实验三 取指令和指令译码设计

实验目的

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。

实验任务

分为以下三个任务：

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。

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。

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）。

实验环境

本次实验涉及的软硬件环境如下：

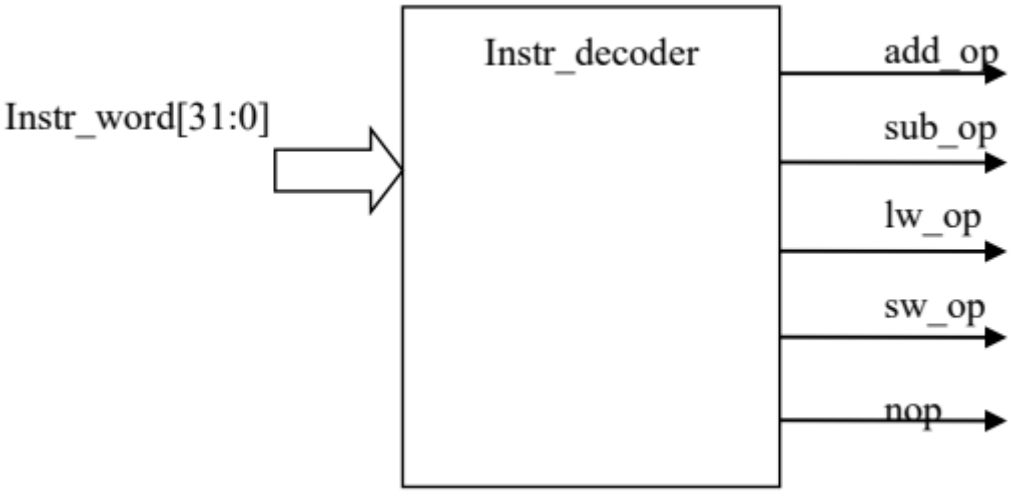
- 硬件：桌面PC
- 软件：Ubuntu 20.04
- ……

注：实验谬误和困难如下

1. 实验第一部分：
    - `jal`应该是立即数寻址，不应有寄存器的参与，所以应该是`jal 100`(具体可以参考green card)
    - 需要注意的是，译码器模块的输出均为1bit数据，比如输入指令是`add`，那么`add_op`应该是`1.U(1.W)`，其余都是`0.U(1.W)`
2. 实验第二部分：
    - 设计图的寄存器文件只给出了写入的数据(`WB_data`)和写入使能信号(`Reg_WB`)，并没有给出写入地址(`WB_addr`)，所以需要额外加上
    - 寄存器可以通过`reg := 1.U`形式赋初值
3. 实验第三部分：
    - 指令内存实际上就是一个内存(DRAM)，在规约上我们得使用`SyncReadMem`这个内置模块(*而不是寄存器*)，需要注意的是，Mem可以创建一个寄存器集合(异步读，同步写)
    - 测试中的`poke()`给予的激励会一直存在，故不能直接对PC调用`poke`，否则PC值并不会变化

译码器的设计

译码器如图：



设计步骤：

1.设计add、sub、lw、sw指令的比特模式，以及每条指令译码后的输出

2.设计译码器的接口，以及译码器的内部实现

本次译码器的设计文件包括 **Instructions.scala**（步骤1的内容）和 **Decoder.scala**（步骤2的内容）文件。

Instructions.scala：

```
package mydesign

import chisel3._
import chisel3.util._
import chisel3.util.BitPat

object Instructions {
    // add_op
    val add_on  = 1.U(1.W)
    val add_off = 0.U(1.W)
    // sub_op
    val sub_on  = 1.U(1.W)
    val sub_off = 0.U(1.W)
    // lw_op
    val lw_on   = 1.U(1.W)
    val lw_off  = 0.U(1.W)
    // sw_op
    val sw_on   = 1.U(1.W)
    val sw_off  = 0.U(1.W)
    // nop
    val nop_on  = 1.U(1.W)
    val nop_off = 0.U(1.W)

    def ADD = BitPat("b000000??????????????00000100000")
    def SUB = BitPat("b000000??????????????00000100010")
    def LW  = BitPat("b100011??????????????????????????")
    def SW  = BitPat("b101011??????????????????????????")
```

```
    val default = List(add_off, sub_off, lw_off, sw_off, nop_on)
    val map = Array(
        //          add_op   sub_op   lw_op   sw_op   nop
        ADD -> List(add_on,  sub_off, lw_off, sw_off, nop_off),
        SUB -> List(add_off, sub_on,  lw_off, sw_off, nop_off),
        LW  -> List(add_off, sub_off, lw_on,  sw_off, nop_off),
        SW  -> List(add_off, sub_off, lw_off, sw_on,  nop_off)
    )
}
```

