五级流水线CPU实现

姓名：高祎珂　　学号：2011743

实验概述

上次实验中已经实现了多周期流水线，本次实验目的即为在上一实验得到的多周期 CPU 基础上实现五级流水线 CPU。上次实验实现的多周期 CPU 中，将一条指令的周期分为了五个阶段分别为取指（IF）、译码（ID）、执行（EXE）、访存（MEM）、写回（WB）。本次实验中，依然将一条指令的执分为五个阶段，应用流水线的原理，最大化利用硬件资源，优化 CPU 性能。

在每个新的时钟周期开始时，根据 PC 寄存器内的内容读入下一条指令放入流水线内，前4条指令进行下一阶段，其中最后一条指令完成写回阶段并返回。

本次实验依旧利用 vivado 平台及 virlog 语言，并利用实验箱动手验证，探究设计原理。

实验原理

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。最理想的情况是，当第一条指令从取指级转换到下一级译码时，第二条指令进入取指级，当第一条指令完成译码进入执行级时，第二条指令进入译码级，第三条指令进入取指级……静态 5 级流水 CPU 就是基于这样的设计思路。

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。

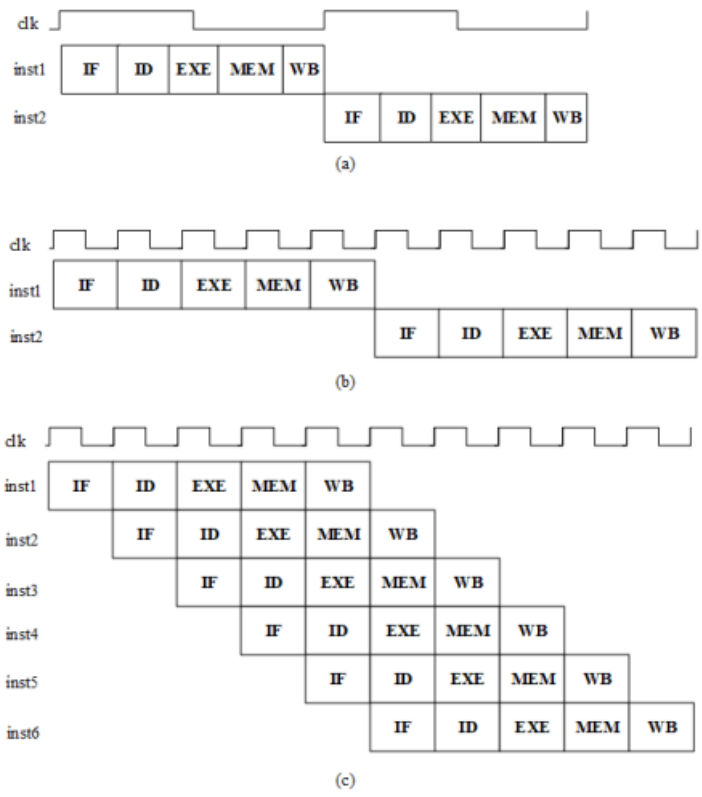
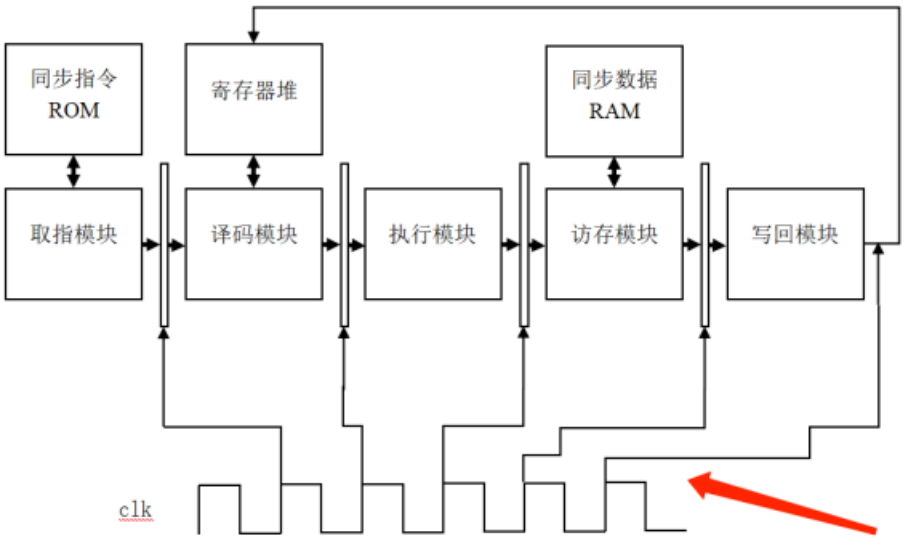
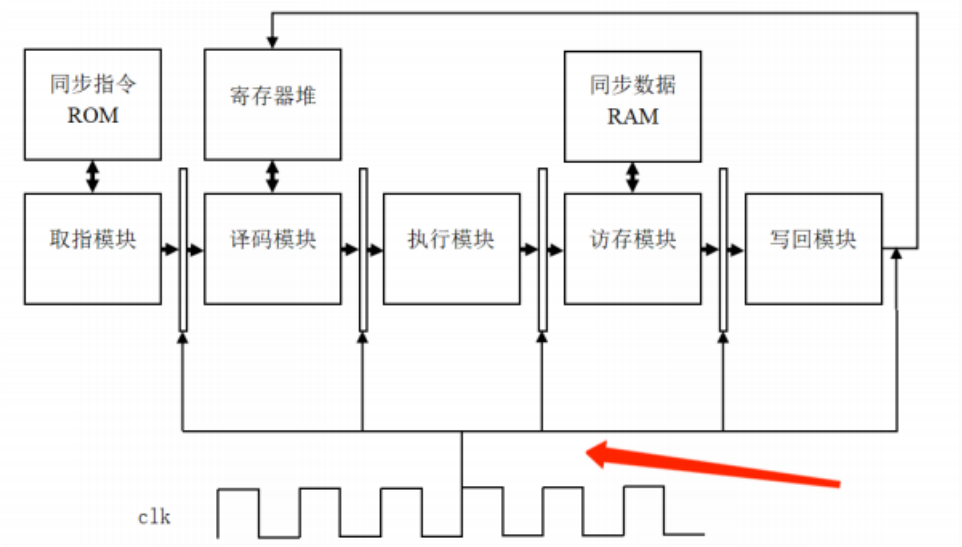


