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：刘刚

报告人：__学号：__班级：__

实 验 时　间：2025 年 10 月 29 日

实验报告提交时间：2025 年 11 月 5 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

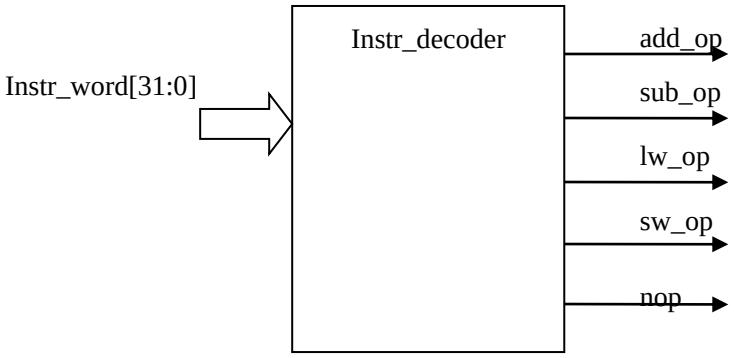
软件：Linux Chisel 开发环境

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：定义了模块的输入输出信号。它包含了一个 32 位的 Secret_word 输入信号以及几个单比特的输出信号，如 add_op、sub_op、lw_op、sw_op 和 nop，用于表示不同操作的信号

（1）JieMaProperties 类定义了一个 Bundle，其中包含了一些输入和输出信号：

- Instr_word：32 位的输入信号，代表指令字。
- add_op、sub_op、lw_op、sw_op、nop：每个都是 1 位的输出信号，表示不同的操作码。

（2）JieMa 类是一个 Module，代表了一个解码器模块。在这个模块中：

- io 是模块的接口，类型为 JieMaProperties。
- JieMaProperties 是一个根据 Instr_word 输入信号查找对应操作码的列表，使用 ListLookup 函数，如果没有找到对应的操作码，则使用 default 值。

- io.add_op、io.sub_op、io.lw_op、io.sw_op、io.nop 分别将解码器的输出连接到对应的操作码信号上，从 JieMaProperties 中获取相应的值。

代码：

```
class JieMaProperties extends Bundle {
 val Instr_word = Input(UInt(32.W))
 val add_op     = Output(UInt(1.W))
 val sub_op     = Output(UInt(1.W))
 val lw_op      = Output(UInt(1.W))
 val sw_op      = Output(UInt(1.W))
 val nop        = Output(UInt(1.W))
}

class JieMa extends Module {
 val io = IO(new JieMaProperties())
 val JieMaProperties = ListLookup(io.Instr_word, default, map)
 io.add_op := JieMaProperties(0)
 io.sub_op := JieMaProperties(1)
 io.lw_op  := JieMaProperties(2)
 io.sw_op  := JieMaProperties(3)
 io.nop    := JieMaProperties(4)
}
```

2：定义了不同操作的开和关信号，如加法操作的 startAdd 和 closeAdd，减法操作的 startSub 和 closeSub，以此类推。

（1）使用 BitPat 定义了具体的指令格式，如加法指令的 addCode，减法指令的 subCode，加载指令的 lwCode，存储指令的 swCode。

（2）定义了默认的操作码设置，其中包含了默认的开和关信号设置。

（3）创建了一个映射数组 map，用于将指令与操作码的对应关系存储起来，便于后续使用。

```
// 定义加法操作的开和关
 val startAdd  = 1.U(1.W)
 val closeAdd = 0.U(1.W)

 // 定义减法操作的开和关
 val startSub  = 1.U(1.W)
 val closeSUb = 0.U(1.W)

 // 定义加载操作的开和关
 val startLw  = 1.U(1.W)
 val closeLW  = 0.U(1.W)

 // 定义存储操作的开和关
 val startSw  = 1.U(1.W)
```

