西安电子科技大学

请

学校代码 10701　　学　号 20111213098
分 类 号 TP332　　密　级 公开

# 西安电子科技大学

## 硕士学位论文

# 一种基于gem5的RISC-V V处理器模拟器的设计与实现

**作者姓名：**黄泽波

**领　　域：**电子信息

**学位类别：**电子信息硕士

**学校导师姓名、职称：** 郭　辉 研究员

**企业导师姓名、职称：** 黄　辉 中工

**学　　院：**微电子学院

**提交日期：**2023 年 6 月

# Implementing and Designing RISC-V V Processor Simulator in Gem5

A thesis submitted to

XIDIAN UNIVERSITY

in partial fulfillment of the requirements

for the degree of Master

in Electronic Information

By

Huang Zebo

Supervisor: Guo Hui Title: Research Fellow

Supervisor: Huang Hui Title: Intermediate Engineer

June 2023

# 摘要

随着工艺制程的进步，芯片的规模与设计复杂度也在不断提升，在设计芯片初期，需要根据芯片的功能与性能需求，设计配置芯片的架构方案，并对设计方案进行评估，验证和调整。对于处理器而言，处理器的性能可以通过升级工艺制程或优化处理器微架构等方式进行迭代提升，微架构优化需要对所设计的处理器进行精确建模和性能校准，才能获得较为准确地反映处理器设计，这就需要用到体系结构模拟器来辅助微架构研究，缩短设计周期，降低开发成本。

Gem5 作为较具有代表性的面向计算机体系结构的软件仿真模拟器，内置许多模拟器构建所需的模型组件，但在模拟器建模方面，对于 RISC-V 指令集的支持工作还正在不断完善中，RISC-V 由于其开源、简单、模块化等特点，受到许多设计人员的追捧，部分扩展集正在探索修订中。本文将通过查阅文献资料，在 gem5 上配置与补充模块，完成对 RISC-V 向量扩展集的处理器微体系结构的配置与支持。实现在 gem5 上对 RISC-V 向量集的微体系结构的配置实现可以帮助扩展 RISC-V 生态系统，促进向量指令和向量微架构的研究和开发，并为后续研究向量架构的人员提供一种向量扩展实现方法。

本文重点研究了基于 CRISTÓBAL RAMÍREZ 团队的 RISC-V 向量扩展实现，通过分析微架构，对代码进行修改，从而实现对 RISC-V 向量指令集的扩充与模拟。首先对 gem5 内置的顺序执行的 Minor CPU 模型进行分析，然后对处理器流水线进行配置调整，使得向量核与标量核解耦合实现。在标量核部分实现了对向量指令的译码分析与标记，在向量核部分拆分访存指令流和算术计算指令流，并搭建了相应的模块与处理单元。

本文实现了对向量核的微架构模拟，通过增添寄存器分组等方式扩充了模拟器支持的向量指令类型，采用了重命名与重排序缓存的方式解决向量执行过程中可能出现的数据冲突问题；采用多通道模拟向量的并行计算；通过实现对 vta 和 vma 的配置，完成对非活跃元素和尾部元素不可知策略的配置，优化重命名机器中存在的无意义的数据搬运行为；初步加入了对指令异常的检测。

最后，本文通过使用 Riscv-gnu-toolchain 工具链将侧重向量计算的应用程序转化为模拟器能够识别的可执行文件，从而实现对模拟器功能、运行速度、向量执行优化效果的评估。

**关 键 词**：RISC-V， gem5， 向量架构， 处理器模拟器， 微体系结构

# ABSTRACT

As the process progresses, the size and design complexity of the chip is also increasing. At the early stage of designing the chip, the architecture of the chip needs to be designed and configured according to the functional and performance requirements of the chip, and the design solution needs to be evaluated, verified and adjusted. For processors, processor performance can be iteratively improved by upgrading the process or optimizing the processor microarchitecture. Microarchitecture optimization requires accurate modeling and performance calibration of the designed processor to obtain a more accurate reflection of the processor design, which requires architecture simulators to assist microarchitecture research, shorten the design cycle and reduce development costs.

Gem5, as a more representative computer architecture-oriented software simulation simulator, have many built-in components that simulator construction required, but in terms of simulator modeling, the support for RISC-V instruction set is still being improved. RISC-V is sought after by many designers because of its open source, simplicity and modularity, and some of the extension sets are being explored for revision. In this paper, we will complete the configuration and support of the processor microarchitecture for RISC-V vector extension set by reviewing the literature and configuring and supplementing the module on gem5, which can help extend the RISC-V ecosystem, facilitate research and development of vector instructions and vector microarchitecture, and provide a vector extension implementation method for subsequent researchers on vector architectures.

This paper focuses on the implementation of RISC-V vector extensions based on the CRISTÓBAL RAMÍREZ team, by analyzing the microarchitecture and modifying the code in order to achieve the expansion and simulation of the RISC-V vector instruction set. First, we analyze the Minor CPU model of sequential execution built into gem5, and then configure the processor pipeline to decouple the vector core from the scalar core. In the scalar core part, we implement the decoding analysis and marking of vector instructions, and in the vector core part, we split the access instruction stream and the arithmetic computation instruction stream, and build the corresponding modules and processing units.

This paper implements a microarchitectural simulation of vector core, expands the types of

vector instructions supported by the simulator by adding register grouping, adopts renaming and reordering cache to solve the data conflict problem that may occur during vector execution; adopts multi-channel simulation of parallel computation of vectors; completes the configuration of inactive elements and tail element agnostic policies by implementing the configuration of VTA and VMA configuration to optimize the meaningless data handling behavior existing in the renaming machine; the initial incorporation of the detection of instruction anomalies.

