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：计算机与软件学院所有专业

指 导 教 师：罗秋明

报告人：林浩晟 学号：2022280310 班级：01

实 验 时 间：2024 年 10 月 11 日

实验报告提交时间：2024 年 10 月 25 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

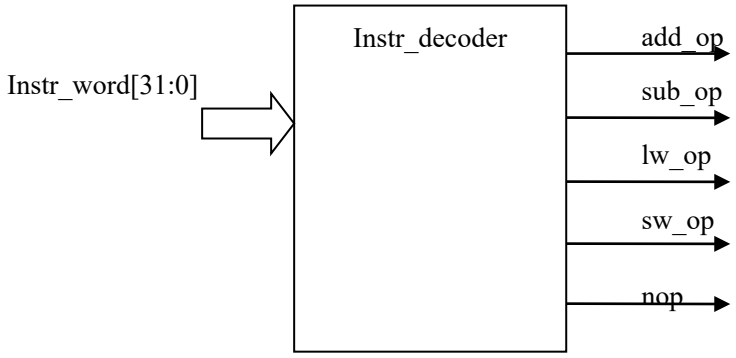
软件：Linux Chisel 开发环境

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，输出信号均为 1bit。

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 100 时，对应的输出波形。



①首先设置输入输出函数，输入为 32bit 的一个机器字，而输出为每种指令类型的布尔信号。

```
3   class InstructionDecoder extends Module {
4     val io = IO(new Bundle {
5       val Instr_word = Input(UInt(32.W)) //输入
6       //输出
7       val add_op = Output(Bool())
8       val sub_op = Output(Bool())
9       val lw_op = Output(Bool())
10      val sw_op = Output(Bool())
11      val nop_op = Output(Bool())
12    })
```

②再定义 opcode 和 func 常量并且从输入指令中提取 opcode 和 func 字段；

```
// 定义opcode和func常量
val AandS     = "b000000".U(6.W)
val ADD_FUNC  = "b100000".U(6.W)
val SUB_FUNC  = "b100010".U(6.W)
val LW_OPCODE = "b100011".U(6.W)
val SW_OPCODE = "b101011".U(6.W)
// 从输入指令中提取opcode和func字段
val opcode = io.Instr_word(31, 26)
val func   = io.Instr_word(5, 0)
```

③译码字段如下：

```
//译码
io.add_op := false.B
io.sub_op := false.B
io.lw_op := false.B
io.sw_op := false.B
io.nop_op := true.B

when(opcode === AandS) {
  when(func === ADD_FUNC) {
    io.add_op := true.B
    io.nop_op := false.B
  }.elsewhen(func === SUB_FUNC) {
    io.sub_op := true.B
    io.nop_op := false.B
  }
}.elsewhen(opcode === LW_OPCODE) {
  io.lw_op := true.B
  io.nop_op := false.B
}.elsewhen(opcode === SW_OPCODE) {
  io.sw_op := true.B
  io.nop_op := false.B
}
```

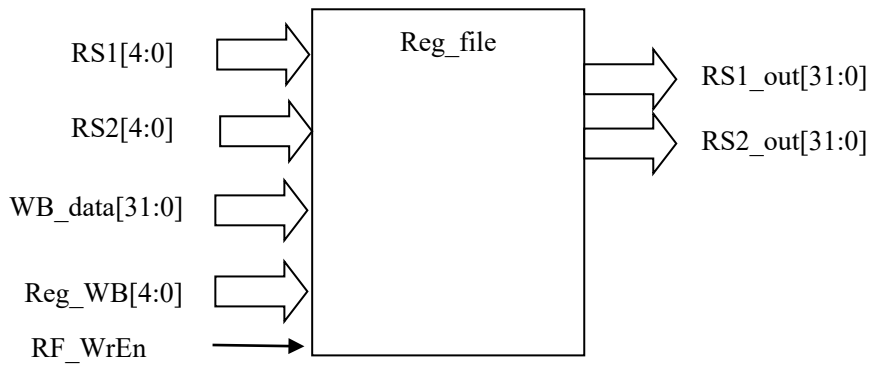
④整体代码

```
import chisel3._

class InstructionDecoder extends Module {
  val io = IO(new Bundle {
    val Instr_word = Input(UInt(32.W)) //输入
    //输出
    val add_op = Output(Bool())
    val sub_op = Output(Bool())
    val lw_op = Output(Bool())
    val sw_op = Output(Bool())
    val nop_op = Output(Bool())
  })
  // 定义 opcode 和 func 常量
  val AandS      = "b000000".U(6.W)
  val ADD_FUNC   = "b100000".U(6.W)
  val SUB_FUNC   = "b100010".U(6.W)
  val LW_OPCODE = "b100011".U(6.W)
  val SW_OPCODE = "b101011".U(6.W)
  // 从输入指令中提取 opcode 和 func 字段
  val opcode = io.Instr_word(31, 26)
  val func    = io.Instr_word(5, 0)

  //译码
```

```
    io.add_op := false.B
    io.sub_op := false.B
    io.lw_op := false.B
    io.sw_op := false.B
    io.nop_op := true.B

    when(opcode === AandS) {
      when(func === ADD_FUNC) {
        io.add_op := true.B
        io.nop_op := false.B
      }.elsewhen(func === SUB_FUNC) {
        io.sub_op := true.B
        io.nop_op := false.B
      }
    }.elsewhen(opcode === LW_OPCODE) {
      io.lw_op := true.B
      io.nop_op := false.B
    }.elsewhen(opcode === SW_OPCODE) {
      io.sw_op := true.B
      io.nop_op := false.B
    }
}
```

