《计算机组成原理实验》

实验报告

（实验三）

学 院 名 称 ：	计算机学院
专业（班级）：	21 计教学 1 班
学 生 姓 名 ：	凌国明
学　　号 ：	21307077
时　　间 ：	2022　年　1　月　18　日

成 绩 ：

实验三 ： 流水线CPU

一. 实验目的

1. 熟悉RISC-V三段和五段流水线的基本架构
2. 学习处理控制冒险、数据冒险的方法

二. 实验内容

1、使用流水线设计提供CPU的吞吐率，缩短关键路径

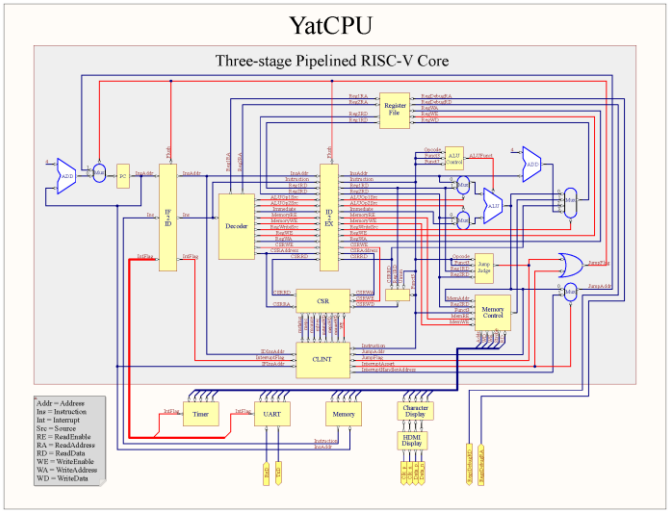
2、正确处理流水线阻塞，处理数据冒险

3、正确处理流水线清空，处理控制冒险

三. 实验原理

流水线寄存器： 流水线寄存器是在流水线中起缓存作用的寄存器，目的是切分组合逻辑，缩短关键路径。它的基本功能非常简单，在每一个时钟周期，根据复位（流水线清空）或阻塞（流水线暂停）的状态，将寄存器内容清空、保持或设置为新的值。寄存器的输出则是寄存器中保存的值。为了方便复用，我们可以定义一个带参数的 PipelineRegister 模块，用来实现不同数据位宽的流水线寄存器。

三级流水线： 以下是结构图，数据通路用蓝线表示，控制信号用红线表示



用 IF2ID 和 ID2EX 这两组流水线寄存器将单周期 CPU 的组合逻辑部分切分为三个阶段：

1、取指（Instruction Fetch，IF）：根据 PC 中的指令地址从内存中取出指令码；

2、译码（Instruction Decode，ID）：将指令码解码为控制信号并从寄存器组中读取操作数；

3、执行（Execute，EX）：包括 ALU 运算、访问内存和结果写回。

使用“清空”的方式解决控制冒险：在三级流水线中，由于所有数据处理操作都在 EX 阶段进行，因此不存在数据冒险，我们只需要处理程序跳转带来的控制冒险。有三种情况可能发生程序跳转：

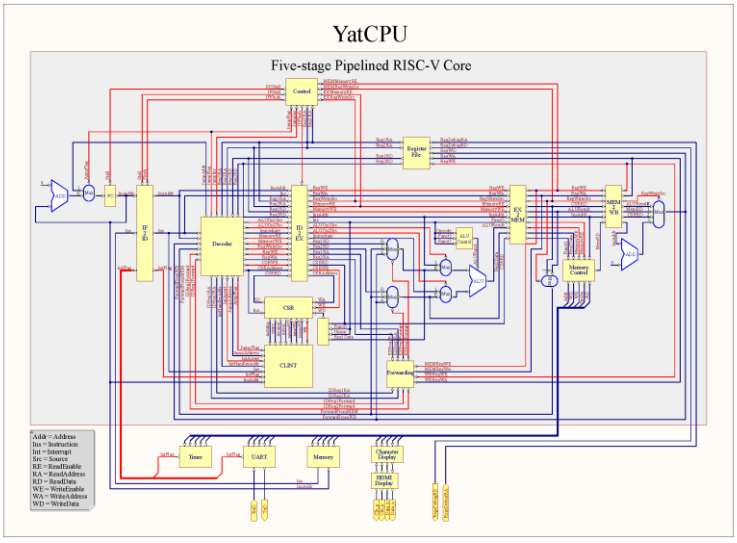
1、EX 段执行到跳转指令

2、 EX 段执行到分支指令且分支条件成立

3、发生中断，EX 段收到 CLINT 发来的 InterruptAssert 信号，这相当于在 EX 段的指令之上叠加了一条跳转指令，EX 段的指令继续执行，IF 段和 ID 段的指令将被丢弃

