**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： 吴嘉楷 学号： 2022150168 班级： 国际班**

**实 验 时 间： 2024年10月24日**

**实验报告提交时间： 2024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。

Instr\_decoder

Instr\_word[31:0]

add\_op

sub\_op

lw\_op

sw\_op

nop

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. 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

**五、实验结果**

1. **译码电路**

译码器如图所示。

Instr\_decoder

Instr\_word[31:0]

add\_op

sub\_op

lw\_op

sw\_op

nop

图 1 译码器示意图

1. **译码器设计**

首先，我们需要设计四种指令的比特模式：加法指令（add）、减法指令（sub）、加载字指令（lw，即load word）、以及存储字指令（sw，即store word）。同时，需要明确的是，译码器模块针对这些指令的输出均为1位二进制数据（1bit）：

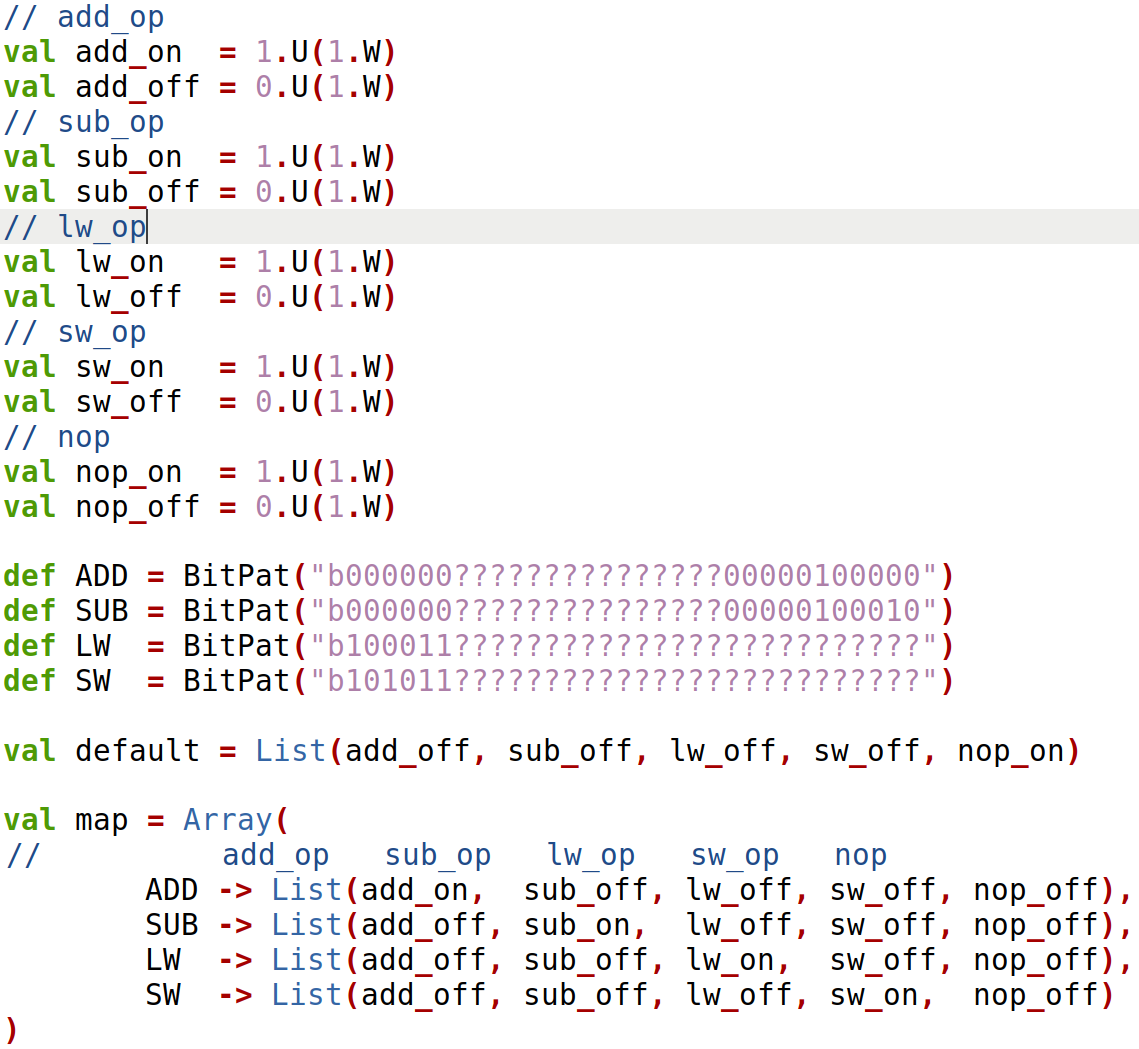


图2 设计指令比特模式的核心代码

根据输入的32位机器指令字（Instr\_word）中包含的操作码和功能码，从而去判断指令的具体类型，并输出相应的布尔信号以标识该指令是否为加法指令（add）、减法指令（sub）、加载字指令（lw）、存储字指令（sw）或空操作指令（nop）。例如，当判断出的指令类型为加法指令（add）时，系统将输出布尔信号add\_op，并将其置为1。核心的代码实现如图3中所示：

