浙江大学

《计算机体系结构》
实验报告

实验题目	：	Lab5 Pipelined CPU multi-cycle operations
姓　　名	：	王晓宇&李安旭
学　　号	：	3220104364&3220102479
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 12 月 5 日

Lab5 Pipelined CPU multi-cycle operations (Out of Order Version)

1 实验目的和要求

2 实验内容和原理

3 实验过程和数据记录及结果分析

4 讨论与心得

Lab5 Pipelined CPU multi-cycle operations (Out of Order Version)

课程名称：　计算机体系结构　　实验类型：　综合

实验项目名称：　Lab5: Pipelined CPU multi-cycle operations (Out of Order Version)

学生姓名：　王晓宇&李安旭　　学号：　3220104364 & 3220102479

实验地点：　玉泉曹西301室　　实验日期：　2024 年 12 月 5 日

1　实验目的和要求

- 设计一款支持乱序执行的多周期流水线CPU。
- 将CPU设计成IF/ID/FU/WB四阶段，且FU阶段支持多周期。
- 支持指令乱序完成，并检测和解决CPU执行过程的冒险情况。

2　实验内容和原理

> Tips：结合代码实现，简要解释：每种冒险的检测和解决方式，ID阶段的等待机制实现，Branch的处理逻辑和Prediction-not-Taken的实现，以及*为什么在本次设计的流水线中没有包含WAR冒险*。

- **WAR冒险：**

  没有WAR冒险是因为在本流水线里面读没有延时，指令达到`ID`阶段后半个周期直接就可以读出数据，不存在前面的指令还未读出数据，后面的指令先写入寄存器的情况。在本流水线中，最快的写入也需要隔一个周期。

- **WAW冒险：**

```
wire WAW = rd != 0 && ( ((rd == FU_write_to[1]) && !(
                                                    reservation_reg[0]==4&&0<FU_delay_cycles[use_FU]
                        ))||
                        ((rd == FU_write_to[2]) && !(
                                                    (reservation_reg[0]==2&&0<FU_delay_cycles[use_FU])||
                                                    (reservation_reg[1]==2&&1<FU_delay_cycles[use_FU])
                        ))||
                        ((rd == FU_write_to[3]) && !(
                                                    (reservation_reg[0]==3&&0<FU_delay_cycles[use_FU])||
                                                    (reservation_reg[1]==3&&1<FU_delay_cycles[use_FU])||
                                                    (reservation_reg[2]==3&&2<FU_delay_cycles[use_FU])||
                                                    (reservation_reg[3]==3&&3<FU_delay_cycles[use_FU])||
                                                    (reservation_reg[4]==3&&4<FU_delay_cycles[use_FU])||
                                                    (reservation_reg[5]==3&&5<FU_delay_cycles[use_FU])||
                                                    (reservation_reg[6]==3&&6<FU_delay_cycles[use_FU])
```

代码部分实现如上，主要思路是先判定该条指令需要写回（`Rd ≠ 0`），然后用该条指令的`Rd`去和当前所有FU部件的`Rd`作比较，如果相同说明之前的指令里面有写回同一个寄存器的。接下去去检测前面的指令能否在本指令写入前完成写入，即预约站上的位置是否更靠前，这里我们采用遍历的方法去检查，因为我们并无法直接获取到某个FU部件离写入还有几个周期。如果在本指令写回的前面没有检测到，说明肯定在本指令的后面，也就是说发生了WAW冒险。

- **RAW冒险：**

```
wire RAW_rs1 = rs1!=0 && (  rs1 == FU_write_to[1]||
                            rs1 == FU_write_to[2]||
                            rs1 == FU_write_to[3]||
                            rs1 == FU_write_to[4]||
                            rs1 == FU_write_to[5]);//TO_BE_FILLED;
```

