计算机体系结构实验课程第 二 次实验报告

实验名称	静态5级流水线CPU实现			班级	李雨森老师
学生姓名	郭裕彬	学号	2114052	指导老师	董前琨
实验地点	A304、A308		实验时间	2023.10.16、10.19	

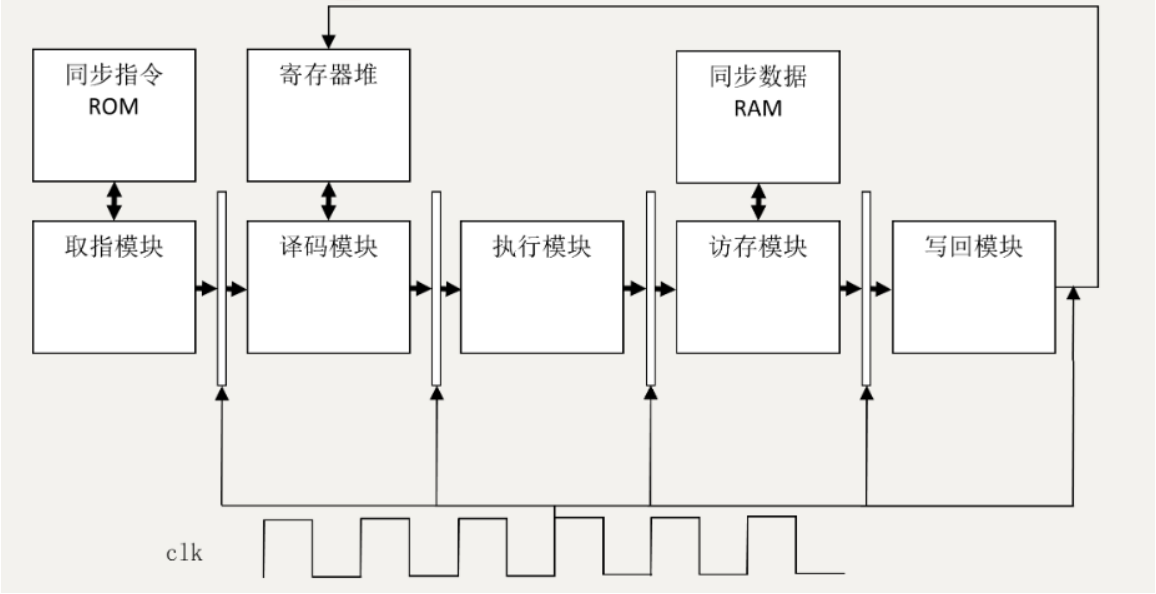
实验目的

1. 在多周期CPU实验完成的提前下，深入理解CPU流水线的概念。
2. 熟悉并掌握流水线CPU的原理和设计。
3. 最终检验运用verilog语言进行电路设计的能力。
4. 通过亲自设计实现静态5级流水线CPU，加深对计算机组成原理和体系结构理论知识的理解。
5. 培养对CPU设计的兴趣，加深对CPU现有架构的理解和深思。

实验内容说明

1. 本课程设计是对之前课程实验的拔高。前期的课程设计准备同多周期CPU的实验，主体部分可以直接使用多周期CPU实验的设计方案，但在多周期CPU中只要求实现了30多条指令，此处要求扩展到40多条指令。
   多周期CPU在单周期基础上提高了时钟频率，但并没有改善执行一条指令的时间，且存在资源闲置的问题，例如当指令在执行级有效时，译码级实际上在空转。若每一级都在执行有效的指令，将解决资源闲置的问题。
   在流水CPU中，当在一时钟周期内完成了某一条指令的全部执行时（写回级完成），则有望在下一时钟周期内完成下条指令的执行，因此依然相当于是一个周期完成一条指令，而时钟频率更高，因此CPU可以运行的更快。
   本次课程就是将上次所实现的多周期CPU更改结构为静态5级流水线CPU。

2. 本次课程设计的设计框图基本同多周期CPU实验的设计框图。需要注意5个部件都是同时运转的，但对每条指令而言，依然是依次工作的。



3. 本次课程设计的关键和难点在于流水线的控制，比如一条指令何时可以从译码级进入执行级，一条指令何时需要堵在流水线中。本次课程设计暂不考虑前递技术，因此有数据相关时就需要堵在流水线中。