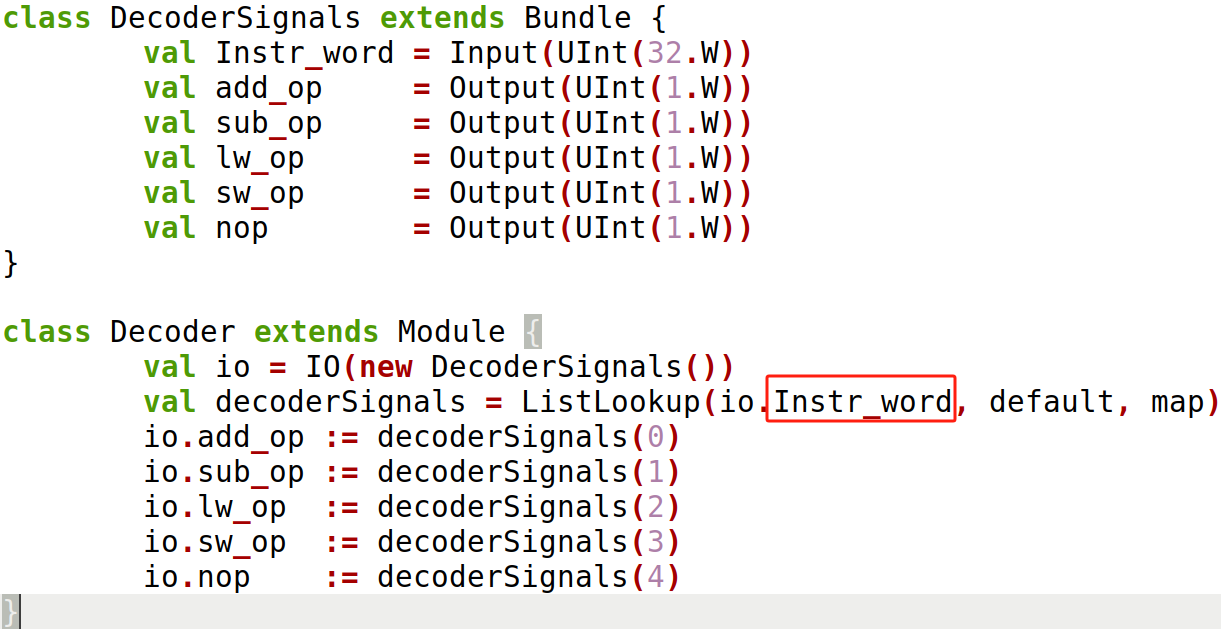


图 3 译码器核心代码

1. **译码器测试代码设计**

根据指令的比特模式，我们可以将add R1,R2,R3、sub R0,R5,R6、lw R5,100(R2)、sw R5,104(R2)以及JAL 100这几条指令编码为相应的二进制形式，并在图4中展示这些二进制编码：

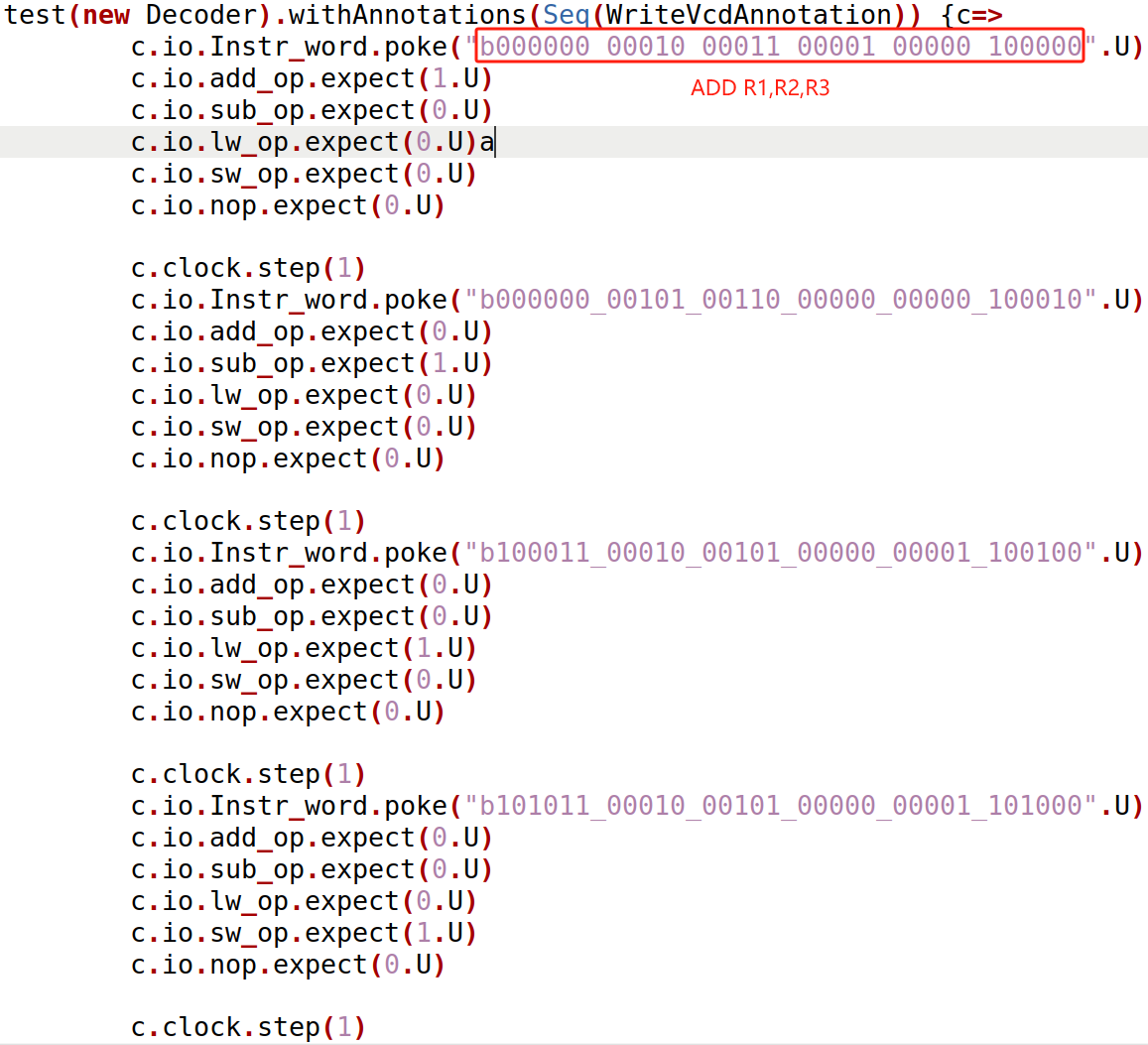


图 4 Decoder测试代码

在模拟过程中，我们使用这五条指令来分别代表加法（add）、减法（sub）、加载字（lw）、存储字（sw）以及一个假设的“空操作或类似nop的指令”。

为了模拟这些指令的执行，我们采用poke和expect方法。Poke方法用于设置输入信号的值，包括寄存器值、内存地址以及指令本身。Expect方法则用于验证输出信号的值，包括运算结果、寄存器更新、内存写入等。

在模拟的每一步中，我们都通过调用clock.step()方法来模拟时钟周期的变化。这个方法会触发指令译码、执行以及任何必要的状态更新。通过多次调用clock.step()，我们可以逐步观察指令的执行过程，以及处理器状态（如寄存器、内存等）的逐步变化。

1. **运行结果**

在完成了代码编写与编译之后，我们进行了代码的运行，并细致地观察了波形。通过截取特定测试阶段的波形图像，我们获得了如图5和图6所示的波形展示。

在图5所呈现的初始时刻波形图中，我们可以清晰地看到io\_nop\_op的输出信号被设定为1，而其他指令的输出信号，如io\_add\_op、io\_sub\_op、io\_lw\_op以及io\_sw\_op等，则均显示为0，这符合我们在初始测试阶段未激活这些指令的预期：

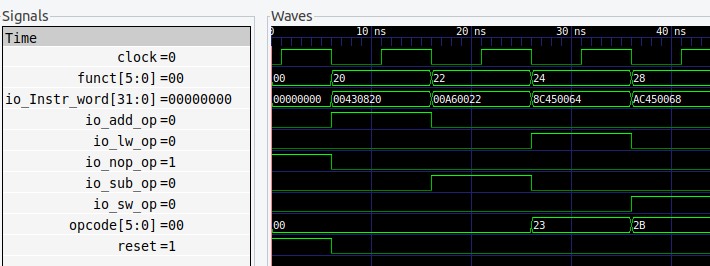


图 5 Decoder测试波形1

进一步地，当我们将输入指令更改为加法（ADD）指令，并观察图6所展示的波形时，我们发现io\_add\_op的输出信号变为了1，这明确地指示了加法指令已被正确译码。与此同时，其他指令的输出信号依然保持为0，这再次验证了译码器对于不同指令的区分能力，并表明它仅对当前激活的加法指令作出了响应。