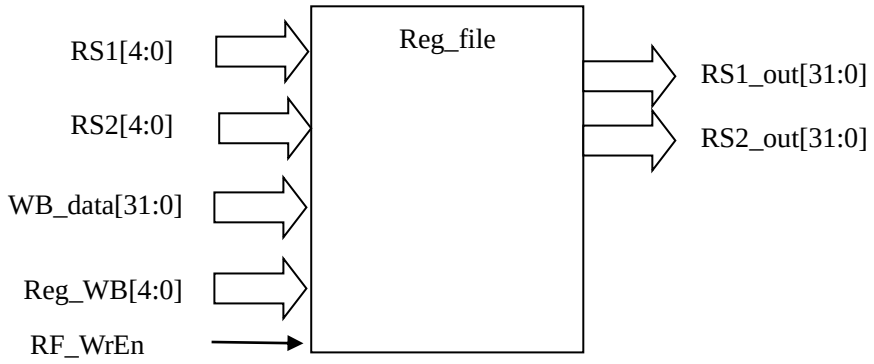
```
    val closeSw  = 0.U(1.W)

    // 定义空操作的开和关
    val startNop  = 1.U(1.W)
    val closeNop = 0.U(1.W)

    // 定义具体的指令格式，使用 BitPat 进行匹配
    def addCode = BitPat("b000000??????????????????00000100000")
    def subCode = BitPat("b000000??????????????????00000100010")
    def lwCode  = BitPat("b100011??????????????????????????????")
    def swCOde  = BitPat("b101011??????????????????????????????")

    // 定义默认的操作码设置
    val default = List(closeAdd, closeSUb, closeLW, closeSw, startNop)

    // 定义指令与操作码的映射关系
    val map = Array(
        // 指令        add_op   sub_op   lw_op   sw_op   nop
        addCode -> List(startAdd,  closeSUb, closeLW, closeSw, closeNop),
        subCode -> List(closeAdd, startSub,  closeLW, closeSw, closeNop),
        lwCode  -> List(closeAdd, closeSUb, startLw,  closeSw, closeNop),
        swCOde  -> List(closeAdd, closeSUb, closeLW, startSw,  closeNop)
    )
```

3：测试代码

```
it should "pass" in {
        test(new JieMa).withAnnotations(Seq(WriteVcdAnnotation)) {c=>
            c.io.Instr_word.poke("b000000_00010_00011_00001_00000_100000".U)
            c.clock.step(1)
            c.io.Instr_word.poke("b000000_00101_00110_00000_00000_100010".U)
            c.clock.step(1)
            c.io.Instr_word.poke("b100011_00101_00010_00000_00000_110010".U)
            c.clock.step(1)
            c.io.Instr_word.poke("b10101100101000100000000000110100".U)
            c.clock.step(1)
            c.io.Instr_word.poke("b00001100001000100000000000110010".U)
        }
    }
```

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: 输入端口包括 RS1 和 RS2，用于指定要读取的两个操作数的索引。Reg_WB 用于表示写回信号的有效性，WB_data 用于传递写回数据。输出端口包括 RS1_out 和 RS2_out，用于输出对应操作数的值。根据输入的操作数索引，模块能够从寄存器文件中读取对应的操作数的值。如果索引为 0，则输出值为 0。当写回信号有效时，模块可以将写回的数据写入寄存器文件中。写回数据会根据相应的操作数索引写入到对应的寄存器中。如果操作数索引为 0，写入的数据将被忽略，不会写入到寄存器文件中。返回数据通过 Reg_WB 信号控制更新到寄存器文件中，确保数据的正确写入。

```
val io = IO(new Bundle {
        // 输入端口定义
        val RS1 = Input(UInt(5.W))          // 第一个操作数的索引
        val RS2 = Input(UInt(5.W))          // 第二个操作数的索引


        val Reg_WB = Input(Bool())          // 写回信号
        val WB_data = Input(UInt(32.W))     // 写回的数据

        val RS1_out = Output(UInt(32.W))    // 第一个操作数的值
        val RS2_out = Output(UInt(32.W))    // 第二个操作数的值
    })

    // 定义一个包含 32 个 32 位寄存器的寄存器文件
    val registers = RegInit(VecInit((0 until 32).map(i => i.U(32.W))))

    // 从寄存器文件中读取第一个操作数的值
    io.RS1_out := Mux(io.RS1 === 0.U, 0.U, registers(io.RS1))
    // 从寄存器文件中读取第二个操作数的值
    io.RS2_out := Mux(io.RS2 === 0.U, 0.U, registers(io.RS2))
```

