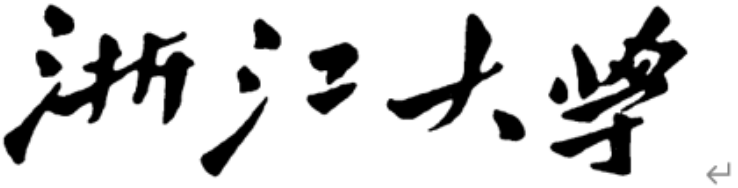


课程名称:	**计算机体系结构**
姓 名:	**杨吉祥**
学 院:	**计算机科学与技术学院**
系:	**竺可桢学院图灵班**
专 业:	**计算机科学与技术**
学 号:	**3230106222**
指导教师:	**常瑞**

一、设计思路

- load_hazard.v

由于load而产生的数据冲突与一般的数据冲突有一些不同,上学期助教说把MEM阶段要写入数据内存的值前递到EX阶段会导致性能不好,所以如果load后面跟着add之类的指令产生数据冲突时要stall一个时钟周期,然后再把WB阶段的数据前递到EX阶段,如果load后面跟着store指令产生数据冲突时,就要等到load指令在WB阶段再前递到store指令的MEM阶段,而不是在load指令的MEM阶段前递到store指令的EX阶段。所以我们将`hazard_out`设置为两位,高位为1表示是load_hazard,为0表示不是;低位为1表示是load后面跟着store,为0表示是load后面跟着非store指令。然后我的load_hazard的检测是将IF,ID的指令进行比较,当ID为load指令,且与IF阶段指令产生数据冲突,则发生load_hazard,再根据IF阶段的指令判断是哪一种load_hazard

```
`timescale 1ns/1ps
module load_hazard(
    input [2:0]load_type_ID,
    input [1:0]store_type_IF,
    input [31:0]inst_IF,
    input [31:0]inst_ID,
    output [1:0]hazard_out //hazard[1]: 0:no hazard 1:hazard hazard[0]:
1:load_store 0:load_no_store
);

reg [1:0] hazard;

assign hazard_out = hazard;

always @(*)begin
    if(load_type_ID[1:0]!=2'b0 && store_type_IF[1:0]!=2'b0
&&inst_ID[11:7]==inst_IF[24:20])begin
        hazard[1:0]=2'b11;
    end
    else if(load_type_ID[1:0]!=2'b0 && store_type_IF[1:0]==2'b0
&&inst_ID[11:7]==inst_IF[24:20])begin
        hazard[1:0]=2'b10;
    end
    else if(load_type_ID[1:0]!=2'b0 && store_type_IF[1:0]==2'b0
&&inst_ID[11:7]==inst_IF[19:15])begin
        hazard[1:0]=2'b10;
    end
    else begin
        hazard[1:0]=2'b00;
    end
end

endmodule
```

- forwarding.v

该部分用来解决数据冲突的情况,当写入寄存器尚未完成时就读该寄存器,就会发生数据冲突,由于ID阶段读寄存器,下一个EX阶段接收读出的值,所以我们要通过MEM和WB阶段来判断是否需要将数据前递给EX阶段,`inst_ID_EX[19:15]`和`inst_ID_EX[24:20]`分别表示读取的寄存器1和2,当MEM或WB阶段的rd寄存器不为0且等于rs1或rs2,并且RegWrite为1,这时候就要将要写入的数据前递,又

