计算机系统结构实验报告

lab6

实验名称: 多周期流水线 mips 处理器

学生姓名: 郭奕臻

学　　号: 519021910507

2021 年 *5* 月 *5* 日

目录

1 实验目的

1. 理解 CPU Pipeline，了解流水线冒险 (hazard) 及相关性，设计基础流水线 CPU
2. 设计支持 Stall 的流水线 CPU。通过检测竞争并插入停顿（Stall）机制解决数据冒险、控制竞争和结构冒险
3. 在 2. 的基础上，增加 Forwarding 机制解决数据竞争，减少因数据竞争带来的流水线停顿延时，提高流水线处理器性能
   PS：也允许考虑将 Stall 与 Forwarding 结合起来实现
4. 在 3. 的基础上，通过 predict-not-taken 策略解决控制冒险/竞争，减少控制竞争带来的流水线停顿延时，进一步提高处理器性能

2 原理分析

2.1 多周期流水线处理器的设计

多周期流水线处理器的在单周期的基础上添加了四个段寄存器，用于保存各阶段指令的参数。图为简单的 8 条指令多周期流水线处理器原理图（原九条指令中去掉 j 指令）：

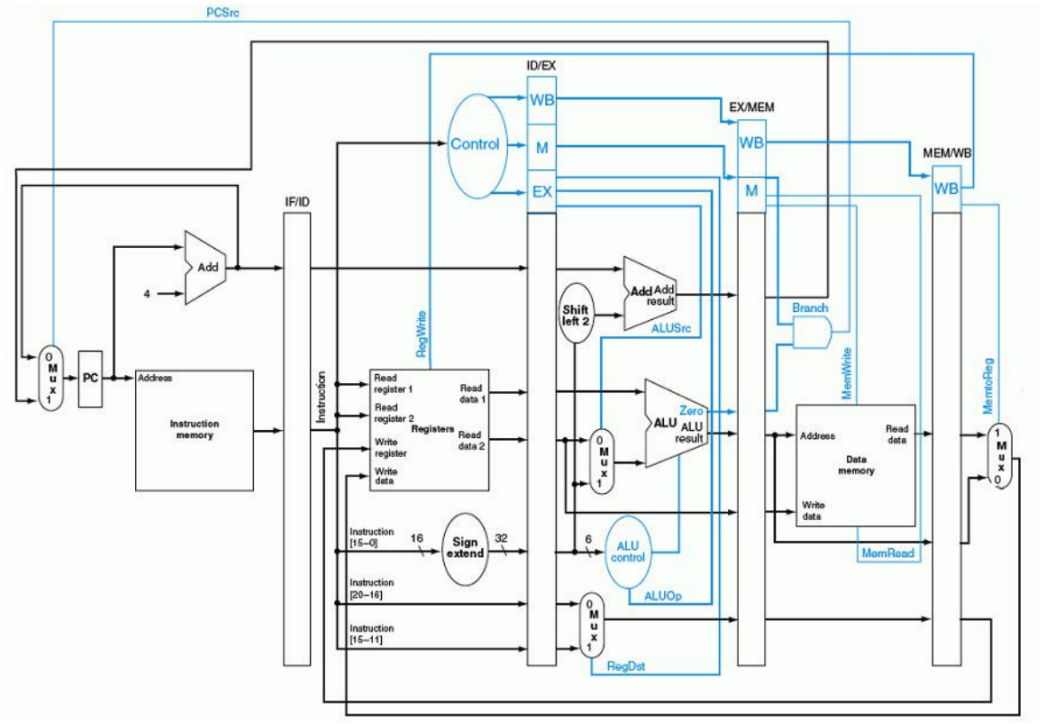

图 1: 八条指令多周期处理器原理图

每个段寄存器就像一个专属的指令缓存，把经过前面环节处理后的指令信息存储起来供本环节使用。但这样的原理图只能满足八条指令的需求，十六条指令版本的流水线原理图，请见功能实现环节。

2.2 STALL 机制的设计

STALL 机制就是将各个段寄存器暂时锁定，在一个周期内不允许其更新寄存器中的数据，也就是说，段寄存器后方对应阶段将保持当前操作的指令不变，实现流水线的停顿。

2.3 Forwarding 机制的设计

Forwarding 机制需要解决数据冒险的问题，由于流水线处理的最后一步才是将计算好的内容写回寄存器，所以可能出现在新内容还未被写回时，就已经被后续指令读取的情况。

