浙江大学

《计算机体系结构》
实验报告

实验指导：	Lab1
姓　　名：	王晓宇
学　　号：	3220104364
电子邮箱：	3220104364@zju.edu.cn
联系电话：	19550222634
授课教师：	姜晓红
助　　教：	简英钊&黄鸿宇

2024 年 9 月 23 日

Lab1 Pipelined CPU supporting RISC-V RV32I Instructions

Lab1 Pipelined CPU supporting RISC-V RV32I Instructions

课程名称： 计算机组成与设计 实验类型： 综合

实验项目名称： Lab1：Pipelined CPU supporting RISC-V RV32I Instructions

学生姓名： 王晓宇 学号： 3220104364 同组学生姓名： 无

实验地点： 玉泉曹西301室 实验日期： 2024 年 9 月 23 日

1 实验目的和要求

- （重新）学会使用Vivado进行硬件设计开发
- 理解 RISC-V RV32I 指令
- 掌握执行 RV32I 指令的流水线 CPU 设计方法
- 掌握流水线Forwarding和Bypass的方法
- 掌握Predict-not-Taken的Stall方法
- 掌握在CPU上执行RISCV程序的方法

2 实验内容和原理

> Tips: 请解释predict not taken的实现思路和3个forward的实现思路，并简要解释顶层RV32core的连线

2.1 Predict Not Taken 实现思路

Predict Not Taken 是一种静态分支预测策略，假设所有的分支指令都不会跳转。具体实现思路如下：

1. 在取指令阶段（IF），假设所有的分支指令都不会跳转，继续顺序执行下一条指令。
2. 在指令译码阶段（ID），检测当前指令是否为跳转指令，计算分支条件是否满足，通过`branch`信号标识。

3. 如果在IF执行阶段发现分支条件满足（即需要跳转），则需要刷新清除流水线中错误取指的指令（Flush），并更新程序计数器（PC）为正确的跳转地址。

2.2 Forwarding 实现思路

2.2.1 Load-Use Hazard Forwarding：

- 这种情况发生在一条指令依赖于前一条加载指令的结果。例如，`load` 指令之后紧接着有一个需要使用该数据的指令（如加法）。
- 当 `load` 指令在内存阶段（MEM）读取数据时，转发机制不可以将这个数据直接传递到后续的使用该数据的指令的执行阶段（EX），相当于时间进行了回退，此时需要插入停顿配合前递来解决load-use问题

2.2.2 ALU-Use Hazard Forwarding：

- 这种情况发生在一条指令需要使用上一条 `ALU` 指令的结果。例如，一条加法指令之后有一条乘法指令需要使用加法的结果。
- 在这种情况下，可以将执行阶段（EX）计算出的结果直接转发给需要该结果的指令的执行阶段。这种转发通常通过检测到前一条指令在执行阶段的输出，并将其传递给后续指令。

2.2.3 Load-store Forwarding：

- **LOAD指令**（例如`lw x1, 0(x2)`）将数据从内存加载到寄存器。
- **STORE指令**（例如`sw x1, 0(x3)`）需要在执行时使用寄存器中的数据。

如果`STORE`指令紧接在`LOAD`指令之后，且`STORE`依赖于`LOAD`所更新的寄存器，这会导致数据不一致。

- 在`LOAD`指令的MEM阶段，检测到数据竞争后，将`foward_ctrl_ls`置1表示数据来源为load指令读取值，接着将数据从流水线寄存器前递到单元`Dataout_EXE`即可。

2.3 顶层 RV32core.v的连线解释

1. 流水线寄存器

流水线寄存器用于在各个流水线阶段之间传递数据和控制信号。每个阶段结束时，数据和控制信号会被存储在流水线寄存器中，并在下一个时钟周期传递到下一个阶段。

- **IF/ID寄存器：**存储取指阶段（IF）到指令解码阶段（ID）的数据和控制信号。
- **ID/EX寄存器：**存储指令解码阶段（ID）到执行阶段（EX）的数据和控制信号。
- **EX/MEM寄存器：**存储执行阶段（EX）到存储访问阶段（MEM）的数据和控制信号。
- **MEM/WB寄存器：**存储存储访问阶段（MEM）到写回阶段（WB）的数据和控制信号。

