深 圳 大 学 实 验 报 告

课 程 名　称：计算机系统(3)

实验项目名称：取指和指令译码设计

学　　院：计算机与软件学院

专　　业：软件工程（腾班）

指 导 教　师：王毅

报告人：黄亮铭　学号：2022155028　班级：腾班

实 验 时　间：2024 年 10 月 18 日

实验报告提交时间：2024 年 11 月 8 日

教务处制

一、 实验目标：

设计完成一个连续取指令并进行指令译码的电路，从而掌握设计简单数据通路的基本方法。

二、实验内容

本实验分成三周（三次）完成：1）首先完成一个译码器（30 分）；2）接着实现一个寄存器文件（30 分）；3）最后添加指令存储器和地址部件等将这些部件组合成一个数据通路原型（40 分）。

三、实验环境

硬件：桌面 PC

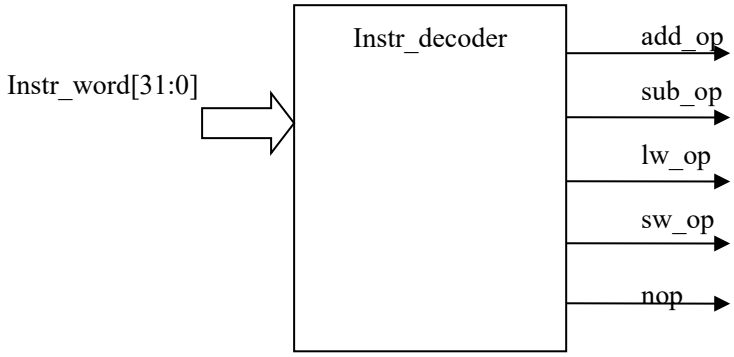
软件：Linux Chisel 开发环境

四、实验步骤及说明

本次试验分为三个部分：

1） 设计译码电路，输入位 32bit 的一个机器字，按照课本 MIPS 指令格式，完成 add、sub、lw、sw 指令译码，其他指令一律译码成 nop 指令。输入信号名为 Instr_word，对上述四条指令义译码输出信号名为 add_op、sub_op、lw_op 和 sw_op，其余指令一律译码为 nop；

给出 Chisel 设计代码和仿真测试波形，观察输入 Instr_word 为 add R1,R2,R3; sub R0,R5,R6， lw R5,100(R2)， sw R5,104(R2)、JAL RA,100(R2)时，对应的输出波形



1.1 设计思路

我将设计译码电路分成以下几个模块以便实现实验的需求：定义模块、初始化模块和判断模块。接下来，我将详细说明每个模块的代码实现。

1.2 定义模块

定义模块的作用主要是定义变量。变量可以分为两部分：**第一部分为负责IO信号的变量**，输入变量为32位无符号整数代表操作，输出位Bool变量，代表该操作在当前时钟周期是否进行；**第二部分为存储操作码的变量和判断操作的变量**。根据MIPS的汇编代码和二进制代码，我们可以知道add、sub为I型指令，左边6个字节均为0，依靠右边6个字节判断操作；而其他的可以依靠左边6个变量判断操作。据此，我们可以将存储操作的变

量设置为相应的6个字节。同时，额外设置一个为0的变量判断是否为R型指令。

具体实现代码如下表格所示，表格1a中为第一部分变量，表格1b为第二部分变量。

```
val io = IO(new Bundle {
    val Instr_word = Input(UInt(32.W))

    val add_op = Output(Bool())
    val sub_op = Output(Bool())
    val lw_op = Output(Bool())
    val sw_op = Output(Bool())
    //val jal_op = Output(Bool())
    val nop_op = Output(Bool())
})
```

图1a：第一部分

```
val I = "b000000".U(6.W)
val ADD_OPCODE = "b100000".U(6.W)
val SUB_OPCODE = "b100010".U(6.W)
val LW_OPCODE = "b100011".U(6.W)
val SW_OPCODE = "b101011".U(6.W)
//val JAL_OPCODE = "b000011".U(6.W)

val opcode1 = io.Instr_word(31, 26)
val opcode2 = io.Instr_word(5,0)
```