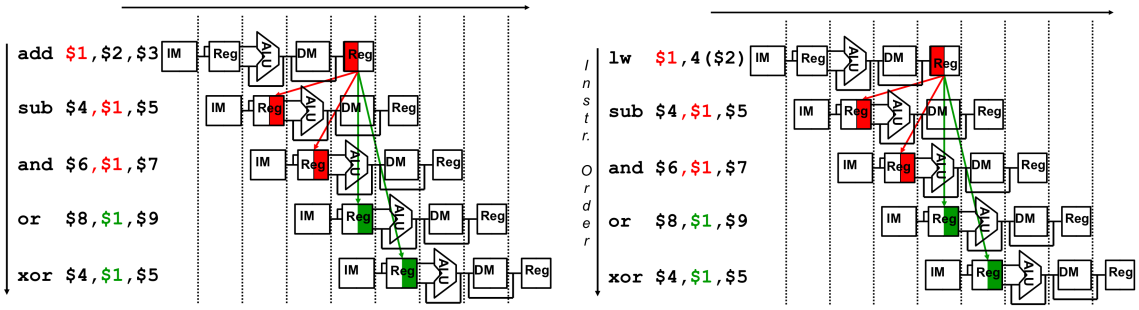

图 2: 写后读数据冒险-读后使用数据冒险

使用 Forwarding 机制，在 ALU 计算出结果后立即将发生冒险的数据回传，可以避免写后使用数据冒险；但对于读后使用数据冒险，由于在 datamemroy 段才能获得读取到的信息，所以必须先 STALL 一个周期，等到信息已经从 Datamemory 中取出后才能传回。

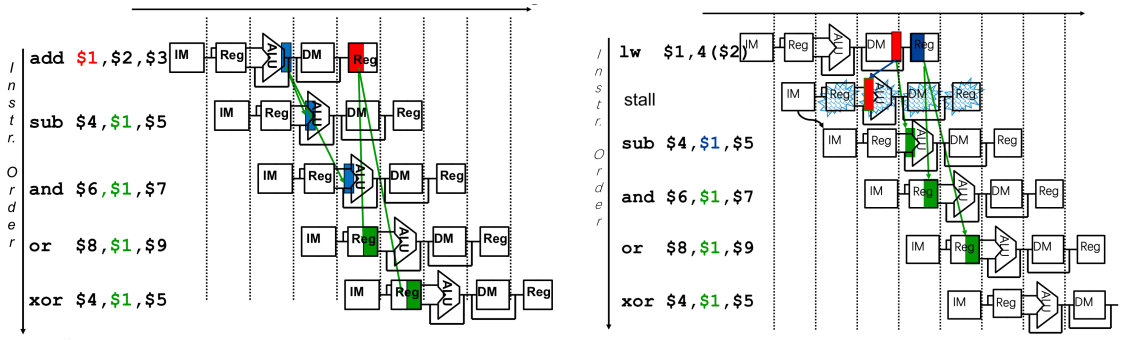

图 3: Fowarding 机制应对写后读数据冒险-读后使用数据冒险

2.4 predict-not-taken 策略的设计

对于跳转指令，如 beq，在执行过程中并不知道是否会发生跳转，如果在判断出是否会跳转前，不允许新指令进入处理器，那么会有三个周期被浪费。使用 predict-not-taken 策略，对每一次跳转都预测其不会发生，从而不中断流水线的运转，那么如果不发生跳转，就不会有周期被浪，但如果发生跳转，则随后的三条指令需要被清空。总体来说，可以减少控制竞争带来的流水线停顿，提高处理器性能。

