**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： 数学科学学院**

**专 业： 信息与计算科学（数学与计算机实验班）**

**指 导 教 师： 罗秋明**

**报告人： 王曦 学号： 2021192010 班级： 数计**

**实 验 时 间： 2023年11月03日**

**实验报告提交时间： 2023年11月03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。

Instr\_decoder

Instr\_word[31:0]

add\_op

sub\_op

lw\_op

sw\_op

nop

* 1. **功能**

输入指令，取出指令的 opcode 和 func ，识别指令为 add 、sub 、lw 、sw 中的何种指令，并设置相应的输出信号，不属于上述四种的指令都视为 nop 。

1.2 **I/O**

输入：32 位的指令。

输出：输出信号 add\_op 、sub\_op 、lw\_op 、sw\_op 、nop 。

1.3 **实现**

1.3.1 I/O

|  |
| --- |
| val io = IO(new Bundle {  val Instr\_word = Input(UInt(32.W)) // 输入信号  // 输出信号  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())  }) |

1.3.2 取出 opcode 和 func 。

|  |
| --- |
| // opcode  val AandS = "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)  // 取出指令的 opcode 和 func  val opcode = io.Instr\_word(31, 26)  val func = io.Instr\_word(5,0) |

1.3.3 译码

|  |
| --- |
| // 指令默认是 nop  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  // 译码  when (opcode === AandS) {  when (func === ADD\_OPCODE) {  io.add\_op := true.B  io.nop\_op := false.B  }.elsewhen (func === SUB\_OPCODE) {  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  } |

1. 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。

Reg\_file

RS1[4:0]

RS2[4:0]

WB\_data[31:0]

Reg\_WB[4:0]

RS1\_out[31:0]

RS2\_out[31:0]

RF\_WrEn

* 1. **功能**

支持两读一写的寄存器文件。

* 1. **I/O**

两读一写。

2.3 **实现**

2.3.1 I/O

|  |
| --- |
| 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))  }) |

2.3.2 寄存器堆

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

2.3.3 设置输出

|  |
| --- |
| 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)) |

2.3.4 写入

|  |
| --- |
| // 根据写信号选择要写入的寄存器  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)  } |

1. 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。

PC

（初值为0）

+4

指令内存

Instr\_decoder

Reg\_file

RS1[4:0]

RS2[4:0]

WB\_data[31:0]

RS1\_out[31:0]

RS2\_out[31:0]

add\_op

sub\_op

lw\_op

sw\_op

nop

Reg\_WB[4:0]

RF\_WrEn

* 1. **指令内存**

3.1.1 I/O

|  |
| --- |
| val io = IO(new Bundle {  // 读  val rdEna = Input(Bool())  val rdData = Output(UInt(32.W))  // 写  val wrEna = Input(Bool())  val wrAddr = Input(UInt(10.W))  val wrData = Input(UInt(32.W))  }) |

3.1.2 PC

|  |
| --- |
| val pcReg = RegInit(0.U(32.W)) |

3.1.3 指令内存

|  |
| --- |
| val mem = SyncReadMem(128,UInt(8.W)) |

3.1.4 写内存

|  |
| --- |
| // 写内存  when (io.wrEna) {  // 分 4 个单元写  mem.write(io.wrAddr, io.wrData(7, 0))  mem.write(io.wrAddr + 1.U, io.wrData(15, 8))  mem.write(io.wrAddr + 2.U, io.wrData(23, 16))  mem.write(io.wrAddr + 3.U, io.wrData(31, 24))  } |

3.1.5 读内存

|  |
| --- |
| // 读内存  when (io.rdEna) {  // 分 4 个单元读  val rdData0 = mem.read(pcReg)  val rdData1 = mem.read(pcReg+1.U)  val rdData2 = mem.read(pcReg+2.U)  val rdData3 = mem.read(pcReg+3.U)  io.rdData := rdData3 ## rdData2 ## rdData1 ## rdData0 // ## 表连接  pcReg := pcReg + 4.U  }.otherwise{  io.rdData := 0.U  } |