2. 算术逻辑单元（ALU）

ALU用于执行算术和逻辑运算。它接收来自寄存器或立即数的数据，并根据控制信号执行相应的运算，如加法、减法、按位与、按位或等。

3. 只读存储器（ROM）

ROM用于存储指令。在取指阶段（IF），程序计数器（PC）指向的地址会被用来从ROM中读取指令，并将其传递到指令解码阶段（ID）。

4. 随机存取存储器（RAM）

RAM用于存储数据。在存储访问阶段（MEM），ALU计算出的地址会被用来从RAM中读取或写入数据。读取的数据会在写回阶段（WB）写回到寄存器文件中。

5. 控制信号产生单元

控制单元根据当前指令生成各种控制信号，这些信号用于控制各个模块的操作，如ALU操作类型、寄存器写使能、存储器读写使能等。控制单元通常在指令解码阶段（ID）生成控制信号。

6. 冲突检测单元

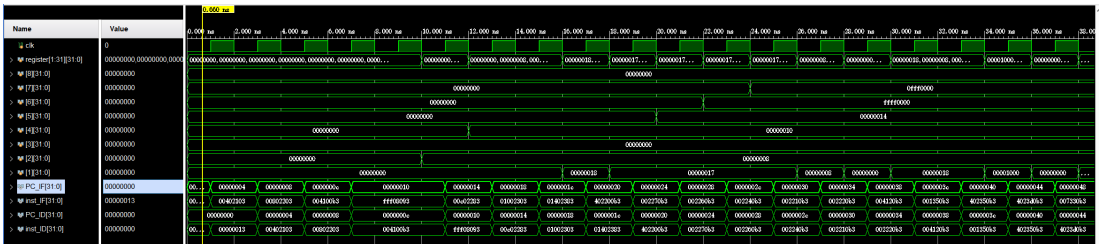
冲突检测单元用于检测数据冒险和控制冒险，并生成相应的控制信号以解决这些冒险。数据冒险发生在指令之间存在数据依赖时，控制冒险发生在分支指令时。冲突检测单元通过插入气泡（停顿流水线）或数据前递（Forwarding）来解决这些问题。

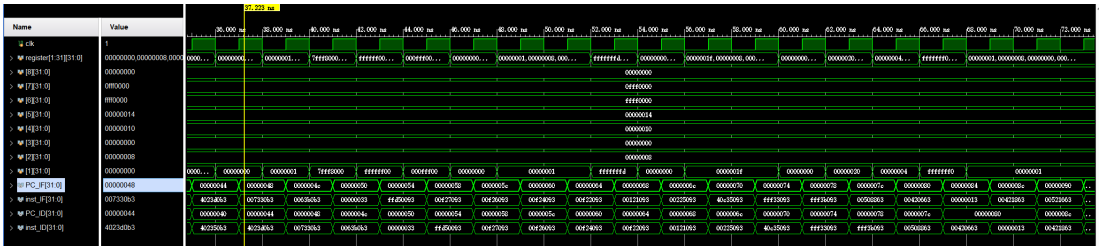
3　实验过程和数据记录及结果分析

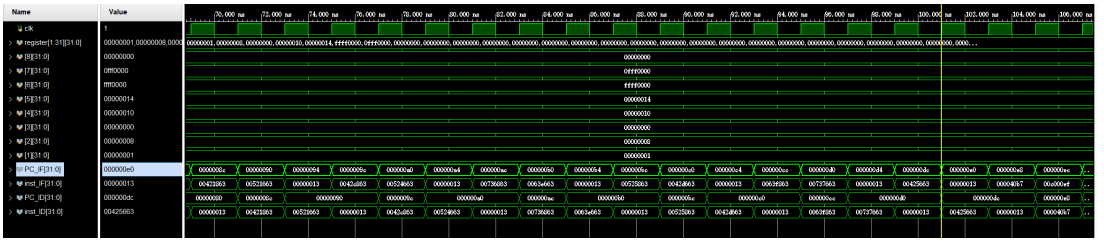
> Tips：请给出本次实验仿真的完整截图与各种forward和predict not taken发生时的截图，并简要解释