3 功能实现

3.1 段寄存器与 STALL 机制的实现

段寄存器只用在时钟下跳沿的时候更新寄存器中的数据，并将数据传出。同时，为了满足 STALL 机制、Forwarding 机制、predict-not-taken 机制的需要，段寄存器还需要**锁定**、**清空**、**保存修改量**的功能。ID/EX 段寄存器的代码示意如下：

```
module ID_EX(
    input [31:0] read_data_1_in,    //input为下一条指令的信息，output为当前指令信息
    output [31:0] read_data_1_out   //其余信息以同样的方式处理，在此省略
    );
    reg [31:0] Read_data_1_out;
    assign read_data_1_out= Read_data_1_out;
    always @(negedge Clk) begin         //在时钟下跳沿时出发更新
        if(ID_EX_STALL==0) begin        //非STALL时才更新数据
        Read_data_1_out=read_data_1_in;
        end
    end
    always @(posedge sel_1_new) begin   //在forwarding更新数据时触发
        Read_data_1_out=new_data_1;     //更新寄存器数据
    end
    always @(reset or posedge EMPTY)begin    //在reset和清空指令时触发
        Read_data_1_out= 0;                  //清空段寄存器数据
    end
endmodule
```

这样的段寄存器，既可以满足基本的储存数据的要求，又在收到 stall 信号时实现停顿，可以在收到 forwarding 更新信号后更新数据并储存，也可以在收到 empty 信号后清空寄存器数据实现 predict-not-taken 的清空部分。

但是，EX/MEM 段寄存器的清零有所不同，由于这一阶段已经位于跳转执行的位置，所以不存在跳转后清零不再执行指令的问题，而是需要解决完成周期后 forwarding 状态的解除，所以它的清零发生在时钟下跳沿到来时，而非收到指令后立即清零。

```
module EX_MEM(
    input [31:0] read_data_1_in,    //input为下一条指令的信息，output为当前指令信息
    output [31:0] read_data_1_out   //其余信息以同样的方式处理，在此省略
    );
    reg [31:0] Read_data_1_out;
    assign read_data_1_out= Read_data_1_out;
    always @(negedge Clk) begin         //在时钟下跳沿时出发更新
        if(EX_MEM_EMPTY==0) begin        //非empty时才更新数据
        Read_data_1_out=read_data_1_in;
        end
        else begin                       //在empty信号时清空寄存器
        Read_data_1_out=0;
```

```
        end
    end
    always @ (reset )begin    //在reset和清空指令时触发
        Read_data_1_out= 0;                //清空段寄存器数据
    end
endmodule
```

