深圳大学考试答题纸

(以论文、报告等形式考核专用)

二〇 二四 ～二〇 二五 学年度第 一 学期

课程编号 1504430001 课序号 01 课程名称 数字集成与系统设计 主讲教师 蔡晔 评分

学 号 2022150221 姓名 何泽锋 专业年级 2022级计算机科学与技术(创新班)

教师评语:

题目: LC-3 处理器设计

目录

一、项目背景

LC-3 处理器作为一种简单、高效的教学工具，被广泛应用于计算机组成原理课程中。传统的 LC-3 处理器设计通常采用 Verilog 等硬件描述语言，开发过程复杂且难以调试。Chisel 语言作为一种新兴的硬件设计语言，具有易用性、可读性和可维护性等优点，能够有效地提高硬件设计的效率和质量。因此，本项目旨在利用 Chisel 语言重新设计 LC-3 处理器，并对其进行仿真验证。本次实验共计分为十个部分，从 Chisel 环境的安装到 FPGA 的硬件运行，接下来将逐一介绍。

二、需求分析

1. 实现 LC-3 处理器的基本指令集，包括数据传输、算术运算、逻辑运算等。
2. 设计并实现处理器的各个模块，如 ALU、寄存器堆、控制器、数据通路等。
3. 实现处理器的输入输出机制，包括 UART 通信协议的实现。
4. 完成处理器的仿真环境搭建，确保处理器能够在 FPGA 中正确运行。

三、设计方案

1. 硬件描述语言

使用 Chisel 语言进行硬件设计，Chisel 是一种基于 Scala 的硬件描述语言，它提供了高级语言的特性，使得硬件设计更加简洁和易于理解。

2. 仿真工具

使用 Verilator 作为仿真器，Verilator 能够将 Verilog 代码转换成 C++代码，从而在软件环境中模拟硬件电路的行为。

3. 模块设计

（1）ALU（算术逻辑单元）模块

• 功能：ALU 是处理器中执行所有算术和逻辑操作的模块。它接收操作数和操作类型作为输入，并产生一个输出结果。LC-3 ALU 支持的运算操作包括加法、按位或、按位与、按位非以及将一个操作数传递到输出。

• 输入：两个操作数（操作数 1 和操作数 2），操作类型。

• 输出：根据操作类型，输出相应的运算结果

（2）寄存器堆模块

• 功能：寄存器堆包含一组寄存器，用于存储处理器的状态和数据。LC-3 的寄存器堆包括一组通用寄存器，程序计数器（PC）等。

• 输入：读/写地址，写数据，写使能信号。

• 输出：根据读地址返回对应的寄存器中的数据。

• 特点：需要两组读端口和一组写端口，写操作需要使能信号来确认

（3）控制器模块

• 功能：控制器是处理器的大脑，负责根据当前状态和输入生成控制信号，这些控制信号指导数据通路的行为。

• 输入：上一周期的运算结果和状态。

• 输出：控制信号（微指令），指导数据通路和存储器的工作。

• 特点：需要根据状态转移图来生成控制信号

（4）数据通路模块

• 功能：数据通路是处理器中传输数据的路径，包括各种寄存器和线网。它负责在处理器的不同部分之间传递数据。

- 输入：来自控制器的控制信号，来自寄存器堆的数据。
- 输出：根据控制信号将数据传输到正确的目的地。
- 特点：包括 PC、IR、MAR、MDR 等关键寄存器，以及各种选择器和线网。

（5）存储器模块

- 功能：存储器模块用于存储程序指令和数据。它是 CPU 执行程序的基础。
- 输入：地址，写数据，写使能信号。
- 输出：根据地址返回存储器中的数据。
- 特点：需要初始化程序代码和数据到指定位置。

（6）UART（通用异步接收/发送）模块

- 功能：UART 模块负责处理器的串行通信，包括数据的发送和接收。
- 输入：要发送的数据。
- 输出：接收到的数据。
- 特点：实现 UART 通信协议，包括起始位、数据位、奇偶校验位和停止位。

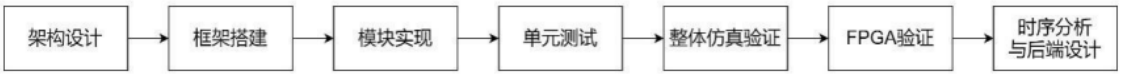
（7）Boot 模块

- 功能：Boot 模块负责在系统启动时加载程序到内存，并设置程序的起始执行地址。
- 输入：需要加载的程序指令。
- 输出：初始化完成信号，程序起始地址。
- 特点：需要与内存模块协作，将程序写入内存。

（8）Top 模块

- 功能：Top 模块是系统的顶层模块，负责连接和协调各个子模块的工作。
- 输入：外部输入，包括程序指令和控制信号。
- 输出：处理器的输出，包括计算结果和状态信息。
- 特点：作为系统的核心，需要确保所有模块正确协同工作。

完整的芯片前端设计流程如下：



四、实现过程

1. 搭建 Chisel 开发环境

- 了解芯片设计开发流程，建立基础概念
- 掌握 Chisel 开发环境的搭建
- 根据实验指导编译与运行 3-8 译码器项目

1.1. 配置 Chisel 环境

Chisel 是一种构建在 Scala 语言之上的领域专用语言，Chisel 相比于传统的硬件开发语言，有更多高级语言的功能。在编译时，会将 Chisel 代码编译生成 verilog 文件，然后再使用 verilog 文件仿真，其中 verilog 是一种传统的硬件设计语言，与 Chisel 的关系类似于汇编语言和 C 语言的关系。

（1）配置 mill

Mill 是一个项目构建工具，主要用于将 Chisel 编译成 verilog 文件。使用命令安装后可以通过 mill -v 指令检查安装是否正确。

```
lc3@lc3-virtual-machine:~$ mill -v
Mill Build Tool version 0.11.8
Java version: 13.0.7, vendor: Private Build, runtime: /usr/lib/jvm/java-13-openjdk-amd64
Default locale: en_US, platform encoding: UTF-8
OS name: "Linux", version: 5.15.0-119-generic, arch: amd64
```

图 1 检查 mill 安装情况

（2）Verilator 介绍

Verilator 是一种开源免费的 verilog 仿真器，可以通过软件模拟的方式仿真运行硬件电路，主要工作原理是读取 RTL 代码，将其转换成用于仿真的 C++项目，再通过编译运行 C++项目，就可以在电脑上对硬件电路进行仿真。使用指令安装后可以通过 verilator --version 查看版本

```
lc3@lc3-virtual-machine:~$ verilator --version
Verilator 4.028 2020-02-06 rev v4.026-92-g890cecc1
```

图 2 检查 verilator 安装情况

1.2. 完成 3-8 译码器的仿真流程

（1）3-8 译码器是将一个 3 位的二进制数 n 转换为 8 个信号，其中第 n 个信号为高电平，用 1 表示，其余的信号为低电平，用 0 表示，其中 3-8 译码器的对应的真值表如下：

表 1 3-8 译码器真值表

输入 (A2, A1, A0)	输出 (Y7, Y6, Y5, Y4, Y3, Y2, Y1, Y0)
000	00000001
001	00000010
010	00000100
011	00001000
100	00010000
101	00100000
110	01000000
111	10000000

（2）编写 3-8 译码器 Chisel 代码，实现思路是根据 io 输入选择对应的 io 输出，使用 switch-is 语句对每种输入进行判断即可

```
package decoder

import chisel3._
import chisel3.util._

// RawModule与Module不同，它不会生成隐式的时钟，由于我们是组合电路，因此现在还没有涉及到时钟的概念，只用RawModule即可
class Decoder extends RawModule {
    val io = IO(new Bundle{
        val in = Input(UInt(3.W)) // 输入的0-7信号，二进制表示只需要3bits宽即可
        val out = Output(UInt(8.W)) // 输出的译码后信号，8bits宽
    })

    // 在Chisel中，可以对同一个变量多次赋值，以最后一次赋的值为最终结果
    // 因此如果下面的switch语句都没有执行，这一行能给io.out一个默认值
    io.out := "b00000000".U

    switch (io.in) {
        is (0.U) { io.out := "b00000001".U }
        is (1.U) { io.out := "b00000010".U }
        is (2.U) { io.out := "b00000100".U }
        is (3.U) { io.out := "b00001000".U }
        is (4.U) { io.out := "b00010000".U }
        is (5.U) { io.out := "b00100000".U }
        is (6.U) { io.out := "b01000000".U }
        is (7.U) { io.out := "b10000000".U }
    }
}

object testMain extends App {
    // 实例化Decoder模块
    Driver.execute(args, () => new Decoder)
}
```

图 3 代码实现 3-8 译码器

（3）执行这个命令后对 3-8 译码器项目进行仿真运行，对比真值表可看到结果正确，根据对应的输入得到了所需的输出

```
lc3@lc3-virtual-machine:~/Frontend/decoder$ ./build/emu
in: 0   out: 00000001
in: 1   out: 00000010
in: 2   out: 00000100
in: 3   out: 00001000
in: 4   out: 00010000
in: 5   out: 00100000
in: 6   out: 01000000
in: 7   out: 10000000
```

图 4 3-8 译码器测试结果

2. Chisel 编译流程介绍

- 掌握 Chisel 基本语法
- 掌握 Chisel 从编译到生成 verilog 的过程
- 能够使用 Chisel 编写简单的电路
- 掌握 Chiseltest 的使用

2.1. 画出 1011 序列检测电路的状态转移图，完成对序列检测电路的修改

（1）使用有限状态机，即表示有限个状态以及在这些状态之间的转移和动作等行为的模型，状态机的下一个状态由当前状态和当前输入共同决定。根据功能需求，可以设置以下几个状态，

