《计算机组成原理实验》

实验报告

（实验一）

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 ：	2023　年　1　月　2　日

成 绩 ：

实验一 ： 单周期CPU设计与实现

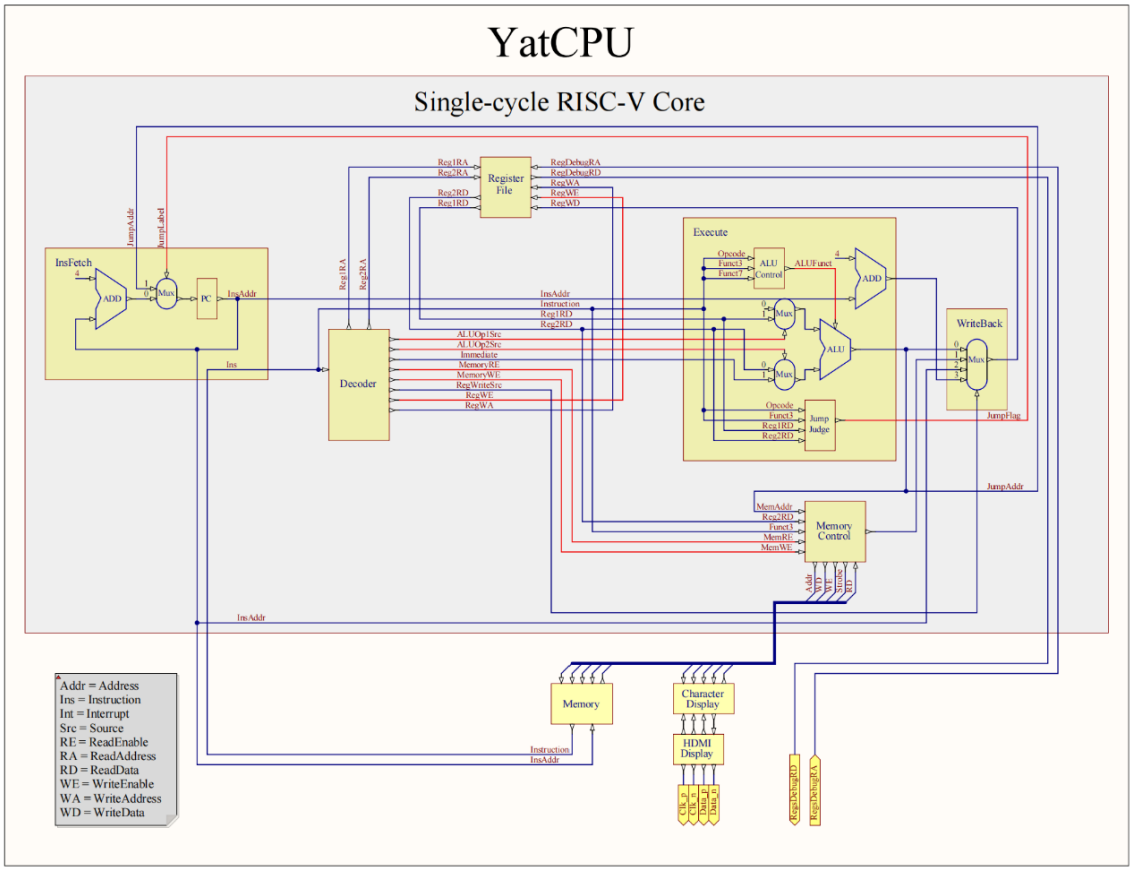
一. 实验目的

1. 了解单周期CPU的基本结构及其指令的执行方式
2. 运用chisel3语言实现运行RV32I指令集的单周期cpu

二. 实验内容

根据已有代码以及CPU结构图，完成单周期CPU的取指、译码、执行部分的代码，最终完成一个简单的RISC-V单周期处理器。

三. 实验原理



数据通路：CPU中各个模块通过数据线连接形成的数据传输路径称为数据通路。数据传输路径上操作或保存数据的部件称为数据通路部件，数据通路控制数据从一个模块流向另一个模块。

控制信号：控制数据通路，使得数据能按照特定的路径从一个模块流向另一个模块，在RISC-V指令中，控制器通过指令的opcode、funct3、funct7字段得知发出什么样的的控制信号。CPU原理图中的Decoder、ALUControl、JumpJudge三个元件都可以认为是控制单元，他们接收指令并输出控制信号。控制信号引导数据通过数据路径，从而使得指令正确执行。