4. MIPS架构中有延迟槽的设定，其本意是加快流水CPU的执行速度。在之前单周期和多周期CPU实验中未支持延迟槽，但在流水CPU中需要支持该设定，因为只有硬件支持了延迟槽技术，用通用编译器编译出来的MIPS二进制执行文件才能在自己设计的CPU中运行正确。

5. MIPS架构中分支和跳转指令参与计算的PC值均为延迟槽指令对应的PC(即分支跳转指令的PC+4),在本课程设计中尤其需要注意这一点。比如一条指令“beq,r0,r0,#2”在不考虑延迟槽的多周期CPU中，其跳转的目标地址为beq指令后面的第2条。而在考虑延迟槽的流水CPU中，其跳转的目标地址为beq指令后面的第3条（即延迟槽指令后面的第2条）。在编写测试程序时就需要注意分支跳转指令的偏移量。

6. 根据设计的实验方案，使用verilog编写相应代码。

7. 对编写的代码进行仿真，得到正确的波形图。

8. 将以上设计作为一个单独的模块，设计一个外围模块去调用该模块，见图9.3。外围模块中需调用封装好的LCD触摸屏模块，观察CPU的内部状态，比如32个寄存器的值，各级PC的值等。并且需要利用触摸功能输入特定数据RAM地址，从该RAM的调试端口读出数据显示在屏上，以达到实时观察数据存储器内部数据变化的效果。通过这些手段，可以在板上充分验证CPU的正确性。

9. 将编写的代码进行综合布局布线，并下载到实验箱中的FPGA板子上进行演示。注意：一般而言控制CPU运转的时钟是由FPGA板上的时钟输出提供的，但为了方便演示我们需要在每一个时钟里查看一条指令的运算结果，故演示时理想的时钟是手动输入的，可以使用FPGA板上的脉冲开关代替时钟。

实验原理：分析总结多周期CPU如何改进成五级流水线CPU

1. 并行优化：在多周期CPU的设计中，在同一个时间周期内只有某一级的结构（IF、ID、EXE、MEM、WB）的Valid信号根据记录当前状态的state的取值置位为有效；而流水线CPU的valid信号允许在同一时间内设置多个阶段为有效，从而使得多条指令可以在不同阶段同时并行执行。
2. 增加数据传送指令和特权指令：这一部分不是必须的操作，添加的指令主要是为了实现更多的功能以及完善CPU控制体系，以期实现系统调用和异常返回等底层交互功能。
3. 增加延迟槽以匹配编译器编译体系：在高级的流水线CPU设计中，有比较完善的预测体系来保证延迟槽中的指令与是否跳转无关，但本实验只是简单地将延迟槽定义为跳转指令下一条指令，并使其执行，故需要在测试程序设计时手动调整指令位置以保证不出现影响。

```
//bd_pc,分支跳转指令参与计算的为延迟槽指令的PC值，即当前分支指令的PC+4
wire [31:0] bd_pc; //延迟槽指令PC值
assign bd_pc = pc + 3'b100;
```

4. 增加多个控制信号：不同于多周期CPU使用state来设置valid信号，流水线CPU在顶层添加了各种信号用于控制valid信号，

```
// syscall和eret到达写回级时会发出cancel信号，
    wire cancel;    // 取消已经取出的正在其他流水级执行的指令

    //各级允许进入信号:本级无效，或本级执行完成且下级允许进入
    assign IF_allow_in  = (IF_over & ID_allow_in) | cancel;
    assign ID_allow_in  = ~ID_valid  | (ID_over  &
EXE_allow_in);
    assign EXE_allow_in = ~EXE_valid | (EXE_over &
MEM_allow_in);
    assign MEM_allow_in = ~MEM_valid | (MEM_over &
WB_allow_in );
    assign WB_allow_in  = ~WB_valid  | WB_over;

    //IF_valid，在复位后，一直有效
   always @(posedge clk)
    begin
        if (!resetn)
        begin
```

```
            IF_valid <= 1'b0;
        end
        else
        begin
            IF_valid <= 1'b1;
        end
    end

    //ID_valid
    always @(posedge clk)
    begin
        if (!resetn || cancel)
        begin
            ID_valid <= 1'b0;
        end
        else if (ID_allow_in)
        begin
            ID_valid <= IF_over;
        end
    end

    //EXE_valid
    always @(posedge clk)
    begin
        if (!resetn || cancel)
        begin
            EXE_valid <= 1'b0;
        end
        else if (EXE_allow_in)
        begin
            EXE_valid <= ID_over;
        end
    end

    //MEM_valid
    always @(posedge clk)
    begin
        if (!resetn || cancel)
        begin
```

```
            MEM_valid <= 1'b0;
        end
        else if (MEM_allow_in)
        begin
            MEM_valid <= EXE_over;
        end
    end

    //WB_valid
    always @(posedge clk)
    begin
        if (!resetn || cancel)
        begin
            WB_valid <= 1'b0;
        end
        else if (WB_allow_in)
        begin
            WB_valid <= MEM_over;
        end
    end
```

