LAB 5 report

实验原理

实验内容

- 理解流水线CPU的结构和工作原理
- 掌握流水线CPU的设计和调试方法，特别是流水线中数据相关和控制相关的处理

模块设计

流水线cpu是在单周期cpu的基础上改进的，很多模块都可以复用单周期的，所以这里只介绍新增或修改的模块，以及附加模块的介绍

Hazard Unit

数据冒险

1. MEM/WB 段写使能为1
2. EX 段某寄存器读使能非零；
3. 上述寄存器读地址等于 MEM/WB 段的写地址；
4. 若为 MEM 段，写回的数并非数据存储器读取结果。

当以上条件成立时，将rf_rd0_fe和rf_rd1_fe信号置1，用来选择前递的数据，同时将选择出来的前递的数据传给rf_rd0_fd和rf_rd1_fd。并且用if和else if来将MEM和WB的前递设置了一个先后关系，当MEM与 WB 同时检测到前递时，前递的数据选择的时MEM的

load—use Hazard

1. MEM/WB 段写使能为1
2. EX 段某寄存器读使能非零；
3. 上述寄存器读地址等于 MEM/WB 段的写地址；
4. 若为 MEM 段，写回的数为数据存储器读取结果。

当以上条件成立时，将if，id，ex阶段的stall信号置1，然后mem阶段的flush信号置1，这样就达成了在两个指令之间插入气泡的效果

控制冒险

当选择的pc不是pc+4时，就发生控制冒险了，这时要跳转，把if，id，ex阶段的flush置1，清刷流水线，然后从新的pc开始执行

```
module Hazard (
    input [4:0]rf_ra0_ex,//
    input [4:0]rf_ra1_ex,//
    input rf_re0_ex,//
    input rf_re1_ex,//
    input [4:0] rf_wa_mem,//
    input rf_we_mem,//
    input [1:0] rf_wd_sel_mem,//
    input [31:0] alu_ans_mem,//
```

```
    input [31:0] pc_add4_mem,//
    input [31:0] imm_mem,//
    input [4:0] rf_wa_wb,//
    input rf_we_wb,//
    input [31:0]rf_wd_wb,//
    input [1:0]pc_sel_ex,//

    output reg rf_rd0_fe,
    output reg rf_rd1_fe,
    output reg [31:0] rf_rd0_fd,
    output reg [31:0] rf_rd1_fd,
    output reg stall_if,
    output reg stall_id,
    output reg stall_ex,
    output reg flush_if,
    output reg flush_id,
    output reg flush_ex,
    output reg flush_mem
);
    initial begin
        stall_if = 0;
        stall_id = 0;
        stall_ex = 0;
        flush_if = 0;
        flush_id = 0;
        flush_ex = 0;
        flush_mem = 0;
    end

    reg [31:0] wb_hazard;

    always @(*) begin
        case(rf_wd_sel_mem)
            2'b00:  wb_hazard = alu_ans_mem;
            2'b01:  wb_hazard = pc_add4_mem;
            2'b11:  wb_hazard = imm_mem;
            default:wb_hazard = 0;
        endcase

    end


    always @(*) begin
        if(rf_we_mem == 1 && (rf_re0_ex == 1 && (rf_ra0_ex == rf_wa_mem)) &&
rf_wd_sel_mem != 2'b10) begin
            rf_rd0_fe = 1;
            rf_rd0_fd = wb_hazard;
        end
        else if(rf_we_wb == 1 && (rf_re0_ex == 1 && (rf_ra0_ex == rf_wa_wb)))
begin
            rf_rd0_fe = 1;
            rf_rd0_fd = rf_wd_wb;
        end
        else begin
            rf_rd0_fe = 0;
```

```
            rf_rd0_fd = 0;
        end
    end

    always @(*) begin
        if(rf_we_mem == 1 && (rf_re1_ex == 1 && (rf_ra1_ex == rf_wa_mem)) &&
rf_wd_sel_mem != 2'b10) begin
            rf_rd1_fe = 1;
            rf_rd1_fd = wb_hazard;
        end
        else if(rf_we_wb == 1 && (rf_re1_ex == 1 && (rf_ra1_ex == rf_wa_wb)))
begin
            rf_rd1_fe = 1;
            rf_rd1_fd = rf_wd_wb;
        end
        else begin
            rf_rd1_fe = 0;
            rf_rd1_fd = 0;
        end
    end


    //load use hazard
    always @(*) begin
        if((rf_we_mem == 1 && (rf_re0_ex == 1 && (rf_ra0_ex == rf_wa_mem)) &&
rf_wd_sel_mem == 2'b10)
        || (rf_we_mem == 1 && (rf_re1_ex == 1 && (rf_ra1_ex == rf_wa_mem)) &&
rf_wd_sel_mem == 2'b10)) begin
           flush_mem = 1;
           stall_ex = 1;
           stall_id = 1;
           stall_if = 1;
        end
        else begin
           flush_mem = 0;
           stall_ex = 0;
           stall_id = 0;
           stall_if = 0;
        end
    end


    //control hazard
    always @(*) begin
        if(pc_sel_ex != 2'b0) begin
            flush_id = 1;
            flush_ex = 1;
            flush_if = 1;
        end
        else begin
            flush_id = 0;
            flush_ex = 0;
            flush_if = 0;
        end
    end
```

```
endmodule
```