图 1: a) 单周期 b) 多周期 c) 五级流水线

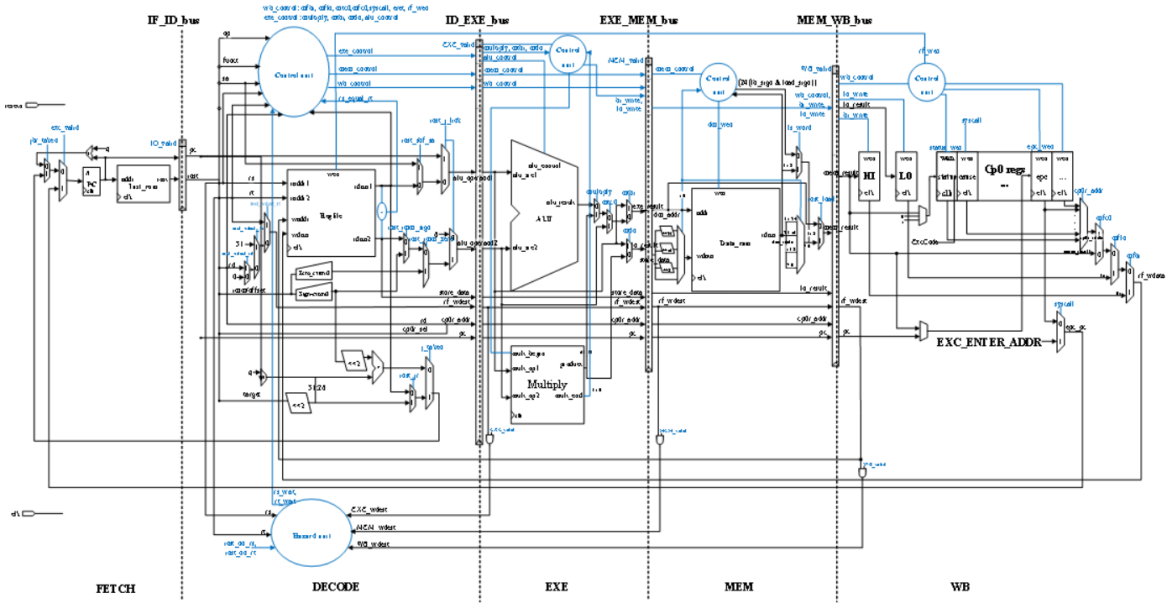
对比而言能够更加清晰地表述流水线 CPU 的原理，故先言多周期 CPU 的工作原理。大致来说，多周期 CPU 的工作过程就是：先利用 PC 取出指令放到指令寄存器中（IF），从指令中得到控制器需要的信号、使能读入写地址的信号跳转指令地址以及立即数等并分析出计算需要的寄存器（ID），之后使用分析出指令所需寄存器的值交由 ALU 配合控制器使能信号进行运算。若指令类型为跳转，则将计算的地址写回 PC，在下一条指令执行时进行跳转；若指令类型为运算等，计算出的结果交由数据存储器，在相应的位置写入数据（MEM）或者写回寄存器读取的数据（WB）。原理大致示意图如图