S0：代表之前输入的信号为 0，当检测到输入信号为 1 时，进入 S1，否则维持 S0。

S1：代表之前输入的信号为 1，当检测到输入信号为 0 时，进入 S2，否则维持 S1。

S2：代表之前输入的信号为 10，当检测到输入信号为 1 时，进入 S3，否则回到 S0。

S3：代表之前输入的信号为 101，当检测到输入信号为 1 时，检测到目标序列，输出高电平，同时进入 S1，否则回到 S2

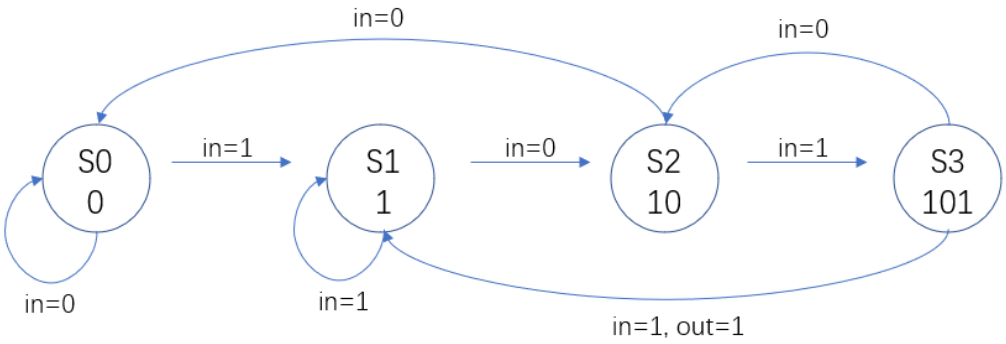


图 5 1011 状态转移图

（2）此处首先介绍 Chisel 的基本用法，要使用 Chisel 首先需要应用 Chisel 相关的包才能使用其语法。接下来定义主体类 Detection，它继承自一个 Module 类，，Module 模块默认的接口中会有一个 clock 时钟信号和一个 reset 信号，是时序逻辑电路。除此之外还有一个 in 和一个 out 信号，分别作为电路的输入和输出信号。

在硬件编程中，寄存器的赋值并不是立即生效，给寄存器赋值后，新的值要在下个周期才会被更新，当周期寄存器独处的数据依然是赋值之前的旧值。且每周期只能给寄存器赋一次值，如果有多条给同一个寄存器赋值的语句，则会以最后一条赋值语句为准。

接下来，可以编写代码识别上述状态转移图，此处采用 switch-is 语句，通过判断当前输入以及当前状态来决定跳转的状态。

```scala
package detection

import chisel3._
import chisel3.util._
import chisel3.stage._

class Detection extends Module {
    val io = IO(new Bundle{
        val in = Input(Bool())
        val out = Output(Bool())
    })

    val S0 = 0.U(2.W) // 0
    val S1 = 1.U(2.W) // 1
    val S2 = 2.U(2.W) // 11
    val S3 = 3.U(2.W) // 110

    val stat = RegInit(0.U(2.W))

    switch(stat) {
        is (S0) { when(io.in) {stat := S1} .otherwise {stat := S0} }
        is (S1) { when(io.in) {stat := S1} .otherwise {stat := S2} }
        is (S2) { when(io.in) {stat := S3} .otherwise {stat := S0} }
        is (S3) { when(io.in) {stat := S1} .otherwise {stat := S2} }
    }

    printf(p"in = ${io.in}, out = ${io.out}\n")
    io.out := stat === S3 && io.in
}

object testMain extends App {
    (new ChiselStage).execute(args, Seq(
        ChiselGeneratorAnnotation(() => new Detection))
    )
}
```

图 6 代码实现 1011 序列检测

（3）要检验上述代码是否能够正确实现其功能，可以采用单元测试。需要注意的是，单元测试是一个软件程序，使用的是 Scala 语言，某些地方与 Chisel 语法是不同的。

修改检测序列，此处定义了一个确定的输入以及对应的输出序列，使得满足 1011 的序列状态检测高电平，其余状态检测低电平。检测逻辑为，将测试值输入到状态图之中，检测其运行结果是否与期待结果一致，若一致则为 true。如下图分别是测试模块的流程图，以及此处的实际测试代码

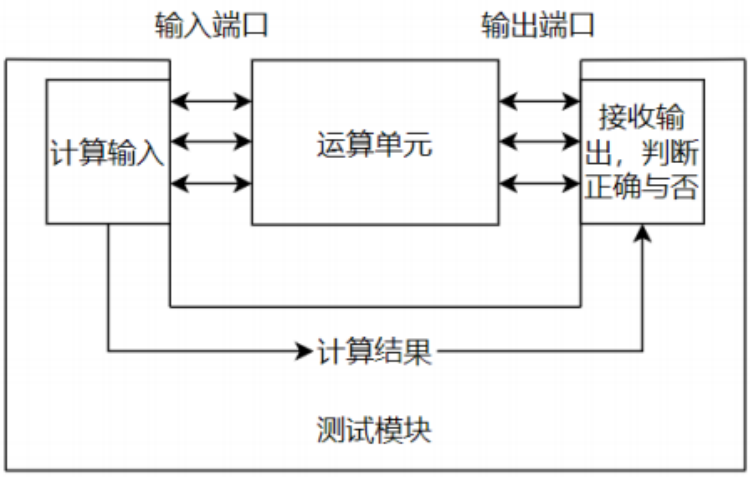


图 7 测试模块逻辑图

```scala
package detection

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

class DetectionTest extends AnyFlatSpec with ChiselScalatestTester {
    behavior of "DetectionTest"

    it should "test Detection" in {
        test(new Detection) { c =>
            var seq_in  = Array(1,1,1,0,1,1,0,1,0,1,1,0,1,0)

            // 检测1101序列的参考输出
            //var seq_out = Array(0,0,0,0,1,0,0,1,0,0,0,0,1,0)

            // 检测1011序列的参考输出
            var seq_out = Array(0,0,0,0,0,1,0,0,0,0,1,0,0,0)

            for (i <- 0 until seq_in.length) {
                c.io.in.poke(seq_in(i).B)
                c.io.out.expect(seq_out(i).B)
                c.clock.step() // 执行一个时钟周期的仿真
            }
        }
    }
}
```

图 8 1011 序列测试代码

（4）使用 mill 编译运行测试和源代码，可以看到通过了测试，仿真结果与测试代码期望的一致

```
lc3@lc3-virtual-machine:~/Frontend/detection$ mill -i detection.test.testOnly detection.DetectionTest
[74/74] detection.test.testOnly
DetectionTest:
DetectionTest
in =  1, out =  0
in =  1, out =  0
in =  1, out =  0
in =  0, out =  0
in =  1, out =  0
in =  1, out =  1
in =  0, out =  0
in =  1, out =  0
in =  0, out =  0
in =  1, out =  0
in =  1, out =  1
in =  0, out =  0
in =  1, out =  0
in =  0, out =  0
- should test Detection
```

图 9 测试 1011 状态转移代码

3. LC3 系统-ALU 模块设计

- 熟悉 LC3 微结构。
- 学习设计 ALU 的前置 Chisel 语法。
- 掌握 LC3 系统中 ALU 模块的设计。
- 掌握使用单元测试验证 LC3 系统中的 ALU 模块。

3.1. 编写 ALU 逻辑

（1）LC3 ALU 模块中需要接收三个输入，即操作数 1，操作数 2，和操作类型，并且得到一个输出。LC3 ALU 中支持的运算操作共四类，加法、按位或、按位反以及取操作数 1。如图为 ALU 模块的 IO 部分以及操作类型

端口名称	方向	位宽
操作数1	输入	16 bits
操作数2	输入	16 bits
操作类型	输入	2 bits
输出数据	输出	16 bits

操作类型	功能
SIG_ALUK = 0	加法
SIG_ALUK = 1	按位与
SIG_ALUK = 2	对操作数1取反
SIG_ALUK = 3	取操作数1

图 10 ALU 输入 IO 以及操作类型

（2）代码实现 ALU 模块，io 输入如上表所示，此处通过 switch-is 语句来根据操作类型对操作数进行运算，具体来说，通过输入的 io.op 决定当前 ALU 模块需要进行的运算，然后使用对应的运算规

则处理输入的数据

```scala
package LC3

import chisel3._
import chisel3.experimental._
import chisel3.util._

class ALU extends Module{
  val io = IO(new Bundle{
    val ina = Input(UInt(16.W))
    val inb = Input(UInt(16.W))
    val op  = Input(UInt(2.W))    //ADD,AND,NOT,PASSA
    val out = Output(UInt(16.W))
  })

  io.out := DontCare

  // 实验三
  // 在此编写ALU逻辑
  switch(io.op) {
    is(0.U) { io.out := io.ina + io.inb } // ADD
    is(1.U) { io.out := io.ina & io.inb } // AND
    is(2.U) { io.out := ~io.ina }         // NOT
    is(3.U) { io.out := io.ina }          // PASSA
  }
}
```

图 11 ALU 选择逻辑

（3）模仿 add 测试用例，在 ALUTest.scala 文件中编写 and, not, pass 功能的测试用例。测试逻辑与前面的一致，通过传入确定的输入值，然后判断各个时钟周期 ALU 的运算结果是否与预期结果一致，若每个周期都一致则通过测试，若 expect 值出现不同则系统输出错误信息。

```scala
  it should "test and" in {
    test(new ALU) { c =>
      for(i <- 0 until TEST_SIZE) {
        c.io.op.poke(1.U)
        c.io.ina.poke(ina(i).U)
        c.io.inb.poke(inb(i).U)
        c.io.out.expect(and_out(i).U(15,0))
        println(s"${i}. ina=${ina(i)}  inb=${inb(i)}")
        println(s"${i}. 标准结果 and_out=${and_out(i) % (1<<16)} 模块结果 io.out=${c.io.out.peek}")
      }
    }
  }

  it should "test not" in {
    test(new ALU) { c =>
      for(i <- 0 until TEST_SIZE) {
        c.io.op.poke(2.U)
        c.io.ina.poke(ina(i).U)
        c.io.out.expect(not_out(i).U(15,0))
        println(s"${i}. ina=${ina(i)}")
        println(s"${i}. 标准结果 not_out=${not_out(i) % (1<<16)} 模块结果 io.out=${c.io.out.peek}")
      }
    }
  }

  it should "test pass" in {
    test(new ALU) { c =>
      for(i <- 0 until TEST_SIZE) {
        c.io.op.poke(3.U)
        c.io.ina.poke(ina(i).U)
        c.io.out.expect(pass_out(i).U(15,0))
        println(s"${i}. ina=${ina(i)}")
        println(s"${i}. 标准结果 pass_out=${pass_out(i) % (1<<16)} 模块结果 io.out=${c.io.out.peek}")
      }
    }
  }
}
```

