作者姓名:王昊

RISC-V指令集模拟器的设计与实现

专业硕士学位论文

(专业学位类型)

专业领域:软件工程

校内导师:汪增福教授

企业导师:侯锐研究员

完成时间:二〇二二年三月六日

Author: Wang Hao

Design and implementation of

University of Science and Technology of China   
A dissertation for master’s degree

(Professional degree type)

RISC-V instruction set simulator

Speciality: Software Engineering

Supervisor: Prof. Wang Zengfu

Advisor: Prof. Hou Rui

Finished time: March 6,2022

中国科学技术大学学位论文授权使用声明

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文,是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外,论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名:签字日期:

作为申请学位的条件之一,学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权,即:学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版,允许论文被查阅和借阅,可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索,可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

控阅的学位论文在解密后也遵守此规定。

□✓公开□控阅(年)

作者签名:导师签名:

签字日期:签字日期:

摘要

众所周知,芯片产业是投入极高、回报极慢的领域,而 RISC-V提供了免费开源、开发周期较短的解决方案。面对国外芯片的生态和专利壁垒,RISC-V有望成为我国自主研制处理器芯片的极好的选择。而在集成度越来越高的今天,面对数千万乃至上亿晶体管的规模,那种"设计硬件原型-实现-评估-改进-再实现"的模式已经无法满足现代设计应用的需求,尤其在芯片后期验证的流程中,对于基础系统软件尤其是操作系统,底层驱动等的适配和验证往往是反馈硬件设计缺陷最频繁的部分,如果不能够提供体系结构模拟器进行辅助验证,并提前进行并行的系统软件开发工作,将极大的延长芯片开发周期.

本文通过在实际 RISC-V 芯片开发过程中总结的经验,设计并实现了一款RISC-V指令集模拟器,能够对具体的硬件设计进行功能模拟,一定程度上能够辅助进行处理器验证,主要功能是脱离硬件平台进行系统软件的移植开发工作,极大的缩短了芯片开发后期的软件移植工作.

本模拟器已在实际的芯片开发过程中承担了系统软件的前期移植工作,在流片之前完成了 linux内核的适配移植,并辅助进行了部分外设的驱动程序开发工作.

关键词:RISC-V;模拟器;PLIC

ABSTRACT

Key Words: University of Science and Technology of China (USTC); Thesis; LATEX   
Template; Bachelor; Master; PhD

目 录

第 1章 绪论 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 1

1.1 系统开发背景 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 1

1.2 国内外发展现状 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 1

1.3 本文主要工作 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 3

1.4 论文的组织结构 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 3

第 2章 相关技术分析ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 5

2.1 指令集架构概述 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 5

2.2 RISC-V架构 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 6

2.3 体系结构模拟器 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 8

2.3.1 解释型指令集模拟 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 10

2.3.2 编译型指令集模拟 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 11

2.4 本章小结 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 12

第 3章 系统需求分析ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 13

3.1 需求导出 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 13

3.2 分析建模 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 13

3.3 非功能性需求 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 15

3.4 本章小结 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 16

第 4章 系统概要设计ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 17

4.1 系统概述 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 17

4.2 系统静态结构 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 17

4.3 系统动态结构 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 18

4.3.1 解码器与指令集功能函数 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 18

4.3.2 指令流程控制 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 18

4.3.3 中断控制器 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 20

4.3.4 交互调试模块 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 21

4.3.5 本章小结 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 21

第 5章 系统详细设计与实现 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 23

5.1 指令集模块的实现 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 23

5.2 CPU和总线的实现 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 25

5.2.1 寄存器模拟 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 25

5.2.2 MMU和缓存模拟 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 26

5.2.3 总线和 I/O模拟 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 28

5.3 中断系统的实现 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 30

5.3.1 PLIC模拟 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 32

5.3.2 RTC模拟 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 34

5.4 调试模块和 UI实现 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 35

5.5 本章小结 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 36

第 6章 系统测试ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 38

6.1 测试概要 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 38

6.1.1 测试环境 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 38

6.1.2 概述 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 38

6.2 测试分析 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 38

6.2.1 模块测试需求获取 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 39

6.2.2 配置项测试需求获取 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 39

6.2.3 系统测试需求获取 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 39

6.3 测试用例设计 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 39

6.3.1 模块测试用例设计 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 40

6.3.2 配置项测试用例设计 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 41

6.3.3 系统测试用例设计 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 42

6.4 测试结果及分析 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 42

6.5 测试结论 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 43

第 7章 结论与展望 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 44

7.1 总结 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 44

7.2 未来工作 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 44

参考文献 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 45

致谢 ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ ꞏ 46

1.1系统开发背景

进入21世纪以后,各国都在积极发展信息安全技术,同时由于知识经济以及人工智能、全球互联时代的到来,集成电路产业在国民经济中发挥着越来越不可替代的作用。我国的集成电路产业经过了几十年的发展,在某些应用领域也取得了不错的成绩,但在高性能处理器以及芯片设计的关键技术中,仍然面临着巨大的挑战。此外,我国的集成电路产业发展还承担着许多来自国际的压力。不管是2018年中美贸易战中"中兴事件"和"华为事件"的发生,还是其他大国对我国实行的禁运政策,都让我国在市场竞争竞争中处于下风。

作为软硬件之间至关重要接口,指令集架构(Instruction Set Architecture, ISA)是处理器的灵魂。RISC-V在众多 ISA中脱颖而出,不仅是因为其后发优势,提出模块化设计的概念,而且因为它开源,包容,提供了高度灵活的配置空间,在积极拥抱开源软件的今天,RISC-V 已经成为未来的主流架构,而随着 RISC-V开源社区的日益壮大,更多的芯片设计厂商选择 RISC-V作为其指令集架构.在集成度越来越高的今天,面对数千万乃至上亿晶体管的规模,那种"设计硬件原型-实现-评估-改进-再实现"的模式已经无法满足现代设计应用的需求[].研究表

明,如果在 CPU的方案论证和设计阶段,没有及时发现问题与瓶颈,将会使后续的工作变得更为困难且代价更加高昂。因此,在开发一个新的体系结构处理器时,为了确保处理器功能特性和性能参数达到设计的预期目标,对体系结构进行验证是一个必不可少的步骤[].除此之外,在芯片开发后期,软硬件适配工作作为测试的重点,往往也需要模拟器环境的支持,厂商可以根据自身产品特性,设计相应的模拟器,并在此基础之上进行软件移植工作和前期软硬件设配工作,以此来提高芯片验证与测试工作的效率。

1.2国内外发展现状

IBM于2012年某研讨会中做了题为"IBM使用模拟器的经验"的报告[],对于 IBM如何在处理器设计过程中使用模拟器进行了介绍.在处理器早期设计研究期间,IBM使用 Mambo []模拟器的时钟精确模式进行微结构探索和粗粒度微结构定义.Mambo模拟器对微结构主要模块和结构进行了模拟,该阶段Mambo由踪迹(trace)驱动,主要运行和研究用户态应用,对处理器的产品竞争力进行

横向比较研究[].在微结构设计实现期间,IBM使用基于公司内部专用"T"语

言编写的时钟精准模拟器M1进行详细模拟处理器微结构[],如图1.1所示.M1是  
图1.1 IBM模拟器框架

以Mambo模拟器或者硬件上抓取的踪迹作为输入,并且可以收集非常详细的微  
结构数据进行性能评估.

为了加速M1模拟器的执行速度,需要对所抓取的踪迹进行取样,同时为了方便调试,M1支持微结构性能数据可视化功能.在处理器验证阶段,IBM使用Mambo []作为处理器验证参考模型辅助进行验证,此阶段 Mambo可以为处理器功能正确性提供参考结果.Mambo模拟了所有处理器的功能特征,把某些性能相关的微结构维护操作(例如 cache维护类指令)翻译成空(nop)操作,对于计算类指令产生准确的结果,并精准追踪处理器寄存器的状态变化,同时支持指令撤销操作,为处理器验证提供参考.IBM基于Mambo模拟器曾发现 PowerPC CPU

的一个控制寄存器存在竞争条件,使得该设计错误在流片之前就被发现并修改[].

在该阶段,IBM还使用自研的由多个 FPGA(field programmable gate array)组成的VHDL(very-high-speed integrated circuithardware description language)仿真加速器Twinstar []进行处理器综合验证.Twinstar是时钟精准的仿真加速器,其推进方式是事件驱动模式,可以对整个处理器芯片进行仿真,以二进制程序作为输入,还支持详细的指令踪迹和处理器状态的实时追踪.该平台运行速度可以达到4MHz并可以运行未经修改的系统软件.类似 Twinstar的验证平台还有帕拉丁[]等.在系统软件开发方面,IBM 基于 Mambo(加速模式)[],Simics [],BGLsim []等多种平台,在流片之前就开始进行固件、操作系统、虚拟机管理器等软件的早期开发.IBM 基于 Mambo模拟器曾开发了 K42操作系统,在芯片可用之后1周

内就启动了操作系统[].IBM 基于 BGLsim-multi []平台和基于 OMNeT++[]开发

的 MARS(message passing interface application replay simulation)模拟平台还可以对机群网络相关的功能进行模拟,模拟器由可执行程序或者踪迹驱动,其中

MARS平台还可以对MPI(message passing interface)类应用进行调优.

此外,比较著名的研究项目还有法国的嵌入式多体系结构模拟器 Qemu []、上海同济大学研制的 PEMU []以及浙江大学开发的WuKong系统[]等。

1.3本文主要工作

为了加快芯片开发项目中的系统软件开发过程,以及辅助进行处理器验证,本文从真实的芯片开发项目工作流程入手,设计并实现了一款针对 RISC-V体系结构的指令集功能模拟器,可以使得芯片开发团队脱离硬件平台进行并发的系统软件移植/开发/测试工作,提供对真实硬件的功能模拟,以及丰富的调试手段,缩  
短软件开发周期,辅助处理器验证工作.主要完成的工作如下:

(1)参照 RISC-V 用户手册,特权级架构手册,以及实际的硬件设计方案,对RISC-V标准拓展指令集共196条汇编指令进行 C++功能函数模拟,其中大部分指令功能可以直接由 RISC-V 指令集手册获取,少部分指令(主要是特权架构指令)参照硬件设计团队具体的 chisel代码进行翻译,对硬件设计的功能框架进行建模,确定模拟器的实现方案,规划具体功能模块的边界.

(2)实现 RISC-V指令集模拟器前后端模块,添加丰富的调试手段,设计并实现平台级中断控制器 PLIC和部分外设模拟,可以直接在模拟器进行外设驱动的开发,结合实际项目需求,添加 UI 可视化界面,进一步加强模拟器的易用性.在实际的芯片开发项目中帮助团队进行系统软件开发工作,并对处理器进行辅助验证.

本人负责的主要工作有进行指令集模块的功能函数设计实现,模拟器后端整体框架的搭建,平台级中断控制器的功能建模与实现,调试交互模块的可视化界面优化,以及测试模拟器是否符合设计需要.

