中国科学技术大学

专业硕士学位论文

（专业学位类型）



**RISC-V指令集模拟器的设计与实现**

作者姓名： 王昊

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： 二〇二二年一月二十八日

University of Science and Technology of China

A dissertation for master’s degree

（Professional degree type）



**Design and implementation of RISC-V instruction set simulator**

Wang Hao

Software Engineering

Prof. Wang Zengfu

Prof. Hou Rui

January 28, 2022

Author：

Speciality：

Supervisors：

Advisor:

Finished time:

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：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

中国科学技术大学学位论文授权使用声明

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除后也遵守此规定。

□公开 □控阅（\_\_\_\_年）

作者签名：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 导师签名：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

签字日期：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 签字日期：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# 摘 要

现行计算机编程语言的内存管理存在很多不安全的因素。像C/C++将内存的使用交由程序员处理，这会导致许多与内存相关的安全漏洞，其中控制流劫持是利用内存安全漏洞所实施的一类常见且危害严重的攻击手段。它通过改变程序运行方向而实现攻击，针对这类攻击业界提出了许多相关的防御机制，但是这些防御机制由于自身的局限性和缺点，无法在安全性与性能上达到较好的平衡。如对C/C++语言实现的完全内存安全防御通常会产生较大的性能损耗，而部分地内存防御通常被证明不具有较强的安全保证。

为了解决这个问题一项针对代码重用攻击的防御机制被提出，其中的代码指针完整性（Code Pointer Integrity，CPI）的概念可以在安全性和性能开销之间取得很好的平衡，从而在保证控制流完整性的同时不会造成较大的性能开销。CPI的关键思想是将内存划分为常规区和安全区，并利用安全区来保护与程序控制流相关的数据。在此基础上增加了相关的程序跳转检查，保证了每一个控制流转移的正确性，从而实现对控制流劫持攻击的防御。但是随着相关的深入研究发现，CPI在X86-64架构上利用信息隐藏来隔离安全区的做法会导致较弱的安全性。攻击者可以利用程序中的内存信息泄漏探测到安全区位置，从而容易地对安全区实施攻击。

CPI是一项很有潜力的安全技术，我们在LLVM编译器平台上实现了基于英特尔内存保护扩展（Memory Protection Extensions，MPX）硬件辅助机制的CPI防御模型，同时我们改进安全区隔离方式来防止内存泄漏攻击，对安全区内存页施加一种基于硬件原语的读写保护机制来限制安全区域的读写权限，保证了即使安全区位置泄漏依然可以限制恶意代码的读写，这种强化的安全区隔离方法能够完全阻止对安全区的非法访问。为了验证所提方法的有效性，我们对所提出的相关模型进行了评估和验证。

**关键词：**代码指针完整 控制流劫持 内存保护扩展 信息隐藏

# ABSTRACT

There are many unsafe factors in memory management of current computer programming languages. Leaving the use of memory in the hands of programmers like C/C ++ can lead to many memory-related security vulnerabilities, among which control-flow hijacking is a common and serious attack method to exploit memory security vulnerabilities. It realizes the attack by changing the running direction of the program. Many related defense mechanisms have been put forward in this kind of attack, but these defense mechanisms cannot achieve a good balance between security and performance because of their own limitations and shortcomings. For example, full memory security defense for C/C++ implementations usually results in a high performance loss, while partial memory defense usually proves to be not a strong security guarantee.

To solve this problem, a defense mechanism against code reuse attacks is proposed, in which the concept of Code Pointer Integrity (CPI) can strike a good balance between security and performance costs, so as to ensure the integrity of control flow without causing a large performance cost. The key idea of CPI is to divide memory into regular zones and safe zones, and use the safe zones to protect data related to program control flow. On this basis, related program jump check is added to ensure the correctness of each control flow transfer, so as to realize the defense against control flow hijacking attack. However, further research revealed that CPI's use of information hiding to isolate security zones on x86-64 architecture resulted in weak security. An attacker can exploit memory leaks in a program to detect the location of a security zone, making it easy to attack a security zone.

CPI is a security technology with great potential. We implemented CPI defense model based on Intel MPX hardware assist mechanism on LLVM compiler platform. Meanwhile, we improved security isolation method to prevent memory leak attacks. A read/write protection mechanism based on hardware primitives is applied to the security zone memory pages to limit the read/write permissions of the security zone, ensuring that malicious code can be restricted even if the security zone location is leaked. This enhanced security zone isolation method can completely prevent illegal access to the security zone. In order to verify the

effectiveness of the proposed method, we evaluate and verify the proposed model

**Keywords:** code pointer integrity，control flow hijacking，memory protection extensions，information hiding

目 录

[摘 要 I](#_Toc29816)

[ABSTRACT III](#_Toc6370)

[第1章 绪 论 1](#_Toc29844)

[1.1 研究背景 1](#_Toc9378)

[1.2 国内外发展现状 4](#_Toc12605)

[1.3 本文工作 6](#_Toc17037)

[1.3.1 CPI缺点分析 6](#_Toc229)

[1.3.2 具体工作 8](#_Toc26610)

[1.4 章节内容 9](#_Toc8640)

[第2章 相关技术分析 10](#_Toc26598)

[2.1 LLVM介绍 10](#_Toc14708)

[2.2 CPI原理分析 12](#_Toc7354)

[2.3 英特尔MPX分析 13](#_Toc24394)

[2.4 英特尔MPK的分析 19](#_Toc13796)

[第3章 系统需求分析 21](#_Toc27880)

[3.1 可行性分析 21](#_Toc5715)

[3.1.1 技术可行性 21](#_Toc29150)

[3.1.2 硬件可行性 23](#_Toc20042)

[3.2 功能性分析 23](#_Toc18568)

[3.2.1 功能概述 23](#_Toc21818)

[3.2.2 初始化需求 25](#_Toc32530)

[3.2.3 静态分析需求 25](#_Toc2860)

[3.2.4 动态插桩需求 25](#_Toc7595)

[第4章 系 统 设 计 26](#_Toc1721)

[4.1 设计概述 26](#_Toc23326)

[4.1.1 系统功能模块概述 26](#_Toc7087)

[4.1.2 系统逻辑层次概述 27](#_Toc7023)

[4.2 静态分析模块设计 28](#_Toc10180)

[4.3 动态插桩模块设计 31](#_Toc12339)

[4.3.1 控制流检查 31](#_Toc24080)

[4.3.2 安全区保护锁 33](#_Toc20654)

[第5章 系 统 实 现 35](#_Toc20258)

[5.1 概述 35](#_Toc12302)

[5.2 初始化实现 37](#_Toc6888)

[5.3 静态分析实现 38](#_Toc2606)

[5.4 控制流检查实现 40](#_Toc25293)

[5.5 安全区保护锁实现 45](#_Toc972)

[第6章 系 统 测 试 47](#_Toc1454)

[6.1 测试概述 47](#_Toc24856)

[6.2 测试方案 47](#_Toc15514)

[6.3 正确性分析 48](#_Toc28160)

[6.4 测试数据 49](#_Toc9853)

[第7章 结论与展望 53](#_Toc3500)

[7.1 总结 53](#_Toc540)

[7.2 未来工作 53](#_Toc21251)

[参 考 文 献 54](#_Toc4375)

[致 谢 58](#_Toc21042)

# 第1章 绪 论

## 1.1 系统开发背景

## 1.2 国内外发展现状

## 1.3 本文的主要工作

### 1.3.1 硬件设计与验证中的软件仿真

### 1.3.2 具体工作

## 1.4 章节内容

# 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,故本章首先介绍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 RISC-V介绍

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

体系结构模拟器理论基础，硬件仿真器、软件模拟器，模拟开发流程图，解释型模拟spike/Gem5(取值译码执行流程图)、编译型模拟qemu(流程图)，RISC-V简介，相关开源社区工具介绍。

第五代精简指令集（RISC-V，Reduced Instruction Set Computer - Five）是加州大学伯克利分校研发的一款新的指令集架构，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 已然成为行业实施的标准免费开源指令集架构[22]。其特点如下：

(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；

(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；

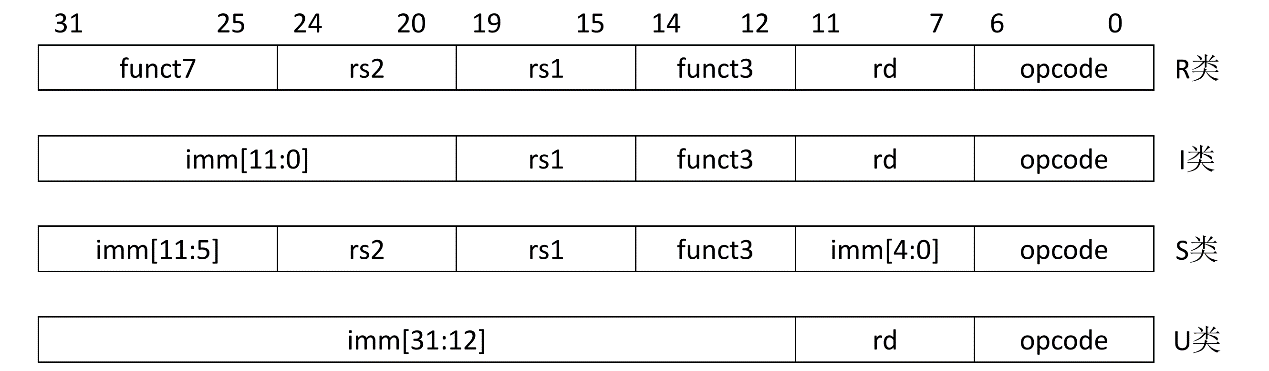
(10)完全可虚拟化的 ISA，可简化虚拟机管理程序开发；

(11)一个简化了新的管理员级和管理程序级的 ISA 设计。

RISC-V 指令集架构 ISA 定义为基本整数指令集架构，以及基本指令集架构的可选扩展，存在于任何以该指令集实现的硬件中。除了没有分支延迟槽并支持可选的可变长度指令编码外，RISC-V 基本整数指令集架构与早期 RISC 处理器非常相似。该基础被严格限制为一组最小的指令集并且足以为编译器、汇编器、链接器和操作系统提供合理的目标，提供了方便的 ISA 和软件工具链，是一款可以构建更多定制处理器的指令集架构。

RISC-V 是一个典型的三操作数、以加载-存储为访存形式的精简指令集架构。仅有 load 和 store 两类指令与外设存储地址交互数据，所以在指令的实现上相对简单，避免了大量的指令访问存储系统造成不必要的访问时间，这也是使用 RISC-V 设计的处理器高速化的一个重要因素[22-24]。指令集组成如表 2.1 所示。

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |
| 扩展指令集 | 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，名称前缀中 RV32 和 RV64 表示处理器使用 RISC-V 指令集同时表明处理器中寄存器的位宽。在该指令集架构中基本整数指令集使用简写“I”来表示 Integer，基本整数集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使用简写“M”表示 Multiply，并添加指令以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使用简写“A”表示 Atomic，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；单精度浮点扩展使用简写“F”来表示 Float，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；标准的双精度扩展使用简写“D”表示 Double，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。

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母来表示。RISC-V最基本也是唯一强制要求实现的指令集部分是由I字母表示的基本整数指令子集，使用该整数指令子集，便能够实现完整的软件编译器。其他的指令子集部分均为可选的模块，具有代表性的模块包括M/A/F/D/C，如表1所示。

为了提高代码密度，RISC-V架构也提供可选的“压缩”指令子集，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。以上这些模块的一个特定组合“IMAFD”，也被称为“通用”组合，由英文字母G表示。因此RV32G表示RV32IMAFD，同理RV64G表示RV64IMAFD。

为了进一步减少面积，RISC-V架构还提供一种“嵌入式”架构，由英文字母E表示。该架构主要用于追求极低面积与功耗的深嵌入式场景。该架构仅需要支持16个通用整数寄存器，而非嵌入式的普通架构则需要支持32个通用整数寄存器。

通过以上的模块化指令集，能够选择不同的组合来满足不同的应用。譬如，追求小面积低功耗的嵌入式场景可以选择使用RV32EC架构；而大型的64位架构则可以选择RV64G。

如图 2.1 所示，RISC-V 基本指令格式有 R/I/S/U 四种及其变种类型。所有固定的32 位指令在内存中必须 4-byte 对齐。其中使用 I 类型指令格式的指令有：整型计算指令中的寄存器-立即数操作指令、左移右移指令、直接跳转指令、Load 指令以及系统调用指令。使用 R 类型指令格式的有：整型计算指令中的寄存器-寄存器操作指令、扩展原子类指令、单精度浮点算数指令。使用 S 类型指令格式的有：Store 类指令。使用 U 类型指令格式的有：LUI 以及 AUIPC 指令。

