中国科学院大学
《计算机组成原理(研讨课)》实验报告

姓名 薛翼舟 学号 2023K8009929044 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 cpu 设计

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、 逻辑电路结构与仿真波形的截图及说明(比如 Verilog HDL 关键代码段 {包含注释} 及其对应的逻辑电路结构图 {自行画图, 推荐用 PPT 画逻辑结构框图后保存为 PDF, 再插入到 LaTeX. 中}、相应信号的仿真波形和信号变化的说明等)

本次是实验所采用的指令集是 RISC-V 的 RV32I 指令集, 并且基于 prj4 的多周期 cpu 来进行流水线的适应性改造. 在本次实验中, 我所实现的是 5 级 cpu 流水线, 包括 IF(取指), ID(译码), EX(执行), MEM(存储器访问), WB(写回). 在这个部分中, 我将讲述我在本次实验中的几个关键设计, 以及各个阶段的分别讲解, 还有潜在问题的讨论以及优化可能的思考.

1.1 流水线改造握手信号

在本次实验所采用的五级流水线之中, 两个相邻的流水线阶段之间利用握手信号连接, 比如 ID 阶段和 EX 阶段, ID 阶段向 EX 部分发送信号 ID_valid, 另外 EX 阶段向 ID 部分发送 EX_ready 信号, 当两个信号同时拉高即握手成功, 这时将 ID 阶段的数据传输给 EX, 之后 EX 阶段对这些数据进行操作.

```
1 wire    IF_to_ID_valid;
2 wire    ID_ready;
3 wire    ID_to_EX_valid;
4 wire    EX_ready;
5 wire    EX_to_MEM_valid;
6 wire    MEM_ready;
7 wire    MEM_to_WB_valid;
8 wire    WB_ready;
```

1.2 流水线阶段之间的数据传输

由于在流水线中, 分支预测的存在, 当前流水线中所执行的命令可能并不是正确的命令, 因此在每个阶段之间传输的数据需要包含当前指令所运行的 PC 值等信息, 来为之后的冲刷和阻塞提供判断信息

```
wire    [`IF_TO_ID_WIDTH-1:0]       IF_to_ID_data;
wire    [`ID_TO_EX_WIDTH-1:0]       ID_to_EX_data;
wire    [`EX_TO_MEM_WIDTH-1:0]      EX_to_MEM_data;
wire    [`EX_ID_BYPATH_WIDTH-1:0]   EX_to_ID_bypath_data;
wire    [`MEM_TO_WB_WIDTH-1:0]      MEM_to_WB_data;
wire    [`MEM_ID_BYPATH_WIDTH-1:0]  MEM_to_ID_bypath_data;
wire    [`WB_ID_BYPATH_WIDTH-1:0]   WB_to_ID_bypath_data;
```

以上是各个阶段之间数据连接的端口, 比如从 ID 到 EX 阶段中, 这个 ID_to_EX_data 包含了以下内容, 包含分支预测的结果, 当前的 PC, 以及之后可能所用到的数据

```
assign ID_to_EX_data =  {
                          mul,                      //添加mul命令
                          U_imm,
                          opcode,
                          PC_ID,
                          prediction_out,
                          prediction_addr,
                          ALUop,
                          ALUop_A,
                          ALUop_B,
                          Shiftop,
                          Shiftop_A,
                          Shiftop_B,
                          AluShi_sel,
                          Jump,
                          Branch,
                          func,           //用于存储分支的类型
                          load,
                          store,
                          RF_wen_i,
                          RF_waddr,
                          rdata2_true    //31:0用于store指令
                        };
```