4. 添加了各类交互信号，用来在关联阶段判断是否存在数据相关，进而是否需要堵塞。

```
    //IF与inst_rom交互
    wire [31:0] inst_addr;
    wire [31:0] inst;

    //ID与EXE、MEM、WB交互
    wire [ 4:0] EXE_wdest;
    wire [ 4:0] MEM_wdest;
    wire [ 4:0] WB_wdest;

    //MEM与data_ram交互
    wire [ 3:0] dm_wen;
    wire [31:0] dm_addr;
    wire [31:0] dm_wdata;
    wire [31:0] dm_rdata;
```

```
    //ID与regfile交互
    wire [ 4:0] rs;
    wire [ 4:0] rt;
    wire [31:0] rs_value;
    wire [31:0] rt_value;

    //WB与regfile交互
    wire        rf_wen;
    wire [ 4:0] rf_wdest;
    wire [31:0] rf_wdata;

    //WB与IF间的交互信号
    wire [32:0] exc_bus;
```

5. 流水级周期数优化：由于没有旁路的设计，流水线的一些处理会存在流水级之间的冲突，需要对某些流水级的持续周期数做一些额外的设计。

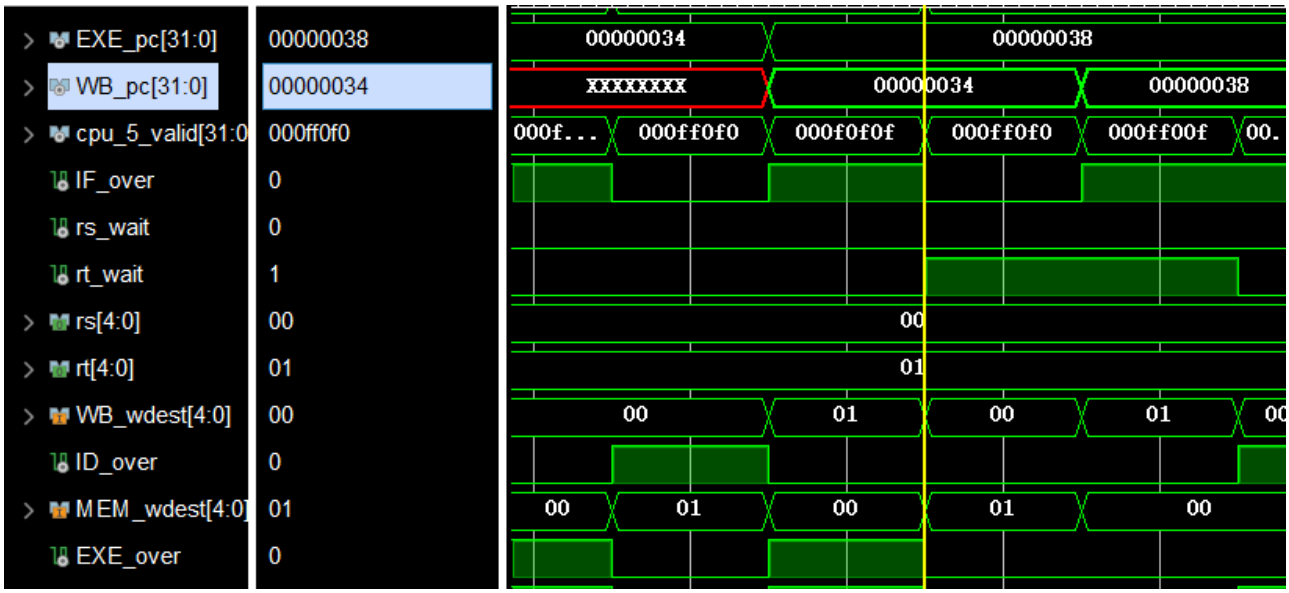
实验步骤

由于实验使用的是同步IP核形式的存储器，存在着一些时钟问题和分支跳转问题，尝试对其进行分析修复。

时钟问题

在上箱过程中，发现第二条指令38H对应的sll $2,$1,#4并没有在38H结束WB阶段后把2号寄存器的值变为10H，而是在3CH结束WB阶段后才写回，且3CH的功能，即把3号寄存器的值变为11H，并没有实现。

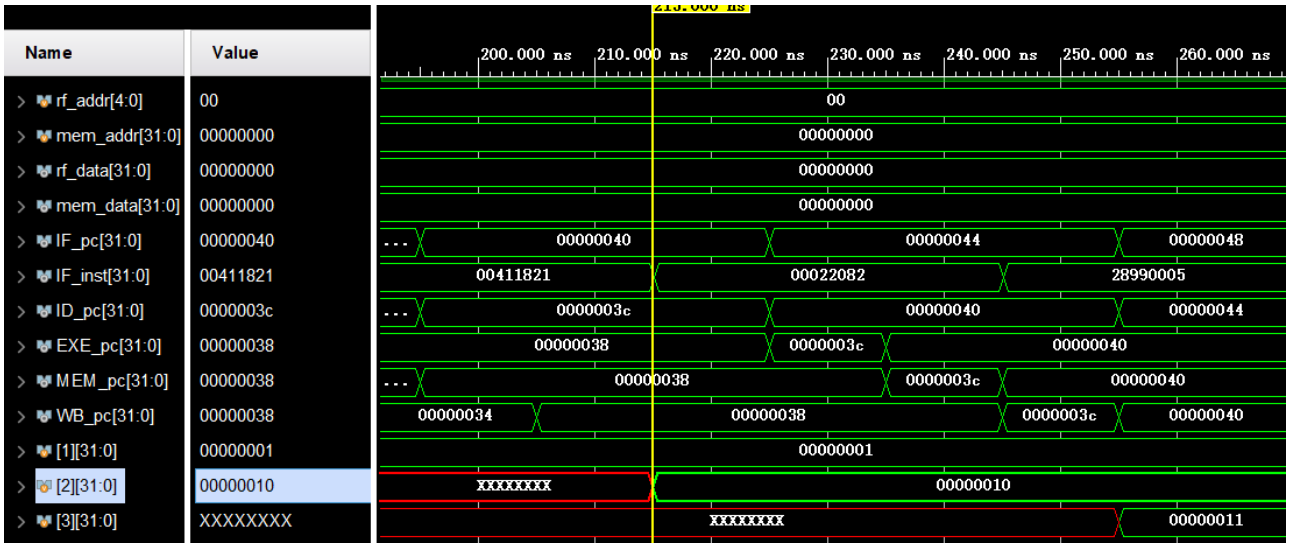
在仿真环境下，监测ID_over信号，原本的设计中，ID阶段的38H指令中的rt寄存器为1号寄存器，此时WD阶段的34H指令中写寄存器wdest也是1号寄存器，ID阶段的rt_wait信号应为1，也就是说，对应的ID_over为0，下一个周期的ID阶段应设置为无效，而EXE_valid信号的赋值通过`always(@posedge clk)`实现，其内部的赋值语句使用的是`<=`符号，故数据的更新具有延后性，接下来的一个周期接受的是再前一个周期传来的ID_over信号，使得EXE_valid信号变为有效，指令38H进入到EXE阶段。