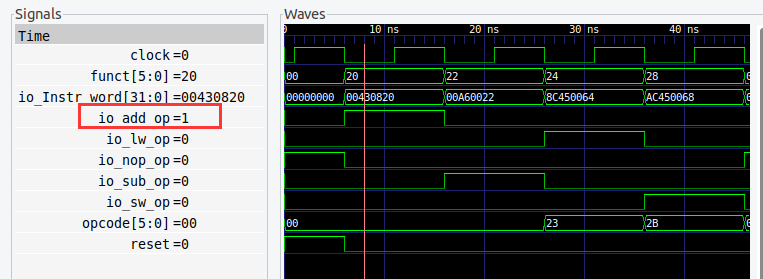


图 6 Decoder测试波形2（ADD指令）

经过对多个指令的测试与波形观察，我们确认所有指令的输出均符合预期，这有力地证明了译码器的设计是成功的。无论是空操作指令，还是加法、减法、加载字或存储字等指令，译码器都能够准确地识别并译码，为后续的指令执行阶段提供了可靠的控制信号。

综上所述，通过编译运行代码、观察波形以及对比分析，我们得出了译码器设计成功的结论。

1. **寄存器文件**

寄存器文件如图7所示。需要说明的是，图7中只给出了写入的数据( WB\_data )和写回数据的寄存器( Reg\_WB )，在实际应用中，应当再添加一个写入使能信号，如图8所示。**（在本实验中，并未实现写入使能信号，可以视作写入使能信号恒为1的情况）**

Reg\_file

RS1[4:0]

RS2[4:0]

WB\_data[31:0]

Reg\_WB

RS1\_out[31:0]

RS2\_out[31:0]

图 7 寄存器文件示意图

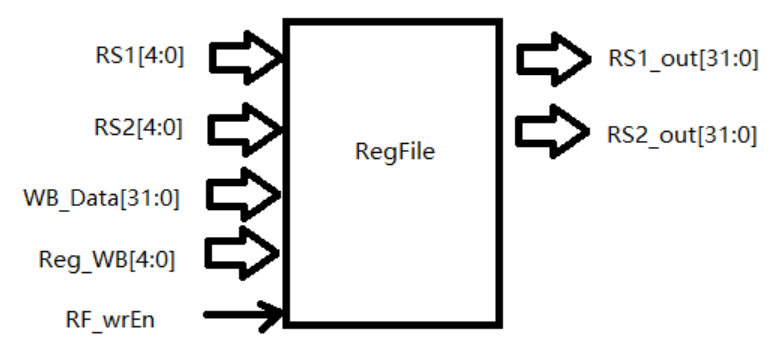


图 8 添加写入使能信号的寄存器文件

1. **寄存器文件设计**

根据寄存器文件示意图定义各个接口。模块定义了输入和输出接口 io，包括写入数据信号、写入数据的寄存器编号、两个输出数据信号，以及两个用于选择要读取的寄存器的输入信号。

首先，我们定义了一个 RegFileIO 类，作为与外部连接的接口，提供了读取源寄存器、写回数据、写回使能等信号。接着，在 RegFile 类中，使用一个32个32位宽度的寄存器数组（regs）来表示寄存器文件，并通过一些逻辑处理实现数据的读写功能。

在寄存器文件中，每个寄存器都可以通过地址来访问，并且可以进行数据写入操作。寄存器的读取操作通过传入的 RS1 和 RS2 地址进行，若地址有效，则输出对应寄存器的内容，否则输出0。为了处理寄存器写入，模块监测 RF\_wrEn（写回使能信号），当该信号为 1 时，会将 WB\_data（写回数据）写入到指定的寄存器 Reg\_WB 中。

整体思路是：寄存器文件在初始化时，将每个寄存器赋值为其索引值（0 到 31）。在每个时钟周期，系统可以根据 RS1 和 RS2 地址，输出相应的寄存器内容；并在使能信号 RF\_wrEn 为 1 时，将新的数据写入到指定寄存器。

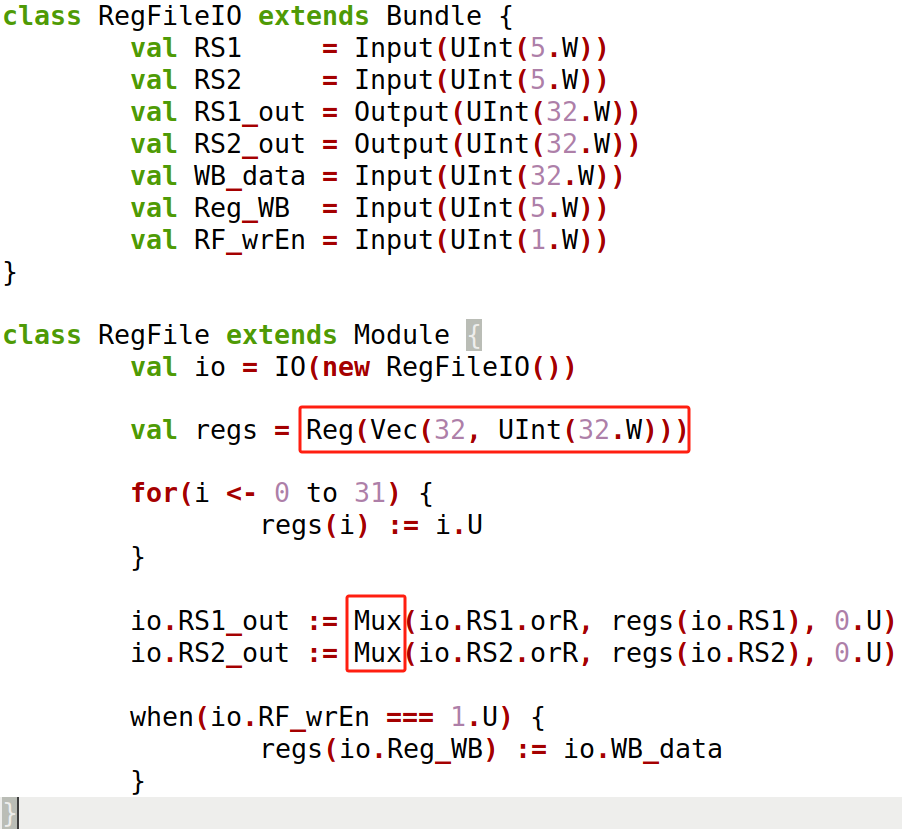


图 9 寄存器文件核心代码

1. **寄存器文件测试代码设计**

针对寄存器文件模块的测试代码，其主要目的在于验证该模块能否准确无误地更新和读取寄存器的值。在测试流程中，我们巧妙地运用了poke函数来设定输入信号的具体数值，并明确指定了写入操作的寄存器编号。随后，利用expect函数来严格校验输出信号是否与预期相符。测试代码如图10所示：