1.4论文的组织结构

本论文总共分为7章,各章节内容安排如下:

第一章:绪论。本章首先简要介绍了 RISC-V指令集架构和体系结构模拟器的背景,然后对国内外主要体系结构模拟器进行举例分析,进而确定了本模拟器的设计思路,最后阐述了本文的主要工作以及论文的组织结构。第二章:相关技术分析。本章对体系结构模拟器的各种技术知识进行分析,包括体系结构各类别和作用,以及指令集模拟策略等相关技术,然后针对 RISC-V指令集架构特性,确定了基于解释型的模拟策略作为本模拟器的设计方案.第三章:系统需求分析。本章主要是采用面向对象分析的方法对模拟器用户的需求进行分析与建模,明确模拟器的功能需求和性能需求,建立系统需求规格说明书。第四章:系统概要设计。本章主要是对 RISC-V指令集模拟器的总体框架进行设计与描述,包括系统的静态软件结构、动态运行流程,确立了模拟器的四个功能模块:指令集模块,CPU和总线模块,中断模块以及调试交互模块。第五章:系统详细设计与实现。本章主要是对 RISC-V指令集模拟器的各个功能模块的具体实现进行阐述,确定每个模块的功能边界、数据结构和接口细节,并进行具体实现。第六章:系统测试与结果分析。本章主要是对 RISC-V指令集模拟器进行单元测试、配置项测试和系统测试,包括测试需求分析和测试用例设计的过程,并依据测试结果,分析模拟器功能和性能的实现情况。第七章:结论与期望。本章对全文工作进行总结,简要地回顾了 RISC-V指令集模拟器的开发背景,介绍了系统的具体设计与实现情况,并指出了当前系统所存在的问题以及未来可能进行改进的方向。

第2章相关技术分析

由于本系统是基于 RISC-V指令集的体系结构模拟器,目标是模拟处理器执行 RISC-V汇编指令前后的软硬件行为,故本章介绍 RISC-V指令集架构相关的内容以及体系结构模拟器的相关技术.

2.1指令集架构概述

指令集架构(Instruction Set Architecture, ISA)是计算机体系结构中定义的软硬件接口规范,是信息技术生态的原始起点。指令集架构的内容包含了基本数据类型,指令集,寄存器,寻址模式,存储体系,中断,异常处理以及外部 I/O。微架构是指处理器的具体硬件实现方法。同一种指令集架构,可以通过实现不同的微架构来设计出性能各异的处理器,其制造成本、功耗、性能可能会有差异,但是软件可以不经过修改直接运行在同种指令集架构的不同处理器上。指令集架构不仅仅是指令的集合,它还定义了处理器的一些硬件信息,比如寻址模式、寄存器设置、存储器、数据类型等等一些需要让程序员了解的硬件信息,以方便程序员进行软件开发。指令集架构主要分为复杂指令集(Complex Instruction Set

Computer, CSIC)和精简指令集(Reduced Instruction Set Computer,RSIC),前者使用不定长的指令设计方案,包含了许多处理器不常使用的复杂指令,使得 CPU设计变得异常复杂,大大增加了硬件设计的时间成本和面积开销;而后者采用定长的指令设计方案,只保留处理器常用指令,因此指令结构较为精简,相应的硬件设计也会比较简洁。随着信息技术的发展,在这几十年间,世界上诞生了许多指令集架构,也消亡了很多指令集架构,现在保留的比较比较知名的有 x86指令集架构和 ARM指令集架构。下面就简单介绍下这两款指令集架构。

x86是由英特尔公司推出的一种复杂指令集架构,采用可变指令长度的设计,经过数代的发展,相继由最初的16位架构发展成如今的64位架构。在 x86诞生初期,CSIC还是业界主流,虽然在之后 RSIC已经取代 CSIC成为主流,但是由于英特尔公司的巨大成功以及为了软件兼容性做出的妥协,x86架构被一直保留了下来。为了克服 CSIC的部分缺点,英特尔也做出了很多优化。例如,Intel采用"微码化"先把复杂的 CISC指令用硬件解码器翻译,变成简单的指令序列,然后再运行,采用流水线的方法,使得即使是 CISC架构的 x86也可以借鉴 RISC架构的优点。但是这样也带来了额外的硬件开销,影响其性能,这是作为 CISC架构的 x86不得不付出的代价。

ARM(Advanced RISC Machines)是一家诞生于英国的处理器设计与软件公

司,其主要业务是设计 ARM架构的处理器,以及向其他 CPU设计制造商提供知识产权(IP),并通过收取专利许可费来获取利润。ARM采用 RISC架构,支持Thumb(16位)/ARM(32位)双指令集,能很好地兼容8位/16位器件(Thumb是 ARM体系结构中一种16位的指令集)。指令执行采用3级流水线/5级流水线技术。寻址方式灵活简单,执行效率高。指令长度固定(在 ARM状态下是32位,在 Thumb状态下是16位)。如今,ARM处理器占领了32位嵌入式处理器的大部分市场,并且是世界上使用最广泛的32位处理器体系结构。来自世界各地的数十家著名的半导体公司都在使用 ARM的授权。然后通过自己的外围电路设计,开发的 ARM处理器可用于许多领域。

2.2 RISC-V架构

第五代精简指令集(RISC-V, Reduced Instruction Set Computer - Five)是由加州大学伯克利分校的 David.Paterson教授团队研发的一套指令集架构,其设计初衷是为了支持计算机体系结构的研究和教育,如今 RISC-V已经成为行业实施的标准免费开源指令集架构.在 RISC-V指令子集中,包括基础指令集和扩展指令集,在使用过程中必须实现基础指令集,而作为 RISC-V的基础指令集- RV32I,仅包含了4种指令、且每种指令的编码方式都非常统一、富有规律,简单明了的  
使用文档也让许多初学者能快速上手[1]。

RISC-V的指令集使用模块化的方式进行组织,每一个模块使用一个英文字符来表示。其中’I’表示的基本整数指令集部分是强制要求实现的部分,其他指令子集部分表示可选的模块。标准拓展的 RISC-V指令集组成如表2.1所示。

表2.1 RISC-V指令集模块

指令集类型类型简写指令数说明

基本指令集

RV32I 47

基本整数指令集,包含算数指令、访存指令、环境调用等指令,具有32个32-bit通用寄存器,32位寻址  
空间

RV32E 47 RV32I指令集简化版本,专为嵌入式设计,与 RV32I   
相比寄存器数量减少为16个

RV64I 59整数指令,32个64-bit通用寄存器,64位寻址空间RV128I 71整数指令,32个128-bit通用寄存器,128位寻址空间

M 8乘除扩展,4条取余数,4条除法,5条乘法操作指令A 11原子扩展,包含原子读-修改-写、原子加减或、异或等F 26单精度浮点扩展,单精度访存、单精度浮点运算指令D 26双精度浮点扩展,双精度访存、双精度浮点运算指令  
Q 26四倍精度浮点指令扩展

C 46压缩指令扩展,16位指令编码,减少静/动态代码大小

其中,RV32和 RV64表示寄存器的位宽,决定了处理器寻址范围的大小。在

该指令集架构中基本整数指令集(Integer)使用简写"I"来表示,该模块包含了整数计算指令、整数加载指令、整数存储指令和控制流指令,实现基本整数指令集是任何一款基于 RISC-V指令集架构的微处理器所必须满足的;标准的整数乘法和除法扩展(Multiply)使用简写"M"表示,指令功能是对整数寄存器中保存的值进行乘法和除法操作;标准的原子指令扩展(Atomic)使用简写"A"表示,添加原子性的读取、修改和写入内存的指令,保证了多核处理器间的访存一致性;单精度浮点扩展(Float)使用简写"F"来表示,该扩展增加了浮点寄存器、单精度计算指令和单精度加载以及存储指令;标准的双精度扩展(Double)使用简写"D"表示,扩展了浮点寄存器,并且增加了双精度计算、加载和存储指令。整数基数集加上四个标准扩展(即"IMAFD")可以缩写为"G",表示实现通用标量指令集。为了提高代码密度,RISC-V架构也提供可选的"压缩"指令子集(Compress),由英文字母"C"表示。压缩指令的指令编码长度为16比特,而普通的非压缩指令的长度为32比特。

RISC-V指令集架构所使用的定点通用寄存器如表2.2所示,浮点通用寄存器如表2.3所示。

表2.2 RISC-V定点通用寄存器

寄存器助记符描述调用与被调用

x0 zero 硬编码为0-

x1 ra 返回地址寄存器调用者

x2 sp 堆栈指针寄存器被调用者

x3 gp 全局指针寄存器-

x4 tp 线程指针寄存器-

x5 t0临时/备用链接寄存器调用者

x6-7 t1-2临时寄存器调用者

x8 s0/fp 保存的寄存器/帧指针被调用者

x9 s1保存的寄存器被调用者

x10-11 a0-1函数参数/返回值寄存器调用者

x12-17 a2-7函数参数寄存器调用者

x18-27 s2-11保存的寄存器被调用者

x28-31 t3-6临时寄存器调用者

表2.3 RISC-V浮点通用寄存器

寄存器助记符描述调用与被调用

f0-7 ft0-7浮点临时寄存器调用者

f8-9 fs0-1浮点保存寄存器被调用者

f10-11 fa0-1浮点参数/返回值寄存器调用者

f12-17 fa2-7浮点参数寄存器调用者

f18-27 fs2-11浮点保存寄存器被调用者

f28-31 ft8-11浮点临时寄存器调用者

RISC-V指令集有以下特点:

(1)一个完全开放的指令集架构,可供学术界和工业界免费使用;

(2)适用于直接本机硬件实现,而不仅仅是模拟或二进制转换的真正的 ISA;

图2.1体系结构模拟器在 CPU开发流程中的作用

(3)避免在微体系结构(例如:顺序、乱序、解耦微处理器)或微技术(例如:全定制、ASIC、FPGA)实现中"过渡架构",并且在这些实现中更有效率的  
一款指令集架构;

(4) RISC-V可以将指令集架构分成一个小的基本整数指令集架构,具备可选的标准扩展,以支持通用软件开发,并且可用于自定义加速器开发或教学;

(5)支持2008年修订的浮点 IEEE-754标准;

(6)一款支持广泛的用户级 ISA扩展和专用变体的指令集架构;

(7)适用于应用程序,操作系统内核以及计算机硬件实现32位或64位地址  
空间变体;

(8)该指令集架构支持包括异构多核处理器在内高度并行多核的实现;

(9)可供用户选则的可变长度指令格式,对可用指令编码空间进行扩展,指令集架构所支持的可选密集指令编码用以提高性能,减小静态代码大小以及提  
升能量效率;

(10)完全可虚拟化的 ISA,可简化虚拟机管理程序开发;

(11)一个简化了新的管理员级和管理程序级的 ISA设计。

2.3体系结构模拟器

模拟器是体系结构量化分析的重要手段,对架构设计、芯片开发有重要的指导作用.基于模拟器辅助进行集成电路设计可以追溯到1980年代,自此模拟器便一直是处理器设计过程中不可或缺的工具.在芯片开发过程中,体系结构模拟

器可以缩短处理器的设计时间,降低开发成本,其具体作用如图2.1所示:

由图1可知:

1)在芯片开发早期,基于模拟器可以进行微结构探索和粗粒度微结构定义,

此时模拟器的开发抽象层次较高.

2)随着处理器设计的不断推进和模拟器的不断完善,基于模拟器可以持续

对芯片微结构进行评估、修改和取舍.

