# 

# 《计算机组成与系统结构》

# 实践报告

|  |  |
| --- | --- |
| **项目名称：** | **基于riscv-sodor的矩阵乘法加速** |
| **项目时间：** | **2020—2021学年春夏学期** |
| **指导老师：** | **黄科杰** |
| **小组成员：** | **何智鹏 温晨怡 林炬乙** |
| **提交日期：** | **2021年6月** |

目 录

[1 项目设计方案 3](#_Toc75270104)

[1.1 设计目标 3](#_Toc75270105)

[1.2 技术方案 3](#_Toc75270106)

[1.3 项目创新点 4](#_Toc75270107)

[2 原理性验证 4](#_Toc75270108)

[2.1 Chisel环境验证 4](#_Toc75270109)

[2.1.1安装intellij 4](#_Toc75270110)

[2.1.2安装scala支持 5](#_Toc75270111)

[2.1.3安装chisel支持 8](#_Toc75270112)

[2.2 测试方法验证 10](#_Toc75270113)

[2.2.1 测试工具与环境 10](#_Toc75270114)

[2.2.2 测试程序生成 13](#_Toc75270115)

[2.2.3 测试流程 17](#_Toc75270116)

[3 具体设计与实现 20](#_Toc75270117)

[3.1数据格式 20](#_Toc75270118)

[3.2整体框架 20](#_Toc75270119)

[3.2.1 cpath.scala 21](#_Toc75270120)

[3.2.2 dpath.scala 23](#_Toc75270121)

[3.2.3 frontend.scala 25](#_Toc75270122)

[3.3乘法器 26](#_Toc75270123)

[3.3.1 无符号乘法器设计 26](#_Toc75270124)

[3.3.2 有符号乘法器设计 27](#_Toc75270125)

[3.3.3 带乘法功能的ALU设计 29](#_Toc75270126)

[3.4 精简数据通路 30](#_Toc75270127)

[3.4.1 确定所需指令 31](#_Toc75270128)

[3.4.2 精简代码 31](#_Toc75270129)

[4 功能测试与性能评测 31](#_Toc75270130)

[4.1 功能正确性测试 31](#_Toc75270131)

[4.1.1 利用shiftmulonly.riscv文件检查mul-sodor核的正常功能 32](#_Toc75270132)

[4.1.2 利用mulonly.riscv文件检查mul-sodor核的mul功能 33](#_Toc75270133)

[4.2 性能评测 36](#_Toc75270134)

[4.2.1 乘法运算的性能比较 36](#_Toc75270135)

[4.2.2 矩阵乘法的性能比较 37](#_Toc75270136)

[5 项目总结 38](#_Toc75270137)

[5.1 项目成果总结 38](#_Toc75270138)

[5.2 成员分工 38](#_Toc75270139)

[5.3 反馈与建议 39](#_Toc75270140)

[[参考文献] 39](#_Toc75270141)

## 1 项目设计方案

### 1.1 设计目标

### 1.2 技术方案

### 1.3 项目创新点

## 2 原理性验证

这一部分我们将可能详尽地阐述我们搭建环境、跑通流程的过程，并将这一部分单独列出作为**chisel代码编写环境实验与软件模拟测试验证实验的参考文档**。我们保证：根据文档进行实验将轻松地搭建基于sodor项目所需要的各种环境，并了解chisel编写的流程和chisel核的软件模拟测试流程。

希望我们的努力能为《计组》这门课以后的实验教学做出一定的贡献。

### 2.1 Chisel环境验证

#### 2.1.1安装intellij

**1.安装jdk1.8**

下载java se：<https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

解压：tar -xvf jdk-8u161-linux-x64.tar.gz

设置环境变量：

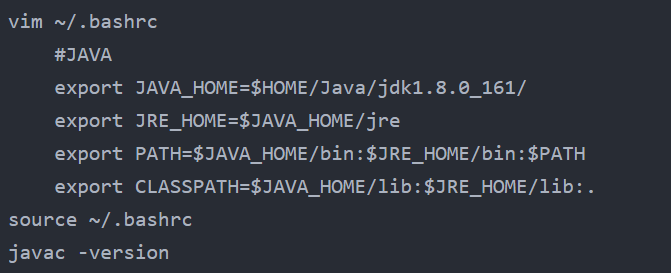


图1 设置环境变量

**2.安装Intellij**

Intellij是一个java编程语言开发的集成环境，chisel是基于scala语言设计的，而scala语言也是在java基础上发展而来的。Intellij支持scala语言的编译，而且该软件功能十分强大，能够在编写chisel代码时提供很多便捷性。

在Intellij官网下载： <https://www.jetbrains.com/idea/download>

|  |  |
| --- | --- |
| 1 | sudo tar -xf ideaIU-2017.3.4-no-jdk.tar.gz |
| 2 | sudo mv idea-IU-173.4548.28 /opt |
| 3 | cd /opt/idea-IU-173.4548.28/bin |
| 4 | ./idea.sh |

**3.申请学生免费授权**

进入申请网站（<https://www.jetbrains.com/zh-cn/community/education/#students>），点击立即申请，并按照要求填写“学校邮箱”和“个人信息”等资源，等待审核过后获得license code。

打开intellij -> configure -> manage license -> Activation code，输入license code。

#### 2.1.2安装scala支持

从GitHub上克隆chisel的教程仓库

|  |  |
| --- | --- |
| 1 | git clone https://github.com/ucb-bar/chisel-tutorial.git |
| 2 | cd chisel-tutorial |

用intellij打开其中任意一个scala代码文件，intellij提示你“未安装scala支持，是否安装”，如下图，请选择安装：

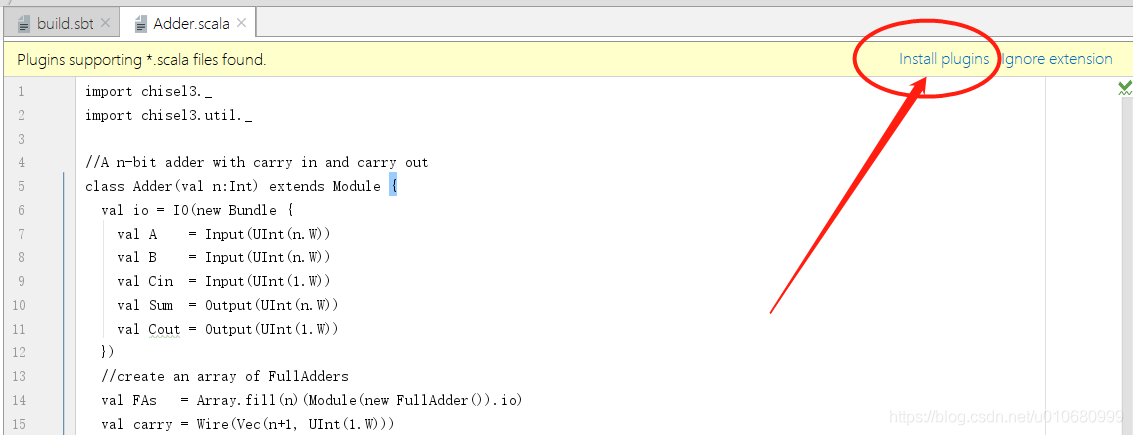


图2 选择安装scala支持

创建新的工程，点击“Create New Project”