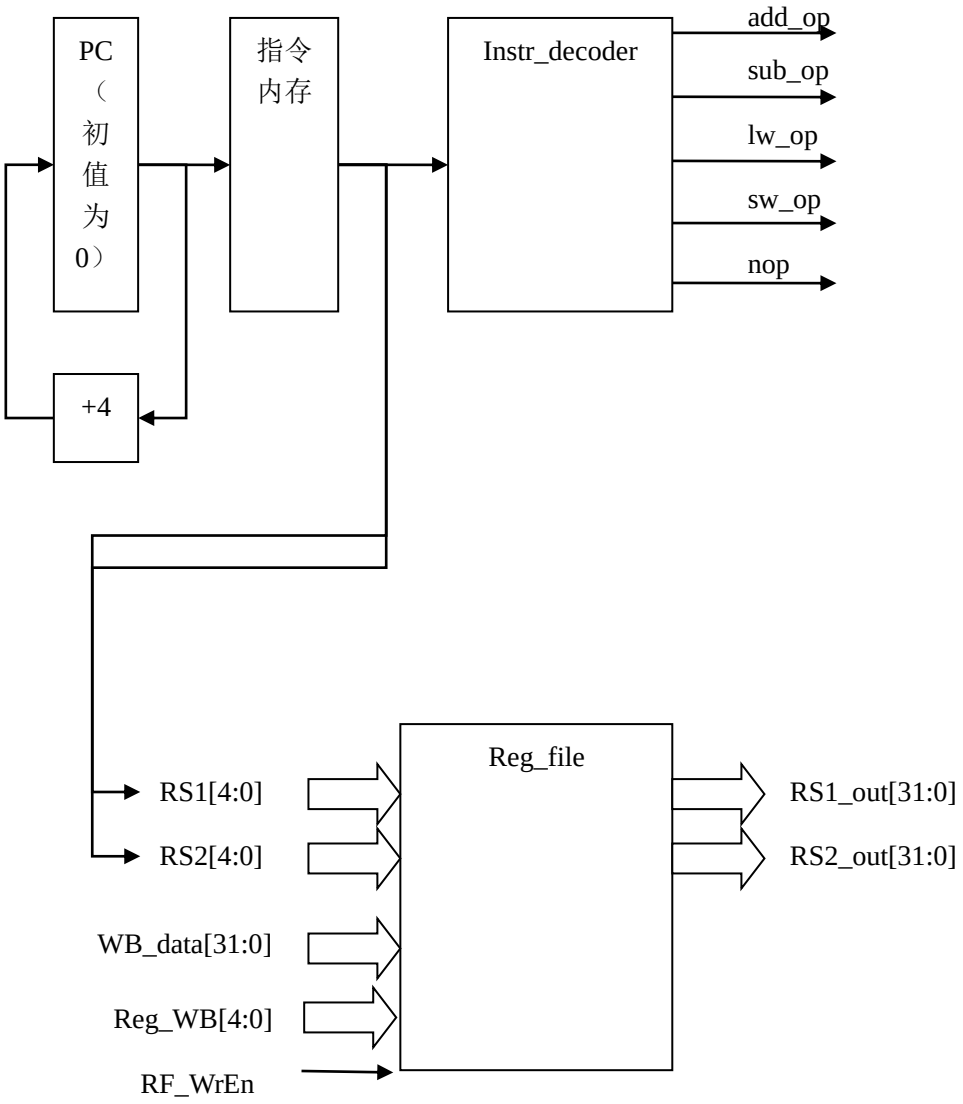
```
    // 当写回信号有效时，将写回的数据写入相应的寄存器
    when(io.Reg_WB) {
        registers(io.RS1) := Mux(io.RS1 === 0.U, 0.U, io.WB_data)
        registers(io.RS2) := Mux(io.RS2 === 0.U, 0.U, io.WB_data)
    }
```

2：测试代码

```
// 设置输入信号
        c.io.RS1.poke(5.U)
        c.io.RS2.poke(8.U)
        c.io.WB_data.poke(0x1234.U)



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

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。



1：程序计数器控制指令的执行顺序，指令解码器识别指令类型并生成相应的控制信号，寄存器文件则用于存储和操作寄存器数据。

（1）PC 模块：

模块接口：

- 输入：
  - pcIn: 32 位无符号整数，用作新的程序计数器值输入。
  - pc_sel: 1 位无符号整数，用于选择新的程序计数器值来源。
- 输出：
  - pcOut: 32 位无符号整数，用作输出的程序计数器值。

功能：

- 使用寄存器 pc 来存储当前的程序计数器值，并初始化为 0。
- 根据输入的 pc_sel 信号，选择新的程序计数器值来源：如果 pc_sel 为 1，则更新程序计数器值为 pcIn，否则程序计数器值加一。

（2）Junction 模块：

模块接口：

- 输入：
  - wrAddr: 5 位无符号整数，用作内存写入地址。
  - wrData: 32 位无符号整数，用作内存写入数据。
  - wrEna: 1 位无符号整数，用作内存写使能信号。
  - pcIn: 5 位无符号整数，用作 PC 模块的输入程序计数器值。
  - pc_sel: 1 位无符号整数，用作 PC 模块的程序计数器值来源选择信号。
- 输出：
  - add_op, sub_op, lw_op, sw_op, nop: 指令解码器的操作信号输出。
  - RS1_out, RS2_out: 寄存器文件的输出。