图 12 测试 ALU 各个操作是否正确

（4）运行测试代码，可以观察到各个功能都与预期一致，此处输入数据虽然是一致的，但经过对应的操作运算的输出结果是不同的，具体数值可参考下方的测试结果：

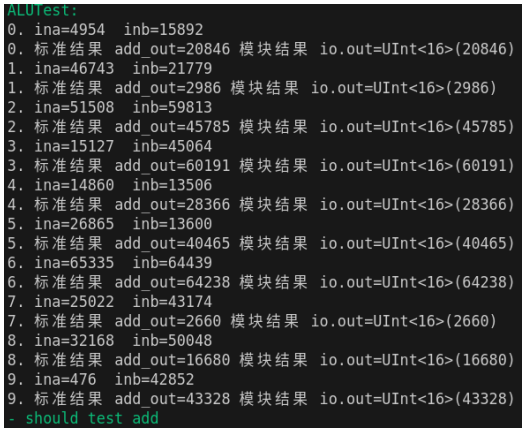


图 13 add 测试

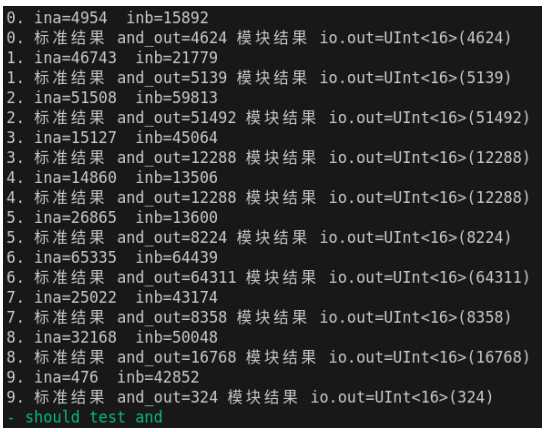


图 14 and 测试

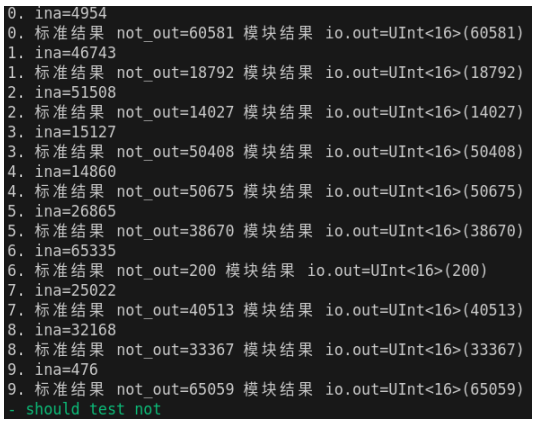


图 15 not 测试

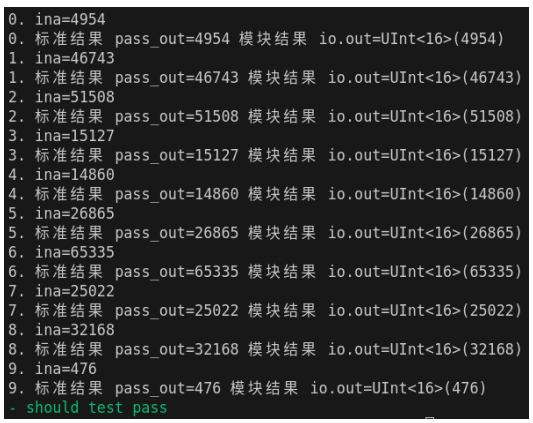


图 16 pass 测试

4. LC3 系统-寄存器堆模块设计

- 掌握 LC3 系统中寄存器堆模块的设计
- 学习设计 Regfile 的前置 Chisel 语法。
- 掌握使用单元测试验证 LC3 系统中的寄存器堆模块。

4.1. 编写寄存器堆逻辑

（1）寄存器堆包含的操作主要是读和写。在 LC3 的指令中最多需要读取两个数据，写入一个数据。因此需要两组读端口一组写端口，每组端口需要一个地址和数据。此外还需要一个信号来判断是否为写信号，只有该信号为真才可以改变寄存器里的数据，由于 LC3 寄存器堆中有 8 个寄存器，因此读写地址可以用 3bits 索引。其次寄存器宽度为 16 位，所以读写数据位宽为 16 位

表 2 寄存器堆输入端口信息

端口名称	方向	位宽
读地址 1	输入	3 bits
读数据 1	输出	16 bits
读地址 2	输入	3 bits
读数据 2	输出	16 bits
写地址	输入	3 bits
写数据	输入	16 bits
写使能	输入	1 bits

（2）代码实现，主要分为读和写两个部分，读的部分根据 io 输入的两个读地址，返回对应地址寄存器中的数据，写部分会多一个使能信号，只有当使能信号为高时才能将数据写入地址对应的寄存器

```scala
package LC3

import chisel3._
import chisel3.util._

class Regfile extends Module{
  val io = IO(new Bundle {
    val wen = Input(Bool())
    val wAddr = Input(UInt(3.W))
    val r1Addr = Input(UInt(3.W))
    val r2Addr = Input(UInt(3.W))
    val wData = Input(UInt(16.W))
    val r1Data = Output(UInt(16.W))
    val r2Data = Output(UInt(16.W))
  })

  io.r1Data := DontCare
  io.r2Data := DontCare

  // 实验四 任务一
  // 在此编写寄存器堆逻辑（定义寄存器堆、写逻辑、读逻辑）
  //val regs = Mem(8, UInt(16.W))
  val regs = RegInit(VecInit(Seq.fill(8)(0.U(16.W))))

  // 写逻辑
  when(io.wen) {
    regs(io.wAddr) := io.wData
  }

  // 读逻辑
  io.r1Data := regs(io.r1Addr)
  io.r2Data := regs(io.r2Addr)
}
```

图 17 寄存器堆实现代码

（3）实现测试代码，输入一个读操作时需要输入 wen, raddr, rdata，输入一个写操作时需要输入 wen, waddr, wdata。需要注意写入后需要一个时钟周期才能读到该数据。此处每轮输入的数据、寄存器地址都是通过随机数产生，使得结果更真实，并且读和写操作是交替进行的，确保写的数据能够并正常读取。

```scala
package LC3

import chisel3._
import chiseltest._
import org.scalatest.flatspec.AnyFlatSpec
import scala.util.Random

class RegfileTest extends AnyFlatSpec with ChiselScalatestTester {
  behavior of "Regfile"

  def TEST_SIZE = 10

  val data1 = Array.fill(TEST_SIZE)(0)
  val data2 = Array.fill(TEST_SIZE)(0)
  val addr1 = Array.fill(TEST_SIZE)(0)
  val addr2 = Array.fill(TEST_SIZE)(0)

  for (i <- 0 until TEST_SIZE) {
    data1(i) = scala.util.Random.nextInt(0xffff)
    data2(i) = scala.util.Random.nextInt(0xffff)
    addr1(i) = scala.util.Random.nextInt(7)
    addr2(i) = scala.util.Random.nextInt(7)
  }

  it should "test r/w" in {
    test(new Regfile) { c =>
      for(i <- 0 until TEST_SIZE) {
        c.io.wData.poke(data1(i).U)
        c.io.wAddr.poke(addr1(i).U)
        c.io.wen.poke(true.B)
        c.clock.step()

        c.io.r1Addr.poke(addr1(i).U)
        c.io.wen.poke(false.B)
        c.io.r1Data.expect(data1(i).U(15,0))
        c.clock.step()

        c.io.wData.poke(data2(i).U)
        c.io.wAddr.poke(addr2(i).U)
        c.io.wen.poke(true.B)
        c.clock.step()

        c.io.r2Addr.poke(addr2(i).U)
        c.io.wen.poke(false.B)
        c.io.r2Data.expect(data2(i).U(15,0))
        c.clock.step()
      }
    }
  }
}
```

图 18 测试寄存器堆读写逻辑

（4）测试结果如下，可以看到通过测试。

```
lc3@lc3-virtual-machine:~/Frontend/chisel_lc3$ mill chisel_lc3.test.testOnly LC3.RegfileTest
[37/71] chisel_lc3.compile
[info] compiling 9 Scala sources to /home/lc3/Frontend/chisel_lc3/out/chisel_lc3/compile.dest/classes
[warn] 201 feature warnings; re-run with -feature for details
[warn] one warning found
[info] done compiling
[71/71] chisel_lc3.test.testOnly
RegfileTest:
Regfile
- should test r/w
```

图 19 寄存器堆逻辑正确

5. LC3 系统-控制器模块设计

- 掌握 LC3 系统中控制器模块的设计
- 掌握使用单元测试验证 LC3 系统中的控制器模块。

5.1. 代码实现状态转移图

Controller 是计算机中最为核心的一个部件，控制着机器的运行行为。在 LC3 中，Controller 的功能为接收上一周期的运算结果，根据上周期的状态生成本周期的状态，最后根据生成的状态查表得到控制信号（微指令），并传送给数据通路和存储器，让其做出对应的工作。