Decoder.scala:

```
package mydesign

import chisel3._
import chisel3.util._
import chisel3.util.BitPat
import Instructions._

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

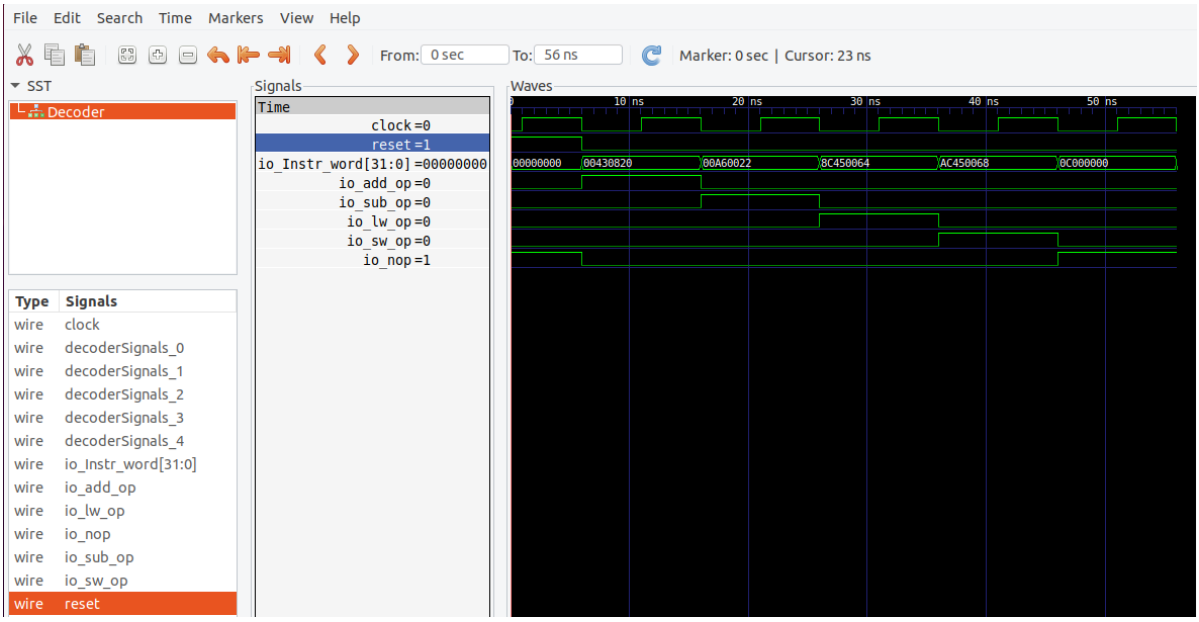
译码器的测试文件位于**test**目录下，为 **DecoderTest.scala** 文件

DecoderTest.scala:

```
package mydesign

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

class DecoderTest extends AnyFlatSpec with ChiselScalatestTester {
    behavior of "Decoder"
    it should "pass" in {
        test(new Decoder).withAnnotations(Seq(WriteVcdAnnotation)) {c=>
            c.clock.step(1)
            c.io.Instr_word.poke("b000000_00000_00000_00000_00000_000000".U)
```

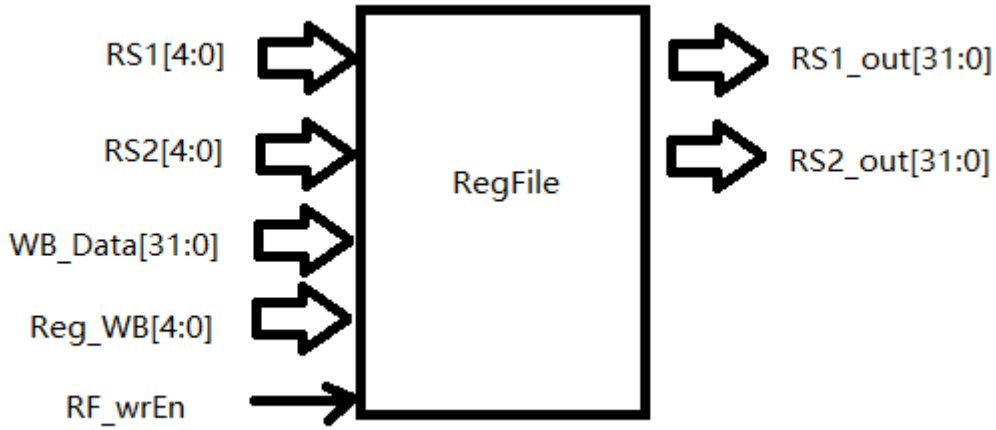
```
            c.clock.step(1)

            c.io.Instr_word.poke("b000000_00010_00011_00001_00000_100000".U)
            c.io.add_op.expect(1.U)
            c.io.sub_op.expect(0.U)
            c.io.lw_op.expect(0.U)
            c.io.sw_op.expect(0.U)
            c.io.nop.expect(0.U)

            c.clock.step(1)
            c.io.Instr_word.poke("b000000_00101_00110_00000_00000_100010".U)
            c.io.add_op.expect(0.U)
            c.io.sub_op.expect(1.U)
            c.io.lw_op.expect(0.U)
            c.io.sw_op.expect(0.U)
            c.io.nop.expect(0.U)

            c.clock.step(1)
            c.io.Instr_word.poke("b100011_00010_00101_00000_00001_100100".U)
            c.io.add_op.expect(0.U)
            c.io.sub_op.expect(0.U)
            c.io.lw_op.expect(1.U)
            c.io.sw_op.expect(0.U)
            c.io.nop.expect(0.U)

            c.clock.step(1)
            c.io.Instr_word.poke("b101011_00010_00101_00000_00001_101000".U)
            c.io.add_op.expect(0.U)
            c.io.sub_op.expect(0.U)
            c.io.lw_op.expect(0.U)
            c.io.sw_op.expect(1.U)
            c.io.nop.expect(0.U)

            c.clock.step(1)
            c.io.Instr_word.poke("b000011_00000_00000_00000_00000_000000".U)
            c.io.add_op.expect(0.U)
            c.io.sub_op.expect(0.U)
            c.io.lw_op.expect(0.U)
            c.io.sw_op.expect(0.U)
            c.io.nop.expect(1.U)

            c.clock.step(1)

            println("SUCCESS!!!\n")
        }
    }
}
```

其测试的波形为：



寄存器文件设计

寄存器文件如图：



设计思路：

1.按寄存器文件图定义接口

2.使用`Vec()`定义32个32位的寄存器，并为每个寄存器赋初值

3.连线

寄存器文件的设计在 **RegFile.scala** 文件中。

RegFile.scala：