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。五个输入信号为 RS1、RS2、WB_data、Reg_WB、RF_WrEn，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，RF_WrEn=1 的输出波形和受影响寄存器的值。



①接下来设置寄存器文件，首先定义输入输出端口；

```
  // 定义输入输出接口
  val io = IO(new Bundle {
    // 两个读端口
    val RS1 = Input(UInt(5.W)) //5位
    val RS2 = Input(UInt(5.W))
    val RS1_out = Output(UInt(32.W))  //32位
    val RS2_out = Output(UInt(32.W))

    // 一个写端口
    val Reg_WB = Input(Bool())
    val WB_addr = Input(UInt(5.W))  //写回的寄存器地址(5位)
    val WB_data = Input(UInt(32.W))  //写回的数据(32位)
  })
```

②再设置寄存器

```
  //初始化32个寄存器，初始值为0
  val registers = RegInit(VecInit(Seq.fill(32)(0.U(32.W))))
```

③设置输出再将数据写入寄存器

```
//如果读取的是寄存器0（x0），强制返回0，因为x0永远是0
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))
//写逻辑：当写使能 (Reg_WB) 为真，并且目标地址不是寄存器0时，执行写操作
when(io.Reg_WB && io.WB_addr =/= 0.U) {
  registers(io.WB_addr) := io.WB_data  // 将数据写入指定的寄存器
}
```