1.3 分支预测器的设计

在本次实验中, 我的分支预测器主要采用了比较经典的四阶段分支预测器 (强分支, 弱分支, 弱不分支, 强不分支), 其设计类似于不倒翁, 当一次分支预测不正确之后, 其分支预测当前阶段向另一侧移动一个阶段, 其设计的主要目的是为了减少分支预测错误带来的流水线冲刷和阻塞, 根据当前指令的分支预测结果逐渐调整预测的方向, 使得分支预测器能够更好地适应当前指令的分支情况.

```
always @ (*) begin
    case (predictor_current_state)
        INIT : begin
            if(rst)begin
                predictor_next_state = INIT ;
                            end
            else begin
                predictor_next_state = S_TAKE ;
                            end
        end
        S_TAKE : begin
            if(predictor_update && prediction_incorrect && is_Branch) begin
                predictor_next_state = W_TAKE ;
                            end
            else begin
                predictor_next_state = S_TAKE ;
                            end
        end
        W_TAKE : begin
            if(predictor_update && prediction_incorrect && is_Branch)begin
                predictor_next_state = W_NTAKE ;
                            end
            else if (predictor_update && ~prediction_incorrect &&is_Branch)begin
                predictor_next_state = S_TAKE ;
                            end
            else begin
                predictor_next_state = W_TAKE ;
                            end
        end
        W_NTAKE : begin
            if(predictor_update && prediction_incorrect && is_Branch)begin
                predictor_next_state = S_NTAKE ;
```

```
                            end
            else if (predictor_update && ~prediction_incorrect &&is_Branch)begin
                predictor_next_state = W_TAKE ;
                            end
            else begin
                predictor_next_state = W_NTAKE ;
                            end
        end
        S_NTAKE : begin
            if(predictor_update && prediction_incorrect && is_Branch)begin
                predictor_next_state = S_NTAKE ;
                            end
            else if(predictor_update && ~prediction_incorrect && is_Branch)begin
                predictor_next_state = W_NTAKE ;
                            end
            else begin
                predictor_next_state = S_NTAKE ;
                            end
        end
        default: begin
            predictor_next_state = INIT;
                    end
    endcase
end

assign  prediction_out = (predictor_current_state == S_TAKE)
        || (predictor_current_state == W_TAKE);
```

1.4 IF 取值阶段的设计

在本次实验中, 由于分支预测的存在, 导致 IF 阶段为本次实验中最为关键, 也设计最为困难和需要构思的阶段, 主要要考虑的方向就是在存在分支预测下, 如何正确地更新 PC 以及如何正确的取指 (如何把错误的指令进行冲刷). 以下是 IF 阶段的状态机设计

```
always @(*) begin
    case (IF_current_state)
            INIT:begin
                    IF_next_state   =   IF;
            end
            IF:begin
```

```
7                       //如果分支预测不正确，那么就保持在取指阶段，最后实现冲刷
8                       if(prediction_incorrect)begin
9                               IF_next_state   =   IF;
10                      end
11                      else if(Inst_Req_Ready && Inst_Req_Valid) begin
12                              IF_next_state   =   IW;
13                      end
14                      else begin
15                              IF_next_state   =   IF;
16                      end
17              end
18              IW:begin
19                      if(prediction_incorrect)begin
20                              IF_next_state   =   IF;
21                      end
22                      else if(Inst_Ready && Inst_Valid)begin
23                              IF_next_state   =   INS_GO;
24                      end
25                      else begin
26                              IF_next_state   =   IW;
27                      end
28              end
29              INS_GO:begin
30                      if(prediction_incorrect)begin
31                              IF_next_state   =   IF;
32                      end
33                      //传输出指令
34                      else if (ID_ready) begin
35                              IF_next_state   =   INS_DONE;
36                      end
37                      else begin
38                              IF_next_state   =   INS_GO;
39                      end
40              end
41              //这个状态实际上是用来更新PC的
42              INS_DONE:begin
43                      if(ID_ready) begin
44                              IF_next_state  =    IF;
```

```
                    end
                    else begin
                            IF_next_state    =    INS_DONE;
                    end
            end
            default:begin
                    IF_next_state    =    INIT;
            end
    endcase
end
```

从这个状态机来看, IF 和 IW 阶段就是原本多周期 CPU 中的 IF 和 IW 阶段, 另外新增了 INS_GO 和 INS_DONE 两个阶段, 分别代表的就是即将向 ID 阶段传输指令和向 ID 阶段传输指令完成之后的状态, 值得一提的时, 这个 INS_DONE 状态是专门用来设计 PC 更新的, 因为我在考虑分支预测的时候, 发现指令 1 的分支预测结果实际上实在指令 3 中实现, 因此如果遇到两条相邻的跳转或者分支指令, 可能会导致 CPU 运行的结果偏离我们的预期, 因此我在这个状态中实现了 PC 的更新, 使得 CPU 能够正确地运行.