3.2 **综合电路**

3.2.1 I/O

|  |
| --- |
| val io = IO(new Bundle {  // Decoder  val Instr\_word = Output(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())  // RegisterFile  val RS1\_out = Output(UInt(32.W))  val RS2\_out = Output(UInt(32.W))  // Instruction  val rdEna = Input(Bool())  val wrAddr = Input(UInt(10.W))  val wrData = Input(UInt(32.W))  val wrEna = Input(Bool())  }) |

3.2.2 实例化各模块

|  |
| --- |
| // 实例化  val decoder = Module(new Decoder)  val registerFile = Module(new RegisterFile)  val instructionMemory = Module(new Instruction) |

3.2.3 连接指令信号

|  |
| --- |
| // 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 |

3.2.4 连接控制信号

|  |
| --- |
| // 连接控制信号  instructionMemory.io.rdEna := io.rdEna  instructionMemory.io.wrEna := io.wrEna  instructionMemory.io.wrAddr := io.wrAddr  instructionMemory.io.wrData := io.wrData |

3.2.5 连接输入数据

|  |
| --- |
| // 连接输入数据  registerFile.io.RS1 := instructionMemory.io.rdData(25,21)  registerFile.io.RS2 := instructionMemory.io.rdData(20, 16)  decoder.io.Instr\_word := instructionMemory.io.rdData  io.Instr\_word := instructionMemory.io.rdData |

3.2.6 初始化寄存器文件的控制信号

|  |
| --- |
| // 初始化寄存器文件的控制信号  registerFile.io.Reg\_WB := false.B  registerFile.io.WB\_data := 0.U |

3.2.7 连接模块输出信号

|  |
| --- |
| // 连接模块输出信号  io.RS1\_out := registerFile.io.RS1\_out  io.RS2\_out := registerFile.io.RS2\_out |

**五、实验结果**

1. **寄存器文件测试**

1.1 模仿实例项目 myLED ，搭建寄存器文件测试的项目。

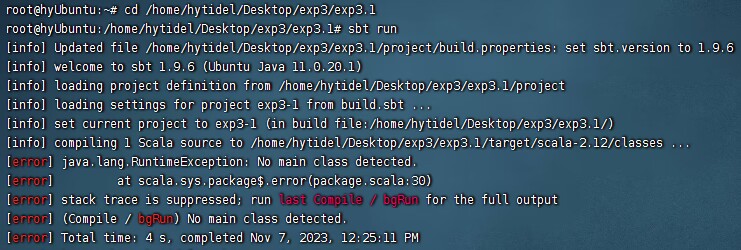


其中测试代码如下：

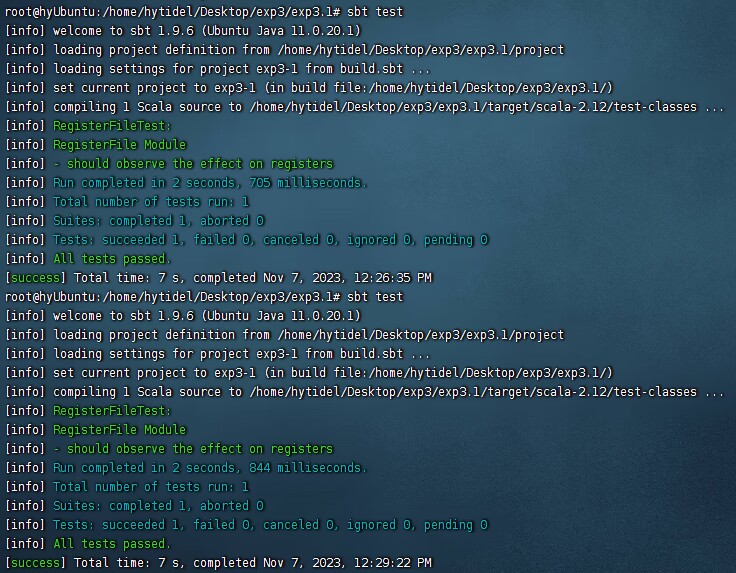
|  |
| --- |
| class RegisterFileTest extends AnyFlatSpec with ChiselScalatestTester {  behavior of "RegisterFile Module"  it should "observe the effect on registers" in {  test (new RegisterFile).withAnnotations(Seq(WriteVcdAnnotation)) { c =>  // 设置输入信号  c.io.RS1.poke(5.U) // RS1 = 5  c.io.RS2.poke(8.U) // RS2 = 8  c.io.WB\_data.poke(0x1234.U) // WB\_data = 0x1234  // 时钟向前一步, 写入  c.io.Reg\_WB.poke(true) // 写使能  c.clock.step(1)  }  }  } |