实现方式：

将指令分成五个不同的阶段执行：

1、取指：从内存中获取指令数据，并更新pc

2、译码：理解指令要做的事情，读取寄存器数据，输出控制信号

3、执行：以ALU的特定模式计算结果

4、访存：读写内存

5、回写：将结果写回寄存器

四. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

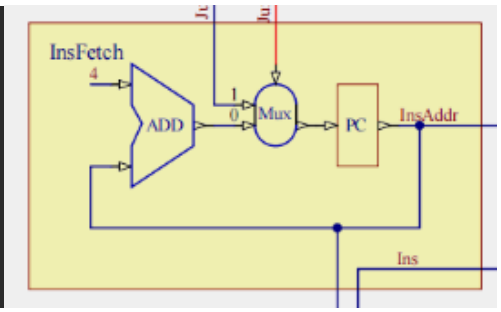
五. 实验过程与结果

设计思想：将CPU的实现分为几个模块，如取指模块、译码模块，执行模块、ALU模块、访存模块、回写模块等等，通过Chisel的端口连接实现模块之间的接线，从而实现CPU各部分的连线。控制信号表在CPU设计中不可或缺，它的作用是控制每个部件的操作，从而实现各部件间信息的传递。

以下是关键部分代码的说明

1、取指模块

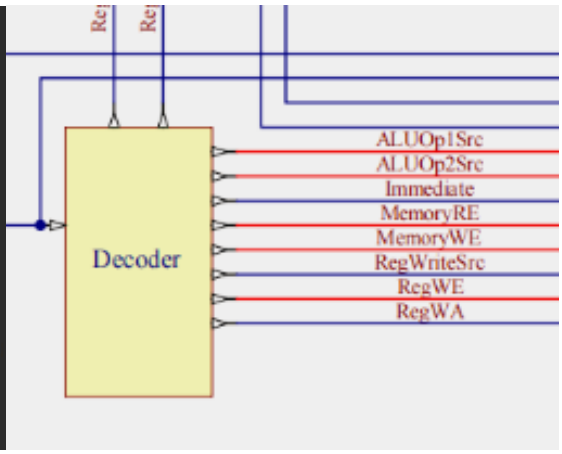
```
when(io.instruction_valid) {
  io.instruction := io.instruction_read_data
  // lab1(InstructionFetch)
  when(io.jump_flag_id){
    pc := io.jump_address_id
  }.otherwise{
    pc := pc + 4.U
  }
  // la1(InstructionFetch) end
```



当io.instruction_valid信号为高电平时，将io.instruction赋值，这部分意味着要读出一条指令。紧接着是下条指令的地址：如果jump_flag_id是高电平，意味着这是一条跳转指令，这时将pc赋值为跳转目标地址；如果不是高电平，意味着程序顺序执行，pc加4以瞄准下条指令。

2、译码模块

```
// lab1(InstructionDecode)
io.ex_aluop2_source := Mux(
  opcode === InstructionTypes.RM,
  ALUOp2Source.Register,
  ALUOp2Source.Immediate
)
io.memory_read_enable := opcode === InstructionTypes.L
io.memory_write_enable := opcode === InstructionTypes.S
io.wb_reg_write_source := MuxLookup(
  opcode,
  RegWriteSource.ALUResult,
  IndexedSeq(
    InstructionTypes.L -> RegWriteSource.Memory,
    Instructions.jal -> RegWriteSource.NextInstructionAddress,
    Instructions.jalr -> RegWriteSource.NextInstructionAddress,
    //Instructions.csr -> RegWriteSource.CSR
  )
)
// lab1(InstructionDecode) end
```



