中国科学院大学
《计算机组成原理(研讨课)》实验报告

姓名 宋俊仪 学号 2023K8009929018 专业 计算机科学与技术

实验项目编号 5.1 实验名称 流水线处理器设计与实现

注 1:	撰写此 Word 格式实验报告后以 PDF 格式保存 SERVE CloudIDE 的 `/home/serve-ide/cod-lab/reports` 目录下(注意: reports 全部小写)。文件命名规则: `prjN.pdf`, 其中 `prj` 和后缀名 `pdf` 为小写, `N` 为 1 至 4 的阿拉伯数字。例如: `prj1.pdf`。PDF 文件大小应控制在 5MB 以内。此外, 实验项目 5 包含多个选做内容, 每个选做实验应提交各自的实验报告文件, 文件命名规则: `prj5-projectname.pdf`, 其中 "-" 为英文标点符号的短横线。文件命名举例: `prj5-dma.pdf`。具体要求详见实验项目 5 讲义。
注 2:	使用 `git add` 及 `git commit` 命令将实验报告 `PDF` 文件添加到本地仓库 master 分支, 并通过 `git push` 推送到实验课 SERVE GitLab 远程仓库 master 分支(具体命令详见实验报告)。
注 3:	实验报告模板下列条目仅供参考, 可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

一、 流水线处理器的设计过程

- 流水线状态转移设计过程

我从之前的的多周期状态转移设计出发, 合并一些状态, 增添了一些状态, 设计的 5 级流水如下:(见图 1)

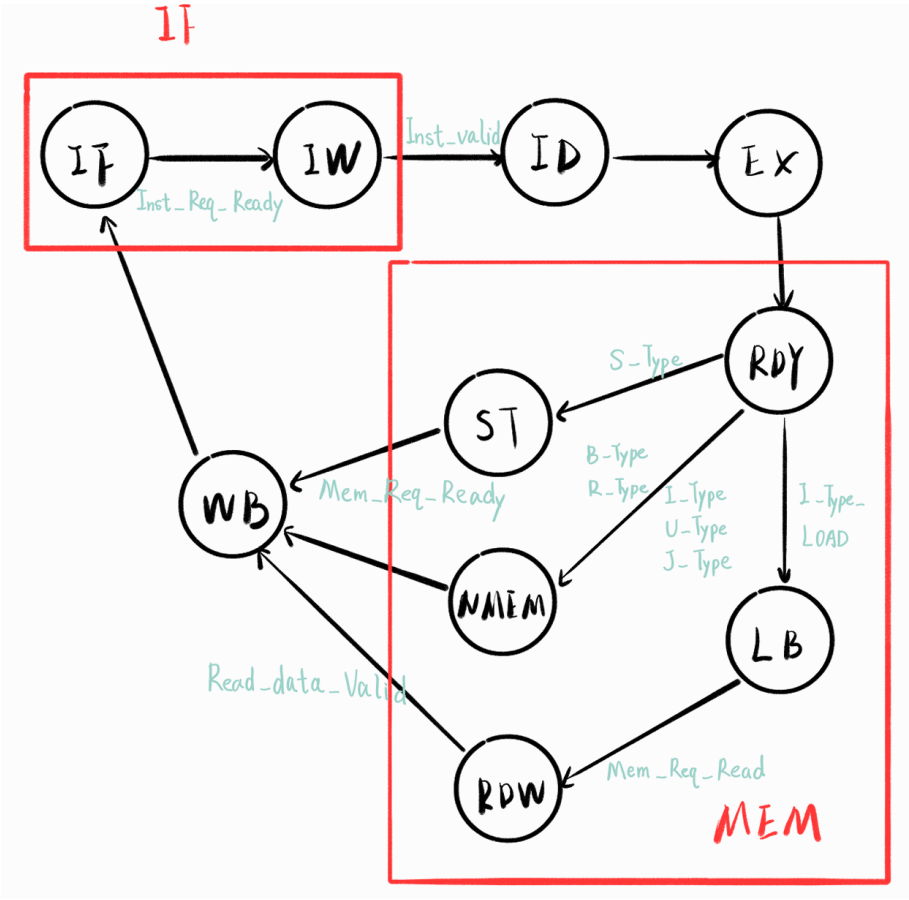

图 1: 流水线处理器状态转移图

上图为我的最初一版的流水线设计（最终实现的状态设计更复杂）。通过状态转移图可以发现，RISC-V 流水线处理器和它对应的多周期处理器的状态转移图是非常相似的。它们都包含了取指（IF）、译码（ID）、执行（EX）、访存（MEM）和写回（WB）等状态。不同点在于：

1. 本实验我打算实现 5 级流水的流水线处理器. 五级流水分别为：IF、ID、EX、MEM 和 WB。因此与多周期处理器相比，我将 IF 和 IW 统一放在了一个状态中，即 IF 阶段，将 ST、LB、RDW 统一放在了一个状态，即 MEM 阶段。

