

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：____计算机科学与技术____

指导教师：____刘 刚____

报告人：____林宪亮____学号：____2022150130____班级：____国际班____

实 验 时 间：____2024 年 11 月 1 日 - 2024 年 11 月 3 日____

实验报告提交时间：____2024 年 11 月 3 日____

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

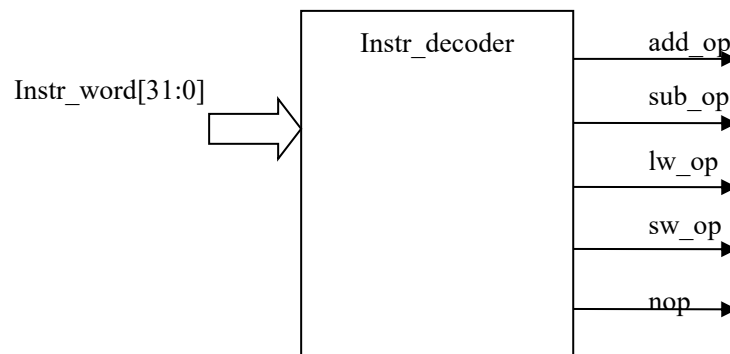
软件：Linux Chisel 开发环境

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。



1.1. I/O 部分

```
class Decoder extends Module {  
  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 nop_op = Output(Bool())  
  })  
}
```

图 1 I/O 代码

在本模块的设计中，我定义了一个输入输出接口，主要用于解码 MIPS 指令。输入端口 Instr_word 接收一个 32 位的无符号整数，表示待解码的指令。通过多个输出端口，我指示

指令的操作类型，包括 add_op、sub_op、lw_op、sw_op 和 nop_op，分别对应加法、减法、加载、存储和无操作指令。每个输出信号为布尔类型，当当前指令匹配相应的操作时，我将该信号置为 true，否则为 false。这一设计确保了指令能够被正确识别，为后续的处理和执行提供了必要的控制信号。

1.2. 定义和取码

```
// 定义操作码
val OPCODE_ADD = "b000000".U
val OPCODE_SUB = "b000000".U
val OPCODE_LW = "b100011".U
val OPCODE_SW = "b101011".U
//定义功能码
val FUNCT_ADD = "b100000".U
val FUNCT_SUB = "b100010".U
// 提取MIPS指令的操作码
val opcode = io.Instr_word(31, 26)
//提取MIPS指令的功能码
val funct = io.Instr_word(5, 0)
```

图 2 定义和取码

我将加法和减法的操作码都设置为 000000，因为它们 R 型指令中共享相同的操作码。我还定义了加载指令（LW）的操作码为 100011，而存储指令（SW）的操作码为 101011。此外，我定义了加法的功能码为 100000，减法的功能码为 100010。通过从输入的指令中提取这两个字段，我能够获取到指令的操作码和功能码，为后续的解码过程奠定基础。

1.3. 译码

```
// 译码
io.add_op := opcode === OPCODE_ADD && funct === FUNCT_ADD
io.sub_op := opcode === OPCODE_SUB && funct === FUNCT_SUB
io.lw_op := opcode === OPCODE_LW
io.sw_op := opcode === OPCODE_SW
io.nop_op := !(io.add_op || io.sub_op || io.lw_op || io.sw_op)
```

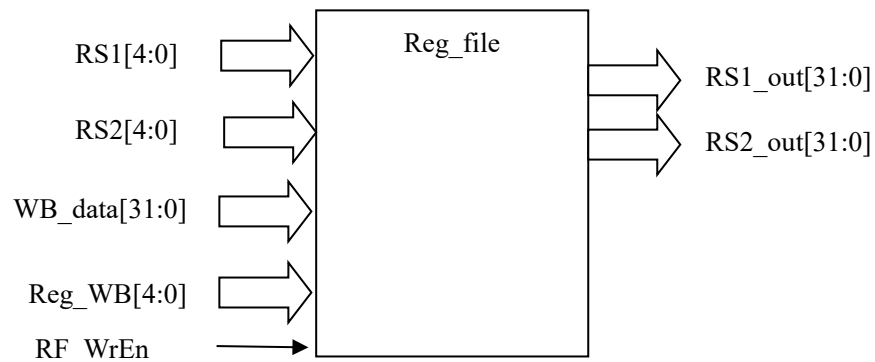
图 3 译码

首先，我检查提取的操作码是否等于加法操作码和功能码。如果都匹配，我将 add_op 信号设为 true，表示当前指令是加法指令。同样的逻辑适用于减法指令，我会检查操作码和功能码是否对应减法。如果匹配，我将 sub_op 设为 true。

接下来，我还检查加载指令和存储指令的操作码，并分别将 lw_op 和 sw_op 信号设为 true，以指示当前指令的类型。最后，我使用逻辑非运算符来确定是否为无操作指令，如果所有已识别的操作均为 false，我将 nop_op 设为 true。通过这些步骤，我能够准确解码输入指令，并生成相应的控制信号。