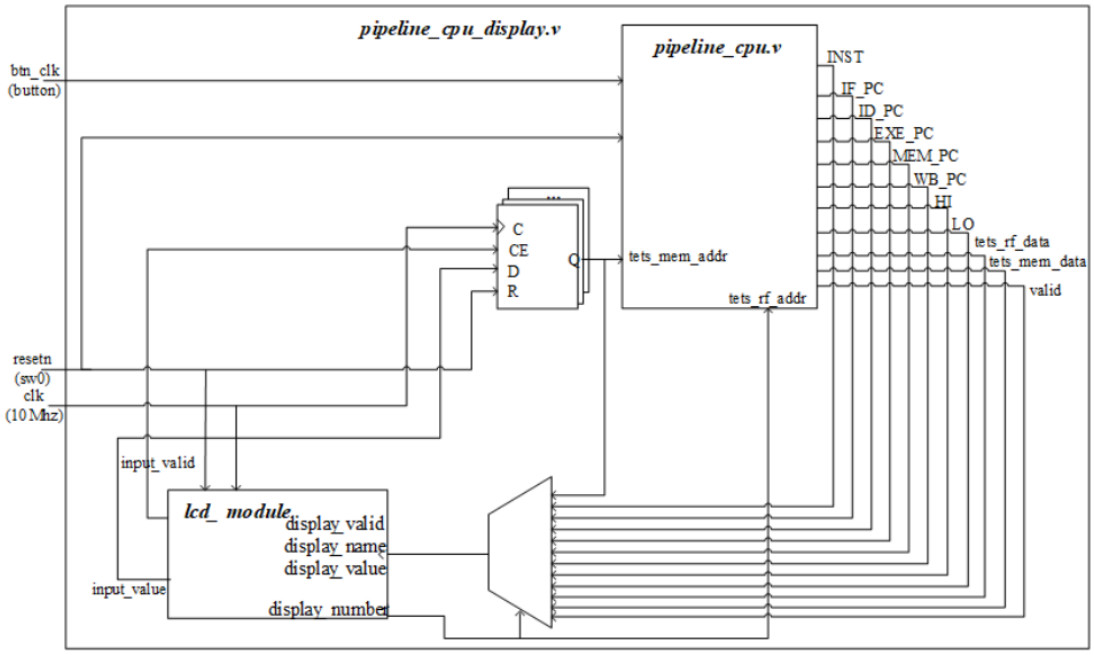
而本次实验的五级流水线 CPU 中，充分利用了硬件资源。多周期 CPU 在执行一条指令的某一阶段时，其他阶段执行所需的硬件资源处于等待状态，而流水线 CPU 中可以面向处理阶段而非指令进行工作，在每个周期都可以读入一条新的指令，并推进已读入 CPU 中的指令的运行进入下一阶段。这样，在每个时钟周期内，CPU 中最多可以包含五条指令，这大大提高了 CPU的性能。原理大致示意图如下图，发现了时钟周期对应执行阶段部分的区别：



本次实验仍然主要通过已经写好的测试指令来观察 CPU 运行过程。也就相当于已经将指令写入了指令存储器，等待 PC 指向其时取出，进入 CPU 执行。CPU 设计原理具体实现如图：