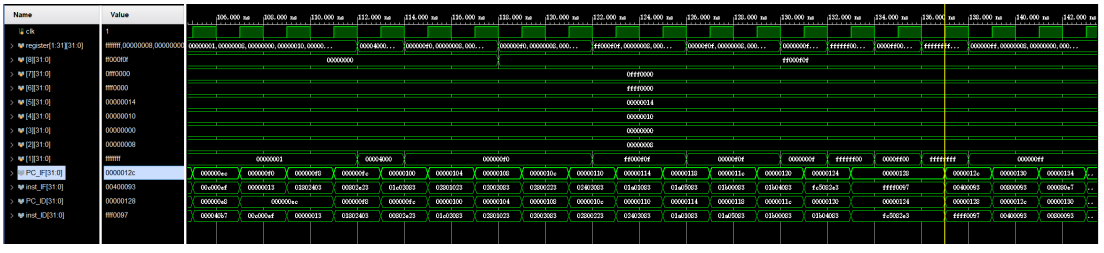
3.1　完整截图

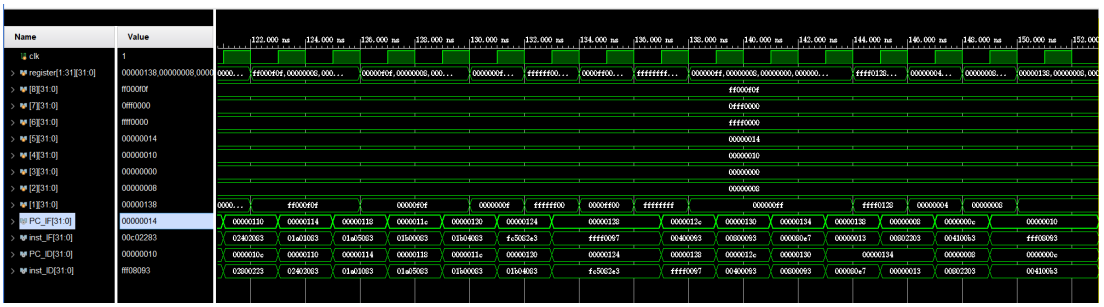
> 给出寄存器和部分PC指令的显示







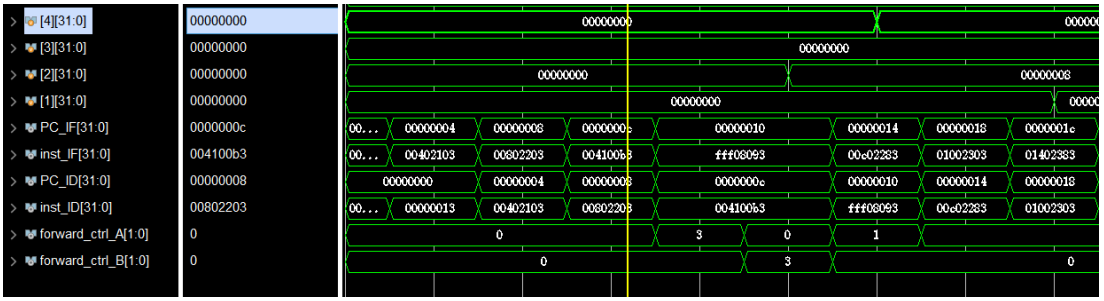




3.2 三种forward仿真截图

3.2.1 load-use前递

```
1 lw x2 4(x0)  ##pc0x4
2 lw x4 8(x0)  ##pc0x8
3 add x1 x2 x4 ##pc0xc
```



当取指到add指令（pc = 0xc）时，lw指令的结果尚未写回（在add的ID阶段需要使用x2和x4的值），因此需要插入stall以避免使用未更新的数据。因此在ID阶段产生了stall。

在 stall 后，使用数据前递（forwarding）机制：

```
MUX4T1_32 mux_forward_A(.I0(rs1_data_reg),.I1(ALUout_EXE),.I2(ALUout_MEM),.I3(Datain_MEM),
    .s(forward_ctrl_A),.o(rs1_data_ID));//TO_BE_FILLED

MUX4T1_32 mux_forward_B(.I0(rs2_data_reg),.I1(ALUout_EXE),.I2(ALUout_MEM),.I3(Datain_MEM),
    .s(forward_ctrl_B),.o(rs2_data_ID));//参考forward_ctrl_A和forward_ctrl_B的定义来填写
```

结合上述选择器可知，forward_ctrl_A为0,forward_ctrl_B为3，表明ALU计算时，x2使用已经写入寄存器的值，x4使用刚刚从MEM读出来的前递值