另外, 这个状态机有很大优化的空间, 因为在我当前的设计下, 一次 IF 阶段需要至少进行 4 个周期, 而考虑到之后的 ID 和 EX 阶段以及 WB 阶段都是只需要一个周期就可完成, 这就导致了 IF 阶段对性能的限制比较高, 考虑不改变当前 IF 阶段总体设计逻辑的情况下, 我目前的设计而言, 这个 INS_GO 基本上是没有用, 完全可以合并到 IW 或者 INS_DONE 阶段.

此外, 这个设计还有另一个潜在的问题, 因为在这个阶段有连续的两个和内存得握手信号 (Ins_req_valid 和 inst_ready), 这两个信号基本是连续进行的, 所以在面临分支预测不正确因而需要冲刷的情况下, 有可能会出现有前一个握手之后冲刷阻塞住, 而后一个握手信号还没有拉高的情况, 这个信号就会一直悬空, 导致程序无法正确进行, 所以最保险的方案就是设计一个信号, 其控制如果遇到冲刷信号之后, 在两个握手都完成后再进行冲刷

以下是 IF 阶段 PC 更新的设计

```
always @(posedge clk) begin
        if(rst) begin
                PC_reg    <=    32'b0;
        end
        else if(prediction_incorrect) begin
                PC_reg    <=    PC_correct;
        end
        else if(IF_current_state == INS_DONE && ID_ready) begin
                if(prediction_yes) begin
                        PC_reg    <=    prediction_addr;
                end
```

```
                else begin
                        PC_reg    <=    PC_reg + 4;
                end
        end
        else begin
                PC_reg    <=    PC_reg;
        end
end
```

其中包含了分支预测的更新, 另外值得一提的是如果遇到分支预测不正确, 这个更新是不依赖于当前 IF 阶段的状态的, 直接将 PC 更新为正确的 PC 值

以下是一些控制信号的设计

```
wire    IF_done;
assign  IF_done  =  (IF_current_state == INS_DONE) && ~prediction_incorrect;

//向ID的握手信号
assign  IF_to_ID_valid    =    (IF_current_state == INS_GO) && (~prediction_incorrect);
```

从这段代码我们可以很直观地看出这个冲刷得逻辑, 就是在 prediction_incorrect 的情况下, IF 阶段不向 ID 阶段流动, 接下来 IF 阶段中得数据就会被正确的 PC 和 ins 所冲刷, 之后向 ID 传输正确的指令

1.5 ID 阶段的设计

ID 阶段的设计比较平凡, 和之前的实验项目基本上没有什么的区别, 在这里只提一些流水线改动中比较特别的部分

以下是 ID 阶段工作状态的更新

```
always @(posedge clk) begin
        if(rst) begin
                ID_work  <=  0;
        end
        else begin
                if(prediction_incorrect) begin
                        ID_work  <=  0;
                end
                else if(ID_ready) begin
                        ID_work  <=  IF_to_ID_valid;
                end
        end
end
```

之后的阶段设计也基本上是这个思路, 之后就不多赘述了, 这里的工作状态比较关键, 概括就是在 ID_ready 时根据握手信号来更新工作状态, ID 阶段比较独特的就是在分支预测不正确的时候阻塞住.