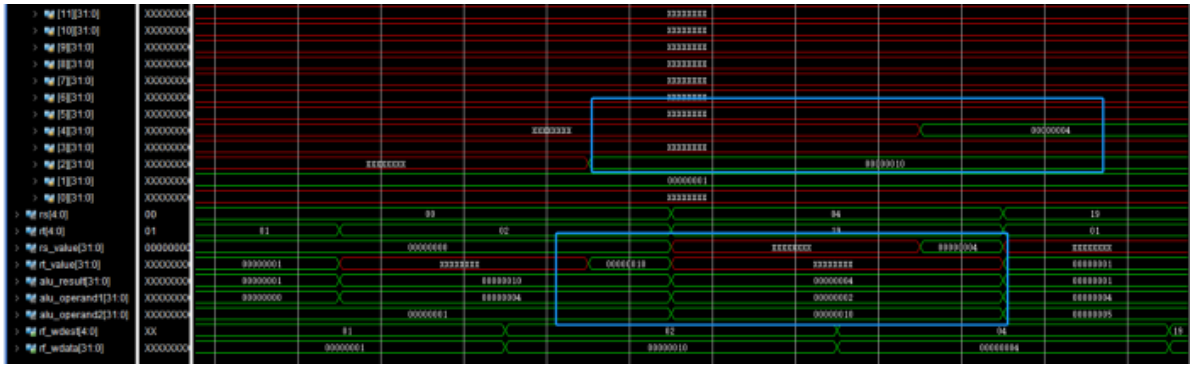


本实验的顶层模块图如下图所示：

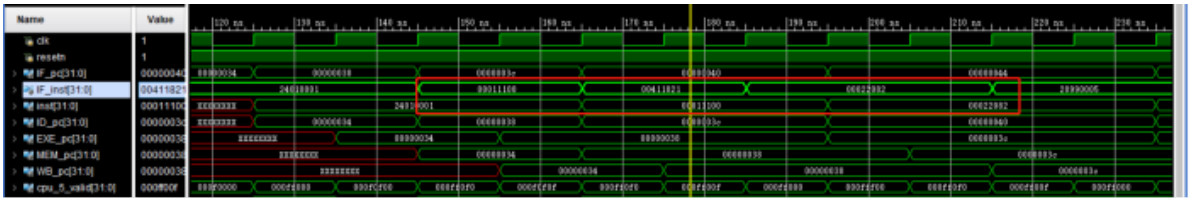


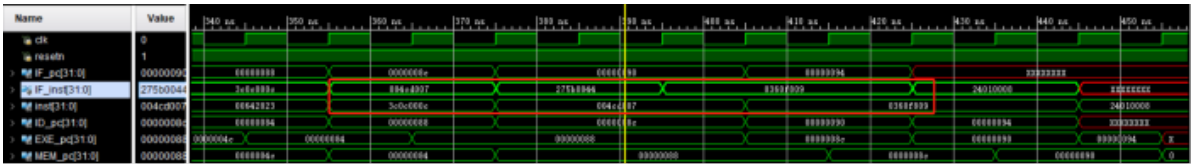
代码修改

运行给的代码，发现bug：3CH指令的结果没有被写入寄存器堆，且该指令对应的运算过程ALU根本就没有执行。



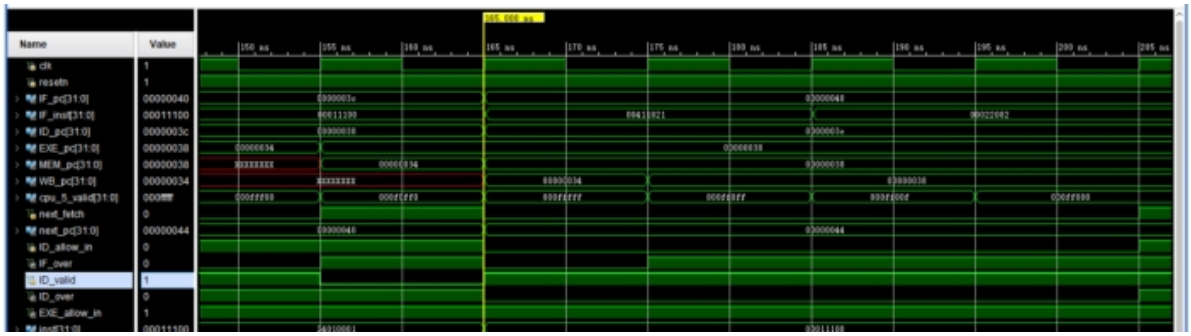
进一步发现，PC阶段获取的一些指令没有被正常的传给ID阶段，如图所示：





一开始我认为是ID级检测到数据依赖而等待执行结果写回，但这时IF级还在执行，从而导致ID级等待时IF级获取的指令被后面的指令覆盖了。也就是说，在ID级等待的过程中，IF级也应该等待。具体说就是不要产生next_fetch信号，不使pc值更新。

然而问题在于，在ID级尚未解析到产生依赖的指令时，IF级就已经收到了nextfetch，这将导致PC+4，也就使两个周期后IFinst被更新一次，而这时ID级还在等待依赖指令执行完毕，这就导致了有一条指令被覆盖（下图00411821）。既然这种需要ID级“预知未来”的思路不可靠，那么能不能修改IF级来达到同样的目的呢?



通过进一步观察发现，IF级和ID级之间的总线上，ID级能够看到的IFIDbusr信号的地址部分与信号部分总是不对应的，这势必导致运行的错误。观察波形不难发现，instrom需要在给出地址后等待两个周期后才能给出正确的指令，可实际上IF级只等待了一个周期。

因此更改方法为：可以将IFover信号锁存一次，使IF级结束延迟一个周期；

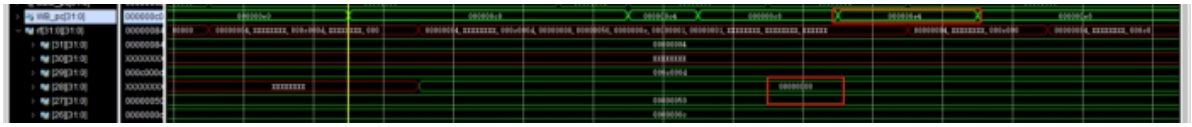
更改代码如下图所示：

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

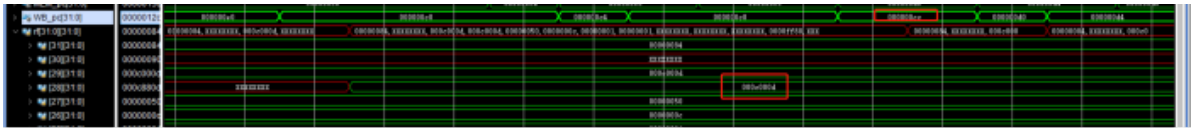
```
            IF_over<=IF_valid;
        end
    end
end
```

修改后使得总线传输重回正常，如下图所示：

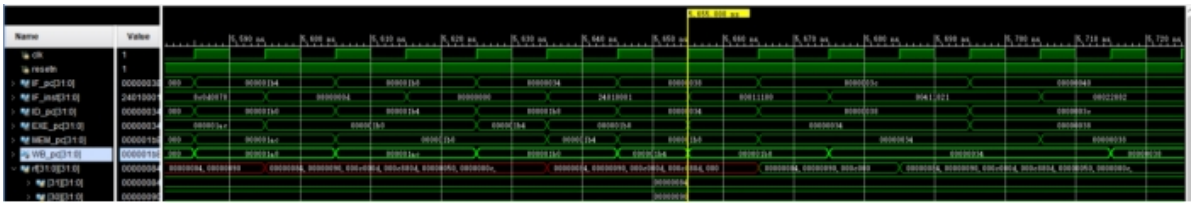
运行后续程序发现，在第一次运行C0H指令时，$28寄存器没有按照预期被改写为000C000DH，而是被刷成了全零，如下图所示，这将导致在C4H指令提前发生跳转，导致错误。怀疑仍然是IP核读写太慢导致的。可以去掉dataram两个读写端口上的Primitives Output Register；



修改后，$28寄存器结果正确且C4H指令第一次跳转、第二次不跳转，运行正常。



此时最后一条指令从WB级退出总时间为5655.0ns



一些改进：加入数据前递并对空泡的插入进行调整

为了能够实现数据前递并方便设计，我们首先设计出冒险检测单元。根据课上学到的数据依赖判断条件，我们的冒险检测单元需要获取如下的信号：

EXE级的寄存器堆取数地址Rs、Rt，MEM级写入地址RdM（已经将取到的数放入了这个寄存器），

WB级写入地址RdW（这主要是考虑到访问寄存器堆在ID级，我们可以将RdW的数据前递到EXE级），

RegWriteM和RegWriteW为1bit数据，意为MEM级、WB级是否写回寄存器堆，用于辅助判断，

RdMUnload意为EXE级发往MEM级的写入地址（此时将要取到的数放入这个寄存器），

noForward为1bit数据，意为是否要关闭前递。

最终冒险检测单元的输入输出为如下：

```
module harzard(
    input wire clk,
    input wire resetn,

    input wire [4:0] RsE,
    input wire [4:0] RtE,
```

```
    input wire [4:0] RdM,
    input wire [4:0] RdW,

    input wire RegWriteM,
    input wire RegWriteW,

    input wire RdMUnload,
    input wire noForward,

    output wire [1:0] Forward1E,Forward2E,
    output reg shouldStall=1'b0
)
```