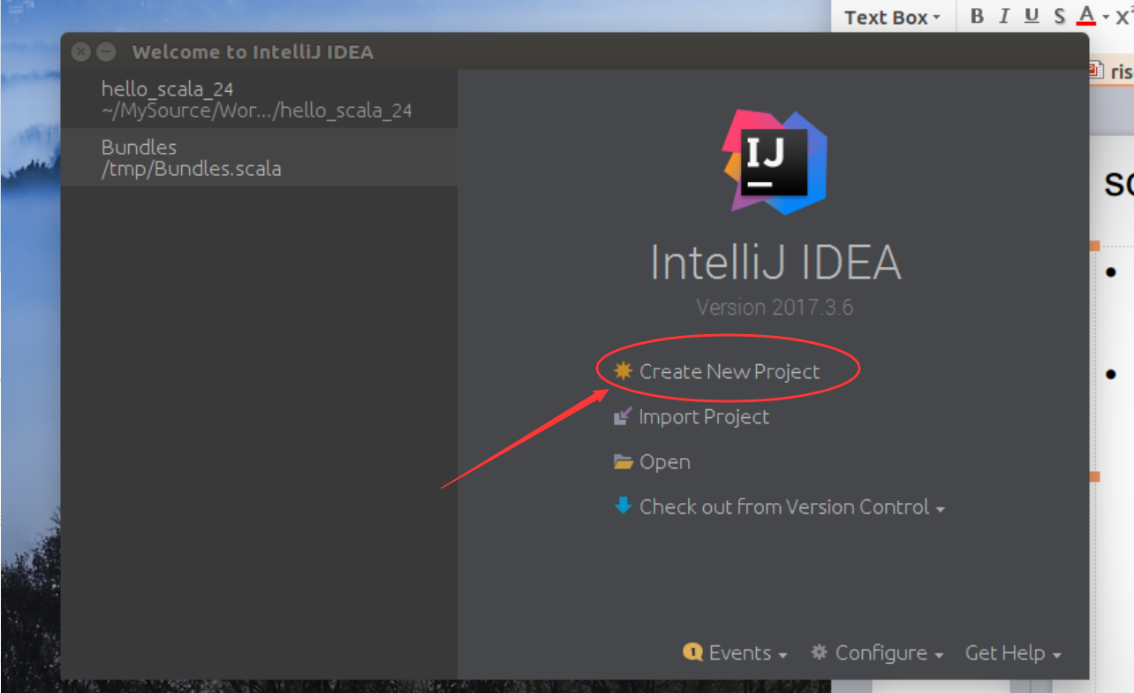


图3 新建工程

先选择scala工程，再选IDEA方式创建。

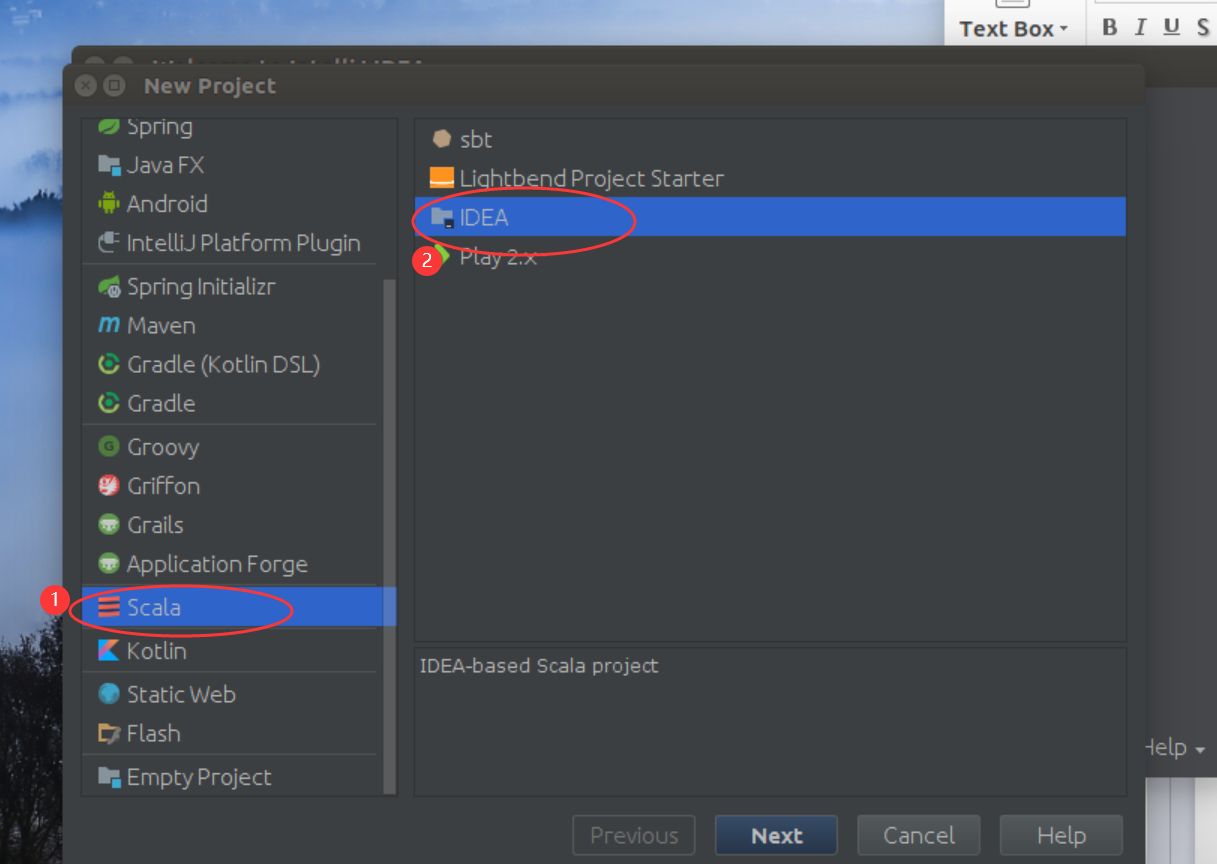


图4 IDEA方式创建

设置工程名、工程地址、JDK版本、Scala SDK版本。

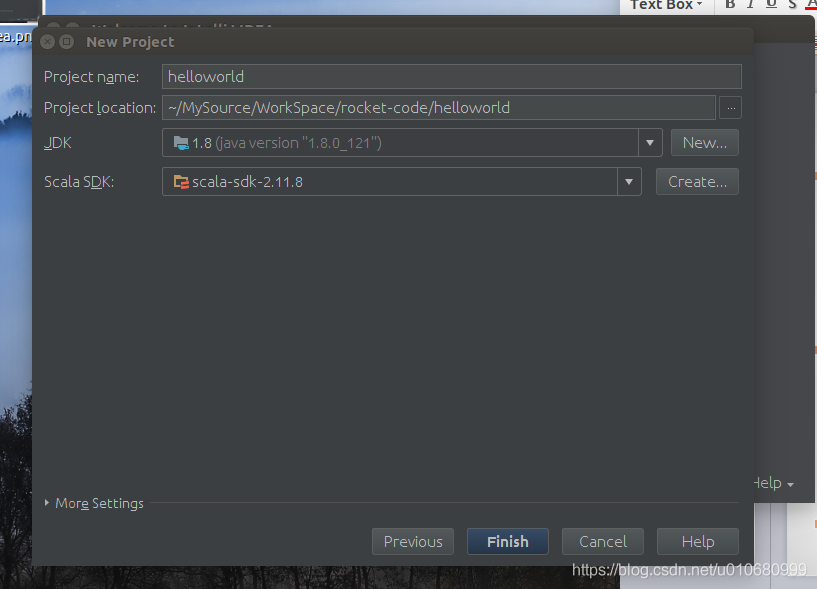


图5 对基本参数进行设置

如果SDK没有安装，选择Create -> download，下载并安装。

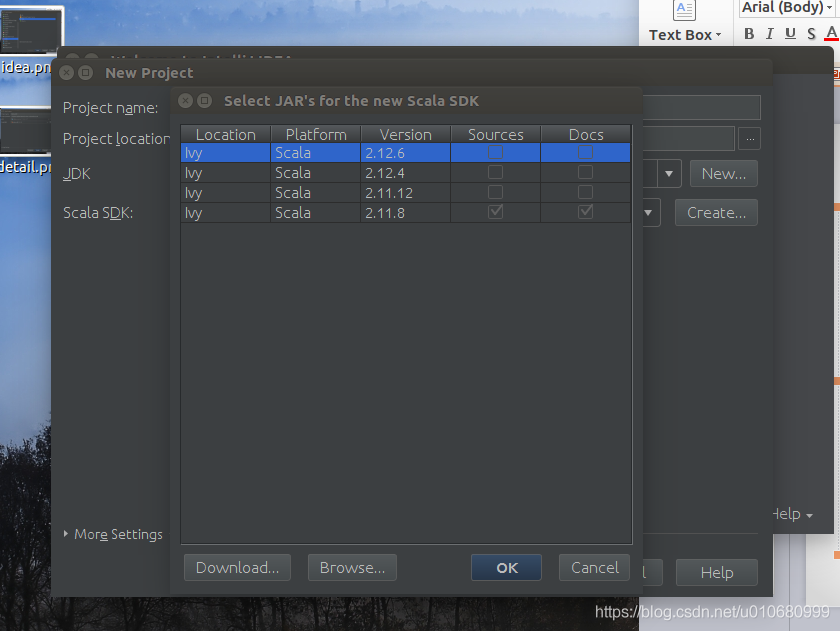


图5 下载安装sdk

点击finish，创建scala工程成功后，右键src->new->scala class，选择为object，命名为“helloworld”。

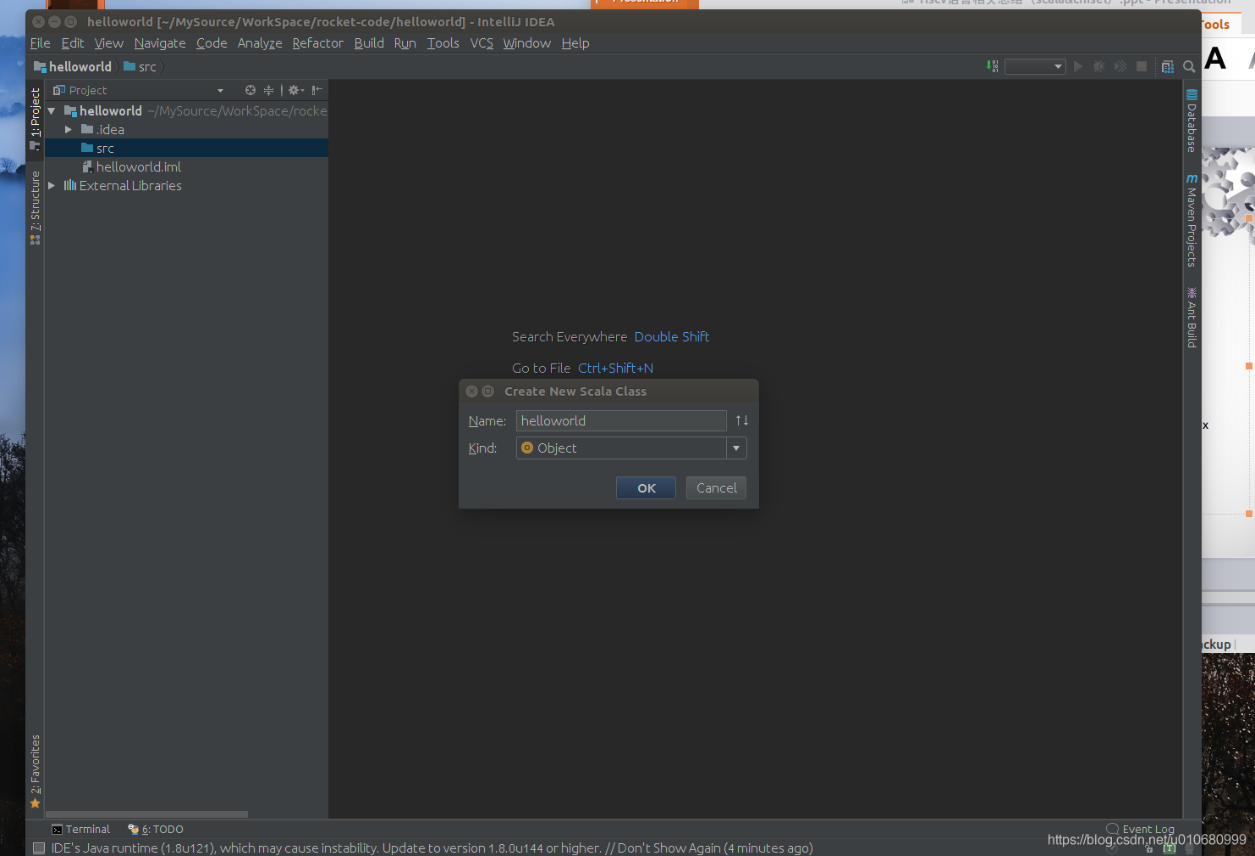


图6 新建scala文件

编写程序，打印“hello world”，并右键run，运行程序。

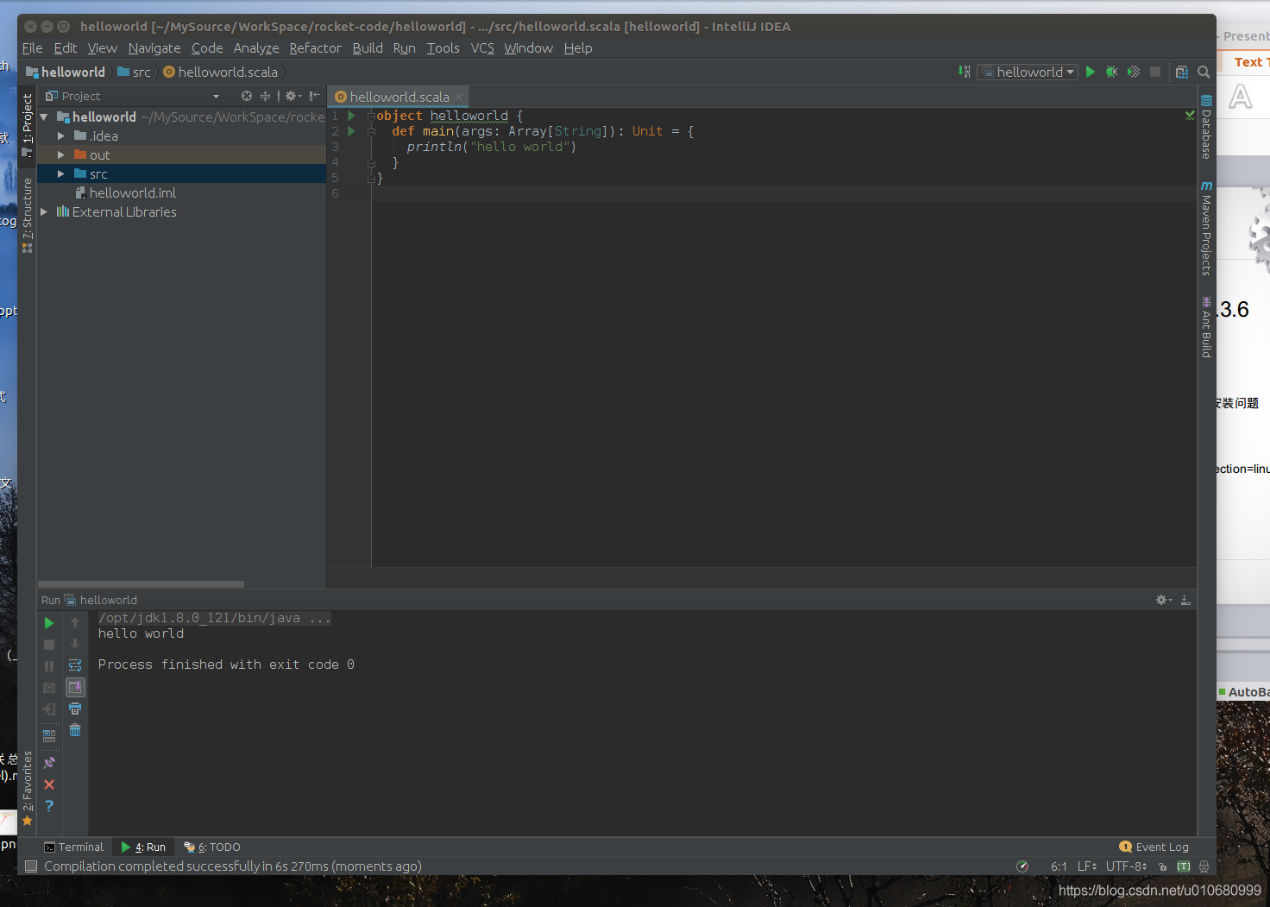


图7 编写并运行helloworld

#### 2.1.3安装chisel支持

按照上一节，创建chisel工程，不过在第二步选择“sbt”，而不是“IDEA”。

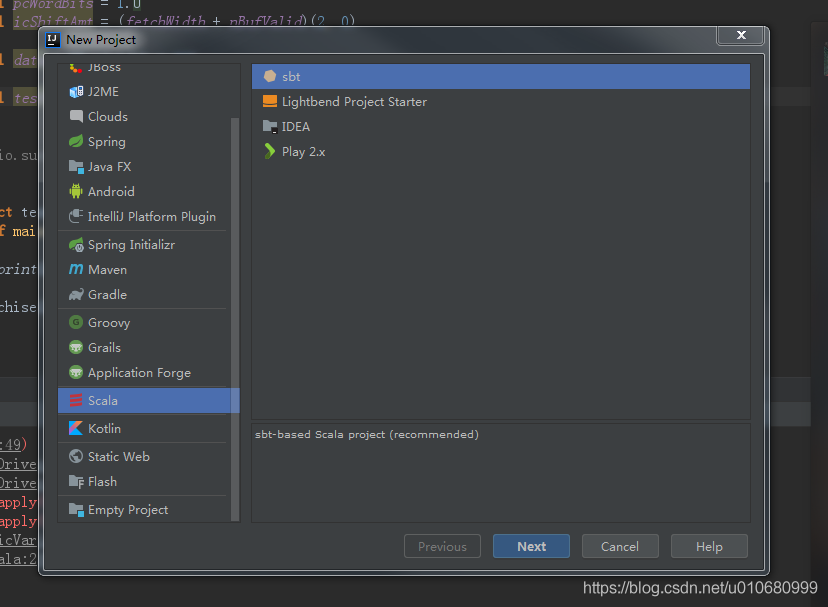


图8 选择以sbt创建

创建chisel工程后，打开build.sbt，并添加下面三行代码，等待库文件更新。

|  |  |
| --- | --- |
| 1 | scalaVersion := "2.11.12" |
| 2 | libraryDependencies += "edu.berkeley.cs" %% "chisel3" % "3.1.2" |
| 3 | libraryDependencies += "edu.berkeley.cs" %% "chisel-iotesters" % "1.2.3" |