在本次实验中, 一个重点就是数据冒险 (在我的设计中, 只有写后读的出现, 其他的数据冒险不会影响我的设计), 我采用的解决方法是数据旁路, 从之后的 EX,MEM,WB 阶段向 ID 阶段连接数据旁路, 用于在比较小的时间开销下解决数据冒险 (这三个数据旁路的选择是有优先级的, 基本的原则就是先选择最" 新鲜" 的数据), 代码如下

```
assign  rdata1_true  =  (rs1==5'b0)?                          32'b0:
                        (EX_write && rs1==EX_waddr)?          EX_data:
                        (MEM_write && rs1==MEM_waddr)?        MEM_data:
                        (WB_wen && rs1==WB_waddr)?            WB_data:
                                                              rdata1;

assign  rdata2_true  =  (rs2==5'b0)?                          32'b0:
                        (EX_write && rs2==EX_waddr)?          EX_data:
                        (MEM_write && rs2==MEM_waddr)?        MEM_data:
                        (WB_wen && rs2==WB_waddr)?            WB_data:
                                                              rdata2;
```

以下是对于数据旁路端口的定义

```
assign  {
        EX_load,
        EX_write,
        EX_waddr,
        EX_data
                }  =  EX_to_ID_bypath_data;

assign  {
        MEM_load,
        MEM_write,
        MEM_waddr,
        MEM_data
                }  =  MEM_to_ID_bypath_data;

assign  {
        WB_wen,
        WB_waddr,
        WB_data
                }  =  WB_to_ID_bypath_data;
```

```
//标记出现了数据冒险
assign  EX_related   =  (EX_write) && (rs1==EX_waddr || rs2==EX_waddr);
assign  MEM_related  =  (MEM_write) && (rs1==MEM_waddr || rs2==MEM_waddr);

//如果遇到load指令才阻塞，不然直接传回来就可以
assign  block        =  (EX_related && EX_load) || (MEM_load && MEM_related);
```

当都地址和写地址相同, 并且确实是写命令时, 就会发生数据冒险, 这时候如果不是 load 指令, 就直接传输数据, 如果是 load 指令, 那么就会阻塞 ID 阶段, 直到 load 出正确的数据. (因为 load 指令在当时并不能有争取的结果, 所以需要等待访存后才能进行, 这里的 EX_related 和 MEM_related 就是这样两个信号)

另外, 我在 ID 阶段进行分支预测, 以下是 ID 阶段对于分支预测的实例化和设计

```
wire    [31:0]  prediction_addr = (Branch)?     PC_Branch:
                                  (Jump)?       PC_Jump:
                                                32'b0;

wire    prediction_out;
reg     predictor_update;

always @(posedge clk) begin
        if(ID_to_EX_valid)
                predictor_update <= 1'b1;
        else
                predictor_update <= 1'b0;
end

//分支预测的结果
wire    prediction_yes  =   (Jump) || ((opcode==`B_type) && prediction_out);

branch_predictor  prediction_ex(
        .clk                            (clk),
        .rst                            (rst),
        .is_Branch                      (Branch),
        .predictor_update               (predictor_update),
        .prediction_incorrect           (prediction_incorrect),
        .prediction_out                 (prediction_out)
);

//对各个数据进行赋值
```

```
assign  predictor_to_IF_data =  {
                                  prediction_yes,
                                  prediction_addr
                                };
```

其中, 分支预测中分支目标地址的生成不必多说, prediction_out 是根据分支预测器当前状态产生的分支结果, 然后分支预测的最终结果是 jump 或者分支且分支预测正确, 此外还有一个信号就是 prediction_update, 就是每一次在 ID 阶段工作时, 进行一次分支预测器状态更新, 否则分支预测器就会一直更新状态而导致分支预测的结果不符合我们的预期, 从而极大程度上提高时间开销

此外, 我在 ID 阶段进行寄存器堆的实例化, 以下是相关代码段

```
assign  {
        WB_wen,
        WB_waddr,
        WB_data
                } =  WB_to_ID_bypath_data;

//实例化regfile, 写部分用的是WB传回来的总线
reg_file reg_file_ex(
    .clk                (clk),
        .waddr          (WB_waddr),
        .raddr1         (rs1),
        .raddr2         (rs2),
        .wen            (WB_wen),
        .wdata          (WB_data),
        .rdata1         (rdata1),
        .rdata2         (rdata2)
);
```