2. 在 MEM 级流水中，我相较于多周期处理器，新增了 RDY 的初始状态（目的是，让 rst 置位时，MEM 级流水处于此状态，并在此状态将 MEM_MOD_Ready 拉高，使得最初时流水线可以正常流动）和 NMEM 的状态（目的是，在 MEM 级流水中，若是非存储类指令，则进入此阶段，保证了所有指令都可以通过所有的 5 级流水）。

3. 上图中的状态转移条件与多周期处理器的状态转移条件是相似的，已在图中标注。

• IF 级流水的完善

在实验过程中，我发现 IF 级流水的设计和实现是非常重要的。而且我最初的设计是过于理想化了，并不能实现流水线处理器的所有工作。因此，在后续的设计和实现中，我对 IF 级流水进行了完善。IF 级流水的设计和实现如下（见图 2）：

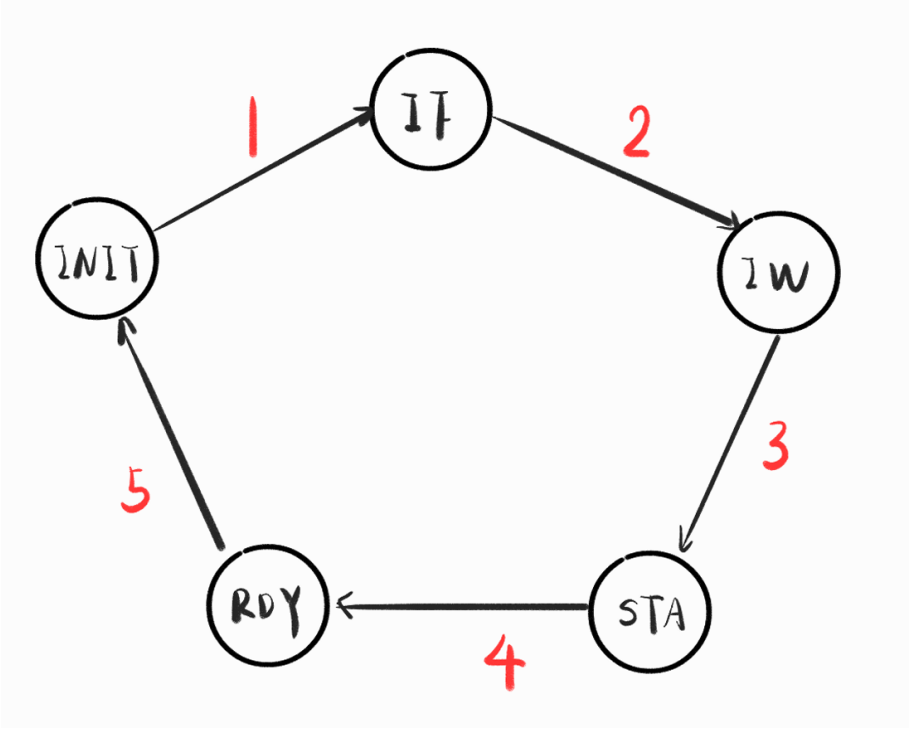

图 2: IF_MOD 级流水内部设计 1

图 2 中的设计是在 debug 过程中，我从最初的，只有 IF 与 IW 两个子阶段构成的 IF 级流水不断改进而形成的最终的 IF_MOD 的设计。对图 2 中的 IF 级流水的设计和实现进行分析：

1.INIT：此阶段用于 rst 置位时，IF 级流水回到此状态，并在此状态将 Inst_Ready 拉高，使得最初时流水线可以正常流动。

2.IF：此阶段 Inst_Req_Valid 拉高，表示 IF 级流水请求指令，等待 CPU 发出可接受指令的握手。

3.IW：在 IF 阶段，当 Inst_Req_Ready 有效时，CPU 握手完成，进入 IW 阶段。此阶段 Inst_Req_Valid 拉低，表示 IF 级流水不再请求指令，等待内存的握手。

4.STA：在 IW 阶段，当 Inst_Valid 拉高时，表示内存握手完成，可以传输指令码给 CPU，进入 STA 阶段。

5.RDY：此阶段是最初 debug 时增加的一个阶段，最初我设计的 IF_MOD 级流水只有 IF 和 IW 两个子阶

段，而且我让 PC 在 IF 阶段进行更新，取 PC = next_PC。而这样的设计，我遇到了第一个严重的逻辑错误：因为 IF_MOD 级流水向 ID_MOD 级流水传递的 next_PC 是组合逻辑，ID_MOD 级流水在接收到 PC 后，计算出 next_pc 也是组合逻辑，ID_MOD 级流水向 IF_MOD 级流水传递的 next_pc 也是组合逻辑。按照我最初设计的，仅有 IF 和 IW 两个子阶段的 IF_MOD 级流水，IF_MOD 级流水在 IF 阶段就会更新 PC。而 IF 阶段向 IW 阶段的状态转移是需要等待 Inst_Req_Ready 的握手信号的。因此，IF_MOD 级流水在 IF 阶段更新 PC 后，IF_MOD 级流水的 next_PC 会被更新为下一个指令的地址，而 ID_MOD 级流水在接收到 PC 后，计算出的 next_pc 也是下一个指令的地址。但是因为可能握手仍未完成，所以每次时钟周期，IF_MOD 级流水的 next_PC 都会被更新为下一个指令的地址，然后又把这个 PC 传给 ID_MOD, 导致 PC 进行反复的 +4 操作，示意图如下（见图 3）：

```
always @(posedge clk)begin
    if (rst) begin
        PC <= 32'b0;
    end
    else if(current_state == IF )begin
        PC <= next_pc;
    end
end
```