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。

|  |  |  |  |
| --- | --- | --- | --- |
| 寄存器 | 助记符 | 描述 | 调用与被调用 |
| x0  x1  x2  x3  x4  x5  x6-7  x8  x9  x10-11  x12-17  x18-27  x28-31 | zero  ra  sp  gp  tp  t0  t1-2  s0/fp  s1  a0-1  a2-7  s2-11  t3-6 | 硬编码为 0  返回地址寄存器  堆栈指针寄存器  全局指针寄存器  线程指针寄存器  临时/备用链接寄存器  临时寄存器  保存的寄存器/帧指针  保存的寄存器  函数参数/返回值寄存器  函数参数寄存器  保存的寄存器  临时寄存器 | -  调用者  被调用者  -  -  调用者  调用者  被调用者  被调用者  调用者  调用者  被调用者  调用者 |

|  |  |  |  |
| --- | --- | --- | --- |
| 寄存器 | 助记符 | 描述 | 调用与被调用 |
| f0-7  f8-9  f10-11  f12-17  f18-27  f28-31 | ft0-7  fs0-1  fa0-1  fa2-7  fs2-11  ft8-11 | 浮点临时寄存器  浮点保存寄存器  浮点参数/返回值寄存器  浮点参数寄存器  浮点保存寄存器  浮点临时寄存器 | 调用者  被调用者  调用者  调用者  被调用者  调用者 |

## 2.2 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．
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.

模拟开发的基本流程如图2.1 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。

体系结构模拟器的驱动方式主要分为两种：执行驱动(又叫程序驱动)和踪迹驱动,这两种驱动方式对应的指令集模拟策略分别是编译型模拟和解释性模拟.

加州大学伯克利分校设计的开放指令集 ＲＩＳＣ－Ｖ 已成为处理器设计和研究领域的 热 门，为 了 实现处理器 的 敏 捷 开 发，伯 克 利 分 校 还 开 发 了 高 度参数化的硬件构建语言 Ｃｈｉｓｅｌ［５５］．使用 Ｃｈｉｓｅｌ语言设计处理 器，可 以 直 接 使 用 面 向 对 象 的 设 计 方 法学描述处 理 器 功 能，这 与 传 统 意 义 上 开 发 周 期 精确模拟 器 的 方 式 很 像，但特别之处在于：通过编写一次硬件代码可 以生成 包含 Ｃ＋＋ 时钟精准模拟器、ＦＰＧＡ　Ｖｅｒｉｌｏｇ和 ＡＳＩＣ　Ｖｅｒｉｌｏｇ这３个目标［５５］．具体地在生成模拟器方面，基 于最新 Ｃｈｉｓｅｌ３ 编写的硬件 代 码 可 以 产 生 Ｆｉｒｒｔｌ中 间 描 述 语 言 （ｉｎｔｅｒ－ｍｅｄｉａｔｅ　ｒｅｐｒｅｓｅｎｔａｔｉｏｎ，ＩＲ），从 Ｆｉｒｒｔｌ可以直接翻译或者转换成 Ｖｅｒｉｌｏｇ，进而通过 Ｖｅｒｉｌａｔｏｒ工具可以生成时钟精准的 Ｃ＋＋模拟器和测试框架．但是，通过这种方式生成的模拟器代码可读性和可修改性都比较差．Chisel语言使逻辑设计和模拟器开发得到了统一，使得处理器设计效率提高一个数量级［５６］，因此可以大幅加速硬件设计，这是新的硬件敏捷开发方法学，同时也是未来处理器设计和模拟器发展的一个重要方向．

### 2.2.1 执行驱动模拟

随着体系结构的研究越来越深入，其设计的复杂度越来越高，这给模拟器的性能带来了一定的负面影响。为了提高模拟性能，研究者们对模拟过程进行了改进，引入了编译的思想——采用一次翻译多次执行的方法来缩短模拟过程中指令译码的时间。译码就是将目标指令翻译成宿主机可识别的指令，宿主机通过这些指令来完成对目标机状态的操作。根据译码过程处于编译还是运行时，编译型指令集模拟器又可分为静态编译型指令集模拟器（Static Compiled ISS）和动态编译型指令集模拟器（Dynamic Compiled ISS）。 由 Zhu and Gajski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 2.5 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。

该技术有一个缺点，那就是要求程序代码为静态的，这就使得静态编译技术只能为一小部分数字信号处理器所使用。与典型的 DSP（数字信号处理器）应用不同的是，微处理器通常面向的是完整的操作系统，而操作系统有一个重要的特点——程序代码在运行时是动态的，这就局限了静态编译型模拟器的广泛使用。

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2.6 所示：

其基本思想是复用译码信息，即在内存中取出一条指令后，判断该条指令是否是第一次执行，若是，那么对其进行译码，并在译码完成后将译码信息保存到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。

对于执行驱动模拟，存在两种不同的观点：一种认为执行驱动模拟器输入的是程序的二进制文件（可执行指令），而不是踪迹信息，因此，模拟器的输入集大小只与程序的静态指令数相关，而与动态指令数无关。如图 2.3(a)所示，由模拟器来执行程序的所有可执行指令。还有一种观点认为，执行驱动模拟器的输入信息（程序的可执行指令）分成两部分：一部分由宿主机代为执行，另一部分则由模拟器执行。如图 2.3(b)所示，这类模拟器只执行与所研究问题相关的指令，其它指令则由宿主机代替执行。这样做的优点在于模拟速度较快，但是这种模拟方式要求宿主机与目标机的指令集体系结构一致，这就限制了它的应用范围。

执行驱动模拟器将用高级语言编写的应用程序经编译和链接后得到的二进制文件作为输入信息，在模拟过程中会模拟系统的动态特征，如动态指令的生成和分支预测等，因此更加接近目标系统的真实情况，精确度更高。然而，也正是由于需要模拟目标系统的动态特征，使得执行驱动模拟器的模拟速度较踪迹驱动模拟器更慢。

### 2.2.2 踪迹驱动模拟

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[31]，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（fetch）－译码（decode）－执行（execute）的循环，如图 2.4 所示。

1. 取指：从模拟器的内存空间中取出一条指令.
2. 译码：对取出的指令进行翻译，得到指令的操作码、操作数、执行函数等信息.

3) 执行：执行指令所对应的执行函数，并修改相应寄存器的值，包括 PC 寄存器.

解释执行的工作流程使得解释型 ISS 设计原理比较简单，易于设计和实现，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，使得其可以很容易的实现调试机制。但是也正是由于需要对指令进行逐条翻译，导致模拟器需要在译码阶段花费大量的时间，所以解释型指令集模拟器的模拟速度一般不是很高。

踪迹驱动模拟将每条指令顺序执行所产生的所有信息作为模拟器的输入，从而模拟某种体系结构处理器的功能和性能。其模拟过程如图 2.2 所示。

假设要模拟的目标系统为 T，目标系统上执行的应用程序为 w，则模拟 T 中 w执行的问题可以分解为两个部分：生成踪迹信息以及输入踪迹信息进行模拟。如果把生成踪迹信息的系统命名为 G，那么，踪迹信息就是 G 执行 w 时产生的所有信息。接下来将踪迹信息作为模拟器的输入，在目标系统 T 中模拟执行 w。踪迹驱动的优点在于比较简单，而且可以不关心其它无关部分而只对局部进行详细的模拟。但是这种方式要求生成踪迹的系统 G与模拟的目标系统 T的体系结构相似，否则由 G 生成的踪迹信息不能正确反映 w 在 T 上的实际运行情况，从而导致获得错误的模拟结果。踪迹驱动的另外一个缺点在于模拟器的输入信息是静态的，不能对目标系统的动态特征进行研究。

## 2.3 本章小结

# 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

近几十年来信息技术飞速发展，对芯片的各种要求越来越高，各种新型处理器不断面世。当今社会是以芯片为基石的信息社会，芯片的开发关系到信息生活和服务的各个方面。芯片的设计需要考虑其是否能够具有足够的市场竞争力，这与支持这款芯片的指令集架构的软件种类数量密切相关。例如，Inter x86 架构的服务器和个人主机，在经过多年的发展，如今在商用和家用领域具有很高的市场占有率，其配套的软件的种类和数量在这么多年发展下，也已经十分丰富[1]。考虑到成本和利润因素，芯片厂家在设计新的芯片时偏向于选用具有成熟生态环境的指令集架构。

中国 CPU 设计行业起步较晚，即使经过几十年的发展，它们仍未占据技术的制高点，自主指令集架构处理器的生态环境还没有完善。国内也还没有相对成熟的指令集架构，很多芯片的设计需要靠国外的技术授权，这不仅极大的提高芯片设计的成本，还会受到授权厂商的制约，市场竞争力不强，而且信息安全也有很大的问题。所有这些都极大地限制了国产自主架构处理器的应用和推广。而且，芯片设计属于核心技术，与国家战略有关，国家和科研机构都迫切需要采用独立自主的指令集架构的处理器，来消除中国在电子信息领域和国防信息安全等方面的问题。

Fpga/流片上的系统软件移植测试：软件交叉编译（vmlinux 5min），粘贴fsbl，vivado平台烧录（15min）/流片验证就是烧写平台（20min），xilinx SoC，启动bbl，内核启动，然后只有单步命令行调试，或者jtag。

软硬件行为的模拟，忽略一些指标如缓存加速等硬件性能指标类的测试。主要关注寄存器状态，内存状态，中断系统工作情况，调试交互信息等。

功能性需求：前端设计(gui、text/寄存器/内存/pc指令流、交互/设置断点/中断下发/内存查询/调试过程、保存快照)；指令集注册/解析(riscv-opcodes)；单条指令step前后的寄存器/内存状态转移(指令对应的功能函数/主要的行为模拟)/即指令的单步执行模拟(这部分是重点，包括寄存器和其他存储部件的模拟)/指令流执行的驱动方式；中断系统模拟(clint时钟/软中断；plic外部中断)；调试模块的设计，主要是在功能函数前后进行断点检查(针对寄存器/内存状态的检查)

非功能性需求：可拓展性，UI易用性，速度，模拟精度，代码膨胀率

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，当前开源社区的指令集模拟器spike由SiFive公司维护，能够定期地更新RISC-V的最新特权级指令，是各厂商优先考虑使用的基础模拟器。厂商可以根据自身产品特性，拓展spike支持的设备，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。因此，模拟器的优化与拓展，应该在硬件层面上与待验证芯片的硬件特性相符合，又要在软件层面上满足调试功能易用性的要求。

RTL级，register transfer level，指的是用寄存器这一级别的描述方式来描述电路的数据流方式，rtl级可以理解为，可以直接给综合工具生成你要的网表的代码。

目前RISC-V开源社区的指令集模拟器是spike，由SiFive公司维护，spike模拟实际代码执⾏过程中的软硬件⾏为，提供指令级别的仿真，本质上就是C++程序模拟每条指令执行过程中的软硬件行为，当前最新版本的spike模拟的设备包括总线，内存，时钟，处理器，和调试模块，这些部分能够使得spike运行一个简单的代理内核，并运行一些简单的RISC-V指令集架构的程序。但是对于真正进行RISC-V芯片设计工作的团队来说，spike模拟的内容还远远不够。真实的设计要复杂得多。

spike模拟器的运行过程包括三个部分，模拟器自身部分，面向开发者(用户)的串口调试部分，以及目标程序部分。目标程序一般指的是移植到RISC-V平台的通用基础软件，如linux内核，处理器所支持外设的驱动程序等，这部分程序涉及到的资源就是模拟器应该支持的部分。因此，本次课题涉及到的需求分为两个部分，一个就是对上述功能模块提供模拟器支持的需求，另一部分是优化串口调试模块的需求。

## 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.
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.

综上所述可以得出如表1.1所示的用户需求描述表.

|  |  |  |
| --- | --- | --- |
| 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1.1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.

|  |  |
| --- | --- |
| 用例名称 | 模拟器配置并启动 |
| 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 |
| 基本事件流 | 1. 输入elf文件路径 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 |

|  |  |
| --- | --- |
| 用例名称 | 切换至调试模式 |
| 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 |

|  |  |
| --- | --- |
| 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 |
| 基本事件流 | 1. 调试窗口添加/移除断点 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 |

|  |  |
| --- | --- |
| 用例名称 | 内存查询 |
| 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 |
| 基本事件流 | 选择地址类型为虚拟地址/物理地址,虚拟地址需要指定核心,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 |

|  |  |
| --- | --- |
| 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 |
| 基本事件流 | 1. 点击mailbox中断源发送中断信号 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 |