输出信号Forward1E、Forward2E为控制ID级和EXE级的MUX的两位信号，用于在原始信号、EXE级前递信号和MEM级前递信号间进行选择。shouldStall用于在MEM读数尚未开始、EXE就要使用的情况下使流水线停顿一拍。

冒险检测单元的判断逻辑如下图所示：

```
reg noForward_r = 1'b0;
always @(posedge clk) begin
    noForward_r <= noForward;
end
always@(clk) begin
    Forward1E[0] <= Rdw != 0 && RegWritew && (Rdw == RsE) && ~(RegWriteM && (RdM
== RsE));
    Forward1E[1] <= RdM != 0 && RegWriteM && (RdM == RsE);// && ~noForward_r;
    Forward2E[0] <= Rdw != 0 && Regwritew &&(RdW == RtE) && ~(RegWriteM && (RdM
== RtE));// && ~noForward_r;
    Forward2E[1] <= RdM != 0 && RegWriteM && (RdM == RtE);// && ~noForward_r;
end
reg shouldStall_r = 1'be;
always @(posedge clk) begin
    if (!shouldSta11)
    begin
        if(|RdMUnload && ~shouldStal1_r)
        begin
            shouldStal1 <= |RdMUnload && (RsE == RdMUnload || RtE == RdMUnload);
            shouldStal1_r <= 1'b1;
        end
        else
        begin
            shouldStal1 <= 1'be;
            shouldStal1_r <= 1'be;
        end
    end
    else
    begin
        if(RsE == RegWritew || RtE == Regwritew)
        begin
            shouldStal1 <= 1'b0;
        end
    end
end
```

冒险检测单元的实际输入如图所示：

```
harzard HZ_module(
    .clk(clk ), // I,1
    .resetn(resetn ),
    .RsE(rs),
    .RtE(rt),
    .RdM(MEM_wdest),
    .Rdw(WB_wdest),
    .RegWriteM(|MEM_wdest),
    .RegWritew(|WB_wdest),
    .RdMUnload(unloadRdM),
    .noForward(noForward),
    .Forward1E(Forward1),
    .Forward2E(Forward2),
    .shouldStal1 (shouldStal1)
);
```

为配合数据冒险检测，其他流水级进行的改动如下。

对ID级，因为其要使用rsvalue和rtvalue判断是否跳转以及为EXE级提供数据来源，故需要将前递数据引入ID级；同时需要将MUX控制信号和流水线停顿信号引入ID级，故添加的数据输入如下：

```
.ForwardData1  (exe_exe),
.ForwardData2  (mem_exe),
.ForwardMux1  (Forward1),
.ForwardMux2  (Forward2),
.shouldStal1 (shouldStal1),
```

对EXE级，增加的输入信号及其原因与ID级类似，增加了输出信号exe_exe，这是将EXE级的运算结果进行前递，增加了unloadRdM，这是将EXE级发往MEM级的写入地址输出用于流水线停顿判断。

```
.ForwardData1  (exe_exe),
.ForwardData2  (mem_exe),
.exe_result     (exe_exe),
.ForwardMux1  (Forward1),
.ForwardMux2  (Forward2),
.shouldStal1    (shouldStal1),
.unloadRdM      (unloadRdM),
```

下面叙述在实验探索过程中，逐步完善设计的过程。

在对EX级结果进行前递时，发现出现了组合环路无法进一步仿真。探索发现，将“任何一个信号变化即触发（*）”改为时钟上下跳沿皆触发（clk）即可，同时注意不要把aluoperand1、2自己赋给自己，而是从IDEXEbusr选择正确的位数赋值，如下图所示。

```
reg [31:0] alu_operand1;
reg [31:0] alu_operand2;
always @(clk) begin
    if(ForwardMux1 == 2'b0) begin
        alu_operand1 <= ID_EXE_bus_r[150:120];
    end
    else if(ForwardMux1 == 2'b1) begin
        alu_operand1<= ForwardData2;
```

```
    end
    else
    begin
        alu_operand1<= ForwardData1;
    end
    if(ForwardMux2 == 2'be) begin
        alu_operand2 <= ID_EXE_bus_r[119:88];
    end
    else if(ForwardMux2 == 2'b1) begin
        alu_operand2 <= ForwardData2;
    end
    else begin
        alu_operand2 <= ForwardData1;
    end
end
```

两个operand如果被赋值两次会出现问题，如下图所示，第一次赋值改成接地（none12是悬空的线）。

```
wire [31:0] none1,none2;
assign {multiply,
    mthi,
    mtlo,
    alu_control,
    none1,
    none2,
    mem_control,
    store_data,
    mfhi,
    mflo,
    mtc0,
    mfc0,
    cper_addr,
    syscall,
    eret,
    rf_wen,
    rf_wdest,
    pc    } = ID_EXE_bus_r;
```