图 3: IF_MOD 级流水 PC 更新逻辑–1

为此，我为了让 PC 不会出现多次 +4 的错误操作，我完善了 PC 的更新逻辑（见图 4）：

```
always @(posedge clk)begin
    if (rst) begin
        PC <= 32'b0;
    end
    else if(next_state_enable)begin
        PC <= next_pc;
    end
end
```

图 4: IF_MOD 级流水 PC 更新逻辑–2

这次我让 PC 只在流水线可以流动的时候更新，就不会出现反复 +4 的错误操作了。但是我运行时第二条指令就取错了，通过检查波形图，我发现了问题所在。之前，我在 prj3 和 4 中，没有注意这一点（虽然我设计的多周期处理器也是正确的），但是在这里我才发现问题所在。就是 PC 传给内存的时间，并不是在 IF 向 IW 状态转移的那个时刻，将 PC 传给了内存，而是我的处理器进入 IF 的时刻，就会将下一条指令的 PC 传给内存。因此，我为了可以在进入 IF 阶段前就更新 PC，我设置了一个新的状态，就是这里的 RDY 状态，RDY 只停留一个时钟周期，是专门用来更新 PC 的一个阶段。IF_MOD 级流水在 RDY 阶段将 PC 传给内存（见图 5）：

```
always @(posedge clk)begin
    if (rst) begin
        PC <= 32'b0;
    end
    else if(current_state == RDY)begin
        PC <= next_pc;
    end
end

//......
RDY:begin
    next_state_reg = INIT;
end
//......
```

图 5: IF_MOD 级流水 PC 更新逻辑–3

STA 状态的设置：

最初我设计的 IF_MOD 级流水只有 IF 和 IW 两个子阶段。在经过上面的二次完善后，增加了 RDY 状态。但是我发现这样的设计仍然不够完善。本次实验中，我并没有实现真正意义上的“分支预测”，我的设计思路是“完美预测”（其实也并不算预测）。我希望在取下一条指令前，我的 ID_MOD，可以提前计算出上一条指令是不是 branch 指令或者是 jump 指令。我此前的 IF_MOD 内部状态转移如下（见图 6）：

```
IF:begin
    if(Inst_Req_Ready == 1'b1 ) begin //
        next_state_reg = IW;
    end
    else begin
        next_state_reg = current_state_reg;
    end
end
IW:begin
    if(next_state_enable) begin
        next_state_reg = RDY;
    end
    else begin
        next_state_reg = current_state_reg;
    end
end
```

图 6: IF_MOD 级流水内部状态更新逻辑–1

上面的设计是这样的：第一次流水线可以流动时，next_state_enable 会拉高，取到第一条指令，但是指令并没有传递给 ID_MOD 级流水。因为每一级流水之间的信息传递需要 next_state_enable 拉高，而第一次拉高时 IF 级流水刚刚取到指令，而流水间的数据传递是时序逻辑，因此仍然会使用旧值，所以第一条指令并没有传递给 ID_MOD 级流水。这会导致我取下一条指令时，上一条指令才会在这个时刻送往 ID_MOD 级流水。那我肯定实现不了“完美预测”，我并不能及时得出上一条指令是不是 branch 指令或者是 jump 指令。于是我又增加了 STA 状态（见图 7）：

```
IF:begin
    if(Inst_Req_Ready == 1'b1 ) begin //
        next_state_reg = IW;
    end
    else begin
        next_state_reg = current_state_reg;
    end
end
IW:begin
    if(Inst_Valid == 1'b1) begin
        next_state_reg = STA;
    end
    else begin
        next_state_reg = current_state_reg;
    end
end
STA:begin
    if(next_state_enable)begin
        next_state_reg = INIT;
    end
    else begin
        next_state_reg = current_state_reg;
    end
end
```

图 7: IF_MOD 级流水内部状态更新逻辑–2

在上面的设计中，IF_MOD 级流水在 STA 状态时，将指令传递给 ID_MOD 级流水。这样，ID_MOD 级流水在取到指令时，就可以及时得出上一条指令是不是 branch 指令或者是 jump 指令了。这样就实现了“完美预测”。

注：上面的设计过程，并没有考虑到流水线中最为复杂的“数据冲突”问题，我的设计思路是，先不考虑数据冲突问题，先实现流水线的基本功能。最后，整体框架完成后，再实现对数据冲突的处理功能。（见第二部分的设计分析）