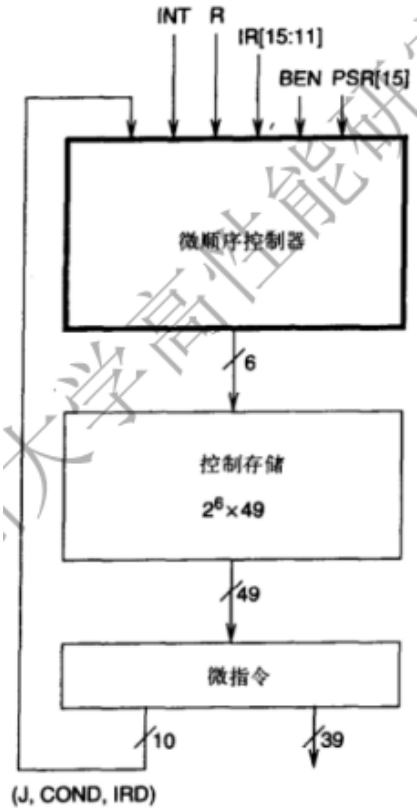
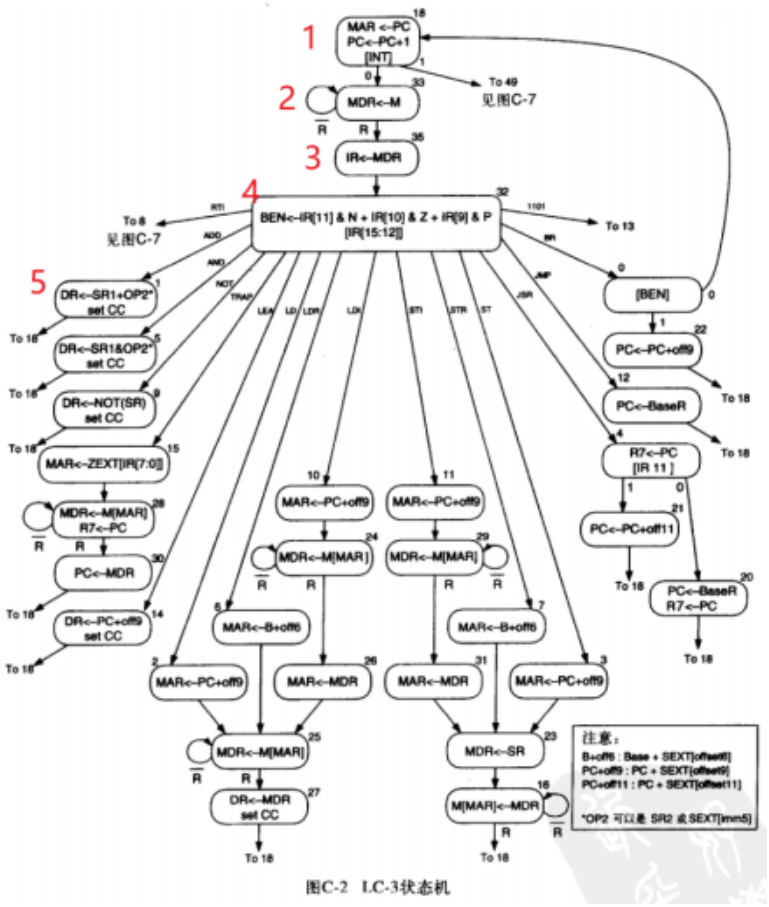


图 20 控制器

（1）参照状态转移图，补充 1~59 的状态转移。需要注意 17 19 46 53 55~58 没有对应状态，判断一条指令是否完成：从状态 18 开始，通过激励满足转移的条件，使得状态进行变化，如此类推直到状态回到 18，表示完成一条指令的执行



图C-2 LC-3状态机

图 21 状态转移图

（2）代码补充如下，可以看到，对于每一种状态（除了部分无对应状态）都根据状态机编写了其转移状态；此处举例状态 8，当处于状态 8 时，下一状态会根据 psr 的信号进行状态转移，当 psr 为真时会跳转到状态 44，当 psr 为假时会跳转到状态 36，对于其他状态的转移代码可参考以下代码：

```
when(io.work && !io.end){
  switch (state) {              // 控制状态机
    // 此处为示例：当前状态为0，下一状态根据ben信号转移，若为真，则下一状态为22，否则为18
    is (0.U) { state := Mux(ben, 22.U, 18.U) }
    is (1.U) { state := 18.U }
    is (2.U) { state := 25.U }
    is (3.U) { state := 23.U }
    is (4.U) { state := Mux(ir(0), 21.U, 20.U) }
    is (5.U) { state := 18.U }
    is (6.U) { state := 25.U }
    is (7.U) { state := 23.U }
    is (8.U) { state := Mux(psr, 44.U, 36.U)}
    is (9.U) { state := 18.U }
    is (10.U) { state := 24.U }
    is (11.U) { state := 29.U }
    is (12.U) { state := 18.U }
    is (13.U) { state := Mux(psr, 45.U, 37.U) }
    is (14.U) { state := 18.U }
    is (15.U) { state := 28.U }
    is (16.U) { state := Mux(r, 18.U, 16.U) }
    // no 17 state
    is (18.U) { state := Mux(int, 49.U, 33.U) }
    // no 19 state
    is (20.U) { state := 18.U }
    is (21.U) { state := 18.U }
    is (22.U) { state := 18.U }
    is (23.U) { state := 16.U }
    is (24.U) { state := Mux(r, 26.U, 24.U) }
    is (25.U) { state := Mux(r, 27.U, 25.U) }
    is (26.U) { state := 25.U }
    is (27.U) { state := 18.U }
    is (28.U) { state := Mux(r, 30.U, 28.U) }
    is (29.U) { state := Mux(r, 31.U, 29.U) }
    is (30.U) { state := 18.U }
    is (30.U) { state := 18.U }
    is (31.U) { state := 23.U }
    is (32.U) { state := ir }
    is (33.U) { state := Mux(r, 35.U, 33.U) }
    is (34.U) { state := Mux(psr, 59.U, 51.U) }
    is (35.U) { state := 32.U }
    is (36.U) { state := Mux(r, 38.U, 36.U)}
    is (37.U) { state := 41.U }
    is (38.U) { state := 39.U }
    is (39.U) { state := 40.U }
    is (40.U) { state := Mux(r, 42.U, 40.U) }
    is (41.U) { state := Mux(r, 43.U, 41.U) }
    is (42.U) { state := 34.U }
    is (43.U) { state := 47.U }
    is (44.U) { state := 45.U }
    is (45.U) { state := 37.U }
    // no 46 state
    is (47.U) { state := 48.U }
    is (48.U) { state := Mux(r, 50.U, 48.U) }
    is (49.U) { state := Mux(psr, 45.U, 37.U) }
    is (50.U) { state := 52.U }
    is (51.U) { state := 18.U }
    is (52.U) { state := Mux(r, 54.U, 52.U) }
    // no 53 state
    is (54.U) { state := 18.U }
    // no 55-58 state
    is (59.U) { state := 18.U }
}
```

图 22 代码实现各个状态的转移

（3）测试指令

①测试 ADD 指令：其状态状态转移顺序为 18, 33, 35, 32, 1 最后转移回到 18。

```
it should "test state machine" in {
  test(new Controller) { c =>

    // 初始状态
    c.io.work.poke(true.B)
    c.io.end.poke(false.B)
    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

    // add指令状态转移
    c.io.in.int.poke(false.B)
    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

    c.io.in.r.poke(true.B)
    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

    c.io.in.ir.poke(1.U)
    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

  }
}
```

```
[71/71] chisel_lc3.test.testOnly
ControllerTest:
Controller
io.state=UInt<6>(18)
io.state=UInt<6>(33)
io.state=UInt<6>(35)
io.state=UInt<6>(32)
io.state=UInt<6>(1)
io.state=UInt<6>(18)
- should test state machine
```

图 23 add 指令状态转移测试代码以及测试结果

②测试 AND 指令：其状态状态转移顺序为 18, 33, 35, 32, 5 最后转移回到 18。

```
it should "test state machine" in {
  test(new Controller) { c =>

    // 初始状态
    c.io.work.poke(true.B)
    c.io.end.poke(false.B)
    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

    // add指令状态转移
    c.io.in.int.poke(false.B)
    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

    c.io.in.r.poke(true.B)
    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

    c.io.in.ir.poke(5.U)
    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

  }
}
```

```
[71/71] chisel_lc3.test.testOnly
ControllerTest:
Controller
io.state=UInt<6>(18)
io.state=UInt<6>(33)
io.state=UInt<6>(35)
io.state=UInt<6>(32)
io.state=UInt<6>(5)
io.state=UInt<6>(18)
- should test state machine
```

图 24 and 指令状态转移测试代码以及测试结果

③测试 LEA 指令：其状态状态转移顺序为 18, 33, 35, 32, 14 最后转移回到 18。

```
it should "test state machine" in {
  test(new Controller) { c =>

    // 初始状态
    c.io.work.poke(true.B)
    c.io.end.poke(false.B)
    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

    // add指令状态转移
    c.io.in.int.poke(false.B)
    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

    c.io.in.r.poke(true.B)
    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

    c.io.in.ir.poke(14.U)
    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

    c.clock.step()
    println(s"io.state=${c.io.state.peek}")

  }
}
```

```
[71/71] chisel_lc3.test.testOnly
ControllerTest:
Controller
io.state=UInt<6>(18)
io.state=UInt<6>(33)
io.state=UInt<6>(35)
io.state=UInt<6>(32)
io.state=UInt<6>(14)
io.state=UInt<6>(18)
- should test state machine
```

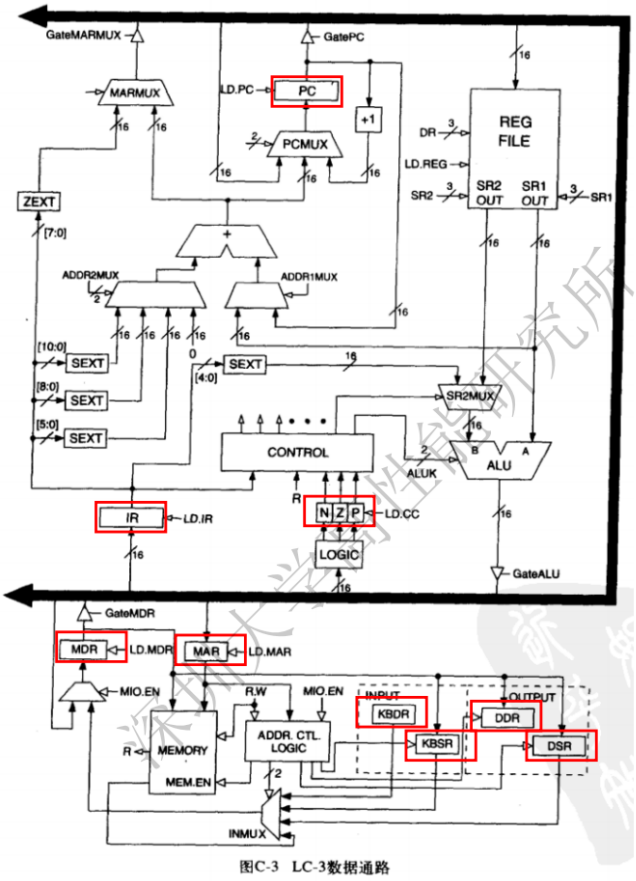
图 25 lea 指令状态转移测试代码以及测试结果

6. LC3 系统-数据通路模块设计

- 学习设计数据通路的前置 Chisel 语法。
- 掌握 LC3 数据通路的关键设计。

6.1. 数据通路模块介绍

在 LC3 的数据通路中，有很多寄存器如 PC、IR、MAR、MDR 等，还有关键线网如经过 IR 寄存器译码出来的立即数等。这些寄存器和线组成基本的数据通路



图C-3 LC-3数据通路

图 26 LC-3 数据通路

从上图数据通路提取关键寄存器和线如下表所示：

表 3 数据通路寄存器

寄存器	描述	寄存器	描述
PC	程序计数器	IR	指令寄存器
MAR	地址寄存器	MDR	数据寄存器
KBDR	键盘输入数据寄存器	KBSR	键盘状态寄存器
DDR	显示器输出数据寄存器	DSR	显示器状态
N Z P	负 零 正 寄存器		