3)当模拟器趋于成熟,可以对微结构、多核互联系统、一致性协议等进行详

细性能分析,基于分析结果对微结构进行微调.

图2.2体系结构模拟器开发流程图

4)在对处理器逻辑设计进行验证的阶段,模拟器可以作为参考模型辅助进

行验证,可以快速定位逻辑设计错误.

5)在未流片之前基于模拟器就可以开展系统软件开发和适配工作,这样可

以在芯片流片结束后以最快速度启动系统软件.

6)流片结束后,基于模拟器可以辅助进行芯片硅后验证环境的搭建以及测试用例编写工作.为了保证模拟器可以顺利辅助进行处理器设计,在整个芯片开发过程中,需要持续对模拟器进行校准,通过持续对比模拟器和寄存器传输层(Register-Transfer Level, RTL)之间的差别,可以互相校准并发现模拟器或者 RTL的设计错误.

体系结构模拟器根据模拟的详细程度可以分为功能模拟器(指令集模拟器ISS,Instruction Set Simulator)和性能模拟器(时钟周期精确模拟器)两类。前者只模拟目标系统的指令集体系结构,比如寄存器状态、指令语义、存储器状态等功能特性;后者除了模拟功能特性外,还模拟目标系统的微体系结构,比如流水线、分支预测、Cache层次等[2]。

模拟开发的基本流程如图2.2所示。进行模拟,首先通过对实际硬件系统建模来将之具体化。建模与具体化的过程中必须保证所建模型的结构与实际硬件系统相近或一致,以确保所建模型的精确性,只有精确度较高的模型才能真实的模拟出硬件系统的行为,最终获得正确的结果。在对硬件系统的建模过程中,需  
图2.3体系结构模拟器在 CPU开发流程中的作用

要考虑所选择的算法是否合适。评价一种算法是否合适的准则在于是否符合模拟的要求和硬件系统的特征。为了保证最终的模拟精度,必须确保所选择算法的精度够高,稳定性够好。选定合适的算法后,进行程序设计,即用程序语言将模型描述出来。待确定程序模型的正确以后,就可以用这个模型来进行模拟实验,得到相应的结果。最后分析模拟结果,结果分析既可以是针对模型本身的数据,对模型本身进行评价或研究,也可以是对模拟的目标系统性能作出评价。对于指令集模拟器 ISS,在设计过程中,需要充分考虑指令集模拟策略以及模拟器驱动方式。指令集模拟策略是 ISS设计的基础,它决定了模拟器的性能,同时也会影响模拟器调试模块的功能实现。因此在 ISS的设计之前,需要首先拟定指令集模拟策略。指令集模拟策略分为两种:基于解释型和基于编译型。

2.3.1解释型指令集模拟

长期以来,由于工作原理简单和模拟精度高,解释型指令集模拟器受到了广泛的关注(如 Gem5、SimpleScalar等)。解释型 ISS最大的特点在于直接将硬件

行为映射到软件[3],从而模拟出真实的硬件环境。由于其对指令进行逐条翻译,使得指令的执行可以很好的被控制。解释型指令集模拟器的工作流程很简单,通常是取指-译码-执行的循环,如图2.3所示。

1)取指:模拟器取出目标程序的单条指令;

2)译码:模拟器对目标指令进行翻译,得到指令对应的功能函数;

3)执行:模拟器执行目标指令所对应的功能函数,完成功能函数中定义的软硬件行为。

解释型 ISS的工作流程使得模拟器设计比较简单,易于建模和实现,且灵活性较好,模拟精度高。由于其对指令进行逐条翻译然后执行,使得模拟器调试木块很容易实现。但是由于模拟器通过软件行为对指令进行逐条译码,相较于真实硬件电路效率太低,所以解释型 ISS的模拟速度一般不是很高。

图2.5体系结构模拟器在 CPU开发流程中的作用

图2.4体系结构模拟器在 CPU开发流程中的作用

第2章相关技术分析

2.3.2编译型指令集模拟

基于编译型的 ISS 通过对译码过程的改进,大幅度提升了模拟的效率,模拟器引入编译的思想,采用一次翻译多次执行的方法,加速译码过程。通过将目标指令翻译成宿主机可识别的指令来完成对目标机状态的模拟。编译型 ISS根据译码过程处于编译还是运行时,又可分为静态编译型指令集模拟器(StaticCompiled ISS)和动态编译型指令集模拟器(Dynamic Compiled ISS)。由 Zhu andGajski 给出的静态编译型指令集模拟器将本处于运行时的指令译码过程转移至编译时,如图2.4所示。目标机二进制代码经编译器编译,之后由代码生成器优化生成宿主机的二进制代码,并最终运行于宿主机。

动态编译型指令集模拟器的典型代表为 Embra[33]及 Shade[34],其工作流  
程如图2.5所示:

其借鉴了 iCache的思想,模拟器取指之后判断该条指令是否是第一次执行,若是,那么对其进行译码,并将译码信息保存到 Cache中,然后执行;若不是,则直接在 Cache中调用该指令的译码信息执行。由于该技术在程序运行时进行指令译码,因此很难进行代码优化。

第2章相关技术分析

2.4本章小结

本章节主要介绍了 RISC-V 指令集架构的特点以及体系结构模拟器的相关技术。对体系结构模拟器的功能和开发流程做了介绍,并对几种常见的模拟器类型作了分析对比,其中详细介绍了指令集模拟器 ISS,对指令集模拟器的模拟流程以及两种指令集模拟策略的优缺点进行了阐述,最终结合 RISC-V指令集架构的特点,拟定了本次模拟器的设计方案:采用基于解释性的指令集模拟策略,结合动态编译型 ISS的译码优化策略,一方面丰富模拟器的调试手段,另一方面也可以提高模拟器执行效率。

第3章系统需求分析

需求分析是软件生产周期中的一个重要环节,本章将采用面向对象分析的方法对体系结构模拟器的需求进行具体分析与建模。明确模拟器所需实现的功能性需求和非功能性需求。

3.1需求导出

在芯片设计及验证的流程中,对于基础系统软件尤其是操作系统,底层驱动等的适配和验证往往是反馈硬件设计缺陷最频繁的部分,这部分的工作不仅是对于前期硬件设计的重要测试,也是后续用户态程序开发的基础。对于系统软件的移植和适配工作,有两种主流方式,一种是在模拟芯片硬件特性的 FPGA开发板上仿真,另一种是通过软件模拟。两种方法各有利弊,FPGA开发板更加接近真实硬件环境,能够获取精确的仿真信号,但是速度相对较慢,并且能够提供的调试信息较少。而模拟器环境下的开发,其运行速度接近宿主机,并且调试方便,虽然信号精度与真实硬件有差异,但是能够在测试的前期反馈大部分的缺陷。所以真实的开发和测试流程一般是先使用模拟器验证,再上 FPGA平台仿真,这样既能够提高开发效率,又不失精度。随着 RISC-V开源社区的日益壮大,更多的芯片设计厂商选择 RISC-V作为其指令集架构,在芯片的验证过程中,软硬件适配工作作为测试的重点,往往需要模拟器环境的支持。当前 RISC-V开源社区提供的模拟器作为教学工具可以帮助的初学者快速上手,但是对于实际的芯片开发团队来说,系统软件的移植和开发工作需要一款符合实际硬件设计的模拟器,对真实的硬件功能进行模拟,同时要能够提供强大的调试工具,方便系统软件的开发迭代。本模拟器主要用于芯片开发项目后期的验证工作和系统软件移植工作,对于一些具体的硬件细节,如流水线,分支预测等不进行模拟,只对处理器功能进行模拟,属于解释型的指令集模拟器。

3.2分析建模

指令集模拟器的主要参与者是进行系统软件开发和移植的程序员,通过对实际芯片开发验证过程的分析和归纳,得出模拟器所需要的主要功能有:

1)设置模拟器启动配置,包括 elf文件路径添加,指令集模块注册,运行模式选择等。

2)模拟器执行流程控制。包括正常运行模式下的 uart串口交互,暂停执行进入调试模式,模拟器重启。

图3.1系统软件开发/移植人员用例图

第3章系统需求分析

3)调试功能。在调试模式下,进行断点设置,内存查询,历史指令查询,单步执行等。

4)模拟外部中断信号发送。

综上所述,用户需求描述表如表3.1所示。

表3.1用户需求描述表

名称参与者说明

模拟器配置并启动系统软件开发/移植程序员设置模拟器启动参数并运行

切换至调试模式系统软件开发/移植程序员模拟器从运行模式切换为调试

模式

切换至运行模式系统软件开发/移植程序员模拟器从调试模式切换为运行

模式

重启模拟器系统软件开发/移植程序员重新加载当前配置项并运行断点设置系统软件开发/移植程序员调试模式下进行断点添加/移除内存查询系统软件开发/移植程序员调试模式下对虚拟地址/物理地  
址内容查询

中断信号发送系统软件开发/移植程序员点击外部中断源按钮,发送对应

的外部中断到 plic

根据用例描述表可以得出系统软件开发/移植程序员的用例图如图3.1所示。

下面分别对系统软件开发/移植程序员的五个主要用例进行详细描述。模拟器配置启动的用例描述如表3.2所示。

表3.2模拟器配置启动用例描述

用例名称模拟器配置启动

用例描述设置模拟器启动参数并运行

触发条件勾选模拟器配置选项,输入 elf文件路径

后置条件模拟器解析配置参数,启动程序

(1)输入 elf文件路径

基本事件流(2)选择启动模式是否为调试模式

(3)其他参数勾选,包括核心数,模拟外设路径等

异常事件流配置参数错误,启动失败

模拟器切换调试模式用例描述如表3.3所示。

表3.3切换调试模式用例描述

用例名称切换调试模式

用例描述模拟器从运行模式切换为调试模式

触发条件点击 run/stop按键

后置条件模拟器进入调试模式/模拟器进入运行模式

基本事件流在运行模式下点击 run/stop按键

异常事件流运行模式下点击 run按键/调试模式下点击 halt按键

模拟器断点设置的用例描述如表3.4所示。

表3.4断点设置用例描述

用例名称断点设置

用例描述调试模式下进行断点添加/移除

触发条件在调试窗口勾选断点类型,输入断点条件,点击"应用"

后置条件

点击 run 进入运行模式,模拟器运行至断点条件触发调

试中断,进入调试模式

(1)调试窗口添加/移除断点

基本事件流(2)程序运行,触发断点

(3)进入调试模式,打印断点信息

异常事件流断点信息填写错误导致无效断点条件

模拟器内存查询的用例描述如表3.5所示。

模拟器 mailbox中断信号发送的用例描述如表3.6所示。

3.3非功能性需求

该指令集模拟器的非功能性需求有:

(1)准确性:体系结构模拟器的首要需求就是准确性,只有准确模拟出真实硬件的行为,才能在模拟器上进行后续的软件开发和移植工作。由于本模拟器的模拟精度在指令级别,不涉及到流水线,乱序执行,分支预测等更细粒度的模拟,因此要求模拟器要和真实硬件在寄存器级别完全一致。