PC寄存器

增加了一个stall信号，其作用等效于(!en)，即使能信号的取反，当stall信号生效时，pc信号不更新，停驻

```
module PC(
    input [31:0]pc_next,
    input clk,
    input stall,
    input rst,
    output reg [31:0]pc_cur
);
    always @(posedge clk) begin
        if(rst)
            pc_cur <= 32'h00002ffc;
        else if(!stall)
            pc_cur <= pc_next;
        else
            ;
    end
endmodule
```

RF寄存器堆

流水线中寄存器堆需要改成写优先的，做出的修改就是在当一个寄存器写的时候，若同时该寄存器有读信号，则读出的数据是当前正在写入的数据，这样就完成了写优先

```
module RF
(
    input clk,
    input[4 : 0] ra0,
    output[31 : 0] rd0,
    input[4: 0] ra1,
    input[4: 0] ra_dbg,
    output[31 : 0] rd1,
    output[31 : 0] rd_dbg,
    input[4 : 0] wa,
    input we,
    input[31 : 0] wd
);

    reg [31 : 0] regfile[0 : 31];

    integer i;
    initial begin
        i = 0;
        while (i < 32) begin
```

```
        regfile[i] = 32'b0;
        i = i + 1;
        end
        regfile [2] = 32'h2ffc;
        regfile [3] = 32'h1800;
    end


    assign rd0 = (we && wa == ra0) ? wd : regfile[ra0];
    assign rd1 = (we && wa == ra1) ? wd : regfile[ra1];
    assign rd_dbg = (we && wa == ra_dbg) ? wd : regfile[ra_dbg];

    always @ (posedge clk) begin
        if (we && wa != 0) regfile[wa] <= wd;
    end
endmodule
```

CTRL

别的信号都跟单周期一致，所以只展示新增的rf_re的代码，还有ebreak。ebreak就是当指令为ebreak时，信号为1，其他时候信号为零。rf_we做出的修改是当写寄存器rd为x0时，信号直接设成0，其他时候就依据指令判断是否为1，这样能保证x0在数据前递的时候一直保持0，不会出错。rf_re信号也是在当前指令需要读寄存器时设成1，同样也处理了x0的情况

```
module CTRL(
    input [31:0]inst,
    output jal, //
    output jalr,//
    output reg [1:0] br_type,//
    output rf_we,
    output reg [1:0] rf_wd_sel,
    output reg alu_src1_sel,//
    output reg alu_src2_sel,//
    output reg [3:0] alu_ctrl,//
    output mem_we,//
    output rf_re0,
    output rf_re1,
    output ebreak       //new
);
    //initial begin

    //end
    assign ebreak = (inst == 32'h00100073);

    assign rf_we = (inst[11:7] == 5'b0) ? 0 : ((inst[6:0] == 7'h13) | (inst[6:0] == 7'h33) | (inst[6:0] == 7'h37) | (inst[6:0] == 7'h17)
                | (inst[6:0] == 7'h6f) | (inst[6:0] == 7'h67) | (inst[6:0] == 7'h03));    //R type, I type, lui, auipc, jal, jalr, lw

    //R type I type jalr B type lw sw
    assign rf_re0 = (inst[19:15] == 5'b0) ? 0 : ((inst[6:0] == 7'h13) | (inst[6:0] == 7'h33) | (inst[6:0] == 7'h67) | (inst[6:0] == 7'h63) | (inst[6:0] == 7'h03) | (inst[6:0] == 7'h23));
```

```
    // R type B type sw
    assign rf_re1 = (inst[24:20] == 5'b0) ? 0 : ((inst[6:0] == 7'h33) |
(inst[6:0] == 7'h63) | (inst[6:0] == 7'h23));


endmodule
```

段间寄存器

这个寄存器的行为模式跟pc寄存器类似，就是由flush和stall信号控制，flush的时候输出全为0，stall的时候就不更新

ebreak

ebreak其实很简单，在wb阶段如果指令为ebreak，则ebreak信号为一，然后把ebreak信号接到top模块中，由于要求运行到ebreak时要led4要亮，相当于调试模式，所以我就把top模块中的rst设置成sw[7]和ebreak的与。并且这个rst信号是取信号的边缘，这样当ebreak信号发生时，就相当于播动了sw7，pdu就进入了调试模式，就达成了断点的目的

```
    assign ebreak_pdu = (inst_wb == 32'h00100073) ? 1 : 0;
```

```
    wire rst_pre;
    assign rst_pre = sw[7] | ebreak_pdu;
    reg btn_r1, btn_r2;
    always @(posedge clk)
        btn_r1 <= rst_pre;
    always @(posedge clk)
        btn_r2 <= btn_r1;
    assign rst = btn_r1 & (~btn_r2);
```

估算你实现的流水线 CPU 的最小需要时钟周期与其相对单周期的指令平均所需时间的 改进（自选数据大致估算即可，无需精确计算）

假设cpu各模块延迟如下

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

大致估算，假设所有的指令都是普通五个阶段都会执行的指令

那么单周期指令平均时间就是250+350+150+300+200 = 1250ps

流水线最小需要的时钟周期就为350ps