1.2 在 exp3.1 目录下，执行 sbt run ，搭建项目。

虽下图显示编译出错，但无关紧要，依旧能得到 project 、target 、test\_run\_dir 三个文件夹。



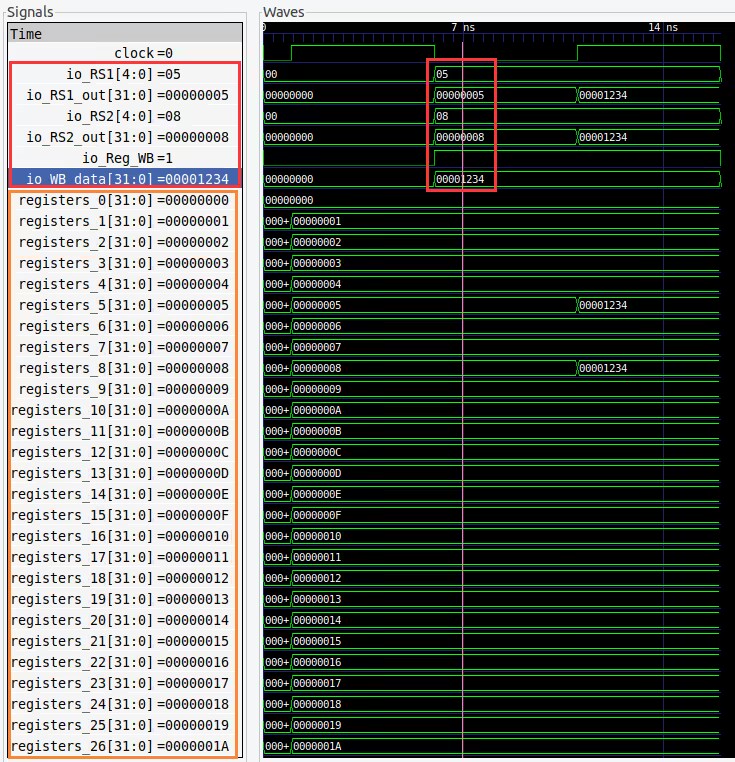
1.3 在 exp3.1 目录下，执行 sbt test ，生成波形文件。



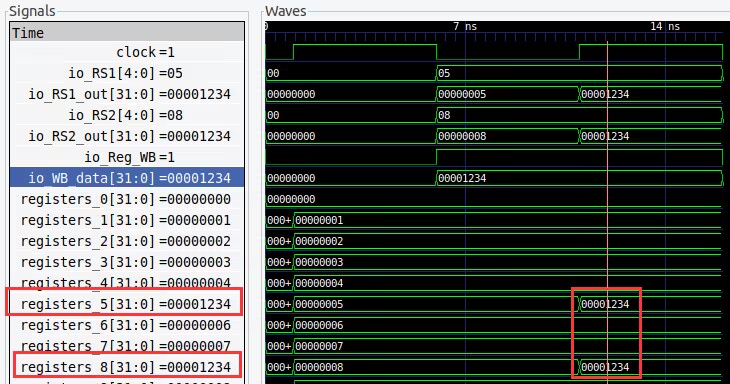
1.4 用 GTKWave 打开波形文件。

观察到有编号 0 ~ 31 的 32 个寄存器，且寄存器的初值为其编号（十六进制，下同），如橙框所示。

观察到 RS1 = 5，RS2 = 8，WB\_data = 0x1234 ，如红框所示。

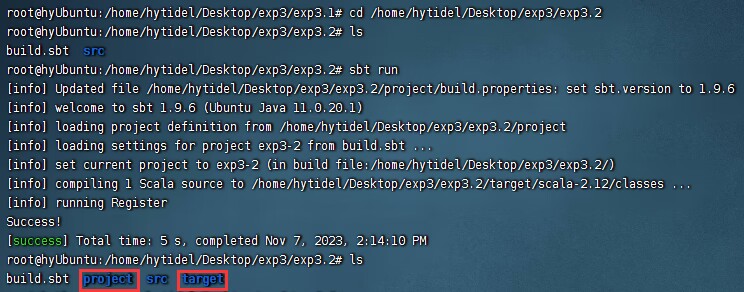


1.5 观察到在后面的时钟周期中，将 0x1234 写入 R5 和 R8 ，如红框所示。



2. **译码器、指令内存、数据通路测试**

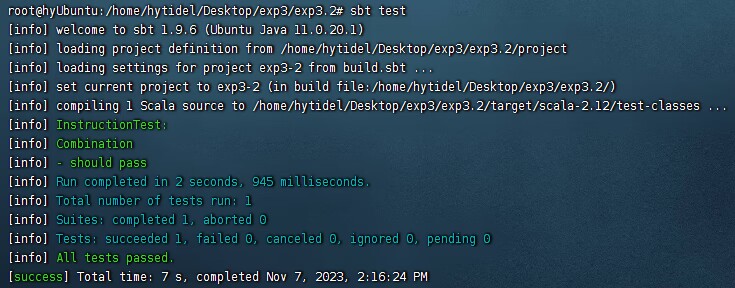
2.1 项目搭建



其中测试代码如下：

|  |
| --- |
| class InstructionTest extends AnyFlatSpec with ChiselScalatestTester {  behavior of "Combination"  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)  }  }  } |