```
// 定义模块的输入输出接口
val io = IO(new Bundle {
    val pcIn  = Input(UInt(32.W))   // 输入：新的程序计数器值
    val pc_sel = Input(UInt(1.W))   // 输入：程序计数器值来源选择信号
    val pcOut = Output(UInt(32.W))  // 输出：程序计数器值
})

// 定义一个寄存器用于存储程序计数器的值，初始值为 0
val pc = RegInit(0.U(5.W))

// 将输出连接到寄存器的值
io.pcOut := pc

// 根据输入的选择信号更新程序计数器的值
pc := Mux(io.pc_sel === 1.U, io.pcIn, io.pcOut + 1.U)

// 定义模块的输入输出接口
val io = IO(new Bundle {
    val wrAddr  = Input(UInt(5.W))    // 输入：内存写入地址
    val wrData  = Input(UInt(32.W))   // 输入：内存写入数据
    val wrEna   = Input(UInt(1.W))    // 输入：内存写使能信号

    val add_op  = Output(UInt(1.W))   // 输出：加法操作信号
    val sub_op  = Output(UInt(1.W))   // 输出：减法操作信号
    val lw_op   = Output(UInt(1.W))   // 输出：加载操作信号
    val sw_op   = Output(UInt(1.W))   // 输出：存储操作信号
    val nop     = Output(UInt(1.W))   // 输出：空操作信号
    val RS1_out = Output(UInt(32.W))  // 输出：寄存器 1 的输出
    val RS2_out = Output(UInt(32.W))  // 输出：寄存器 2 的输出

    val pcIn = Input(UInt(5.W))       // 输入：PC 模块的输入程序计数器值
    val pc_sel = Input(UInt(1.W))     // 输入：PC 模块的程序计数器值来源选择
```

```
信号
    })

    // 实例化其他模块：Decoder、RegFile 和 Mem
    val decoder = Module(new Decoder())
    val regFile = Module(new RegFile())
    val mem     = Module(new Mem())
    val pc      = Module(new PC())

    // 将 PC 模块的输入连接到 Junction 模块的输入
    pc.io.pc_sel := io.pc_sel
    pc.io.pcIn := io.pcIn

    // 将内存模块的地址、数据和使能信号连接到 Junction 模块的输入
    mem.io.rdAddr := pc.io.pcOut
    mem.io.wrAddr := io.wrAddr
    mem.io.wrData := io.wrData
    mem.io.wrEna  := io.wrEna

    // 将 Decoder 模块的输出连接到 Junction 模块的输出
    decoder.io.Instr_word := mem.io.rdData
    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    := decoder.io.nop

    // 将寄存器文件模块的输出连接到 Junction 模块的输出
    regFile.io.RS1 := mem.io.rdData(25, 21)
    regFile.io.RS2 := mem.io.rdData(20, 16)
    regFile.io.Reg_WB := mem.io.rdData(15, 11)
    regFile.io.RF_wrEn := decoder.io.add_op | decoder.io.sub_op

    // 根据指令解码结果进行寄存器操作和数据计算
    when(regFile.io.Reg_WB === 1.U) {
        when(decoder.io.add_op === 1.U) {
            regFile.io.WB_data := regFile.io.RS1 + regFile.io.RS2
        }.otherwise {
            regFile.io.WB_data := regFile.io.RS1 - regFile.io.RS2
        }
    }.otherwise {
        regFile.io.WB_data := 0.U
    }
```

```
    // 将寄存器文件模块的输出连接到 Junction 模块的输出
    io.RS1_out := regFile.io.RS1_out
    io.RS2_out := regFile.io.RS2_out
```

2：可以根据输入的地址读取数据，并根据写使能信号和地址写入数据

```
// 定义模块的输入输出接口
    val io = IO(new Bundle {
        val rdAddr = Input(UInt(5.W))    // 读取地址输入
        val rdData = Output(UInt(32.W))  // 读取数据输出
        val wrAddr = Input(UInt(5.W))    // 写入地址输入
        val wrData = Input(UInt(32.W))   // 写入数据输入
        val wrEna  = Input(UInt(1.W))    // 写使能输入
    })

    // 创建一个大小为 32 的同步读写内存，每个数据位宽为 32 位
    val mem = SyncReadMem(32, UInt(32.W))

    // 将读取数据输出连接到内存读取操作
    io.rdData := mem.read(io.rdAddr)

    // 当写使能为高时执行写操作
    when(io.wrEna === 1.U) {
        mem.write(io.wrAddr, io.wrData)
    }
```

3：测试代码