为了减少指令执行周期，修改后ID处插入空泡为仅考虑控制冒险的情况。如下所示。

```
assign ID_over=ID_valid &(~inst_jbr|IF_over);
```

此时发现48H跳转指令无法正常执行，这是因为这条指令使用了上一条计算指令的结果，但由于没有给ID级加前递，此时尚无法决定是否要跳转，于是应给ID级加上前递，如下图所示。更改后，发现能够正常跳转。

```
reg [31:0] rs_value;
reg [31:0] rt_value;
always @(clk) begin
    if(ForwardMux1 == 2'b0) begin
        rs_value <= rs_value_i;
    end
    else if(ForwardMux1 == 2'b1) begin
        rs_value <= ForwardData2;
```

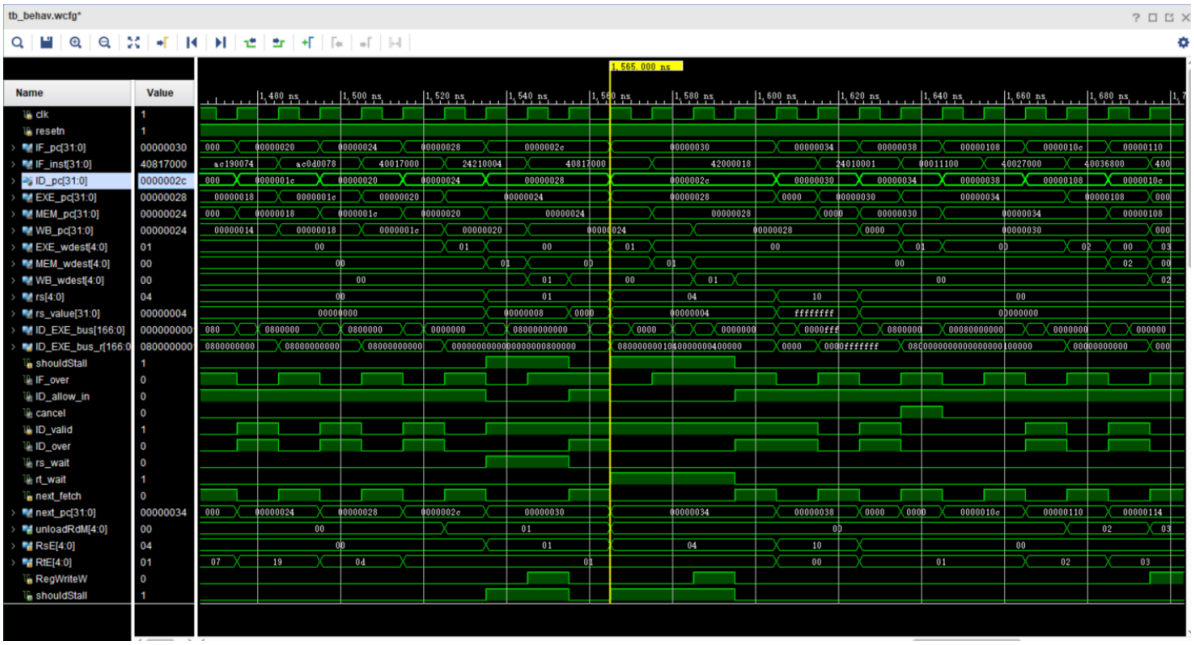
```
    end
    else begin
        rs_value <= ForwardData1;
    end
    if(ForwardMux2 == 2'b0) begin
        rt_value <= rt_value_i;
    end
    else if(ForwardMux2 == 2'b1) begin
        rt_value <=ForwardData2;
    end
    else begin
        rt_value <= ForwardData1;
    end
end
```

之后发现无法从异常处理函数中跳出，观察发现，在2CH指令执行时两个前递都被打开，然而28H的mfc0指令的结果并非是用ALU算出的，打开前递反而得不到正确的值。此时让EXE级等待任何意义也没有，因为ID给的还是旧的值，应该让ID级等、重新从寄存器堆里面取值。

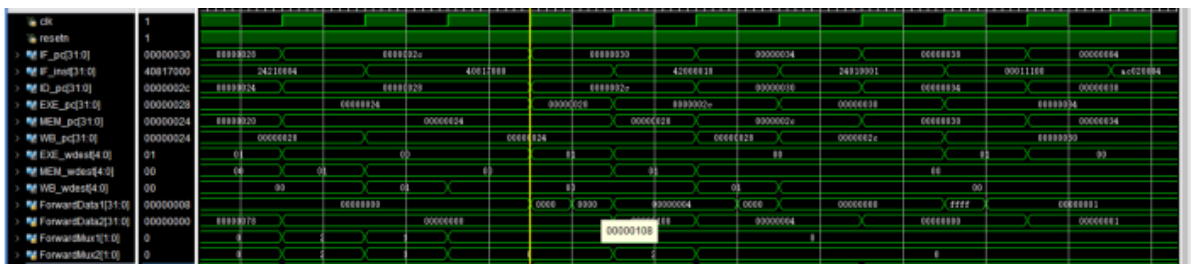
通过反思mfc0指令格式发现，该指令和必须使ID级等待一拍的情况一致，只不过是数据的来源从内存变为了cp0寄存器。