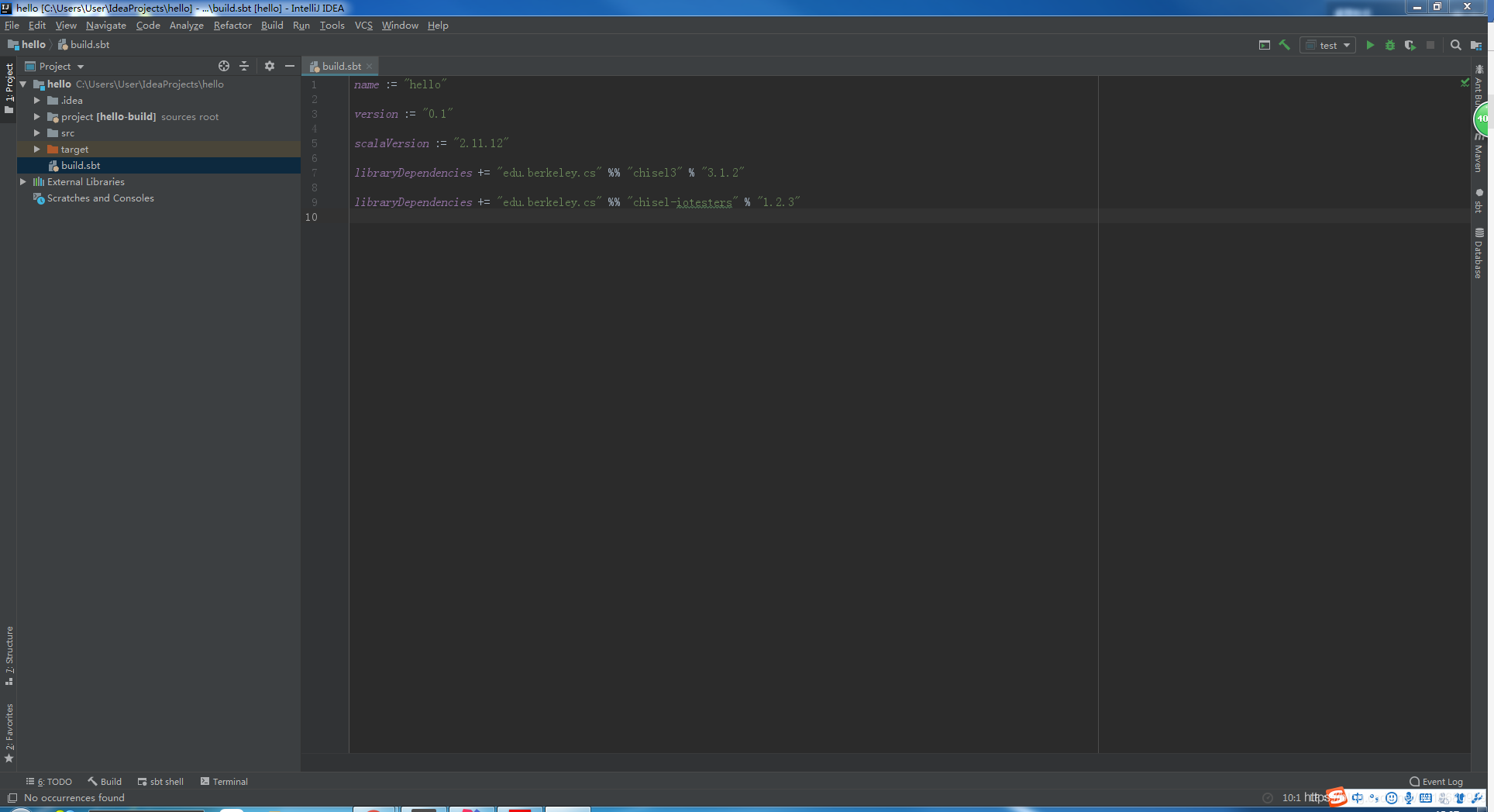


图9 更改build.sbt文件

编写程序，打印“nihao”和“hello”，并右键run，运行程序。

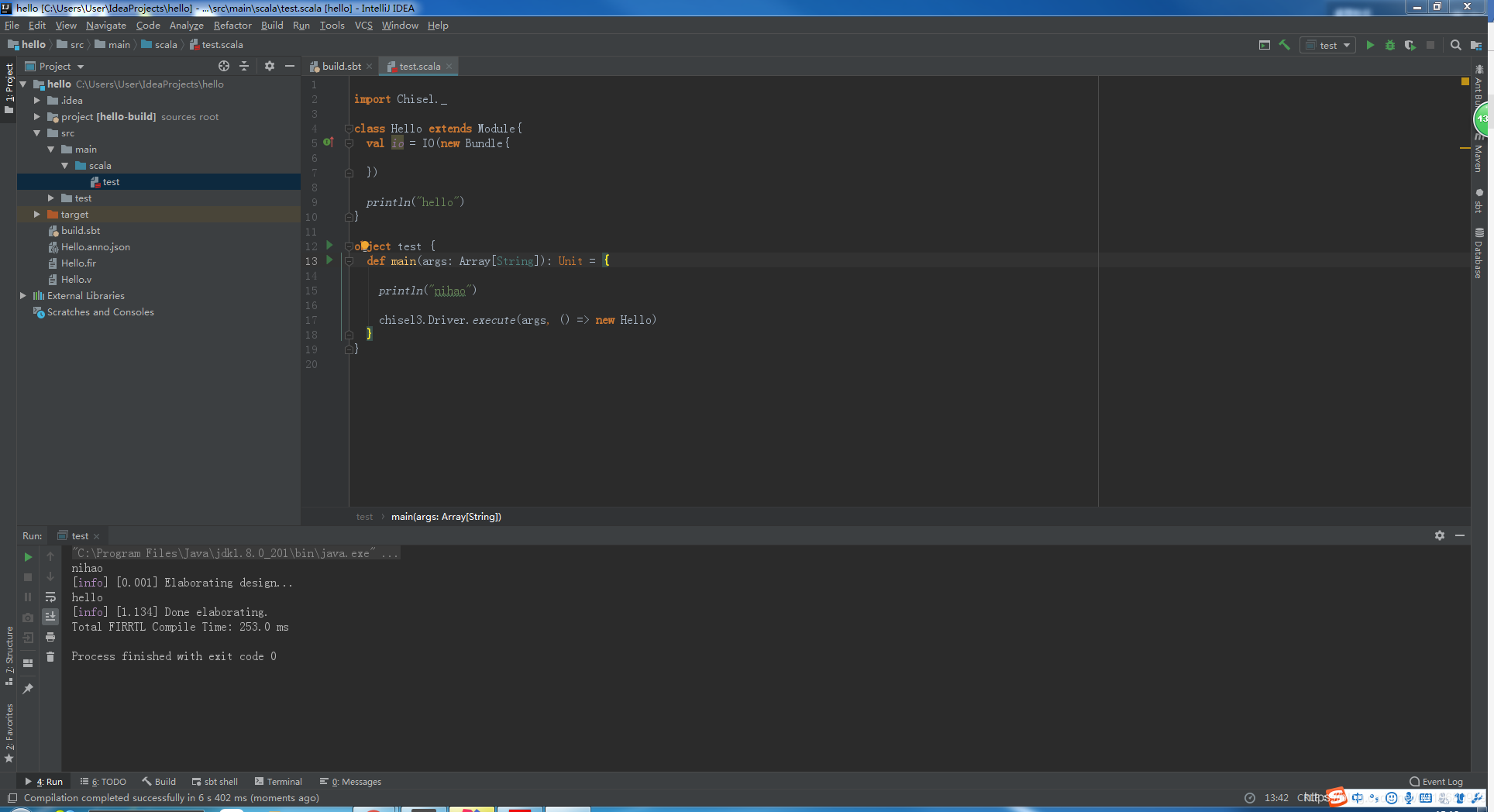


图10 编写hello文件并运行

至此，Chisel编译环境安装测试完毕。

### 2.2 测试方法验证

#### 2.2.1 测试工具与环境

由于**sodor**只支持**rv32i**指令集，所以测试程序必须由riscv32工具进行编译、生成。为了形成自己的benchmark，我们需要搭建相关的环境。

表1 测试环境搭建相关参数

|  |  |
| --- | --- |
| 开发环境 | VMware Ubuntu 16.4 |
| 前置要求 | 在Ubuntu系统中完成了chipyard部署，可以直接使用老师的VMware Ubuntu源（此后称为cosa机）。 |
| 目标工具链 | riscv32-unknown-elf-gcc |
| 目标指令集与链接库 | arch=rv32i, abi=ilp32 |

Chipyard环境中已有下载好的riscv-gnu-toolchain，因此我们无须再从github下载riscv-gnu-toolchain，这可以避免由于链接网页超时导致的一系列问题。但是chipyard的toolchain已经编译成了riscv64工具，我们需要重新对工具链进行编译。

下面详述环境搭建的流程：

**1.为riscv32工具创建文件夹，作为riscv32工具的位置**

我在/home/cosa/code下创建了文件夹rv32i-tool，我的工具链将安放于此（即我的工具链目录为/home/cosa/code/rv32i-tool）。

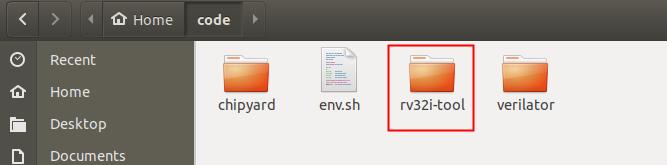


图1 创建rv32i-tool文件夹

**2.转至chipyard中riscv-gnu-toolchain所在的文件夹**

riscv-gnu-toolchain所在目录：**/home/cosa/code/chipyard/toolchains/riscv-tools/riscv-gnu-toolchain**。进入目录后能看到riscv-gcc、riscv-gdb等工具。

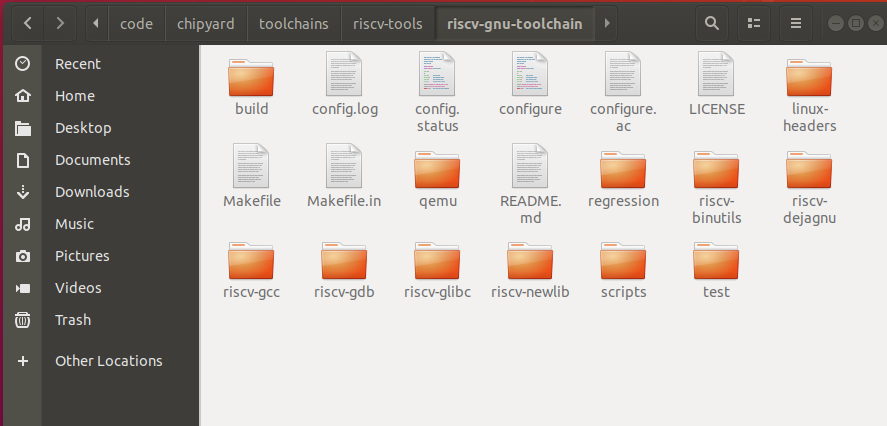


图2 riscv-gnu-toolchain文件夹

若没有配置好的chipyard环境，则可以直接从github下载riscv-gnu-toolchain。建议搭配科学上网的方式。

**3.重新编译riscv-gnu-toolchain**

在riscv-gnu-toolchain所在目录打开terminal（空白处右键-open in terminal），并输入以下指令。

|  |  |
| --- | --- |
| 1 | make clean |
| 2 | ./configure --prefix={the place you put your riscv32 tools} --with-arch=rv32i --with-abi=ilp32 |
| 3 | make |

{the place you put your riscv32 tools}，我用/home/cosa/code/rv32i-tool（步骤1的文件夹目录）代替。在make后等待约1小时，riscv32工具链编译完毕。

在这里特别强调：--with-arch=rv32i不能写成--with-arch=rv32imc，否则你使用riscv32编译出来的测试文件将包含sodor不支持的扩展指令（如mul、div等），该测试文件将无法运行。

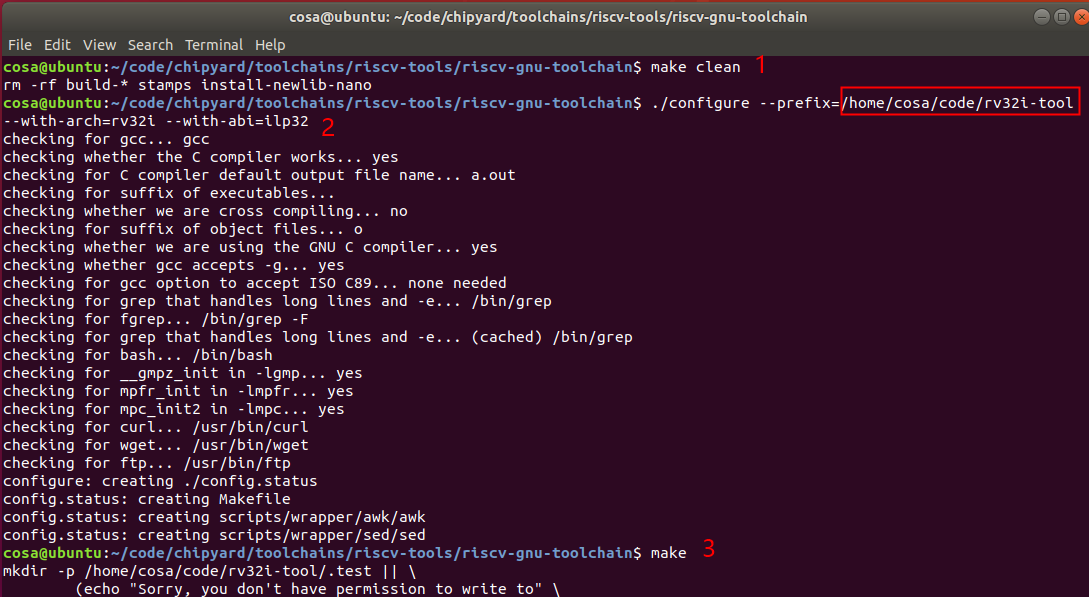


图3 步骤3代码运行效果图

**4.将riscv32工具链的bin添加至环境变量中**

只有将riscv32工具链的bin添加至环境变量中，我们才能使用riscv32-unknown-elf-gcc工具。

在**任意目录**打开terminal，输入：**export PATH="{the place you put your riscv32 tools}/bin:$PATH"**。比如我会输入：export PATH="/home/cosa/code/rv32i-tool/bin:$PATH"。

此时你暂时在这一terminal中添加了riscv32工具的bin的目录至PATH中，若更换terminal或重启虚拟机则这一添加将失效。**若更换terminal或重启虚拟机，想要使用riscv32工具则需要重复步骤4**。

相比于永恒添加riscv32/bin至环境变量中，我更推荐步骤4的暂时添加的做法。因为配置好chipyard自带riscv64工具链，若永恒添加riscv32工具至环境变量可能会导致工具链冲突，产生兼容性问题。

**5.检查riscv32工具链是否正常工作**

在执行了步骤4的terminal中输入：**riscv32-unknown-elf-gcc -v**。

若出现riscv32-unknown-elf-gcc的相关信息，则riscv32工具链成功搭建，并能正常使用。



图4 步骤4、步骤5代码运行效果图

#### 2.2.2 测试程序生成

如果想编写自己的sodor测试程序，确实可以参考chipyard/generators/riscv-sodor/test/custom-bmarks中的内容。但我更推荐直接在riscv-test的架构中略作修改后再编写自己的testbench，因为这样可以更好地利用已经编写好的测试程序源代码、各种计时计数函数、接口函数。

下面详述基于riscv-test生成测试程序的流程：

**1.复制riscv-test文件夹，避免干扰原有文件**

我建议在chipyard文件夹下新建一个空文件夹放置自己的所有代码（如我在chipyard下建立了文件夹wcy\_code放置我的代码），以免我们的代码对chipyard环境产生影响和干扰。

riscv-test的目录为：**/chipyard/toolchains/riscv-tools/riscv-test**。

我将riscv-test文件夹复制到wcy\_code下，并改名为sodor-test。

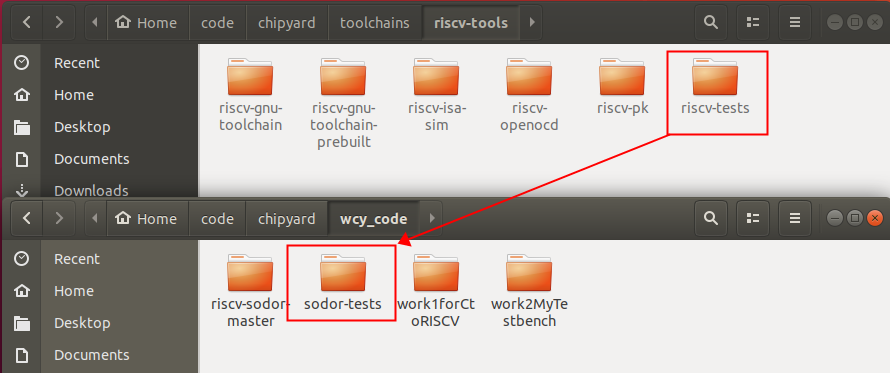


图5 步骤1示意图

**2.参照已有benchmark的源代码编写自己的测试文件**

已有benchmark的源代码位于**/sodor-test/benchmark**中，每一个文件夹都代表着一个测试程序。如multiply文件夹中存放着multiply测试程序生成所需要的所有C源码（部分头文件、函数会存放在common文件夹和../env文件夹中，但那些都是已经编写好的函数和接口，无须自行编写，最好也不要进行更改）。