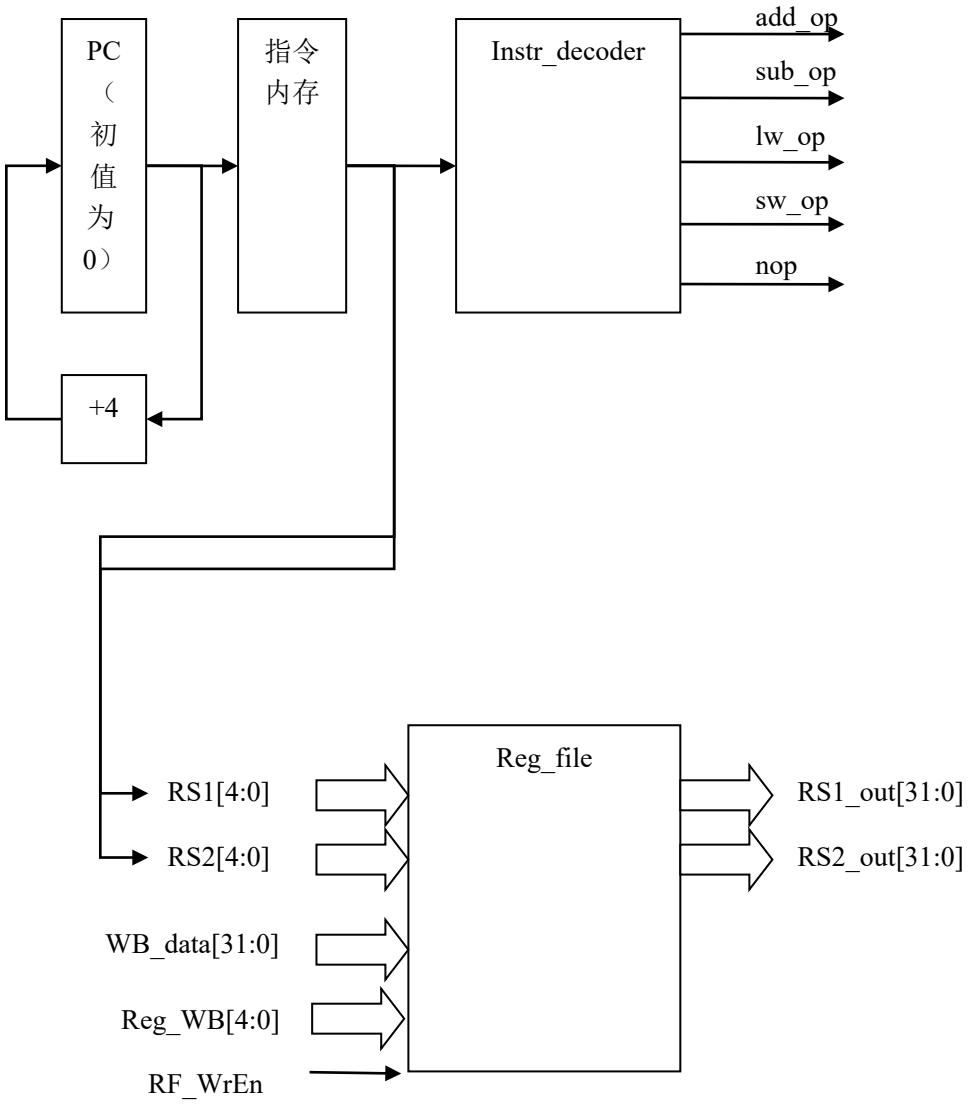
④完整代码

```
import chisel3._

class RegFile extends Module {
  // 定义输入输出接口
  val io = IO(new Bundle {
    // 两个读端口
    val RS1 = Input(UInt(5.W)) //5 位
    val RS2 = Input(UInt(5.W))
    val RS1_out = Output(UInt(32.W))   //32 位
    val RS2_out = Output(UInt(32.W))

    // 一个写端口
    val Reg_WB = Input(Bool())
    val WB_addr = Input(UInt(5.W))   //写回的寄存器地址(5 位)
    val WB_data = Input(UInt(32.W))   //写回的数据(32 位)
  })
```

```
  //初始化 32 个寄存器，初始值为 0
  val registers = RegInit(VecInit(Seq.fill(32)(0.U(32.W))))

  //如果读取的是寄存器 0（x0），强制返回 0，因为 x0 永远是 0
  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))
  //写逻辑：当写使能 (Reg_WB) 为真，并且目标地址不是寄存器 0 时，执行写操作
  when(io.Reg_WB && io.WB_addr =/= 0.U) {
    registers(io.WB_addr) := io.WB_data   // 将数据写入指定的寄存器
  }
}
```

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），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。



指令代码：

①首先设置输入输出

```
val io = IO(new Bundle {
    // 读信号接口
    val rdEna = Input(Bool())          // 读使能信号，用于控制是否进行内存读取
    val rdData = Output(UInt(32.W))    // 读取的数据输出，32 位宽

    // 写信号接口
    val wrEna = Input(Bool())          // 写使能信号，用于控制是否进行写入
    val wrAddr = Input(UInt(10.W))     // 写入地址输入（10 位），用于定位写入的存储单元
    val wrData = Input(UInt(32.W))     // 写入数据输入，32 位宽的数据
})
```

②进行写操作，将 32 位数据拆成 4 个 8 位字节存储；

```
val pcReg = RegInit(0.U(32.W))         // 定义一个 32 位的程序计数器寄存器，初始化为 0
val mem = SyncReadMem(128, UInt(8.W))  // 定义一个 128 字节的同步存储器，每单元 8 位
when(io.wrEna) {                       // 如果写使能信号为真
    // 将 32 位数据的每 8 位拆分，并写入连续的内存单元
    mem.write(io.wrAddr, io.wrData(7, 0))         // 写入最低 8 位数据
    mem.write(io.wrAddr + 1.U, io.wrData(15, 8))  // 写入第 9-16 位数据
    mem.write(io.wrAddr + 2.U, io.wrData(23, 16)) // 写入第 17-24 位数据
    mem.write(io.wrAddr + 3.U, io.wrData(31, 24)) // 写入最高 8 位数据
}
```