代码实现如上，主要思路是看本条指令是否需要源寄存器，如果需要源寄存器的话再去检测目前的FU部件是否有写入，如果有重复则说明发生了RAW冒险

- **结构冒险：**

```
wire WB_structure_hazard = reservation_reg[FU_delay_cycles[use_FU]] != 0;//TO_BE_FILLED;
wire FU_structure_hazard = FU_status[use_FU] == 1 && (reservation_reg[0]!=use_FU);//TO_BE_FILLED;
```

代码实现如上，结构冒险分为写回结构冒险和FU部件冒险，写回结构冒险需要检测同一个时钟周期是否有寄存器需要写回，FU部件冒险检测本指令到达时所需的FU部件是否处于空闲状态，值得注意的是，在写回的周期`FU_status`的值还未改变但是实际上FU部件已经被视为空闲，所以要另加判断

- **ID阶段的等待机制：**

```
assign reg_IF_en = ~FU_hazard | branch_ctrl;

assign reg_ID_en = reg_IF_en;

assign branch_ctrl = (B_in_FU & cmp_res_FU) |  J_in_FU;
```

```
always @(posedge clk or posedge rst) begin
    if(rst) begin
        IR_ID <= 32'h00000013;
        PCurrent_ID <= 32'h00000000;
        valid <= 0;
    end
    else if(flush) begin
        PCurrent_ID <= PCOUT;                    //
        IR_ID <= 32'h00000013;                 //IR waiti
        valid <= 0;
    end
    else if(EN)begin
        IR_ID <= IR;                          //正常取指,
        PCurrent_ID <= PCOUT;                 //当前取指PC
        valid <= 1;
    end
end
```

我们使用`en`信号来实现等待机制，可以看到，任意的冒险都会触发等待机制，此时`en`被置低位，使得`Reg_ID`和`Reg_IF`不进行任何操作保持原样

- `Branch`的处理逻辑与`Prediction-not-taken`的实现：

  `Branch`的处理逻辑较为复杂，首先是在**always**块里面对`B_in_FU`进行赋值，这样可以使下一个时钟周期的`B_in_FU`置为这个时钟周期的`B_valid`。这样做的目的是和**FU**的结果同步，因为在本周期，`Regs`会输出对应的应该比较的两个源寄存器值，`FU_jump`在下一个周期将这些值赋给自己内部的寄存器（出于延时的考虑）并且比较的结果直接送回给`Ctrl_unit`。所以如果要跳转的话，在下一个周期，`B_in_FU`会置高位，比较的结果也会送过来，`branch_ctrl`会置高位，`ID_EN`和`IF_EN`置低位，`ID_FLUSH`置高位，同时使得**FU**部件的**EN**置低位，这样就清空了错误的指令。再下一个周期，用`ID_FLUSH_NEXT`这个信号保持预约站的移动，同时由于`flush`过了所以执行的是**nop**指令。至于`Predicition-not-taken`，其实也在上面一大串里面，就是先假定不执行跳转，等比较结果出来再决定是否`flush`，这就是预测不执行。

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关键信号截图，并结合波形简要解释每种冒险的检测和解决，包括Prediction-not-Taken机制的效果。