图6 测试程序源代码

我建议参照multiply文件夹编写自己的测试代码，因为multiply测试程序结构非常简单，易于理解。我将multiply文件夹进行复制，并改名为hello。

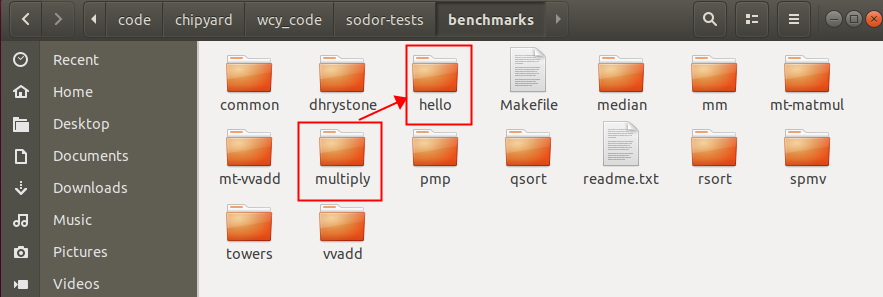


图7 复制multiply文件并更名为hello

进入hello文件夹，删除multiply\_main.c以外的所有文件，并将muliply\_main.c改名为hello.c。并按照图9进行代码修改并保存。

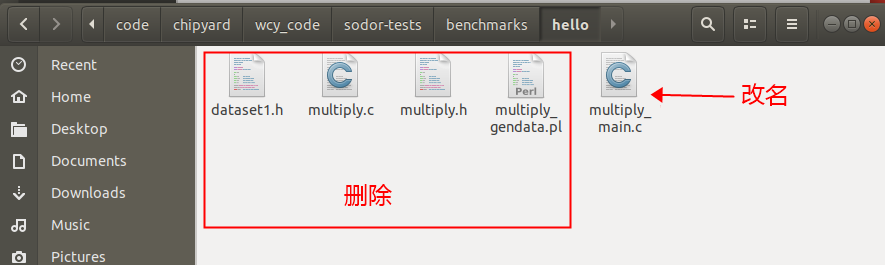


图8 删除不需要的文件

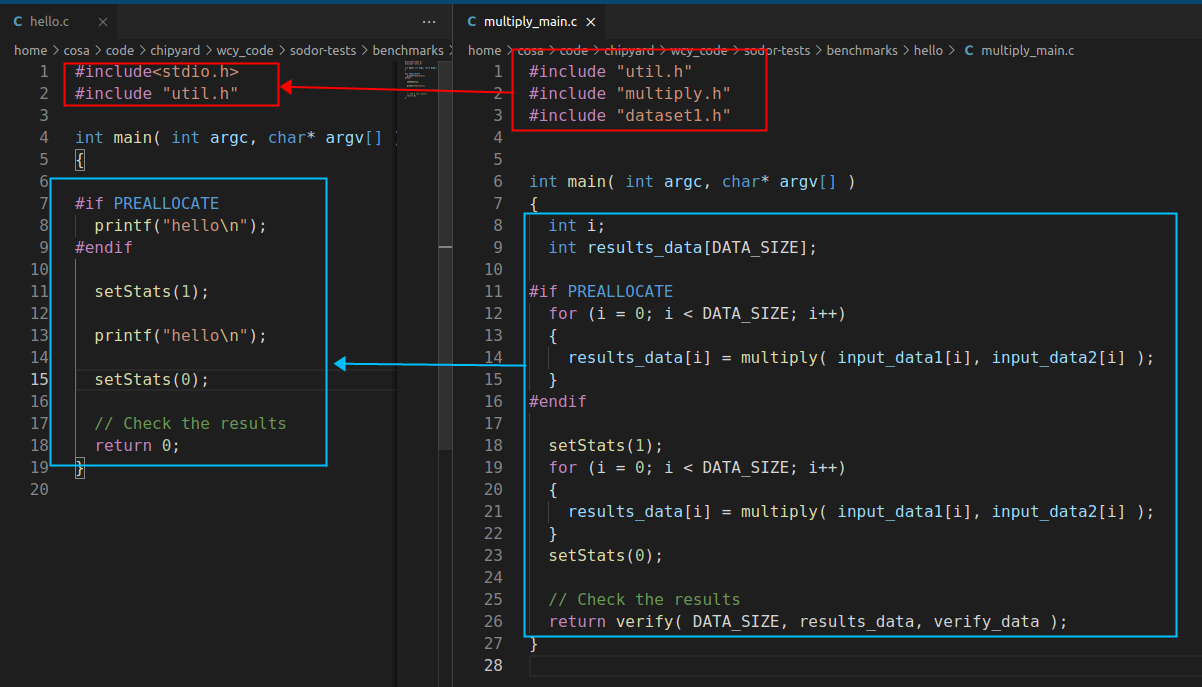
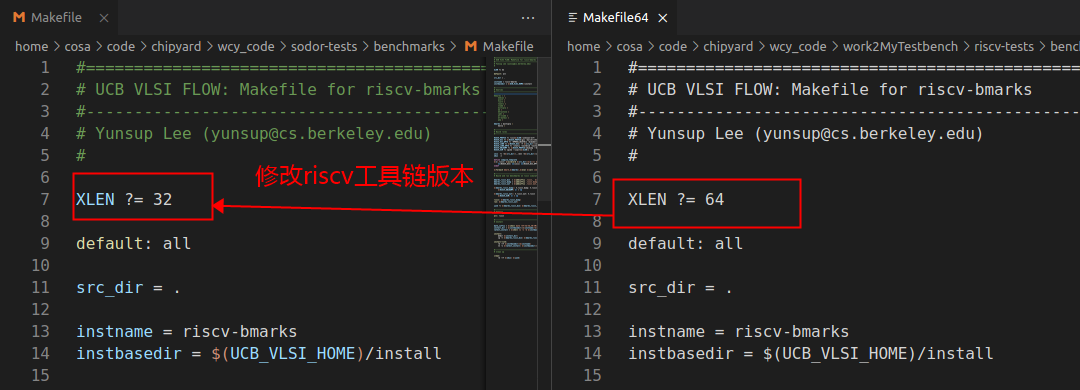


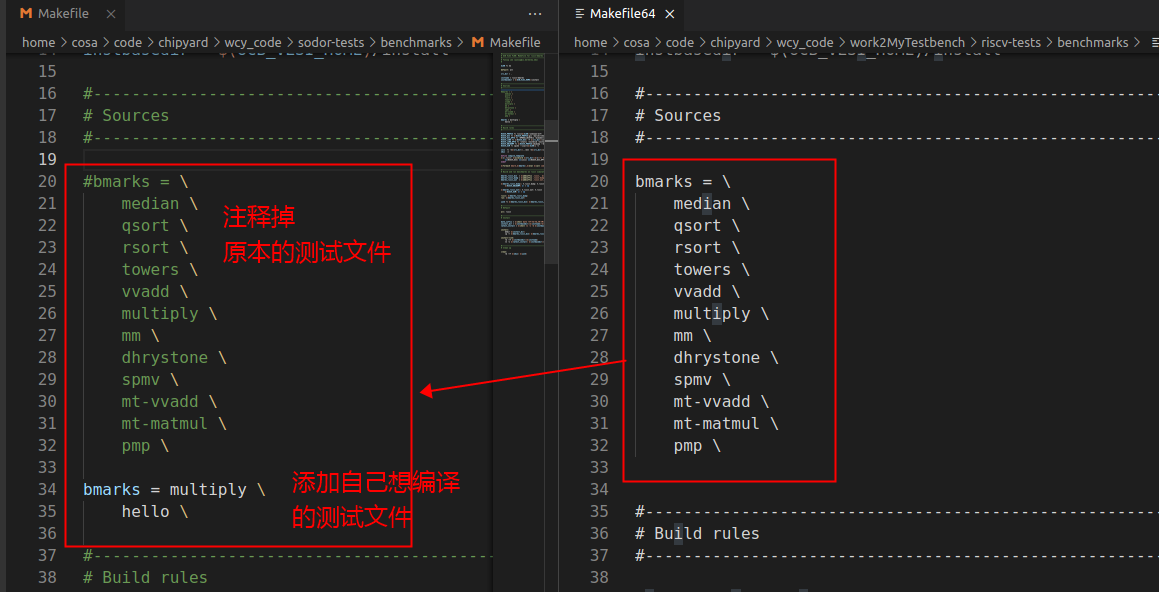
图9 hello.c代码编写

Hello.c代码说明：该代码实现功能为打印hello；setStats函数将记录riscv核跑的周期数和时间，最好不要删除。

**3.修改Makefile文件使之能生成rv32i架构的测试代码**

Makefile文件位于/sodor-test/benchmark中。由于我们的sodor-test是从chipyard中得来的，原始的makefile文件将生成rv64架构的测试代码。而sodor只能运行rv32i架构的测试代码，所以我们要对makefile进行修改并保存。Makefile的修改参考图10。





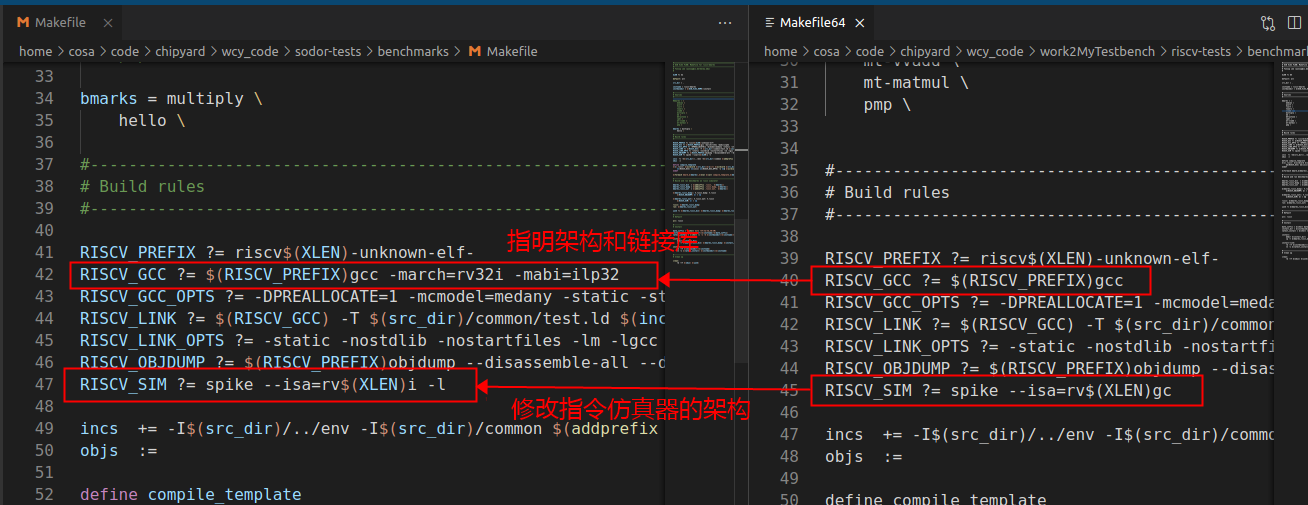


图10 Makefile的修改

**4.编译生成.riscv可执行文件作为测试文件**

在/sodor-test/benchmark处打开terminal，执行2.2.1的步骤4、5配置RISCV32工具。然后输入：**make**，利用makefile自动生成测试文件。

Make将根据makefile中的bmark决定对哪些文件夹进行编译（我的makefile中规定bmark包含hello和multiply）。

Make后将生成对应的.riscv文件和.riscv.dump文件。.riscv文件为可执行的二进制文件；.riscv.dump为.riscv的反汇编文件，它将.riscv的机器码翻译成了汇编，可直接被打开阅读。

.riscv文件就是我们想要的测试文件。至此，我们自己的测试文件成功生成。

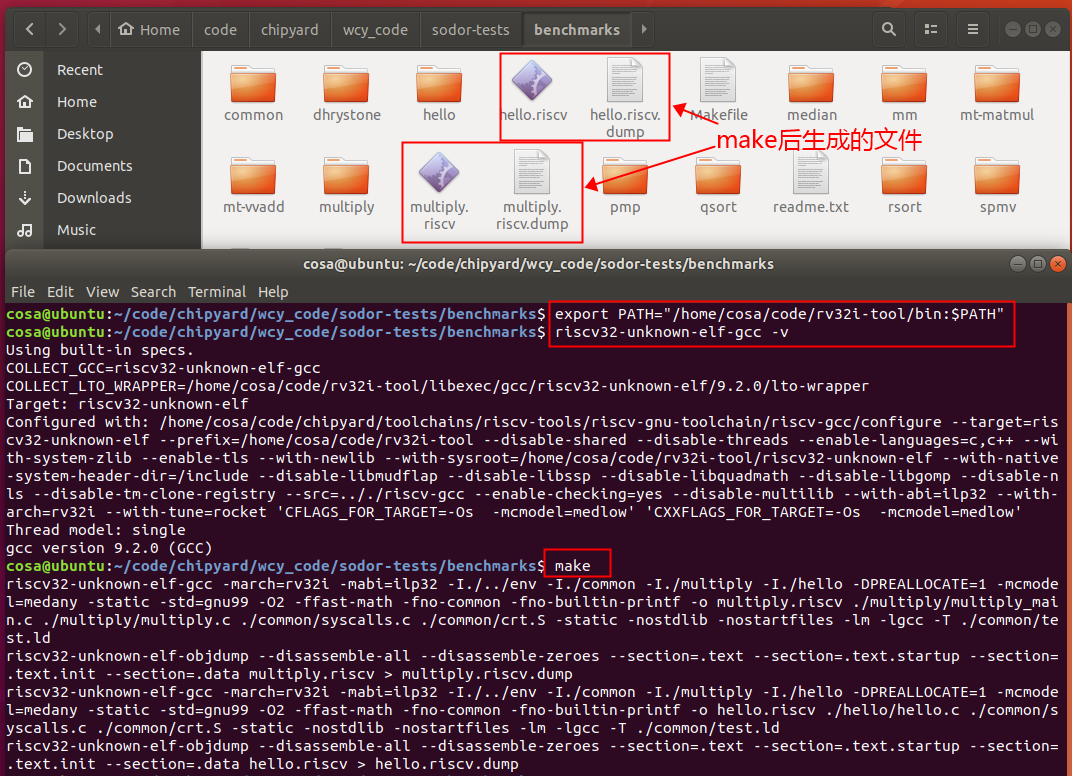


图11 步骤4示意图

#### 2.2.3 测试流程

拥有自己的测试文件后，我们就可以尝试生成自己的sodor核，在sodor核上跑测试文件。下面将详细阐述利用测试文件测试sodor核的流程，此处选择3级流水线的sodor核。

**1.生成sodor核**

在chipyard目录处打开terminal，执行以下命令。

|  |  |
| --- | --- |
| 1 | source env.sh |
| 2 | cd sims/verilator |
| 3 | make CONFIG=Sodor3StageConfig |

通过make命令我们将生成一个3阶的Sodor核，可以在/sims/verilator中看到，名为simulator-chipyard-Sodor3StageConfig。