这里直接利用 WB 阶段的数据旁路来进行数据的写入

以下是控制信号的设计

```
assign  ID_ready    =   ~ID_work  ||  (ID_done && EX_ready);
```

比较值得一提的就是, ID_ready 信号拉高有两个情况, 一是没有工作, 而是正好可以向下流, 在这两种情况下, 在下一个周期可以接受来自 IF 阶段的信号, 之后其他阶段的设计也都是相同的设计, 之后不多赘述

1.6 EX 阶段的设计

EX 阶段的设计也同样比较平凡, 值得一提的就是对于分支预测阶段的处理, 代码段如下

```
wire    Branch_check;
assign  Branch_check  = (func==3'b000)?                          Zero:
                        (func==3'b001)?                          ~Zero:
                        (func==3'b100 || func==3'b110)?          ~Zero:
                        (func==3'b101 || func==3'b111)?          Zero:
                                                                 0;

assign  Branch_real = Branch_check && Branch;

reg     block_cancel;

always@(posedge clk) begin
        if(rst) begin
                block_cancel  <=  1;
        end
        else if(ID_to_EX_valid) begin
                block_cancel  <=  0;
        end
        else begin
                block_cancel  <=  1;
        end
end

assign prediction_incorrect   =  (Branch_real ^ prediction_result)
        && ~block_cancel;//异或表示分支预测结果和实际相反

assign  PC_correct   =  (Branch_real || Jump)?  prediction_addr : PC_EX+4;
```

这里的异或表示真正的分支结果和分支预测的结果不相同, 另外, 值得一提的是 block_cancel 信号, 这个信号的作用是为了在分支预测不正确的时候, 取消阻塞 ID 阶段, 因为如果没有这个信号, 面临分支预测不正确, 那么 prediction_incorrect 信号就会被一直拉高, 而导致流水线被一直阻塞住, 不能继续往下进行

1.7 MEM 阶段的设计

MEM 阶段也包括原来多周期中的多个状态, 内部状态机设计如下

```
always @(*) begin
    case (MEM_current_state)
            INIT: begin
```

```
                MEM_next_state = SL_BEFORE;
        end
        SL_BEFORE: begin
                //标记有没有用MEM, 是不是load或者store指令
                //如果没有，则直接结束
                if(MEM_work) begin
                        if(load || store) begin
                                MEM_next_state = SL;
                        end
                        else begin
                                MEM_next_state = SL_DONE;
                        end
                end
                else begin
                        MEM_next_state = SL_BEFORE;
                end
        end
        SL: begin
                //Load会进入RDW阶段等待读出数据握手
                if(load && Mem_Req_Ready) begin
                        MEM_next_state = RDW;
                end
                else if(store && Mem_Req_Ready) begin
                        MEM_next_state = SL_DONE;
                end
                else begin
                        MEM_next_state = SL;
                end
        end
        RDW: begin
                if(Read_data_Valid && Read_data_Ready) begin
                        MEM_next_state = SL_DONE;
                end
                else begin
                        MEM_next_state = RDW;
                end
        end
        SL_DONE: begin
```

```
                    //如果EX发起了握手请求，就返回
                    if(EX_to_MEM_valid) begin
                            MEM_next_state = SL_BEFORE;
                    end
                    else begin
                            MEM_next_state = SL_DONE;
                    end
            end
            default: begin
                    MEM_next_state = INIT;
            end
    endcase
end
```

比较值得一提的是设计了 SL 这个阶段, 用于判断是读指令还是写指令, 不过这个阶段貌似也没有什么作用, 可以直接从 SL_BEFORE 跳转至对应的 RDW 和 SL_DONE 阶段, 来进行时间开销下的优化, 此外 MEM 阶段和之前的也比较类似, 在此不多赘述