于是尝试通过给各个阶段增加一个时钟周期以读取到更新后的控制信号，这一过程在IF阶段实现以影响所有阶段。

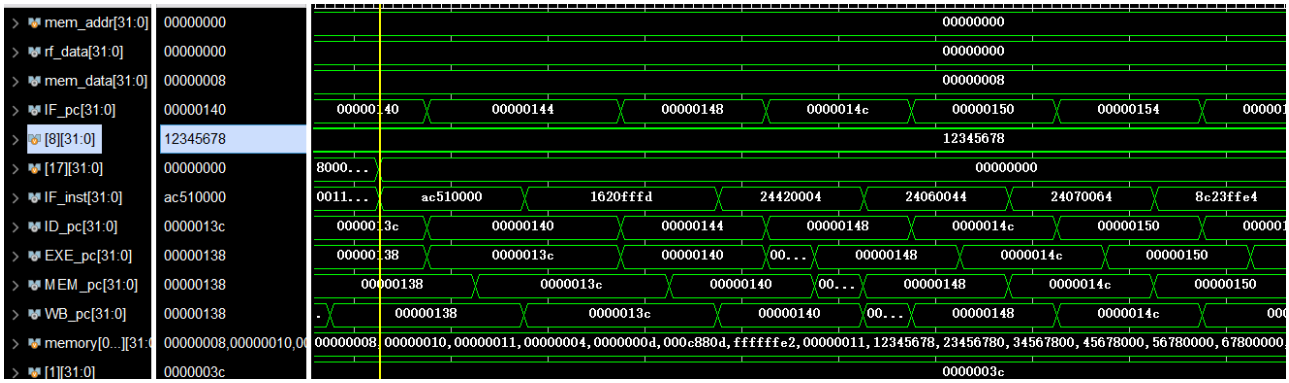
```
reg temp;
always @(posedge clk)
    begin
        if (!resetn || next_fetch)
        begin
            IF_over <= 1'b0;
            temp <= 1'b0;
        end
        else
        begin
            IF_over <= temp;
            temp <= IF_valid;
        end
    end
```

这一修改使得IF阶段需要三个时钟周期来完成，同时影响之后的所有阶段，通过仿真实验观察，2号寄存器能够在正确的时候被赋值，并且3号寄存器也能够正常赋值。



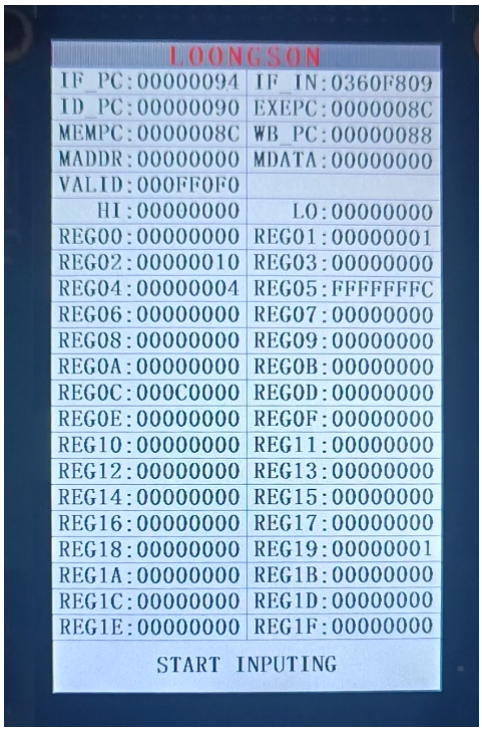
跳转问题

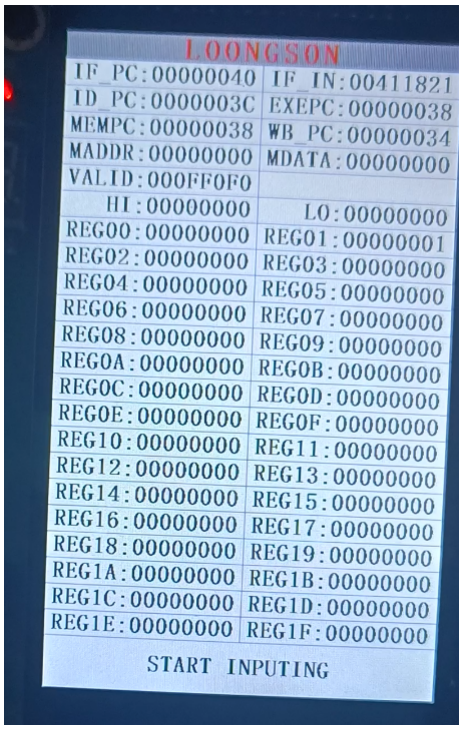
完成修改后继续仿真实验，发现程序运行到144H时，程序设计的是根据$17的值，会跳转到13CH循环数次后才不跳转，而实际运行时$17的值却是00000000H，使得一次跳转都不会发生，进而导致后面的运行错误。往前追溯，在138H时$17本应被赋值为1234_5678H，值的来源是$1-28，此时$1存的值为0000_003CH，查看对应内存地址，发现0000_0020H存放的数据也是1234_5678H，查看lw指令的相关使用，发现在此之前都是正向偏移，此处为第一次使用偏移量为负的lw指令，初步判断是指令实现时出现了相关的错误，导致取到了某处存放的0000_0000H的值。