为了使该情况能够被正确处理，也就是使流水线等待一个周期，我们对ID级结束条件进行如下修改，增加shouldStall条件：

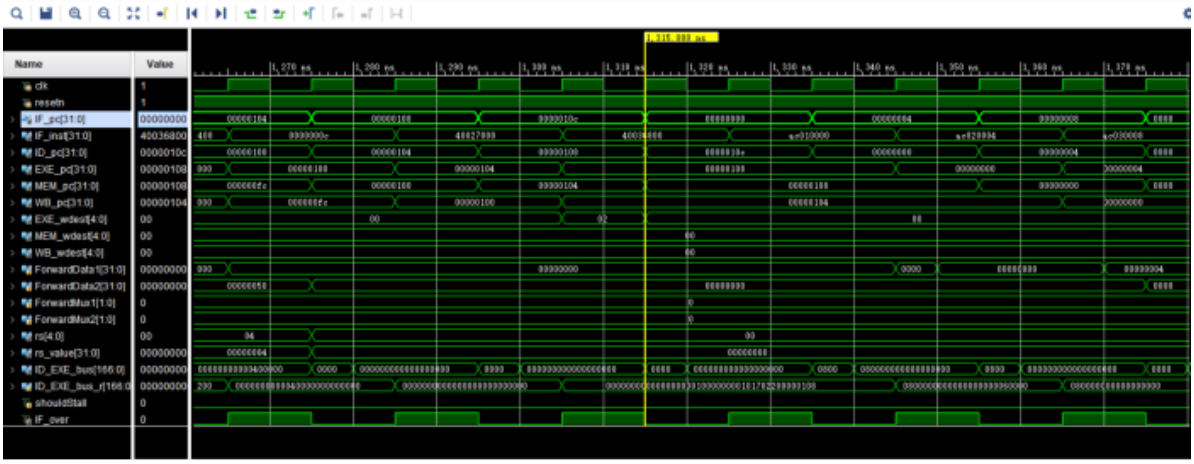
```
assign ID_over=ID_valid&(~inst_jbr|IF_over)&~shouldStall;
```



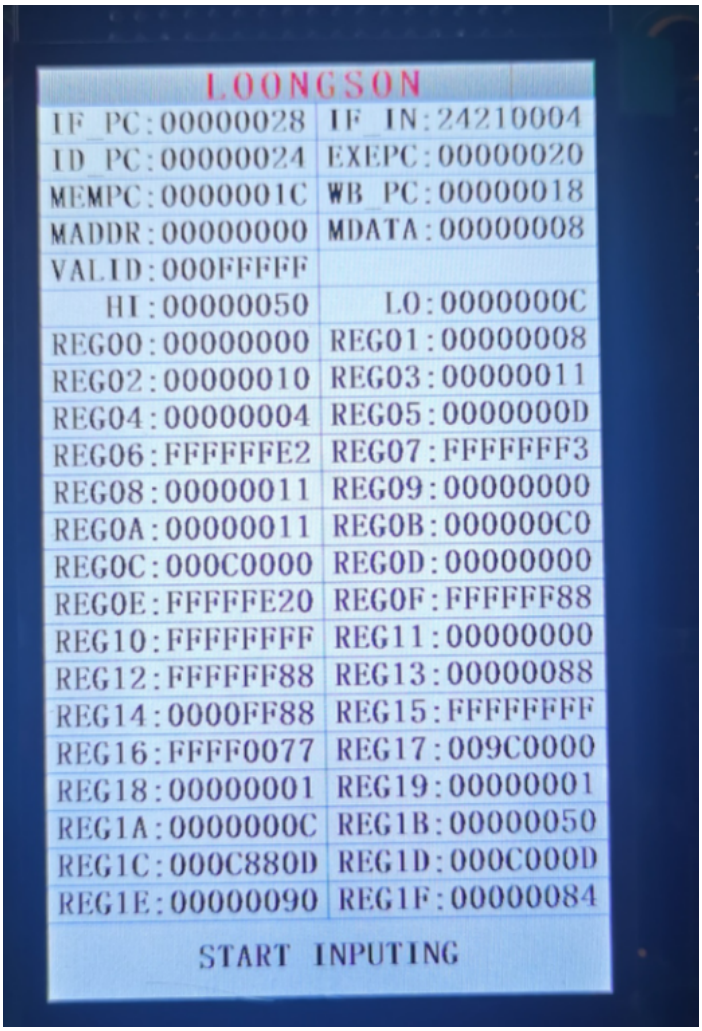
但这时候仍不能正确的处理这种情况，如图所示，之后的EXE级减少了一个时钟周期，也能够算出结果，但似乎不能将数据正常的前递，导致2CH处的mtc0指令无法将返回地址写入cp0寄存器，导致无法跳出异常处理函数。