- 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。



2.1. I/O

```
class RegisterFile extends Module {
  val io = IO(new Bundle {
    val RS1 = Input(UInt(5.W)) // RS1输入信号，用于选择要读取的寄存器
    val RS2 = Input(UInt(5.W)) // RS2输入信号，用于选择要读取的寄存器
    val WB_data = Input(UInt(32.W)) // 写入数据信号，用于写入寄存器
    val Reg_WB = Input(UInt(5.W)) // 选择写入数据的寄存器
    val RS1_out = Output(UInt(32.W)) // RS1输出数据
    val RS2_out = Output(UInt(32.W)) // RS2输出数据
  })
}
```

图 4 IO

在我的寄存器文件模块中，我定义了一个输入输出接口，包含多个信号。首先，RS1 和 RS2 是输入信号，用于选择要读取的寄存器。接着，WB_data 是一个输入信号，表示要写入寄存器的数据，而 Reg_WB 则选择将数据写入哪个寄存器。

此外，我定义了两个输出信号：RS1_out 和 RS2_out，分别用于输出读取到的寄存器数据。在整个模块中，这些信号的设计使得我能够灵活地读取和写入寄存器，从而支持后续的计算和数据处理。

2.2. 写入和输出数据

```
val registers = RegInit(VecInit((0 until 32).map(_ => 0.U(32.W)))) // 32个32位寄存器，初始值等于寄存器编号
registers(io.Reg_WB) := io.WB_data // 写入数据到寄存器
io.RS1_out := Mux(io.RS1 === 0.U, 0.U, registers(io.RS1)) // RS1输出数据，0号寄存器固定读出位0
io.RS2_out := Mux(io.RS2 === 0.U, 0.U, registers(io.RS2)) // RS2输出数据，0号寄存器固定读出位0
}
```

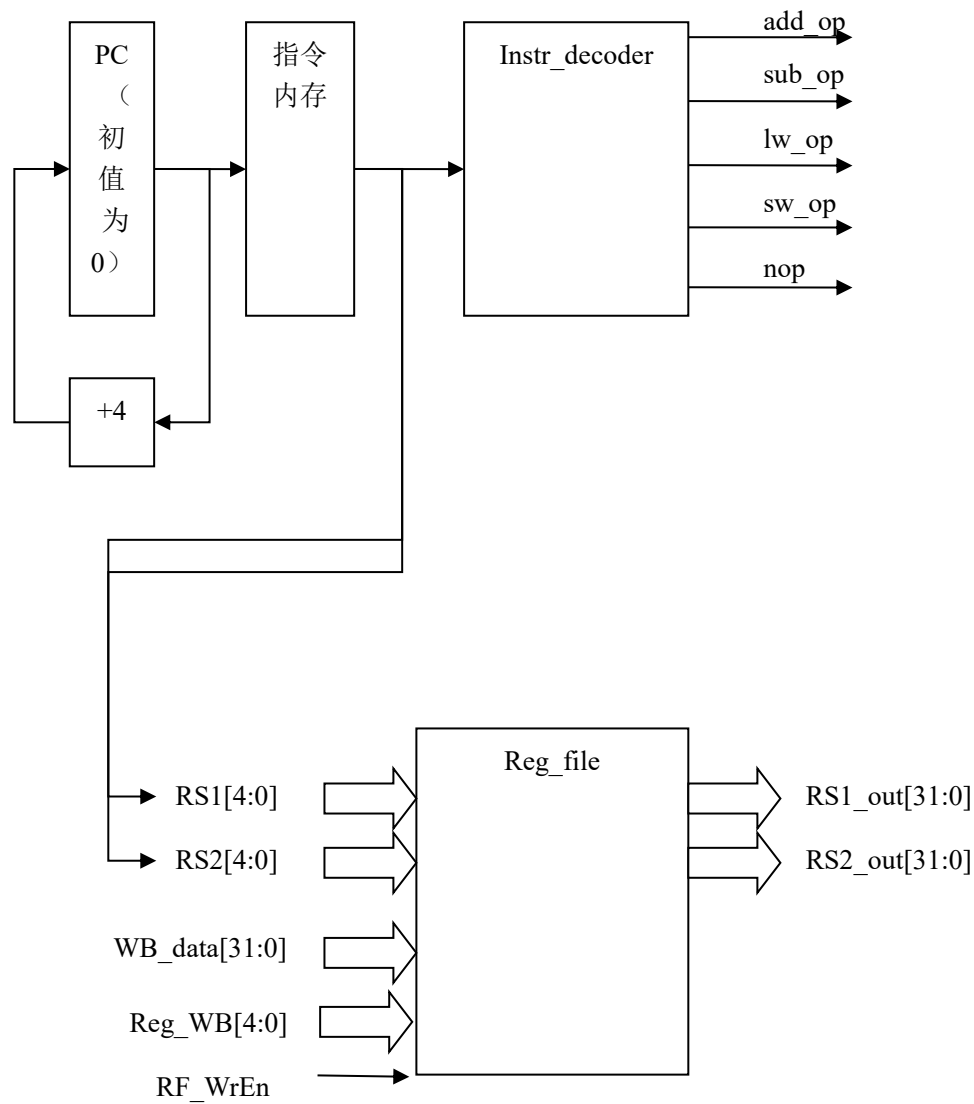
图 5 写入和输出数据

我首先定义了一个寄存器数组，使用 RegInit 和 VecInit 初始化 32 个 32 位的寄存器，初始值设置为各自的寄存器编号。这意味着寄存器 0 到 31 分别存储 0 到 31 的值。