表3.5内存查询用例描述

用例描述调试模式下对虚拟地址/物理地址内容查询

触发条件查询窗口输入内存地址,点击查询

后置条件输出内存对应地址内容

(1)选择地址类型为虚拟地址/物理地址

基本事件流(2)虚拟地址需要指定核心 id

(3)输入16进制地址,点击查询

异常事件流输入无效地址导致访存失败

表3.6 mailbox中断信号发送用例描述

用例名称 mailbox中断信号发送

用例描述点击外部中断源按钮,发送对应的外部中断到 plic

触发条件点击对应中断源的中断信号发送按键后置条件模拟器响应中断,执行中断处理程序  
(1)点击 mailbox中断源发送中断信号

基本事件流(2)弹出 mailbox消息窗口,填写消息,点击发送

(3)系统接受外部中断,执行自定义的中断处理函数,显示在 mailbox窗口  
异常事件流 mailbox中断发送后系统无响应

(2)可靠性:可靠性要求模拟器要能够在使用过程中持续稳定运行,不会因为宿主机上程序的设计缺陷导致模拟过程发生崩溃。如果遇到异常情况,模拟器需要能够在不修改启动配置的情况下重启成功,且模拟过程是可复现的。

(3)实时性:作为一个基于指令集翻译的体系结构模拟器,虽然本身的设计初衷不是为了测试 CPU性能,但是模拟器运行速度至少需要达到10MIPS。

(4)友好性:模拟器需要提供一个结构清晰的可视化界面,调试模式下需要能够及时更新处理器状态,包括寄存器值,历史汇编指令列表等。需要提供便捷的内存查询功能,外部中断模拟发送等。

3.4本章小结

本章主要是采用面向对象分析的方法,在实际的芯片开发项目过程中,对系统软件移植/开发/测试流程进行剖析,确定了指令集模拟器需要实现的功能以及其他的非功能性需求,最终形成了系统的需求规格说明书,为后续的系统概要设计,具体实现以及最后的系统测试工作奠定了基础.

图4.1模拟器整体功能模块图

第4章系统概要设计

4.1系统概述

本系统是针对 RISC-V 芯片开发团队在系统软件开发和移植过程中使用的体系结构模拟器.将编译好的 RISC-V架构可执行代码加载到模拟器上运行,观察执行结果,能够脱离实际硬件平台进行系统软件的调试,也能帮助开发人员及时发现硬件实现可能存在的缺陷,从而提高整个芯片开发过程的效率。

4.2系统静态结构

RISC-V指令集模拟器的整体功能模块如图4.1所示,主要包含四个功能模块:预加载模块,指令流执行模块,调试模块和 UI显示模块。其中,预加载模块包括模拟器参数配置,指令集注册,加载 elf文件功能;指令流执行模块包括了主要的 Hart模拟,中断控制器模拟,内存模拟,外设模拟等功能,是模拟器的主体功能模块;调试模块包括断点设置,内存查询,模拟中断信号发送功能;UI显示模块包括目标层序执行窗口,调试窗口等的可视化界面和模拟器状态查询功能。

指令流执行模块是模拟器的主体功能模块,该模块模拟了单条指令执行过程的硬件行为,包括寄存器,总线,内存,MMU,缓存,通过内存映射的 I/O设备等。模拟出的 RISC-V CPU整体架构如图4.2所示,每个处理器都有独立的寄存器组,内存管理单元,所有处理器共享同一个 iCache,dCache,紧随其后的是  
L2Cache和主存。

图4.2处理器结构图

处理器通过总线和其他内存映射的 I/O设备通信,包括 BootRom,RTC,UART串口控制器,平台级中断控制器,Mailbox等等.

4.3系统动态结构

本模拟器是 trace-accurate的指令集模拟器(功能模拟器),模拟器运行的基本流程如图4.3所示。

首先使用 RISC-V交叉编译工具链将目标程序编译为 RISC-V架构的 ELF文件,然后模拟器解析该 elf文件,将对应的指令流搬运到 bootrom,模拟器在配置启动后为处理器注册指令集,绑定解码器,逐条进行译码,执行。指令译码器完成包括操作数在内的指令信息提取,找到该条指令注册时对应的功能函数,执行该功能函数,然后将更新后的寄存器状态信息,内存状态信息同步到前端 UI显示模块。在模拟器运行的过程中,用户还可以通过前端交互调试窗口来切换模拟器运行模式,设置断点触发条件,进行单步调试,状态查询等操作。

4.3.1解码器与指令集功能函数

RISC-V指令集是模块化的,它的核心是一个名为 RV32I的基础 ISA,可选的标准扩展包括MAFDC,根据应用程序的需要,硬件可以包含或不包含这些扩展。本模拟器实现了特权指令集1.9版本,和用户指令集2.1版本的标准拓展指令集共196条指令的模拟。模拟器预加载时通过解析配置参数选择相应的指令集模块进行注册,并初始化解码器.流程如图4.4所示.

4.3.2指令流程控制

指令的执行,分为取指、译码、执行三个步骤。对于单条指令,在逻辑上这三个步骤是顺序的,同步的。所以对于功能模拟器,仍然可以把实际的流水线设  
图4.3模拟器运行流程图

第4章系统概要设计

计看作是单周期的 CPU。处理器核的功能模拟主要包括存储管理单元MMU,高速缓存,以及寄存器组,这部分硬件功能模块都作为处理器核对象的私有成员;模拟器对象管理公有的内存部分,译码器,以及总线核外设.

MMU的功能模拟主要体现在处理器各个运行状态下的地址翻译功能.在真实的芯片设计中,缓存和MMU是两个独立的硬件模块,但是在功能模拟器中,为了实现的方便,可以将高速缓存模块放到 MMU功能模块中,在逻辑上仍然属于两个独立的功能模块,这样做对于处理器行为的模拟没有影响.由于模拟器对于缓存的模拟只能记录缓存的命中率等信息,并不能够真正起到硬件加速的效果,此外本次设计的模拟器作为功能模拟器并不需要对处理器性能指标进行模拟,所以直接舍弃 L2Cache的模拟,只在MMU模块中设置 iCache和 dCache,后续可以使用哈希的方式尝试进行部分优化.MMU和缓存的概要设计如图4.5所示。

图4.4指令集模块注册流程图

4.3.3中断控制器

在CPU运行过程中,存在两种指令流程,一种是常规的逻辑控制流,包括顺序的指令流和分支跳转;另一种称为异常控制流,用来响应处理器状态的某些变化,表现为中断或异常.之前的章节已将介绍了 CPU指令流程控制的设计,主要是取值/译码/执行的逻辑控制流程,本模拟器的一大设计需求就是要能够模拟器部分外设,从而进行对应驱动程序的开发和测试.大部分的外设都需要通过外部中断的方式与处理器核进行交互,所以本模拟器必须要实现平台级中断控制器 PLIC,下面将介绍终端模块的整体设计.首先,对于中断模块的被动响应方,处理器核对于中断信号的响应需要在单个指令执行周期内完成,所以模拟器在单条汇编指令的取值动作之前将对处理器核的状态寄存器进行检查,判断当前是否有待处理的中断信号,然后完成相应的硬件动作;其次,对于中断的主动发起方,PLIC需要进行多个外部中断源的优先级裁决,控制中断源的使能情况,并对所有处理器核的优先级门限进行筛查,对符合条件的所有中断目标发起外部中断请求.PLIC连接到模拟器的所有处理器核心,对内表现为黑盒,处理器不关心中断控制器的内部逻辑,只针对相应的外部中断信号,以及中断源 id,进行对应的中断响应.PLIC与处理器核的关系如图4.6所示.

图4.5 MMU和缓存工作流程图

第4章系统概要设计

4.3.4交互调试模块

本模拟器的主要功能就是进行目标程序的调试工作,包括 bootloader,linux内核,驱动程序等.因此本模拟器的设计不仅需要提供丰富的调试手段,还要能够提供友好的可视化界面,方便进行目标程序调试,加快系统软件的开发/测试/迭代周期.调试模块的功能集成在上述的处理器指令流程控制模块以及前端 UI模块之中.其中,UI模块的可视化界面提供调试信息的设置,查询功能,后端模拟器指令执行流程中将对设置的调试信息进行匹配,将断点匹配结果,调试查询结果反馈到前端 UI模块,与开发人员进行调试交互.该模块需要提供断点设置,单步执行,寄存器/内存查询等功能.UI显示界面主要包含了断点设置窗口,查询窗口和执行交互窗口.

4.3.5本章小结

本章主要是根据系统的需求规格说明书,对 RISC-V指令集模拟器进行概要设计,确定了本模拟器的四个主要功能模块:指令集模块,CPU和总线模块,中断  
图4.6 PLIC外部中断

第4章系统概要设计

模块,以及交互调试模块.描述了模拟器整体运行流程,并对各模块的功能逻辑进行设计,明确了 RISC-V 指令集模拟器的总体框架,动态流程以及各模块设计方案.

图5.1 RISC-V32I指令格式

第5章系统详细设计与实现

系统的详细设计主要是对概要设计中各个功能模块的实现细节进行阐述,给出具体的设计方案.本章节将分别对模拟器四个主要功能模块的具体设计和实现细节进行阐述和说明.

5.1指令集模块的实现

指令集模块主要实现了两个功能,指令集功能函数的翻译,以及解码器对指令列表的解析和注册.

对应单条汇编指令的取值,译码,执行过程,模拟器从 PC地址读取一条32位的汇编指令,通过解码器进行解码,找到对应的功能函数,然后执行该功能函数.核心的数据结构就是指令类 insn\_t,除了包含32位的 uint 数据成员来保存指令内容,还定义了一系列的接口,方便读取 RISC-V指令格式所定义的指令码,寄存器,立即数等的位域信息.下面详细介绍模拟器对于指令类数据结构的定义和实现.由前面的章节可以知道,RISC-V汇编指令的格式是非常明晰的,图5.1展示了

RISC-V32I所包含的全部六种指令类型的格式。在实际编码过程中,编码位置的安排都是有意义的。例如3个寄存器索引号在不同指令格式中的编码位置是永远不变的,Rd在 bit 7-11,rs1在 bit 15-19,rs2在 bit 20-24。即使有些指令中可能没有用到部分寄存器,比如第二个指令类型 I-type中没有 rs2,但是 rs1和 rd的索引号也在对应的位置上。又例如在 S-type里 funct3在 bit 12-14,与在 R-type中的位置一致。Opcode是所有指令格式都有的,而且位置不变,永远都是 bit 0-6。