关键线定义如下表：

表 4 关键线

线	描述
ADDR1MUX	ADDR1MUX 选择器输出
ADDR2MUX	ADDR2MUX 选择器输出
PCMUX	PCMUX 选择器输出
DRMUX	DRMUX 选择器输出
SR1MUX	SR1MUX 选择器输出

SR2MUX	SR2MUX 选择器输出
VectorMUX	VectorMUX 选择器输出
PSRMUX	PSRMUX 选择器输出
GATEOUT	总线输出端
addrOut	ADDR1MUX 与 ADDR2MUX 相加输出
aluOut	ALU 输出
r1Data	寄存器堆读数据端口 1
r2Data	寄存器堆读数据端口 2
offset5	IR 寄存器低 5 位作符号扩展成 16 位
offset6	IR 寄存器低 6 位作符号扩展成 16 位
offset9	IR 寄存器低 9 位作符号扩展成 16 位
offset11	IR 寄存器低 11 位作符号扩展成 16 位
offset8	IR 寄存器低 8 位作符号扩展成 16 位

6.2. 完成选择器的连接

在 Datapath.scala 中完成选择器 DRMUX、SR1MUX、SR2MUX、MARMUX 的连接。连接方式参考数据通路控制信号，此处举例 SR1MUX，是一个三选一选择器，输入的值为 IR[11:9]，选择的输出为 IR[11:9]、IR[8:6]以及 SP，对于其他选择器的构造可以参考下图所示：

```
DRMUX := MuxLookup(SIG.DR_MUX, IR(11,9), Seq(
  0.U -> IR(11,9),
  1.U -> R7,
  2.U -> SP
))

SR1MUX := MuxLookup(SIG.SR1_MUX, IR(11,9), Seq(
  0.U -> IR(11,9),
  1.U -> IR(8,6),
  2.U -> SP
))

SR2MUX := Mux(IR(5), offset5, regfile.io.r2Data)

SPMUX := MuxLookup(SIG.SP_MUX, regfile.io.r1Data+1.U, Seq(
  0.U -> (regfile.io.r1Data+1.U),
  1.U -> (regfile.io.r1Data-1.U),
  2.U -> SP, // TODO: Supervisor StackPointer
  3.U -> SP  // TODO: User StackPointer
))

MARMUX := Mux(SIG.MAR_MUX, addrOut, offset8)
```

图 27 选择器实现代码

```
DRMUX/2:        11.9            ;destination IR[11:9]
                R7              ;destination R7
                SP              ;destination R6
SR1MUX/2:       11.9            ;source IR[11:9]
                8.6             ;source IR[8:6]
                SP              ;source R6
ADDR1MUX/1:     PC,BaseR
ADDR2MUX/2:     ZERO            ;select the value zero
                offset6         ;select SEXT[IR[5:0]]
                PCoffset9       ;select SEXT[IR[8:0]]
                PCoffset11      ;select SEXT[IR[10:0]]
SPMUX/2:        SP+1            ;select stack pointer+1
                SP-1            ;select stack pointer-1
                Saved SSP       ;select saved Supervisor Stack Pointer
                Saved USP       ;select saved User Stack Pointer
MARMUX/1:       7.0             ;select ZEXT[IR[7:0]]
                ADDER           ;select output of address adder
```

图 28 数据通路控制信号

6.3. 实例化寄存器堆并连接端口

在 Datapath.scala 中完成选择器例化寄存器堆，并连接端口，此处需要将 regfile 的每一个端口都进行连接，连接的端口参考 LC-3 完整数据通路

```
/************ Regfile Interface ***************/
// 实验五-任务二：寄存器堆例化与端口连接
val regfile  = Module(new Regfile)
regfile.io.wen :=SIG.LD_REG
regfile.io.wAddr :=DRMUX
regfile.io.r1Addr:=SR1MUX
regfile.io.r2Addr:=IR(2,0)
regfile.io.wData:=GATEOUT
r1Data:=regfile.io.r1Data
r2Data:=regfile.io.r2Data

val dstData = WireInit(regfile.io.wData)
```

图 29 实例化寄存器堆

6.4. 完成总线连接

LC3 数据通路中只有一根总线，总线有多个输入端和多个输出端。在源码中定义 GATEOUT 线作为总线输出，其可以连接到多个线或寄存器如图中的 MAR, IR 数据均来自总线输出，因此可以接这些寄存器和线与 GATEOUT 相连。

在 Datapath.scala 中完成总线连接，连接的代码如下所示

```
// 实验五-任务三：连接总线
GATEOUT := PC
when(SIG.GATE_PC){ GATEOUT := PC }
when(SIG.GATE_MDR){ GATEOUT := MDR }
when(SIG.GATE_ALU){ GATEOUT := aluOut }
when(SIG.GATE_MARMUX){ GATEOUT := MARMUX}
when(SIG.GATE_VECTOR){ GATEOUT := Cat(1.U(8.W), 0.U) }
when(SIG.GATE_PC1){ GATEOUT := PC - 1.U }
when(SIG.GATE_PSR){ GATEOUT := Cat(Seq(0.U(13.W), PSRMUX)) }
when(SIG.GATE_SP){ GATEOUT := SPMUX }
```

图 30 连接总线

（4）测试模块功能，使用指令 mill chisel_lc3.test.testOnly example.LookupTest，可以看到答案正确

```
lc3@lc3-virtual-machine:~/Frontend/chisel_lc3$ mill chisel_lc3.test.testOnly example.LookupTest
[71/71] chisel_lc3.test.testOnly
LookupTest:
select:UInt<2>(0), result:UInt<4>(1)
select:UInt<2>(1), result:UInt<4>(2)
select:UInt<2>(2), result:UInt<4>(4)
select:UInt<2>(3), result:UInt<4>(8)
- should test lookup
```

图 31 测试数据通路

7. LC3 系统-存储器模块设计

- 掌握 LC3 系统中存储器的功能和原理
- 掌握存储器仿真时需要初始化的原因
- 了解存储器仿真时初始化的解决方法

7.1. 存储器模块介绍

存储器是 LC3 系统的核心组件之一，用于存储程序指令和数据。它为 CPU 提供了数据来源和指令执行结果存储的地方，是程序运行的基础。由于存储器初始状态为全零，无法直接运行程序。因此，需要将程序代码和数据写入存储器指定位置，才能使 LC3 系统正常运行。

在仿真用存储器模块的实现原理上，采用软件模型代替硬件模块以便于初始化。其软硬件模型如下图所示：

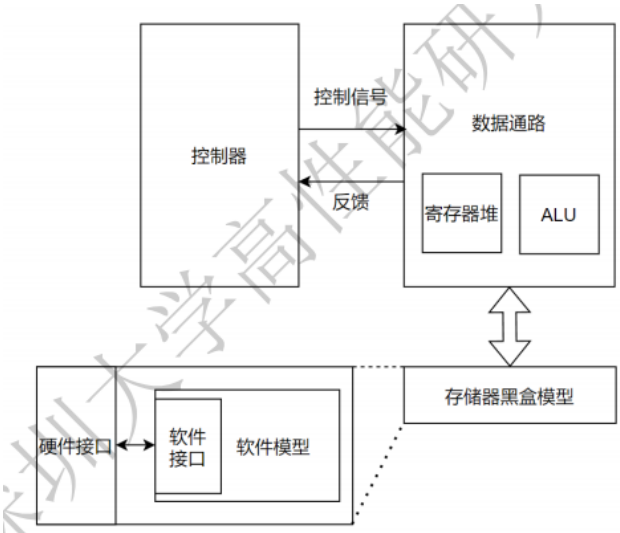


图 32 软硬件模型

DPI-C 可以使用 C 语言来仿真实现 Verilog 中某个模块的功能，而 BlackBox 可以在 chisel 文件中使用 Verilog 编写的模块。因此，在仿真中，存储器模块的实现如下图所示，使用 C 来模拟实现一个存储器，这样的话就可以在 LC3 仿真开始前，用 C 对齐进行初始化，然后使用 DPI-C，把 C 实现的存储器模块用 Verilog 包裹起来，再使用 BlackBox 将这个 Verilog 模块导入到 Chisel 中使用。

（1）定义了一个 C 语言静态的全局变量 ram，ram 本质上是一个一维数组，其中的每个元素是一个 16bits 的无符号整数。这样就可以在 Verilator 的顶层仿真文件中，调用 init_ram 函数，确保在 LC3 仿真正式开始前，存储器中已经初始化完毕，已经写入了需要运行的程序。

```
// src/test/csrc/ram.cpp

#define RAMSIZE 65536          // 存储器大小

typedef uint16_t paddr_t;      // 存储器宽度为16

static paddr_t ram[RAMSIZE];   // 定义存储器数组

// ...

extern "C" void ram_helper(paddr_t rIdx, paddr_t *rdata, paddr_t wIdx, paddr_t wdata, uint8_t wen) {
    int rIdxReg = rIdx;
    *rdata = ram[rIdxReg];          // 存储器读
    if (wen) ram[wIdx] = wdata;     // 存储器写
    //printf("[debug] rIdx=%4x, *rdata=%4x, wIdx=%4x, *wdata=%4x, wen=%x\n", rIdx,  *rdata, wIdx, wdata, wen);
}
```