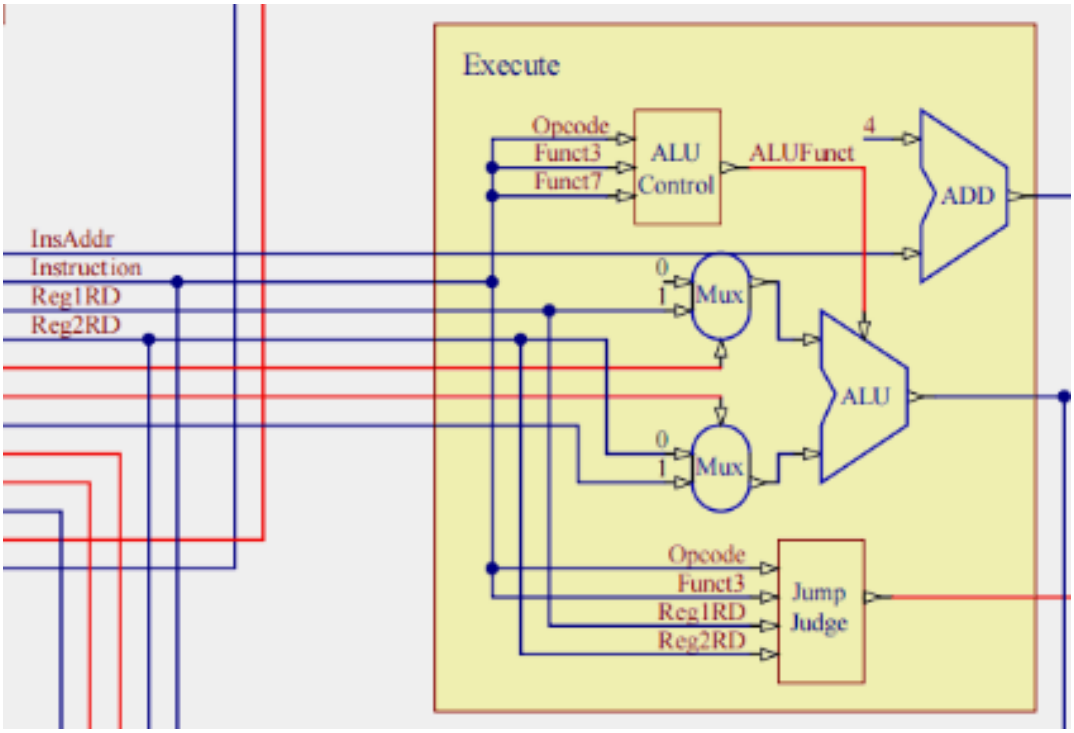
译码部分的工作是读取寄存器和输出控制信号。最重要的控制信号之一是ALU两个操作数的来源，是来自寄存器还是立即数，要利用opcode字段区分指令类型，并根据指令类型输出相应的控制信号。

当指令属于auipc、jal或B类时，ex_aluop1_source置为0，控制ALU第一个操作数的输入为指令地址；其他情况 ex_aluop1_source置为1，控制 ALU 第一个操作数的输入为寄存器。当指令属于RM类型时，ALU的第二个操作数取自寄存器2，否则就取自立即数。

判断opcode，若属于L指令则读使能为高电平，若属于S指令则写使能为高电平。

3、执行模块

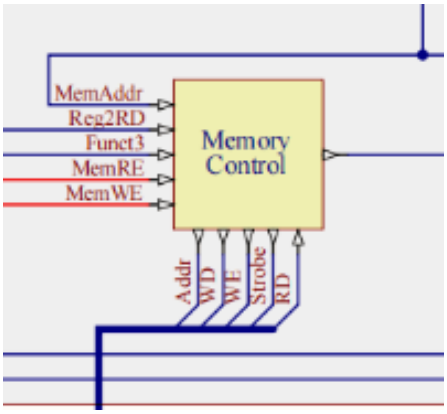
```
// lab1(Execute)
alu.io.func := alu_ctrl.io.alu_funct
alu.io.op1 := Mux(
  io.aluop1_source === ALUOp1Source.InstructionAddress,
  io.instruction_address,
  io.reg1_data,
)
alu.io.op2 := Mux(
  io.aluop2_source === ALUOp2Source.Immediate,
  io.immediate,
  io.reg2_data,
)
io.if_jump_flag := opcode === Instructions.jal ||
  (opcode === Instructions.jalr) ||
  (opcode === InstructionTypes.B) && MuxLookup(
    funct3,
    false.B,
    IndexedSeq(
      InstructionsTypeB.beq -> (io.reg1_data === io.reg2_data),
      InstructionsTypeB.bne -> (io.reg1_data =/= io.reg2_data),
      InstructionsTypeB.blt -> (io.reg1_data.asSInt < io.reg2_data.asSInt),
      InstructionsTypeB.bge -> (io.reg1_data.asSInt >= io.reg2_data.asSInt),
      InstructionsTypeB.bltu -> (io.reg1_data.asUInt < io.reg2_data.asUInt),
      InstructionsTypeB.bgeu -> (io.reg1_data.asUInt >= io.reg2_data.asUInt)
    )
  )
io.if_jump_address := io.immediate + Mux(opcode === Instructions.jalr, io.reg1_data, io.instruction_address)
// lab1(Execute) end
```