所以在指令类 insn\_t中可以定义统一的接口来获取 RISC-V指令中的位域信  
息.代码片段如下:

t ypede f qu i n t 64 i n s n \_ b i t s \_ t ;

c l a s s i n s n \_ t

i n s n \_ t ()= de f au l t ;

i n s n \_ t ( i n s n \_ b i t s \_ t b i t s ): b ( b i t s ){}

i n s n \_ b i t s \_ t b i t s (){ re turn b ;}

i n t l e n g t h (){ re turn i n s n \_ l e n g t h ( b );}

i n t 64\_ t i\_imm (){ re turn i n t 64\_ t ( b )>>20;}

i n t 64\_ t s\_imm (){ re turn x (7,5)+( xs (25,7)<<5);}

i n t 64\_ t sb\_imm (){ re turn ( x (8,4)<<1)+( x (25,6)<<5)+( x (7,1)<<11)+( imm\_sign ()<<12);}

i n t 64\_ t u\_imm (){ re turn i n t 64\_ t ( b )>>12<<12;}

i n t 64\_ t uj\_imm (){ re turn ( x (21,10)<<1)+( x (20,1)<<11)+( x (12,8)<<12)+( imm\_sign ()<<20);

qu i n t 64 rd (){ re turn x (7,5);}

q u i n t 64 r s 1(){ re turn x (15,5);}q u i n t 64 r s 2(){ re turn x (20,5);}q u i n t 64 r s 3(){ re turn x (27,5);}

q u i n t 64 rm (){ re turn x (12,3);}

q u i n t 64 c s r (){ re turn x (20,12);}

i n t 64\_ t rvc\_imm (){ re turn x (2,5)+( xs (12,1)<<5);}i n t 64\_ t rvc\_zimm (){ re turn x (2,5)+( x (12,1)<<5);}

i n t 64\_ t rvc\_addi4spn\_imm (){ re turn ( x (6,1)<<2)+( x (5,1)<<3)+( x (11,2)<<4)+( x (7,4)<<6)

;}

i n t 64\_ t rvc\_addi16sp\_imm (){ re turn ( x (6,1)<<4)+( x (2,1)<<5)+( x (5,1)<<6)+( x (3,2)<<7)

+( xs (12,1)<<9);}

i n t 64\_ t rvc\_lwsp\_imm (){ re turn ( x (4,3)<<2)+( x (12,1)<<5)+( x (2,2)<<6);}i n t 64\_ t rvc\_ldsp\_imm (){ re turn ( x (5,2)<<3)+( x (12,1)<<5)+( x (2,3)<<6);}

i n t 64\_ t rvc\_swsp\_imm (){ re turn ( x (9,4)<<2)+( x (7,2)<<6);}

i n t 64\_ t rvc\_sdsp\_imm (){ re turn ( x (10,3)<<3)+( x (7,3)<<6);}

i n t 64\_ t rvc\_lw\_imm (){ re turn ( x (6,1)<<2)+( x (10,3)<<3)+( x (5,1)<<6);}

i n t 64\_ t rvc\_ld\_imm (){ re turn ( x (10,3)<<3)+( x (5,2)<<6);}

i n t 64\_ t rvc\_j\_imm (){ re turn ( x (3,3)<<1)+( x (11,1)<<4)+( x (2,1)<<5)+( x (7,1)<<6)+( x

(6,1)<<7)+( x (9,2)<<8)+( x (8,1)<<10)+( xs (12,1)<<11);}

i n t 64\_ t rvc\_b\_imm (){ re turn ( x (3,2)<<1)+( x (10,2)<<3)+( x (2,1)<<5)+( x (5,2)<<6)+( xs   
(12,1)<<8);}

i n t 64\_ t rvc\_simm3(){ re turn x (10,3);}

q u i n t 64 r v c\_ r d (){ re turn rd ();}

q u i n t 64 r v c \_ r s 1(){ re turn rd ();}

q u i n t 64 r v c \_ r s 2(){ re turn x (2,5);}

q u i n t 64 r v c \_ r s 1 s (){ re turn 8+ x (7,3);}q u i n t 64 r v c \_ r s 2 s (){ re turn 8+ x (2,3);}

pr i v a t e :

i n s n \_ b i t s \_ t b ;

q u i n t 64 x ( i n t lo , i n t l e n ){ re turn ( b >> l o )&(( i n s n \_ b i t s \_ t (1)<< l e n )−1);}

qu i n t 64 xs ( i n t lo , i n t l e n ){ re turn i n t 64\_ t ( b )<<(64− lo − l e n )>>(64− l e n );}

qu i n t 64 imm\_sign (){ re turn xs (63,1);}

};

我们可以在后续功能函数的实现中使用上述的接口,极大的提高模拟效率.区别于其他指令集架构的设计,RISC-V 的译码过程是比对 opcode 和 func位域信息,通过 riscv-opcodes 工具生成的头文件包含了所有标准指令集模块的指令格式信息,每条汇编指令都包含一对 MASK 和 MATCH 信息,真实的译码过程是将指令内容与 MASK 取位与运算,得到的结果和 MATCH 一致表示是该条汇编指令.在模拟器实现过程中只需要为解码器开辟一块内存空间,存放(MATCH,MASK,insn\_t)的三元组即可,为了加快译码速度,在模拟器设计中,采用了哈希表的数据结构进行存储.具体的实现细节如图所示.

定义了指令数据结构和解码器之后,取指,译码的过程就完成了,接下来介绍执行步骤的核心内容,指令集功能函数的实现.

指令集功能函数是整个指令集模拟的核心部分,理论上说,汇编指令的功能函数需要和实际的硬件设计一一对应,由于硬件设计所参考的指令集架构版本已经定义了各个汇编指令的功能和具体行为,所以对于指令集的功能模拟只需要参照相应的指令集手册.本模拟器的设计过程依托于具体的芯片项目,由于大多数的硬件设计团队针对不同的性能指标往往会进行一些取舍,尤其是对于 RISC-V这样的开源架构,实际的设计肯定会和官方版本有所出入,所以在模拟器的设计  
图5.2状态寄存器位域信息

上还是需要参照硬件设计团队的代码.

本次设计依托的芯片开发项目使用 chisel进行硬件设计,通过其生成的 scala代码,可以指导模拟器指令功能函数的翻译,如图所示为 scala代码到 c语言模拟程序的翻译过程.

5.2 CPU和总线的实现

上一节介绍了指令集模块的实现,从一个较高的抽象层次对模拟器的功能进行了定义,本节将详细介绍硬件的模拟细节,主要包含 CPU内部的各个功能部件,以及 CPU与片外通信的桥梁-总线的设计与实现.

5.2.1寄存器模拟

寄存器是 CPU 预先定义的可以用来存储数据的位置,汇编指令的执行过程主要就是对寄存器和其他存储器的读写过程,因此对于寄存器的模拟需要做到精确且高效.

RISC-V体系结构中,定义了两类寄存器,整数和浮点寄存器 XPR/FPR;控制与状态寄存器(control and status register, CSR).由于前者属于通用数据寄存器,在用户模式和机器模式下的访问方式一致,因此只需要将其定义为处理器类内部的共有成员,可以直接由处理器对象获取并修改.CSR 寄存器主要由特权集指令进行位操作,表示 CPU状态的改变,后续的调试模块也需要重点关注 CSR的状态,因此在模拟实现上需要提供统一的接口,一方面为了简化指令集功能函数的实现,另一方面也可以节省参数传递等过程带来的性能损失.

寄存器类图如图所示.总体设计上,三组寄存器均封装在 processor\_t类内部,其中通用寄存器组可以被处理器类对象直接调用,进行读写操作;CSR 寄存器组被声明为类私有成员,对外提供 get\_csr(),set\_csr()统一接口.将 CSR寄存器访问权限,状态寄存器位域信息获取等操作封装在接口内部,这样就可以使指令集功能函数的实现不必关心具体的寄存器实现细节.对于 CSR 寄存器接口的实现主要需要考虑三个因素,检查寄存器组所需的指令集模块支持;处理器当前权限模式检查;以及状态寄存器写入格式的检查.

以状态寄存器 CSR\_MSTATUS 为例.如图5.2所示,mstatus 寄存器是一个 XLEN 位的可读/可写寄存器,其格式如图所示。mstatus 寄存器持续跟踪和控制硬件线程的当前操作状态。在写 status 寄存器的过程中,需要检查VM,MPP,MPRV,PUM,MXR 位是否有变化,如果上述的位域发生改变,表示处理器状态发生改变,需要在寄存器读写之前判断当前特权等级。如图是汇编指令 mret的定义以及功能函数实现。

r e q u i r e \_ p r i v i l e g e (PRV\_M);

s e t \_ p c ( p−> g e t \_ s t a t e (). mepc );

r e g \_ t s = STATE . ms t a t u s ;

r e g \_ t p r ev\_p rv = g e t \_ f i e l d ( s , MSTATUS\_MPP);

s = s e t \_ f i e l d ( s , MSTATUS\_UIE << prev\_prv , g e t \_ f i e l d ( s , MSTATUS\_MPIE));

s = s e t \_ f i e l d ( s , MSTATUS\_MPIE ,1);

s = s e t \_ f i e l d ( s , MSTATUS\_MPP, PRV\_U);

p−> s e t \_ p r i v i l e g e ( p r ev\_p rv );p−> s e t \_ c s r (CSR\_MSTATUS, s );

在 set\_csr()接口的实现中,需要对于 CSR的读写进行检查,一方面是为了确保读写时所处的特权级模式是否支持读写,另一方面需要针对状态寄存器的改变做出相应的动作,比如 tlb的清除等.

case CSR\_MSTATUS:

{

i f (( v a l ^ CSR . ms t a t u s )&(MSTATUS\_VM | MSTATUS\_MPP | MSTATUS\_MPRV | MSTATUS\_PUM | MSTATUS\_MXR))

mmu−> f l u s h \_ t l b ();

r e g \_ t mask = MSTATUS\_SIE | MSTATUS\_SPIE | MSTATUS\_MIE | MSTATUS\_MPIE

| MSTATUS\_SPP | MSTATUS\_FS | MSTATUS\_MPRV | MSTATUS\_PUM   
| MSTATUS\_MPP | MSTATUS\_MXR ;

i f ( v a l i d a t e \_vm ( max\_xlen , g e t \_ f i e l d ( va l , MSTATUS\_VM)))

mask |= MSTATUS\_VM;

CSR . ms t a t u s =(CSR . ms t a t u s &~mask )|( v a l & mask );

bool d i r t y =(CSR . ms t a t u s & MSTATUS\_FS)== MSTATUS\_FS ;

d i r t y |=(CSR . ms t a t u s & MSTATUS\_XS)== MSTATUS\_XS;

i f ( max\_xlen ==32)

CSR . ms t a t u s = s e t \_ f i e l d (CSR . ms t a tu s , MSTATUS32\_SD , d i r t y );

e l s e

CSR . ms t a t u s = s e t \_ f i e l d (CSR . ms t a tu s , MSTATUS64\_SD , d i r t y );

x l e n = max\_xlen ;

break ;

}

除了上述的寄存器,每个处理器对象都维护私有的 PC程序计数器,初始化过程中 PC被初始化为 bootrom地址.