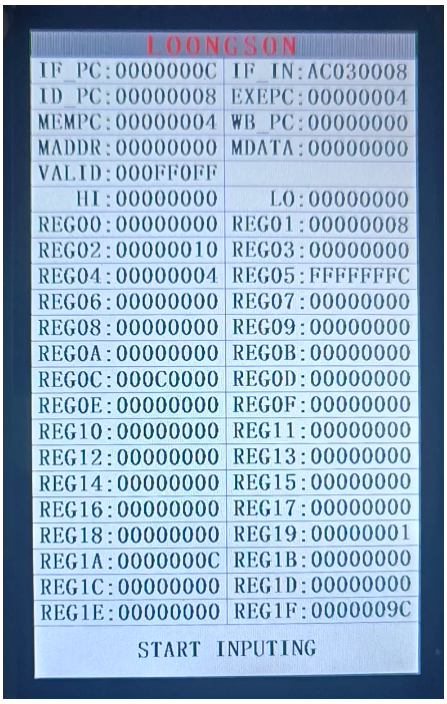


受时间因素影响，此处只将指令修改为`lui $17,#1234H 3C111234`，使得匹配后面的相关位移运算指令，观察运行结果，注意到程序能够成功跳转回34H，实验成功。对于指令内部运行的相关测试和修改，留待下一次实验进行。

实验结果


LOONGSON
IF_PC:00000094 IF_IN:0360F809
ID_PC:00000090 EXEPC:0000008C
MEMPC:0000008C WB_PC:00000088
MADDR:00000000 MDATA:00000000
VALID:000FF0F0
HI:00000000 LO:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000000
REG04:00000004 REG05:FFFFFFFC
REG06:00000000 REG07:00000000
REG08:00000000 REG09:00000000
REG0A:00000000 REG0B:00000000
REG0C:000C0000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000001
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING


LOONGSON
IF_PC:00000040 IF_IN:00411821
ID_PC:0000003C EXEPC:00000038
MEMPC:00000038 WB_PC:00000034
MADDR:00000000 MDATA:00000000
VALID:000FF0F0
HI:00000000 LO:00000000
REG00:00000000 REG01:00000001
REG02:00000000 REG03:00000000
REG04:00000000 REG05:00000000
REG06:00000000 REG07:00000000
REG08:00000000 REG09:00000000
REG0A:00000000 REG0B:00000000
REG0C:00000000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING



实验中的修改都在仿真环境下进行，实验箱上箱时使用的是原始文件，故省略对结果的分析。

优化改进

1. 分支预测：分支预测能够通过一些机制猜测分支指令跳转与否，当预测正确时即可使流水线省去等待的时间，预测失败则会产生错误指令的撤销。可以采用理论课上学到的一级、二级饱和计数器或者混合分支预测等方法来实现。例如比较完善的二级饱和计数器，采用2bits的PHT来记录历史状态和进行预测。
2. 冒险处理：数据冒险发生时，根据情况有数据冲突产生时，可以使用数据前推或旁路、指令调度的方式来继续处理。
3. 延迟槽设计：本次实验延迟槽的设置固定为下一条指令，这会使得程序设计时需要格外注意指令的相关性和影响，因此可以设计一些优化算法，一个简单的优化方式是使用不同的调度方式，但值得思考的是这一改进能否通过硬件设计而非软件来实现。
4. 在上述实验中，只是简单地通过在IF阶段增加一个时钟周期来实现同步IP核的存储器CPU的正常工作，更进一步的优化思路是细化到不同的指令来进行不同的设置。