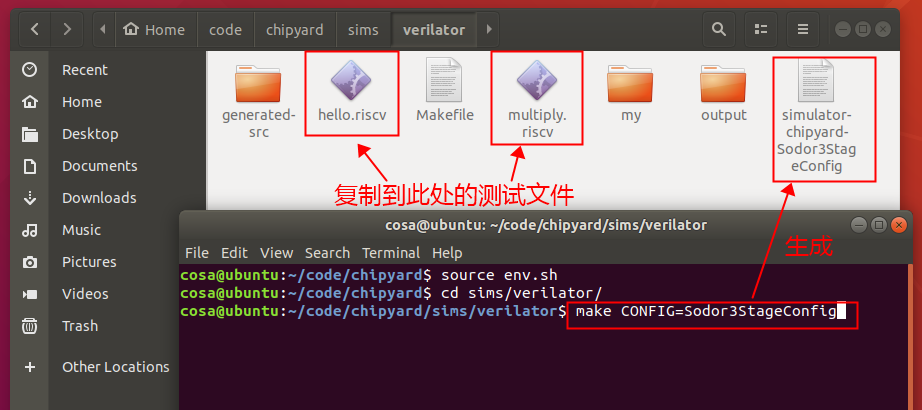


图12 生成sodor核

**2.直接用测试文件对sodor核进行测试**

**我们能够**编写自己的benchmark，通过检查报错与输出内容，验证自己riscv核的功能正确性。我们的benchmark会自动打印出mcycle和minstret，从而可以直接计算CPI，也可以对测试程序运行的时间进行直观对比。

输入以下命令，能直接用测试文件对sodor核进行测试。

|  |  |
| --- | --- |
| 1 | ./simulator-chipyard-Sodor3StageConfig multiply.riscv |
| 2 | ./simulator-chipyard-Sodor3StageConfig hello.riscv |

测试结果呈现于图13。在每个测试程序跑完后，会自动打印出mcycle和minstret，从而可以直接计算CPI，也可以对测试程序运行的时间进行直观对比。这种测试方法较简便。也可以看到在运行hello.riscv的时候确实会打印出hello的字样，说明我们自己的测试程序编写、编译等都没有出错。

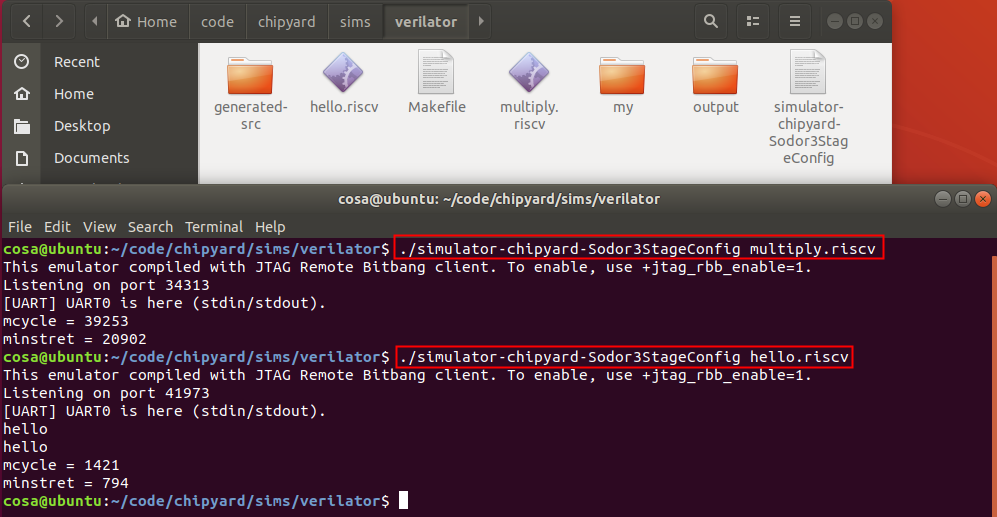


图13 直接用测试文件对sodor核进行测试的结果

**3.打印每一周期的执行指令，并对指令进行统计**，从而比较性能（如CPI、周期数、指令数、bubble等）。

在/chipyard/generators/riscv-sodor下有scripts文件夹，里面存放着可以统计指令的python脚本。我们将这一文件夹复制到verilator文件夹中。

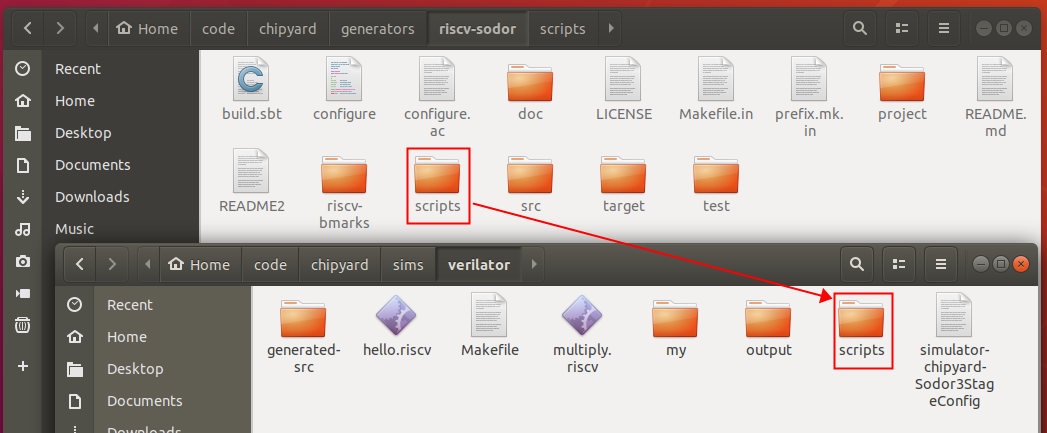


图14 复制统计用的脚本

然后执行命令：**make CONFIG=Sodor3StageConfig BINARY=multiply.riscv run-binary**，则会在output文件夹中生成对应测试文件的**.riscv.out**文件，这种文件记录了riscv核每一周期执行的命令。

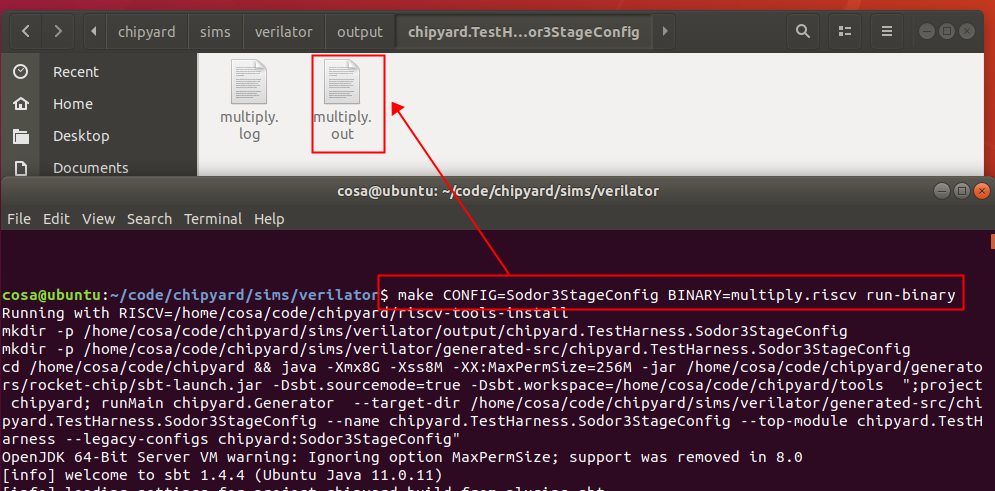


图15 生成.riscv.out文件

然后将.riscv.out文件复制到scripts文件夹中，在scripts文件夹中打开terminal，并输入以下命令：**python3 tracer.py xxx.riscv.out**，即可得到统计数据。

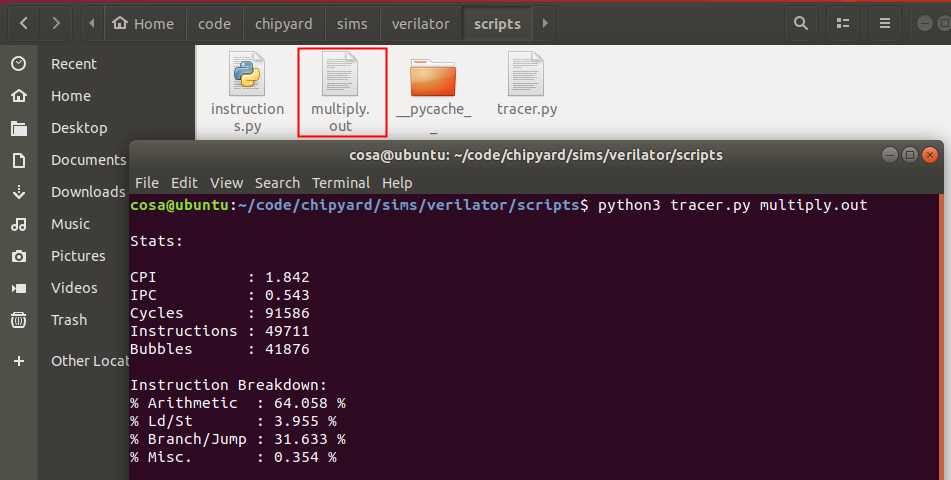


图16 由脚本统计执行的指令

## 3 具体设计与实现

### 3.1数据格式

目前主流的数据存储格式是整型数和浮点数，整型数的计算效率高，但是数据表示范围较小，而浮点数表示范围大，但是计算需要分成两个部分，导致计算效率低下。因此老师提供了另外一种量化方式-伪浮点数：对所有的数据使用相同的量化因子进行量化，把数据限定在-1和1之间，即第一位表示符号位，后续位表示0-1之间的数。

经过分析：在乘法过程中，8bit数x 8bit数的结果并不会超过16bit，但是在累加过程中可能会出现越界行为。有符号8bit数的矩阵运算（127x127x256）10的结果是一个23bit的有符号数。如果采取伪浮点数的量化方式，用16bit数去量化23bit数势必存在7bit的精度损失，而且我们拟采用的sodor内核仅支持32位整型数，若使用伪浮点数的量化方式，需要设计新的硬件数值转换电路进行大量的数值转换，由此来带的延时和精度损失会很大。若将数据内存整体转换成伪浮点数存储，也需要设计额外的电路来保证跳转、分支指令的正常运行，同样该方式带来的存储性能提升不如浮点数，但是硬件资源消耗和计算性能却更差。

综合考虑后，我们决定仍然使用8bit有符号整数进行运算

### 3.2整体框架

我们采用RISCV-Sodor-rv32-3stage作为我们的base，在Sodor的基础上进行修改，下图是Sodor-rv32-3stage的框架示意图：

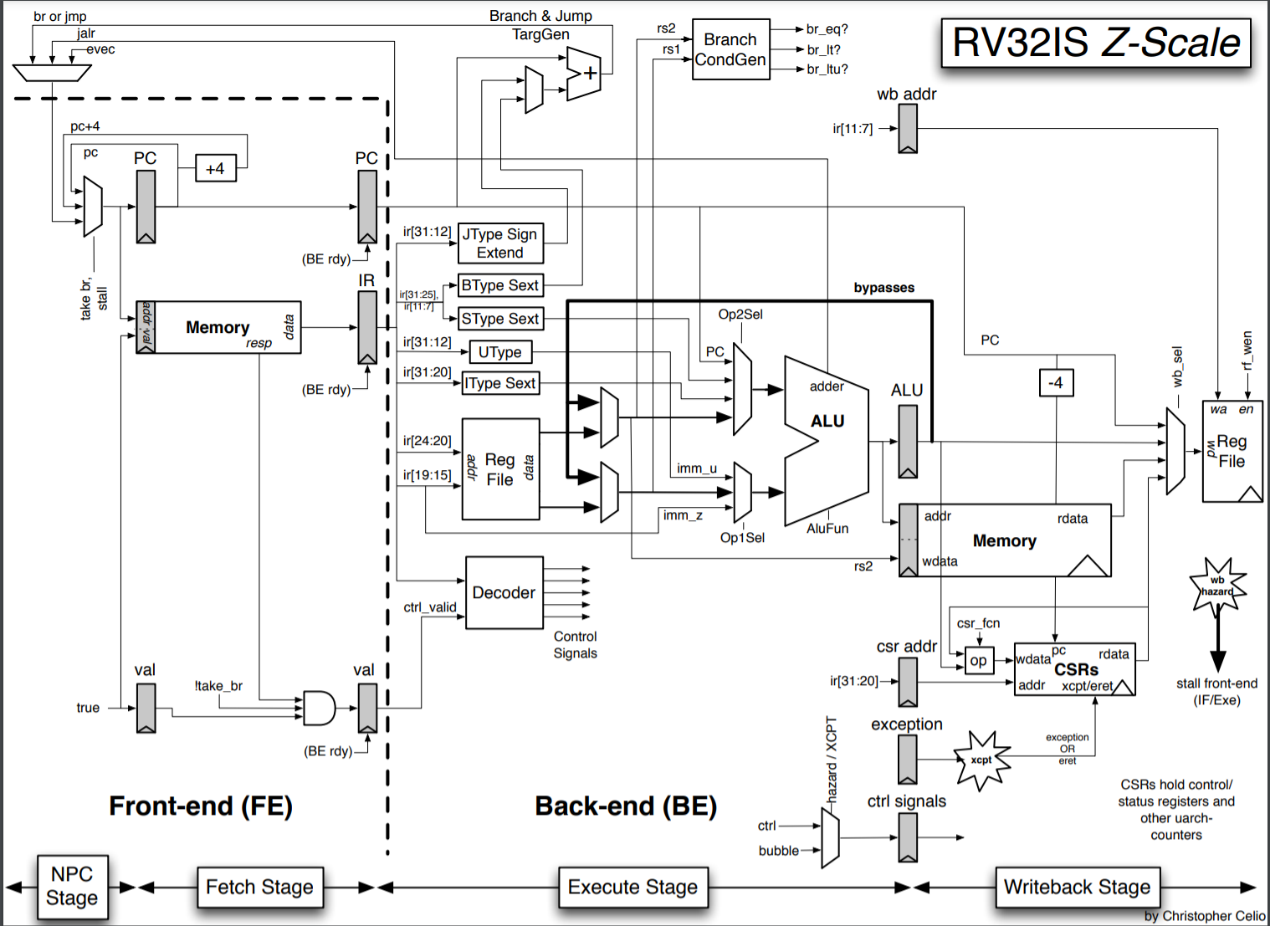


图1 Sodor-rv32-3stage的框架示意图

下面对Sodor中的代码进行一些分析，通过sodor的代码，我们可以更容易去理解课堂上学到的流水线、分支预测等知识，sodor的代码框架如下：

表1 详解sodor的代码框架

|  |  |
| --- | --- |
| alu.scala | 实现算术运算单元 |
| arbiter.scala | 内存仲裁器（并未使用） |
| consts.scala | 定义一些使用的常量 |
| cpath.scala | 逻辑控制通路 |
| dpath.scala | 数据通路 |
| frontend.scala package.scala |  |