图1b：第二部分

1.3 初始化模块

根据实验文档，除了文档中提到的四种指令以外，其余的均认为是nop指令。因此，我们可以默认所有指令均为nop指令，然后判断当前时钟周期的指令是否位文档中提到的四种指令。

```
29    io.add_op := false.B
30    io.sub_op := false.B
31    io.lw_op := false.B
32    io.sw_op := false.B
33    //io.jal_op := false.B
34    io.nop_op := true.B
35
```

图2：初始化

1.4 判断模块

该模块首要任务是判断指令的26-31字节的数值是否为0，如果为0，则说明是I型指令，，即：add和sub中的一种。然后我们可以判断0-5字节的数值，进而可以知道当前指令是什么指令，再将对应指令的output变量设置为True，nop指令对应变量设置为False。如果不为0，则可以直接通过26-31字节判断是什么指令，接下来的做法与上述相同。具体代码见下图。

```
when (opcode1 === I) {
    when (opcode2 === ADD_OPCODE) {
        io.add_op := true.B
        io.nop_op := false.B
    }.elsewhen (opcode2 === SUB_OPCODE) {
        io.sub_op := true.B
        io.nop_op := false.B
    }
}.elsewhen (opcode1 === LW_OPCODE) {
    io.lw_op := true.B
    io.nop_op := false.B
}.elsewhen (opcode1 === SW_OPCODE) {
    io.sw_op := true.B
    io.nop_op := false.B
}//.elsewhen (opcode1 === JAL_OPCODE) {
    //io.jal_op := true.B
    //io.nop_op := false.B
//}
```

图3：判断指令

1.5 仿真测试

在对应的Test文件中，我分别设置Instr_word为add R1,R2,R3; sub R0,R5,R6，lw R5,100(R2)，sw R5,104(R2)、JAL RA,100(R2)。具体代码如下所示。

```
class InstructionTest extends AnyFlatSpec with ChiselScalatestTester {
  behavior of "Test"
  it should "pass" in {
    test(new Decoder).withAnnotations(Seq(WriteVcdAnnotation)) { d =>
      val ADD = "b00000000010000110000100000100000".U(32.W)  // add R1, R2, R3
      val SUB = "b00000000010100110000000000100010".U(32.W)  // sub R0, R5, R6
      val LW  = "b10001100101000100000000000100010".U(32.W)  // lw R5, 100(R2)
      val SW  = "b10101100101000100000000000110100".U(32.W)  // sw R5, 104(R2)
      val JAL = "b00001111111000100000000001001000".U(32.W)  // jal 100(R2)

      d.clock.setTimeout(0)  // 初始化时钟

      // ADD 指令
      d.io.Instr_word.poke(ADD)
      d.clock.step(1)

      // SUB 指令
      d.io.Instr_word.poke(SUB)
      d.clock.step(1)

      // LW 指令
      d.io.Instr_word.poke(LW)
      d.clock.step(1)

      // SW 指令
      d.io.Instr_word.poke(SW)
      d.clock.step(1)

      // JAL 指令
      d.io.Instr_word.poke(JAL)
      d.clock.step(1)

      // 结束
      d.clock.step(10)
    }
  }
}
```

图4：测试代码

终端打开对应项目的根目录，然后输入命令*sbt test*，再输入命令*gtkwave*，最后根据实验PPT给出的步骤进行操作即可观察到波形。

观察波形，发现第一个时钟周期add指令对应的输出信号为True，其余为False；第二个时钟周期sub指令对应的输出信号为True，其余为False；第三个时钟周期lw指令对应的输出信号为True，其余为False；第四个时钟周期sw指令对应的输出信号为True，其余为False；其他时钟周期nop指令对应的输出信号为True，其余为False。由波形图可知：add、sub、lw和sw均能在运行的时钟周期内被识别到，而jal指令没有对应输出信号，也被归类为nop指令。