5.2.2 MMU和缓存模拟

内存管理单元(Memory Management Unit, MMU)是一种负责处理 CPU内存访问请求的计算机硬件。它的功能包括虚拟地址到物理地址的转换(即虚拟内存管理)、内存保护、中央处理器高速缓存的控制等。

在 RISC-V 体系结构中,与 MMU 有关的 CSR 寄存器主要有控制与状态寄存器 mstatus 和管理员页表基址寄存器 sptbr.在 mstatus 状态寄存器中,虚拟化管理字段 VM[4:0]指示了当前活跃的虚拟化方案,包括虚拟存储器翻译和保护。图5.3给出了当前定义好的虚拟化方案。对于一个 RISC-V硬件实现,只有Mbare模式是强制要求的,该模式没有存储器管理或翻译,因此所有的有效地址,无论其特权模式,都被认为是机器物理地址,是复位时进入的模式,理论上在该模式下不需要经过MMU进行地址翻译。Sv39和 Sv48是针对 RV64系统的基于页面  
图5.3 RISC-V虚拟化方案

的虚拟存储器体系结构,提供了一个39位或者48位的虚拟地址空间,被设计成支持现代管理员级操作性,包括基于 Unix的系统。Sv39、Sv48需要实现支持M、S和U特权级。本模拟器支持上述三种虚拟化方案,但是为了实现方便,所有的主存访问请求都需要经过 MMU,通过 MMU模块的统一接口进行访存,当 mstatus寄存器 VM位为0时,无需进行地址翻译.以 linux内核加载过程中从物理地址向虚拟地址过渡的逻辑可以看出,当内核支持MMU时,会进入到 relocate代码段进行页表基地址寄存器的初始化,以后首级页表会常驻内存,内核通过 setup\_vm()函数进行了首级页表的加载,然后在 relocate段计算了首级页表基地址,写入 sptbr寄存器,然后通过一条 mret指令跳转到 U-mode执行,接下来就全是加载虚拟地址了,MMU开始工作.

# i f d e f CONFIG\_MMU

r e l o c a t e :

l i a1, PAGE\_OFFSET

l a a2,\_ s t a r t

sub a1, a1, a2

add ra , ra , a1

l a a2,1 f

add a2, a2, a1

csrw CSR\_TVEC, a2

s r l a2, a0, PAGE\_SHIFT

l i a1, SATP\_MODE

or a2, a2, a1

l a a0, t r ampo l i n e \_ p g \_ d i r   
s r l a0, a0, PAGE\_SHIFT

or a0, a0, a1

s f e n c e . vma

csrw sp t b r , a0

. a l i g n 2

1:

/\* S e t t r a p v e c t o r t o s p i n f o r e v e r t o he l p debug \*/

l a a0,. L seconda ry\_pa rk

csrw CSR\_TVEC, a0

/\* Reload t h e g l o b a l p o i n t e r \*/

. o p t i o n push

. o p t i o n n o r e l a x

l a gp ,\_\_ g l o b a l \_ p o i n t e r $

. o p t i o n pop

csrw sp t b r , a2

s f e n c e . vma

# e n d i f /\* CONFIG\_MMU \*/

在本模拟器的实现中,MMU模块包含了快表 TLB,加速地址翻译,本质上就

是开辟了一片内存用来存放翻译过的地址映射,为了方便起见,只实现了直接映射的 TLB.另外,将 iCache,dCache的功能也一并放到 MMU模块中,不再实现单独的缓存硬件模块,缓存采用只写的方式.这样的设计和真实硬件的差异很大,会导致缓存模拟的不准确.考虑到本模拟器并不进行缓存相关的性能模拟,所以可以忽略这部分的差别,在功能模拟上没有影响.模拟器的存储结构模型如图1.1所示.

MMU模块对取值做特殊处理,取值首先会查找 iCache,当 iCache未命中时,退化为其余类型的访存.访存请求通过模板函数提供的统一的接口 load/store进行请求,通过模板可以忽略具体的数据类型,在功能函数的实现中也可以更加方  
便的使用,直接调用MMU的接口对存储单元进行操作.模板函数定义如下:

template < c l a s s T>

i n l i n e T load ( r e g \_ t add r )

{

i f ( add r &( s i z e o f (T )−1))

throw t r a p \_ t ( t r a p \_ l o a d \_ a d d r e s s \_m i s a l i g n e d , add r );

r e g \_ t vpn = add r >> PGSHIFT ;

i f ( l i k e l y ( t l b \_ l o a d \_ t a g [ vpn % TLB\_ENTRIES ]== vpn ))

re turn \*(T\*)( t l b \_ d a t a [ vpn % TLB\_ENTRIES ]+ add r );

T r e s ;

l o ad\_ s l ow\_pa t h ( addr , s i z e o f (T ),( u i n t 8\_ t \*)&r e s );

re turn r e s ;

}

其中,load\_slow\_path()意味着 TLB miss,需要查找页表,通过页表进行地址翻  
译的伪代码如图所示:

综上,整个MMU和缓存模块的实现流程如图??所示.

5.2.3总线和 I/O模拟

总线是 CPU 与外部设备进行数据交换的桥梁,按照功能划分可以分为地址总线,数据总线和控制总线.本模拟器对总线设备进行了抽象,将总线设计为某一块物理地址区间内的 IO控制器,提供统一接口,处理 CPU和内存映射的 IO设备之间的通信.

总线在模拟器初始化的过程中会根据配置挂载设备,在总线类 bus\_t中维护一个物理地址和设备类对象的 map,模拟器在接收到访存请求时,首先检查物理地址是否是主存地址范围,不在主存地址区间内的 I/O请求都是内存映射的 I/O请求(Memory-Mapped I/O, MMIO),模拟器将调用总线设备的 IO接口,对内存映射的外设进行读写操作.

模拟器根据物理地址划分为主存,和内存映射的 IO 设备,图5.4是 SiFive公司提供的内存映射参考.本模拟器的总线设备需要处理的内存映射空间就是

0x000000000x80000000,提供这块内存区间上的 IO 模拟.总线设备可以挂载各种通过内存映射的 IO设备,某些具有特殊用途的寄存器也能够挂载在总线,比如timecmp寄存器,ipi寄存器.

总线设备的定义如下:

图5.4 SiFive公司的内存映射方案

c l a s s bu s\_ t : pub l i c a b s t r a c t \_ d e v i c e \_ t   
{

pub l i c :

bool l o ad ( r e g \_ t addr , s i z e \_ t l en , u i n t 8\_ t\* by t e s );

bool s t o r e ( r e g \_ t addr , s i z e \_ t l en , cons t u i n t 8\_ t\* by t e s );

void add\_dev i c e ( r e g \_ t addr , a b s t r a c t \_ d e v i c e \_ t\* dev );

void a d d \_ r e g i s t e r ( r e g \_ t addr , a b s t r a c t \_ d e v i c e \_ t\* dev , q i n t 32 o f f s e t );

pr i v a t e :

s t d :: map< r eg\_ t , a b s t r a c t \_ d e v i c e \_ t\*> d e v i c e s ;

s t r u c t d e v i c e \_ r e g

{

d e v i c e \_ r e g (){}

d e v i c e \_ r e g ( a b s t r a c t \_ d e v i c e \_ t \*dev , q i n t 32 o f f s e t ): dev ( dev ), o f f s e t ( o f f s e t ){}

a b s t r a c t \_ d e v i c e \_ t \*dev ;

q i n t 32 o f f s e t ;

};

QMap< r eg\_ t , d ev i c e\_ r eg > r e g s ;

};

bus\_t类型的 load/store属于 mmio接口,首先会检查挂载在总线上的设备或寄存器地址是否匹配,然后才会对具体的设备进行读写操作.下面介绍几种以MMIO方式挂载在总线上的设备模拟.

所有支持 Secure Boot的 CPU都会有一个 bootrom固件.CPU在通电之后执行的第一条指令就在 bootROM的入口。bootROM拥有最高的执行权限,也就是机器模式M-mode。它将初始化 Secure Boot安全机制,加载 Secure Boot Key等密钥、从存储器加载并验证 First Stage Bootloader(FSBL),最后跳转进 FSBL中。本次芯片设计项目也包含了该部分的设计,但是由于涉密,在模拟器设计中并不涉及 FSBL的加载和验证过程,bootrom固件程序仅仅用来跳转置目标程序入口.具体的实现方式包括以下的步骤,首先在配置中读取 bootrom的地址,默认0x1000,

然后初始化 bootrom设备,填写 bootrom固件内容,本模拟器的 bootrom程序内容  
如图所示,包含两条汇编指令:

auipc t0,0x7ffff

图5.5 MMIO请求流程

jr t0

模拟器启动后,PC初始化0x1000,以机器模式执行上述两条指令,跳转到主存起始位置,即目标程序的入口地址.接下来的一系列指令执行由目标程序决定,主要包括 bootloader的 reset vector以及加载 linux内核的过程.

总线对象使用 add\_device接口将物理地址0x1000和 bootrom设备对象的映射保存在字典中.

mmio的通信流程如图5.5所示.

5.3中断系统的实现

在 CPU运行过程中,存在两种指令流程,一种是常规的逻辑控制流,包括顺序的指令流和分支跳转;另一种称为异常控制流,用来响应处理器状态的某些变化,表现为中断或异常.之前的章节已将介绍了 CPU指令控制流程的模拟,其中包括了响应中断的逻辑,CPU 在取值之前检查当前是否有中断信号,根据状态寄存器判断是否响应中断,进入异常控制流逻辑.本节将详细阐述模拟器中断系统的实现.包括两种中断控制器以及部分中断源的模拟.

RISC-V一共有两大类的中断类型:局部中断(Local Interrupts)以及全局中断(Global Inerrupts)。局部中断是指直接与 hart相连的中断,可以直接通过 CSRs当中的 xcause(mcause、scause、ucause)中的值得知中断的类型。在局部中断当中,只有两种标准的中断类型:计时中断(timer)以及软件中断(software)。全局

中断实际上就是外部中断(External Interrupts)。它与 PLIC相连(Platform-Level   
Interrupt Controller,平台级中断控制器)。实际上全局中断在多个硬件线程的情况下最为常用。PLIC用于对外部中断进行仲裁,然后再将仲裁的结果送入核内

的中断控制器。RISC-V架构中规定了一些硬件行为来实现异常事件的响应和处理,这些行为通过控制状态寄存器来反应异常事件信息.涉及到如下几个寄存器,见表5.1.

表5.1中断相关的寄存器

mstatus

状态寄存器,保存全局中断使能以及许多其他的状态。只讨论机器模式的情况下,有效的域只有 MIE 和 MPIE。MIE表示中断全局使能,只有该位有效时,才能产生中断。MPIE 位保存进入中断处理之前以及恢复现场之前MIE的状态。

mie 外部中断使能寄存器,外部中断响应的使能位有效时才

能进入处理

mtvec 异常入口基地址寄存器,可配置为查询或向量访问模式。高位 BASE域表示异常向量的基地址。

mscratch 上下文切换时用于保存当前的指针信息

mepc 异常 PC寄存器,进入中断处理时,硬件会自动将当前遇

到的中断的 PC保存 mepc中

mcause

异常原因寄存器,mcause[31]为中断域,其余位为异常编号域。如果是有中断事件待处理,会将 interrupt位置

1。Exception Code字段保存待处理事件的标识代码。中间位读时返回0,作为保留位可以在之后用于支持异常  
编码的字段扩展

mip 外部中断事件标志信号,当有外部中断事件发生时,mip

中对应中断标志位置高

mbadaddr 外部中断事件标志信号,当有外部中断事件发生时,mip

中对应中断标志位置高

本模拟器实现了这部分的硬件行为,包括中断源的产生,和中断响应,下面进行详细的介绍.在处理器类内定义了两个 uint64型数据,分别用来表示时钟中断和外部中断的 pending,总线上挂载的设备可以直接对这两个 pending数据进行设置,用来模拟外设"异步"的中断请求发起.处理器在下一个取值周期之前,使用try-catch 语句来处理可能的中断响应,如果处理器决定响应中断,就会抛出一个

trap\_t类型的异常,进入到响应中断的异步逻辑.trap\_t的定义如下:

c l a s s t r a p \_ t

{

pub l i c :

t r a p \_ t (){}

t r a p \_ t ( r e g \_ t t yp e ): c au se ( t yp e ), a d d r \_ v a l i d ( f a l s e ){}

t r a p \_ t ( r e g \_ t type , q u i n t 64 add r ): c au se ( t yp e ), a d d r \_ v a l i d ( t rue ), add r ( add r ){}

r e g \_ t c au se ;

bool a d d r \_ v a l i d ;

q u i n t 64 add r ;

s t a t i c QMap< r eg\_ t , QSt r ing > names ;

s t a t i c qu i n t 64 d e l e g a b l e \_ e x c e p t i o n s ;

QS t r i ng &name (){ re turn names [ c au se ];}

处理器响应的流程如图5.6所示.