③再进行读操作，从连续的内存单元读取 32 位数据。

```
when(io.rdEna) {                          // 如果读使能信号为真
    // 从内存中读取连续 4 个字节的数据
    val rdData0 = mem.read(pcReg)           // 读取最低 8 位
    val rdData1 = mem.read(pcReg + 1.U)     // 读取第 9-16 位
    val rdData2 = mem.read(pcReg + 2.U)     // 读取第 17-24 位
    val rdData3 = mem.read(pcReg + 3.U)     // 读取最高 8 位

    // 将 4 个字节的数据拼接为一个 32 位数据
    io.rdData := rdData3 ## rdData2 ## rdData1 ## rdData0  // ## 用于拼接数据

    // 程序计数器增加 4，指向下一条指令的地址
    pcReg := pcReg + 4.U
}.otherwise {
    io.rdData := 0.U // 如果没有启用读操作，输出数据为 0
}
```

④完整代码

```
val io = IO(new Bundle {
    // 读信号接口
    val rdEna = Input(Bool())              // 读使能信号，用于控制是否进行内存读取
    val rdData = Output(UInt(32.W))        // 读取的数据输出，32 位宽

    // 写信号接口
    val wrEna = Input(Bool())              // 写使能信号，用于控制是否进行写入
    val wrAddr = Input(UInt(10.W))         // 写入地址输入（10 位），用于定位写入的存储单元
    val wrData = Input(UInt(32.W))         // 写入数据输入，32 位宽的数据
})

val pcReg = RegInit(0.U(32.W))             // 定义一个 32 位的程序计数器寄存器，初始化为 0
val mem = SyncReadMem(128, UInt(8.W))  // 定义一个 128 字节的同步存储器，每单元 8 位
when(io.wrEna) {                           // 如果写使能信号为真
    // 将 32 位数据的每 8 位拆分，并写入连续的内存单元
    mem.write(io.wrAddr, io.wrData(7, 0))          // 写入最低 8 位数据
    mem.write(io.wrAddr + 1.U, io.wrData(15, 8))   // 写入第 9-16 位数据
    mem.write(io.wrAddr + 2.U, io.wrData(23, 16)) // 写入第 17-24 位数据
```

```
        mem.write(io.wrAddr + 3.U, io.wrData(31, 24)) //  写入最高  8  位数据
}
when(io.rdEna) {                                    //  如果读使能信号为真
        //  从内存中读取连续  4  个字节的数据
        val rdData0 = mem.read(pcReg)                  //  读取最低  8  位
        val rdData1 = mem.read(pcReg + 1.U)         //  读取第  9-16  位
        val rdData2 = mem.read(pcReg + 2.U)         //  读取第  17-24  位
        val rdData3 = mem.read(pcReg + 3.U)         //  读取最高  8  位

        //  将  4  个字节的数据拼接为一个  32  位数据
        io.rdData := rdData3 ## rdData2 ## rdData1 ## rdData0    // ##  用于拼接数据

        //  程序计数器增加  4，指向下一条指令的地址
        pcReg := pcReg + 4.U
}.otherwise {
        io.rdData := 0.U //  如果没有启用读操作，输出数据为  0
}
```