波形图如下图所示。

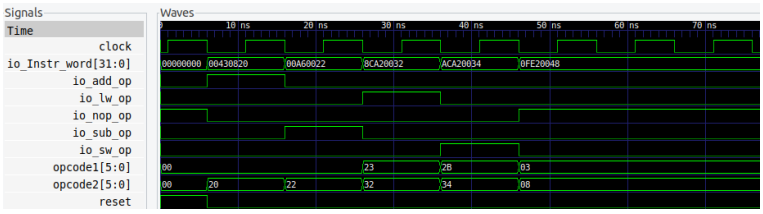
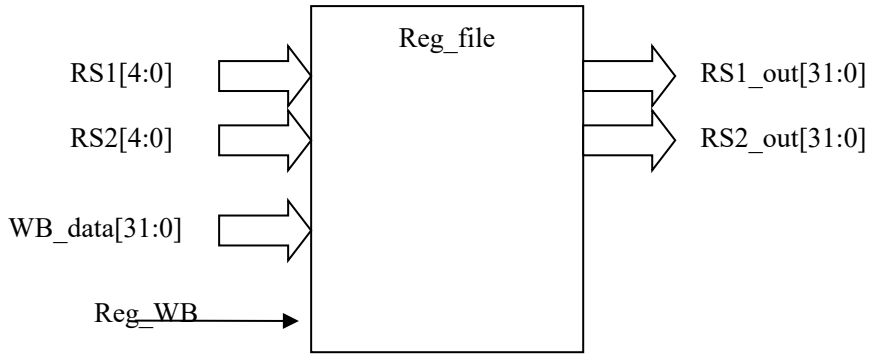


图5：输出波形

2）设计寄存器文件，共 32 个 32bit 寄存器，允许两读一写，且 0 号寄存器固定读出位 0。四个输入信号为 RS1、RS2、WB_data、Reg_WB，寄存器输出 RS1_out 和 RS2_out；寄存器内部保存的初始数值等同于寄存器编号

给出 Chisel 设计代码和仿真测试波形，观察 RS1=5,RS2=8，WB_data=0x1234,Reg_WB=1 的输出波形和受影响寄存器的值。



2.1 设计思路

我将设计寄存器文件分为两个模块：定义模块和写入模块。接下来，我将详细说明每个模块的代码实现。

2.2 定义模块

根据上述给出的寄存器文件的设计图和文档的要求，我定义了两个读入变量和两个输出变量以及写使能变量、数据存储变量等。然后我对32个寄存器进行初始化，目的是使寄存器内部保存的初始数值等同于寄存器编号，满足实验文档的要求。具体代码如下图所示。

```
val io = IO(new Bundle {
    // 两读
    val RS1 = Input(UInt(5.W))
    val RS2 = Input(UInt(5.W))
    val RS1_out = Output(UInt(32.W))
    val RS2_out = Output(UInt(32.W))

    // 一写
    val Reg_WB = Input(Bool())
    val WB_data = Input(UInt(32.W))
})

val registers = RegInit(VecInit((0 until 32).map(i => i.U(32.W))))
```

图6：定义变量并初始化部分变量

2.3 写入模块

该模块将寄存器连接到输出变量上，然后根据写使能变量判断是否可以写入寄存器，如果可以则写入对应寄存器，否则不写入。具体代码如下图所示。

```
io.RS1_out := Mux(io.RS1 === 0.U, 0.U, registers(io.RS1))
io.RS2_out := Mux(io.RS2 === 0.U, 0.U, registers(io.RS2))

when (io.RF_WB) {
    registers(io.RS1) := Mux(io.RS1 === 0.U, 0.U, io.WB_data)
    registers(io.RS2) := Mux(io.RS2 === 0.U, 0.U, io.WB_data)
}
```

图7：写入模块

2.4 仿真测试

在对应的Test文件中，我进行如下设置RS1=5,RS2=8，WB_data=0x1234,Reg_WB=1。具体代码如下。

```
import chisel3._
import chiseltest._
import org.scalatest.flatspec.AnyFlatSpec

class RegisterFileTest extends AnyFlatSpec with ChiselScalatestTester {
    behavior of "RegisterFile Module"

    it should "observe the effect on registers" in {
        test (new RegisterFile).withAnnotations(Seq(WriteVcdAnnotation)) { c =>
          c.io.RS1.poke(5.U) //          RS1 = 5
          c.io.RS2.poke(8.U) //          RS2 = 8
          c.io.WB_data.poke(0x1234.U) // WB_data = 0x1234

          c.io.Reg_WB.poke(true)
          c.clock.step(1)
      }
    }
}
```