```
test(new Junction).withAnnotations(Seq(WriteVcdAnnotation)) { c =>
            // 设置写入地址为 0
            c.io.wrAddr.poke(0.U)
            // 使能写入
            c.io.wrEna.poke(1.U)
            // 写入数据
            c.io.wrData.poke("b000000_00010_00011_00001_00000_100000".U)

            // 模拟一个时钟周期
            c.clock.step(1)

            // 设置写入地址为 1
            c.io.wrAddr.poke(1.U)
            // 使能写入
            c.io.wrEna.poke(1.U)
            // 写入数据
            c.io.wrData.poke("b000000_00101_00110_00000_00000_100010".U)
```

```
// 模拟一个时钟周期
c.clock.step(1)

// 设置写入地址为 2
c.io.wrAddr.poke(2.U)
// 使能写入
c.io.wrEna.poke(1.U)
// 写入数据
c.io.wrData.poke("b100011_00101_00010_00000_00000_110010".U)

// 模拟一个时钟周期
c.clock.step(1)

// 设置写入地址为 3
c.io.wrAddr.poke(3.U)
// 使能写入
c.io.wrEna.poke(1.U)
// 写入数据
c.io.wrData.poke("b10101100101000100000000000110100".U)

// 模拟一个时钟周期
c.clock.step(1)

// 设置 pcIn 为 0
c.io.pcIn.poke(0.U)
// 设置 pc_sel 为 1
c.io.pc_sel.poke(1.U)

// 模拟两个时钟周期
c.clock.step(1)
c.clock.step(1)

// 设置 pc_sel 为 0
c.io.pc_sel.poke(0.U)
// 模拟两个时钟周期
c.clock.step(1)
c.clock.step(1)
// 设置输入信号
c.io.RS1.poke(5.U)
c.io.RS2.poke(8.U)
c.io.WB_data.poke(0x1234.U)
```

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

五、实验结果

1. 设计译码电路

1.1：运行 sbt run 搭建项目

虽然有报错，但依然能成功生成

```
To turn this message off, check the last line of $HOME/.bashrc
user@ubuntu:~/Desktop/yubaobao/test/test2$ sbt run
[info] welcome to sbt 1.10.1 (Private Build Java 1.8.0_422)
[info] loading project definition from /home/user/Desktop/yubaobao/test/test2/pr
oject
[info] loading settings for project test2 from build.sbt ...
[info] set current project to test2 (in build file:/home/user/Desktop/yubaobao/t
est/test2/)
[error] java.lang.RuntimeException: No main class detected.
[error]         at scala.sys.package$.error(package.scala:30)
[error] stack trace is suppressed; run last Compile / bgRun for the full output
[error] (Compile / bgRun) No main class detected.
[error] Total time: 1 s, completed Nov 21, 2024 11:19:25 PM
```

1.2：运行 sbt test 生成波形文件