- `FU_Unit_Harzard`：

  见示例1，出现了该冒险，原因是此时`FU_MEM`部件正在使用，所以无法把这条指令放进预约站里，此时`stall`一个周期。

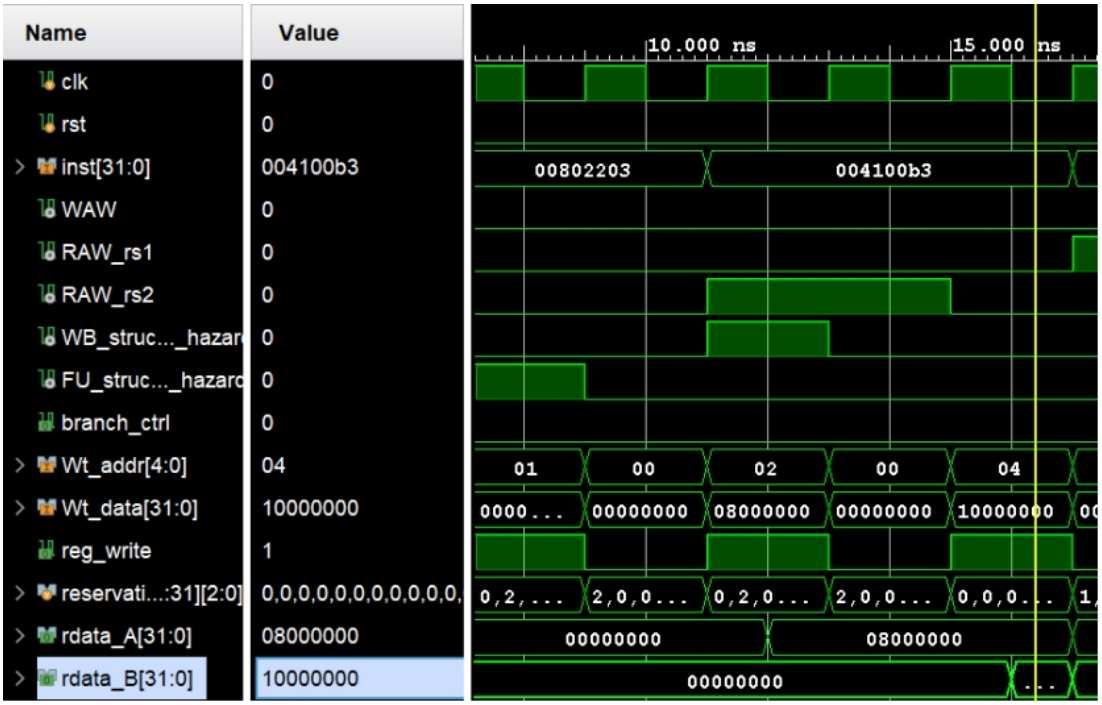
- `WB_Harzard`：

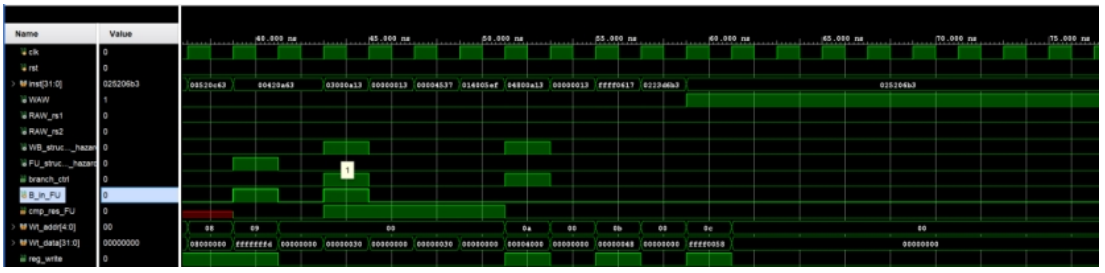
  见示例2，此时执行add指令，也就是说需要在`reserve[1]`的位置写入， 但是可以看到由于我们的代码是写入后立马移动一个周期，所以上一条lw指令此时占据了`reserve[1]`的位置，这样就出现`WB_Harzard`， `stall`一个周期。

- `RAW`：

  同样可见示例2，此时的`add x1, x2, x4`指令与前两个`lw`指令都构成了`RAW`冲突，所以停了两个周期，可以看到第三个周期时预约站清空了，说明在本周期中的下降沿， `Regs`就会将数据写入目标寄存器，那么本指令就可以读到正确的寄存器值，如下图所示。此时指令可以写入预约站。