无论哪种情况，都是由 EX 段向 IF 段发送跳转信号 jump_flag 和跳转的目标地址 jump_address，但在 jump_address 写入 PC 并从该处取出指令前，流水线的 IF 和 ID 段已经各有两条不需要执行的指令，好在这两条指令的结果还没有写回，我们只需要清空对应的流水线寄存器，把它们变成两条空指令即可。

五级流水线：三级流水线中，执行阶段逻辑复杂，仍然可能导致较大的延迟。我们扩展流水线级数，将执行阶段进一步分为 ALU 阶段、访存阶段以及写回阶段



把三级流水线进一步分割为五级流水线将带来更加复杂的数据冒险，下面我们将尝试使用阻塞的方式解决数据冒险，得到一个功能完整的五级流水线 CPU。接着我们可以使用旁路和将分支跳转提前到 ID 阶段进一步提升 CPU 效率，这两部分将作为拓展实验供同学们选做。注意，上面的 CPU 结构图是我们完成所有实验之后的结果，在完成“缩短分支延迟”实验之前，我们 CPU 的结构将与上图稍有不同。例如，我们紧接着讨论的五级流水线 CPU 在 EX 阶段判断程序是否发生跳转，而不是 ID 阶段。

利用“阻塞”的方法解决数据冒险：

当处于 ID 阶段的指令要读取的寄存器依赖于 EX 或 MEM 阶段的指令时，发生数据冒险。此时，我们可以保持 IF 和 ID 两个阶段状态不变，直到被依赖的指令执行完成，即 ID 段能够从寄存器组获得它所需要的数据，再继续执行。

四. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五. 实验过程与结果

1、流水线寄存器：

```
// Lab3(PipelineRegister)
val reg = RegInit(UInt(width.W),defaultValue)
when(io.stall){
  reg := io.out
}.otherwise{
  when(io.flush){
    reg := defaultValue
  }.otherwise{
    reg := io.in
  }
}
io.out := reg
// Lab3(PipelineRegister) End
```

若流水线正常运转，则将流水线寄存器的值更新为新的输入值；若阻塞，则流水线寄存器保持不变；若清空，则流水线寄存器的值更新为默认值。

2、三级流水线CPU部分代码：

```
class Control extends Module {
  // Lab3(Flush)
  val io = IO(new Bundle {
    val jump_flag = Input(Bool())
    val clint_jump_flag = Input(Bool())
    val if2id_flush_flag = Output(Bool())
    val id2ix_flush_flag = Output(Bool())
  })

  io.if2id_flush_flag := Mux(
    io.clint_jump_flag || io.jump_flag,
    true.B,
    false.B
  )

  io.id2ix_flush_flag := io.if2id_flush_flag
}
```

输入：是否跳转，CLINT是否跳转； 输出：if2id是否清空，if2ix是否清空

```
// Lab3(Flush)
ctrl.io.jump_flag := ex.io.if_jump_flag
ctrl.io.clint_jump_flag := ex.io.clint_jump_flag
if2id.io.flush := ctrl.io.if2id_flush_flag
id2ex.io.flush := ctrl.io.id2ix_flush_flag
// Lab3(Flush) End
```