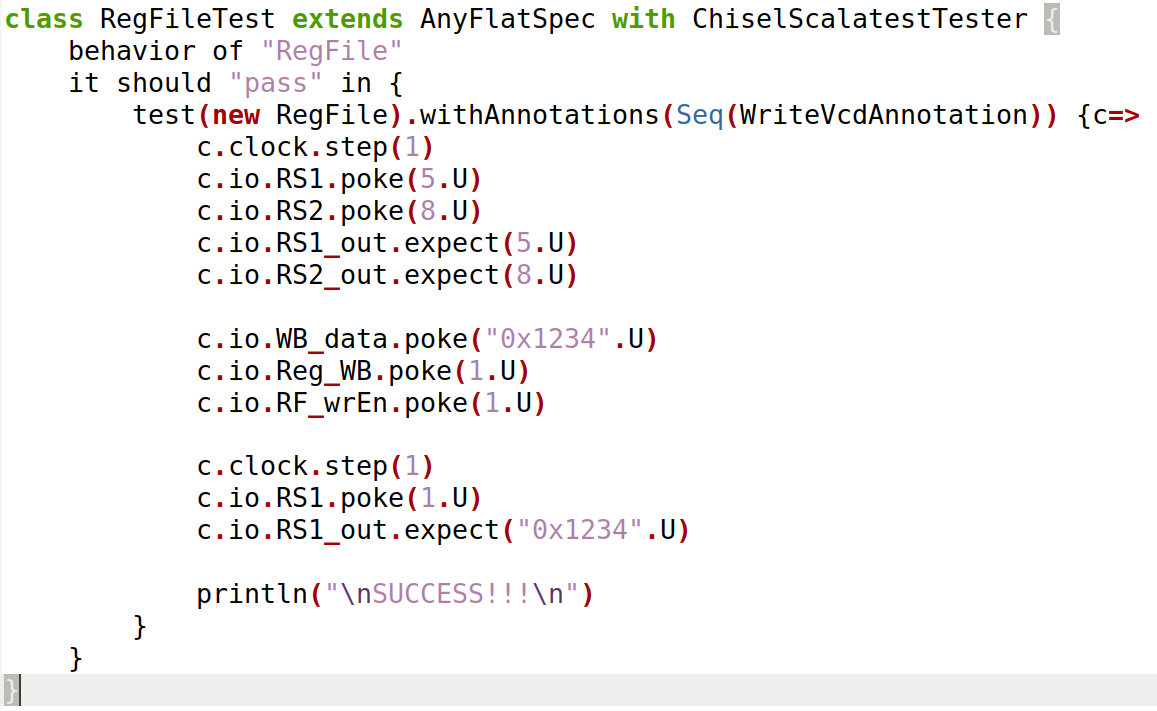


图 10 寄存器文件测试代码

1. **运行结果**

编译运行代码并查看波形图，如图11所示，在clock = 0时，RS1\_OUT和RS2\_OUT分别设置成了5和8，写入数据寄存器为1，写入数据为0x1234，而31个寄存器的值默认都被设置为了对应的寄存器编号。