顶层模块：

```
import chisel3._
import chisel3.util._

class TopModule extends Module {
  //定义输入输出接口
  val io = IO(new Bundle {
    //Decoder 输出信号
    val Instr_word = Output(UInt(32.W)) //指令字输出
    val add_op = Output(Bool()) //加法操作标志
    val sub_op = Output(Bool()) //减法操作标志
    val lw_op = Output(Bool())   //加载操作标志
    val sw_op = Output(Bool())   //存储操作标志
    val nop_op = Output(Bool()) //空操作标志

    //RegisterFile 的输出信号
    val RS1_out = Output(UInt(32.W)) //第一个寄存器数据输出
    val RS2_out = Output(UInt(32.W)) //第二个寄存器数据输出

    //内存的控制信号
    val rdEna = Input(Bool()) //读使能信号
    val wrAddr = Input(UInt(10.W)) //写入地址
    val wrData = Input(UInt(32.W)) //写入数据
    val wrEna = Input(Bool()) //写使能信号
  })
```

```
  // 例化 DecoderRegisterFile、Instruction Memory
  val decoder = Module(new Decoder) //指令解码模块
  val registerFile = Module(new RegisterFile) //寄存器文件模块
  val instructionMemory = Module(new Instruction) //指令存储模块

  //连接解码器的输出信号到顶层模块的接口
  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

  //连接顶层模块的控制信号到指令存储模块
  instructionMemory.io.rdEna := io.rdEna //读使能信号传递
  instructionMemory.io.wrEna := io.wrEna //写使能信号传递
  instructionMemory.io.wrAddr := io.wrAddr //写地址传递
  instructionMemory.io.wrData := io.wrData //写数据传递

  //连接指令存储模块的输出数据到寄存器文件
  registerFile.io.RS1 := instructionMemory.io.rdData(25, 21) //提取指令中的 RS1
  registerFile.io.RS2 := instructionMemory.io.rdData(20, 16) //提取指令中的 RS2

  //将指令存储模块的输出数据传递到解码器和顶层模块的指令输出
  decoder.io.Instr_word := instructionMemory.io.rdData //解码器接收指令字
  io.Instr_word := instructionMemory.io.rdData //顶层模块输出指令字

  //初始化寄存器文件的写回控制信号
  registerFile.io.Reg_WB := false.B //写回信号默认为 false
  registerFile.io.WB_data := 0.U //写回数据初始化为 0

  //将寄存器文件的输出连接到顶层模块接口
  io.RS1_out := registerFile.io.RS1_out //输出 RS1 的数据
  io.RS2_out := registerFile.io.RS2_out //输出 RS2 的数据
}

class Instruction extends Module {
  val io = IO(new Bundle {
    //内存的控制信号
    val rdEna = Input(Bool()) //读使能信号
    val rdData = Output(UInt(32.W)) //读取的数据
    val wrEna = Input(Bool()) //写使能信号
    val wrAddr = Input(UInt(10.W)) //写地址
    val wrData = Input(UInt(32.W)) //写数据
  })
```

```
  val pcReg = RegInit(0.U(32.W)) //程序计数器寄存器，初始为 0
  val mem = SyncReadMem(128, UInt(8.W)) //定义同步存储器，每个单元 8 位，大小 128 字节

  //写内存逻辑：当写使能信号为真时，将 32 位数据分成 4 个 8 位块写入连续的内存单元
  when(io.wrEna) {
    mem.write(io.wrAddr, io.wrData(7, 0)) //写入最低 8 位数据
    mem.write(io.wrAddr + 1.U, io.wrData(15, 8)) //写入第 9-16 位
    mem.write(io.wrAddr + 2.U, io.wrData(23, 16)) //写入第 17-24 位
    mem.write(io.wrAddr + 3.U, io.wrData(31, 24)) //写入最高 8 位
  }

  //读内存逻辑：当读使能信号为真时，从 4 个连续的内存单元读取数据并组合为 32 位
  when(io.rdEna) {
    val rdData0 = mem.read(pcReg) //读取第一个字节
    val rdData1 = mem.read(pcReg + 1.U) //读取第二个字节
    val rdData2 = mem.read(pcReg + 2.U) //读取第三个字节
    val rdData3 = mem.read(pcReg + 3.U) //读取第四个字节
    io.rdData := rdData3 ## rdData2 ## rdData1 ## rdData0 //将 4 个字节拼接为 32 位数据

    pcReg := pcReg + 4.U //程序计数器递增 4，指向下一条指令
  }.otherwise {
    io.rdData := 0.U //如果未启用读操作，输出 0
  }
}

class RegisterFile extends Module {
  val io = IO(new Bundle {
    //输入端口
    val RS1 = Input(UInt(5.W)) //第一个寄存器地址
    val RS2 = Input(UInt(5.W)) //第二个寄存器地址
    val Reg_WB = Input(Bool()) //写回使能信号
    val WB_data = Input(UInt(32.W)) //写回数据

    //输出端口
    val RS1_out = Output(UInt(32.W)) //第一个寄存器的数据
    val RS2_out = Output(UInt(32.W)) //第二个寄存器的数据
  })

  //初始化 32 个寄存器为 0
  val registers = RegInit(VecInit(Seq.fill(32)(0.U(32.W))))

  //读操作：根据地址读取寄存器数据
```

```
  io.RS1_out := Mux(io.RS1 === 0.U, 0.U, registers(io.RS1)) //寄存器 0 永远为 0
  io.RS2_out := Mux(io.RS2 === 0.U, 0.U, registers(io.RS2))

  //写操作：当写回使能信号为真时，将数据写入寄存器
  when(io.Reg_WB && io.RS1 =/= 0.U) {
    registers(io.RS1) := io.WB_data //写入 RS1 地址的寄存器
  }
}
```

