# 

# 《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](#_Toc75105022)

[1.1 设计目标 3](#_Toc75105023)

[1.2 技术方案 3](#_Toc75105024)

[1.3 项目创新点 3](#_Toc75105025)

[2 原理性验证 3](#_Toc75105026)

[2.1 RISC-V环境验证 3](#_Toc75105027)

[2.2 测试方法验证 3](#_Toc75105028)

[2.2.1 测试工具与环境 3](#_Toc75105029)

[2.2.2 测试程序搭建 3](#_Toc75105030)

[2.2.3 测试流程 3](#_Toc75105031)

[3 具体设计与实现 4](#_Toc75105032)

[4 功能测试与性能评测 4](#_Toc75105033)

[4.1 功能正确性测试 4](#_Toc75105034)

[4.2 性能评测 4](#_Toc75105035)

[5 项目总结 4](#_Toc75105036)

[5.1 项目成果总结 4](#_Toc75105037)

[5.2 成员分工 4](#_Toc75105038)

[5.3 反馈与建议 5](#_Toc75105039)

[[参考文献] 5](#_Toc75105040)

## 1 项目设计方案

### 1.1 设计目标

精简RISC-V指令，定制数据通路和额外指令，来为DLA量体裁衣

### 1.2 技术方案

使用单核RISC-V来计算矩阵乘法运算

我们主要使用课堂学习的知识来设计 RISC-V处理器。设计有符号乘法器来提升计

算速度，使用流水线来提升计算性能，使用分支预测来提升 for 循环的判断速度，使用合理的缓存大小来减少数据搬移，提升命中率。 由于RISC-V是相对通用的处理器，很多指令在执行该任务时不会被用到。对指令集进行裁剪可以优化数据通路，减少硬件资源，提升计算速度。我们设计新的指令并修改数据通路和控制模块，来提升计算的性能。

### 1.3 项目创新点

1.由于RISC-V是相对通用的处理器，很多指令在执行该任务时不会被用到。对指令集进行裁剪，可以优化数据通路，减少硬件资源，提升计算速度。

2.此外，由于使用了新的功能模块，我们设计新的指令并修改数据通路和控制模块，来提升计算的性能。

3.我们编译了自己的toolchain,在 riscv-test 的架构中略作修改后,再编写自己的 testbench，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。我们修改了Makefile 文件使之能生成 rv32i 架构的测试代码.

4. 我们撰写了完整的软件模拟测试流程, 为sodor的教学提供助力.

## 2 原理性验证

### 2.1 RISC-V环境验证

### 2.2 测试方法验证

#### 2.2.1 测试工具与环境

#### 2.2.2 测试程序搭建

#### 2.2.3 测试流程

## 3 具体设计与实现

### 3.2cache的设计方案

Scala的文档可以搜索关键字 scala document来寻找. Csdn许多都是一知半解盲人摸象.

L2 Cache的实现

我们要把tile中cache和其他模块连接起来.

#### Rocket中cache

需要连接 ROCC加速器, rocket core, 这些Tile连接到一个4-bank L2 Cache，该缓存本身通过AXI互连连接到外部I/O和存储系统.

HasL1CacheParameters 是一个cache类.

L1cache.scala:

是一个底层的接口. 定义了trait.定义了组相联,组数, cache大小等参数.

HellaCache.scala:

有 dcache的参数, 定义了对齐异常.

定义了HellaCacheIO, 因为rocket具有很多复杂设计,所以需要非常多的接口. 对于sodor我们不需要这么多接口.

dcache.scala: 包括数据request类, 数据metadata request类,

Cache可以借鉴rocket中 L1cache的实现.

# Sodor

这里应该有一些和智鹏的重复了, 可以分别加进去.