```
user@ubuntu:~/Desktop/yubaobao/test/test2$ sbt test
[info] welcome to sbt 1.10.1 (Private Build Java 1.8.0_422)
[info] loading project definition from /home/user/Desktop/yubaobao/test/tes
oject
[info] loading settings for project test2 from build.sbt ...
[info] set current project to test2 (in build file:/home/user/Desktop/yubao
est/test2/)
[info] compiling 1 Scala source to /home/user/Desktop/yubaobao/test/test2/t
/scala-2.12/test-classes ...
[info] DecoderTest:
[info] JieMa
[info] - should pass
[info] Run completed in 3 seconds, 716 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
```

1.3：观察波形图

（1）初始状态如下，wrData 中依次出现了各指令的十六进制，与输入的各指令一一对应。

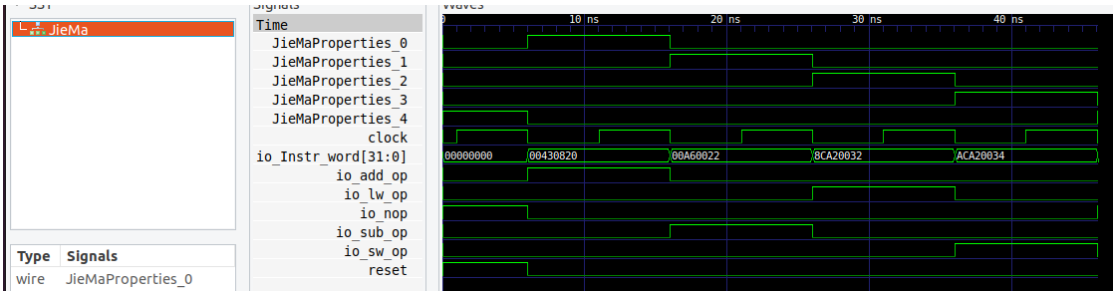
add R1, R2, R3 的十六进制为 00430820

sub R0, R5, R6 的十六进制为 00A60022

lw R5, 100(R2) 的十六进制为 8CA20032

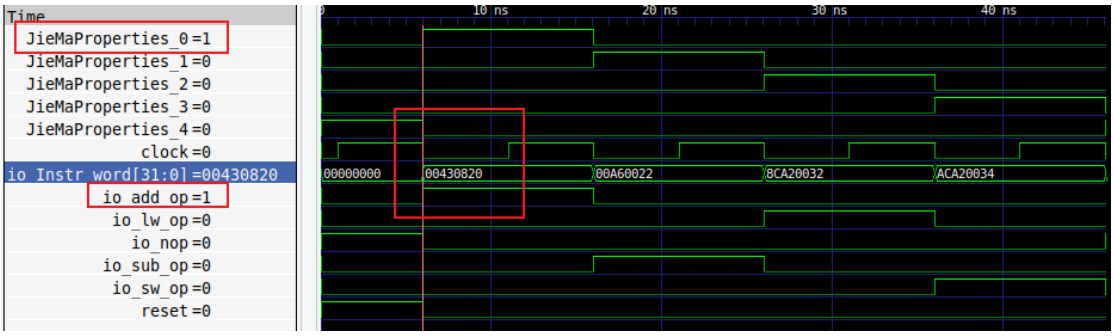
sw R5, 104(R2) 的十六进制为 ACA20034

jal 100 的十六进制为 0C220032

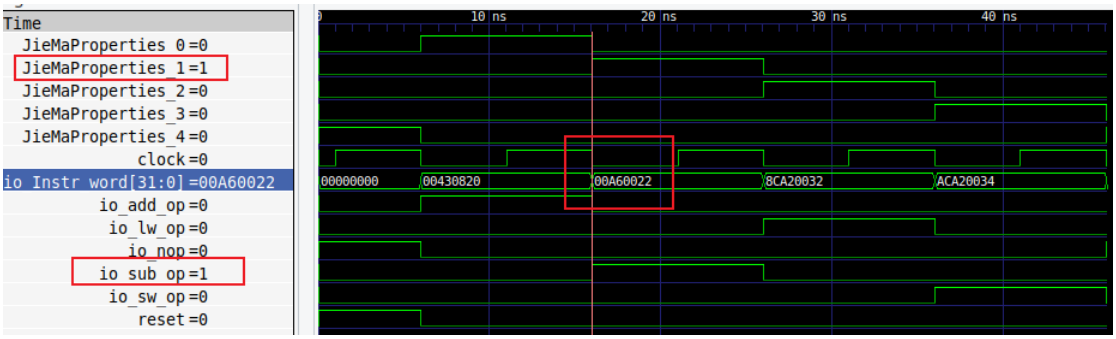


（2）观察各指令

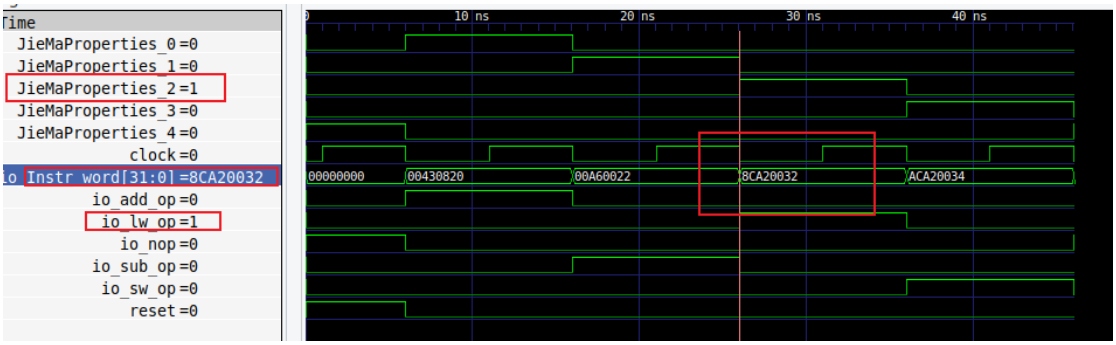
add 指令



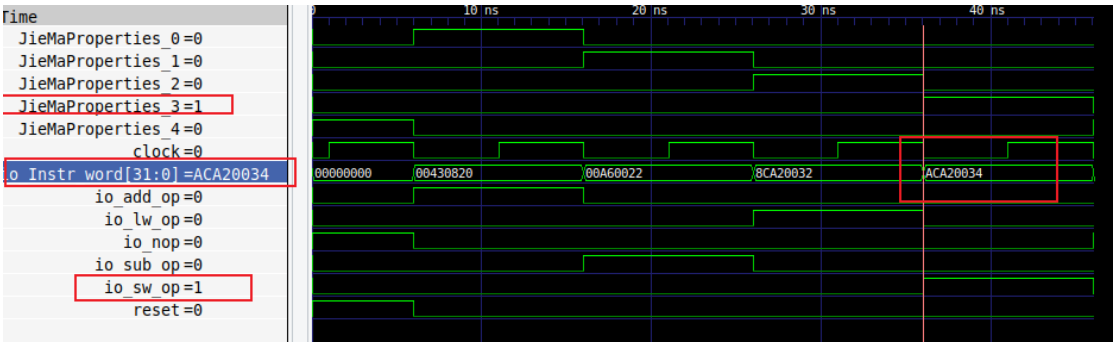
sub 指令



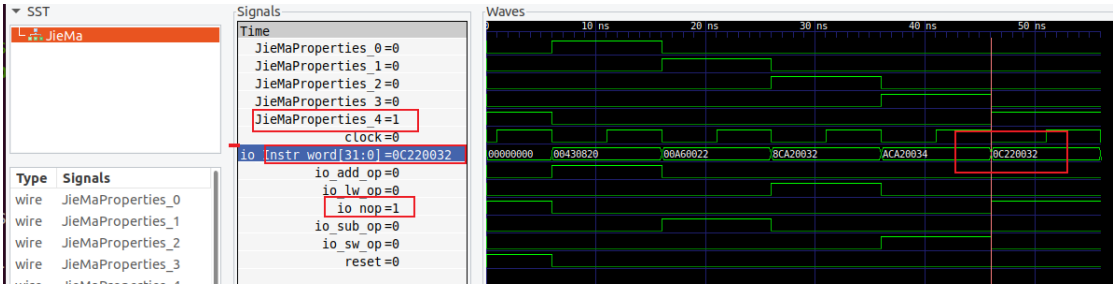
lw 指令



sw 指令



Jal 指令，属于其他指令，所以输出 nop



2. 设计寄存器文件

2.1：运行 sbt run 搭建项目

虽然报错，但依然能生成

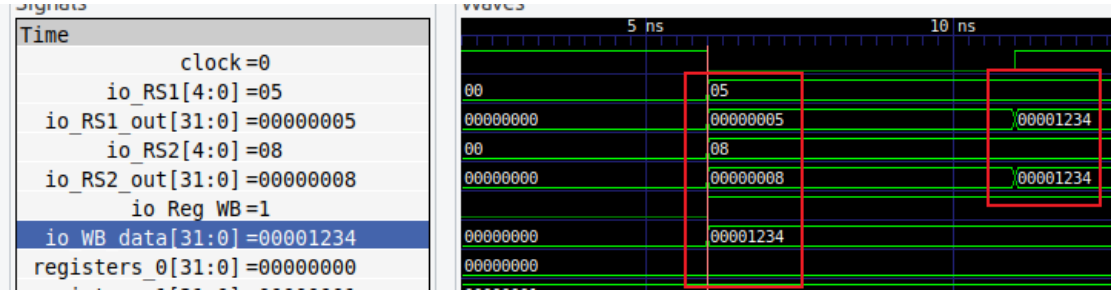
```
info] compiling 1 Scala source to /home/user/Desktop/yubaobao/test/test1/target
scala-2.12/classes ...
error] java.lang.RuntimeException: No main class detected.
error]         at scala.sys.package$.error(package.scala:30)
error] stack trace is suppressed; run last Compile / bgRun for the full output
error] (Compile / bgRun) No main class detected.
error] Total time: 7 s, completed Nov 21, 2024 11:21:39 PM
ser@ubuntu:~/Desktop/yubaobao/test/test1$
```

2.2：运行 sbt test 生成波形图文件