#### 3.2.1 cpath.scala

通过该模块我们可以更快的了解到sodor实现的功能：

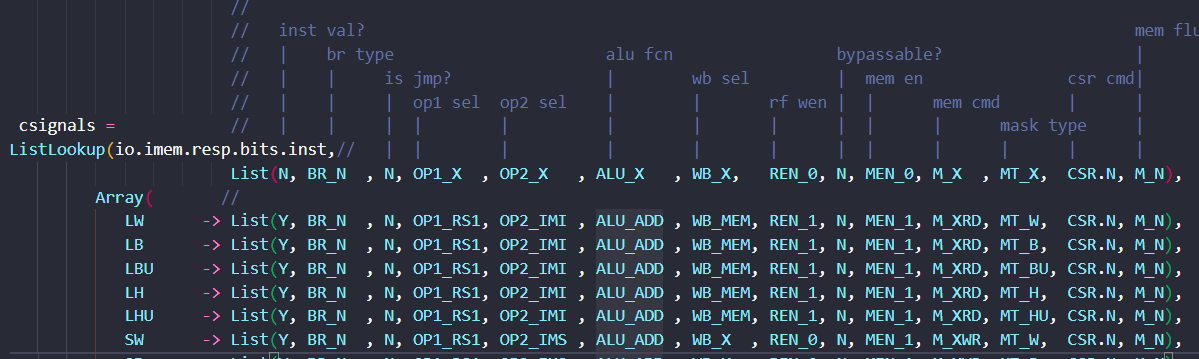


图2 sodor功能详解

该模块中，对输入的数据io.dat.inst进行查找，与指令的操作码进行比较，当符合相应指令时，将相应的控制信号赋给csignals，每一个指令的控制信号并不同，具体的控制信号定义可以在const.scala中查看。在这里我们可以看到其中实现了bypass功能：

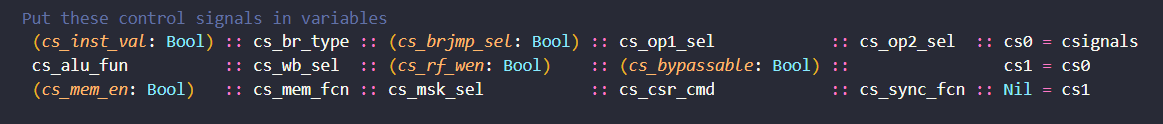


图3 bypass功能示意

这里将csignals中的值，逐一赋给下面的变量。

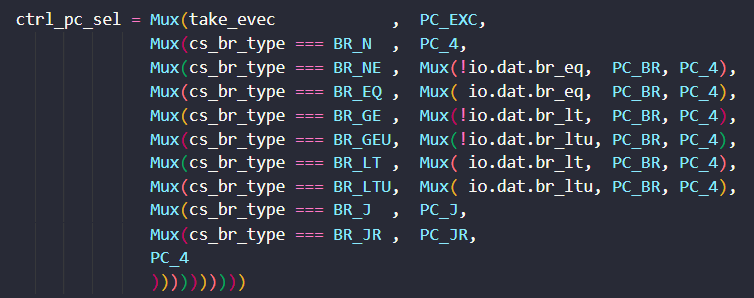


图4 变量赋值

利用cs\_br\_type的新值和io.dat.br\_x中的返回值判断下一个pc是否为跳转的情况。使用io.dat.br\_x来帮助判断因为某些循环会用到上次的值，例如要循环100次，判断cs\_br\_type知道指令为跳转指令，但不知道循环了多少次，所以需要用io.dat.br\_x来记录。



图5 IMemory的有效信号

图5的代码定义了IMemory的有效信号，当出现异常指令或者dpath还没有执行完毕时，暂停从imem中读取指令。最后是异常处理的代码：

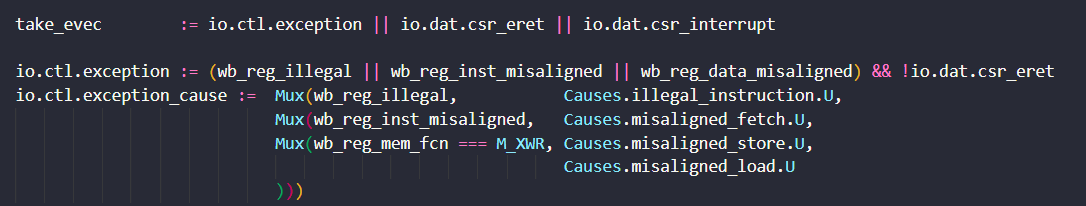


图6 异常处理代码

异常的原因主要有：imem暂停，非法指令，分支预测失败。

#### 3.2.2 dpath.scala

dpath.scala在取指后加了一级执行的寄存器，在写回阶段也增加了一级寄存器。当在执行阶段时，取指寄存器可以预先取指，这样能够提高系统的效率。如果没有执行阶段的寄存器，在执行非单步指令时，cpu流水线要暂停取指，等当前指令执行完后才能进行下一步的取指操作，这样会大大降低系统的效率。当在写回阶段时，可以提前将执行阶段的结构存入写回寄存器，然后执行阶段又可以开始执行下一个指令。同时在dpath中设计了一个bypass通路，可以把ALU的计算结果直接送回ALU，这样可以减少数据冲突，提高流水线的效率。



图7 各阶段寄存器

图7是定义的各阶段的寄存器。

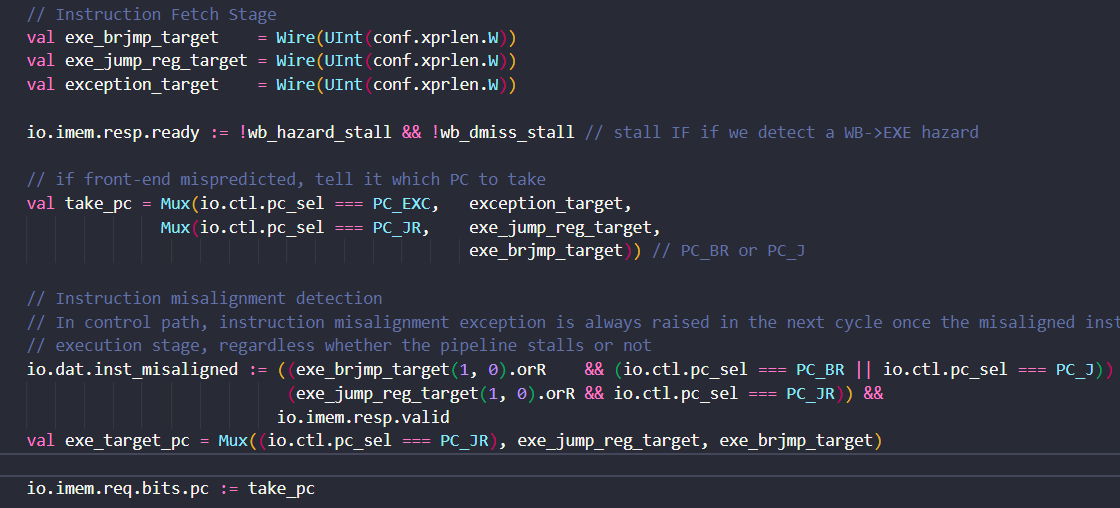


图8 控制信号

图8实现了从取指阶段到执行阶段的控制信号过度。当执行阶段出现了冒险或者missdata时，设置imem无效。然后是分支预测失败时的异常定义：当上一条指令进行了分支跳转不需要继续执行当前指令或者imem出于暂停状态表示指令分配错误。这些代码，作者都给了很详细的注释帮我们去理解代码的逻辑。

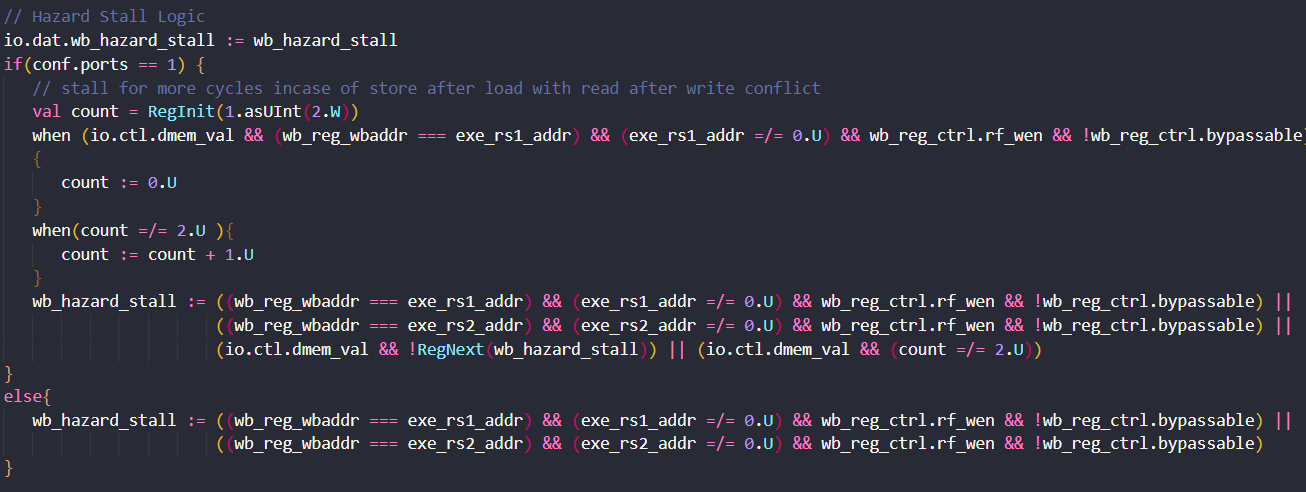


图9 针对冲突的定义

图9这段代码是针对冲突的定义：

在sodor-3stage的配置文件中conf.port = 2，所以我们只关心else中的代码块：当前要写回的地址，是下一条指令的寄存器地址，且bypass信号为0时，表示出现了冲突wb\_hazard\_stall = 1，需要进行异常处理。

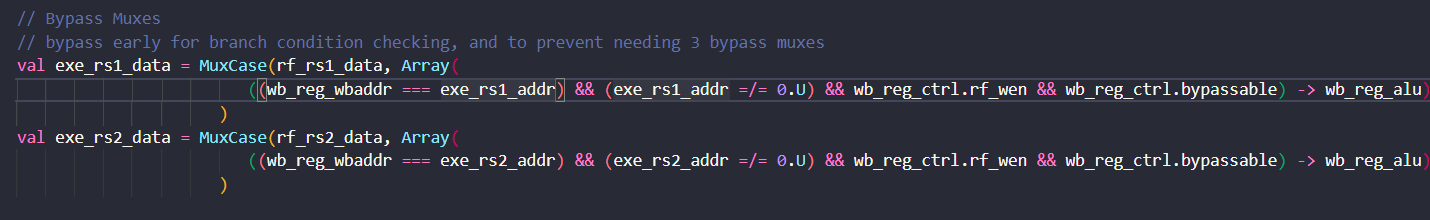


图10 选择数据来源

上面的代码选择执行阶段进行操作的数据来源，默认都是从Reg File中的寄存器取值，当出现数据冲突，下一条指令的源寄存器是当前指令的写回寄存器且允许bypass的时候，操作数据来源于上一阶段的ALU。

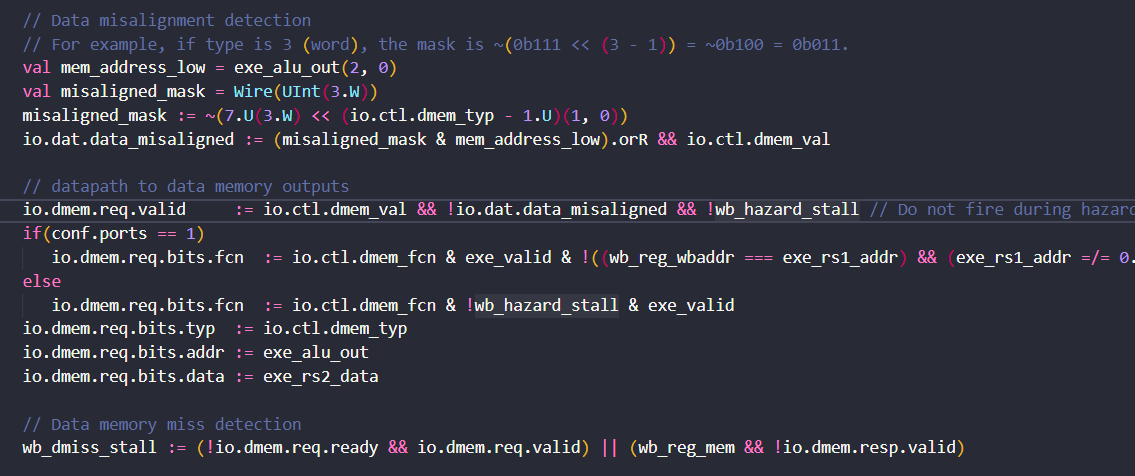


图11 数据未命中异常

图11的代码定义了数据未命中异常。

#### 3.2.3 frontend.scala

该模块将取指阶段和imem集成在一起构成一个前端，执行阶段和写回阶段在dpath中构成一个后端。

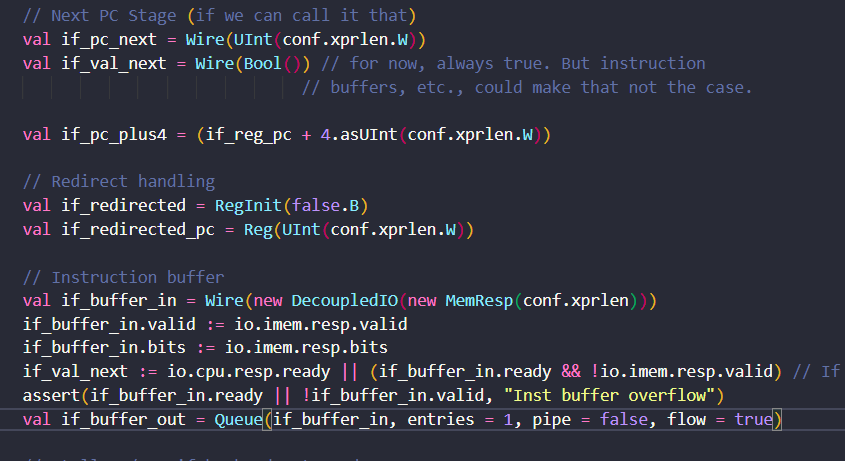


图12 imem缓冲器

图12的代码定义了一个imem的缓冲器，同时规定了sodor的分支预测总是预测不跳转，执行pc+4的指令。

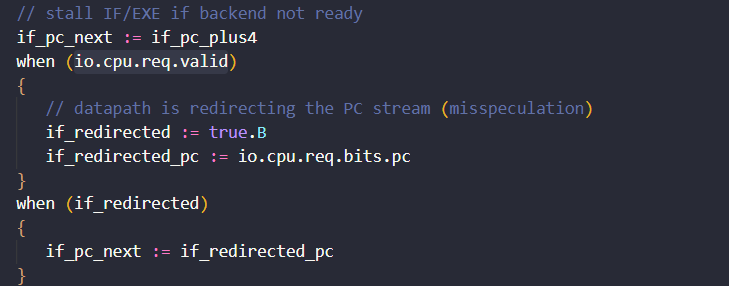


图13 取指阶段的暂停

图13的代码实现取指阶段的暂停，io.cpu.req.valid是imem的req.valid信号，若bakcend仍在执行，则io.cpu.req.valid为1，PC保持不变。