3.2 16 条指令多周期流水线处理器的实现

在原先十六条指令单周期流水线的基础上，添加段寄存器并对各个区域进行划分，为了避免出现不必要的冲突，我的阶段划分如下图所示：

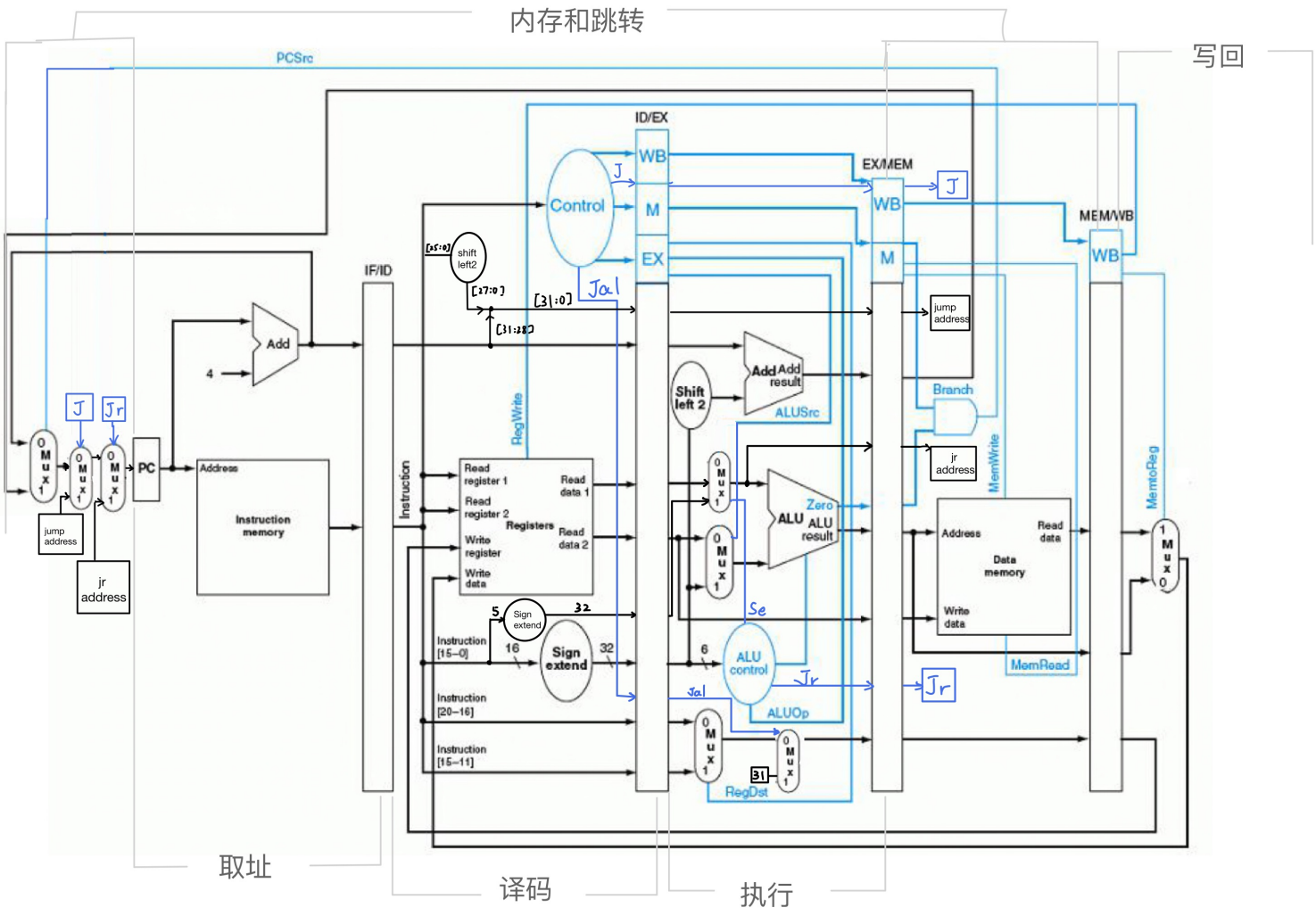

图 4: 16 条指令多周期流水线处理器原理图

五个阶段分别为：取址、译码、执行、内存与跳转、写回。取址只取址并且计算出 pc+1；译码阶对代码进行编译并完成寄存器的读取、带符号数的扩展、jump 地址的合成；执行阶段进行 alu 的选路和计算、进行写回地址的选路、进行 beq 指令跳转地址的合成；内存和写回阶段进行内存的访问、所有跳转指令的跳转；写回阶段只将数据写回寄存器。

这样在原 8 条指令流水线处理器的基础上，添加了 j、jal、jr、sll、srl、andi、addi、ori 八条指令后，也不会出现结构冒险，访问、修改、跳转的顺序都能得到保障，各个阶段分工明确，同时也便于 forwarding 策略后续模块的添加。