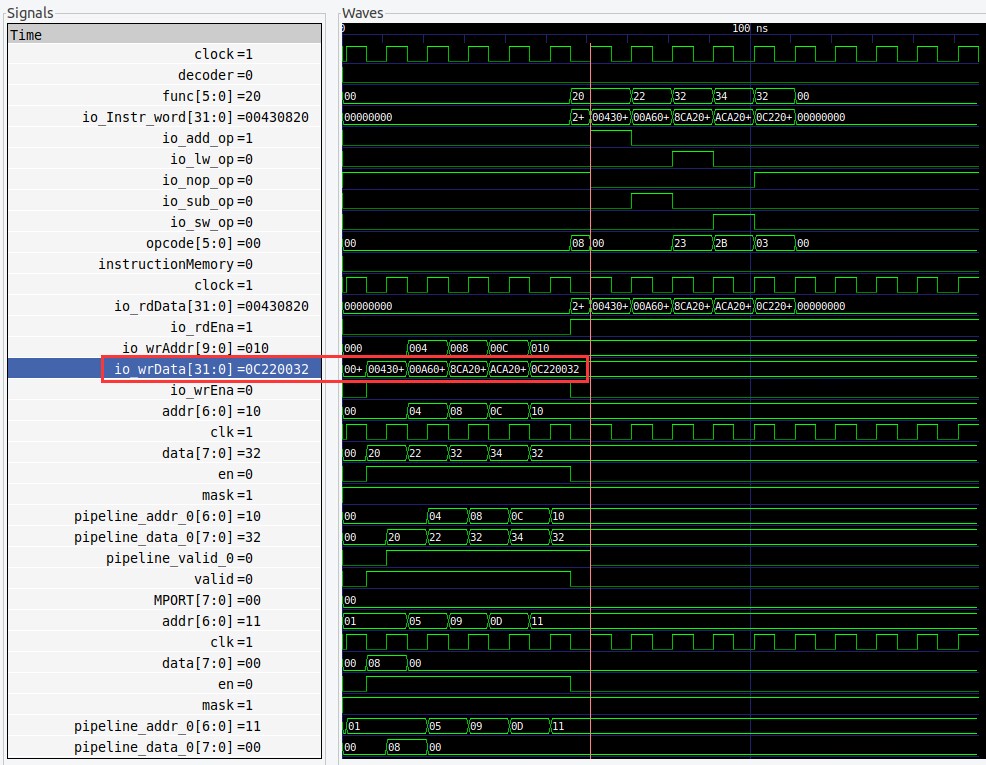
2.2 生成波形文件。



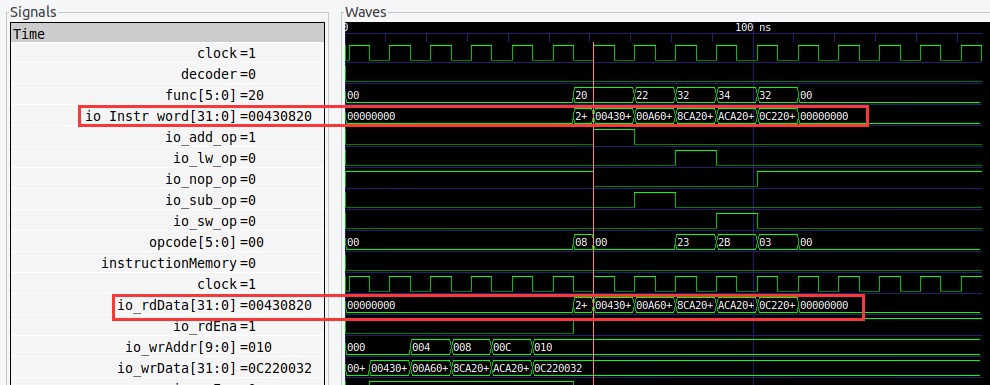
2.3 观察波形图

在前几个时钟周期中，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 |



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。

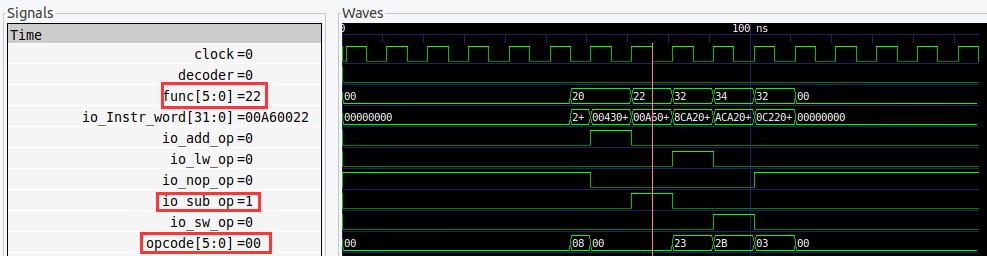


2.4 观察各指令的 opcode 、func 、op 信号

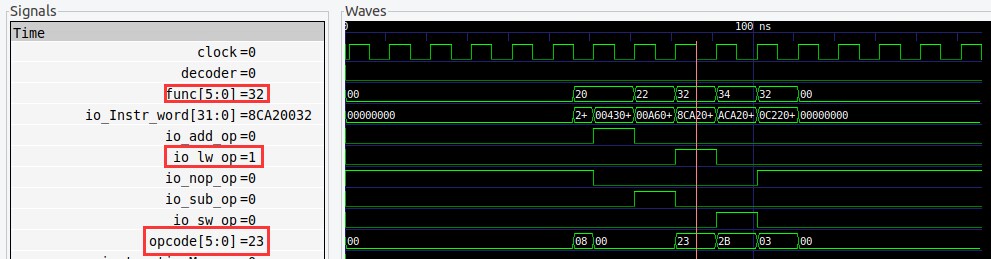
2.4.1 add 指令



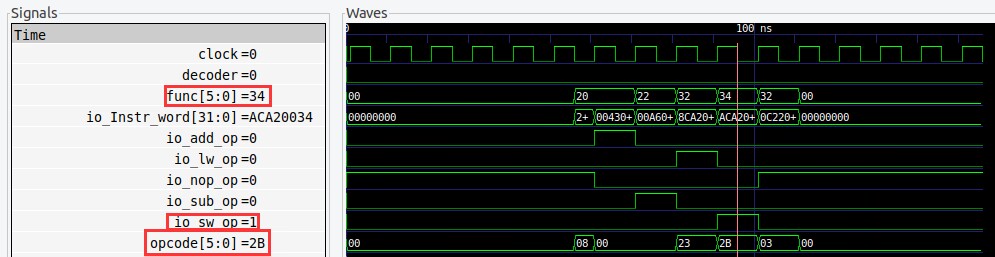
2.4.2 sub 指令



2.4.3 lw 指令