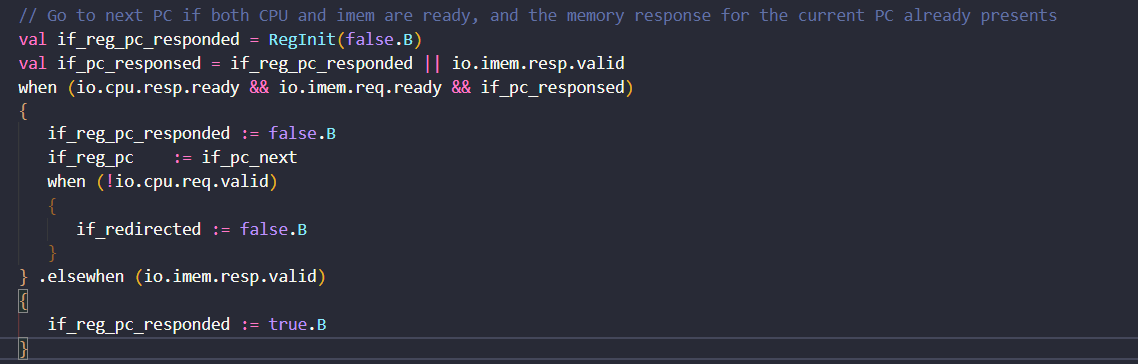


图14 指令的顺序执行

上面的代码实现了指令的顺序执行，当imem准备好且backend执行完毕，CPU取出下一条指令进行执行。

通过对sodor-3stage的代码阅读，可以更加直白地理解流水线、分支预测和bypass的作用，虽然没有实际自己实现这些功能，但是理解这些代码的逻辑同样能巩固课堂上学到的知识。

### 3.3乘法器

sodor仅支持risc32i指令集，该指令集中并没有乘法指令，若要实现乘法效果，必须要多次使用risc-v指令的加法和移位指令来实现乘法指令，这会极大地降低矩阵乘法的效率，因此我们将在sodor中定义并实现mul功能。

#### 3.3.1 无符号乘法器设计

Chisel（Constructing Hardware In a Scala Embedded Language）是UC Berkeley开发的一种开源硬件构造语言。它是建构在Scala语言之上的领域专用语言（DSL），支持高度参数化的硬件生成器。Chisel代码可以通过FIRRTL模块转化为可综合的RTL代码，因此我们只需要在chisel代码层面实现乘法，系统会自动生成相应的verilog代码。

按照2.1.3节的方法新建一个chisel工程，配置好库文件后，编写Mul模块的代码如下：

图15定义2个4比特的输入，1个8比特的输出。

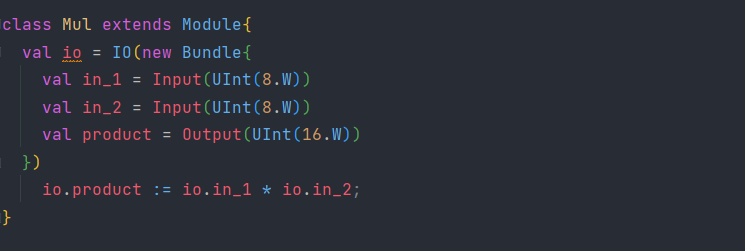


图15 mul模块代码

定义一个测试类：

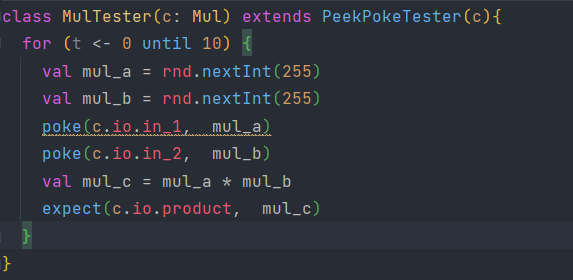


图16 定义测试类

进行测试：

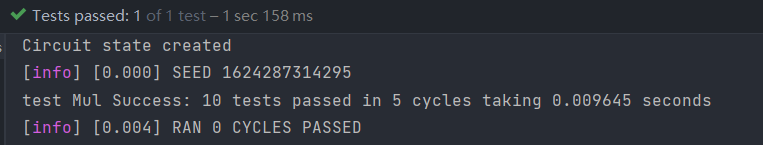


图17 测试结果

十次测试均正常通过，无符号数乘法器功能正常。

#### 3.3.2 有符号乘法器设计

在无符号数乘法器的基础上实现有符号乘法器的设计。虽然Chisel中支持带符号整数数据SInt，但是考虑到与sodor代码中的ALU功能模块数据格式一致，在设计带符号乘法器时，仍然使用UInt数据，通过人工转换为对应的带符号数据进行验证。

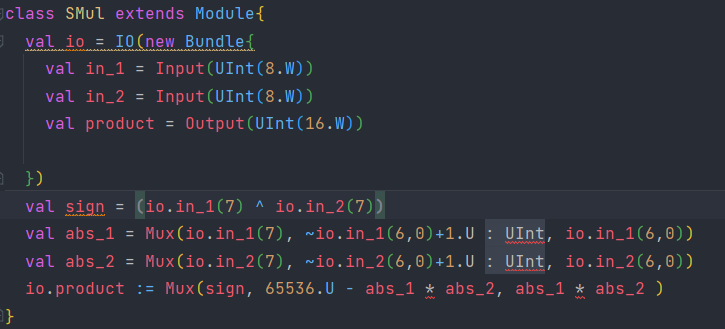


图18 有符号乘法器的代码

定义一个sign表示最终结果的符号位，若同号，则结果是正数，若异号，则结果是负数。然后将输入的值转换为对应的绝对值：若为正数，取其低七位；若为负数，第七位取补码转化为对应的绝对值。将输入转化为绝对值进行乘法运算，最终的输出要根据符号位决定，若为正数，则将无符号乘法结果输出，若为负数，将其转换为对应的补码形式。

对上述代码进行测试：

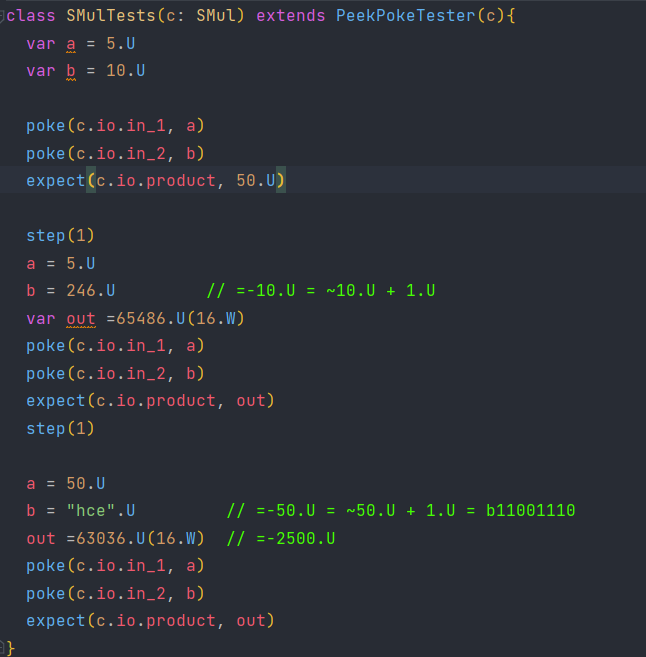


图19 对有符号乘法器进行测试

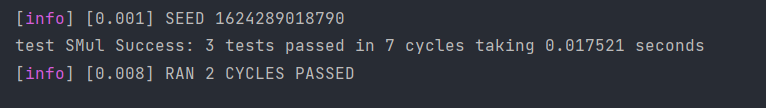


图20 测试结果

三个测试均通过，说明该有符号整数乘法功能正确。

#### 3.3.3 带乘法功能的ALU设计

加有符号整数的乘法器添加到Sodor的ALU.scala中。在这个乘法器中我们定义将sign作为一个外部输入，同时定义了一个使能信号en：



图21 乘法器代码

在ALU中我们根据ALU的控制信号，定义了一些运算的使能信号，这样能够避免一次ALU执行，把每个ALU功能均实现，能够减小硬件电路的能耗。

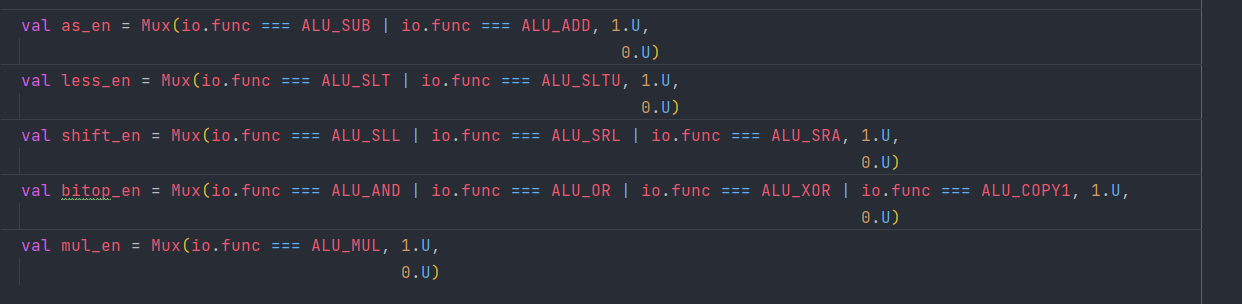


图22 ALU代码

在ALU中调用乘法模块：

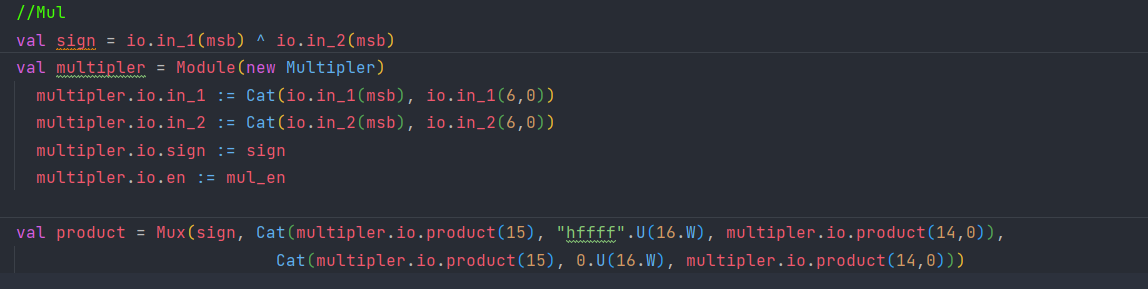


图23在ALU中调用mul模块

因为sodor中数据的位数都是32位，而我们设计的乘法器是8位输入，16位输出，所以乘法计算的最终结果仍然要根据符号位进行调整：将乘法器输出结果的符号位赋给ALU中的product的最高位，若结果是负数，则中间位数补0，否则补1。

对添加了乘法器的ALU进行最终测试，分别测试了7个指令：



图24 对乘法器进行测试

测试结果均正确：

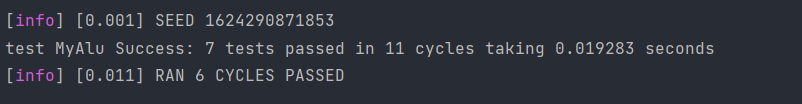


图25 测试结果

### 3.4 精简数据通路

Sodor中实现了riscv32i的所有指令，但是在实现矩阵乘法的过程中，并非需要用到riscv32i的所有指令，因此我们对sodor的数据通路以及控制模块进行了精简，这样能够减小硬件资源消耗，提升运行速度。

#### 3.4.1 确定所需指令

首先对测试使用的testbench.c文件进行汇编，根据编译后的到的testbench.dump文件进行指令统计。经统计后确认以下riscv32i指令未经使用：

LH、LHU、SH、SLTI、SLTIU、SLT、XOR、SRA、ECALL、EBREAK、WFI、FENCE\_I

为了验证矩阵乘法的正确性，以及打印出相关指令的统计结果，实际上.dump文件中仍然包含了很多矩阵乘法所用不到的指令。实现矩阵乘法所需的指令如表1所示。

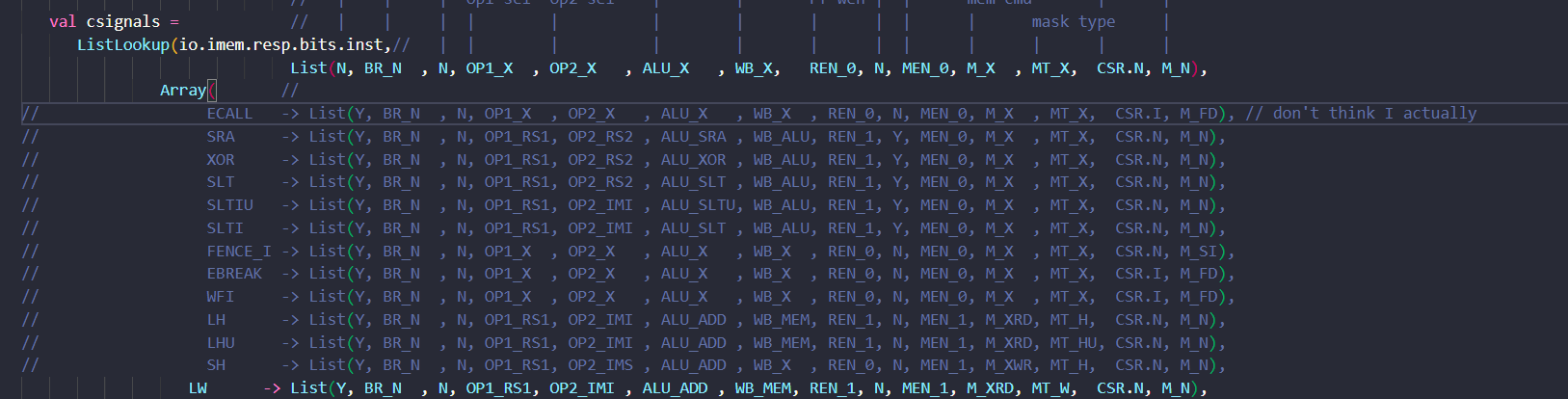
表1 实现矩阵乘法所需的指令

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Add | bge | csrw | lw | sw |
| Addi | bgeu | j | mv |  |
| Andi | bltz | jal | ret |  |
| Auipc | csrr | li | slli |  |
| Beqz | csrs | lui | srai |  |

为了测试方便，我们仍然保留验证和打印所需的指令。

#### 3.4.2 精简代码

将sodor中实现上述无用指令的模块裁去。首先注释掉cpath中的查找表：



由于无用指令较少，dpath中除了Alu模块无其他改动：Alu模块中SLT功能不再需要，将其裁去。

## 4 功能测试与性能评测

### 4.1 功能正确性测试

这一部分我们将添加了mul模块的三阶sodor核（以下称添加了Mul模块的sodor核为mul-sodor核，原来的sodor核称为org-sodor核）按照2.2.3的步骤1进行生成，然后利用我们自己编写的benchmark——mulonly.riscv与shiftmulonly.riscv进行mul模块的功能正确性测试。