### 3.2.1 串口调试模块

spike本身具备单步调试的功能，但是在模拟真实硬件行为的复杂场景下，命令行式的单步调试显然难以使用，根据以往的调试经验，人性化的UI设计需要涉及到处理器状态寄存器，当前特权级模式，程序运行窗口等信息，还需要设置对应多核的寄存器触发条件，实现mailbox中断主动触发，保存快照方便复现bug等功能。

这部分的功能需求主要有:

1. 对于多核心独立csr寄存器触发条件的支持
2. 对内存/指令格式等触发条件的支持
3. 能够查询历史指令序列执行情况
4. 实时监测状态寄存器，当前特权级模式
5. 主动发送mailbox中断信号

### 3.2.2 平台级中断控制器PLIC

RISC-V核心(hart)包含local中断源和global中断源。只有global中断源可以被PLIC Core响应，通常为I/O设备，PLIC负责将中断源(global interrupt sources)链接到中断对象(interrupt targets)，也就是处理器核心。spike自身不支持PLIC设备，所以对于外部中断的支持只能通过轮询来实现，效率相当低下，并且不符合实际处理器的特性。因此，首先对spike的拓展便是PLIC的支持，对于uart，spi等串口控制器，可以在后续通过设备树直接挂载为PLIC的中断源。多个外设作为独立的中断源，通过PLIC Core仲裁，将外部中断信号传递给相应的核心。PLIC中断控制流程如图1所示。

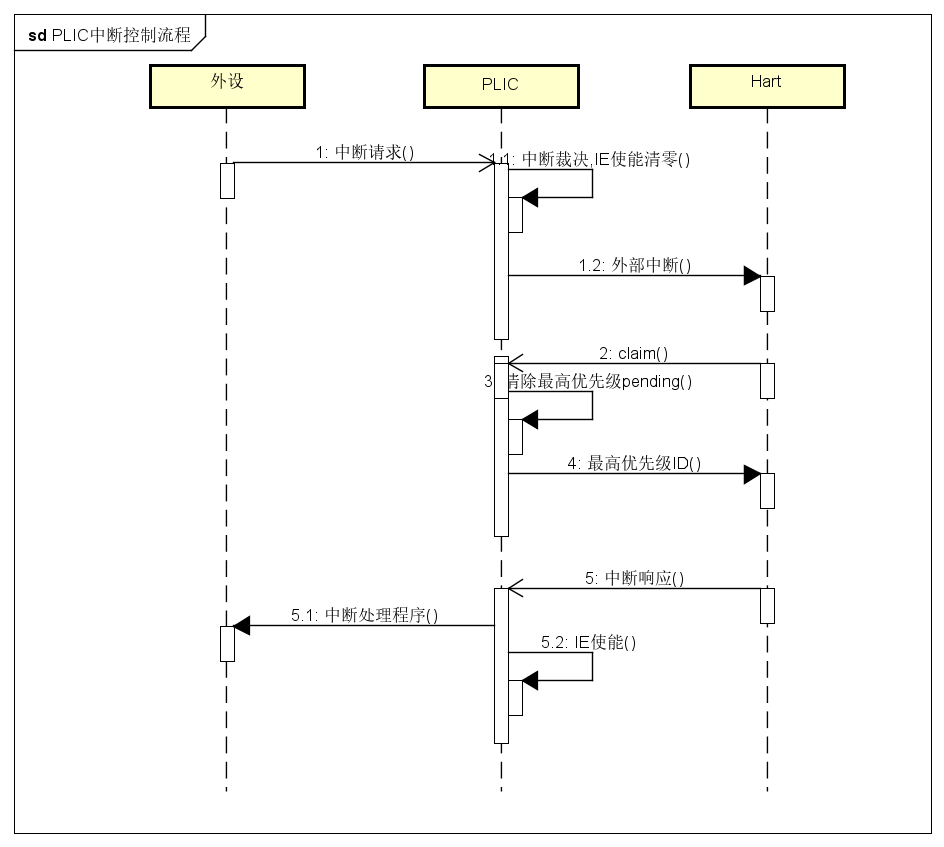


图1 PLIC中断控制流程

对PLIC设备模块的需求是，模拟器启动linux内核，并且uart以中断号2成功挂载到PLIC，通过查看/proc/interrupts进行验证，具体的设计见系统概要设计部分。

### 3.2.3 uart16550串口控制器

spike提供HTIF模拟串口通信控制器, HTIF 是伯克利处理器的非标准工具，因此没有文档。随着 RISC-V 平台规范的发布和内核的更新为自托管，HTIF很快就会消失。HTIF 是主机/目标接口，它通过riscv-fesvr与目标设计（Sodor）通信。riscv-fesvr 通过 HTIF mem 端口将二进制文件加载到 Sodor 内存中，然后通过状态寄存器告知内核。程序完成后，Sodor 告诉 riscv-fesvr，它通过重置主机 CSR 完成交互，模拟结束。

在bootloader加载内核之前,串口通信控制器就必须能够工作,提供裸机的交互功能,是前期进行串口调试的重要工具,能够结合spike的单步调试功能完成前期的大部分调试工作,其实现的优先级很高。

总体来说,HTIF提供了串口通信的功能,但是其本质上还是需要主机的轮询查询,效率很低,考虑到实际硬件设计中必不可少的串口通信控制模块(uart16550a芯片),需要为spike添加uart模拟,通过外部中断的方式实现串口通信.也可以在模拟器上完成对uart的前期测试,包括波特率的设置,传输模式的选择等等.

### 3.2.4 mailbox核间通信模块

由于本次设计涉及到的芯片是一款主动安全处理器芯片,通过和其他厂商计算核心的互联发挥作用,核间的mailbox通信部分至关重要. 区别于ipi处理器内部中断方式, mailbox是一种框架，通过消息队列和中断驱动信号处理多处理器间的通讯.

对mailbox的模拟,需要满足高并发的异步通信需求,鉴于无法真实模拟与另一个计算核心的通信, 需要对模拟器核心间通信添加mailbox支持, 来模拟真实的交互环境.

## 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.
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
4. 友好性:

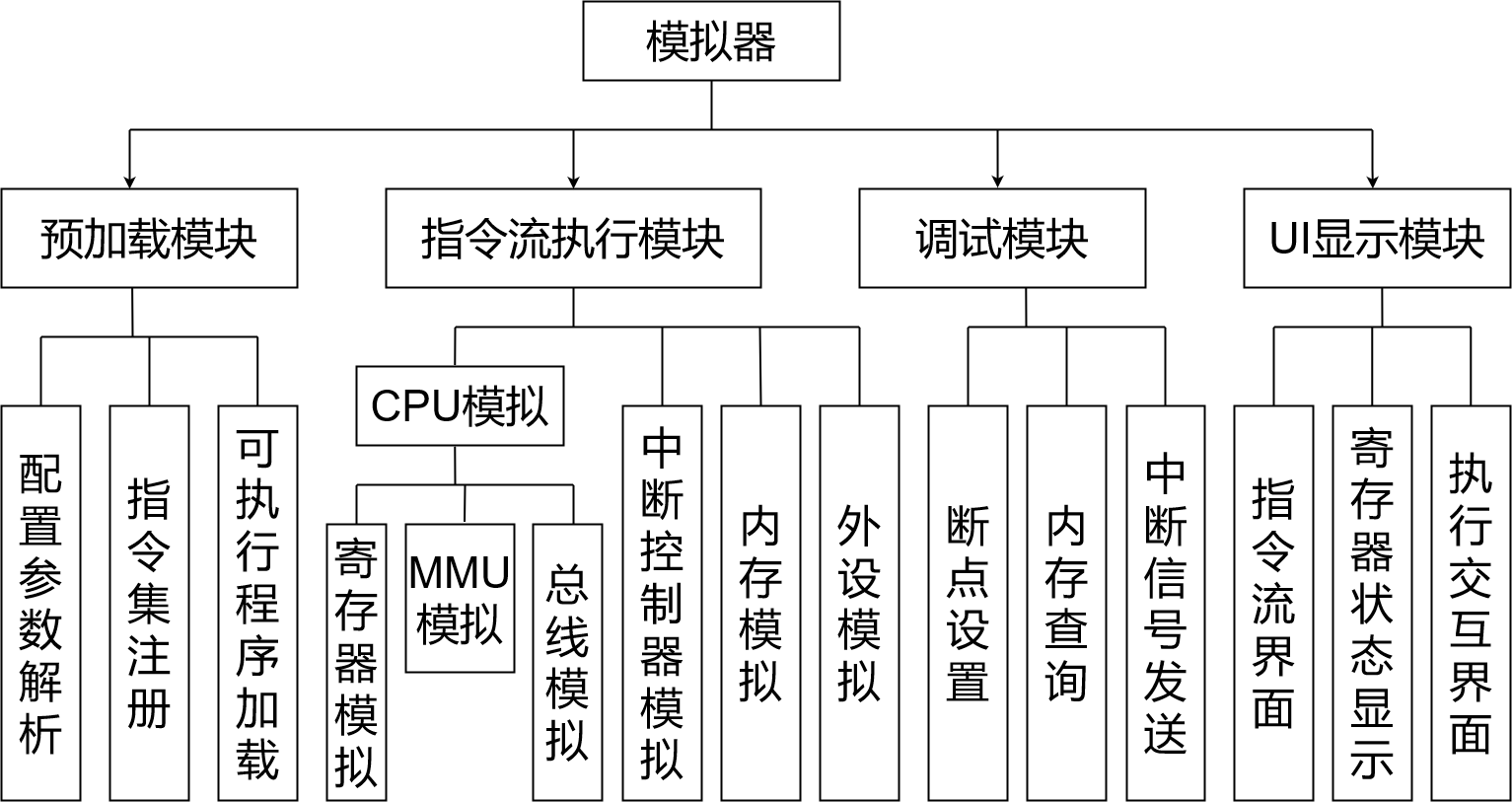
# 第4章 系统概要设计

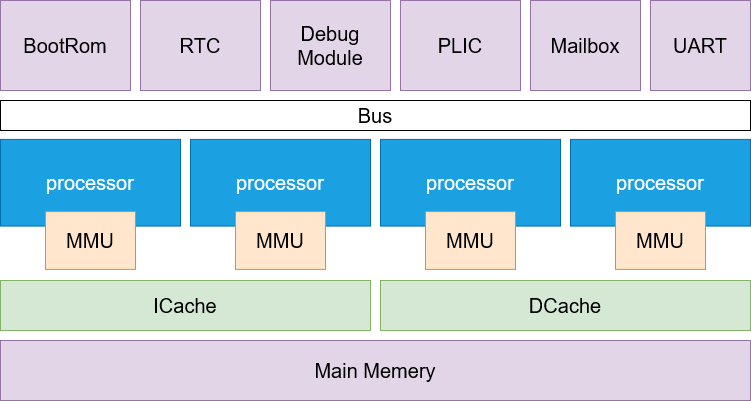
## 4.1 系统概述

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。

指令集模拟(riscv-opcodes)，取值译码执行过程；寄存器/存储器设计；中断系统设计；前端设计。整体各模块层次图。

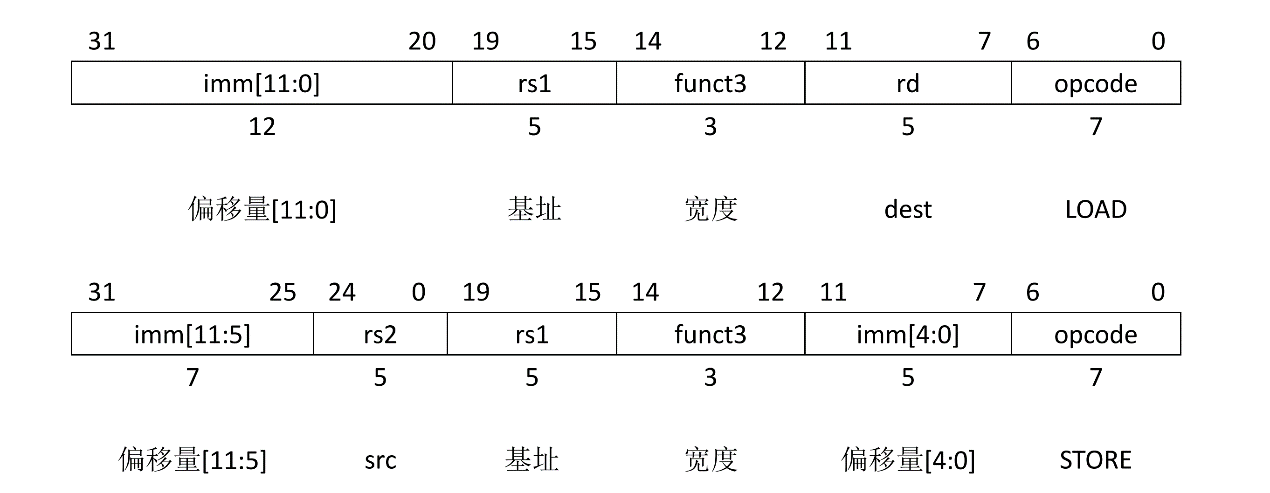
## 4.2 系统静态结构

RISC-V指令集模拟器的整体功能模块如图1.1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1.1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。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,PLIC,Mailbox,Debug Module.