接下来，我将输入信号 WB_data 写入指定的寄存器，寄存器的选择通过 Reg_WB 信号确定。为了输出寄存器的数据，我使用了多路选择器 (Mux) 来判断。如果选择的寄存器 RS1 或 RS2 为 0，我将输出 0，因为寄存器 0 是一个特殊寄存器，固定输出 0；否则，我输出相应寄存器的内容。

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。



3.1. 指令内存

3.1.1. I/O

```
val io = IO(new Bundle {  
    val address = Input(UInt(5.W)) // 32个字, 需要5位地址  
    val instruction = Output(UInt(32.W))  
})
```

图 6 IO

我首先定义了一个寄存器数组，使用 RegInit 和 VecInit 初始化 32 个 32 位的寄存器，初始值设置为各自的寄存器编号。这意味着寄存器 0 到 31 分别存储 0 到 31 的值。接下来，我将输入信号 WB_data 写入指定的寄存器，寄存器的选择通过 Reg_WB 信号确定。为了输出寄存器的数据，我使用了多路选择器 (Mux) 来判断。如果选择的寄存器 RS1 或 RS2 为 0，我将输出 0，因为寄存器 0 是一个特殊寄存器，固定输出 0；否则，我输出相应寄存器的内容。

3.1.2. 四条指令与指令存储器

```
// 创建一个32个字的指令存储器  
val mem = Mem(32, UInt(32.W))  
// 初始化存储器, 存储MIPS指令  
mem.write(0.U, "b0000000_00010_00011_00001_00000_100000".U) // add R1, R2, R3  
mem.write(1.U, "b0000000_00101_00110_00000_00000_100010".U) // sub R0, R5, R6  
mem.write(2.U, "b100011_00010_00101_00000000001100100".U) // lw R5, 100(R2)  
mem.write(3.U, "b101011_00010_00101_00000000001101000".U) // sw R5, 104(R2)  
// 从存储器中读取指令  
io.instruction := mem.read(io.address)
```

图 7 指令与存储器

我创建了一个能够存储 32 条 32 位指令的存储器，使用 Mem(32, UInt(32.W)) 定义。接下来，我通过 write 方法初始化存储器中的指令，依次存入四条 MIPS 指令：

第一条指令是加法操作 add R1, R2, R3，存储在地址 0。

第二条指令是减法操作 sub R0, R5, R6，存储在地址 1。

第三条指令是加载指令 lw R5, 100(R2)，存储在地址 2。

第四条指令是存储指令 sw R5, 104(R2)，存储在地址 3。

最后，我通过 io.address 输入信号读取指定地址的指令，并将其输出到 instruction 信号中。这一设计确保了我可以有效地存储和访问指令，为后续的指令执行提供必要的数据支持。

3.2. 电路

3.2.1. I/O

```
val io = IO(new Bundle {  
    // 寄存器的输入输出  
    val WB_data = Input(UInt(32.W)) // 写入数据信号，用于写入寄存器  
    val Reg_WB = Input(UInt(5.W)) // 选择写入数据的寄存器  
    val RS1_out = Output(UInt(32.W))  
    val RS2_out = Output(UInt(32.W))  
    // 译码  
    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())  
})
```

图 8 IO

我定义了一个输入输出接口，用于与寄存器进行交互。首先，WB_data 是一个 32 位宽的输入信号，表示要写入寄存器的数据。通过 Reg_WB 输入信号，我可以选择具体要写入的寄存器编号。

此外，我定义了两个输出信号，RS1_out 和 RS2_out，它们分别用于输出从寄存器文件中读取的寄存器数据。为了支持指令的译码，我还包含了五个输出信号：add_op、sub_op、lw_op、sw_op 和 nop_op，它们用于指示当前指令的操作类型。这种设计使我能够灵活地管理寄存器的读写，同时提供必要的信息用于指令的解析和执行。

3.2.2. 创建实例

```
val instructionMemory = Module(new InstructionMemory)  
val registerFile = Module(new RegisterFile)  
val decoder = Module(new Decoder)  
val pc = RegInit(0.U(5.W))
```

图 9 创建实例

在我的电路模块中，我实例化了三个子模块，以构建一个基本的处理器架构。首先，我创建了一个指令存储器模块 instructionMemory，用于存储和读取指令。接着，我实例化了一个寄存器文件模块 registerFile，它负责管理寄存器的读写操作。