**Frontend.scala有:**

**Dpath.scala有:**

**Core.scala有:**

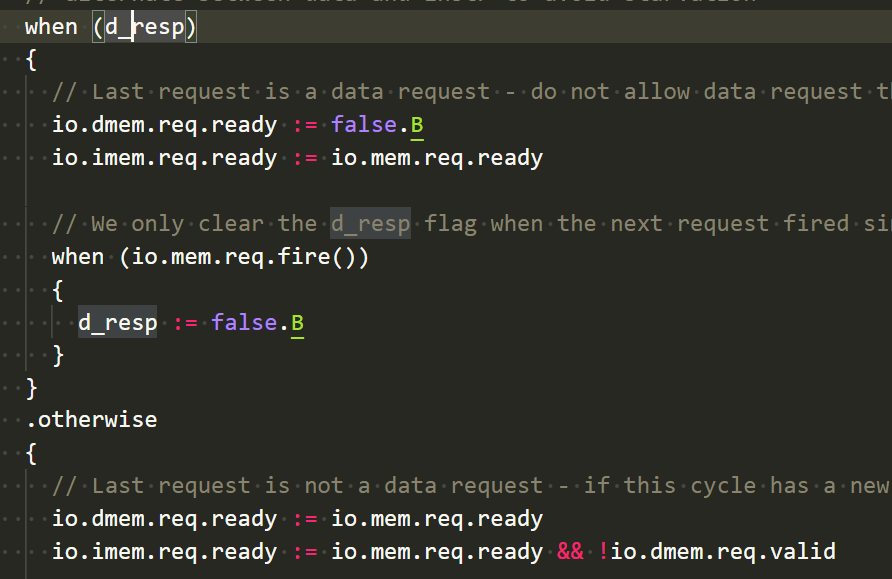
Coreio类, 定义了core和mem的接口

**Consts.scala有:**

定义了各种常量, 如PC的跳转信号, 寄存器选择信号, 写回选择信号等.

**Arbitrer.scala 有:**

这个属于普林斯顿架构中的一部分, 和mem关系比较大, 有很多对内存的控制. 在多模块共享同一资源时，需要仲裁器来决定谁可以使用共享资源（如memory access).所以输入是各模块的请求（req）. 这里就输入简单两个, 一个是data load/store请求, 一个是 instruction fetch请求. 输出就是mem



Arbiter处理数据response

如图我们可以看到 当有response的时候, 数据mem就进入忙状态, 不再接受request.

这个SodorMemArbiter类会在internaltile中实例化, 我们需要L2cache的话, 就需要改进这个类, 让他增加对L2cache的接口.

Common 文件夹中

Master\_adapter.scala

SameCycleRequestBuffer类可以起到缓冲效果, 当上一个request的response到达时再发送下一个memory request.

**Memory.scala**

定义了 MemPortIo类, 这是一个非常重要的类, 广泛用于各个模块中. 他可以接受request, 然后回复response.

ScratchPadMemoryBase类, 里面有和core的io还有debug的io, extends这个类的有两种,一种是同步的, 一种是异步的.

**scratchpad\_adapter.scala**

SodorScratchpadAdapter有hellacache到memportio的接口,一边连着memportio,一边连着cache. 可能里面的实现依然是cache?

SodorRequestRouter类在下面的tile内部连接中有用到, 会转发所有不属于scratchpad的请求. 是作为core, scratchpad 和master的中转站.

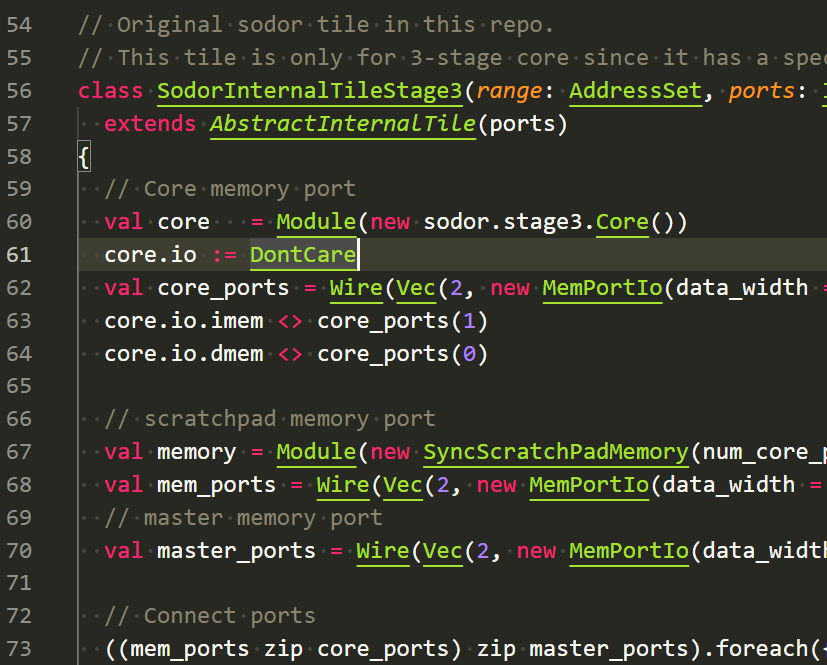
**sodor\_internal\_tile.scala**

SodorInternalTileStage3类 里面定义了tile的内部组成 , tile中包括core, memory,arbiter等, 我们可以修改memory是同步的还是异步的.