- **MEM 级流水的完善**

在实验过程中，我发现 MEM 级流水的设计和实现是非常重要的。而且我最初的设计是过于理想化了，并不能实现流水线处理器的所有工作。因此，在后续的设计和实现中，我对 MEM 级流水进行了完善。MEM 流水的设计和实现如下（见图 8）：

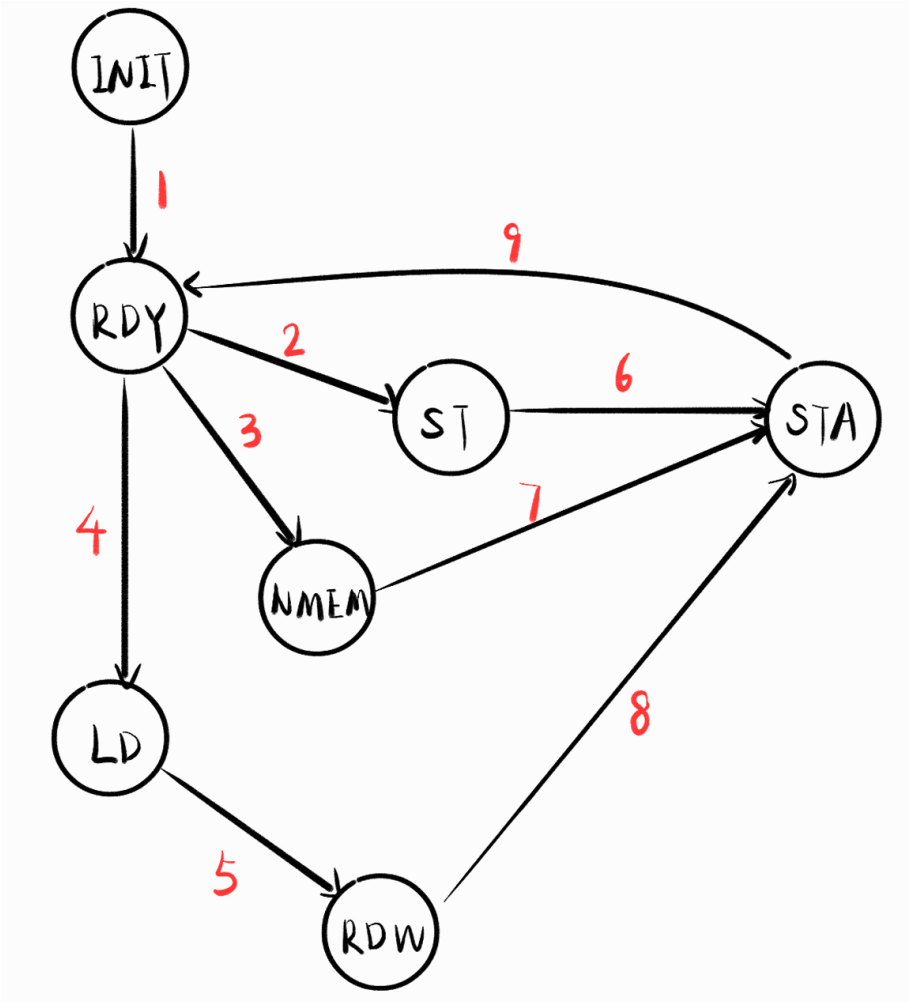

图 8: MEM_MOD 级流水内部设计

图 8 中的设计是在 debug 过程中，我从最初的，只有五个子阶段构成的 MEM 级流水不断改进而形成的最终的 MEM_MOD 的设计。对图 8 中的 MEM 级流水的设计和实现进行分析：

1.INIT：并不像 IF_MOD 级流水一样，MEM_MOD 级流水的 INIT 状态并不是用来让 MEM 级流水回到此状态，而是用来让 MEM 级流水在 rst 置位时，回到此状态，并在此状态将 MEM_MOD_Ready 拉高，使得最初时流水线可以正常流动。

2.RDY：这个阶段才是工作时，MEM_MOD 的初始阶段。这相当于一个准备工作的阶段，在这个阶段，MEM_MOD 级流水会通过译码得到的结果来选择之后的工作路径。图中：2–> 指令为 S_type 指令，3–> 指令为无访存型指令，4–> 指令为 LOAD 型指令，

3：ST、NMEM、LD–>RDW: 这一段的状态的工作方式与多周期中的 MEM 阶段的工作是一样的，这里不再赘述。

4：STA：在设计这个阶段之前，我的初次设计如下图（见图 9）：

```
//........
RDY:begin
    if(RDY_LD && next_state_enable)begin
        next_state = LD;
    end
    else if(RDY_NMEM && next_state_enable)begin
        next_state = NMEM;
    end
    else if(RDY_ST && next_state_enable) begin
        next_state = ST;
    end
    else begin
        next_state = STA;
    end
end
//........
assign MEM_Mod_Ready    = (current_state == INIT) | (current_state == RDY);
```