此外，我还创建了一个译码器模块 decoder，用于解析从指令存储器中读取到的指令并识别其操作类型。最后，我定义了一个程序计数器 pc，使用 RegInit(0.U(5.W)) 将其初始化为 0，这样可以指向当前执行的指令地址。这种设计将指令存储、寄存器管理和指令译码结合在一起，为后续的指令执行奠定了基础。

3.2.3. 取指令

```
// 根据pc的值取出指令寄存器相应指令  
instructionMemory.io.address := pc  
decoder.io.Instr_word := instructionMemory.io.instruction  
registerFile.io.RS1 := instructionMemory.io.instruction(25, 21)  
registerFile.io.RS2 := instructionMemory.io.instruction(20, 16)  
registerFile.io.WB_data := (0.U(32.W))  
registerFile.io.Reg_WB := (0.U(5.W))
```

图 10 取指令

我根据程序计数器 pc 的值从指令存储器中获取当前指令。通过将 pc 的值赋给 instructionMemory.io.address，我能够读取指令存储器中对应地址的指令。

接下来，我将从指令存储器中读取到的指令传递给译码器，通过 decoder.io.Instr_word 来进行解析。同时，我提取指令中的源寄存器字段，将它们分别赋值给寄存器文件的输入信号 RS1 和 RS2，以便从寄存器中读取相关数据。

为了初始化寄存器写入操作，我将 WB_data 和 Reg_WB 分别设为 0，这样可以在没有有效写入数据时避免错误。

3.2.4. 更新输出

```
// 更新输出
io.RS1_out := registerFile.io.RS1_out
io.RS2_out := registerFile.io.RS2_out
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
// 更新PC
pc := pc + 1.U
```

图 11 更新输出

我首先更新输出信号，以反映寄存器文件和译码器的当前状态。我将寄存器文件中读取到的 RS1_out 和 RS2_out 信号分别赋值给输出接口的 io.RS1_out 和 io.RS2_out，这样外部模块可以获取到这些寄存器的数据。同时，我将译码器识别到的操作类型信号，如 add_op、sub_op、lw_op、sw_op 和 nop_op，分别更新到输出接口中，提供指令的具体操作信息。最后，我更新程序计数器 pc，通过将其值加 1，指向下一条指令。这一过程确保了在每个时钟周期中，模块能够正确地获取、解析指令，并为后续的指令执行做好准备。