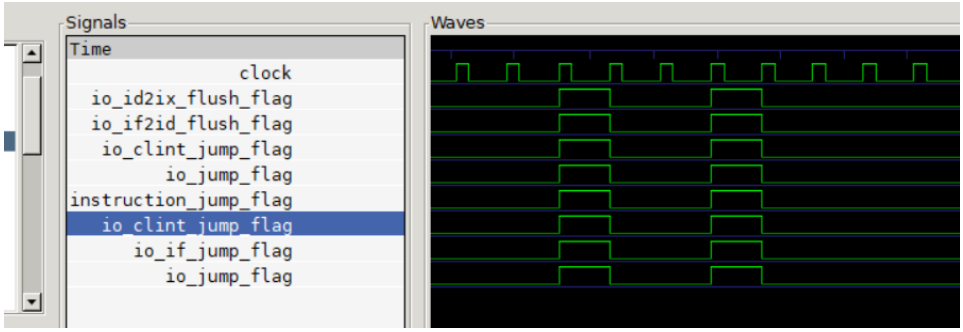
模块间的接口如上图

3、五级流水线：

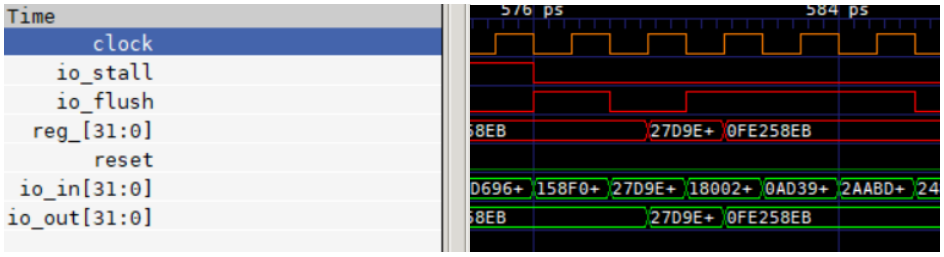
```
// Lab3(Stall)
io.if_flush := io.jump_flag
io.id_flush := io.jump_flag
io.pc_stall := false.B
io.if_stall := false.B
when(io.rs1_id === io.rd_mem || io.rs2_id === io.rd_mem){
  when(io.reg_write_enable_mem && io.rd_mem =/= 0.U && io.jump_flag === false.B){
    io.pc_stall := true.B
    io.if_stall := true.B
    io.id_flush := true.B
  }
}
when(io.rs1_id === io.rd_ex || io.rs2_id === io.rd_ex) {
  when(io.reg_write_enable_ex && io.rd_ex =/= 0.U && io.jump_flag === false.B) {
    io.pc_stall := true.B
    io.if_stall := true.B
    io.id_flush := true.B
  }
}
// Lab3(Stall) End
}
```

根据各控制信号和标志位设置pc_stall、if_stall和id_flush的值

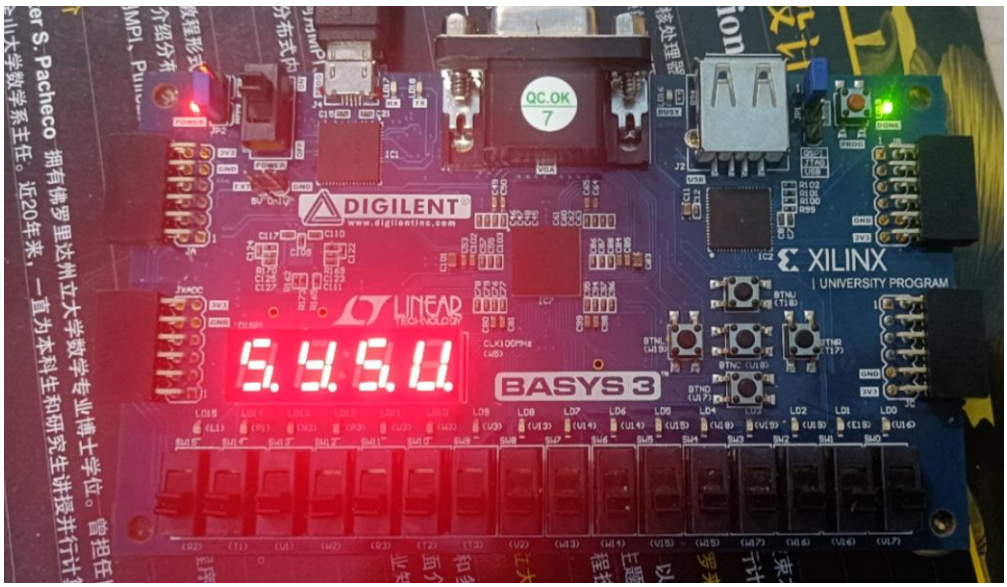
波形测试：



各jump_flag和flush_flag的值相等



波形与预期情况符合



六. 实验心得

三阶段流水线CPU在取值、译码、执行阶段的操作与单周期CPU对应的操作大同小异，流水线部分的核心是冒险的处理，包括数据冒险和控制冒险（因为三阶段流水线中数据操作都在EX阶段执行，所以没有数据冒险）。于是学习了通过清空流水线来处理控制冒险的方法：通过控制单元检测控制冒险的发生，控制流水线的清空。

五阶段流水线将执行阶段进一步分为ALU阶段、访存阶段以及写回阶段，使得吞吐率更高。但是，正因为ALU阶段与写回阶段的分隔，导致了流水线中可能出现数据冒险。这个时候我们要保持部分流水线寄存器的状态，直到被依赖的数据写回寄存器。这就是阻塞的过程。

值得一提的是，lab3相对于前两个实验，要处理的接口更多，这意味着需要更加了解各个模块的功能及各个模块之间的关系，需要有全局观，以俯视的角度看待各个模块，这样才能更容易地正确处理接口。

通过lab3，我学习了流水线中处理数据冒险和控制冒险的简单方法——清空和阻塞。写代码的过程用到了分情况讨论的方法，如流水线寄存器的更新和五级流水线中pc_stall、if_stall和id_flush的赋值。分情况讨论一定要考虑到每一种情况，做到不重复不遗漏，这样才能写出正确的代码。