图8：测试文件

终端打开对应项目的根目录，然后输入命令*sbt test*，再输入命令*gtkwave*，最后根据实验PPT给出的步骤进行操作即可观察到波形。

观察波形，我们可以发现，寄存器5和寄存器8的值确实被修改为0x1234，说明我设计的寄存器文件是正确的。

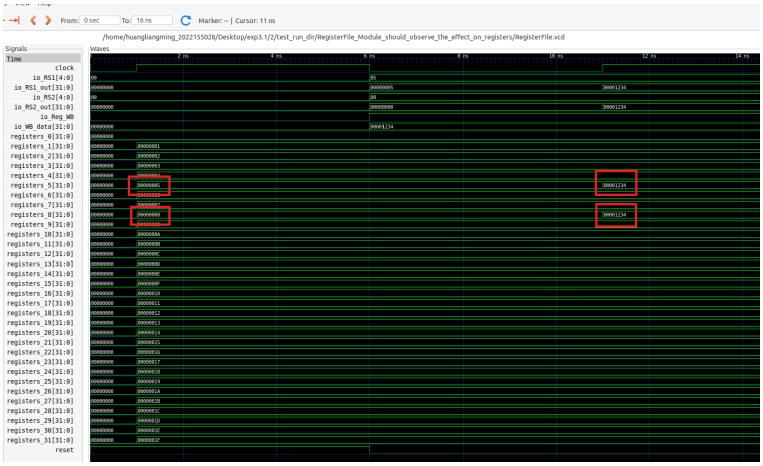
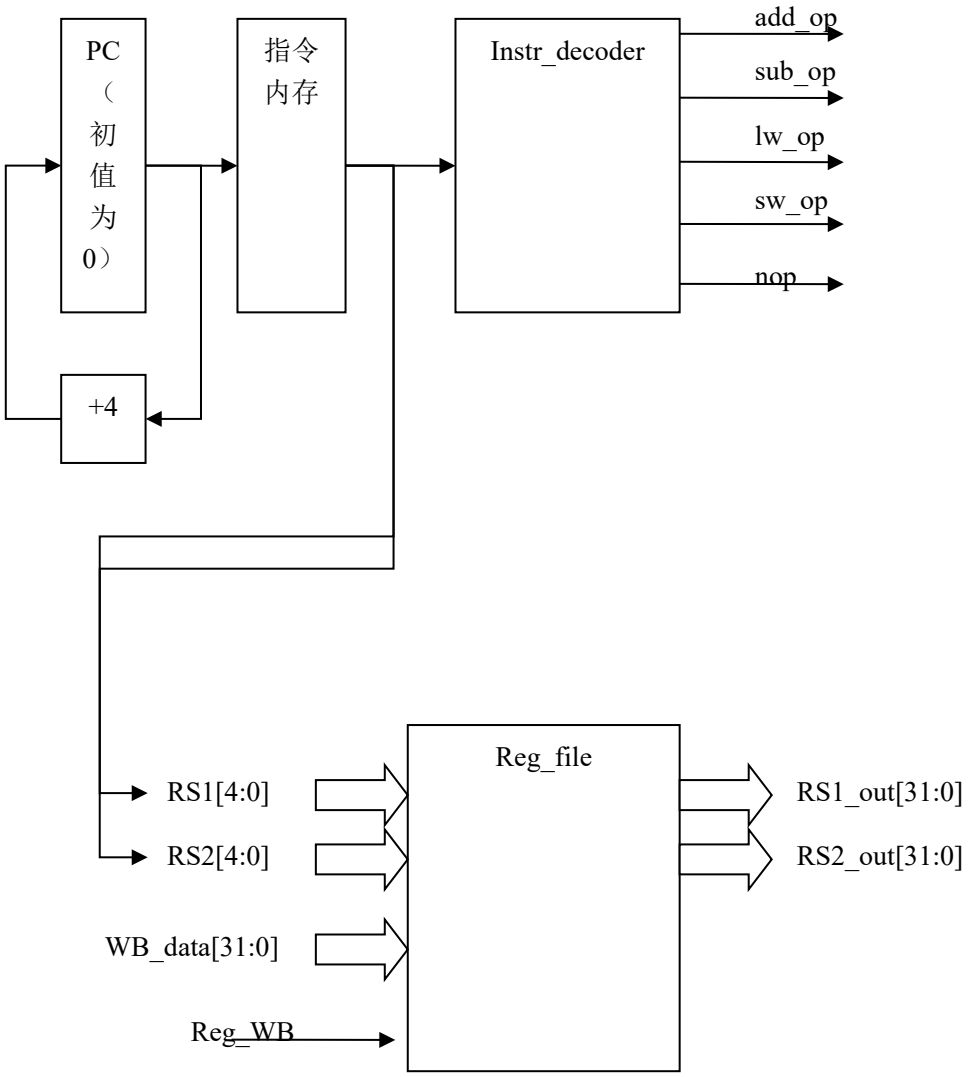