模拟器抛出异常后,根据捕获到的 trap\_t,进入中断处理流程.RISC-V特权架构对这部分的处理器行为做了一定规范,本模拟器的实现也参照了特权架构定义.

图5.6处理器响应中断流程

第5章系统详细设计与实现

1)根据处理的中断类型将信号源编号记录到 mcause寄存器中;

2)地址不对齐或者发生访问异常,将导致错误的指令部分保存到 mbadaddr;

3)更新状态寄存器 mstatus,记录中断处理前的状态;

4)保存当前 PC到 mepc寄存器中,以便于处理完中断后返回;

5)停止当前执行程序流,设置 PC为 mtvec中断向量表入口地址并开始执行.以上就是中断产生和响应过程对应的处理器硬件行为模拟,接下来将着重阐述平台级中断控制器 PLIC的实现.

5.3.1 PLIC模拟

PLIC的功能是接受外部中断源发出的中断信号,对中断请求进行裁决,将裁决结果和外部中断信号发送给处理器.本模拟器参照了 SiFive公司的 PLIC规范文档进行设计,通过设备树的挂载对内存映射的存储器进行读写,配置 PLIC.图5.7是 SiFive公司给出的 PLIC规范框架图.

中断源(interrupt sources)是挂载在 PLIC 上的设备的统称,一般是 I/O 设  
图5.7 SiFive公司的 PLIC框架

备.PLIC 理论上支持任意多个中断源,每个中断源可以是不同触发类型,电平  
触发或者边沿触发、PLIC为每个中断源分配了如下的信息:

1)闸口(Gateway)和 IP(interrupt pending).闸口控制中断源发送中断请求.IP标识当前中断源是否有请求.IP拉高时,闸口不再接受该中断源的请求.

2)设备编号 ID.PLIC为每个中断源分配了一个独一无二的编号 ID.该 ID也作为在多个 sources具有相同优先级是的选择条件,数值较小优先级更高。

3)优先级(Priority).每个中断源都有一个内存映射的 Priority寄存器,表示该中断源的优先级,软件可以通过配置(设备树)将其设置成不同的优先级.优先级的数字越大,表示优先级越高.

中断使能(Enable).每个中断源均分配了一个内存映射的中断使能寄存器,和优先级一样也可以通过设备树进行配置.IE寄存器配置为0表示该中断源被中断目标屏蔽,反之则是使能.

中断目标(interrupt targets)对应为 RISC-V 处理器核心的各个特权级模式。PLIC 产生的外部中断请求会分别标示在处理器的 mip 寄存器的 meip/seip 位,对应于机器模式和监管模式。每个中断目标都有对应的内存映射的优先级门限(priority threshold)寄存器,只有中断源优先级高于该门限,PLIC才会将中断源 ID发送给对应中断目标。

PLIC Core负责所有中断请求的仲裁和分发.图5.8展示了 PLIC处理外部中断的流程.

将中断源和中断目标分别抽象成 plic\_dev\_info\_t类和 plic\_core\_info\_t类,在外设模拟中,通过 connect\_plic 接口初始化设备类中的 plic\_dev\_info\_t 对象,并

将自身挂载到 PLIC设备上,后续设备的 IO请求可以通过 plic\_dev\_info\_t::raise()

图5.8 PLIC中断处理流程

接口发送到 PLIC 设备,模拟了闸口的功能.PLIC 设备类型 plic\_t 中定义了 ac-

cept\_interrupt 接口用来拉高对应设备的 pengding 寄存器,plic\_t::raise()接口实现了中断信号裁决,将裁决后的中断 ID写入 mclaim和 sclaim寄存器,并且向处理器发送外部中断信号,当处理器响应中断时,会读取这两个寄存器的值,然后进入中断向量表,查找对应的中断处理函数,通常会调用定义在内核 drives目录下的驱动程序.中断处理结束后,处理器会给 PLIC core发送中断完成信息,具体的动作就是对 PLIC中断源指定位置 claim寄存器发送一条 store指令,根据 SiFive公司给出的规范,中断源 ID为 n的的 claim/complete寄存器内存映射位置等于(plic基地址+0x200004+0x1000\*n),因此该位置进行 store请求时直接调用处理器对应plic\_core\_info\_t对象的 gate\_open()接口来打开闸口.此时 plic完成一次外部中断请求周期,可以挂起下一次的中断请求.

根据以上的流程,实现 PLIC中断控制器的设备类图如图5.9所示。

5.3.2 RTC模拟

CLINT(Core-Local Interruptor, PLIC)局部中断控制器是一个存储器地址映射模块,CLINT只负责处理软件中断和时钟中断,因为这两个中断是 RISC-V架构中定义的,经过 CLINT不需要进行任何的仲裁,直接将中断信号写入对应的寄存器内即可。软件中断只需要向 CLINT的MSIP0或者 SSIP0寄存器的最高位写1即可,处理完中断后,将其置为0,这样就能够清除掉软件中断的标志位。计时  
图5.9 PLIC设备类图

器中断作为 riscv内核特有的中断,其用法就是往MTIMECMP或者 STIMECMP中写特定的值,当mtime达到该值时产生中断,此时继续填写特定的 tick就可以继续产生下个中断,反复如此,便可产生周期性的 tick中断。

5.4调试模块和 UI实现

调试模块能够提供断点设置,单步执行,寄存器/内存查询等功能.UI 显示界面主要包含了断点设置窗口,查询窗口和执行交互窗口.本模块使用 Qt 的 UIDesigner 工具进行开发,通过拖拽摆放各种窗口控件并进行属性设置,观察界面整体效果,可以方便地进行可视化界面的设计.另外 QtWidget 库还提供了信号(Signals)和槽(Slots)机制,用于对象间通信,只需要在对应窗口子类化 widgets来添加自定义信号,然后实现自定义的槽函数,连接到该信号即可.模拟器前端整体界面如图5.10所示.

前端 UI窗口和后端模拟器之间的信号定义如下:

enum sim\_cmd

{

sim\_cmd\_pause\_sim ,

sim\_cmd\_run\_sim ,

s im\_cmd\_ run\_ s im\_s i l e n t l y ,

s im\_cmd\_step\_sim ,

s im\_cmd\_rese t\_s im ,

sim\_cmd\_access\_memory ,

s im\_cmd\_se t\_b reaks ,

s im\_cmd\_key\_input ,

s im\_cmd\_mai lbox\_ inpu t

enum window\_cmd

window\_cmd\_update\_reg ,window\_cmd\_update\_mem ,window\_cmd\_sim\_output ,window\_cmd\_gst\_output ,

window\_cmd\_pause\_sim ,

window\_cmd\_update\_mailbox

图5.10模拟器前端整体界面

其中 sim\_cmd\_key\_input 和 sim\_cmd\_mailbox\_input 信号分别用来模拟UART 和 mailbox 外部中断源信号.其余信号用于断点和查询信息的交互,以及模拟器单步运行等流程的控制.

断点设置窗口如图所示,可以添加任意多个断点,断点匹配类型分为如下几种:PC,寄存器,内存,中断类型.当模拟器包含多核时需要指定相应的核心id 号.当模拟器处于调试模式时,中断指令流程,此时可以进行断点设置,内存查询等操作,点击运行按键,模拟器恢复至正常运行模式,在每一次指令执行周期完成后模拟器将对当前断点信息进行检查,一旦匹配上任意断点,将发送 window\_cmd\_sim\_output 信号用于告知触发断点信息,并接着发送 win-dow\_cmd\_pause\_sim 信号,中断指令流程,模拟器进入调试模式.每发送一次单步执行信号,模拟器都需要将寄存器更新信号发送给 UI前端,刷新寄存器窗口.模拟器单步执行的流程如图5.11所示.

5.5本章小结

本章主要在概要设计的基础上对 RISC-V 指令集模拟器四个功能模块的详细设计和实现细节进行了阐述,并根据实际工程项目中使用的设备和具体型号 IP文档,进行了部分外设的功能模拟,验证了模拟器的可用性,最后完成了前后端所有功能模块的代码实现,并给出了相应的演示效果图.

图5.11调试模式下单步执行流程

第5章系统详细设计与实现

6.1测试概要

系统测试是系统开发过程中必不可少的重要阶段,是确保系统开发质量的关键,它贯穿于系统开发的整个过程,尽可能地发掘系统中存在的问题和错误,可以及时地进行纠错,促进系统开发工作高效有质量地进行.

6.1.1测试环境

RISC-V模拟器的测试环境分为软件模拟环境和硬件对照环境,软件模拟环境是系统设计使用到的第三方库和模拟器的宿主机环境,硬件对照环境主要是芯片开发过程中的 FPGA开发板 S2C Single U440平台以及流片后的真实芯片环境.具体的模拟器环境如表6.1所示.

表6.1系统测试环境

名称说明

Ubuntu20.04模拟器的宿主机环境

Qt5 Qt5的 UI类库

BBL(berkele-bootloader) RISC-V官方的 bootloader   
linux-v5.10目标程序 linux内核版本

busybox-1.32.1 linux命令集合

6.1.2概述

本模拟器的测试采用测试驱动开发方式进行,主要过程如下:

(1)根据需求分析中确定的芯片开发团队成员需求和系统概要设计中划分的四个功能模块,采用黑盒法设计对应的测试用例,当各个功能模块代码实现后立刻开始测试.

(2)根据需求分析中的需求规格说明,使用场景法对模拟器进行配置项测试,测试模拟器的业务功能是否满足需求.

(3)最后进行系统测试,重点测试 linux内核移植过程中的 MMU启动,挂载PLIC外设,以及调试功能是否满足后续的系统软件移植开发需求.

6.2测试分析

测试分析主要对模拟器的测试需求进行分析,包含可以直接获取的显式功能性需求和系统中隐含的隐性需求比如模拟器运行速度.在系统开发的不同阶段,测试的需求也有所不同,本节所分析的测试需求有:模块测试需求,配置项测试需求和系统测试需求.下面对主要的测试需求展开分析.

6.2.1模块测试需求获取

根据系统概要设计,本模拟器分为四个功能模块,下面对各个功能模块的测

试需求进行分析:

(1)指令集模块:该模块解析模拟器启动参数,获取指令集配置,对相应的指令集模块进行注册,为后续的指令译码,以及功能函数调用提供所需的数据,依据概要设计的要求,主要测试该模块能否将所需指令集全部注册进解码器,并且在后续译码执行过程中调用正确的功能函数.