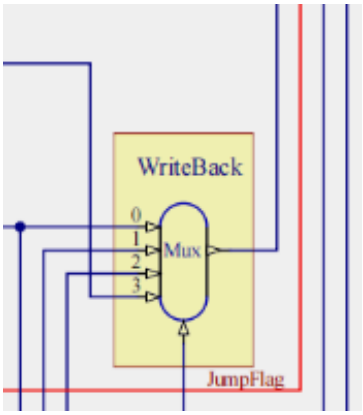


执行部分需要进行ALU的计算，并判断是否跳转判断。如果是无条件跳转指令则直接跳转，如jal指令；如果是分支指令则根据相应的跳转条件判断是否跳转，如beq指令。跳转时将控制信号if_jump_flag置为高电平。

4、访存部分：译码阶段判断opcode，若属于L指令则读使能为高电平，若属于S指令则写使能为高电平。读使能和写使能决定了访存阶段是读内存还是写内存。

5、回写部分：以MUX的形式，把计算的结果或读取的数据写入特定寄存器。

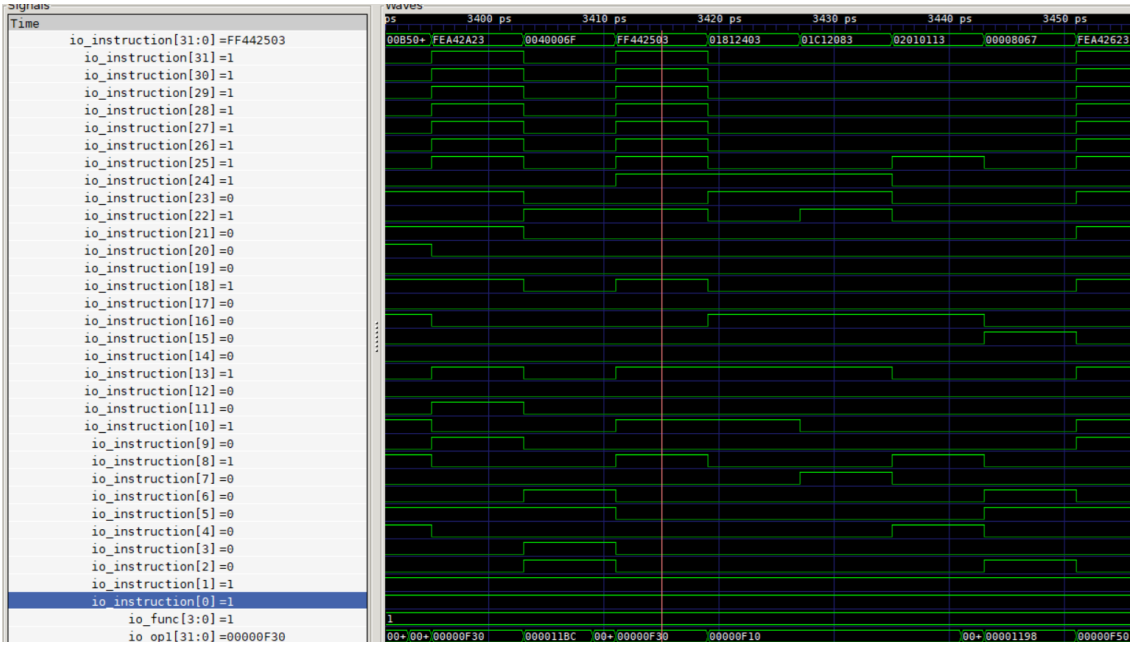




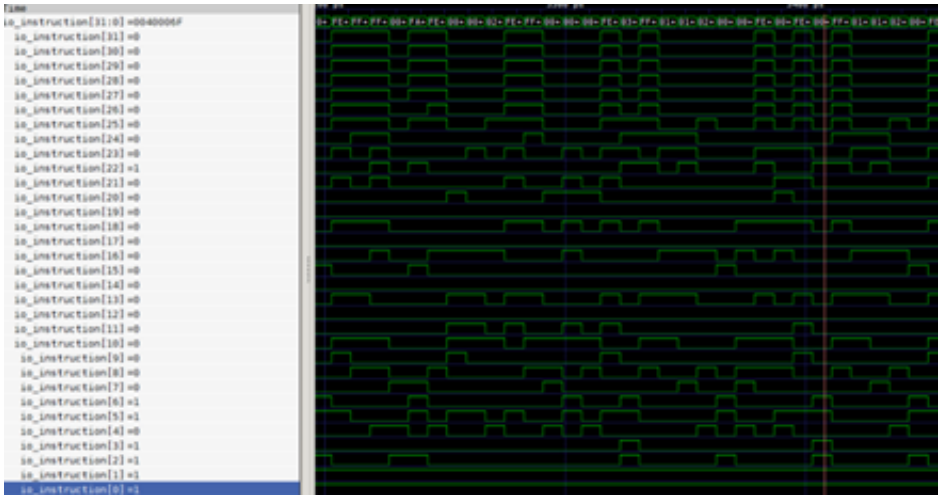
验证正确性：

使用Vivado进行仿真，得到VCD波形图，使用GTKWave软件查看波形图，与理想计算结果进行对比验证，以此验证CPU的正确性。

1、lw指令

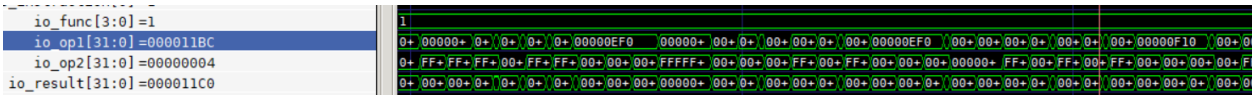


2、jal指令



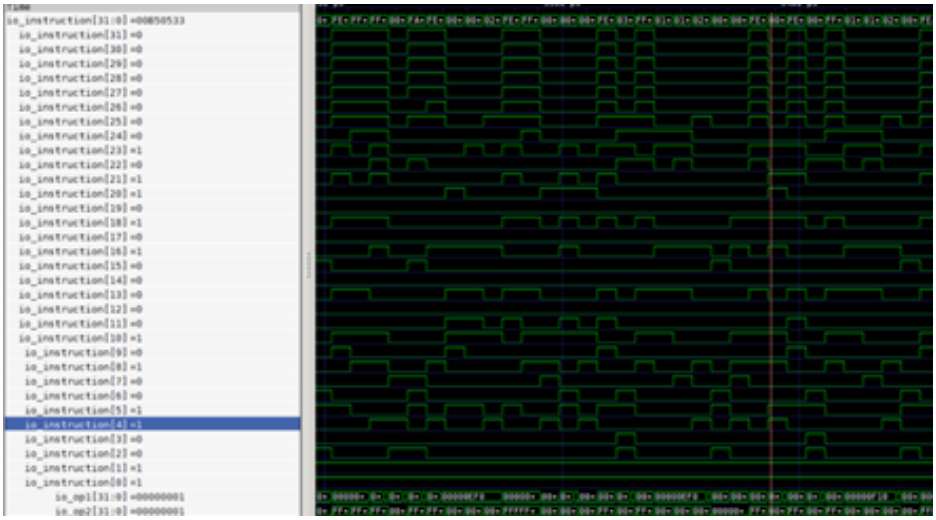
pc为

ALU结果为



跳转后指令地址为

3、add指令



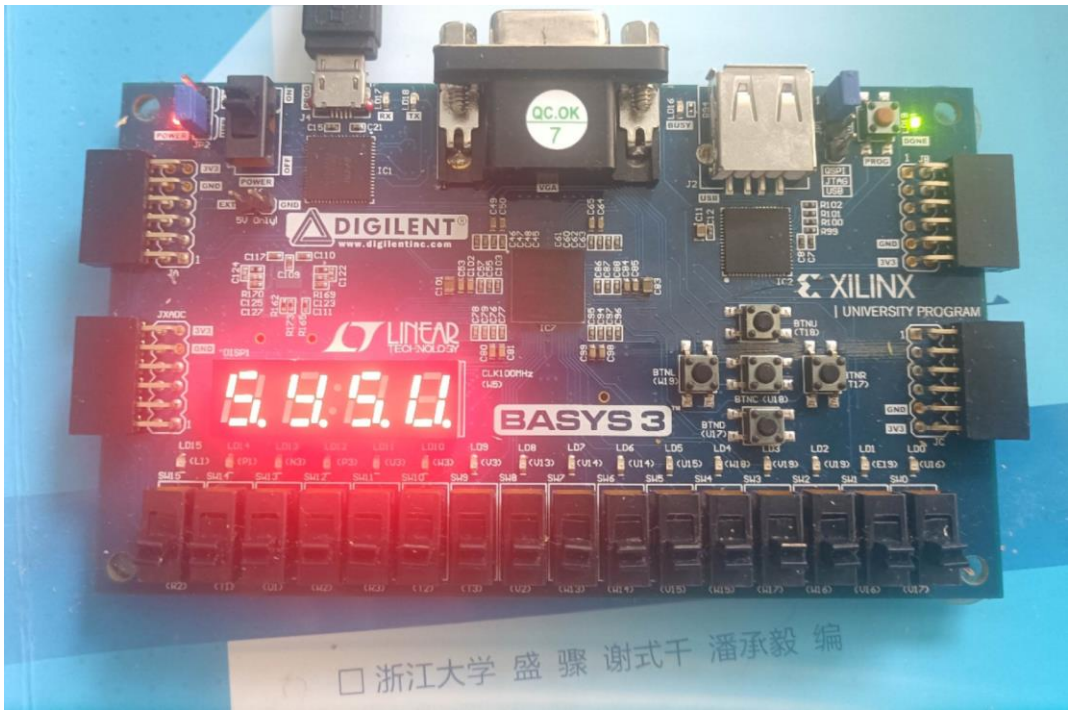
ALU模块：

寄存器初始状况：

执行add后寄存器的状况：