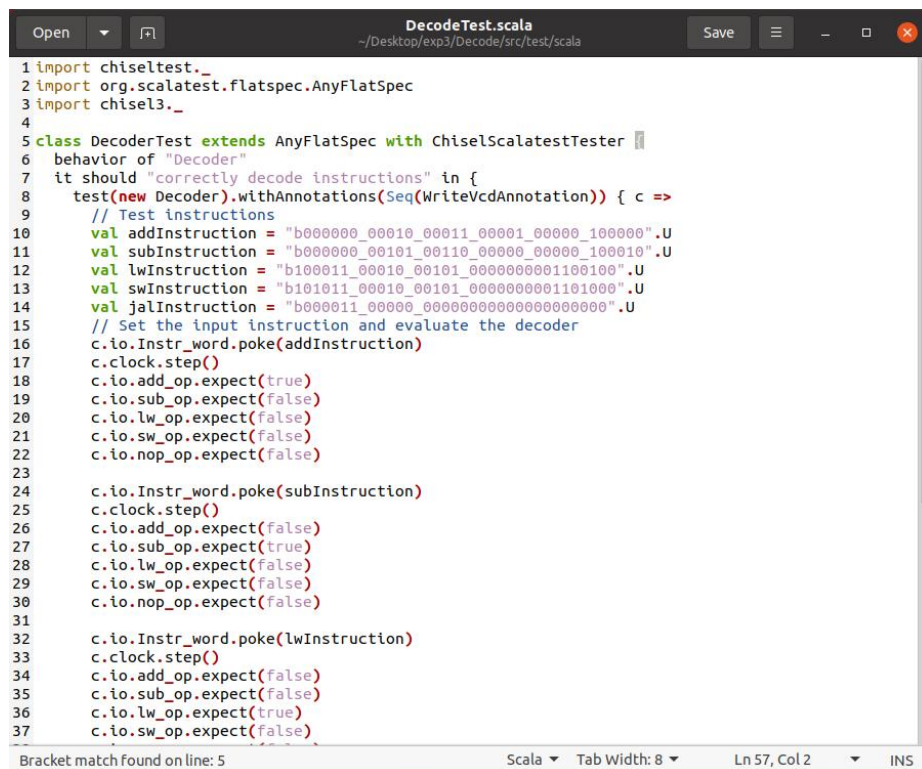
五、实验结果

1. 译码电路测试

根据 myLED 的工程创建方法进行译码器的工程创建，文件夹如下图 12 所示。

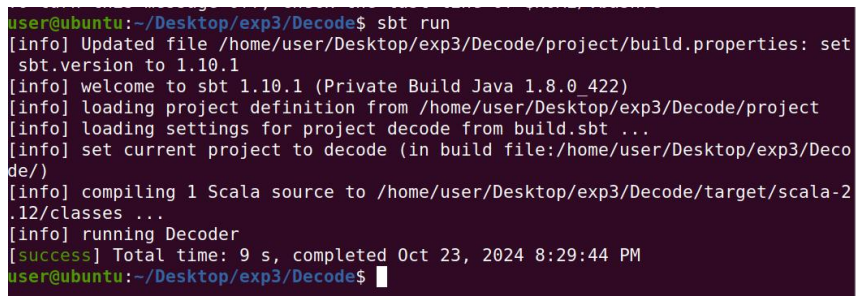
```
user@ubuntu:~/Desktop/exp3$ tree 1
1
├── build.sbt
├── src
│   ├── main
│   │   └── scala
│   │       └── Decode.scala
│   └── test
│       ├── scala
│       └── DecodeTest.scala
5 directories, 3 files
user@ubuntu:~/Desktop/exp3$
```

图 12 创建工程



```
1 import chiseltest._
2 import org.scalatest.flatspec.AnyFlatSpec
3 import chisel3._
4
5 class DecoderTest extends AnyFlatSpec with ChiselScalatestTester {
6   behavior of "Decoder"
7   it should "correctly decode instructions" in {
8     test(new Decoder).withAnnotations(Seq(WriteVcdAnnotation)) { c =>
9       // Test instructions
10      val addInstruction = "b000000_00010_00011_00001_00000_100000".U
11      val subInstruction = "b000000_00101_00110_00000_00000_100010".U
12      val lwInstruction = "b100011_00010_00101_0000000001100100".U
13      val swInstruction = "b101011_00010_00101_0000000001101000".U
14      val jalInstruction = "b000011_00000_00000000000000000000".U
15      // Set the input instruction and evaluate the decoder
16      c.io.Instr_word.poke(addInstruction)
17      c.clock.step()
18      c.io.add_op.expect(true)
19      c.io.sub_op.expect(false)
20      c.io.lw_op.expect(false)
21      c.io.sw_op.expect(false)
22      c.io.nop_op.expect(false)
23
24      c.io.Instr_word.poke(subInstruction)
25      c.clock.step()
26      c.io.add_op.expect(false)
27      c.io.sub_op.expect(true)
28      c.io.lw_op.expect(false)
29      c.io.sw_op.expect(false)
30      c.io.nop_op.expect(false)
31
32      c.io.Instr_word.poke(lwInstruction)
33      c.clock.step()
34      c.io.add_op.expect(false)
35      c.io.sub_op.expect(false)
36      c.io.lw_op.expect(true)
37      c.io.sw_op.expect(false)
```

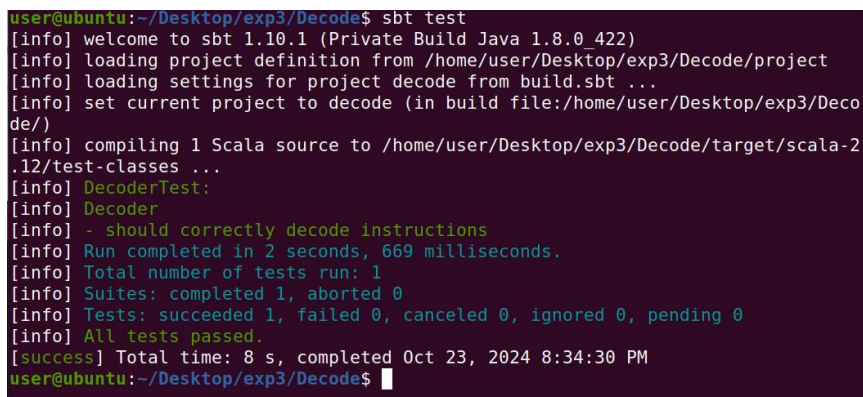
图 13 decode 测试代码



```
user@ubuntu:~/Desktop/exp3/Decode$ sbt run
[info] Updated file /home/user/Desktop/exp3/Decode/project/build.properties: set
sbt.version to 1.10.1
[info] welcome to sbt 1.10.1 (Private Build Java 1.8.0_422)
[info] loading project definition from /home/user/Desktop/exp3/Decode/project
[info] loading settings for project decode from build.sbt ...
[info] set current project to decode (in build file:/home/user/Desktop/exp3/Decode/)
[info] compiling 1 Scala source to /home/user/Desktop/exp3/Decode/target/scala-2.12/classes ...
[info] running Decoder
[success] Total time: 9 s, completed Oct 23, 2024 8:29:44 PM
user@ubuntu:~/Desktop/exp3/Decode$
```