3.3 Forwarding 机制的实现

从前文中的图 3 可以看出，forwarding 元件要能够读取执行段正在操作哪些寄存器、内存段要把读取的数据写回哪个寄存器、以及写回段要写回哪些寄存器。所以，我设计的 forwarding 元件从 ID/EX、EX/MEM、MEM/WB 三个段寄存器获取数据，对是否发生冒险加以判断，再将来自 EX/MEM、MEM/WB 段寄存器的新数据写入 ID/EX。同时，如果发生读后用数据冒险，则对后方寄存器（ID/EX、IF/ID）和 pc 执行 STALL，让流水线停顿一个周期，同时清空 EX/MEM 寄存器避免发生无休止的数据冒险。同时，如果内存和跳转段执行的是跳转指令，则没有数据冒险会发生，只是应该将后方寄存器清零。forwarding 部分核心代码如下：

```
always @(reg_read_add_1 or reg_read_add_2 or negedge Clk) begin
    sel_1_new=0;    //新数据选择信号初始化为0
    sel_2_new=0;
    ID_EX_STALL=0;  //stall信号初始化为0
    IF_ID_STALL=0;
    EX_MEM_EMPTY=0;  //empty信号初始化为0
    EMPTY=0;
    # 10
    if(jump==1 || jal==1 || jr==1 || beq==1) begin
    //发生跳转 ，无需再forwarding
        EMPTY=1;                          //后方寄存器全部清零
        EX_MEM_EMPTY=1;                   //EX/MEM也要（延时）清零
    end
    else begin
    if(reg_read_add_1==reg_write_add_WB && regwrite_WB==1 && reg_read_add_1!=0)begin
    //发现一号位数据，存在写回段 写后读数据冒险
            new_read_data_1=reg_write_data_WB;  //输出更新后的数据
            sel_1_new=1;                        //发出更新一号数据的信号
    end
    if(reg_read_add_2==reg_write_add_WB && regwrite_WB==1 && reg_read_add_2!=0)begin
    //发现二号位数据，存在写回段 写后读数据冒险
            new_read_data_2=reg_write_data_WB;
            sel_2_new=1;                        //发出更新二号数据的信号
    end

    if(reg_read_add_1==reg_write_add_MEM && regwrite_MEM==1 && reg_read_add_1!=0) begin
    //发现一号位数据，存在内存段数据冒险
            if(memread_MEM==1)begin
            //发现是读后用数据冒险，需要STALL!
                ID_EX_STALL=1;
                IF_ID_STALL=1;
```

```
                EX_MEM_EMPTY=1;     //清空ex_mem段寄存器
            end
            else begin
            //发现是写后用数据冒险，更新数据
                new_read_data_1=reg_write_data_MEM;
                sel_1_new=1;
            end
        end
    if(reg_read_add_2==reg_write_add_MEM && regwrite_MEM==1 && reg_read_add_2!=0 )begin
    //发现二号位数据，存在内存段数据冒险
        if(memread_MEM==1)begin
        //发现是读后用数据冒险，需要STALL!
            ID_EX_STALL=1;
            IF_ID_STALL=1;
            EX_MEM_EMPTY=1;
        end
        else begin
        //发现是写后用数据冒险，更新数据
            new_read_data_2=reg_write_data_MEM;
            sel_2_new=1;
        end
    end
    end
end
```

这样 forwarding 机制就能处理好数据冒险，减少流水线停顿了。

3.4 predict-not-taken 策略的实现

predict-not-taken 策略的实现相对简单，首先，在 pc 输出的过程中，不因为出现跳转指令就停止 pc+1；第二，在确定发生跳转时，执行清空段寄存器的指令，将后方段寄存器清空，阻值指令的进行。代码的实现已经写在上方 forwarding 模块中，即发生跳转时禁用 forwarding 并清空后方段寄存器。这样每次发生 beq 不跳转的情况，就可以节省三个周期。

4 支持 16 条指令的 Mips 多周期流水线处理器的测试

首先对 instmemory 和 datamemory 中的文件进行设置。datamemory 中从 0 到 5 储存了数字 1 到 6。instmemory 中的指令如下（为方便理解，已经改变参数顺序，所有指令后第一个参数为被写入的寄存器的编号）：

```
0.j 2
1.空
2.空
3.空
4.lw $1,1($0)
5.lw $2, 2($0)
6.add $3,$1,$2
7.sub $3,$1,$2
```

```
8.or $1,$3,$2
9.and $3,$1,$2
10.slt $3,$1,$2
11.addi $1,$3,4
12.andi $1,$3,1
13.ori $1,$3,3
14.sll $2,$1,1
15.srl $2,$1,1
16.jal 22
17.lw $1,1($0)
18.beq $1,$2,7
19.空
20.空
21.sw $2,$3,2
22.jr 31
```

十六条指令均包含，并且，指令 6 发生两个位置（内存和写回段）的数据冒险，同时需要 STALL 和 forwading 机制；指令 7 发生写回段的数据冒险；指令 8 也在两个阶段，内存和写回段发生数据冒险，但应该采用内存段更新的数据替换；指令 9 发生内存段的数据冒险；指令 10 发生写回段的数据冒险；指令 18 发生读后用数据冒险，同时运用 predict-not-taken 策略连续执行后续指令，发现真的不需要跳转，sw 指令被直接完成；其他 j 型指令均执行了后续指令，但在跳转发生时后续指令被清空。