1.8 WB 阶段的设计

WB 阶段的设计同样也比较简单, 主要的用处就是将数据写回给寄存器堆 (寄存器堆在 ID 阶段实例化), 此外还有一个点就是 inst_retire 的产生, 代码段如下

```
//用来控制inst_retired的控制信号，保证这个不是一直改变的
//用来避免重复的比较
wire    inst_retire_valid   =   WB_done && WB_work;
wire    inst_wen            =   inst_retire_valid && RF_wen;
assign  inst_retired  = {
                          inst_wen,
                          RF_waddr,
                          RF_wdata,
                          PC_WB
                        };
```

这里的 inst_wen 的信号主要是保证每次直在 WB 阶段有效的时候才进行 inst 的比较, 来避免多次的比较, 导致和金标准不同, 在本地测试时出现 ERROR (不过这个不影响本身 CPU 的工作)

二、 实验过程中遇到的问题、对问题的思考过程及解决方法

在本次实验之中, 遇到的 bug 非常多, 下面将堆一些比较关键的 bug 和解决方法进行讲述

2.1 流水线无法正确流动

由于在本次实验中, 流水线的运行实际上非常依赖于握手信号的产生, 如果不能正确的产生握手信号, 那么流水线就会一直卡在握手的环节, 导致一直一直等待, 无法进行, 所以这种 bug 除了干瞪眼以外, 首先就是可以利用改变 sim_main.cpp 中 TIME_LIMIT 宏, 来确定是否是产生死循环或者一直等待, 之后就是利用 emu 加速仿真的, 改变仿真的时钟周期数, 之后观察波形来检测哪里的握手信号一直悬空来 debug

2.2 分支预测导致的一直阻塞

在调试过程中, 同样有一种一直卡住的问题, 就是分支预测错误后的冲刷, 因为我们实际上需要的 prediction_incorrect 是一个脉冲信号, 而非一直拉高, 如果没有一个信号来控制它的位置, 那么整个流水线就会一直停留在阻塞的状态不继续向下进行, 于是我利用了 block_cancel 来进行阻塞的取消. 这样一个周期之后 ID 和 IF 阶段就会有正确的结果, 之后恢复流水线的运作

2.3 数据冒险中面临的问题

我再调试过程中, 发现了一个问题就是面对 LUI 和 AUIPC 指令, 虽然我加入了数据旁路, 但是并不能正确地写回正确的值, 后来经过调试我发现, 我直接复制粘贴的原本多周期的代码, 而在我多周期的代码中, 有关 AUIPC 和 LUI 的写回代码如下

```
assign  RF_wdata  =     (opcode==`JAL ||opcode==`JALR)?         PC_MEM + 4:
                        (opcode==`AUIPC)?                       PC_MEM + U_imm:
                        (opcode==`LUI)?                         U_imm:
                        (opcode==`I_type_l)?
                        ((func[1:0]==2'b00)?    ((eff==2'b00)?  ((func[2]==0)?  {{24{read_byte_0[7]}},read_byte_0} :    {{24{1'b0}},read_byte_0}):
                                                 (eff==2'b01)?  ((func[2]==0)?  {{24{read_byte_1[7]}},read_byte_1} :    {{24{1'b0}},read_byte_1}):
                                                 (eff==2'b10)?  ((func[2]==0)?  {{24{read_byte_2[7]}},read_byte_2} :    {{24{1'b0}},read_byte_2}):
                                                                ((func[2]==0)?  {{24{read_byte_3[7]}},read_byte_3} :    {{24{1'b0}},read_byte_3}) ):
                         (func[1:0]==2'b01)?    ((eff==2'b00)?  ((func[2]==0)?  {{24{read_byte_1[7]}},Read_data_current [15:0]} : {{24{1'b0}},Read_data_current [15:0]}):
                                                                ((func[2]==0)?  {{24{read_byte_3[7]}},Read_data_current [31:16]} : {{24{1'b0}},Read_data_current [31:16]})):
                         Read_data_current
                        ):
                        Result;
```