指令集架构：

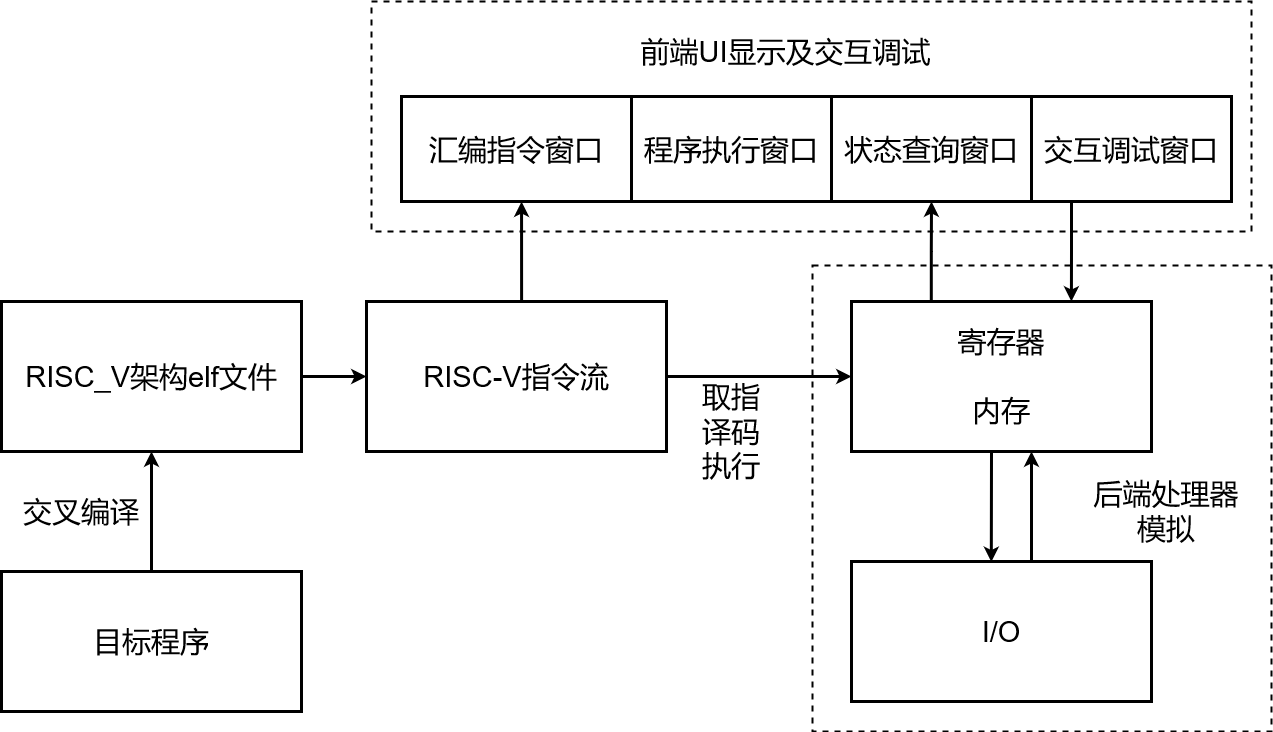
RV32I是一个load-store体系结构，也就是说，只有load和store指令可以访问存储器，而 算术指令只在CPU寄存器上进行操作运算。RV32I提供了一个32位用户地址空间，它是字节寻址并且是小端的。执行环境将定义这个地址空间的哪些部分是可以合法访问的

RISC-V的不同寻常之处，除了在于它是最近诞生的和开源的以外，还在于：和几乎所 有以往的ISA不同，它是模块化的。它的核心是一个名为RV32I的基础ISA，运行一个完整 的软件栈。RV32I是固定的，永远不会改变。这为编译器编写者，操作系统开发人员和汇 编语言程序员提供了稳定的目标。模块化来源于可选的标准扩展，根据应用程序的需要， 硬件可以包含或不包含这些扩展。这种模块化特性使得RISC-V具有了袖珍化、低能耗的特 点，而这对于嵌入式应用可能至关重要。RISC-V编译器得知当前硬件包含哪些扩展后，便 可以生成当前硬件条件下的最佳代码。惯例是把代表扩展的字母附加到指令集名称之后作 为指示。例如，RV32IMFD将乘法（RV32M），单精度浮点（RV32F）和双精度浮点 （RV32D）的扩展添加到了基础指令集（RV32I）中。

第二章已经详细介绍了RISC-V指令集架构，本系统模拟了RV64imafd共196条指令，这些指令的添加参照

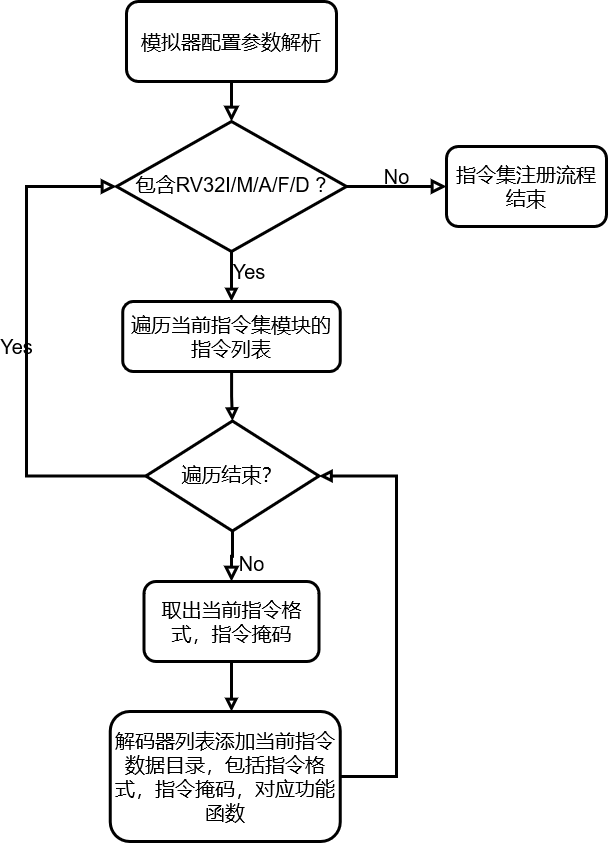
由于本系统忽略了具体硬件实现上的流水线细节，所以就指令控制流程来说本模拟器是一个单周期的CPU，流水线对应的取值，译码，执行，在模拟器上对应解码器解析汇编指令，执行汇编指令对应的功能函数。

## 4.3 系统动态结构

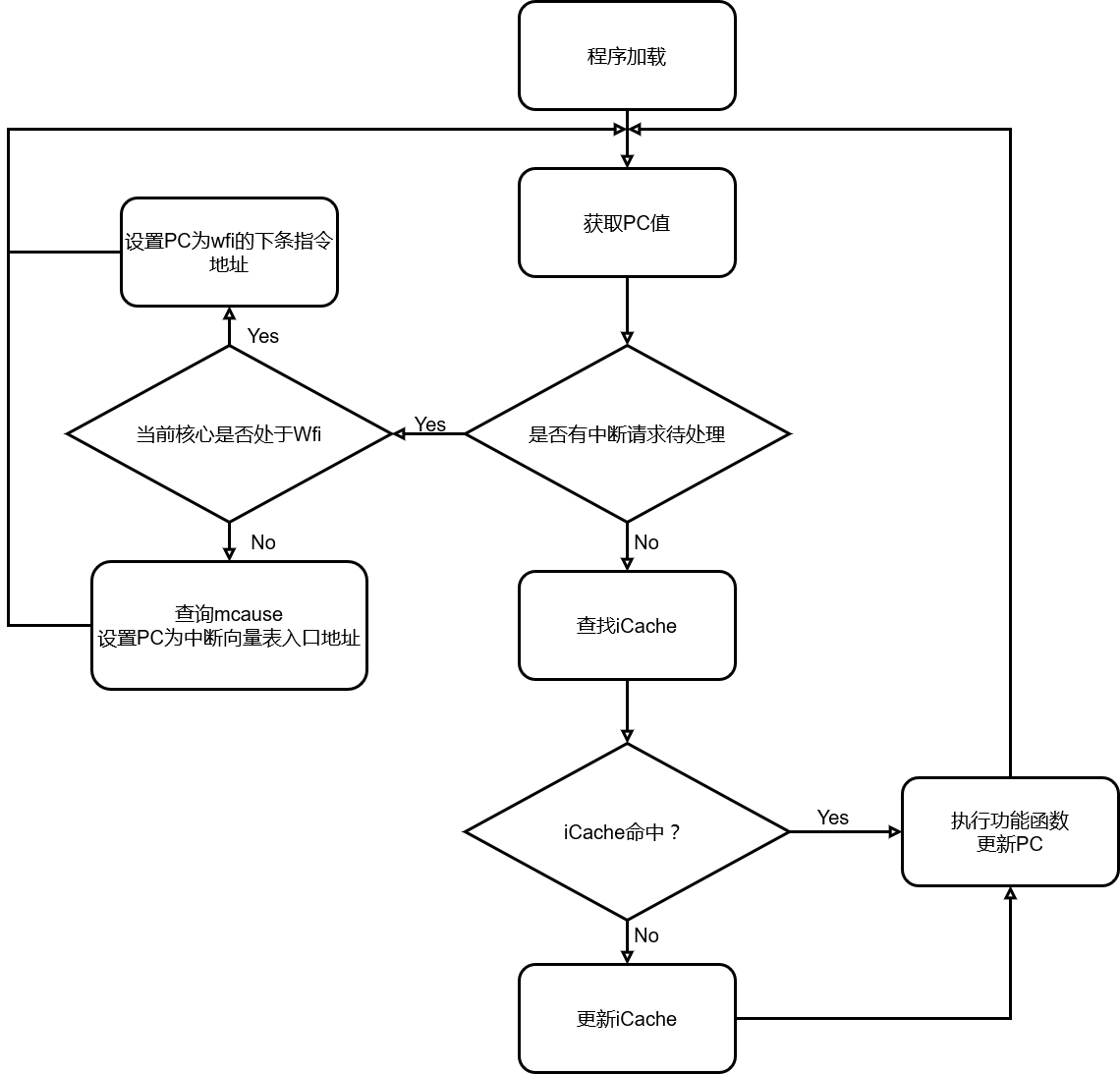
本模拟器是trace-accurate的指令集模拟器（功能模拟器），模拟器运行的基本结构如图1.1所示。