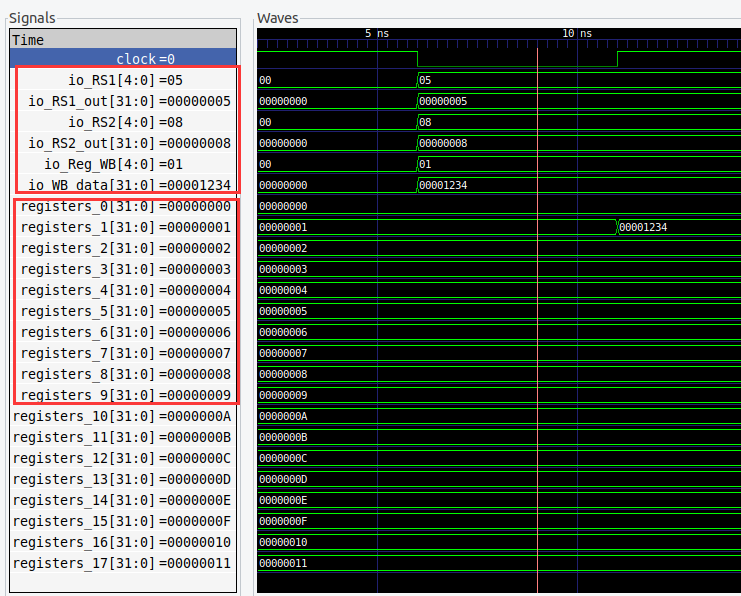


图 2 寄存器文件测试波形图1

时钟向前推进后，clock = 1，如图11所示，寄存器1的值变为了0x1234，即与要写入的数据相等，说明数据成功写入，寄存器文件的设计成功。

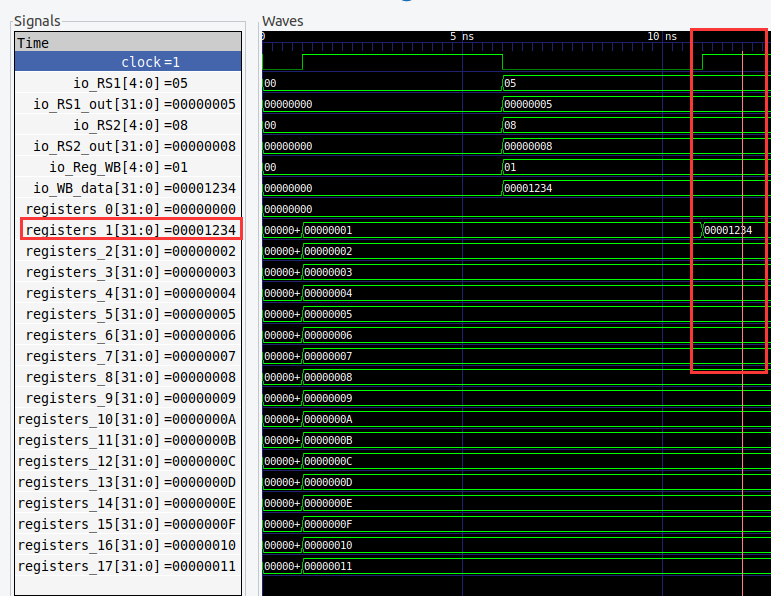


图 3 寄存器文件测试波形图2

1. **指令存储器**

指令存储器的设计图如下所示。其中包括5位的读写地址和32位的写入数据和输出数据以及一个使能信号。

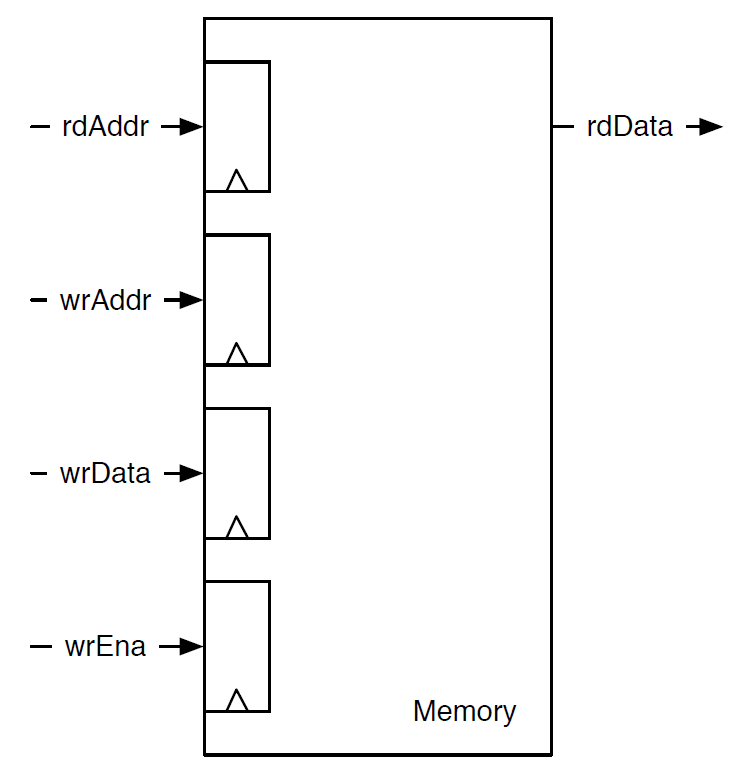


图 4 指令存储器完整设计文件

1. **指令存储器的设计**

Mem类实现了基本的读取和写入操作。模块通过 IO 接口提供了读取地址 rdAddr、读取数据 rdData、写入地址 wrAddr、写入数据 wrData 以及写使能信号 wrEna。在内部，使用 SyncReadMem 定义了一个具有 32 个 32 位宽单元的内存。读取操作通过 mem.read(io.rdAddr) 完成，它会根据传入的地址 rdAddr 从内存中读取数据并输出到 rdData。写入操作通过 mem.write(io.wrAddr, io.wrData) 完成，只有当 wrEna 为 1 时，才会将数据写入指定的地址 wrAddr。

总之，这个模块提供了一个简单的内存读写机制，能够根据给定的地址读取和写入 32 位数据。代码如图13所示：

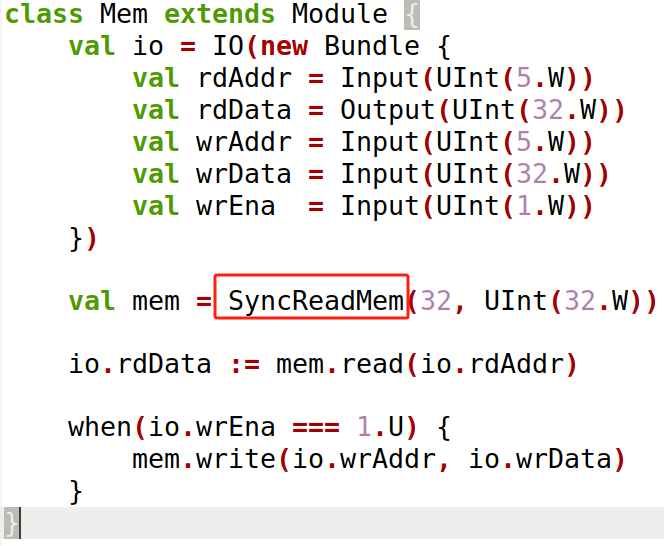


图 5 指令存储器代码

1. **模块连接**

将指令寄存器，寄存器文件，译码电路进行连接，完整的设计图如图14所示。

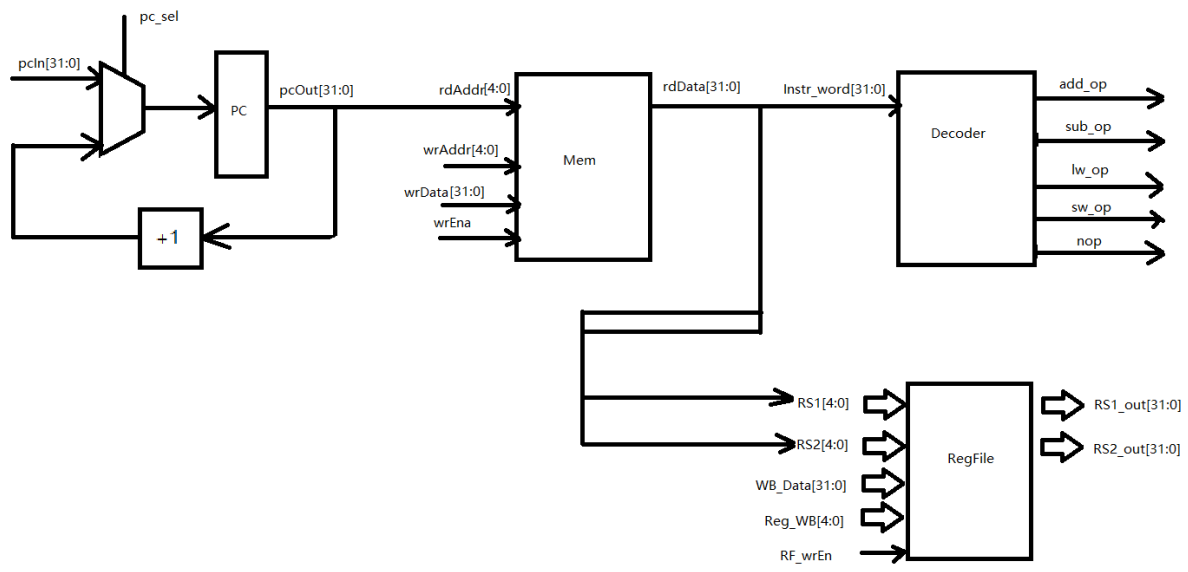


图 6 模块连接完整设计图

首先，设计PC：指令存储器以字编址，pcOut需要+1。

在 PC 模块中，pcIn 是输入信号，表示新的程序计数器值；pc\_sel 是选择信号，决定程序计数器的更新方式；pcOut 是输出信号，表示当前的程序计数器值。模块内部通过一个 RegInit(0.U(5.W)) 定义了一个 5 位宽的寄存器 pc，并初始化为 0。io.pcOut := pc 将寄存器值输出给 pcOut。