因为MEM阶段执行的指令是在WB阶段执行的指令之后,所以如果MEM和WB阶段的rd寄存器相同,那前递的数据应该选择MEM阶段的数据,前递的数据按照WB阶段那样根据MemtoReg控制信号进行选择就行。另外如果是load_hazard的话,需要注意load_store这种情况,因为这是在load指令在WB阶段的时候前递到store指令的MEM阶段,所以判断MEM阶段是否要前递到EX阶段的时候,需要`hazard`不等于2'b11,即load_store的情况。forwardingA为1表示rs1值需要前递,forwardingB为1表示rs2值需要前递,ALU_C表示前递的rs1值,ALU_D表示前递的rs2值

```
`timescale 1ns/1ps
module forwarding(
    input [31:0]inst_ID_EX,
    input ALUsrc_ID_EX,
    input RegWrite_EX_MEM,
    input [2:0]Memtoreg_EX_MEM,
    input [4:0]rd_EX_MEM,
    input [31:0]PC1_EX_MEM,
    input [31:0]PC2_EX_MEM,
    input [31:0]ALU_EX_MEM,
    input [31:0]imm_EX_MEM,
    input [31:0]data_change_MEM,
    input RegWrite_MEM_WB,
    input [2:0]Memtoreg_MEM_WB,
    input [4:0]rd_MEM_WB,
    input [31:0]PC1_MEM_WB,
    input [31:0]PC2_MEM_WB,
    input [31:0]ALU_MEM_WB,
    input [31:0]imm_MEM_WB,
    input [31:0]data_change_MEM_WB,
    input [1:0]hazard,
    output forwardingA,
    output forwardingB,
    output [31:0]ALU_C,
    output [31:0]ALU_D
);

reg A,B;
reg [31:0]C,D;
wire [31:0]data1,data2;

assign forwardingA = A;
assign forwardingB = B;
assign ALU_C = C;
assign ALU_D = D;

Data_select x0(
    .data_change(data_change_MEM),
    .ALU_result(ALU_EX_MEM),
    .immediate(imm_EX_MEM),
    .PC1(PC1_EX_MEM),
    .PC2(PC2_EX_MEM),
    .Memtoreg(Memtoreg_EX_MEM),
    .data_out(data1)
);

Data_select x1(
    .data_change(data_change_MEM_WB),
```

```
    .ALU_result(ALU_MEM_WB),
    .immediate(imm_MEM_WB),
    .PC1(PC1_MEM_WB),
    .PC2(PC2_MEM_WB),
    .Memtoreg(Memtoreg_MEM_WB),
    .data_out(data2)
);

always @(*) begin
    if(RegWrite_EX_MEM && rd_EX_MEM != 0 && rd_EX_MEM == inst_ID_EX[19:15] &&
hazard!=2'b11) begin
        A = 1;
        C = data1;
    end
    else if(RegWrite_MEM_WB && rd_MEM_WB != 0 && rd_MEM_WB == inst_ID_EX[19:15])
begin
        A = 1;
        C = data2;
    end
    else begin
        A = 0;
        C = 32'b0;
    end
    if(RegWrite_EX_MEM && rd_EX_MEM != 0 && rd_EX_MEM == inst_ID_EX[24:20] &&
hazard!=2'b11) begin
        B = 1;
        D = data1;
    end
    else if(RegWrite_MEM_WB && rd_MEM_WB != 0 && rd_MEM_WB == inst_ID_EX[24:20])
begin
        B = 1;
        D = data2;
    end
    else begin
        B = 0;
        D = 32'b0;
    end
end

endmodule
```

- stall的判断

  我的跳转指令是通过stall两个时钟周期来实现的,然后我通过判断ID,EX阶段是否是跳转指令来决定是否要stall,另外如果是load后面跟着add之类的指令产生数据冲突时也要stall一个时钟周期。根据上面这些实现了stall的判断

  ```
  assign stall=(jump_ID==2'b10)||(jump_ID==2'b01)||(branch_ID[1]==1'b1)||
  (jump_out_ID_EX==2'b10)||(jump_out_ID_EX==2'b01)||
  (branch_out_ID_EX[1]==1'b1)||(hazard_ID==2'b10);
  assign stall_IF=stall;
  assign stall_IF_ID=stall;
  ```

- stall的实现

我的stall是通过IF阶段和IF_ID寄存器实现的,,stall_IF为1表示正在执行跳转指令,由于此时的PC_IF已经不是正在执行的跳转指令的地址,PC_IF_ID才是该跳转指令的地址,所以根据stall_IF去给PC1正确赋值,IF_ID寄存器比其他中间寄存器多了一个stall信号,表示正在执行跳转指令,那这时候就不能执行跳转指令接下来的指令,所以当stall为1时,该寄存器输出的指令为32'h13,表示的是add x0,x0,x0,相当于nop

```
assign PC1=(stall_IF==0)?PC_in_IF:PC_IF_ID;
```

```
always @(posedge clk_IF_ID or posedge rst_IF_ID) begin
    if(rst_IF_ID) begin
        PC_out <= 0;
        inst_out <= 0;
    end
    else begin
        if(stall_IF_ID) begin
            PC_out <= PC_out;
            inst_out <= 32'h00000013;
        end
        else begin
            PC_out <= PC_in_IF_ID;
            inst_out <= inst_in_IF_ID;
        end
    end
end
```