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。

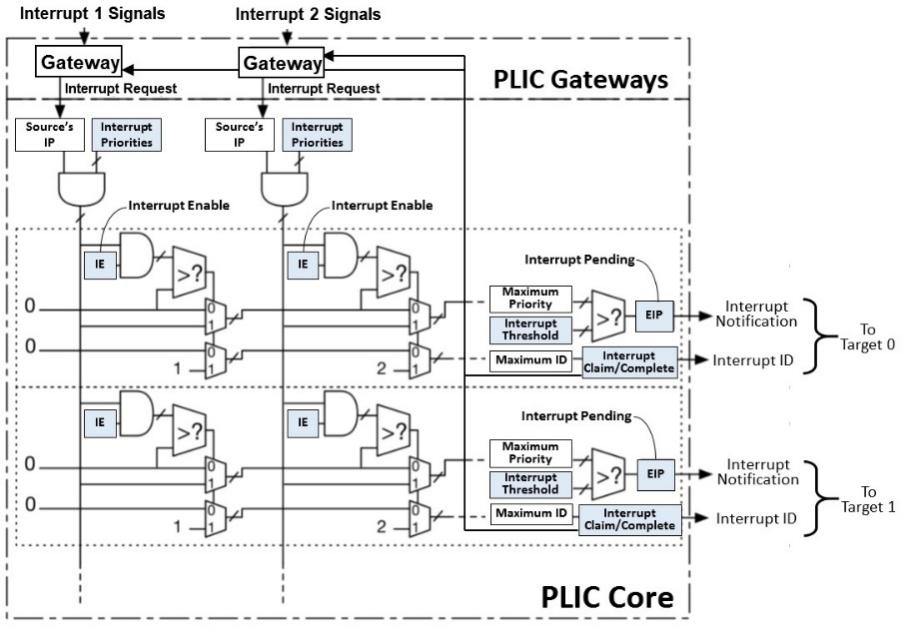
### 4.3.1 解码器与指令集功能函数

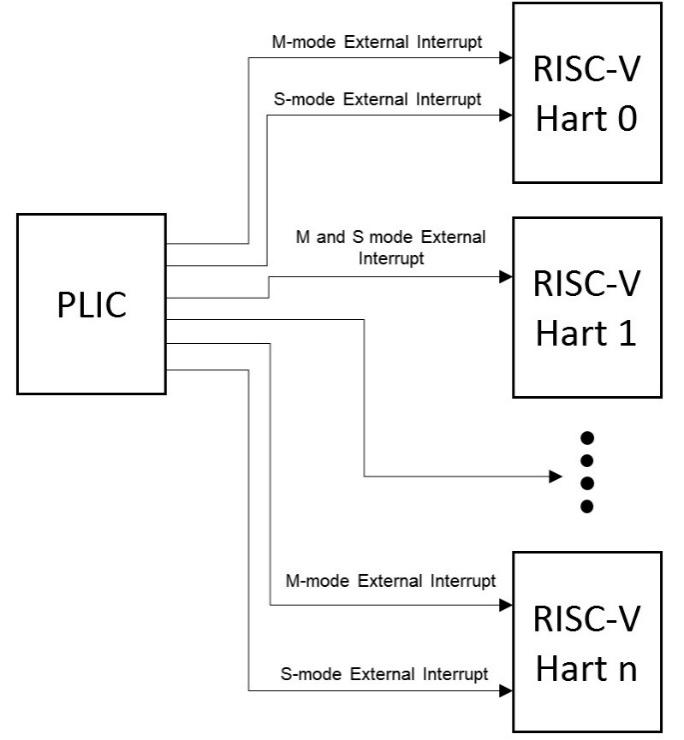
RISC-V指令集是模块化的，它的核心是一个名为RV32I的基础ISA，可选的标准扩展包括MAFDC，根据应用程序的需要，硬件可以包含或不包含这些扩展。本模拟器实现了特权指令集1.9版本，和用户指令集2.1版本的标准拓展指令集共196条指令的模拟。模拟器预加载时通过解析配置参数选择相应的指令集模块进行注册，并初始化解码器，流程如图1.1所示。

### 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计看作是单周期的CPU。

### 4.3.3 平台级中断控制器





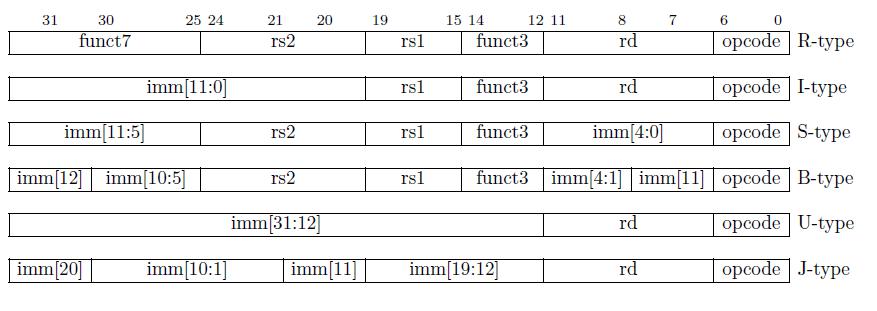
### 4.3.4 交互调试模块

# 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1.1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:

typedef quint64 insn\_bits\_t;

class insn\_t

{

public:

insn\_t() = default;

insn\_t(insn\_bits\_t bits) : b(bits) {}

insn\_bits\_t bits() { return b; }

int length() { return insn\_length(b); }

int64\_t i\_imm() { return int64\_t(b) >> 20; }

int64\_t s\_imm() { return x(7, 5) + (xs(25, 7) << 5); }

int64\_t sb\_imm() { return (x(8, 4) << 1) + (x(25,6) << 5) + (x(7,1) << 11) + (imm\_sign() << 12); }

int64\_t u\_imm() { return int64\_t(b) >> 12 << 12; }

int64\_t uj\_imm() { return (x(21, 10) << 1) + (x(20, 1) << 11) + (x(12, 8) << 12) + (imm\_sign() << 20); }

quint64 rd() { return x(7, 5); }

quint64 rs1() { return x(15, 5); }

quint64 rs2() { return x(20, 5); }

quint64 rs3() { return x(27, 5); }

quint64 rm() { return x(12, 3); }

quint64 csr() { return x(20, 12); }

int64\_t rvc\_imm() { return x(2, 5) + (xs(12, 1) << 5); }

int64\_t rvc\_zimm() { return x(2, 5) + (x(12, 1) << 5); }

int64\_t rvc\_addi4spn\_imm() { return (x(6, 1) << 2) + (x(5, 1) << 3) + (x(11, 2) << 4) + (x(7, 4) << 6); }

int64\_t rvc\_addi16sp\_imm() { return (x(6, 1) << 4) + (x(2, 1) << 5) + (x(5, 1) << 6) + (x(3, 2) << 7) + (xs(12, 1) << 9); }

int64\_t rvc\_lwsp\_imm() { return (x(4, 3) << 2) + (x(12, 1) << 5) + (x(2, 2) << 6); }

int64\_t rvc\_ldsp\_imm() { return (x(5, 2) << 3) + (x(12, 1) << 5) + (x(2, 3) << 6); }

int64\_t rvc\_swsp\_imm() { return (x(9, 4) << 2) + (x(7, 2) << 6); }

int64\_t rvc\_sdsp\_imm() { return (x(10, 3) << 3) + (x(7, 3) << 6); }

int64\_t rvc\_lw\_imm() { return (x(6, 1) << 2) + (x(10, 3) << 3) + (x(5, 1) << 6); }

int64\_t rvc\_ld\_imm() { return (x(10, 3) << 3) + (x(5, 2) << 6); }

int64\_t rvc\_j\_imm() { return (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); }

int64\_t rvc\_b\_imm() { return (x(3, 2) << 1) + (x(10, 2) << 3) + (x(2, 1) << 5) + (x(5, 2) << 6) + (xs(12, 1) << 8); }

int64\_t rvc\_simm3() { return x(10, 3); }

quint64 rvc\_rd() { return rd(); }

quint64 rvc\_rs1() { return rd(); }

quint64 rvc\_rs2() { return x(2, 5); }

quint64 rvc\_rs1s() { return 8 + x(7, 3); }

quint64 rvc\_rs2s() { return 8 + x(2, 3); }

private:

insn\_bits\_t b;

quint64 x(int lo, int len) { return (b >> lo) & ((insn\_bits\_t(1) << len)-1); }

quint64 xs(int lo, int len) { return int64\_t(b) << (64-lo-len) >> (64-len); }

quint64 imm\_sign() { return 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1.1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

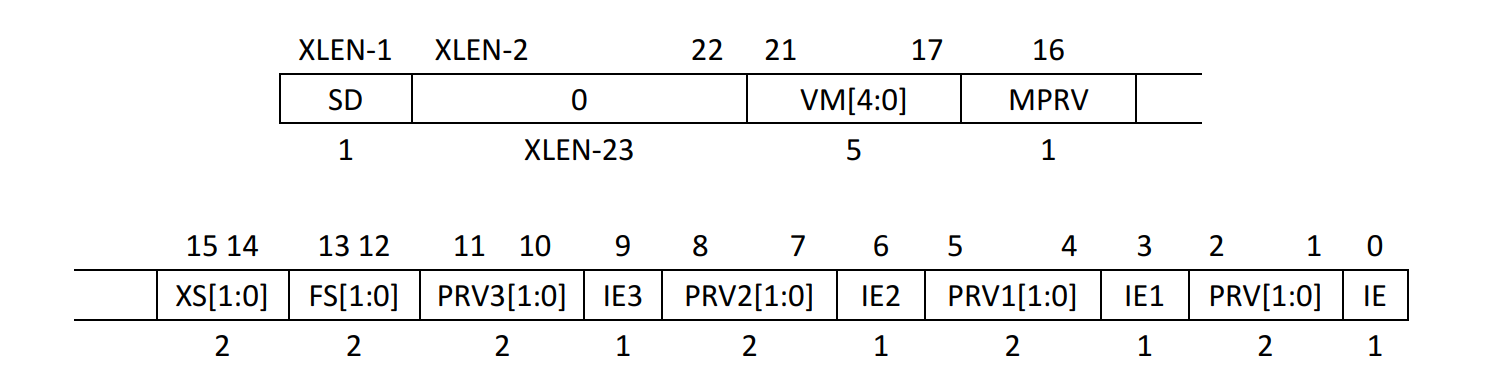
上一节介绍了指令集模块的实现,从一个较高的抽象层次对模拟器的功能进行了定义,本节将详细介绍硬件的模拟细节,主要包含CPU内部的各个功能部件,以及CPU与片外通信的桥梁--总线的设计与实现.

### 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你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.

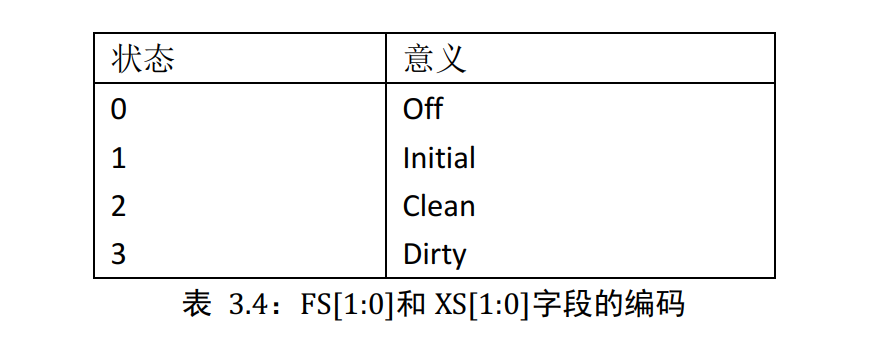
寄存器类图如图1.1所示.总体设计上,三组寄存器均封装在processor\_t类内部,其中通用寄存器组可以被处理器类对象直接调用,进行读写操作;CSR寄存器组被声明为类私有成员,对外提供get\_csr(),set\_csr()统一接口.将CSR寄存器访问权限,状态寄存器位域信息获取等操作封装在接口内部,这样就可以使指令集功能函数的实现不必关心具体的寄存器实现细节.对于CSR寄存器接口的实现主要需要考虑三个因素,检查寄存器组所需的指令集模块支持;处理器当前权限模式检查;以及状态寄存器写入格式的检查.