图 14 运行 sbt run 指令

在编写完测试代码后，我接着执行了 sbt run 和 sbt test 指令，如图 13 和图 14 所示。



```
user@ubuntu:~/Desktop/exp3/Decode$ sbt test
[info] welcome to sbt 1.10.1 (Private Build Java 1.8.0_422)
[info] loading project definition from /home/user/Desktop/exp3/Decode/project
[info] loading settings for project decode from build.sbt ...
[info] set current project to decode (in build file:/home/user/Desktop/exp3/Decode/)
[info] compiling 1 Scala source to /home/user/Desktop/exp3/Decode/target/scala-2.12/test-classes ...
[info] DecoderTest:
[info] Decoder
[info] - should correctly decode instructions
[info] Run completed in 2 seconds, 669 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: 8 s, completed Oct 23, 2024 8:34:30 PM
user@ubuntu:~/Desktop/exp3/Decode$
```

图 15 运行 sbt test 指令

此时，在原文件夹中生成了一个名为 test_run_dir 的新文件夹。进入该文件夹两次后，我发现了 Decoder.vcd 文件。我使用 GTKWave 打开该文件，以观察波形，如图 15 所示。

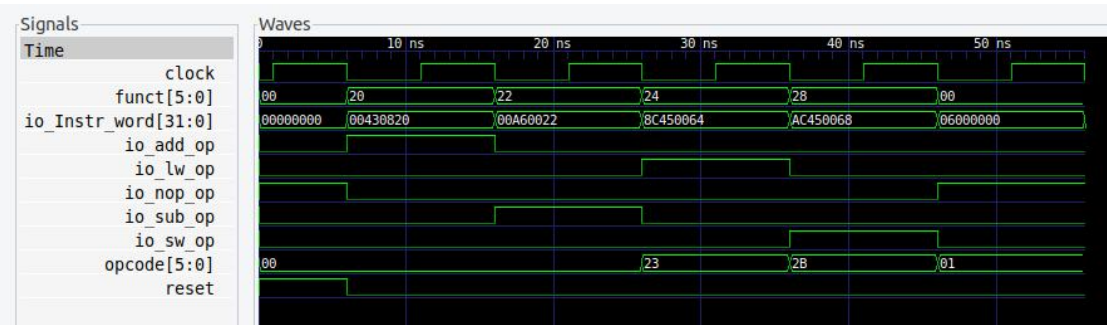


图 16 Decode 输出波形

我观察到输入的 Instr_word 包含指令 add R1, R2, R3、sub R0, R5, R6、lw R5, 100(R2)、sw R5, 104(R2)和 JAL 100。波形显示，在指令解析阶段，Instr_word 的值随着每条指令的更新而变化，译码器相应的操作信号如 add_op 和 sub_op 在执行对应指令时变为 1，表明指令正在被识别并运行。对于 lw 和 sw 指令，相关的寄存器读写信号显示了从寄存器中读取的数据，确保了数据的正确传递。此外，程序计数器 pc 在每个时钟周期内递增，反映了指令的顺序执行。最后，在处理 JAL 100 指令时，波形展示了跳转信号的变化，表明程序流程的转移。

后面寄存器和存储器波形前步骤同理。

2. 寄存器文件测试

```
user@ubuntu:~/Desktop/exp3/Register$ tree
.
├── build.sbt
├── src
│   ├── main
│   │   ├── Register.scala
│   │   └── RegisterTest.scala
│   └── test
│       └── RegisterTest.scala
5 directories, 3 files
```

图 17 创建工程

```
Open RegisterTest.scala Save
~/Desktop/exp3/Register/src/test/scala

1 import chisel3._
2 import chiseltest._
3 import org.scalatest.flatspec.AnyFlatSpec
4 import chisel3.util._
5
6 class RegisterFileTest extends AnyFlatSpec with ChiselScalatestTester {
7   behavior of "RegisterFile"
8   it should "correctly update and read registers" in {
9     test(new RegisterFile).withAnnotations(Seq(WriteVcdAnnotation)) { c =>
10       // 设置输入信号
11       c.io.RS1.poke(5.U)
12       c.io.RS2.poke(8.U)
13       c.io.WB_data.poke(0x1234.U)
14       c.io.Reg_WB.poke(1.U)
15
16       c.clock.step()
17       c.io.RS1_out.expect(5.U)
18       c.io.RS2_out.expect(8.U)
19     }
20   }
21 }
22
```

图 18 Register 测试代码

测试代码如图 7 所示，输出波形如图 8 所示。通过观察波形，我可以看到 RS1 的值为 5，RS2 的值为 8，而 WB_data 的值为 0x1234。在随后的时间周期中，0x1234 被写入了寄存器。