图 33 存储器模块

（2）在 Chisel 中定义软件黑盒接口，新建一个类 RAMHelper 继承 BlackBox, 其中 clk 为时钟信号，剩下五个信号对应硬件前五个端口，硬件软口中的 R 和 mio_en 则用内部硬件逻辑描述。因此用这五个信号即可完成存储器的读写功能

```
// src/main/scala/LC3/Memory.scala
...
class RAMHelper() extends BlackBox {
  val io = IO(new Bundle {
    val clk = Input(Clock())
    val rIdx = Input(UInt(16.W))
    val rdata = Output(UInt(16.W))
    val wIdx = Input(UInt(16.W))
    val wdata = Input(UInt(16.W))
    val wen = Input(Bool())
  })
}
```

图 34 存储器读写功能

（3）最后，编写黑盒模型将上述硬件接口、软件接口连起来，代码如下：

```
// src/main/scala/LC3/Memory.scala
val mem = Module(new RAMHelper())
mem.io.clk := clock
mem.io.rIdx := io.raddr
io.rdata := mem.io.rdata
mem.io.wIdx := io.waddr
mem.io.wdata := io.wdata
mem.io.wen := io.wen
```

图 35 软硬件接口连接

8. LC3 系统-UART 介绍

- 掌握 LC3 输入输出机制
- 了解 UART 通信协议

8.1. IOMap 模块介绍

（1）LC3 系统输入输出机制

LC-3 系统通过四个特殊寄存器实现输入输出功能：

- KBDR (键盘数据寄存器)：存储从键盘输入的数据。
- KBSR (键盘状态寄存器)：指示是否有新的键盘输入。
- DDR (显示数据寄存器)：存储要显示到屏幕上的数据。
- DSR (显示状态寄存器)：指示是否有数据需要显示。

这些寄存器被映射到内存中的特定地址，使得可以通过内存访问指令来读写这些寄存器。尽管这些寄存器被配置为 2 字节宽，但实际上只有 1 字节的数据是有效的，其余位是冗余的。

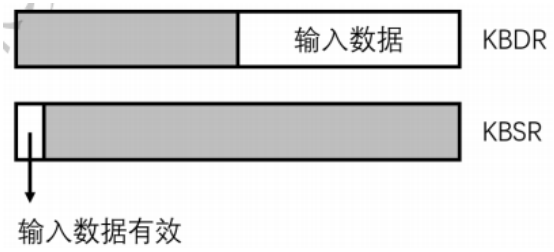


图 36 键盘输入寄存器

（2）UART 通信协议简述

UART 作为异步串行通信协议的一种，工作原理是将传输数据的每个二进制位一位接一位地传输。在 UART 通信协议中信号线上的状态为高电平时代表‘1’,信号线上的状态为低电平时代表‘0’。比如使用 UART 通信协议进行一个字节数据的传输时就是在信号线上产生八个高低电平的组合

UART 的数据传输格式如图所示。其中各位的意义如下

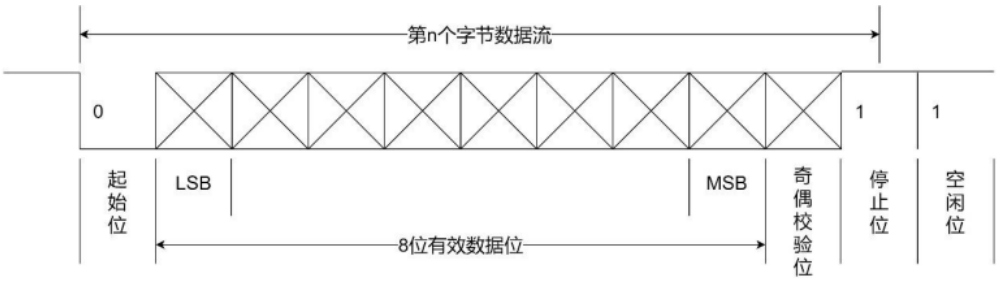


图 37 UART 传输格式

- 空闲位：总线空闲时为高电平（1）。
- 起始位：通信开始时发送低电平（0）。
- 数据位：传输实际数据，通常是 8 位。
- 奇偶校验位：用于校验数据正确性。
- 停止位：表示数据传输结束，可以是 1 位、1.5 位或 2 位高电平。

（3）UART 实现以及与 LC3 输入输出机制的连接

- 输入：当 LC-3 读取内存且地址是 KBDR 的内存映射地址时，会从 KBDR 读取输入数据。
- 输出：当 LC-3 写入内存且地址是 DDR 的内存映射地址时，会将数据写入 DDR 以输出显示。
- 特殊寄存器地址：

表 5 特殊输入寄存器

寄存器名	内存映射地址
KBSR	0xfe00
KDBR	0xfe02
DSR	0xfe04
DDR	0xfe06

8.2. 编写 IOMap 模块

（1）采用条件语句，在 io_io_en 为真的条件下，根据提供的内存地址 io_mar 和读写标志 io_r_w 来设置相应的输入输出信号，从而实现对外部设备的读写访问。

```
// Need write
// 特殊寄存器的内存映射地址
val KBSR_ADDR = 0xFE00.U
val KBDR_ADDR = 0xFE02.U
val DSR_ADDR  = 0xFE04.U
val DDR_ADDR  = 0xFE06.U

// 初始化输出信号
io.r_kbsr := false.B
io.r_kbdr := false.B
io.r_dsr  := false.B
io.r_mem  := false.B
io.w_kbsr := false.B
io.w_dsr  := false.B
io.w_ddr  := false.B

// 当mio_en为true时，根据r_w的值进行读或写操作
when(io.mio_en) {
  when(io.r_w) { // 写操作
    when(io.mar === KBSR_ADDR) {
      io.w_kbsr := true.B
    }
    when(io.mar === DSR_ADDR) {
      io.w_dsr := true.B
    }
    when(io.mar === DDR_ADDR) {
      io.w_ddr := true.B
    }
  } .otherwise { // 读操作
    when(io.mar === KBDR_ADDR) {
      io.r_kbdr := true.B
    }
    when(io.mar === KBSR_ADDR) {
      io.r_kbsr := true.B
    }
    when(io.mar === DSR_ADDR) {
      io.r_dsr := true.B
    }
    when(io.mar < 0xfe00.U) {
      io.r_mem := true.B
    }
  }
}
```

图 38 IOMap 模块

（4）测试 IOMap 模块功能，使用指令 mill chisel_lc3.test.testOnly LC3.IOMaptest，可以看到通过了测试

```
lc3@lc3-virtual-machine:~/Frontend/chisel_lc3$ mill chisel_lc3.test.testOnly LC3.IOMaptest
[71/71] chisel_lc3.test.testOnly
IOMaptest:
DataPath
- should test IOMap
```

图 39 测试 IOMap 模块

9. LC3 系统整体仿真

- 掌握 Boot 模块的工作原理
- 掌握 LC3 顶层 Top 模块的设计
- 能够对 LC3 系统进行整体仿真

9.1. Boot 模块介绍

Boot 模块在整个 LC3 系统中负责启动时的程序动态加载功能，即告诉 LC3 系统应该执行什么程序。在 LC3 真正开始运行前，对内存进行初始化，将需要运行的程序写入内存，并且告诉 LC3 从哪个地址开始执行程序

表 6 接口及其作用

接口名字	作用
uartRx	从 UART 模块接收需要运行的程序指令
work	告诉 LC3 系统初始化是否完成
initPC	告诉 DataPath 程序的起始地址
initMem	向内存中写入需要运行的程序

9.2. Top 模块介绍

Top 模块包括了 LC3 所有相关模块的顶层模块，负责连接其中的各大主要模块，Top 模块的接

口非常的简单，只有两根 1bit 的信号线。因为 UART 协议是 LC3 接收输入输出的唯一手段，因此 Top 的接口只需要一根接收 UART 信号的线和一根传输 UART 信号的线即可，如下图为 Top 模块的连接：

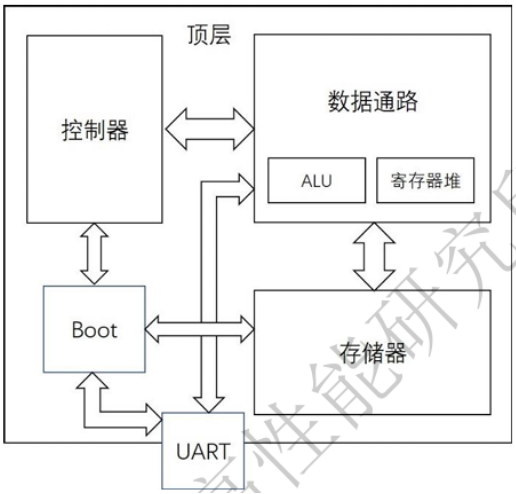


图 40 Top 模块

9.3. 整体仿真

（1）在 chisel_lc3 目录下运行"make emu"命令，得到如下输入，此时默认执行的是 dummy.asm 汇编文件，汇编文件内容如下所示，可以看到是在输出“Hello!”语句，运行指令后在终端成功输出

```
lc3@lc3-virtual-machine:~/Frontend/chisel_lc3$ make emu TRACE=-t
./build/emu -i ./image/dummy.obj -t
The image is ./image/dummy.obj
start addr = 3000
start addr = 400
start addr = 430
start addr = 450
Read ./image/dummy.obj content to uart_buffer
File size: 20B
Get UART output: 
fullValid: 3000
fullValid: e001
fullValid: f022
fullValid: 0048
fullValid: 0065
fullValid: 006c
fullValid: 006c
fullValid: 006f
fullValid: 0021
fullValid: 0000
Mem init finished, LC3 start work
Get UART output: H
Get UART output: e
Get UART output: l
Get UART output: l
Get UART output: o
Get UART output: !
```

```
1   ;.ORIG x3000
2   ;AND R0,R0,X0
3   ;TRAP x20
4   ;trap x21
5   ;ADD R1,R1,R0
6   ;TRAP x20
7   ;ADD R2,R2,R0
8   ;.END
9
10  .ORIG x3000
11  LEA R0, hello
12  PUTS
13  hello .STRINGZ "Hello!"
14  .ENDS
```