程序计数器的更新逻辑通过 Mux 来实现：当 pc\_sel 等于 1 时，程序计数器值将更新为 pcIn；否则，程序计数器的值将自增 1（即 pcOut + 1）。这种设计常见于流水线 CPU 中，pc\_sel 可以控制程序计数器是跳转到指定地址（通过 pcIn）还是按顺序执行下一条指令（pcOut + 1）

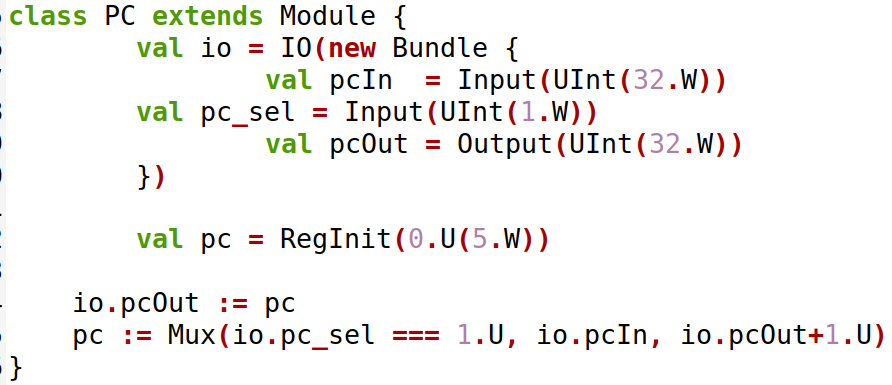


图 7 模块实例化

然后，定义大模块的接口，并实例化译码器、寄存器文件、指令存储器和PC。

Junction 模块集成了多个子模块来执行复杂的计算任务。模块通过接收输入信号并与子模块交互来生成不同的操作信号。输入信号包括写地址 (wrAddr)、写数据 (wrData)、写使能信号 (wrEna)、程序计数器输入 (pcIn)、程序计数器选择信号 (pc\_sel) 等。输出信号包括不同操作类型的控制信号（如加法、减法、加载、存储、无操作）以及从寄存器文件读取的操作数 RS1\_out 和 RS2\_out。

在模块内部，实例化了四个子模块：Decoder 负责解码指令并生成相应的操作信号，RegFile 实现了寄存器文件功能，Mem 模块处理内存的读写操作，PC 模块则负责程序计数器的管理。

Junction 模块通过这些子模块的协作来处理指令，控制各个操作，并根据指令类型更新寄存器或内存，确保计算机执行过程的正常进行：

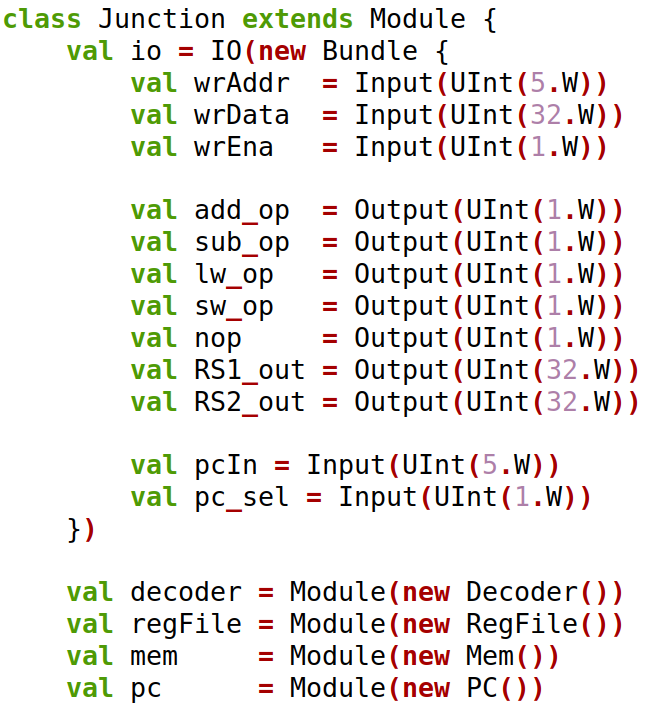


图 8 定义各模块接口

最后，连接PC、译码器、寄存器文件和指令存储器。

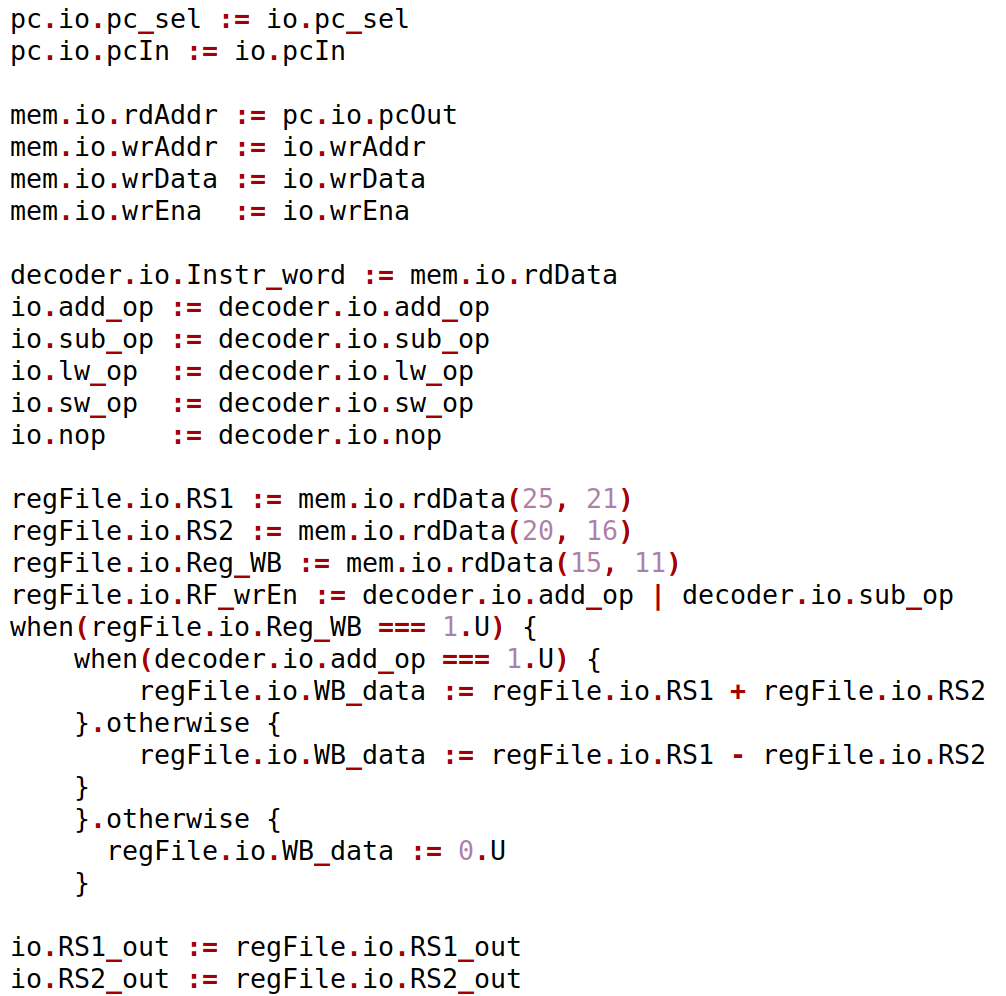


图 9 连接各个模块

首先，程序计数器 (pc) 的输入和选择信号 (pcIn, pc\_sel) 由外部输入传递。pc 模块根据这些信号计算出当前的程序计数器值并输出到内存模块 (mem)。内存模块的读地址 (rdAddr) 由程序计数器的输出 (pcOut) 提供，而写地址 (wrAddr) 和写数据 (wrData) 以及写使能信号 (wrEna) 由外部输入。