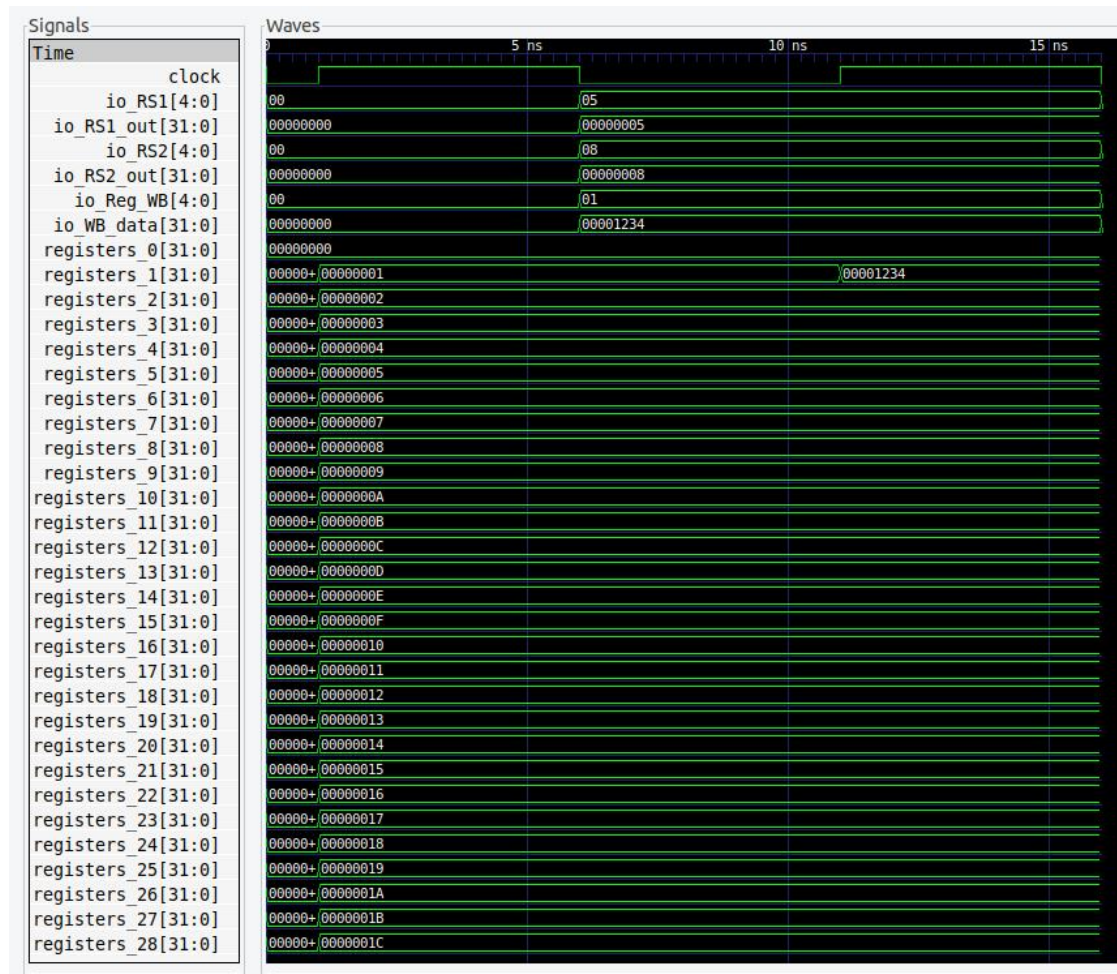


图 19 Register 输出波形

在输出波形分析中，我观察到 RS1 和 RS2 的值分别为 5 和 8，显示了寄存器文件正确地对应的寄存器中读取了数据。当 RF_WrEn 信号为 1 时，WB_data 的值 0x1234 成功写入寄存器 1，这在波形中表现为寄存器 1 的输出在写入周期内从其初始值变为 0x1234。波形图还显示了在写入发生时，寄存器 1 的输出在时钟周期内有明显的上升沿，而 RS1 和 RS2 的输出则保持稳定，确认了寄存器文件的读写操作是异步的。这些观察结果表明寄存器文件的设计能够有效处理两路读出和一路写入的操作，并确保了正确的数据流动。

3. 字指令存储器

由于代码中需要运用到前面实验的 Decoder 和 Register，故需要我们将前面实验中各自 main 文件夹的 scala 文件复制到现 main 文件夹，工程如图 20 所示，测试代码如图 21 所示。

```
user@ubuntu:~/Desktop/exp3/Circuit$ tree .
.
├── build.sbt
├── src
│   ├── main
│   │   └── scala
│   │       ├── Circuit.scala
│   │       ├── Decode.scala
│   │       ├── InstructionMemory.scala
│   │       └── Register.scala
│   └── test
│       └── scala
│           └── CircuitTest.scala
5 directories, 6 files
```

图 20 创建工程

```
CircuitTest.scala
~/Desktop/exp3/Circuit/src/test/scala

1 import chiseltest._
2 import org.scalatest.flatspec.AnyFlatSpec
3 import chisel3._
4
5 class CircuitTest extends AnyFlatSpec with ChiselScalatestTester {
6   behavior of "Circuit"
7   it should "correct circuit" in {
8     test(new Circuit).withAnnotations(Seq(WriteVcdAnnotation)) { c =>
9       c.clock.step()
10      c.clock.step()
11      c.clock.step()
12      c.clock.step()
13    }
14  }
15 }
```