SodorInternalTile类, 是除了3stage之外的所有cores, 我们使用的3stage是比较特殊的, 其他的cores用的都是异步的memory而我们用的是同步的memory, 这个类中会创建memory = SyncScratchPadMemory的实例

case class Stage3Factory(ports: Int = 2) 是一个构造器, 3stage是有ports参数的, 而1,2,5 stage和ucode 的构造器是没有参数的, 所以用 case object .

**这里是如何连接ports的呢?我们可以看到mem ports对应core port , 然后和master ports对应,**



Tile内部的连接

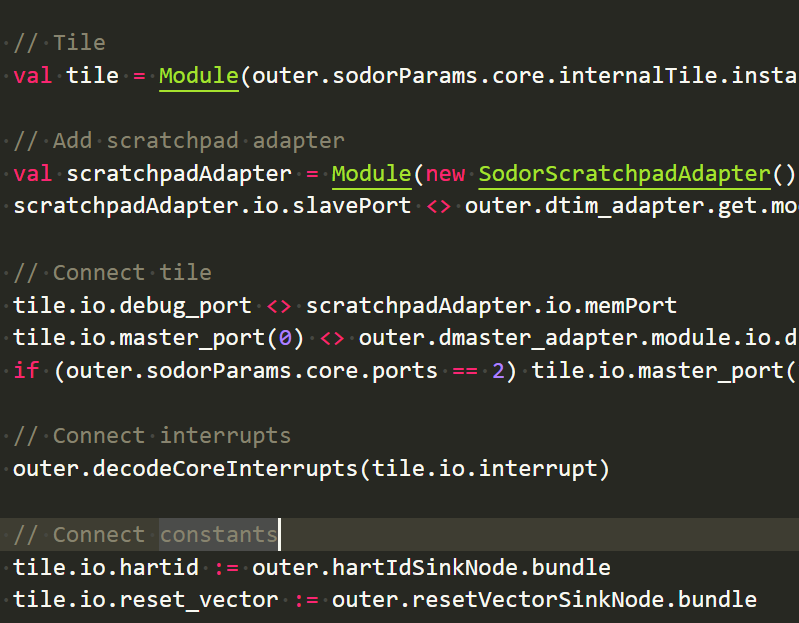
**scratchPort和memport连接, memport是两条线,一条连接dmem一条连接imem. 目前只有scratchpad\_arbiter一边连接memport, 一边连着core\_port**

**sodor\_tile.scala**

SodorCoreParams类, 里面有许多是从rocket复制来的.

**SodorTile类, extend了basetile**

**SodorTileModuleImp类, 他会进行检查, 比如会提示没有scratchpad , 或者请求的地址长度不对. 这个类是比较高层次的模块, 是调用上面的sodor\_internal\_tile.scala中的类 会把tile和 scratchpad, 中断处理等模块连接起来.**



**Sodor如何连接tile和外围模块**

**WithNSodorCores类**

**这应该是对于多核的一个尝试, 也是非常谨慎的检查各个参数.**

#### Cache的设计方案

**根据下图我们可以看到, IF从icache中取指令, 然后mem存回dcache, icache核dcache通过arbiter沟通, arbiter确定icache和dcache的优先级, 和L2cache沟通, L1 miss后, arbiter 向L2 请求数据.**

**所以我们希望sodor增加L2cache , 那就需要改进arbiter, arbiter应该多一些对L2cache的接口, 以及需要承担L2 到L1cache 的转发功能. 因此arbiter是我们着重要研究的.**



**Sodor的internal\_tile架构**

**如上图, Imem和Dmem定义在common的memory.scala中,每个memory有各自的io, 连接在arbiter上, 事实上每个模块都有各自的io, 发送request还用了request buffer, 这里不再画出.**

**我们添加了L2cache后应该是下图这个样子**



添加了L2cache的架构

## 4 功能测试与性能评测

### 4.1 功能正确性测试

Mulonly+shiftmulonly

### 4.2 性能评测

## 5 项目总结

### 5.1 项目成果总结

### 5.2 成员分工

1.跑通了sodor的emulator测试,编译了riscv32-unknown-elf toolchain,

2. 尝试在sodor 3-stage中添加L2cache

3. 撰写实践报告的1项目设计方案, 3-2 cache的设计方案

4.撰写了实验文档《如何运行sodor的emulator》

### 5.3 反馈与建议

## [参考文献]

1.https://github.com/ucb-bar/riscv-sodor

2. [ Chipyard documentation](<https://chipyard.readthedocs.io/en/latest/Chipyard-Basics/Initial-Repo-Setup.html#initial-repository-setup>)

3.[Veripool](https://www.veripool.org/verilator/)

4.[Welcome to RISCV-BOOM’s documentation! — RISCV-BOOM documentation (boom-core.org)](https://docs.boom-core.org/en/latest/)