图 41 整体仿真测试

（2）观察仿真波形，分析输出结果：执行"make emu TRACE=-t"命令获取波形，使用 gitwave 观察波形如下，可以看到十六进制下的数值 0x48、0x64、0x6c、0x6c、0x6F、0x21，对应 ASCII 码的字符为“H”、“e”、“l”、“l”、“o”、“!”，可以看到符合汇编代码所输出的语句

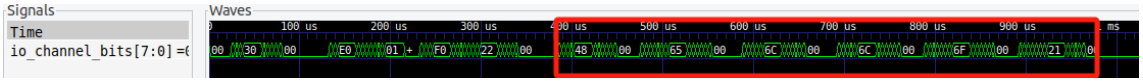


图 42 仿真测试波形

10. FPGA 运行 LC3 系统

- 掌握配置生成用于 FPGA 的可综合代码的方法
- 掌握 vivado 的安装与使用
- 掌握 vivado 中 LC3 项目的搭建
- 掌握烧录 FPGA 的流程

10.1. 在 FPGA 运行 LC3 系统

（1）创建 Vivado 工程，将项目命名为 lc3_fpg，并选择 RTL Project

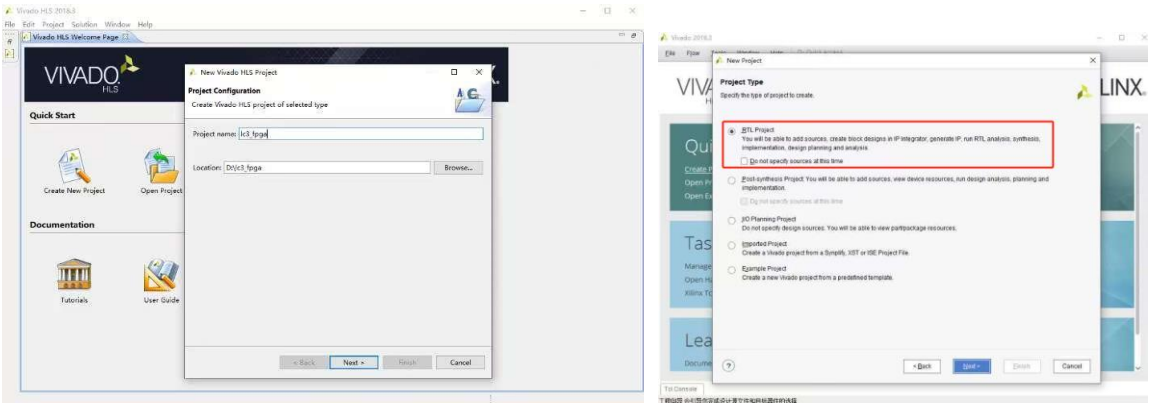


图 43 创建 vivado 项目

（2）选择正确的 FPGA 芯片型号，实验中使用的芯片型号是 xc7a35tfgg484-2，下图分别是选择所需芯片以及建立项目后的界面图

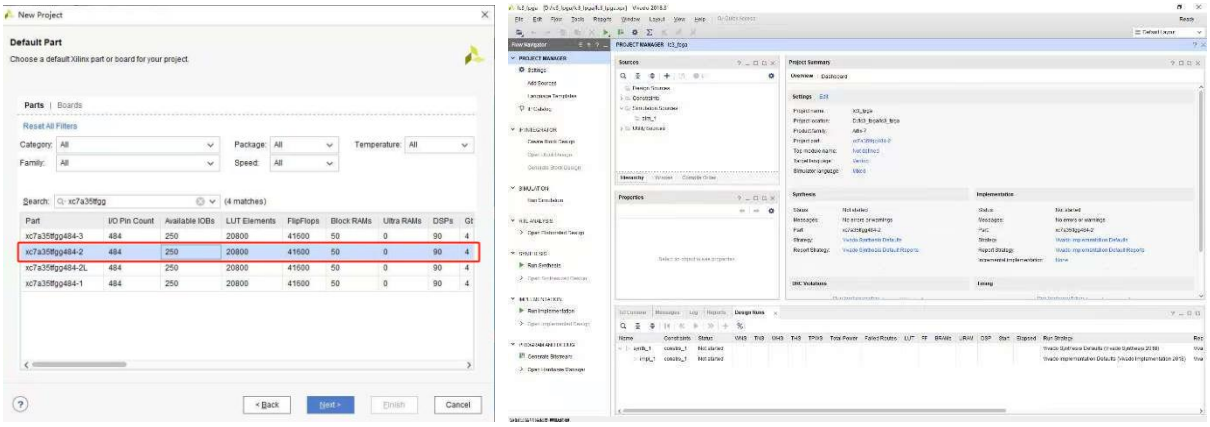


图 44 选择 FPGA 芯片

（3）将 TopMain.v 文件复制到刚建好的 Vivado 项目路径下， 然后在项目中点击 Add Sources 按钮，将 TopMain.v 文件添加到项目中

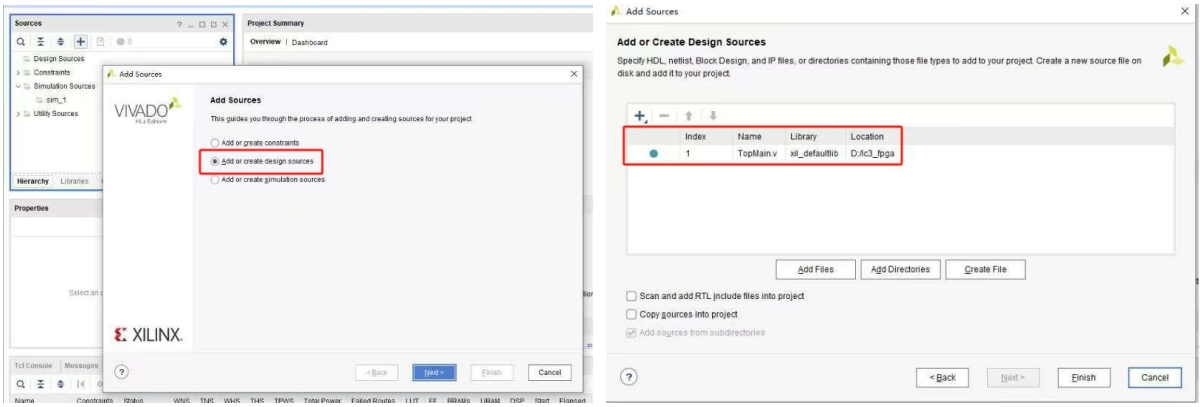


图 45 添加 TopMain.v 文件

（4）添加 IP 核，点击 Flow Navigator 窗口中的 IP Catalog 按钮，在弹出的窗口中搜索 Block Memory，选择 Block Memory Generator，

图 46 添加 ip 核

（5）配置 IP 核，传入用于内存初始化的 init.coe 文件，用于替代 LC3 项目中不可综合的内存初始化逻辑

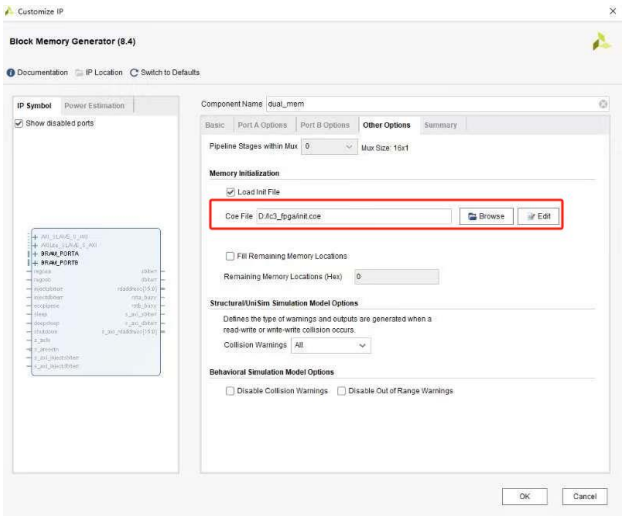


图 47 配准 ip 核

（6）配置完成后点击 OK 按钮，点击 Generate 按钮等待生成 IP 核，编译完成后在 Sources 窗口下点击切换到 IP Sources 标签页，可以看到之前 chisel 中定义的 dual_men 接口与生成的 ram 接口是一致的

（7）要为 vivado 项目添加一个约束文件，点击 Add Sources 按钮，选择 Add or create constraints 进行添加，约束文件的作用是将 LC3 的时钟、复位接口和 Uart 接口与 FPGA 上对应的引脚连接，约束文件内容如下：

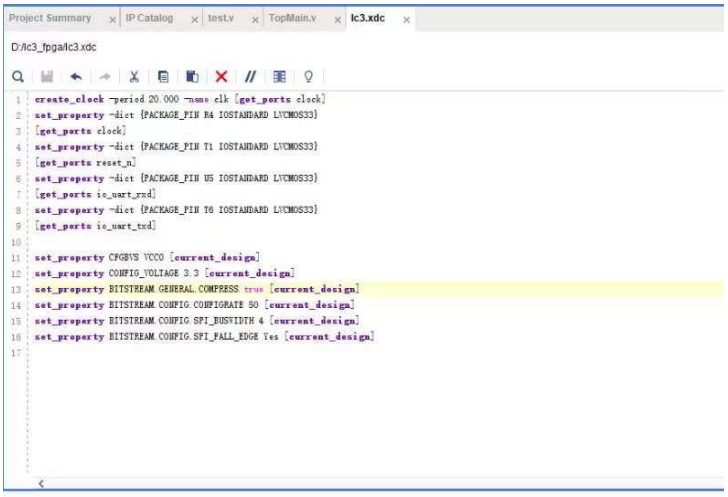


图 48 添加约束文件

（8）在 vivado 中指定一个顶层，将 Top 模块再包一层。选择 Add or create simulation sources 选项进行添加，文件内容如下：

```
`timescale 1ns / 1ps
module test();
reg sys_clk;
reg sys_rst_n;
wire txd; // 连接 UART 接口
initial begin // 给时钟赋初值，reset 信号最开始是有
效的，在 100ns 后 reset 信号撤销
sys_clk = 1'b0;
sys_rst_n = 1'b1;
#100
sys_rst_n = 1'b0;
end
always #10 sys_clk = ~sys_clk; // 设置时钟每 10ns反转一次，则一个时钟周期是 20ns

Top top( // 实例化 LC3 顶层模块
.clock(sys_clk),
.reset(sys_rst_n),
.io_uart_rxd(1'b0),
.io_uart_txd(txd)
);
endmodule
```

