：

```
Mux4 #(.WIDTH(32)) pc_mux(
    .mux_sr1(pc_add4_if),
    .mux_sr2(pc_jalr_ex),
    .mux_sr3(pc_jal_id), // jal 跳转地址
    .mux_sr4(alu_ans_ex),
    .mux_ctrl(pc_sel_ex),
    .mux_out(pc_next)
);
```

注意还要更改 pc 选择信号的产生模块，代码如下：

```
encoder pc_sel_gen(
    .jal(jal_id),
    .jalr(jalr_ex),
    .br(br_ex),
    .ebreak(ebreak_ex),
    .pc_sel(pc_sel_ex)
);
......
always @(*) begin
```

```
    if (jalr)
        pc_sel = 2'b01;
    else if (br | ebreak)
        pc_sel = 2'b11;
    else if (jal)
        pc_sel = 2'b10;
    else
        pc_sel = 2'b00;
end
......
```

jalr 和 br 的优先级比 jal 更高，因为他们在 EX 段才完成判断，是在当前的 jal 指令前面的指令，应该先处理。

ebreak 断点实现

首先更改 control 模块使之能识别 ebreak 指令，添加代码如下：

```
assign ebreak = (inst[6:0] == 7'b1110011) ? 1'b1 : 1'b0; // ebreak 指令
......
always @(*) begin
    case (inst[6:0])
        ......
        7'b1110011: begin // ebreak
            rf_we = 1'b0;
            rf_re0 = 1'b0;
            rf_re1 = 1'b0;
            rf_wd_sel = 2'b00;
            mem_we = 1'b0;
            alu_src1_sel = 1'b1;
            alu_src2_sel = 1'b1;
            alu_func = 4'b0000;
            imm_type = 3'b110;
        end
        ......
    endcase
end
```

这里我们设计的 ebreak 处理流程类似 beq 等跳转指令，当在 EX 段检测到 ebreak_ex 时，冲刷流水线（不让 ebreak 之后的指令执行），但又要保证之后可以继续运行断点后的内容，所以需要在下个周期将 PC 设为 pc_cur_ex + imm(4)（ebreak 后的第一个指令），所以这里给它一个独特的立即数类型(编码 110)，生成固定的 4

要冲刷流水线，还需要更改 `Hazard` 模块，上面已给出；要设置 PC，还需要更改 `pc_sel_gen` 模块，上面也已给出。

现在我们已经能做到检测断点，阻塞断点后指令的执行。下一个问题是如何实现在断点前的指令都执行完后通知 PDU，让 PDU 接管。

我们需要将 ebreak 信号在级间传递，当 ebreak_mem 为 1 时，断点前最后一个指令已经进入 WB 阶段，所以只需将 ebreak_mem 传给 PDU，让 PDU 在下一周期进入 WAIT 状态即可。

修改 PDU 代码如下：

```
reg ebreak_edge, temp;
always @(posedge clk) begin
    temp <= ebreak;
    ebreak_edge <= ebreak & ~temp;
end
// FSM Part 1
always @(posedge clk) begin
    if (rst || ebreak_edge)
        main_current_state <= WAIT;
    else
        main_current_state <= main_next_state;
end
```

注意 PDU 对 ebreak 信号需要取上升沿，因为一旦 PDU 停止 CPU 的时钟，ebreak 就变不了了，将恒为 1，这将使 PDU 困在 WAIT 状态，无法实现断点后继续运行。

效率比较

各阶段延迟假设如下：

IF	ID	EX	MEM	WB
250ps	350ps	150ps	300ps	200ps

若执行 1000 条指令，不考虑停顿或冲刷：

对于流水线，时钟周期 350ps，所需时间为 $t1 = 350 \times 4 + 350 \times 1000 = 351400ps$；

对于单周期，时钟周期 1250ps，所需时间为 $t2 = 1250 \times 1000 = 1250000ps$

效率提升大约 4 倍

实验总结

bugs

- 接线接错，没声明（线实在太多了。。。）

- 我的不跳转编码是 010，不是 000 (br_type = inst[14:12] 偷懒了)
- 多驱动，锁存器，组合逻辑要注意
- 仿真没毛病，一上板就寄（debug 都无从下手）
- 能用 ADD 就别用 ALU

评价和建议

难度中等，无建议