五、实验结果

仿真测试：

①首先建立好项目，

```
user@ubuntu:~/Desktop/exp$ sbt run
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.jline.terminal.impl.exec.ExecT
erminalProvider$ReflectionRedirectPipeCreator (file:/home/user/.sbt/boot
/scala-2.12.19/org.scala-sbt/sbt/1.10.2/jline-terminal-3.24.1.jar) to co
nstructor java.lang.ProcessBuilder$RedirectPipeImpl()
WARNING: Please consider reporting this to the maintainers of org.jline.
terminal.impl.exec.ExecTerminalProvider$ReflectionRedirectPipeCreator
WARNING: Use --illegal-access=warn to enable warnings of further illegal
 reflective access operations
WARNING: All illegal access operations will be denied in a future releas
e
[info] Updated file /home/user/Desktop/exp/project/build.properties: set
 sbt.version to 1.10.2
```

②生成波形文件，

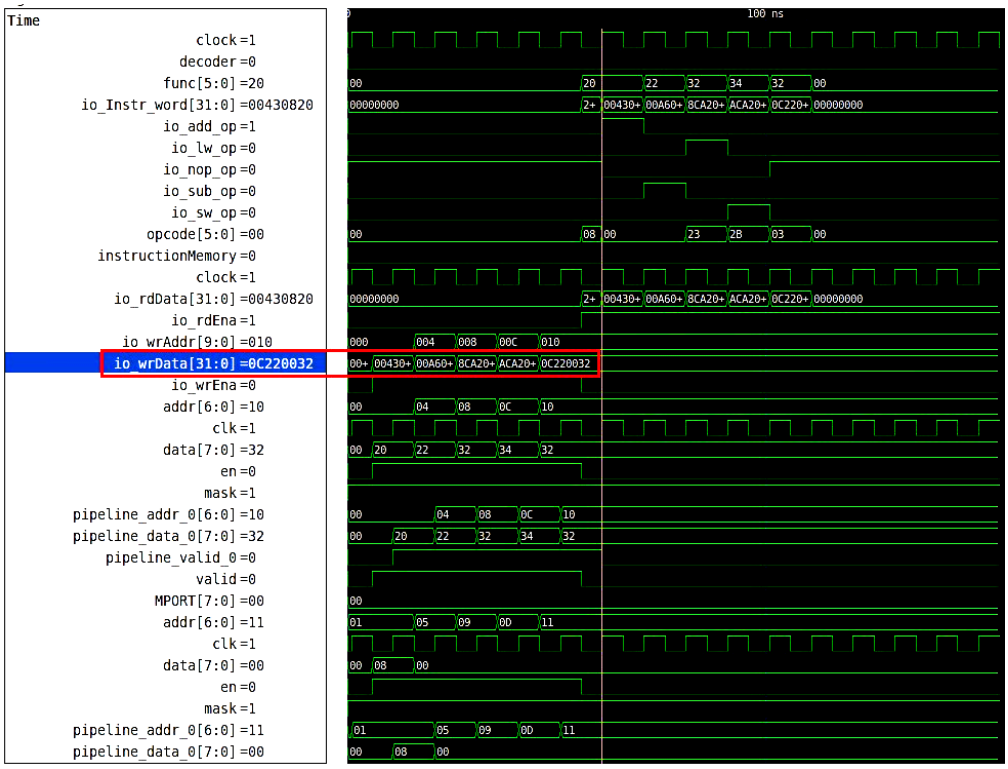
```
user@ubuntu:~/Desktop/exp$ sbt test
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.jline.terminal.impl.exec.ExecT
erminalProvider$ReflectionRedirectPipeCreator (file:/home/user/.sbt/boot
/scala-2.12.19/org.scala-sbt/sbt/1.10.2/jline-terminal-3.24.1.jar) to co
nstructor java.lang.ProcessBuilder$RedirectPipeImpl()
WARNING: Please consider reporting this to the maintainers of org.jline.
terminal.impl.exec.ExecTerminalProvider$ReflectionRedirectPipeCreator
WARNING: Use --illegal-access=warn to enable warnings of further illegal
 reflective access operations
WARNING: All illegal access operations will be denied in a future releas
e
[info] welcome to sbt 1.10.2 (Ubuntu Java 11.0.24)
```