内存模块的输出 (rdData) 被传递给解码器 (decoder)，解码器根据指令信息生成不同的操作控制信号，比如加法、减法、加载、存储和无操作信号 (add\_op, sub\_op, lw\_op, sw\_op, nop)。这些控制信号被传递到 Junction 的输出接口，用于外部控制。

接下来，寄存器文件 (regFile) 根据解码器生成的控制信号从内存中读取的指令（rdData）提取操作数地址。regFile 根据这些地址从寄存器中读取数据，执行加法或减法操作，并将结果写回寄存器。写回的数据 (WB\_data) 取决于解码器生成的操作，如果是加法或减法操作，则执行相应的运算，否则将写回数据设置为零。

最后，Junction 模块将寄存器文件的输出（即 RS1\_out 和 RS2\_out）传递到模块的输出接口，提供最终的寄存器值。这些寄存器值将作为后续计算或其他操作的输入。

1. **测试代码的设计**

用clock.step()进行时钟步进，模拟4个时钟周期，并依次用expect函数验证输出接口的值是否符合预期：

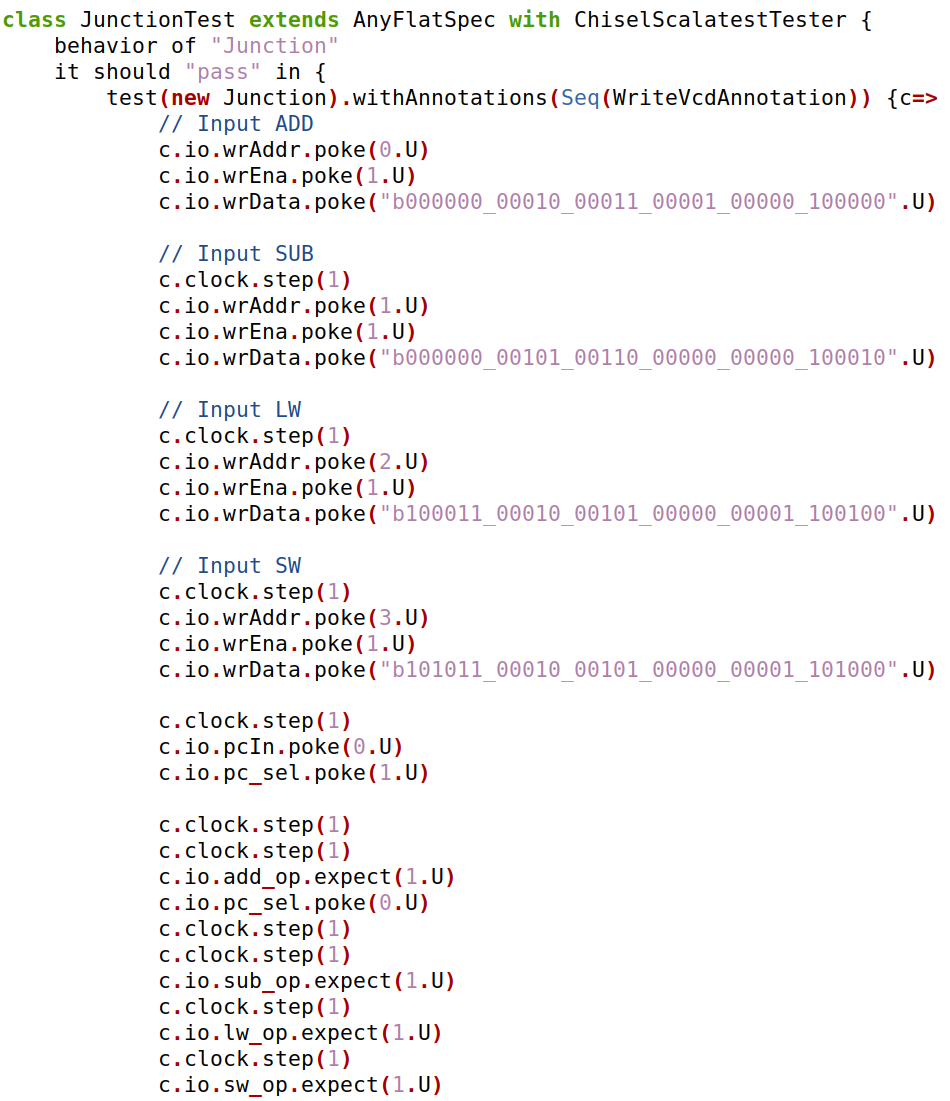


图 10 模块连接测试代码

首先，使用 poke 方法向模块的输入信号传递指令。每个指令是 32 位的二进制指令，代表不同的操作。然后，测试通过时钟 step 来模拟指令的逐步执行。

在每个指令输入后，测试使用 clock.step(1) 来模拟时钟的推进，确保指令能正确地在模块中被处理并产生正确的输出。接着，检查各个操作信号（add\_op、sub\_op、lw\_op、sw\_op）是否按预期触发。例如，检查在输入 ADD 指令后，add\_op 信号是否为 1，表示加法操作已被触发。

最终，测试验证了 Junction 模块在接收到不同类型的指令后，能够正确解码并生成相应的操作信号。

1. **运行结果**

编译运行代码，查看相关的波形图。初始时刻如图19所示，此时程序计数器PC为0，读取了地址0的ADD指令，因此io\_add\_op=1，同时opcode=00，等于ADD指令的操作码，表明电路正在对该条指令进行译码，符合预期。

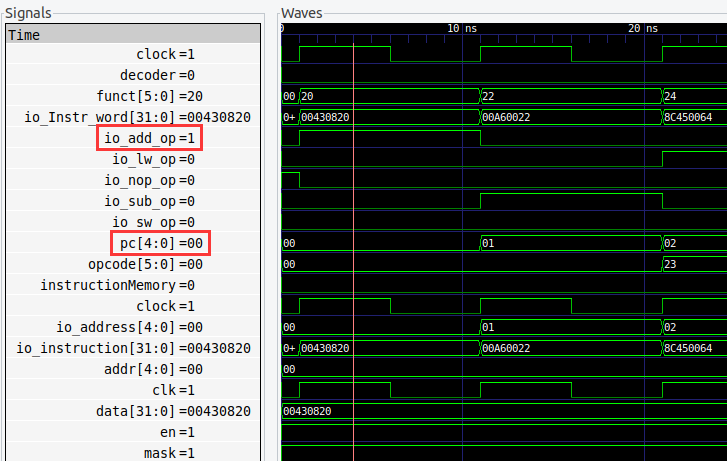


图 19 测试波形图1(ADD指令)