图 9: MEM_MOD 级流水状态设计–1

上图中，我让 ST、NMEM、RDW 三个状态都可以进入 RDY 状态。并且，流水线是否流动，会在 RDY 阶段产生影响，控制 MEM_MOD 级流水的内部状态转移。但是这样又有问题产生，我应该是先发生流水线流动，才可以把下一条指令送到 MEM_MOD，才可以得出 RDY 应该选择那一条工作路径。而按照上图的逻辑设计，流水线流动的时刻，指令传递是时序逻辑，所以这一个时刻，MEM_MOD 级流水使用的仍是 IR 的旧值，而不是下一条指令的 IR。因此，我在 RDY 阶段增加了一个状态 STA（见图 10）：

```
//........
STA:begin
    if(next_state_enable)begin
        next_state = RDY;
    end
    else begin
        next_state = current_state;
    end
end
//........
assign MEM_Mod_Ready    = (current_state == INIT) | (current_state == STA);
```

图 10: MEM_MOD 级流水状态设计–2

二、 流水线处理器的最终实现和设计分析

前面第一部分，具体讲解了我的设计过程，我是如何从最开始的多周期处理器，一点一点完善，设计出我的流水线处理器的。接下来，我将对我的流水线处理器的最终实现的代码进行分析。具体讲解“处理前递”，“数据冲突”等方面的设计和实现。

- **数据前递的设计和代码实现**

与多周期处理器相比，流水线设计过程中，最最最最为“讨厌”的一个问题就是“数据冒险”————先写后读。这个问题的处理相当棘手，我在处理这个问题的时候，采取了两种策略：数据前递和数据等待。

数据前递，顾名思义，就是先把下一条指令要用到的数据，提前传递给它使用。本来在流水线处理器中，上一条指令在 WB 阶段才会把数据写回寄存器，而下一条指令在 ID 阶段就需要使用这个数据。因此，我在设计中，利用流水线的特性，将上一条指令在 EX 阶段计算出的数据，提前传递给下一条指令的 ID 阶段使用。这样就可以完美解决，后面的指令需要用到前面的指令写入的数据的问题。我的数据前递的设计和代码实现如下（见图 11）：

```
wire [4:0] ID_reg1   = IF_ID_IR[19:15];
wire [4:0] ID_reg2   = IF_ID_IR[24:20];
wire [4:0] EX_waddr  = ID_EX_IR[11:7];
wire [4:0] MEM_waddr = EX_MEM_IR[11:7];
wire [4:0] WB_waddr  = MEM_WB_IR[11:7];
wire [31:0] EX_RF_wdata;
wire W_before_R_reg1; //rs1的先写后读
wire W_before_R_reg2; //rs2的先写后读
assign W_before_R_reg1 = ID_reg1 == 5'b0 ? 1'b0 :
  (ID_reg1 == EX_waddr && ID_EX_RF_wen )  ||
  (ID_reg1 == MEM_waddr && EX_MEM_RF_wen )||
  (ID_reg1 == WB_waddr && MEM_WB_RF_wen ) ;
assign W_before_R_reg2 = ID_reg2 == 5'b0 ? 1'b0 :
  (ID_reg2 == EX_waddr && ID_EX_RF_wen )  ||
  (ID_reg2 == MEM_waddr && EX_MEM_RF_wen )||
  (ID_reg2 == WB_waddr && MEM_WB_RF_wen ) ;
assign Ahead_value1 = (ID_reg1 == EX_waddr && ID_EX_RF_wen )   ? EX_RF_wdata :
                      (ID_reg1 == MEM_waddr && EX_MEM_RF_wen ) ? MEM_RF_wdata:
                                                                 MEM_WB_RF_wdata;
assign Ahead_value2 = (ID_reg2 == EX_waddr && ID_EX_RF_wen )   ? EX_RF_wdata :
                      (ID_reg2 == MEM_waddr && EX_MEM_RF_wen ) ? MEM_RF_wdata:
                                                                 MEM_WB_RF_wdata;
```

图 11: 数据前递的代码实现

上图代码中：我对 reg1 和 reg2 都考虑了数据前递，因为大多数指令的操作数会取两个寄存器的值。并且数据前递有多条指令都可能发生，对于在 ID 级流水的指令，它后面的 EX、MEM、WB 阶段的三条指令都还没有“写入”，所以我在这里使用了“||”，并列处理了这三条指令的情况。除此以外，我认为有一个很容易忽略的点，同时也是我犯过错误的地方（足足 de 了 3 个小时的 bug 才发现这个问题），就是零号寄存器是 * 不可以 * 使用数据前递的!!!