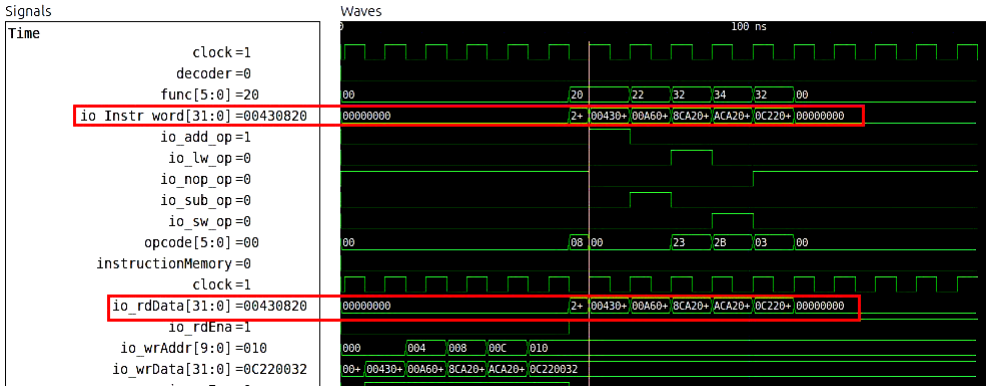
其中测试代码如下：

```
class InstructionTest extends AnyFlatSpec with ChiselScalatestTester {
    behavior of "TopModule"
    it should "pass" in {
        test(new Combination).withAnnotations(Seq(WriteVcdAnnotation)) { c =>
            val ADD = "b00000000010000110000100000100000".U(32.W)    // add R1, R2,
R3 的十六进制为 00430820
            val SUB = "b00000000101001100000000000100010".U(32.W)   // sub R0, R5, R6
的十六进制为 00A60022
            val LW    = "b10001100101000100000000000110010".U(32.W)     // lw R5,
```

```
100(R2) 的十六进制为 8CA20032
            val SW    = "b10101100101000100000000000110100".U(32.W)    // sw R5,
104(R2) 的十六进制为 ACA20034
            val JAL = "b00001100001000100000000000110010".U(32.W)    // jal 100
的十六进制为 0C220032

            c.clock.setTimeout(0)  // 初始化时钟

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

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

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

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

            // JAL 指令
            c.io.wrEna.poke(true)
            c.io.wrAddr.poke(16.U)
            c.io.wrData.poke(JAL)
            c.clock.step(1)

            // 结束
            c.io.wrEna.poke(false)
            c.io.rdEna.poke(true)
            c.clock.step(10)
        }
    }}
```

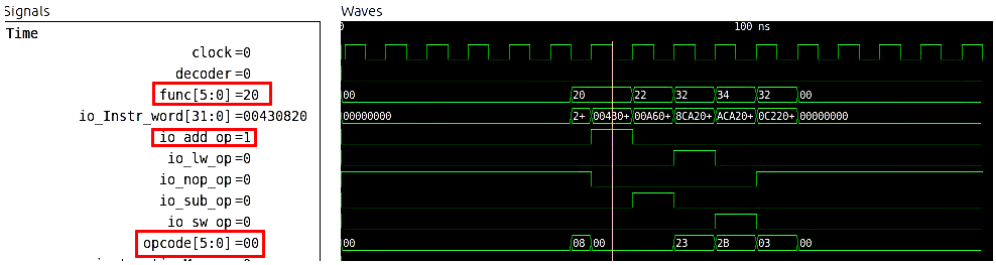
③可以发现，在前几个时钟周期中，wrData 中依次出现了各指令的十六进制，并且与输入的各指令一一对应，同时在后几个时钟周期中，Instr_word 和 rdData 依次出现各指令的十六进制，如下图所示。