- 从 lw x2 的WB阶段将数据前递到 add 的 EX 阶段。
- 同样，从 lw x4 的 MEM 阶段将数据前递到 add 的 EX 阶段。

3.2.2 ALU-Use数据前递

```
1 slt x1 x2 x4  ##pc0x38
2 srl x1 x6 x1  ##pc0x3c
```

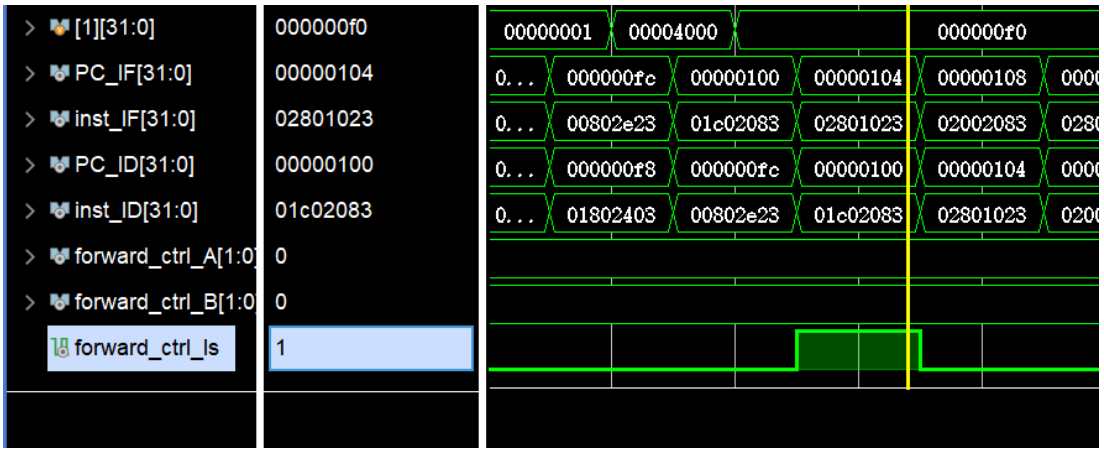
查看指令可知，上述两条指令存在数据冒险，我们在第二条指令的ALU单元运算时要更改reg2的来源

```
MUX4T1_32 mux_forward_B(.I0(rs2_data_reg),.I1(ALUout_EXE),.I2(ALUout_MEM),.I3(Datain_MEM),
    .s(forward_ctrl_B),.o(rs2_data_ID));//参考forward_ctrl_A和forward_ctrl_B的定义来填写
```

从信号来看，当冲突检测单元检查到这类冲突时，控制信号`forward_ctrl_A`为1，表示该算子来源是上一个ALU运算结果`ALUout_EXE`，从而使用前递来正确运算。

3.2.3 Load-store数据前递

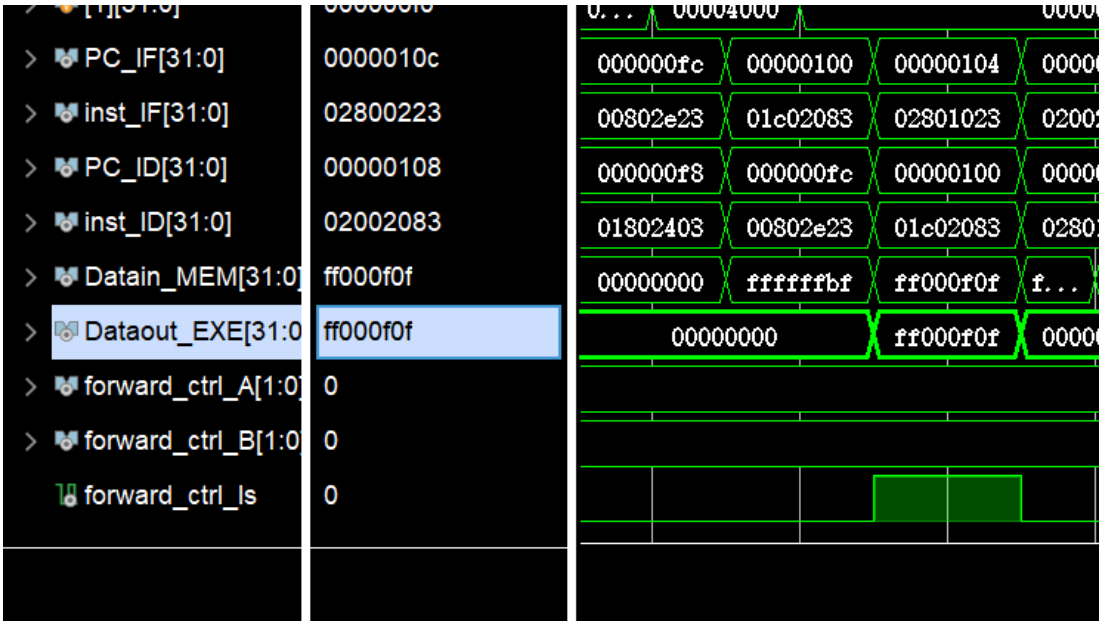
```
1  lw x8 24(x0)   ##pc0xf8
2  sw x8 28(x0)   ##pc0xfc
```



load-store冒险发生在`STORE`指令紧接在`LOAD`指令之后，且`STORE`依赖于`LOAD`所更新的寄存器，我们使用数据前递解决。

在上述汇编码中，都用到了x8寄存器，我们在发现此类冲突时，冲突检测单元将`forward_ctrl_ls`置1，表示此存储指令的数据来源寄存器是load指令正要准备更新的寄存器值，此时将数据前递到`Dataout_EXE`，以下是选择器对`Dataout_EXE`的筛选单元。

```
MUX2T1_32 mux_forward_EXE(.I0(rs2_data_EXE),.I1(Datain_MEM),.s(forward_ctrl_ls),.o(Dataout_EXE));//
```