之前我没有考虑这一点是因为，我一直以为零号寄存器是不可以写入的，也就是说，不会有指令是往零号寄存器写东西。但是我的最初的想法是错误的，指令是很随意的，当然可以有往零号寄存器写入的指令（事实上，我们的 30 个本地测试程序有很多就是有这样的指令的）。而我反过来再看我在 Prj1 中，自己写的 Reg_file 的代码（见图 12），其实零号寄存器也确实是可以写入的，只是读取的时候会置零。

```
reg [`DATA_WIDTH - 1:0] reg_file [(1<<`ADDR_WIDTH)-1:0];

always @(posedge clk)begin
  if(wen == 1)begin
    reg_file[waddr]<=wdata;
    end
end

assign rdata1 = (raddr1 == 5'b0) ? 32'b0 : reg_file[raddr1];
assign rdata2 = (raddr2 == 5'b0) ? 32'b0 : reg_file[raddr2];
```

图 12: 寄存器堆的代码

- **数据等待的设计和代码实现**

对于可以数据前递的情况，我自然会有先选择数据前递，因为这样不会“浪费”流水。但是并不是所有的数据冒险情况我们都可以简单的使用数据前递来解决。例如这样的情况：

我的 ID 级流水的指令，需要用到 10 号寄存器的值，但是我 EX 级流水的指令会往 10 号寄存器写入一个新值。按照指令前递的思路，我在 EX 阶段算出写入值，提前传给 ID 级流水的指令即可，但是如果 EX 级流水的那条指令，是一个 LOAD 型指令，那这样就行不通了，因为 LOAD 型指令，写入寄存器的值是在经过了 MEM 级流水后，访存结束以后，才可以得到的。那么我现在显然是得不到这个值的，所以无论如何也不可能在这个时刻实现数据前递。

于是，我对这一类数据冒险的处理放式为：先进行数据等待，再进行数据前递。

数据等待，就是等待例如刚刚在 EX 级流水的那条 LOAD 型指令进入 MEM 级流水，等它读取出它所要写入寄存器的值。但是，需要注意的是，我所设计的流水线，是“一动即全动”，我的各级流水之间的数据传递是一起动的，我并不能只让刚刚在 EX 级的流水到 MEM 级，而别的不动，这显然也是会出错的。要是 MEM 级流水的东西不动，不传给 WB，那 EX 级传来的数据不就会覆盖掉 MEM 级流水原有的东西吗？这不就是平白无故“删掉”了一条指令吗？这是很危险的。

所以，为了保证正常的整体数据流动，我引入了“空泡”——也就是空指令。例如刚刚的情形，我就会让各个流水级做如下的工作：

IF 级流水：内部子状态保持不动，不能读入新的指令，否则会覆盖掉原有的指令。但 IF–>ID 的数据传递会正常进行，换句话说就是会再次向 ID 传递同一条已经传递过的指令，也就是让 ID 阶段保持不变。

ID 级流水：处在 ID 级流水的指令不变，它向 EX 传递的数据会改变，会强制传递“空泡”，即 IR、PC 以及各种译码信号等均会把全 0 信号传递给 EX 级流水。

EX、MEM、WB 级流水：正常工作即可。

具体代码实现展示如下（见图 13）：

```
//有访存的先写后读，需要NOP等待;EX阶段有一个LOAD指令，但是LOAD值在MEM之后才会得到，所以要等待
//EX阶段的，要等一次流水
    wire EX_I_LOAD_type = ID_EX_IR[6:0] == 7'b0000011;
    assign W_before_R_LD =
    (ID_reg1 == EX_waddr && EX_I_LOAD_type && ID_EX_RF_wen && ~(ID_reg1 == 5'b0)) ||
    (ID_reg2 == EX_waddr && EX_I_LOAD_type && ID_EX_RF_wen && ~(ID_reg1 == 5'b0));
//MEM阶段冲突，看似不冲突，但是因为访存之前就会先行计算is_jorb,所以应该在访存结束后再更新一次PC
    wire MEM_I_LOAD_type = EX_MEM_IR[6:0] == 7'b0000011;
    assign W_before_R_LD_MEM =
    (ID_reg1 == MEM_waddr && MEM_I_LOAD_type && EX_MEM_RF_wen && ~(ID_reg1 == 5'b0)) ||
    (ID_reg2 == MEM_waddr && MEM_I_LOAD_type && EX_MEM_RF_wen && ~(ID_reg1 == 5'b0));
```

图 13: 数据等待的控制信号代码实现

上图为实现数据等待的控制信号的产生逻辑，最开始我设计这一模块，只有前面那一部分。也就是 ID 级流水的指令要用到 EX 级流水中 LOAD 型指令的结果，而且也可以成功运行几个测试程序，但是在后续 debug 过程中，我发现了“数据冒险”在更新 PC 上的影响，因此我把可能遇到的情况分为了两种：