图9：输出波形

3）实现一个 32 个字的指令存储器，从 0 地址分别存储 4 条指令 add R1,R2,R3; sub R0,R5,R6，lw R5,100(R2)， sw R5,104(R2)。然后组合指令存储器、寄存器文件、译码电路，并结合 PC 更新电路（PC 初值为 0）、WB_data 和 Reg_WB 信号产生电路，最终让电路能逐条指令取出、译码（不需要完成指令执行）。**给出 Chisel 设计代码和仿真测试波形，观察四条指令的执行过程波形，记录并解释其含义。**



3.1 设计思路

译码电路和寄存器文件设计并实现的任务上面已经完成。因此，该任务我们需要完成的部分为指令读写部分和整合译码电路、寄存器文件和指令读写三部分，将其合为一体。

3.2 指令读写

在该模块，我们首先需要定义相关的读写变量：读写使能变量、读写数据存储变量和读写地址变量等。

```
val io = IO(new Bundle {
    // 读
    val rdEn = Input(Bool())
    val rdData = Output(UInt(32.W))

    // 写
    val wrEn = Input(Bool())
    val wrAddr = Input(UInt(10.W))
    val wrData = Input(UInt(32.W))
})
```

图10：定义变量

然后我们实现读写操作。因为有四条指令，所以读写操作会拆分出四个部分。

```
when (io.wrEn) {
    m.write(io.wrAddr, io.wrData(7, 0))
    m.write(io.wrAddr + 1.U, io.wrData(15, 8))
    m.write(io.wrAddr + 2.U, io.wrData(23, 16))
    m.write(io.wrAddr + 3.U, io.wrData(31, 24))
}

when (io.rdEn) {
    val rdData0 = m.read(pcReg)
    val rdData1 = m.read(pcReg+1.U)
    val rdData2 = m.read(pcReg+2.U)
    val rdData3 = m.read(pcReg+3.U)
    io.rdData := rdData3 ## rdData2 ## rdData1 ## rdData0
    pcReg := pcReg + 4.U
}.otherwise{
   io.rdData := 0.U
}
```

图11：读写操作

3.3 整合部分

这一部分的操作类似于C++和Java中的对象组合和继承的结合体。将上述三个对象内嵌到整合部分中，然后在整合部分中定义变量，将上述三个对象的值赋予在整合部分中定义的对应的变量。具体代码如下图所示。

```
// 连接指令信号
io.add_op := decoder.io.add_op
io.sub_op := decoder.io.sub_op
io.lw_op := decoder.io.lw_op
io.sw_op := decoder.io.sw_op
io.nop_op := decoder.io.nop_op

// 连接控制信号
ins.io.rdEn := io.rdEn
ins.io.wrEn := io.wrEn
ins.io.wrAddr := io.wrAddr
ins.io.wrData := io.wrData

// 连接输入数据
registerFile.io.RS1 := ins.io.rdData(25,21)
registerFile.io.RS2 := ins.io.rdData(20, 16)
decoder.io.Instr_word := ins.io.rdData
io.Instr_word := ins.io.rdData

// 初始化控制信号
registerFile.io.Reg_WB := false.B
registerFile.io.WB_data := 0.U

// 连接输出信号
io.RS1_out := registerFile.io.RS1_out
io.RS2_out := registerFile.io.RS2_out
```