可以看到store的寄存器内容被更新为最新，冲突解决。

3.3 predict not taken仿真截图

```
beq x1 x5 16  #0x7c
beq x4 x4 12  #0x80
```

以下是跳转指令的产生来源：

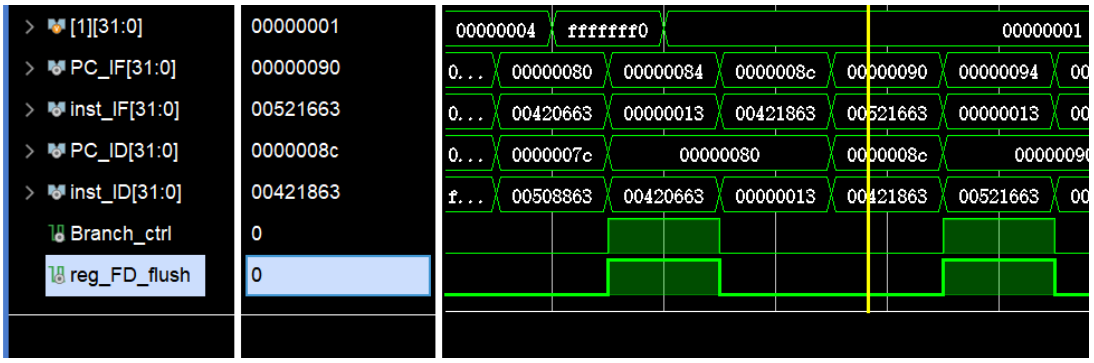
```
assign Branch = JALR | (B_valid & cmp_res) | JAL ;/
```

```
assign reg_FD_flush = Branch_ID;//TO_BE_FILLED;//什么时候IF到ID的寄存器需要被清空
```

```
/reg[31:0]PCurrent_ID,IR_ID;
   always @(posedge clk or posedge rst) begin
       if(rst) begin
            IR_ID <= 32'h00000000;                                  //复位清零
            PCurrent_ID <= 32'h00000000;                            //复位清零
           end
       else if(EN)begin
               if(Data_stall)begin
                   IR_ID <= IR_ID;                                  //IR waiting
                   PCurrent_ID <= PCurrent_ID;  end                 //保存对应PC
               else if(flush)begin
                       IR_ID <= 32'h00000013;                       //IR waiting
                       PCurrent_ID <= PCurrent_ID;  end             //清除指令的指
               else begin
                       IR_ID <= IR;                                 //正常取指,传送
                       PCurrent_ID <= PCOUT;  end                   //当前取指PC地
           end
           else begin
               IR_ID <= IR_ID;
               PCurrent_ID <= PCurrent_ID;
           end
   end

ndmodule
```

可以看到flush信号决定了取指的内容变为nop



在上述两条汇编码中，均为跳转指令，已知x1、x5不相等，所以在第一条指令后并不会跳转，根据预测分支跳转总不发生的方案，下一条指令依旧会正常读入，即PC_IF依然会读到0x80的汇编码；

而在第二条跳转指令读入时，在ID阶段检测到该指令会执行跳转命令，而根据预测分支跳转总不发生的方案下一条指令已经读入，我们此时要对读入的指令做flush，将0x8c的指令变为00000013，只是由于其原本的指令本身就为00000013，所以这里看不出变化，之后取指地址跳转到0x8c，跳过了0x88

4 讨论与心得

> Tips: 请写出对本次实验内容的深入讨论或者本次实验的心得体会。

本次实验重温了五级流水线的流程，完成了控制单元和冲突检测单元的填空，算是对计组的复习了，没有计组从零开始的繁琐，填空的好处是减轻任务，填写要认真理解上下文即可。

其实可以改进的我觉得是那段验收代码，其实可以写出目标寄存器的期望值，这样我们可以更好地去检查仿真是否正确，当然这也确实防止了面向结果编程（