#### 4.1.1 利用shiftmulonly.riscv文件检查mul-sodor核的正常功能

此处的shiftmul即利用移位的方法实现乘法操作，这是适用于在原本没有Mul模块的sodor中进行乘法操作。利用shiftmul，原本的sodor即使不适应rv32im的指令架构，也能在rv32i的架构（该架构中不支持Mul指令）下完成乘法操作。

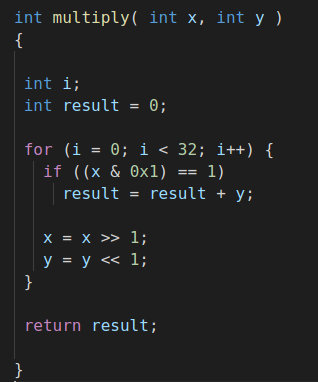


图17 shiftmul的C代码实现

为了检查添加的Mul模块是否对sodor本身的功能产生了影响，我们编写了shiftmulonly.c文件，并编译成了.riscv可执行文件。图18展示了shiftmulonly.c的main函数，从中可以看出若功能不正确，mul-sodor核会返回非0值。

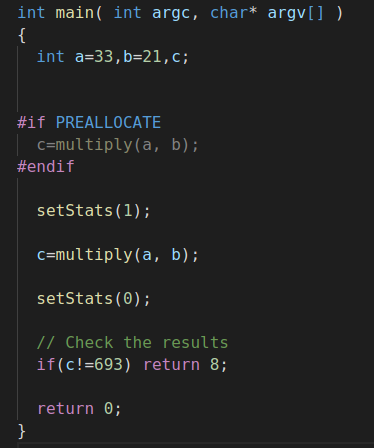


图18 shiftmulonly.c文件的main函数

生成shiftmulonly.riscv后进行了测试，测试结果如图19所示——正常地返回了周期数和指令数，没有诸如\*\*\*fail\*\*\*(tohost=8)的报错。这说明mul-sodor核的Mul模块并没有对sodor本身的功能产生了影响，除了mul以外的指令都能正常执行。

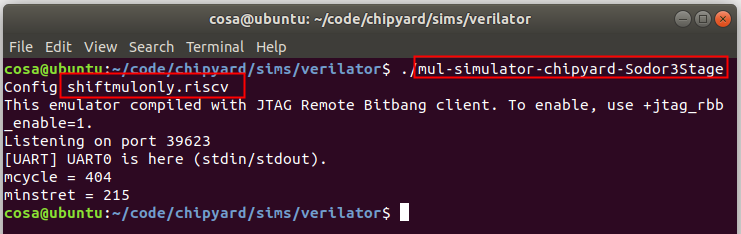


图19 利用shiftmulonly.riscv测试mul-sodor核

#### 4.1.2 利用mulonly.riscv文件检查mul-sodor核的mul功能

mul模块是我们自行添加的运算功能模块，需要用我们自定义的指令进行调用。因此在编写benchmark的时候需要注意以下两点：如何在C代码的编写中使用我们的自定义指令？如何使编译器理解并正确将其编译？下面将分三个部分展开描述。

**1.使用.inst汇编指令编写自定义指令**

对于rv32i架构，其指令的类型如图20所示。

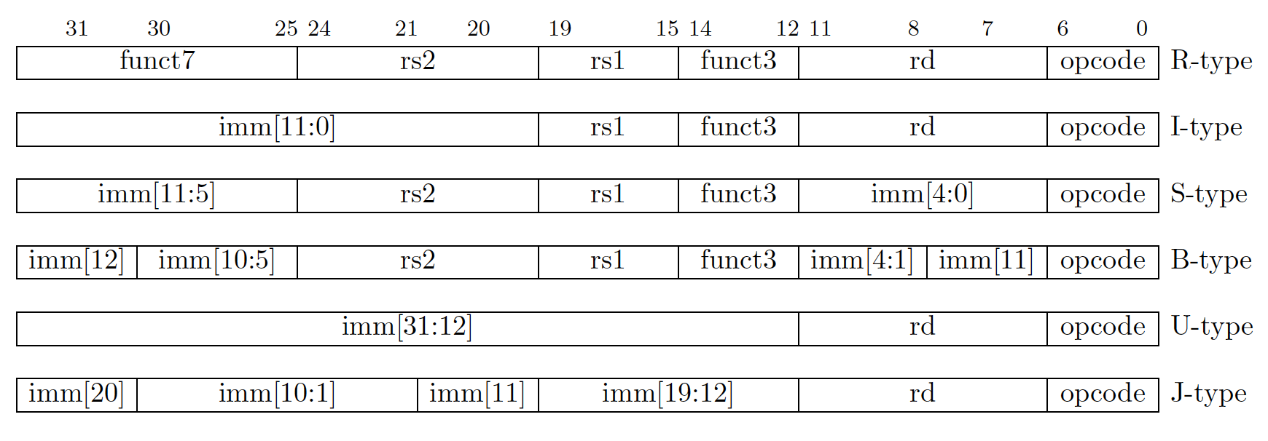


图20 rv32i架构的指令类型

我们的mul指令是R-type的扩展指令，采取了与rv32imc架构中m拓展包的mul指令相同的机器码。Mul指令的具体内容与各部分拆解见表2。

表2 mul指令具体内容

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | | | | |
| Funct7 | rs2 | rs1 | funct3 | rd | opcode |
| 000001 | ????? | ????? | 000 | ????? | 0110011 |

查找资料发现，扩展指令在汇编中可以以.insn 的方式表征，如下1所示。则我们的mul指令将以2进行表示，其中%0表示rd，%1%2表示rs1和rs2。使用.insn时，该指令会以机器码的形式直接呈现给编译器，编译器并不会尝试对其进行编译或识别，而是直接将那一行机器码放在对应的位置上。

|  |  |
| --- | --- |
| 1 | .insn r opcode, func3, func7, rd, rs1, rs2 |
| 2 | .insn r 0x33, 0, 1, %0, %1, %2 |

我们将riscv汇编与C程序进行混编，在C程序中采取内联汇编的形式编写riscv汇编。最终C代码中的mul函数如图21所示。

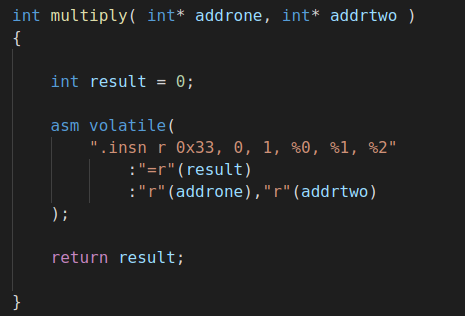


图21 C代码中的mul指令

为了检查添加的Mul模块是否能正确运行，我们编写了mulonly.c文件，并编译成了.riscv可执行文件。图22展示了mulonly.c的main函数，从中可以看出若功能不正确，mul-sodor核会返回非0值。

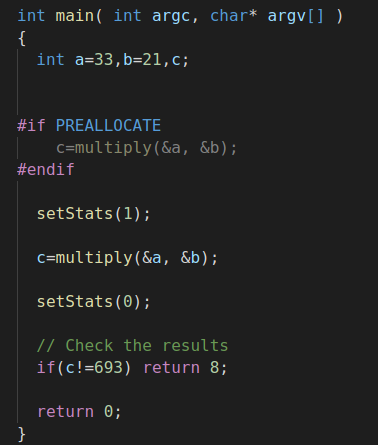


图23 mulonly.c文件的main函数

**2.编译benchmark**

我们的mul指令通过内联汇编、直接内嵌机器码的形式编写在了C中，编译器看到.insn并不会尝试进行编译而直接将机器码放置在原位,所以我们并不需要对riscv编译工具riscv32-unknown-elf-gcc进行修改。此外，我们的mul指令的机器码与rv32imc扩展包中mul指令的机器码是一致的，我们的riscv32工具链中的反汇编工具本来就能将rv32imc的mul的机器码进行识别，所以我们也不需要对反汇编工具进行修改。

如图24所示，我们直接使用rv32i架构的riscv32-unknown-elf-gcc对mulonly.c文件进行编译，得到.riscv和.riscv.dump。

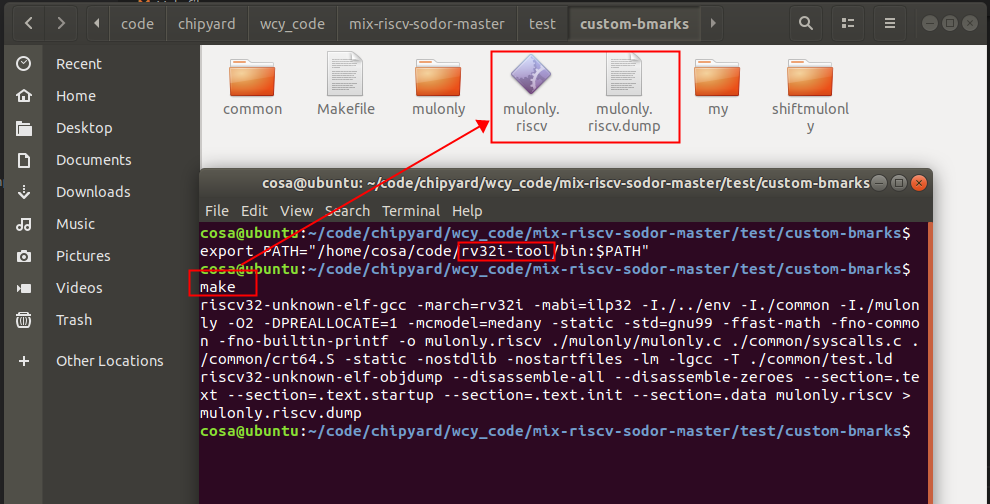


图24 使用rv32i架构的riscv32编译工具链对mulonly进行编译

打开.riscv.dump文件，可以看到我们的扩展mul指令以成功地以机器码的形式被放置在multiply函数下，并被反汇编工具成功识别（见图25）。

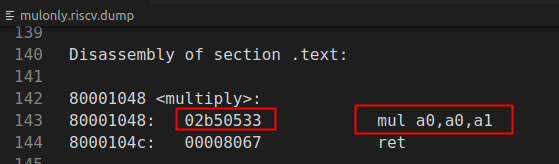


图25 multiply函数中的mul指令

表3 对multiply函数中的mul指令的详解

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Hex: 02b50533 | | | | | |
| Bit: 0000001 01011 01010 000 01010 0110011 | | | | | |
| Funct7 | rs2 | rs1 | funct3 | rd | opcode |
| 000001 | 01011(a1) | 01010(a0) | 000 | 01010(a0) | 0110011 |

**3.利用mulonly.riscv对mul-sodor进行测试**

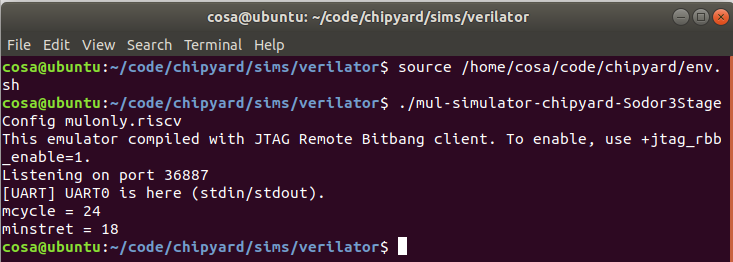


图26 利用mulonly对mul-sodor进行测试

利用含有mul指令的benchmark对mul模块进行测试，测试结果如图26所示——正常地返回了周期数和指令数，没有诸如\*\*\*fail\*\*\*(tohost=8)的报错。这说明我们的Mul模块功能正确。

### 4.2 性能评测

#### 4.2.1 乘法运算的性能比较

由于原始的org-sodor核只能用移位乘法实现乘法，所以我们在org-sodor上利用shiftmulonly进行测试，在mul-sodor上利用mulonly进行测试。我们利用2.2.3中的步骤3的方法，通过统计各个周期的指令，对移位乘法和利用mul模块的乘法的性能进行比较。结果如表4所示。

表4 移位乘法和mul模块乘法的对比

|  |  |
| --- | --- |
| 移位乘法 | 利用Mul模块的乘法 |
|  |  |
| **分析：**在这个案例中，两者CPI与IPC几乎一致，但利用mul的乘法在**周期数和指令数**上只有是移位乘法的**66%**，即运行时可以**提速约40%**。此外，移位乘法的运行时间、周期数、CPI、分支预测准确率都会与乘数有关，性能并不稳定；而mul乘法由于利用了新的硬件，执行乘法在固定的周期内必然完成。  **结论：利用mul的乘法的性能远好于移位乘法。** | |

#### 4.2.2 矩阵乘法的性能比较

矩阵乘法算法采用最普通的循环算法，具体代码见图27。我们改变了参与运算的矩阵的大小，分别尝试了两个5\*5方阵相乘、两个10\*10方阵相乘、两个20\*20方阵相乘。测试方法仍然采用2.2.3中的步骤3的方法，通过统计各个周期的指令，对org-sodor和mul-sodor在矩阵乘法运算上的性能进行比较，结果见表5。

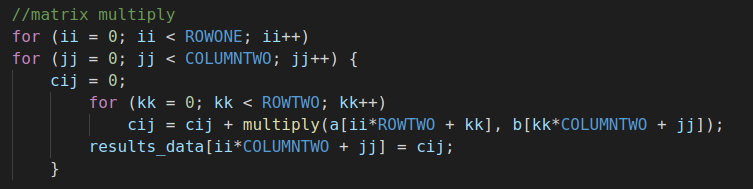


图27 矩阵乘法的代码

表5 org-sodor和mul-sodor在矩阵乘法运算上的性能比较

|  |  |
| --- | --- |
| org-sodor | mul-sodor |
|  |  |
|  |  |
|  |  |
| **CPI分析：**org-sodor与mul-sodor的CPI都随着矩阵规模扩大而上升，但mul-sodor的CPI始终比org-sodor的CPI低0.2左右。 | |
| **周期数与指令数分析：**mul-sodor的周期数始终比org-sodor的小1~2个数量级。运行时间差距巨大。 | |
| **分支跳转分析：**两个核的分支跳转数基本与矩阵规模无关，但mul-sodor的分支跳转占比始终比org-sodor的少，则分支预测错误带来的影响小。 | |
| **总结：**mul-sodor与org-sodor在运行时间上有数量级上的差距，mul-sodor在CPI核分支跳转上也有一定的优势。可见添加了Mul模块的sodor核在矩阵运算上有大幅的性能提升。 | |

## 5 项目总结

### 5.1 项目成果总结

### 5.2 成员分工

### 5.3 反馈与建议

## [参考文献]