1.ID 级流水需要用到的“先写后读”数据来自 EX 级流水的指令的结果：控制等待的信号为：W_before_R_LD，其作用为，当 ID 级流水的指令要用到 EX 级流水中 LOAD 型指令的结果时，会出现如下的“等待”（见图 14，15）：

```
always @(posedge clk)begin
    if(rst)begin
        IF_ID_PC  <= 32'b0;
        ID_EX_PC  <= 32'b0;
        EX_MEM_PC <= 32'b0;
        MEM_WB_PC <= 32'b0;
    end
    else begin
        if(next_state_enable && ~W_before_R_LD )begin
            IF_ID_PC  <= IF_PC;
            ID_EX_PC  <= IF_ID_PC;
            EX_MEM_PC <= ID_EX_PC;
            MEM_WB_PC <= EX_MEM_PC;
        end
        else if(next_state_enable && W_before_R_LD )begin
            IF_ID_PC <= IF_ID_PC;
            ID_EX_PC <= 32'b0;
            EX_MEM_PC<= ID_EX_PC;
            MEM_WB_PC<= EX_MEM_PC;
        end
```

图 14: 数据等待时的各级流水间数据传递–PC

```
always @(posedge clk)begin
    if(rst)begin
        IF_ID_IR  <= 32'b0;
        ID_EX_IR  <= 32'b0;
        EX_MEM_IR <= 32'b0;
        MEM_WB_IR <= 32'b0;
    end
    else begin
        if(next_state_enable && ~W_before_R_LD )begin
            IF_ID_IR  <= IF_IR;
            ID_EX_IR  <= IF_ID_IR;
            EX_MEM_IR <= ID_EX_IR;
            MEM_WB_IR <= EX_MEM_IR;
        end
        else if(next_state_enable && W_before_R_LD )begin
            IF_ID_IR <= IF_ID_IR;
            ID_EX_IR <= 32'b0;
            EX_MEM_IR<= ID_EX_IR;
            MEM_WB_IR<= EX_MEM_IR;
        end
```

图 15: 数据等待时的各级流水间数据传递–IR

对于一般的数据冲突，通过上图的传递 NOP“空泡”，实现等待 EX 级指令进入 MEM 级流水，读取到数据，再执行数据前递即可解决。但是对于更新 PC 时的“数据冒险”，还需要更复杂的处理：

2.ID 级流水需要用到的“先写后读”数据来自 EX 级流水的指令的结果：这个情况很容易忽略，而且也不经常出现这种问题，这样导致我一开始没有处理这种情况，也可以 PASS 一些测试程序。但是在这样的情况下就会出现问题，例如：ID 阶段的指令是一条跳转指令，计算出 next_pc 需要用到 10 号寄存器的值，但是 MEM 级流水的的一条指令是 LOAD 型指令，它会有访存操作，然后将访存结果存入 10 号寄存器，这个时候如果我只是单纯将数据前递给 ID 级流水，是来不及的，因为 next_pc 的计算会在上一次流水运动后立马进行，然后在 IF 级流水，它会在 RDY 阶段更新 PC。那么问题也就出现了，在 IF 级流水更新 PC 时，我的 MEM 级流水内部其实还没有到 RDW 阶段，也就是访存结果还没有得到，现在即使有前递操作，也不是正确的寄存器的值。因此的引入了第二中控制信号：W_before_R_LD_MEM（见图 13）。

但是，在完成了流水线之后，我又进行了设计优化，我突然发现 W_before_R_LD_MEM 可以不需要，直接使用 MEM 级流水的 MEM_MOD_Ready 信号反而会更加简便，于是我在优化了流水线设计后，删去了 < 图 13> 中的后半部分，同时更改了我的 IF_MOD 的设计。对 IF 级流水的状态转移做出了一些调整（见图 16）：

```
INIT:begin
    if(rst)begin
        next_state_reg = current_state_reg;
    end
    else begin
        if(jorb_type)begin
            if(W_before_R_LD)begin
                next_state_reg = current_state_reg;
            end
            else if(~W_before_R_LD && ~MEM_Mod_Ready)begin
                next_state_reg = current_state_reg;
            end
            else begin
                next_state_reg = IF;
            end
        end
        else begin
            if(W_before_R_LD)begin
                next_state_reg = current_state_reg;
            end
            else begin
                next_state_reg = IF;
            end
        end
```

图 16: INIT 等待访存结果的代码实现

上图中，我第一个 if 的判断就是根据，当前在 ID 级流水的指令是不是一个有关跳转的指令来判断的。如果不是的话，那自然和最开始我的设计没有任何区别；但是如果是的话，就需要考虑到上的的那种情况，因此我对 INIT–>MOD 的状态转移的控制更为严格。对于 MEM 级流水内部还没有准备好，也就是并没有访存结束的时候，我也不让 INIT 状态转移到 IF 状态，这样等访存结束后，数据已经前递给了 ID 级流水，它也算出了正确的 next_pc, 这时候再让 PC 更新，就不会出现错误了。

三、 实验中的难点和优化

1 实验中的难点

—— 由于我是所有选做实验完成后开始的实验报告，这里不具体结合波形图和错误代码进行讲解说明（现在只有最终正确的一版代码）

本实验中，我的设计流水线花费 1 天，完成代码 2 天，debug 足足弄了 4 天，而且是全天啥也不干，只有 debug。对于流水线的 debug，我有很多感悟：