以状态寄存器CSR\_MSTATUS为例.mstatus 寄存器是一个XLEN位的可读/可写寄存器，其格式如图 3.4 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 PRV[1:0]字段保存了硬件线程当前特权模式，其编码如表 1.1 所示。如果实现仅提供M-mode，那么这两位被硬连线到 11,处理器寻址不经过MMU。IE 位指示了在当前特权模式，是否使能中断（1=使能， 0=禁止），其主要用途是禁止中断，以便在与当前特权级的中断处理函数相关时，能够保证原子性。当一个硬件线程运行在一个给定的特权模式时，更高特权模式的中断总是使能的，而更低特权模式的中断总是禁用的。更高特权等级的代码可以在移交控制给较低特权级之前，使用每个中断单独的使能位，来禁用选定的中断。

为了支持嵌套的自陷，提供了一个 PRV 和 IE 位的栈，这个栈的深度与支持的特权模式数量相等，此处 PRV0 是活跃的特权模式 PRV（即 PRV0-PRVN 对应 N 级特权模式），除了如果实现仅支持机器模式，在此种情形下，这个栈的深度是 2，并且所有的 PRV 字段都被硬连线为 11。当处理一个自陷时，这个栈被向左边 push，并且 PRV 被设置为活跃自陷处理函数的特权模式，且 IE=0。当从一个自陷处理函数返回时（使用 ERET 指令），栈被向右 pop，并且最左边的项（PRVN）被设置为所支持的最低特权等级，且使能中断（即，在一个只有 M模式的机器上， PRV1=M 且 IE1=1，而在一个有两个或者更多模式的机器上，当从自陷处理函数返回时， PRVN=U 且 IEN=1）。在通常操作中，这个栈应该包含从左到右单调增加的特权模式（最老的到最新的）。

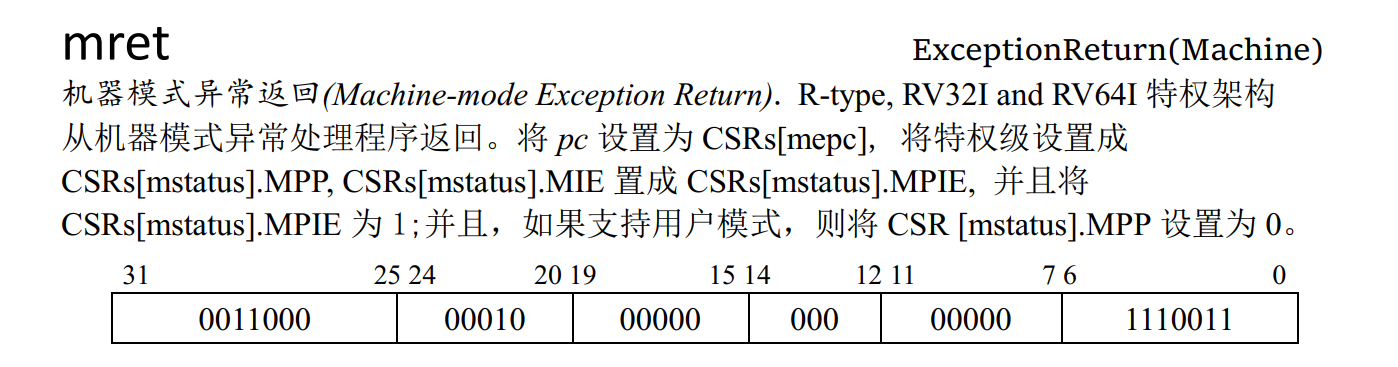
MPRV 位修改了 load 和 store 执行的特权级。当 MPRV=0 时，翻译和保护如同寻常一样。当MPRV=1时，数据存储器地址就如同PRV被设置为当前PRV1字段的值一样被翻译和保护。指令地址翻译和保护不受影响。当出现一个异常时， MPRV 被复位为 0。

支持丰富的扩展，是 RISC-V 的一个主要目标，因此我们定义了一个标准接口，允许不改变特权模式代码，特别是一个管理员级操作系统，来支持任意用户模式状态扩展。FS[1:0]和 XS[1:0]可读/可写字段被用于减少保存和恢复上下文的开销，这是通过设置和跟踪浮点单元和任何其他用户模式扩展的状态来实现的。 FS 字段编码了浮点单元的状况，包括 CSR fcsr 和浮点数据寄存器 f0-f31，而 XS 字段编码了任何额外用户模式扩展及相关联的状态的状况。 SD 位是一个只读位，指明了是否 FS 字段或者 XS 字段编码了一个脏的状态，需要将扩展的用户上下文写入到存储器中。在没有浮点单元的系统中， FS 字段被硬连线到零，而在没有需要新状态的额外用户扩展的系统中， XS 字段被硬连线到零。如果 FS 字段和XS 字段都被硬连线到零，那么 SD 也总是零。

  
FS 字段和 XS 字段使用了相同的的状况编码，如表 3.4 所示，其四个可能的的状况值是Off、 Initial、 Clean 和 Dirty

当的状况被设置为 Off 时，任何指令试图读写相应的状态都会导致一个异常。当的状况是 Initial 时，对应的状态应当具有一个初始的常数值。当的状况是 Clean 时，对应的状态可能与 Initial 时的状态不同，但与上下文切换时保存的最后的值相匹配。当的状况是 Dirty 时，对应的状态可能自上次上下文保存时以来，已经被修改。当进行一个上下文保存时，负责的特权代码仅在对应的状态的状况是 Dirty 时，才需要将上下文写入存储器，然后就可以复位其状况为 Clean。当进行一个上下文恢复时，仅在状况是 Clean 时（在恢复时，应该永远不会是 Dirty），才需要从存储器中读取上下文。如果状况是 Initial，在恢复上下文时，上下文必须被设置为一个初始值，以避免一个安全漏洞，但这不需要访问存储器即可完成。例如，浮点寄存器可以全部初始化为立即数值 0。当继续一个用户上下文时， FS 字段和 XS 字段可被特权代码设置，并在保存上下文之前，被特权代码读取。在执行指令时，状况字段也会被更新，而不管特权模式如何。

如图是汇编指令mret的定义以及功能函数实现

require\_privilege(PRV\_M);

set\_pc(p->get\_state().mepc);

reg\_t s = STATE.mstatus;

reg\_t prev\_prv = get\_field(s, MSTATUS\_MPP);

s = set\_field(s, MSTATUS\_UIE << prev\_prv, get\_field(s, MSTATUS\_MPIE));

s = set\_field(s, MSTATUS\_MPIE, 1);

s = set\_field(s, MSTATUS\_MPP, PRV\_U);

p->set\_privilege(prev\_prv);

p->set\_csr(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:

{

if ((val ^ CSR.mstatus) & (MSTATUS\_VM | MSTATUS\_MPP | MSTATUS\_MPRV | MSTATUS\_PUM | MSTATUS\_MXR))

mmu->flush\_tlb();

reg\_t mask = MSTATUS\_SIE | MSTATUS\_SPIE | MSTATUS\_MIE | MSTATUS\_MPIE

| MSTATUS\_SPP | MSTATUS\_FS | MSTATUS\_MPRV | MSTATUS\_PUM

| MSTATUS\_MPP | MSTATUS\_MXR ;

if (validate\_vm(max\_xlen, get\_field(val, MSTATUS\_VM)))

mask |= MSTATUS\_VM;

CSR.mstatus = (CSR.mstatus & ~mask) | (val & mask);

bool dirty = (CSR.mstatus & MSTATUS\_FS) == MSTATUS\_FS;

dirty |= (CSR.mstatus & MSTATUS\_XS) == MSTATUS\_XS;

if (max\_xlen == 32)

CSR.mstatus = set\_field(CSR.mstatus, MSTATUS32\_SD, dirty);

else

CSR.mstatus = set\_field(CSR.mstatus, MSTATUS64\_SD, dirty);

// spike supports the notion of xlen < max\_xlen, but current priv spec

// doesn't provide a mechanism to run RV32 software on an RV64 machine

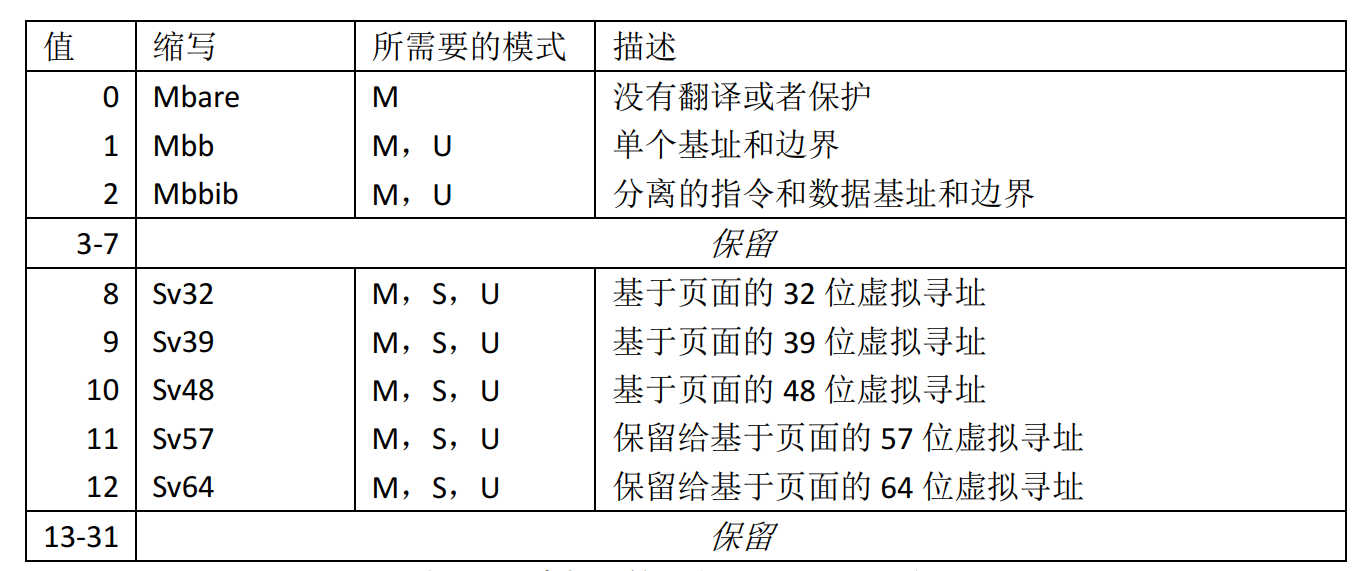
xlen = max\_xlen;

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。表 3.3 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。Sv32、 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.

#ifdef [CONFIG\_MMU](https://elixir.bootlin.com/linux/v5.8.2/K/ident/CONFIG_MMU)[relocate](https://elixir.bootlin.com/linux/v5.8.2/C/ident/relocate):

/\* Relocate return address \*/

[li](https://elixir.bootlin.com/linux/v5.8.2/C/ident/li) [a1](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a1), [PAGE\_OFFSET](https://elixir.bootlin.com/linux/v5.8.2/C/ident/PAGE_OFFSET)

[la](https://elixir.bootlin.com/linux/v5.8.2/C/ident/la) [a2](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a2), [\_start](https://elixir.bootlin.com/linux/v5.8.2/C/ident/_start)

[sub](https://elixir.bootlin.com/linux/v5.8.2/C/ident/sub) [a1](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a1), [a1](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a1), [a2](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a2)

[add](https://elixir.bootlin.com/linux/v5.8.2/C/ident/add) [ra](https://elixir.bootlin.com/linux/v5.8.2/C/ident/ra), [ra](https://elixir.bootlin.com/linux/v5.8.2/C/ident/ra), [a1](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a1)

/\* Point stvec to virtual address of intruction after satp write \*/

[la](https://elixir.bootlin.com/linux/v5.8.2/C/ident/la) [a2](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a2), 1f

[add](https://elixir.bootlin.com/linux/v5.8.2/C/ident/add) [a2](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a2), [a2](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a2), [a1](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a1)

csrw [CSR\_TVEC](https://elixir.bootlin.com/linux/v5.8.2/C/ident/CSR_TVEC), [a2](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a2)

/\* Compute satp for kernel page tables, but don't load it yet \*/

srl [a2](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a2), [a0](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a0), [PAGE\_SHIFT](https://elixir.bootlin.com/linux/v5.8.2/C/ident/PAGE_SHIFT)

[li](https://elixir.bootlin.com/linux/v5.8.2/C/ident/li) [a1](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a1), [SATP\_MODE](https://elixir.bootlin.com/linux/v5.8.2/C/ident/SATP_MODE)

[or](https://elixir.bootlin.com/linux/v5.8.2/C/ident/or) [a2](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a2), [a2](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a2), [a1](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a1)

/\* \* Load trampoline page directory, which will cause us to trap to \* stvec if VA != PA, or simply fall through if VA == PA. We need a \* full fence here because setup\_vm() just wrote these PTEs and we need \* to ensure the new translations are in use. \*/

[la](https://elixir.bootlin.com/linux/v5.8.2/C/ident/la) [a0](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a0), [trampoline\_pg\_dir](https://elixir.bootlin.com/linux/v5.8.2/C/ident/trampoline_pg_dir)

srl [a0](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a0), [a0](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a0), [PAGE\_SHIFT](https://elixir.bootlin.com/linux/v5.8.2/C/ident/PAGE_SHIFT)

[or](https://elixir.bootlin.com/linux/v5.8.2/C/ident/or) [a0](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a0), [a0](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a0), [a1](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a1)

[sfence](https://elixir.bootlin.com/linux/v5.8.2/C/ident/sfence).[vma](https://elixir.bootlin.com/linux/v5.8.2/C/ident/vma)

csrw [CSR\_SATP](https://elixir.bootlin.com/linux/v5.8.2/C/ident/CSR_SATP), [a0](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a0).[align](https://elixir.bootlin.com/linux/v5.8.2/C/ident/align) 21:

/\* Set trap vector to spin forever to help debug \*/

[la](https://elixir.bootlin.com/linux/v5.8.2/C/ident/la) [a0](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a0), .[Lsecondary\_park](https://elixir.bootlin.com/linux/v5.8.2/C/ident/Lsecondary_park)

csrw [CSR\_TVEC](https://elixir.bootlin.com/linux/v5.8.2/C/ident/CSR_TVEC), [a0](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a0)

/\* Reload the global pointer \*/.[option](https://elixir.bootlin.com/linux/v5.8.2/C/ident/option) [push](https://elixir.bootlin.com/linux/v5.8.2/C/ident/push).[option](https://elixir.bootlin.com/linux/v5.8.2/C/ident/option) norelax

[la](https://elixir.bootlin.com/linux/v5.8.2/C/ident/la) [gp](https://elixir.bootlin.com/linux/v5.8.2/C/ident/gp), \_\_global\_pointer$.[option](https://elixir.bootlin.com/linux/v5.8.2/C/ident/option) [pop](https://elixir.bootlin.com/linux/v5.8.2/C/ident/pop)

/\* \* Switch to kernel page tables. A full fence is necessary in order to \* avoid using the trampoline translations, which are only correct for \* the first superpage. Fetching the fence is guarnteed to work \* because that first superpage is translated the same way. \*/

csrw [CSR\_SATP](https://elixir.bootlin.com/linux/v5.8.2/C/ident/CSR_SATP), [a2](https://elixir.bootlin.com/linux/v5.8.2/C/ident/a2)

[sfence](https://elixir.bootlin.com/linux/v5.8.2/C/ident/sfence).[vma](https://elixir.bootlin.com/linux/v5.8.2/C/ident/vma)

ret#endif /\* 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<class T>

inline T load(reg\_t addr)

{

if (addr & (sizeof(T)-1))

throw trap\_t(trap\_load\_address\_misaligned,addr);

reg\_t vpn = addr >> PGSHIFT;

if (likely(tlb\_load\_tag[vpn % TLB\_ENTRIES] == vpn))

return \*(T\*)(tlb\_data[vpn % TLB\_ENTRIES] + addr);

T res;

load\_slow\_path(addr, sizeof(T), (uint8\_t\*)&res);

return res;

}

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1.1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0x00000000~0x80000000,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:

class bus\_t : public abstract\_device\_t

{

public:

bool load(reg\_t addr, size\_t len, uint8\_t\* bytes);

bool store(reg\_t addr, size\_t len, const uint8\_t\* bytes);

void add\_device(reg\_t addr, abstract\_device\_t\* dev);

void add\_register(reg\_t addr, abstract\_device\_t\* dev,qint32 offset);

private:

std::map<reg\_t, abstract\_device\_t\*> devices;

struct device\_reg

{

device\_reg(){}

device\_reg(abstract\_device\_t \*dev,qint32 offset):dev(dev),offset(offset){}

abstract\_device\_t \*dev;

qint32 offset;

};

QMap<reg\_t,device\_reg> regs;

};

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

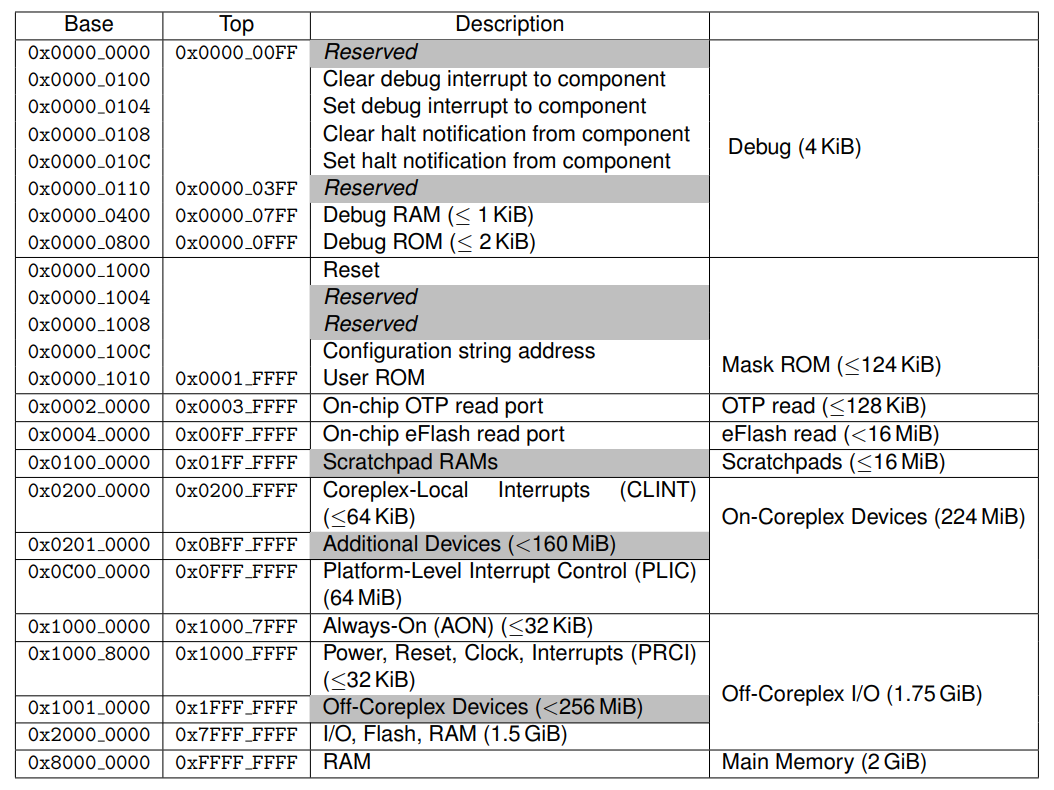
jr t0

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1.1所示.

## mmio.drawio5.3 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.

|  |  |
| --- | --- |
| 寄存器 | 功能 |
| 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 |  |

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:

class trap\_t

{

public:

trap\_t(){}

trap\_t(reg\_t type):cause(type),addr\_valid(false){}

trap\_t(reg\_t type,quint64 addr):cause(type),addr\_valid(true),addr(addr){}

reg\_t cause;

bool addr\_valid;

quint64 addr;

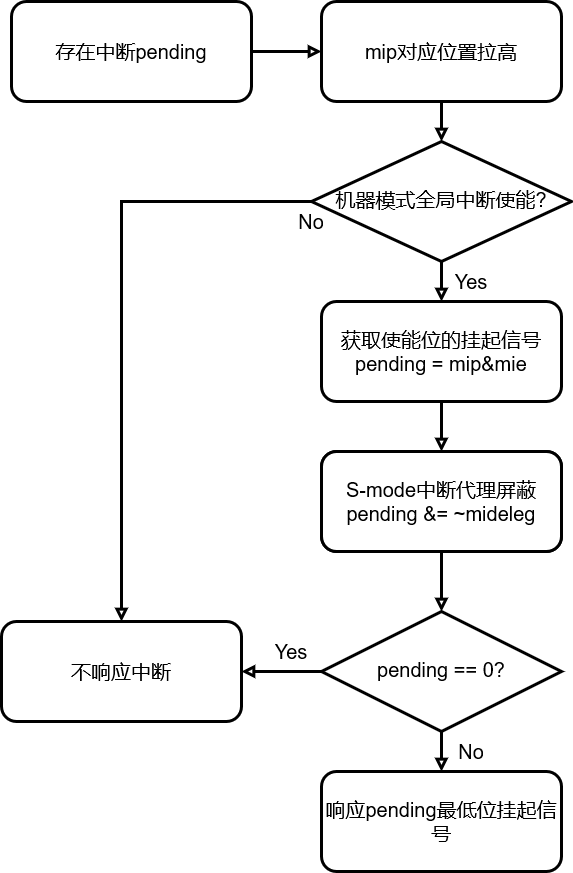
static QMap<reg\_t,QString> names;

static quint64 delegable\_exceptions;

QString &name(){return names[cause];}

};

处理器响应的流程如图1.1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.

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.

### plic5.3.1 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1.1是SiFive公司给出的PLIC规范框架图.

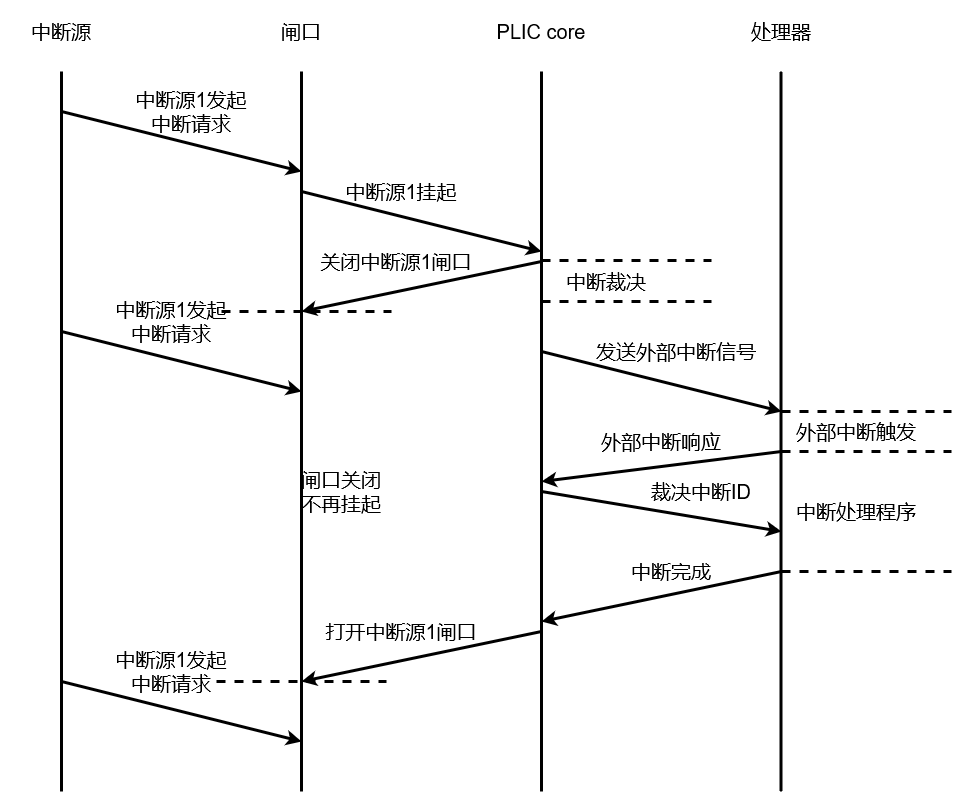
中断源(interrupt sources)是挂载在PLIC上的设备的统称,一般是I/O设备.PLIC理论上支持任意多个中断源，每个中断源可以是不同触发类型，电平触发或者边沿触发、PLIC为每个中断源分配了如下的信息:

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1.1展示了PLIC处理外部中断的流程.



中断源首先通过闸口给PLIC core发送中断请求,PLIC core将对应中断源的pending寄存器置位,并将该中断源闸口关闭.PLIC core进行中断信号裁决,获得最高优先级的中断源,通过对mip寄存器meip/seip置位将外部中断信号发送给处理器.当处理器接受了外部信号,会发送一个claim信号给PLIC core,表现为读取裁决结果的最高优先级中断源,同时将对应的该中断源pending拉低.当处理器完成了中断处理,发送一个中断完成信息给PLIC core.表示中断完成,打开对应中断源闸口.