```
user@ubuntu:~/Desktop/yubaobao/test/test1$ sbt test
[info] welcome to sbt 1.10.1 (Private Build Java 1.8.0_422)
[info] loading project definition from /home/user/Desktop/yubaobao/test/t
t1/project
[info] loading settings for project test1 from build.sbt ...
[info] set current project to test1 (in build file:/home/user/Desktop/yub
bao/test/test1/)
[info] compiling 1 Scala source to /home/user/Desktop/yubaobao/test/test1
arget/scala-2.12/classes ...
[info] compiling 1 Scala source to /home/user/Desktop/yubaobao/test/test1
arget/scala-2.12/test-classes ...
[info] RegisterFileTest:
[info] RegisterFile Module
[info] - should observe the effect on registers
[info] Run completed in 4 seconds, 844 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 15 s, completed Nov 22, 2024 5:09:32 AM
```

2.3：观察波形图

观察到 RS1 = 5，RS2 = 8，WB_data = 0x1234 ，并且之后 0x1234 写入 R5 和 R8



3. 实现一个 32 个字的指令存储器

3.1：运行 sbt run 搭建项目

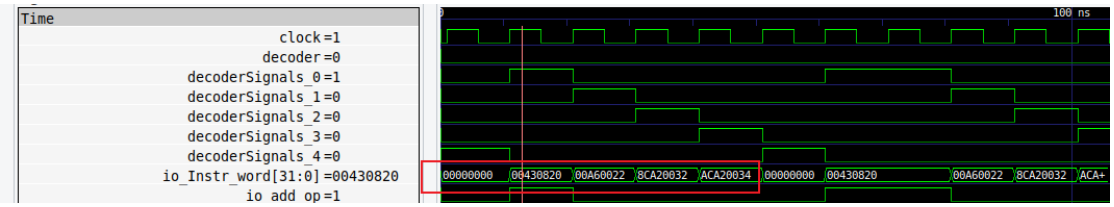
```
user@ubuntu:~/Desktop/yubaobao/test/test3$ sbt run
[info] welcome to sbt 1.10.1 (Private Build Java 1.8.0_422)
[info] loading project definition from /home/user/Desktop/yubaobao/test/test3/pr
oject
[info] loading settings for project test3 from build.sbt ...
[info] set current project to test3 (in build file:/home/user/Desktop/yubaobao/t
est/test3/)
[info] compiling 5 Scala sources to /home/user/Desktop/yubaobao/test/test3/targe
t/scala-2.12/classes ...
```

3.2：运行 sbt test 生成波形图文件