```
package mydesign

import chisel3._

class RegFileIO extends Bundle {
    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))
    val WB_data = Input(UInt(32.W))
```

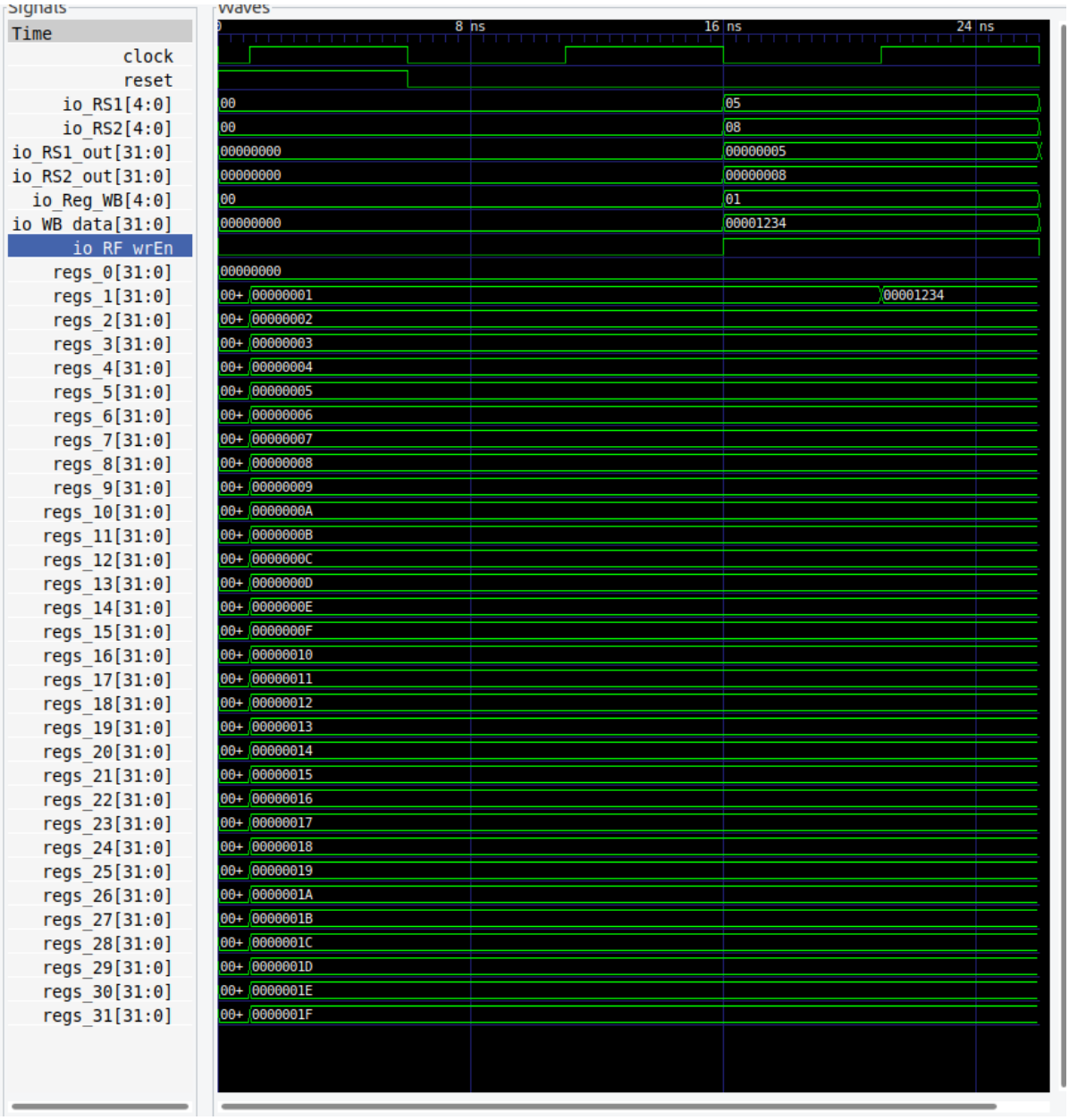
```
    val Reg_WB  = Input(UInt(5.W))
    val RF_wrEn = Input(UInt(1.W))
}

class RegFile extends Module {
    val io = IO(new RegFileIO())
    val regs = Reg(Vec(32, UInt(32.W)))

    for(i <- 0 to 31) {     // 初始化寄存器
        regs(i) := i.U
    }

    io.RS1_out := Mux(io.RS1.orR, regs(io.RS1), 0.U)
    io.RS2_out := Mux(io.RS2.orR, regs(io.RS2), 0.U)

    when(io.wrEn === 1.U) {
        regs(io.Reg_WB) := io.WB_data
    }
}
```

译码器的测试文件位于**test**目录下，为 **RegFileTest.scala** 文件

RegFileTest.scala：