(2) CPU和总线模块:该模块用于模拟处理器执行过程中的硬件行为,依据概要设计的要求,主要测试该模块的寄存器读写过程;访存请求涉及到的 MMU行为,包括快表查询,iCache查询,通过页表的地址翻译过程;经过总线的 mmio请求过程.主要测试其正确性.

(3)中断模块:该模块用于模拟中断控制器的软硬件行为,依据概要设计的要求,主要测试通过 PLIC挂载设备的中断请求过程能够被处理器响应,中断控制器能够做到规范文档的功能要求.

6.2.2配置项测试需求获取

根据 RISC-V指令集模拟器的需求规格说明书,对于系统软件开发测试人员的需求进行配置项测试,主要是对模拟器加载目标程序的流程测试,以及后续目标程序的流程控制测试,包括设置断点,单步调试,寄存器/内存查询,外部中断信号发送.系统软件开发测试人员通过对目标程序的流程控制,可以方便地进行软件移植工作,加快系统软件的开发迭代.

6.2.3系统测试需求获取

根据需求规格说明书,本模拟器需要进行系统软件的移植测试.包括了 boot-load和 linux内核,在此移植过程中,对模拟器的整体功能部件进行测试,主要有linux加载过程中的MMU启动,PLIC挂载外设,并将中断源通过设备注册到操作系统.

6.3测试用例设计

针对系统开发的各个阶段,测试用例的设计和采用的设计方法都有所不同,下面对各个阶段的测试用例进行设计.

6.3.1模块测试用例设计

依据模块测试的需求,对 RISC-V指令集模拟器的四个功能模块进行测试,各  
个模块的测试用例设计如下:

(1)指令集模块:根据测试需求,该模块的功能是根据模拟器启动参数加载对应的指令集模块,对指令列表进行注册,将指令格式和对应的功能函数绑定起来,完成解码器的初始化工作,该模块检测到未定义指令模块,会发出告警信息,提示未定义行为.具体的测试用例设计如表6.2所示.

表6.2指令集模块测试用例表

序号输入预期输出说明

1指令集标准拓展

rv64imafdc

模拟器注册指令列表

的全部内容,解码器保

存指令格式和功能函

数的映射

验证功能模块的有效

性

2未定义指令集拓展

rv64jkl

提示未定义的指令集

拓展

验证功能模块的健壮

性

3未定义功能函数的自

定义指令

解码器提示功能函数

未定义

验证功能模块的健壮

性

(2) CPU和总线模块:根据测试需求,该模块的功能是模拟指令执行过程中的硬件行为,包括寄存器,MMU,缓存,内存,IO控制器等,测试目的是验证该模块的硬件行为模拟和真实硬件行为一致.该模块的输入总是已注册的指令,不会有其他未定义输入,主要检测是否达到预期输出,具体的测试用例设计如表6.3所示.

表6.3 CPU和总线测试用例表

序号输入预期输出说明

4汇编指令 ld t01000通用数据寄存器 t0内

容0x1000

验证功能模块的有效

性

5汇编指令 csrw mtevc,

t0

CSR 寄存器 mtevc 被

写入 t0寄存器的值

验证功能模块的有效

性

6机器模式下的内存查

询请求

MMU 显示 Mbare 模

式,不进行地址翻译

验证功能模块的有效

性

7监管模式下的 sv39内

存查询请求

MMU 查询页表,输出

地址翻译过程

验证功能模块的有效

性

和序号7同一地址的

监管模式下的 sv39内

存查询请求

MMU 查询快表,TLB

命中,输出查询内容

验证功能模块的有效

性

(3)中断控制器模块:根据测试需求,该模块的功能是将内存映射的 I/O设备挂载到中断控制器,和处理器进行通信.测试目的为了验证中断控制器能够对外部中断信号源进行有效裁决,并配合处理器完成外部中断的流程,以及测试当外部中断源优先级低于处理器门限寄存器时能否屏蔽该中断源.具体的测试用例设计如表6.4所示.

(4)调试模块和 UI:根据测试需求,该模块的功能是对目标程序运行流程进  
表6.4中断控制器测试用例表

序号输入预期输出说明

9对 UART 的 mmio 请

前端窗口输出请求回

验证功能模块的有效

UART 以优先级2挂

载到 PLIC,处理器机

器模式和监管模式的

门限优先级设置为3

UART 中断源的外部

中断请求

验证功能模块的健壮

性

行控制,切换至调试模式进行断点设置,寄存器/内存查询,单步调试等功能,测试目的是验证该模块通过 UI将调试信号发送给后端模拟器,并进行断点匹配的过程,还有验证寄存器/内存查询的正确性,以及查询无效内存地址给出告警信息的过程.具体测试用例设计如表6.5所示.

表6.5调试模块和 UI测试用例表

序号输入预期输出说明

11调试模式下单步执

行信号

模拟器执行完一条指令

后发出更新信号,前端 UI

刷新寄存器状态

验证功能模块的有效

性

12断点信息 pc core0

0x80000000

核0在 PC=0x80000000

处匹配断点,进入调试模

式,输出断点信息

验证功能模块的有效

性

内存地址

0x80000000查

询信号

0xif80006f

对应汇编指令 j pc+504

表示跳转到 reset\_vector

验证功能模块的有效

性

14内存地址0xffffffff

查询信号

发出警告信息,无效内存

地址

验证功能模块的健壮

性

6.3.2配置项测试用例设计

依据配置项测试需求,采用基于实际业务的场景设计法对 RISC-V指令集模拟器的 RISC-V架构目标程序调试功能进行测试用例设计.该功能设计的基本流有:1.设置断点信息,点击应用后,调试窗口发送断点信号到模拟器后端,添加新的断点信息到处理器断点检测列表;2.删除处理器断点检测列表中的某一项;3.点击单步执行,模拟器进行一次取值,译码,执行周期,更新寄存器状态窗口,输出当前指令到指令历史窗口;4.输入待查询内存地址,点击查询,模拟器进入 MMU访存逻辑,查询当前地址内容;5.点击运行,模拟器进入运行模式,将交互信息输出到交互窗口;该功能的备选流有:1.设置断点信息错误,导致处理器断点检测列表添加失败;2.输入无效地址导致内存查询失败,发出告警信息.基本流和备选流可以组合成各个场景,进而对每个场景设计测试用例,具体的测试用例设计如表6.6所示.

表6.6配置型测试用例表

序号操作预期输出说明

15设置断点信息,点击应

处理器断点检测列表

基本流1,备选流1

16删除某一断断点信息删除成功基本流2

17点击单步执行

寄存器状态更新,指令

历史窗口更新

基本流3

18输入内存地址,点击查

询

对应内存地址的内容基本流4,备选流2

19调试模式下点击运行

更新目标程序的交互

信息

基本流5

6.3.3系统测试用例设计

依据系统测试需求,主要对 RISC-V指令集模拟器进行稳定性测试,容错性测  
试和性能测试,下面分别对各个测试进行测试用例设计:

(1)稳定性测试:依据系统测试需求,模拟器需要在加载给定的目标程序后不间断地稳定运行.因此稳定性测试设计为模拟器加载 linux 内核运行 top 程序三天,预期结果是模拟器可以一直持续运行,并且实时检测模拟 CPU的进程状态.

(2)容错性测试:依据系统测试需求,从一下两个方面进行测试用例设计:加载含有未定义指令的程序,检测程序能否陷入对应的异常(illegal instruction);设置错误的断点信息,检测模拟器在端点检测逻辑是否会发生因设计错误导致的程序崩溃.

(3)性能测试:依据系统测试需求,记录模拟器在断点检测列表为空的状态下,和含有断点检测列表的状态下的 MIPS(Million instruction per-second),和实际的硬件 MIPS 做对比,测试模拟器模拟速度.另外从模拟器进行系统软件调试的功能考虑,记录了在模拟器和真在实硬件平台上的调试周期,进行对照测试,分析模拟器在软件移植/开发/测试流程中的性能.

6.4测试结果及分析

使用各个测试阶段所设计的测试用例,对 RISC-V指令集模拟器展开了单元  
测试,配置项测试和系统测试,具体测试结果如下:

(1)在单元测试中对模拟器的四个功能模块都进行了测试,经过五次回归实验后,各个功能模块实际输出与预期输出达成一致,各个模块的功能和健壮性都得到了验证,达到了需求规格说明书中的需求。

(2)在配置项测试中对模拟器的目标程序调试功能进行测试,各个测试用例的实际输出与预期输出一致,所测业务功能满足设计需求。

(3)在系统测试中对模拟器进行稳定性测试、容错性测试和性能测试.稳定性

图6.1模拟器正常运行 UI界面

测试时系统正常运行情况如图6.1所示;容错性测试达到了预测的结果.从测试结果可以看出吸烟检测系统的可靠性满足需求规格说明书中的需求;通过和真实硬件平台的对照,在系统软件移植开发流程中,模拟器的软件迭代周期要明显比硬件平台效率更高,性能要求也达到了预期的结果.

6.5测试结论

经过一系列的测试,RISC-V指令集模拟器能够长时间稳定运行,进行目标程序的跨平台模拟执行.对于可能的异常行为,模拟器提供了丰富的调试手段,方便芯片开发团队进行问题排查,针对发现的硬件缺陷或者软件错误,及时进行修正,进行下一轮的测试迭代.一方面可以对硅后处理器验证起到辅助验证的作用,一方面也能够加快系统软件的开发速度,脱离硬件进行并行的移植开发工作,大大缩短了系统软件的开发周期.

但是在实际的使用过程中,本模拟器对于后续的驱动程序开发,需要经常对各种外设进行模拟,这部分的模拟主要参照各个设备厂商 IP,对于模拟过程可能产生的错误较难及时发现.此外在芯片开发过程中,需要及时对模拟器进行迭代,修正与真实芯片的功能差异,这就需要软件开发人员和硬件设计人员及时有效地沟通,无法做到职责分离.因此本模拟器还是存在较大的改进空间.

7.1总结

7.2未来工作

参 考 文 献

[1] 胡振波. 手把手教你设计 CPU——RISC-V 处理器篇[M]. 1 版. 北京: 人民邮电出版社,

[2] 喻之斌,金海,邹南海. 计算机体系结构软件模拟技术[J]. 软件学报, 2008(4): 1051-1067.[3] S.DWARKADAS, J.R.JUMP. Execution-driven simulation of mu1tiprocessors:address and

timing analysis[J]. ACMTransactions onModeling and Computer Simulation, 1994, 4(4): 314-

338.

致谢

在研究学习期间,我有幸得到了三位老师的教导,他们是:我的导师,中国科大 XXX研究员,中科院 X昆明动物所马老师以及美国犹他大学的 XXX老师。三位深厚的学术功底,严谨的工作态度和敏锐的科学洞察力使我受益良多。衷心感谢他们多年来给予我的悉心教导和热情帮助。

感谢 XXX 老师在实验方面的指导以及教授的帮助。科大的 XXX 同学和XXX同学参与了部分试验工作,在此深表谢意。