```
user@ubuntu:~/Desktop/yubaobao/test/test3$ sbt test
[info] welcome to sbt 1.10.1 (Private Build Java 1.8.0_422)
[info] loading project definition from /home/user/Desktop/yubaobao/test/test3/pr
oject
[info] loading settings for project test3 from build.sbt ...
[info] set current project to test3 (in build file:/home/user/Desktop/yubaobao/t
est/test3/)
[info] compiling 2 Scala sources to /home/user/Desktop/yubaobao/test/test3/targe
t/scala-2.12/test-classes ...
[info] MemTest:
[info] Junction

SUCCESS!!!

[info] - should pass
[info] JunctionTest:
[info] Junction
[info] - should pass
[info] Run completed in 5 seconds, 585 milliseconds.
[info] Total number of tests run: 2
[info] Suites: completed 2, aborted 0
[info] Tests: succeeded 2, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 13 s, completed Nov 22, 2024 5:26:32 AM
```

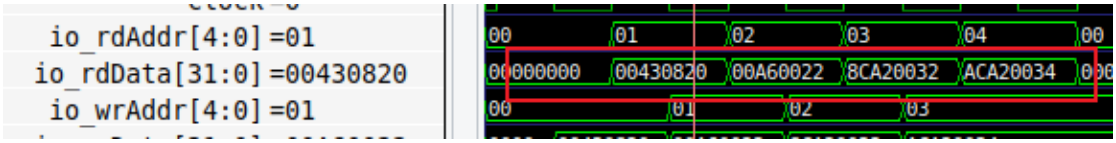
3.3：观察波形图

Add 指令执行过程波形



依次出现四条指令

add R1, R2, R3 的十六进制为 00430820
sub R0, R5, R6 的十六进制为 00A60022
lw R5, 100(R2) 的十六进制为 8CA20032
sw R5, 104(R2) 的十六进制为 ACA20034



五、实验总结与体会

译码器设计与实现：

译码器的设计是整个数据通路设计的核心部分，负责将输入的机器指令转换为控制信号，指导数据通路的其他部分如何响应这些指令。通过实现不同指令的解码，我学会了如何处理和分析二进制编码的信息，并将其映射到具体的硬件操作上。

寄存器文件的构建：

寄存器文件作为 CPU 中的核心部件，存储了运行时的临时数据。实验中，我不仅实现了基本的读写操作，还确保了 0 号寄存器的特殊行为。通过这部分的设计和测试，我对寄存器文件的工作原理和其在整个处理器中的作用有了更深的理解。

指令存储器的集成：

指令存储器的实现让我学习到了如何在 Chisel 中处理内存操作，尤其是如何初始化存储器和执行读取操作。通过将指令存储器、译码器和寄存器文件结合起来，我更直观地理解了 PC（程序计数器）的更新如何影响处理器逐条执行指令。

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。