这个过程将$10和$11中的值做加法，将结果（1+1=2）写回$10中，是正确的。

烧板结果：



六. 实验心得

整个实验过程中，我不仅学习到了专业知识，而且还把计算机组成原理的理论知识运用到实践中，通过硬件编程实现简单的单周期CPU，这个过程巩固了理论知识，更提升了实践能力。我学习到的最重要的东西是解决问题的能力。

首先，配置环境的过程：安装Vivado以及Intellij IDEA；在安装Scala的过程中，因下载速度太慢，借助科学上网的方法才成功下载，结果下载版本不兼容，合适的版本又无法通过Intellij IDEA下载。通过查询网上的资料，在网络上下载了Scala-sdk-2.12.13包，手动添加到项目库中，这才得以运行。这个过程中还遇到许多其他问题，如环境变量的配置以及中文路径的问题，都是通过网上查阅资料解决的，这也提升了我的搜索能力和解决问题的能力。

Chisel允许我们在Scala中构造硬件，是一门比Verilog更好用的硬件描述语言。在这门语言的学习过程中，我遇到和解决了许多问题，提升了自己发现问题和解决问题的能力：首先，Chisel的语法和C++语言有相通之处，比如面向对象的程序设计思想，继承与派生的思想，通过类比的方法可以做到快速入门，这给我带来启示：矛盾具有普遍性，许多编程语言都是相通的，通过类比学习法，可以快速掌握。但是Chisel有许多独特的关键字，如when和otherwise，初接触Chisel时，不知道这两个关键字，于是我用if和else取代它们，尽管逻辑上是一致的，但是一直过不了单元测试，询问同学后才知道要用这两个关键字。