根据以上的流程,模拟器实现的PLIC设备类图如图1.1所示.

interrupt notification，由PLIC Core 发给各target的中断请求，PLIC Core为每个target赋予了一个(external interrupt pending bit) EIP，表示有待处理的中断。EIP的值可以被source，target或者其他源进行修改。EIP发送给target的过程被称为interrupt notification。PLIC Core只支持multicasting，即中断信号会发给所有符合条件的target，并不会选择其中之一发送。这样虽然响应较快，但会带来一些冗余处理。软件可以通过控制IE 位间接改善。首先claim的target会负责该中断的处理，PLIC Core只保证对于EIP的修改所有对应的targets都可见。

interrupt claim，由target返回给PLIC Core的响应信号，表示请求接受。PLIC Core收到claim后，会选出最高优先级的source ID，并将其对应的IP清除。这个ID会发送给target。如果ID为0，表示没有需要处理的中断。claim被PLIC Core接受后，次优先级的中断会显现出来，因此对应的EIP可能并不会被清0。因此target可以在退出中断服务程序前检查本地Xeip 位(X表示当前特权模式)，确认是否有pending 的EIP。PLIC支持target在EIP=0的情况下仍然发起claim。这是为了支持某些target的threshold设置为最高，不接受中断打断，但使用claim方式进行查询。

interrupt completion，由target 返回给PLIC Core的完成响应信号。

global interrupt handling，基本过程是，global sources首先发送给interrupt gateway，由gateway负责产生interrupt request，发送给PLIC Core， PLIC Core将每个interrupt request储存在内部的(interrupt pending bits) IP中，如果该targets有enabled pending interrupt， 并且priority超过了per-target threshold ，PLIC Core将interrupt notification发送给一个或者多个targets。当target接受了该外部中断，会发送一个interrupt claim request给PLIC Core，用以取得对应该target的最高优先级的pending interrupt，同时将对应的IP位清零，当target完成了中断服务，需发送一个interrupt completion message给对应的interrupt gateway表示中断完成。接下来gateway可以发送另一个interrupt request给该target。

### 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。

### 5.3.3 外部中断源模拟

## 5.4 UI显示模块的实现

## 5.5 安全区保护锁实现

# 第6章 系统测试

## 6.1 测试概述

## 6.2 测试方案

表6.1 测试环境描述表

|  |  |
| --- | --- |
| 工具 | 说明 |
| 操作系统 | Ubuntu 16.04.1 |
| 处理器类型 | Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz |
| 内核 | Linux 4.15.0-66-generic |
| 处理器架构 | x86-64 |
| 内存总量 | 128G |

## 6.3 正确性分析

## 6.4 测试数据

# 第7章 结论与展望

## 7.1 总结

控制流劫持是一类普遍且能够造成严重危害的攻击，而代码重用则是进行控制流劫持的主流攻击方法，本文介绍了面向跳转和面向返回地址的两种基本代攻击方法以及它们的衍生。详细论述了攻击的流程以及实现的条件，同时也分析了不同防御方法的原理和优缺点。利用了一种较为创新的防御思想CPI来实现我们的防御模型，这是一种较好代码重用攻击防御方法，主要的原理是保证程序中代码指针的完全性。我们也分析了原有实现的薄弱性，增加了新的内存隔离方法，在原有基础上使程序得到更加强大的安全保证。本文详细地介绍了攻击模型的原理，实现所依赖的技术，分析设计了系统的结构，并提出了具体的解决方案。同时也论证了模型的有效性，测试了具体的性能损耗。

## 7.2 未来工作

我们将进一步完善我们的模型，找出更加精确的静态分析算法和性能更低的插桩实现。我们现有实现上静态分析不够精准，我们希望进一步结合程序之间的数据流分析方法来做更加精确的静态分析。又因为软件种类的繁多，其中包含各种内存漏洞，我们将进一步测试更多的程序来完善我们的模型。对于性能损耗上我们希望实现不同的安全区组织方法来对比这之间的性能差异，寻找出更加高效且节省内存的方法。

# 参 考 文 献

1. Szekeres L , Payer M , Tao W , et al. SoK: Eternal War in Memory[C]// IEEE Symposium on Security & Privacy. IEEE Computer Society, 2013.
2. 邵思豪, 高庆, 马森,等. 缓冲区溢出漏洞分析技术研究进展[J]. 软件学报, 2018, 029 (005):1177-1198.
3. 王清. 0day安全:软件漏洞分析技术(第2版)[J]. 信息安全与通信保密, 2013(11):130-131.
4. Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016.
5. VAN DER V EEN , V., D UTT S HARMA , N., C AVALLARO , L., AND BOS , H. Memory errors: The past, the present, and the future. In Proceedings of the 15th International Symposium on Research in Attacks, Intrusions and Defenses, 2012.
6. 张超. 针对控制流劫持攻击的软件安全防护技术研究[D]. 北京:北京大学, 2013.
7. 柳童,史岗,孟丹.代码重用攻击与防御机制综述[J].信息安全学报, 2016,1(02):15-27.
8. 王丰峰,张涛,徐伟光,孙蒙.[进程控制流劫持攻击与防御技术综述](https://kns.cnki.net/kcms/detail/detail.aspx?filename=WXAQ201906002&dbcode=CJFD&dbname=CJFDTEMP" \t "https://schlr.cnki.net/en/Detail/index/journal/_blank)[J].网络与信息安全学 报, 2019(06).
9. Microsoft. A Detailed Description of the Data Execution Prevention (DEP) Feature in Windows XP Service Pack 2,Windows XP Tablet PC Edition 2005, and Windows Server 2003.<https://support.microsoft.com/en-us/kb/875352>.
10. 黄志军,郑滔.基于Return-Oriented Programming的程序攻击与防护[J].计算机科学, 2012,39(S1).
11. Kornau T. Return oriented programming for the ARM architecture[D]. Master’s thesis, Ruhr-Universität Bochum, 2010.
12. T. Bletsch, X. Jiang, V.Freh, and Z. Liang. Jump Oriented Programming:A New Class of Code-Reuse[C]. In Proceedings of the 2016 the 6th ACM Symposium on Information, Computer and Communications Security(ASIACCS ’11), 2011:30-40.
13. 邢骁. 自动化构造Jump-Oriented Programming攻击[D].南京大学, 2012.
14. PAX Team. “Address Space Layout Randomization” <http://pax.grsecurity.net/docs/aslr.txt>, 2003.
15. 林键,郭玉东,周少皇,蔄羽佳.基于基址重定位的地址随机化技术[J].信息工程大学学报, 2016,17(06):748-753.
16. SNOW K Z, MONROSE F, DAVI L, et al. Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization[C]. IEEE Symposium on Security and Privacy, 2013:574-588.
17. BITTAU A, BELAY A, MASHTIZADEH A, et al. Hacking blind[C]. IEEE Symposium on Security and Privacy, 2014:227-242.
18. J. Seibert, H. Okhravi, and E. Soderstrom. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code[C]. In Proceedings of the 21st ACM Confer- ence on Computer and Communications Security (CCS), 2014.
19. R. Strackx, Y. Younan, P. Philippaerts, F. Piessens,S. Lachmund, and T. Walter. Breaking the memory secrecy assumption[C]. In Proceedings of the Second European Workshop on System Security (EUROSEC '09), 2009.
20. M. Abadi, M. Budiu, J. Ligatti, and U. Erlingsson. Control-Flow Integrity[C]. in Proc the 12th ACM Conference on Computer and Communications Security(CCS’05), 2005:340-353.
21. Oleksenko O , Kuvaiskii D , Bhatotia P , et al. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack[J]. In Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2018, 2(2):1-30.
22. Evans I , Fingeret S , Gonzalez J , et al. Missing the Point(er): On the Effectiveness of Code Pointer Integrity[C]. IEEE Symposium on Security and Privacy, 2015.
23. J. P. Anderson. Computer security technology planning study[N]. volume 2. Technical report, DTIC Document,1972.
24. X. Chen, D. Caselden, and M. Scott. New zero-day exploit[J]. targeting internet explorer versions 9 through 11 identified in targeted attacks, 2014.
25. S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: compiler enforced temporal safety for c. In ACM Sigplan Notices, volume 45, pages 31–40. ACM, 2010.
26. G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type safe retrofitting of legacy code[J]. ACM SIGPLAN Notices, 2002,37(1):128-139.
27. Dan G , Hicks M , Jim T , et al. Cyclone: A Type-Safe Dialect of C[J]. C-C++ Users Journal, 2005, 23(1):6,8,10,12-13.
28. P. Akritidis. Cling: A memory allocator to mitigate dangling pointers[C]. In USENIX Security Symposium, 2010:177-192.
29. N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation[J]. In ACM Sigplan Notices, volume 42, ACM, 2007:89-100.
30. N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking[C]. In IEEE/ACM Symp. on Code Generation and Optimization, 2012.
31. K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker[C]. In USENIX Annual Technical Conference, 2012:309-318.
32. MASHTIZADEH A J, BITTAU A, MAZIERES D, et al. Cryptographically enforced control flow integrity[J]. arXiv preprint arXiv, 2014:1408-1451.
33. CRISWELL J, DAUTENHAHN N, ADVE V. KCoFI: complete control-flow integrity for commodity operating system kernels[C]. IEEE Symposium on Security and Privacy, 2014:292-307.
34. ZHANG C, WEI T, CHEN Z, et al. Practical control flow integrity and randomization for binary executables[J]. IEEE Symposium on Security & Privacy, 2013:559-573.
35. ZHANG M, SEKAR R. Control Flow Integrity for COTS Binaries[C]. Presented as part of the 22nd Security Symposium, 2013:337-352.
36. L. Davi, A. Sadeghi, D. Lehmann. and F. Monrose, Stitching the Gadgets On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection[C]. In Proceedings of the 23rd USENIX Security Symposium.(Usenix’14), 2014:401-416.
37. E. Athanasopoulos.H. Bos,G. Portokalidis, and E. Goktas. Out of Control Overcoming Control-Flow Integrity[C]. IEEE Symposium on Security and Privacy.(SP’14),2014:575-589.
38. Lu K , Song C , Lee B , et al. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks[C]. In Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS '15). ACM, 2015.
39. GIUFFRIDA C, KUIJSTEN A, TANENBAUM A S. Enhanced operating system security through efficient and fine-grained address space randomization[C]. Presented as Part of the 21st USENIX Security Symposium (USENIX Security 12), 2012:475-490.
40. Chen X , Xue R , Wu C . Timely address space rerandomization for resisting code reuse attacks[J]. Concurrency and Computation, 2017, 29(16):1-14.
41. CHEN Y, WANG Z, WHALLEY D, et al. Remix: on-demand live randomization[C]. The sixth ACM Conference on Data and Application Security and Privacy, 2016: 50-61.
42. HAWKINS W, NGUYEN-TUONG A, HISER J D, et al. Mixr: flexible runtime rerandomization for binaries[C]. The 2017 Workshop on Moving Target Defense. 2017:27-37.
43. BIGELOW D, HOBSON T,RUDD R, et al. Timely rerandomization for mitigating memory disclosures[C]. ACM Sigsac Conference on Computer & Communications Security, 2015.
44. 雷啸. 内存信息泄露的运行中随机化防御方法的研究与改进[D]. 南京: 南京大学, 2017.
45. MORTON M, KOO H, LI F, et al. Defeating zombie gadgets by re-randomizing code upon disclosure[C]. International Symposium on Engineering Secure Software and Systems, 2017:143-160.
46. V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity[C]. In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, 2014:6-8.
47. Hedayati M , Gravani S , Johnson E , et al. Hodor: intra-process isolation for high-throughput data plane libraries[C]. In the Proceedings of the 2019 USENIX Annual Technical Conference, 2019:10-12.
48. Park S , Lee S , Xu W , et al. libmpk: Software Abstraction for Intel Memory Protection Keys [J]. 2018.
49. Linux. The Linux Kernel Archives[OL]. https://www.kernel.org/doc/html/v5.4/x86/intel\_mpx. [html](https://www.kernel.org/doc/html/v5.4/x86/intel_mpx.html).
50. N. Burow, X. Zhang and M. Payer, SoK: Shining Light on Shadow Stacks[J], 2019 IEEE Symposium on Security and Privacy (SP), 2019:985-999.
51. Ben Niu and Gang Tan. Modular control-flow integrity[C]. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14), 2014:577-587.
52. P. Akritidis, C. Cadar, C. Raiciu, M. Costa and M. Castro, Preventing Memory Error Exploits with WIT.[C]. IEEE Symposium on Security and Privacy (sp 2008), 2008:263-277.
53. N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses[C]. In USENIX Security Symposium, 2014:20-24.

# 致 谢

忙碌的读研生活就要结束了，而我也将再一次经历别离，两年时光里我遇到了许许多多博学可爱的人，在这邻近毕业之际我也只能通过这短短的几句话来表达我对身边可敬之人的感谢

首先我要感谢我的论文指导老师汪老师，感谢老师在论文完成过程中的给予的帮助。其次我要感谢实习单位的领路人应师兄，师兄在我实习的过程中给予了极大的帮助，教授了我许多的知识，在我们平常的学习，论文的撰写都给予了许多宝贵的意见。我也感谢科大给了我学习的机会，在以后的生活中我会一直会以是科大学子而感到自豪，最后感谢各位百忙之中参加评审答辩的老师。

而今我坐在这窗台前，迎着清晨第一缕阳光写下这最后一句话，再见了我18年的学生生涯。

2021年10月28日