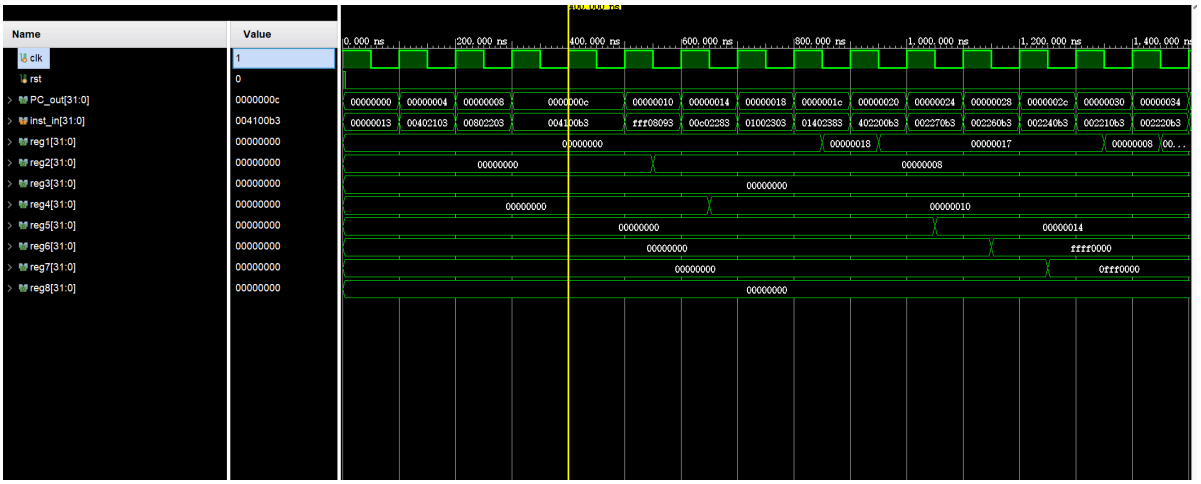
二、思考题

- 可以看到`add x1, x2, x4`和`addi x1, x1, -1`发生数据冲突,按照正常情况下需要stall两个时钟周期,但是我们可以看到仿真图中400ns,PC=0XC执行了`add x1, x2, x4`(之前300ns时的PC也为0XC,但是由于load指令与它产生了数据冲突,需要stall一个时钟周期,所以400ns时才执行`add x1, x2, x4`),500ns,PC=0X10执行了`addi x1, x1, -1`,可以发现950ns时,x1的值被写入0X17,正是在执行`addi x1, x1, -1`指令后经过四个时钟周期,在第五个时钟周期的下降沿,也就是WB阶段的下降沿写入寄存器,由于forwarding机制中间并没有发生stall

```
lw x4, 8(x0) # PC = 0x8, x4 = 0x00000010

add x1, x2, x4 # PC = 0xC, x1 = 0x00000018

addi x1, x1, -1 # PC = 0x10, x1 = 0x00000017
```



- # 1. 比较器在 ID 阶段的优劣

优势

1. **减少分支延迟**
   - 对于条件分支指令（如 `beq`，`bne`），比较操作在 ID 阶段完成，可以立即判断分支是否跳转，无需等待 EX 阶段的结果。
   - 分支决策提前，**减少分支预测错误的惩罚**（从 2 周期惩罚降低到 1 周期），提升流水线效率。

劣势

1. **增加 ID 阶段关键路径时延**
   - ID 阶段需同时完成指令译码、寄存器读取和比较操作，可能导致 **ID 阶段的逻辑复杂度增加**，延长该阶段的时延。
   - 若 ID 阶段成为关键路径，处理器的 **最大时钟频率可能降低**，整体性能受限。

2. 比较器在 EX 阶段的优劣

优势

1. **平衡流水线时延**
   - EX 阶段通常设计用于处理复杂计算（如 ALU 操作），比较操作与时延较长的运算（如乘法、移位）并行执行，不会显著增加关键路径时延。
   - **保持各阶段时延均衡**，有利于提高处理器时钟频率。
2. **简化 ID 阶段设计**
   - ID 阶段仅需完成译码和寄存器读取，逻辑更简单，**降低设计复杂度。**

劣势

1. **增加分支延迟**
   - 分支指令需等待 EX 阶段完成比较操作后才能决策，导致 **分支预测错误惩罚增加**（通常为 2 周期）。
   - 流水线需插入气泡等待比较结果，降低效率。

三、感想

第一个实验与上学期的计组实验高度重合,所以还是相对比较简单的,报告内容要求也变低了,这学期继续加油!