时钟步进，PC+1，如图20所示，在该上升沿处，clock变为1，PC为1，io\_sub\_op=1，表明已取出第二条指令（SUB指令）。

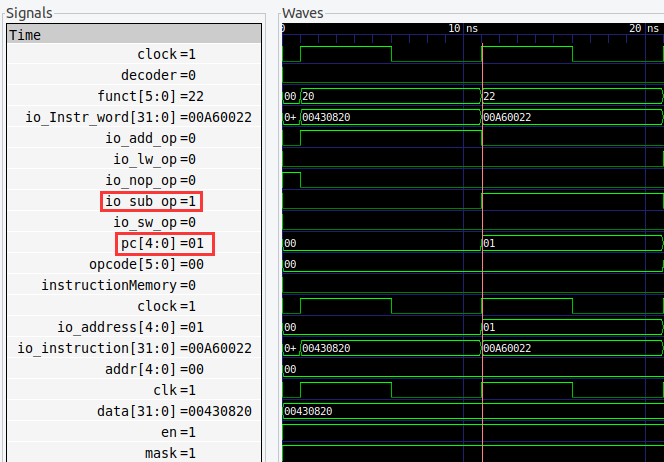


图 20 测试波形图2(SUB指令)

时钟继续推进，结果类似图20，PC++，对应的op值变为1，操作码的值与译码器中的设置相同，表明已读取，

lw和sw指令的测试波形图分别如图21和图22所示:

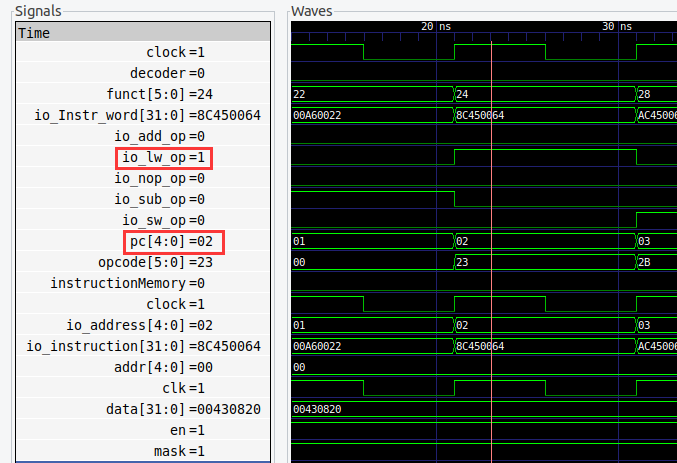


图 21 测试波形图3(lw指令)

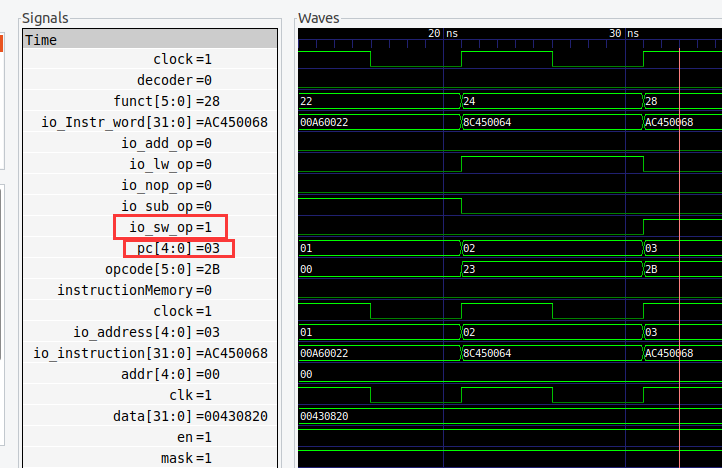


图 22 测试波形图4(sw指令)

测试代码的正确结果以及所观察到符合预期的波形图，说明指令存储器的设计成功，且成功将指令存储器，寄存器文件和译码器连接成电路，实现了电路逐条指令取出、译码的功能。

**六、实验总结与体会**

**思路总结：**

1. 在译码器设计中，采用的方法为，比较指令的 opcode 和 funct来生成控制信号，逻辑较为清晰也更易实现，但如需添加新的指令，需要直接修改译码逻辑，封装性和可扩展性稍弱。
2. 在寄存器文件的设计图中，并未考虑写入使能信号。与同学朋友进行相关的研讨后，认为设计写入使能信号是更为严谨的，而本实验可以默认写入使能信号恒为1，故在实验中未进行深入的设计与研究。如果考虑写入使能信号，可以使用when语句增加条件判断，当写入使能信号为1时，才进行写入，如图25所示。



图 23 写入使能信号判断

1. 指令存储器的设计较为简单，将4条指令从地址0按顺序存入存储器，通过输入地址可以选择相应的指令输出。而模块的连接通过PC来取出指令并更新数据，较为简单。在本实验的设计中，PC的更新为简单的+1，实际上需要考虑跳转指令所导致的变化，可以采取增加写入使能信号的思路，添加条件判断语句来实现PC的更新，如图17所示。

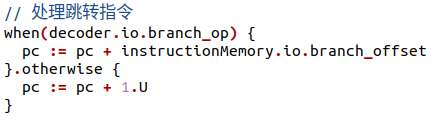


图 24 程序计数器根据跳转指令更新

**体会：**

1. 通过本次实验，我初步掌握了Chisel设计数字系统，并学习了其基本应用。通过使用 Chisel，我学习了如何表达电路的结构和行为，同时也掌握了仿真和验证的方法。
2. 通过本次实验，我学会了如何使用 Chisel 设计一个简单的指令译码电路。通过解析 MIPS 指令格式，我成功实现了对 add、sub、lw、sw 指令的译码，并将其他指令译码成 nop 操作。这个实验让我对数字电路中的信号处理和逻辑设计有了更深的理解。
3. 我对寄存器文件的基本功能和工作原理有了更深的理解，并设计了一个简单的寄存器文件，之后我将指令存储器、寄存器文件和译码电路组合在一起，实现了一个简单的控制电路。通过观察 PC 的变化、指令的取出和译码过程，我更深入地理解了一个基本的 CPU 控制流程。

本次实验的难度较大，但加深了我对计算机体系结构和硬件设计工作的理解，学习到了更多数字电路和计算机组成原理的知识，同时，在分析与解决各种问题的过程中，我的综合能力都得到了较大的提升。

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