```
package mydesign

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

class RegFileTest extends AnyFlatSpec with ChiselScalatestTester {
    behavior of "RegFile"
    it should "pass" in {
        test(new RegFile).withAnnotations(Seq(WriteVcdAnnotation)) {c=>
            c.clock.step(1)
            c.io.RS1.poke(5.U)
            c.io.RS2.poke(8.U)
            c.io.WB_data.poke("h1234".U)
            c.io.Reg_WB.poke(true.B)
            c.io.wrAddr.poke(1.U)

            c.clock.step(1)
            c.io.RS1_out.expect(5.U)
            c.io.RS2_out.expect(8.U)
            c.clock.step(1)

            println("\nSUCCESS!!!\n")
        }
    }
}
```

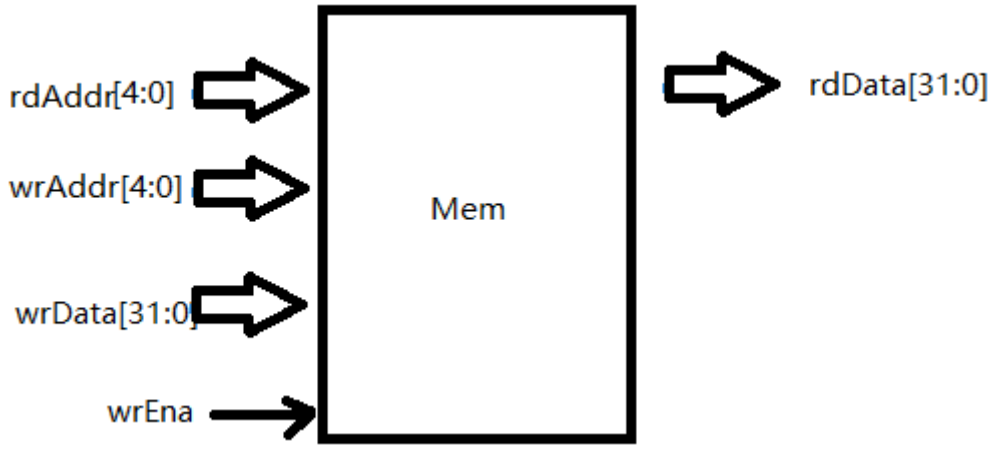
其测试的波形为：



指令存储器设计并将各个模块连接

指令存储器设计

指令存储器如图：



设计思路：定义接口，使用`SyncReadMem()`定义一个32x32bit的指令存储器，然后连线

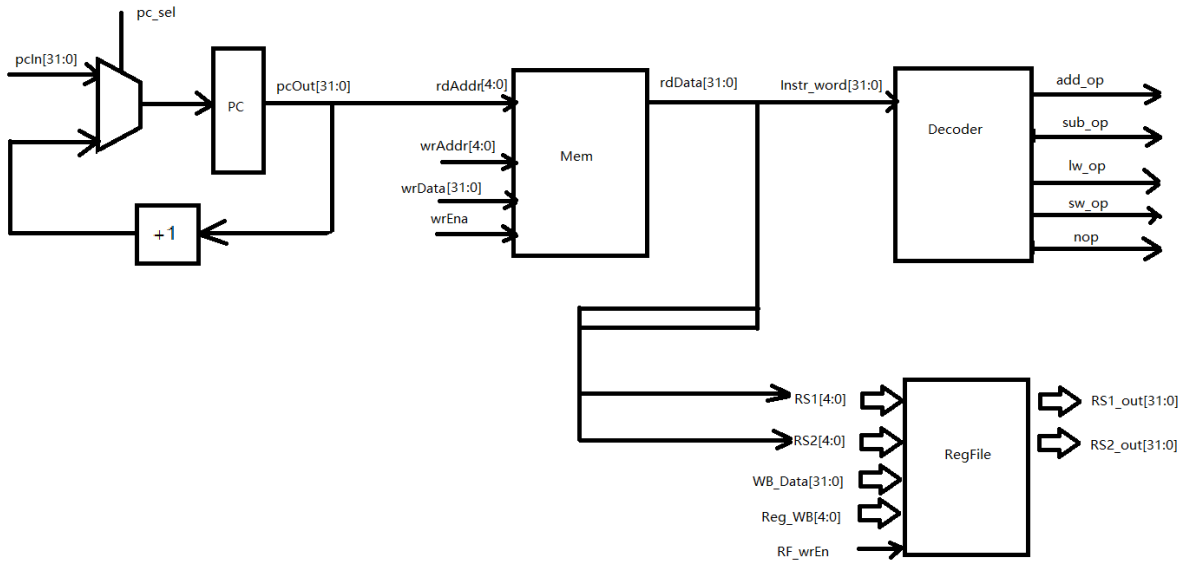
指令存储器设计在 **Mem.scala** 文件中

Mem.scala:

```
package mydesign

import chisel3._

class Mem extends Module {
    val io = IO(new Bundle {
        val rdAddr = Input(UInt(5.W))
        val rdData = Output(UInt(32.W))
        val wrAddr = Input(UInt(5.W))
        val wrData = Input(UInt(32.W))
        val wrEna  = Input(UInt(1.W))
    })

    val mem = SyncReadMem(32, UInt(32.W))

    io.rdData := mem.read(io.rdAddr)

    when(io.wrEna === 1.U) {
        mem.write(io.wrAddr, io.wrData)
    }
}
```

各个模块的连接

几个模块连接如图：



设计思路：

1.设计PC：指令存储器以字编址，pcOut需要+1

2.定义大模块的接口，并实例化译码器、寄存器文件和指令存储器

3.连接PC、译码器、寄存器文件和指令存储器

模块连接在 **Junction.scala** 文件中。

Junction.scala:

```
package mydesign
```

```
import chisel3._

class PC extends Module {
    val io = IO(new Bundle {
        val pcIn  = Input(UInt(32.W))
        val pc_sel = Input(UInt(1.W))
        val pcOut = Output(UInt(32.W))
    })

    val pc = RegInit(0.U(5.W))

    io.pcOut := pc
    pc := Mux(io.pc_sel === 1.U, io.pcIn, io.pcOut+1.U)
}


class Junction extends Module {
    val io = IO(new Bundle {
        val wrAddr  = Input(UInt(5.W))
        val wrData  = Input(UInt(32.W))
        val wrEna   = Input(UInt(1.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))
        val RS1_out = Output(UInt(32.W))
        val RS2_out = Output(UInt(32.W))

        val pcIn = Input(UInt(5.W))
        val pc_sel = Input(UInt(1.W))
    })

    val decoder = Module(new Decoder())
    val regFile = Module(new RegFile())
    val mem     = Module(new Mem())
    val pc      = Module(new PC())
    // pc的连接
    pc.io.pc_sel := io.pc_sel
    pc.io.pcIn := io.pcIn
    // 指令存储器的连接
    mem.io.rdAddr := pc.io.pcOut
    mem.io.wrAddr := io.wrAddr
    mem.io.wrData := io.wrData
    mem.io.wrEna  := io.wrEna
    // decoder的连接
    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
    // 寄存器文件的连接
    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
    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
        }

    io.RS1_out := regFile.io.RS1_out
    io.RS2_out := regFile.io.RS2_out
}
```

连接后模块的测试文件位于**test**目录下，为 **JunctionTest.scala** 文件

JunctionTest.scala：

```
package mydesign

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

class JunctionTest extends AnyFlatSpec with ChiselScalatestTester {
    behavior of "Junction"
    it should "pass" in {
        test(new Junction).withAnnotations(Seq(WriteVcdAnnotation)) {c=>
            // Input ADD
            c.io.wrAddr(0.U)
            c.io.wrData("b000000_00000_00000_00000_00000_000000".U)
            // Input SUB
            c.io.wrAddr(1.U)
            c.io.wrData("b000000_00010_00011_00001_00000_100000".U)
            // Input LW
            c.io.wrAddr(2.U)
            c.io.wrData("b000000_00101_00110_00000_00000_100010".U)
            // Input SW
            c.io.wrAddr(2.U)
            c.io.wrData("b100011_00010_00101_00000_00001_100100".U)

            c.clock.step(1)
            c.io.add_op.expect(1.U)
            c.clock.step(1)
            c.io.sub_op.expect(1.U)
            c.clock.step(1)
            c.io.lw_op.expect(1.U)
            c.clock.step(1)
            c.io.sw_op.expect(1.U)
```

```
            println("\nSUCCESS!!!\n")
        }
    }
}
```

其测试的波形为：