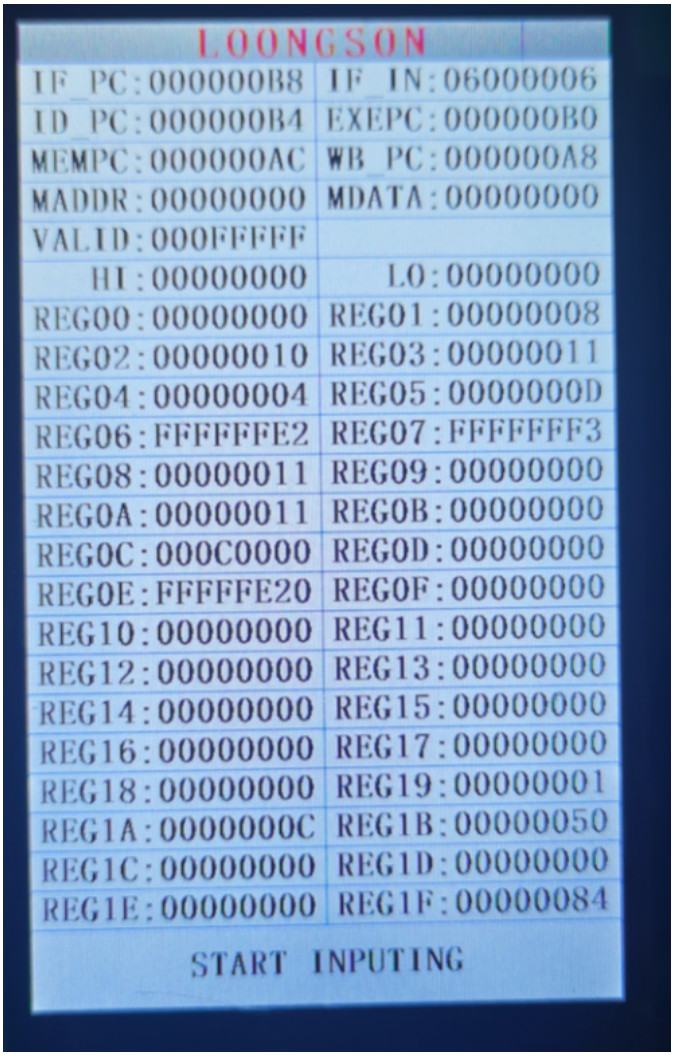


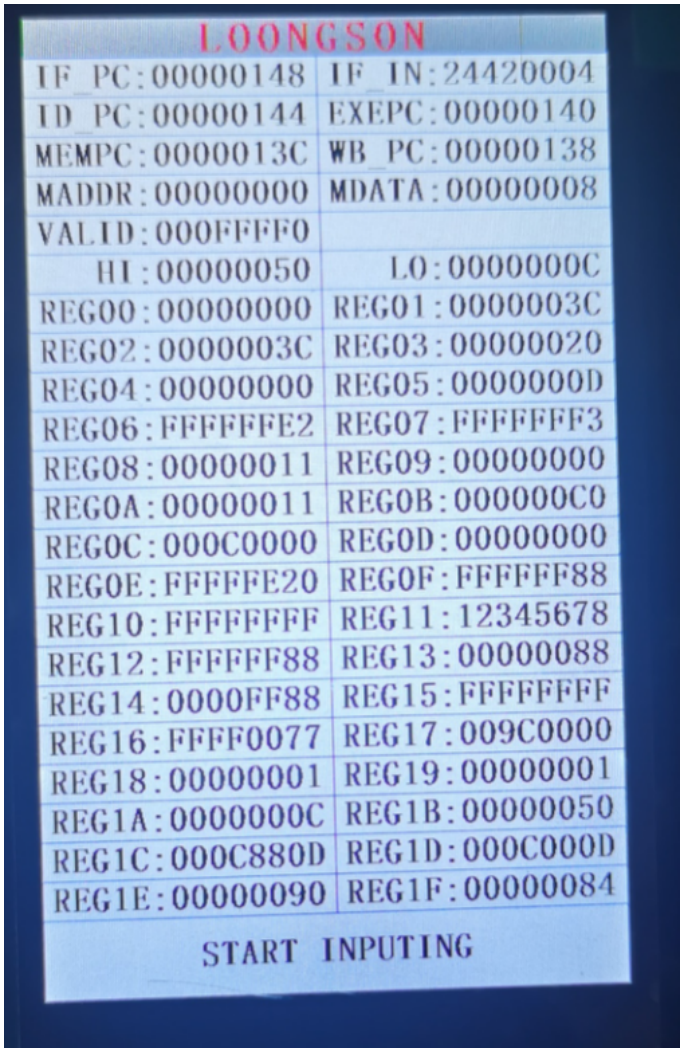
加入前递之后：1315.0ns，加速比1.61



上箱验证




LOONGSON
IF_PC:000000B8
IF_IN:06000006
ID_PC:000000B4
EXEPC:000000B0
MEMPC:000000AC
WB_PC:000000A8
MADDR:00000000
MDATA:00000000
VALID:000FFFFF
HI:00000000
LO:00000000
REG00:00000000
REG01:00000008
REG02:00000010
REG03:00000011
REG04:00000004
REG05:0000000D
REG06:FFFFFFE2
REG07:FFFFFFF3
REG08:00000011
REG09:00000000
REG0A:00000011
REG0B:00000000
REG0C:000C0000
REG0D:00000000
REG0E:FFFFFE20
REG0F:00000000
REG10:00000000
REG11:00000000
REG12:00000000
REG13:00000000
REG14:00000000
REG15:00000000
REG16:00000000
REG17:00000000
REG18:00000000
REG19:00000001
REG1A:0000000C
REG1B:00000050
REG1C:00000000
REG1D:00000000
REG1E:00000000
REG1F:00000084
START INPUTING



总结

在本次实验中，通过对实验设计原理的分析和实验的操作，复习了计算机组成原理的基本知识，熟悉了 Vivado 平台和 virlog 语言。另外，通过实验我进一步了解到了流水线 CPU 的工作原理，也更明白了冲突发生的过程，同时也为今后计算机体系结构的学习打下了基础。