④观察各指令的 opcode 、func 、op 信号

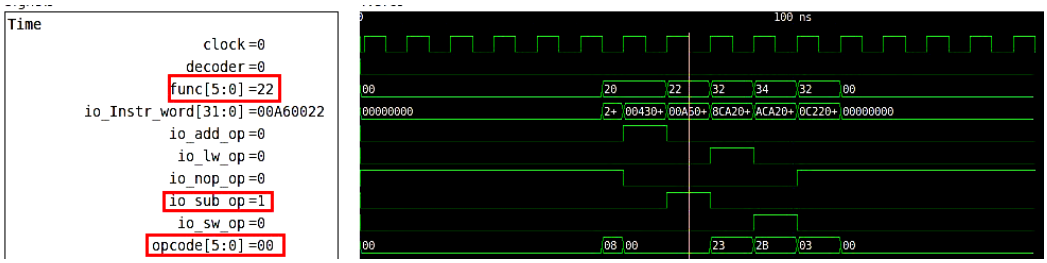
1. add 指令



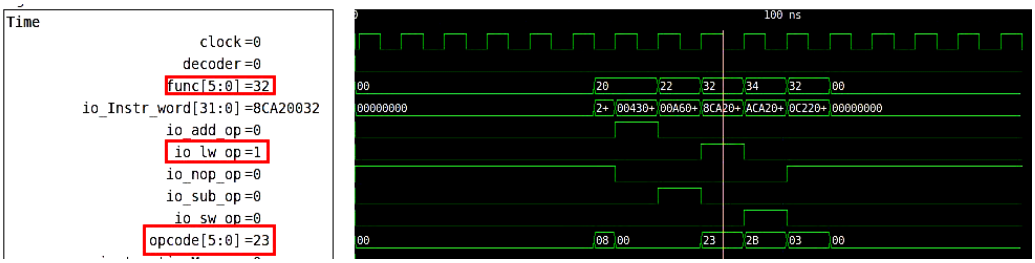
2. sub 指令

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。

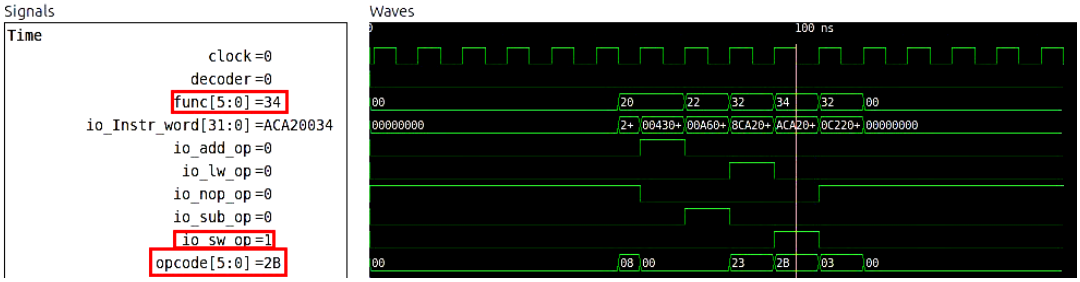
2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。



3. lw 指令

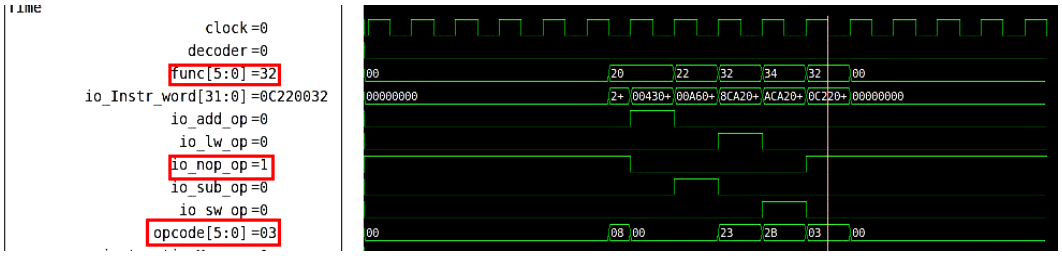


4. sw 指令



5. jal 指令

因 jal 指令不在定义的 opcode 中，故译码为 nop 指令，但 opcode 和 func 仍可正确取出。



五、实验总结与体会

实验较为成功。

在这个实验中，我们的主要目标是设计一个能顺序取指令并解码的电路。这个过程中，我学会了基础数据通路设计，以及如何用 Chisel 进行硬件描述和模拟。我们首先设计了译码器，学习了 Chisel 编程、定义端口和构建逻辑门。译码器是数据通路的核心。接着，我们实现了寄存器文件，用于存储和控制寄存器状态。最后，我们将译码器、寄存器文件与指令存储器和地址部件整合，构建了完整的数据通路原型。

这个实验让我掌握了数字电路设计的关键概念和 Chisel 基础，包括硬件描述语法、逻辑设计原则和模块连接方法。我也提高了解决问题和调试能力，学会了识别和

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。

解决硬件设计中的错误。这次实验是宝贵的学习经历，让我对 Chisel 有了更深的理解，并期待未来能进一步应用这些知识。

指导教师批阅意见： **成绩评定：** 指导教师签字： 年　月　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。