图 49 添加顶层

（9）手动修改 TopMain.v 的复位信号，将其取反，具体操作为在 xdc 约束文件中将 reset 引脚的名字改为 reset_n

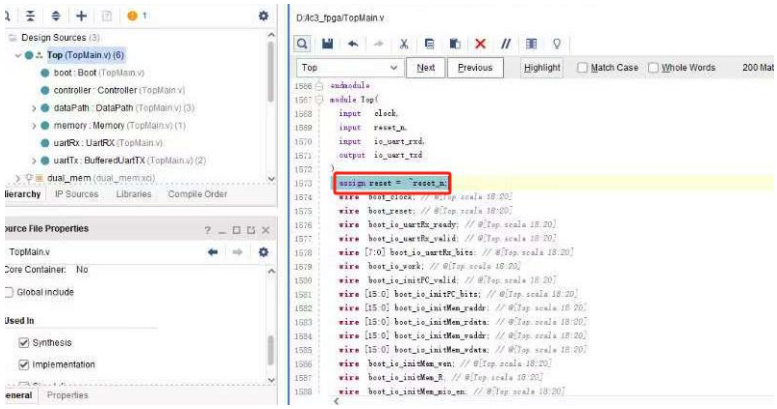


图 50 修改复位信号

（10）将 LC3 烧录到 FPGA 板卡上点击 Flow Navigator 窗口中的 Generate Bitstream 按钮，接着按照如图的格式进行配置，生成 mcs 文件

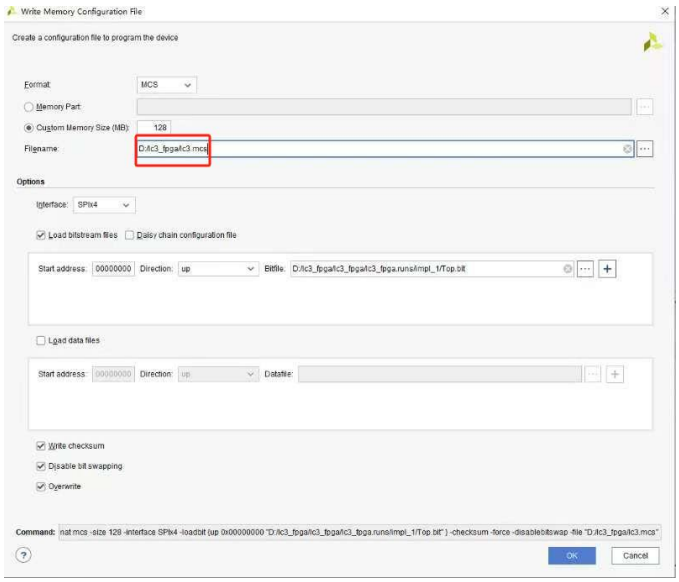


图 51 烧录

（11）连接开发板，在 Flow Navigator 窗口中点击 Open Hardware Manager 按钮，再点击 Hardware 窗口中的 Auto Connect 按钮，进行连接。然后为开发板添加固化 Flash 部件，选择 FPGA 板卡上的 Flash 芯片型号

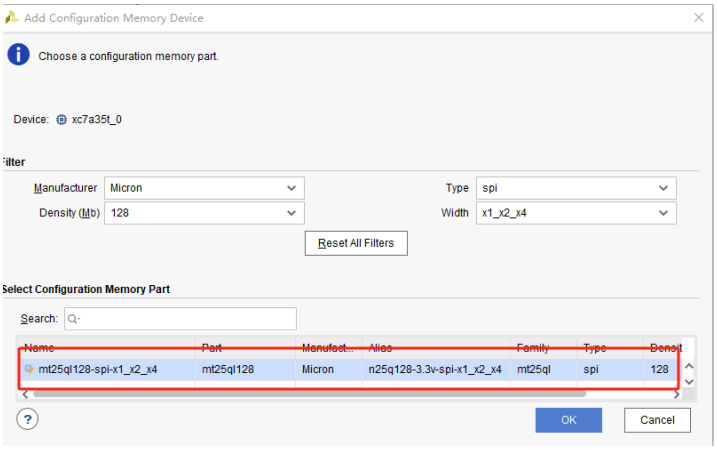


图 52 添加固化 Flash 部件

（12）选择生成的 mcs 文件，以及和 mcs 文件在同一目录下的 prm 文件，配置完成后点击 OK，即可开始烧录

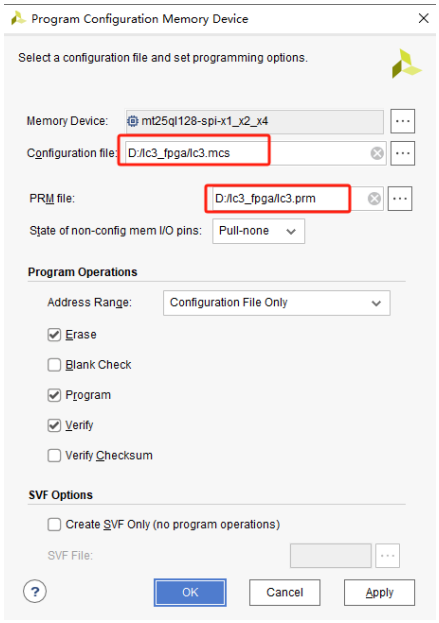


图 53 生成 mcs 文件

10.2. 在 FPGA 运行 obj 程序

打开串口调试助手，在 FPGA 启动的情况下，选中 FPGA 对应的串口，然后配置好对应的波特率等参数，选择发送文件，然后将希望运行的程序的 obj 文件通过串口传输给 FPGA，此处测试了 dummy.obj，成功发送了“Hello!”，以及 nim.obj，实现游玩游戏，结果如下图所示：

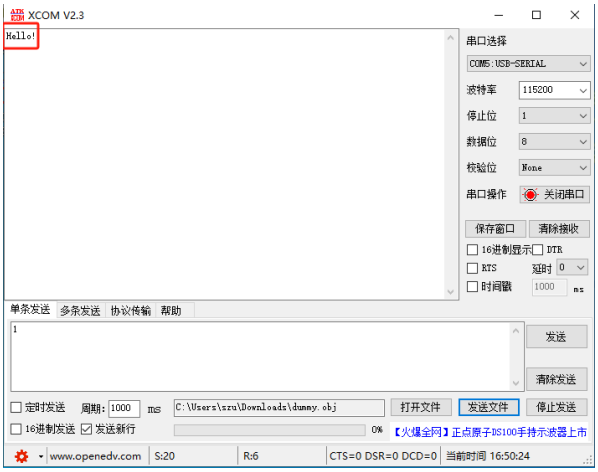


图 54 测试 dummy.obj 文件

图 55 测试 nim.obj 文件

五、测试结果

1. 模块测试

本项目中，对 LC-3 处理器的各个模块进行了详细的测试，以确保每个部分都能正常工作：

（1）ALU 模块

• 对加法、按位或、按位与、按位非以及传递操作数 1 的操作进行了测试。ALU 模块能够正确执行所有指定的操作，并且输出结果与预期一致。

（2）寄存器堆模块测试

• 通过读写操作测试了寄存器堆模块的功能。测试结果表明，寄存器堆能够正确地进行数据的读取和写入，且写入后的数据能在下一个时钟周期被正确读取。

（3）控制器模块测试

• 对状态转移图进行了补充，并测试了 ADD、AND、LEA 指令的状态转移顺序。测试结果符合预期，控制器模块能够正确地根据状态转移图生成控制信号。

（4）数据通路模块测试

• 完成了选择器和寄存器堆的连接，并测试了总线连接。

（5）存储器模块测试

• 存储器模块使用软件模型代替硬件模块以便于初始化。测试结果表明，存储器模块能够正确地存储程序指令和数据，并且能够在仿真开始前完成初始化

（6）UART 模块测试

• 对 UART 通信协议的实现进行了测试，包括数据的发送和接收。测试结果符合预期，UART 模块能够正确地实现串行通信

（7）Boot 模块测试

• Boot 模块负责程序的动态加载和内存初始化。测试结果显示，Boot 模块能够正确地将程序写入内存，并设置正确的起始执行地址

（8）Top 模块测试

• Top 模块能够确保所有模块正确协同工作

2. 整体仿真测试

FPGA 运行测试

• 将 LC-3 系统烧录到 FPGA 板卡上，并成功运行了 dummy.obj 和 nim.obj 程序，分别输出了“Hello!”和实现了游戏功能

六、实验总结

本次实验的目标是使用 Chisel 语言设计 LC-3 处理器，并在仿真环境中验证其功能，最终在 FPGA 硬件上实现运行。实验内容涵盖了从项目背景研究、需求分析、设计方案制定，到具体的实现过程，包括搭建开发环境、模块设计、仿真测试，以及最终的 FPGA 运行。整个实验流程严谨而全面，确保了设计的系统性和实用性。

通过本次课程设计，学习并掌握了 Chisel 语言，这是一种基于 Scala 的硬件描述语言，提供了高级语言的特性，使得硬件设计更加简洁和易于理解。学习了 Verilator 仿真器的使用，以及如何在软件环境中模拟硬件电路的行为。通过对 LC-3 处理器各个模块的设计和测试，加深了对计算机组成原理的理解，特别是在算术逻辑单元、寄存器堆、控制器、数据通路等核心组件的设计和实现上。