- **代码设计易错点**

1. 零号寄存器不进行数据前递（因为零号寄存器读数必然为 0）

2.ID 级流水模块传递回 IF 级流水模块 next_pc 应该是组合逻辑，但是 IF 级流水模块传递给 ID 级流水模块的 PC 必须是时许逻辑，只有整个流水线动的时候，才可以传递（这样才可以避免 PC 反复 +4 操作）

3.is_jorb 信号不适用于控制 INIT 向 IF 的状态转移，需要引入 jorb_type 信号（因为 is_jorb 在 MEM 访

存结束前后是可能会改变的，它的意思是，这条指令要不要跳转；而 jorb_type 不一样，它只与 IR 有关，它的意思是，这条指令是不是一条跳转类型的指令）（见图 16）

```
//控制流水线是否前进
  wire next_state_enable;
  wire jorb_type; //流水线中ID阶段的指令是不是（可能）会跳转的指令
  wire is_jorb;   //流水线中ID阶段的指令是不是要跳转的指令
  wire IF_Mod_Ready;
  wire MEM_Mod_Ready;
  wire W_before_R_LD;  //先写后读（有访存）
```

图 17: is_jorb 与 jorb_type

4.WB 级流水的输出，必须只有流水线流动时输出有效，其余时刻应该无效。可以采取拉低输出的策略来使之无效，这样才可以避免多次提交（见图 18）

```
//WB模块
module WB_Mod(
    input  [ 4:0] RF_waddr,
    input         RF_wen,
    input         clk,
    input         rst,
    input  [31:0] RF_wdata,
    input  [31:0] PC,
    input         next_state_enable,
    output [69:0] inst_retire
);
    assign inst_retire = {70{next_state_enable}} & {RF_wen, RF_waddr, RF_wdata,PC};
endmodule
```

图 18: WB 级流水的输出设计

• **debug 难点和总结**

一下几点是我长达 4 天 debug 的亲身感受，流水线 debug 相较于之前的实验，难点有：

1. 模块太多，对于一个 PC 出错，就需要在 CPU 总模块中调出 clk,instruction,next_state_enable, inst_retire 等信号，还需要在 IF_MOD 模块中调出 pc,IR,next_pc,current_state,jorb_type 等信号，要是这个 PC 有关的上一条指令还是 LOAD 型，而且出现了数据冲突，那还得调出来 ID_MOD 和 MEM_MOD 模块的相关信号。

2. 既有很多组合逻辑，又有很多时序逻辑。在看波形图时，某一个信号，在某一次上升沿时拉高了，但是它不一定在这个时刻使用的拉高之后的值，这得看这个信号是 wire 型还是 reg 型。这对我们看波形图也是很大的挑战。

3. 这次实验的波形图只有我们自己的输出的波形图，没有“金标准”模块，这样对我们找错又产生了很大的影响。这会导致我工作量激增，因为在有标准的波形图参考时，例如我的 a=b+c 这样的一个 a 信号出错了，我可以调出 b 和 c 的信号，并于标准模块的波形图对比。如果 b 和 c 没有问题，说明我的 a 信号的产生逻辑，即“a=b+c”是错误的，反过来，要是 b 或者 c 与标准不同，那就是 b 和 c 出错了。但是在本次 debug 中，由于没有标准来参考，我必须同时考虑以上两种可能，这样倒推回去，工作量会指数上涨。

2 优化改进

第一次 Dhrystone 测试程序处理性能为 4800 左右（已经完成了 cache），之后我又检查了我的流水线的 IF_MOD 的设计。我发现最开始我添加的 RDY 状态，用来更新 PC 的这个作用，其实已经改进为 INIT 状态更新 PC 了，但是我最开始设计的 RDY 状态还在。我反复检查过后，发现完全可以实现删除 RDY 这个状态，这样可以让我买两次流水线流动之间减少 1 个时钟周期。在修改完成代码以后，Dhrystone 测试程序处理性能增加至 5420 左右，效率提升十分显著。

四、 思考题

问题：如何处理先写后读的数据依赖？

对于非 LOAD 类型写寄存器指令，通过数据前递的方式处理；对于 LOAD 型指令，采用等待的方式，在译码级填充“空泡”。具体内容详见实验报告 8 13 页内容。

实验总结和感悟：

本次流水线设计实验让我深刻体会到，将复杂的指令执行过程分解为取指、译码、执行、访存、写回等阶段，是提升处理器吞吐率的核心思想。实践中，最大的挑战在于处理数据冒险和正确实现 PC 的更新。通过设计前递和等待单元解决数据依赖，我不仅掌握了关键技术，更感悟到现代 CPU 设计的精妙，正是源于对这些冲突的无数次精巧权衡与优化。这不仅是编码能力的锻炼，更是对计算机体系结构思想的深度领会。

五、 实验所耗时间

在课后，你花费了大约 70（设计流水线 5 小时，代码完成 20 小时，Debug 完成 45 小时，完成实验报告 10 小时）小时完成此次实验。

致谢：感谢韩初晓同学在我第一天设计流水线上的帮助和朱徐塬助教在实验 debug 过程中给予的指导