图12：整合部分

3.4 仿真测试

该阶段的仿真测试类似于1中的仿真测试，但是需要添加额外的信息：写使能、写地址。具体测试代码如下图所示。

```
test(new ALL).withAnnotations(Seq(WriteVcdAnnotation)) { a =>
    val ADD = "b00000000010000110000100000100000".U(32.W)  // add R1, R2, R3
    val SUB = "b00000000101001100000000000100010".U(32.W)  // sub R0, R5, R6
    val LW  = "b10001100101000100000000000110010".U(32.W)  // lw R5, 100(R2)
    val SW  = "b10101100101000100000000000110100".U(32.W)  // sw R5, 104(R2)
    val JAL = "b00001111111100010000000000001000".U(32.W)  // jal 100(R2)

    a.clock.setTimeout(0)  // 初始化时钟

    // ADD 指令
    a.io.wrEn.poke(true)
    a.io.wrAddr.poke(0.U)
    a.io.wrData.poke(ADD)
    a.clock.step(1)

    // SUB 指令
    a.io.wrEn.poke(true)
    a.io.wrAddr.poke(4.U)
    a.io.wrData.poke(SUB)
    a.clock.step(1)

    // LW 指令
    a.io.wrEn.poke(true)
    a.io.wrAddr.poke(8.U)
    a.io.wrData.poke(LW)
    a.clock.step(1)

    // SW 指令
    a.io.wrEn.poke(true)
    a.io.wrAddr.poke(12.U)
    a.io.wrData.poke(SW)
    a.clock.step(1)

    // JAL 指令
    a.io.wrEn.poke(true)
    a.io.wrAddr.poke(16.U)
    a.io.wrData.poke(JAL)
```

图13：测试代码

测试波形如下图所示。四条指令的内容首先出现在wrData中表示正在写入内存，然后四条指令的内容出现在rdData和Instr_word中，说明这四条指令正在从内存中读出并被传送的译码电路中进行译码。随后我观察到和任务1中类似的指令输出信号。add、sub、lw和sw均能在运行的时钟周期内被识别到，而jal指令没有对应输出信号，被归类为nop指令。

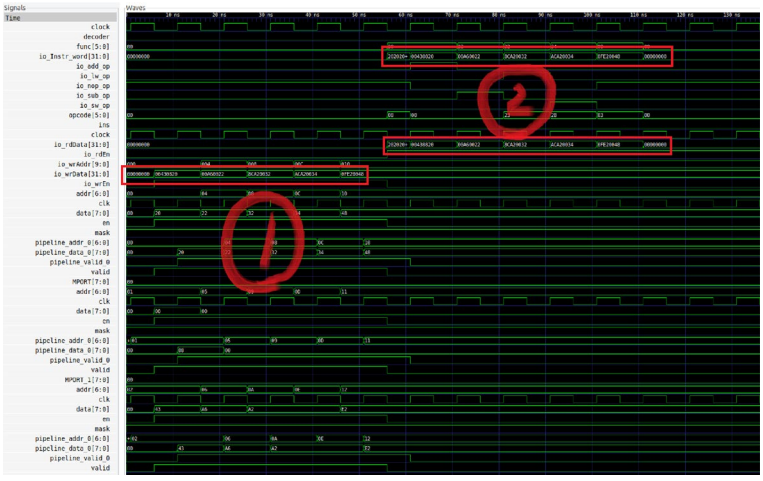


图14：测试波形

五、实验结果

设计译码电路

终端打开对应项目的根目录，然后输入命令*sbt test*，再输入命令*gtkwave*，最后根据实验PPT给出的步骤进行操作即可观察到波形。

观察波形，发现第一个时钟周期add指令对应的输出信号为True，其余为False；第二个时钟周期sub指令对应的输出信号为True，其余为False；第三个时钟周期lw指令对应的输出信号为True，其余为False；第四个时钟周期sw指令对应的输出信号为True，其

余为False；其他时钟周期nop指令对应的输出信号为True，其余为False。由波形图可知：add、sub、lw和sw均能在运行的时钟周期内被识别到，而jal指令没有对应输出信号，也被归类为nop指令。