测试结果如图：

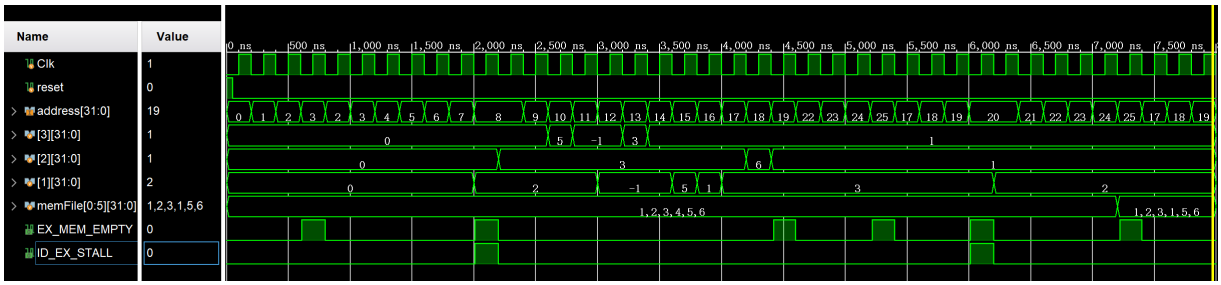

图 5: 16 条指令多周期流水线处理器测试结果

注释说明：如看不清，请放大。图中最后一行为 STALL 信号，在发生读后用数据冒险时发出，锁定后方寄存器；倒数第二行为 ex/mem 段寄存器的 empty 信号，在发生跳转或读后用数据冒险时发出，在下一个周期开始时清空 ex/mem 段寄存器；address 为指令地址，与上方提供指令对应；中间三个分别为一号寄存器、二号寄存器、三号寄存器的数据，用于观察操作结果；倒数第三行为 memfile，内存数据，只有在 sw 指令写入后发生改变。

按照指令地址编号阅读，先从地址 address 解读：任何跳转指令从取值到跳转需要四个周期（因为我的跳转部分全部设计到了内存段中），address 在 0 处执行 j 2 指令，经过四个周期跳到了 2，并且这中间每个周期指令取值都在不断自增取新的指令，实现了 predict-not-taken。随后开始顺序执行。直到 16 执行 jal22，四个周期后跳转到了 22 并将 pc+1 也就是 17 存到了 31 号寄存器中，随后 22 指令为 jr 31，即四个周期后 address 跳转回 17 随后顺序执行，到 18 指令 beq 时由于一号寄

存器和二号寄存器中数值不相同，所以没有发生向后跳转 7，predict-not-taken 策略成功，sw 指令被直接执行，没有周期被浪费。之后的 22 指令 jr 继续执行重新跳回 17，程序循环执行。

从数据角度解读，指令 4 和 5 为 lw 指令，在五个周期后完成数据的写回，1 号寄存器中被写入 2，二号寄存器中被写入 3，在 lw 执行的过程中，发生了读后用数据冒险，所以后方指令产生了一个周期停顿，但 lw 指令并不停顿。随后经历了一些列数据冒险的指令依次向一、二、三号寄存器中写入和、差、按位或、按位与、slt、加立即数 4、按位与立即数 1、按位或立即数 3、左移一位、右移一位，均能够用最新的数据进行运算并写入，同时也不需要引入停顿，实现了 forwarding。在执行 jal 指令后后续 lw 指令虽然被放入处理器运行了三个周期，但在发现跳转发生后被清空，没有改变一号寄存器的数据。跳转后通过 jr 跳转回 17 指令 lw 后一号寄存器才被修改，触发读后用数据冒险产生生一个周期的停顿，并且使 beq 不满足条件，使得 sw 指令没有被清零而是直接写入内存，没有周期被浪费。十六条指令全部正常执行。

5 总结与反思

设计实现十六条指令的多周期流水线处理器，对我来是是一次不小的挑战，从结构的设计改造，功能的理解和实现，从一开始混乱的试错，一点点认识到分区要避免的结构冲突是如何发生的，认识到数据冲突的反馈需要什么阶段执行，认识到 empty 的重要性，认识到避免死循环通过 forwarding 在阶段间发生，种种试错中我对处理器运行有了完全不同的认识，流水线平缓的流淌，支流从下游能逆转汇入上流，要避免漩涡，这样形象的概念已经出现在我的大脑中，在 debug 过程中起到了很重要的作用。再次，我也充分意识到理论对于实践、实践对于理论的相互作用，很多时候一头雾水的，就需要回到理论课老师的 ppt 中寻找解决方法，我的实验报告中使用了不少老师 ppt 上的图片，这正是我在实践过程中理解理论的痕迹。