而我直接将阶段代码复制到了 MEM 当中, 然后最终在 WB 阶段谢欢, 这也就意味着我的这个 LUI 和 AUIPC 这两个不需要访存的结果只有在一直运行到 MEM 和 WB 阶段之后才能产生, 这样实际上就落后了时序, 导致我的数据旁路并不能取出正确的数据, 于是后来我将这几个命令的写回结果的产生前提到 EX 阶段的 Result 中, 这样就能在最早的产生的时候就能取出正确的结果, 代码如下

```
1  assign  Result     =     (opcode==`JAL ||opcode==`JALR)?          PC_EX + 4:
2                           (opcode==`AUIPC)?                        PC_EX + U_imm:
```

```
                        (opcode==`LUI)?                         U_imm:
                        (mul)?                                  mul_result[31:0]:
                        (AluShi_sel)?                           Shifter_result:
                                                                ALU_result;
```

2.4 inst_retired 的错误比较

在一开始, 我在 WB 阶段有关 inst_retired 的代码段如下

```
assign  inst_retired  = {
                          RF_wen,
                          RF_waddr,
                          RF_wdata,
                          PC_WB
                        };
```

这样就会导致不管 WB 阶段有没有工作, 都一直进行比较, 然后就会导致和金标准的 CPU 的时序产生偏差, 导致不能产生正确的结果 (实际上这个比较的信号在我的理解中应该不会影响 cpu 的运行, 但是会导致和金标准的比较出现偏差), 于是我加入了几个控制信号, 形成了如下代码

```
//用来控制inst_retired的控制信号，保证这个不是一直改变的
//用来避免重复的比较
wire    inst_retire_valid   =   WB_done && WB_work;
wire    inst_wen            =   inst_retire_valid && RF_wen;
assign  inst_retired  = {
                          inst_wen,
                          RF_waddr,
                          RF_wdata,
                          PC_WB
                        };
```

通过添加 inst_retire_valid, 并借此产生 inst_wen 信号, 使得 inst_retired 的比较只在 WB 阶段工作的时候进行, 这样就能避免重复的比较, 进而和金标准一致

三、 性能评估与比较

根据几个性能计数器, 我们比较流水线 riscv 和多周期 riscv 的性能, 其结果如下

		指令数	周期数	时间(ms)	CPI	CPI加速比	时间加速比
15pz	流水线	5224461	397491252	3989.62	76.08	1.316711697	1.315609
	多周期	5224461	523381381	5248.78	100.18		
bf	流水线	452834	32335500	551.84	71.41	1.20028863	1.246865
	多周期	452834	38811933	688.07	85.71		
dinic	流水线	16671	1184976	4873.24	71.08	1.245437882	1.3576307
	多周期	16671	1475814	6616.06	88.53		
fib	流水线	2549505	180782438	1823.42	70.91	1.001691414	1.0019085
	多周期	2549505	181088216	1826.90	71.03		
md5	流水线	4895	346992	161.40	70.89	1.108616337	1.5730483
	多周期	4895	384681	253.89	78.59		
queen	流水线	81470	5776753	70.57	70.91	1.185683895	1.1609749
	多周期	81470	6849403	81.93	84.07		
sieve	流水线	10175	721380	20.87	70.90	1.032218803	1.0364159
	多周期	10175	744622	21.63	73.18		
ssort	流水线	619025	43979362	1179.16	71.05	1.017112345	1.1420672
	多周期	619025	44731952	1346.68	72.26		
qsort	流水线	9460	670637	753.02	70.89	1.167136618	1.6092401
	多周期	9460	782725	1211.79	82.74		
平均CPI加速比		平均时间加速比					
1.141655291		1.271528856					

我们可以看到, 流水线 cpu 对于多周期 cpu 的有一定程度上的性能提升, 但是这个提升不是很大, 主要是我的设计中有一些多余的状态设计, 另外, 在这整个指令的执行过程中, 主要的时间开销还是在访存上, 所以如果放存的周期很长, ID 和 EX 阶段实际上都是组合逻辑的, 一个周期就可以完成, 而流水线的 cpu 和多周期的 cpu 在访存上没有什么区别, 所以加速情况不是特别理想, 但是 cache 的加速就很理想, 详见下一个报告

四、 实验所耗时间

在课后，你花费了大约____31____ 小时完成此次实验。