另外，本实验给的模板中的`riscv.txt`指出第五条指令中有`FU_Unit_Harzard`，但是实际上如示例3所示，没有FU部件冲突。原因是此时上一条add指令已经位于`reserve[0]`的位置了，按照我们的代码，确实不应该有冲突，下一个周期它就被移出预约站了，刚好可以让这一条add指令写进预约站。

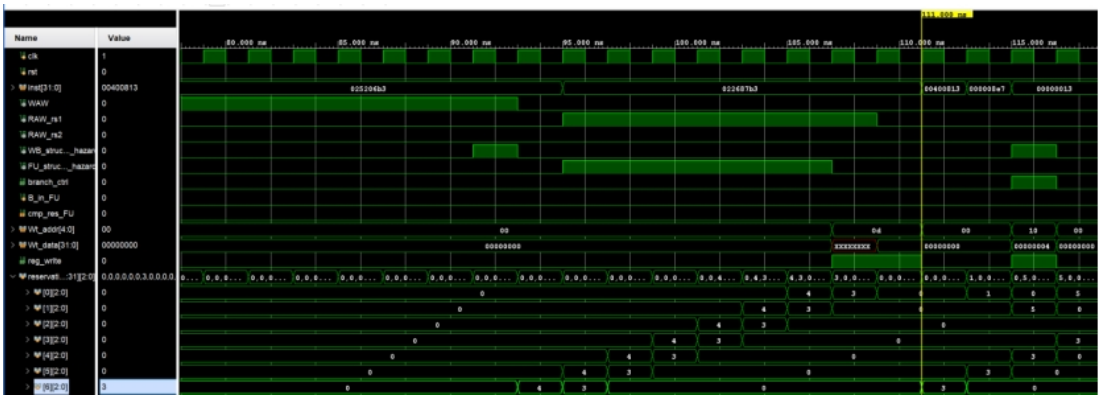




关于`branch`指令的详细处理逻辑我已经在上一小节详细分析过了，这里就不加赘述。这里用4、5两个示例说明一下不跳转和跳转的波形。示例4中，不跳转，在下一个周期`B_in_FU`置为高位，比较的结果变成有效，但是是0，所以不会发生跳转。然后我们看示例5，这条指令会跳转，但是一开始因为FU部件冲突所以`B_in_FU`并不会被赋值。详细原因见如下代码，会先进入前面的`elif`分支。这就保证了`branch_ctrl`的赋值有效，必然发生在指令结束后下一个周期。然后接着看示例5，下面两个周期的指令实际上是都被无效了的，这就是`Prediction-not-taken`。

关于Jal指令的跳转部分与branch一样，不加赘述，我们关注一下写入部分，如示例6所示，两个周期后写入，符合假设

```
else if (FU_hazard | reg_ID_flush) begin
    for (i = 0; i < 31; i=i+1)begin
        reservation_reg[i] <= reservation_reg[i+1];
    end
    reservation_reg[31] <= 0;
    //TO_BE_FILLED <= 0; //这里需要编写多行代码，完成reserva
    B_in_FU <= 0;
    J_in_FU <= 0;
    end
else if(valid_ID) begin  // regist FU operation
    reservation_reg[FU_delay_cycles[use_FU]] = use_FU;//mu
    for (i = 0; i < 31; i=i+1)begin
```



- WAR：

  如示例**7**所示，这条指令是**mul**指令，前面有一条**divu**指令，所以它必须等待到**div**在预约站中的位置移动到它前面才可以写入预约站，符合代码逻辑

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。

本次实验的逻辑与之前的单周期流水线差异较大，所以本人花了较多时间在理解多周期流水线的跳转逻辑上。其中很多地方是上升沿触发的赋值给我造成了较大的干扰，我对于上升沿触发的理解是相当于在下一个周期的一开始将右值在当前周期的值赋给左值，在阅读代码时如果不考虑到这一点会造成一些关于时钟周期的困扰。当然这是将**verilog**作为一个编程语言来解读，事实上应该把它作为硬件来解读（对我来说有点太抽象了）。