这也给我带来启示：矛盾具有特殊性，不能只凭经验对新事物作出判断，一定要查阅相关资料，多作了解，若是先入为主地作出判断，很可能会造成错误。

项目的大部分代码已经提供，只有少数核心逻辑部分是空缺的。其中访存部分、ALU部分、回写部分已经提供，指令译码部分也已提供各指令的机器码，这节省了我们很多时间，也使得我们可以专注于核心逻辑部分，学习效率更高，这里为老师和助教点赞！通过阅读已有代码补全空缺代码的过程就像一个合作的过程，我不仅学习了RISCV单周期CPU的核心逻辑，而且了解了搭建项目的大概方法，受益匪浅。

补完空缺代码，通过测试后，通过Vivado生成了VCD波形图，通过GTKWave打开波形图，进行逐个验证。先在InstructionFench阶段找到对应指令，如第n个ps的32位instruction对应的是add指令，那么我就去相关模块的对应端口查看输入输出，查看相应的控制信号、结果数据与地址，与自己手动计算的结果进行对比，以此验证指令是否正确执行。这是一个费时费力的过程，而且需要注意力集中，一旦看错一位01，就会造成错误，验证波形的过程非常需要细心与耐心。在这个过程中，我培养了自己的细心与耐心，受益良多。

疑问：InstructionDecode中有乘法指令mul，但是ALU部件代码里却没有乘法指令，也没有通过移位加法的方式实现乘法的代码，那么乘法是如何执行的呢？