波形图如下图所示。

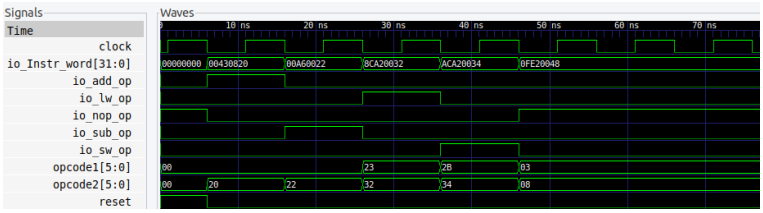


图15：输出波形

设计寄存器文件

终端打开对应项目的根目录，然后输入命令*sbt test*，再输入命令*gtkwave*，最后根据实验PPT给出的步骤进行操作即可观察到波形。

观察波形，我们可以发现，寄存器5和寄存器8的值确实被修改为0x1234，说明我设计的寄存器文件是正确的。

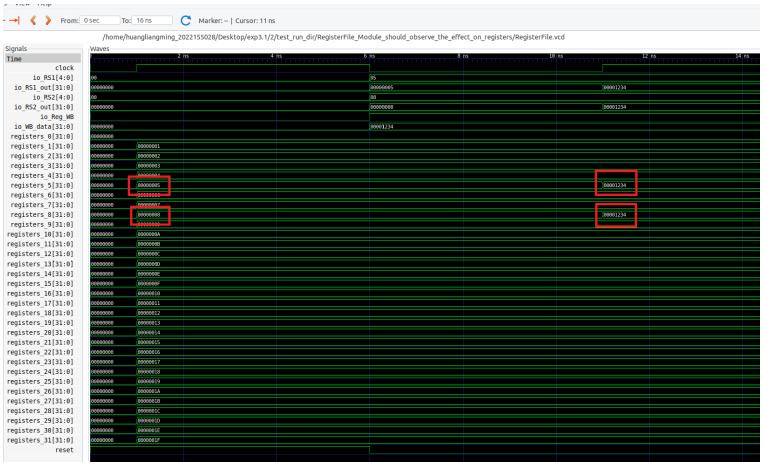


图 16：输出波形

设计指令存储器

测试波形如下图所示。四条指令的内容首先出现在wrData中表示正在写入内存，然后四条指令的内容出现在rdData和Instr_word中，说明这四条指令正在从内存中读出并被传送的译码电路中进行译码。随后我观察到和任务1中类似的指令输出信号。add、sub、lw和sw均能在运行的时钟周期内被识别到，而jal指令没有对应输出信号，被归类为nop指令。

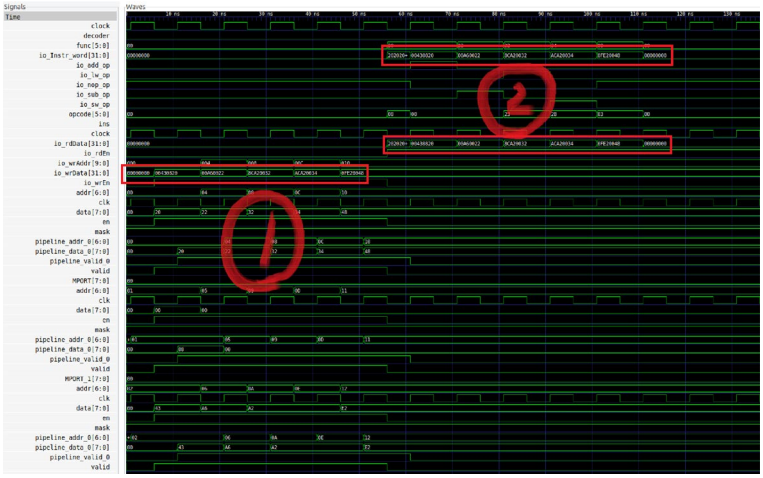


图17：测试波形

六、实验总结与体会

1) 通过本次实验，我深入理解并掌握了计算机系统中取指和指令译码设计的关键概念和实现方法。
2) 通过本次实验，我加深了对指令格式和译码逻辑的理解。
3) 通过本次实验，我了解了如何设计寄存器文件。
4) 在本次实验中，我实现了一个 32 个字的指令存储器，并将其与译码电路和寄存器文件结合起来，构建了一个完整的数据通路原型。完成上述任务让我对整个计算机系统的指令执行流程有了更全面的认识。
5) 通过本次实验，我将理论知识与实践操作相结合，加深了对计算机系统设计的理解。
6) 在本次实验开始之初，我对 Chisel 完全不了解，这对我实验的进行造成了较大的阻碍。但是，我通过浏览 PPT：Chisel 语言简介和阅读官方文档学会了如何使用 Chisel 进行编程。

指导教师批阅意见： **成绩评定：** 指导教师签字：王毅 2024 年 11 月 28 日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。
2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。