图 21 Circuit 测试代码

接着，按照步骤操作后，我得到了输出波形，如图 22 所示。首先，从 Instr_word 中可以看到题目中的四条指令对应的十六进制值。图 23 展示了 add、sub、lw 和 sw 四条指令的相关波形。在不同指令的时间段内，对应指令的波形值变为 1，这表明指令正在运行。

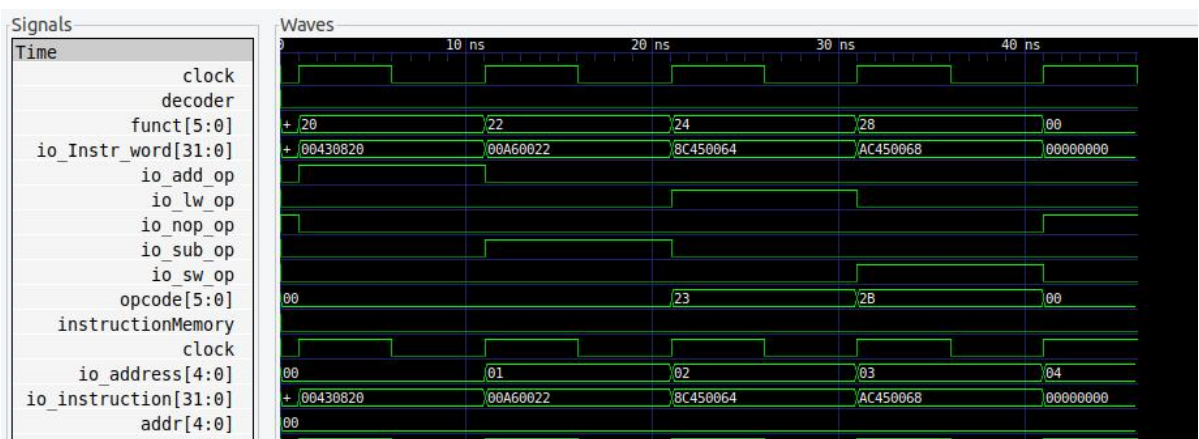


图 22 Circuit 测试波形



图 23 指令相关波形

在观察四条指令的执行过程波形时，可以明显看到每条指令在时钟周期中的激活情况。首先，当 PC 值为 0 时，指令存储器读取到的第一条指令为 add R1, R2, R3，此时 add_op 信号被激活，表明当前正在执行加法操作。接下来，随着 PC 自增到 1，读取到的第二条指令为 sub R0, R5, R6，此时 sub_op 信号变为高，指示当前为减法操作。继续自增到 2 时，读取到的第三条指令为 lw R5, 100(R2)，激活了 lw_op 信号，表示开始执行加载操作。最后，在 PC 为 3 时，第四条指令 sw R5, 104(R2) 被读取，sw_op 信号被激活，显示出正在进行存储操作。

五、实验总结与体会

在本次实验中，我设计并实现了一个连续取指令并进行指令译码的电路，达成了预设的实验目标。实验分为三周进行，每一周都有不同的重点和任务，这让我逐步掌握了设计简单数据通路的基本方法。

第一周：译码器设计

在第一周，我完成了译码器的设计。通过使用 Chisel 语言，我实现了一个功能完整的译码器，可以解析 MIPS 指令的操作码和功能码。这一过程让我深入理解了指令的结构，以及如何将输入信号转换为不同的控制信号。译码器的设计为后续的寄存器文件和指令存储器的实现奠定了基础。

第二周：寄存器文件实现

第二周的任务是实现寄存器文件。我设计了一个包含 32 个 32 位寄存器的模块，支持两读一写的操作，并确保 0 号寄存器总是输出 0。通过测试不同的输入信号，我验证了寄存器文件的功能，确认其能正确读写寄存器的数据。这一部分的实现让我对寄存器的工作原理有了更深入的理解，同时也锻炼了我的调试能力。

第三周：数据通路原型组合

在最后一周，我将之前实现的译码器和寄存器文件与指令存储器及地址部件结合，形成了一个完整的数据通路原型。通过调整各个模块之间的连接和信号传递，我成功实现了指令的连续读取和译码。观察波形和仿真结果，让我清晰地看到了指令流的动态变化，加深了对数据通路结构的理解。

通过这三周的实验，我不仅学会了如何使用 Chisel 进行硬件设计，还掌握了数据通路的基本构建块，包括译码器、寄存器文件和指令存储器的设计与实现。每一部分的实验都让我体会到理论知识与实际应用的结合，提升了我的工程思维能力和动手实践能力。我期待在今后的学习中，能够进一步探索更复杂的电路设计与实现。

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。