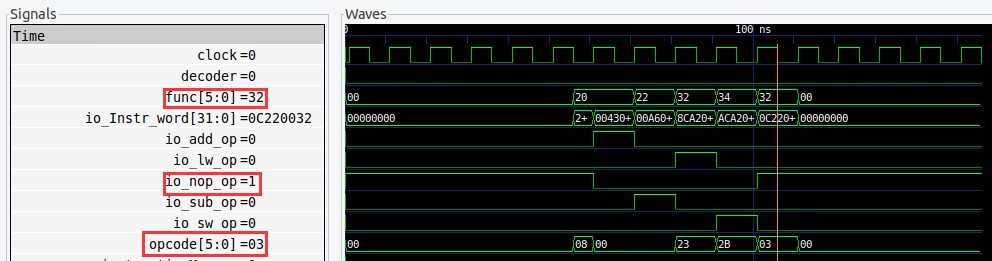


2.4.4 sw 指令



2.4.5 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。



**六、实验总结与体会**

在这个Chisel实验中，我们的主要目标是设计一个能够连续取指令并进行指令译码的电路。通过这个实验，我们获得了关于设计简单数据通路的基本方法，深入理解了数字电路的工作原理以及如何使用Chisel进行硬件描述和模拟。

设计译码器： 实验的第一步是设计一个译码器。在这个过程中，我们学会了如何使用Chisel编写硬件描述，定义输入和输出端口以及设计逻辑门电路，以实现我们的译码器。这个步骤是实验的基础，因为译码器是整个数据通路中的重要组成部分。

实现寄存器文件： 接下来，我们实现了一个寄存器文件。这是一个用于存储寄存器值的模块，它允许我们在指令执行过程中读取和写入寄存器。这一步教会了我们如何定义多个寄存器，以及如何在Chisel中管理它们的状态。

添加指令存储器和地址部件： 在完成译码器和寄存器文件后，我们将这些部件组合起来，以构建一个完整的数据通路原型。这涉及到设计指令存储器和地址部件，以便能够获取指令并确定下一条指令的地址。这一步骤展示了如何将不同的硬件模块连接在一起，以构建一个功能完整的系统。

在这个实验中，我学到了很多关于数字电路设计的重要概念和Chisel语言的基础知识。这些包括硬件描述的基本语法、组合逻辑和时序逻辑的设计原则、多模块系统的连接方法等。

此外，通过完成这个实验，我还锻炼了问题解决和调试的能力。在设计和实现硬件时，出现错误是很常见的，但通过分析问题并逐步修复它们，我学到了如何识别和解决不同类型的错误。

总的来说，这个Chisel实验是一个非常有价值的学习经验，它不仅帮助我掌握了数字电路设计的基本技能，还使我更加熟悉了Chisel这个强大的硬件描述语言。我期待在未来的实验中继续深入研究和应用这些知识。

|  |
| --- |
| **指导教师批阅意见：**  **成绩评定：**  指导教师签字：  年 月 日 |
| 备注： |