Finally, this paper converts the application focusing on vector computation into an executable recognized by the simulator by using the Riscv-gnu-toolchain, which enabling the evaluation of the simulator functionality, running speed, and vector execution optimization effects.

**Keywords**: RISC-V, Gem5, Vector Architecture, Processor Simulator, Microarchitecture

# 插图索引

# 表格索引

# 缩略语对照表

| 缩略语 | 英文全称 | 中文对照 |
| --- | --- | --- |
| AVE | Advanced Vector Extensions | 高级向量扩展 |
| CPU | Central Processing Unit | 中央处理器 |
| DRAM | Dynamic Random Access Memory | 动态随机存取存储器 |
| EEW | Effective Element Width | 有效元素宽 |
| EMUL | Effective Register Group Multiplier | 有效向量寄存器组系数 |
| FPGA | Field Programmable Gate Array | 现场可编程门阵列 |
| FRL | Free Register List | 空闲寄存器表 |
| FS | Full System | 全系统模拟 |
| FU | Functional Unit | 功能单元 |
| GPU | Graphics Processing Unit | 图像处理单元 |
| ISA | Instruction Set Architecture | 指令集体系结构 |
| MIMD | Multiple Instruction Multiple Data | 多指令多数据流 |
| MMX | MultiMedia eXtensions | 多媒体扩展 |
| NOP | No Operation | 空操作 |
| PCI | Peripheral Component Interconnect | 局部总线标准 |
| RAT | Register Alias Table | 向量映射表 |
| RAW | Read After Write | 读后写相关 |
| RDN | Round-down | 向下舍入 |
| RNE | Round-to-nearest-even | 向近舍入，相等偶数舍入 |
| RNU | Round-to-nearest-up | 向近舍入，相等向上舍入 |
| ROB | Reorder Buffer | 重排序缓存 |
| ROD | Round-to-odd | 向奇数舍入 |
| SE | System call Emulation | 系统调用模拟 |
| SIMD | Single Instruction Multiple Data | 单指令多数据流 |
| SSE | Streaming SIMD Extension | 数据流单指令序列 |
| VILL | Vector Type Illegal | 向量类型非法标记 |
| VLEN | Vector Length | 向量长度 |
| VLMAX | Vector Maximum Length | 向量最大长度 |
| VLMUL | Vector Register Group Multiplier | 向量寄存器组系数 |
| VMA | Vector Mask Agnostic | 掩码元素不可知策略标记位 |

| | | |
|---|---|---|
| VRF | Vector Register File | 向量寄存器堆 |
| VSEW | Vector Selected Element Width | 向量选择元素宽 |
| VTA | Vector Tail Agnostic | 尾部元素不可知策略标记位 |
| WAR | Write After Read | 写后读相关 |
| WAW | Write After Write | 写后写相关 |

# 目　录

# 第一章 绪论

## 1.1 研究背景与研究意义

随着芯片复杂度的不断提升，对于大规模集成电路而言，出于优化设计周期，预估设计性能，迭代设计产品，论证设计可行性，降低技术风险等因素的考虑，需要在实际电路设计之前对设计进行仿真模拟。常用的方案有硬件仿真，FPGA 原型设计[1]，虚拟原型设计[2]等。实际生产中常采用多种方案混合保证设计的可行性与仿真结果的准确度，其中虚拟原型设计不依赖于 RTL，可以更加灵活地重新配置设计，帮助确定设计架构组成，并能在一定程度上分析和迭代设计功能，校正设计错误。对于处理器设计而言，简单的数学模型不能精确地体现体系结构方面如分支预测、寄存器重命名、乱序执行等许多设计，需要采用特定的体系结构模拟器来对设计进行建模和仿真，以选定架构配置、预估合理的设计指标，并在中后期校准模型并进行新产品的迭代设计。体系结构模拟器可以在宿主机上运行并模拟目标机器的行为并估算性能，许多体系结构方面的先进成果就是基于模拟器研究优化的[3-5]。

随着处理器对信息处理能力的提高，神经网络计算、人工智能、大数据分析等对数据处理需求量大的学科也在蓬勃发展，产生许多数据计算需求，计算目标则是大量相似度高且相互独立的数据，数据并行计算能很好满足这些计算需求。数据并行处理包含 Single Instruction Multiple Data 单指令多数据流和 Multiple Instruction Multiple Data 多指令多数据流两种[6]，前者使用单个处理器处理译码、发送和数据处理等操作；后者则具有多个处理器，每个处理器发送自身的指令并操作自身的数据，设计较为复杂但更具灵活性，制作成本较前者也更加高昂。

SIMD 具有三种变体：向量体系架构、多媒体 SIMD 指令集扩展和图像处理单元（Graphics Processing Unit）。对于处理器而言，不同指令集对 SIMD 的实现方式不尽相同，大多数指令体系结构实现的是多媒体 SIMD 指令集扩展，如 intel X86 架构为了提高程序计算尤其是浮点方面的计算速度，一直在扩展发展 SIMD 指令。从 1996 年开始的 MultiMedia eXtensions 多媒体扩展，允许在固定长度的向量寄存器上执行一组预定义操作；再到后续不断扩展补充的 Streaming SIMD Extensions 数据流单指令序列；再到更加复杂，更大位宽的 Advanced Vector Extensions 高级向量扩展，都在不断地提高数据的并行计算能力，但也带来了指令数量激增的问题。RISC-V 作为新兴的指令集，为了避免传统增量型指令集的新处理器必须实现之前的所有扩展以保证应用前后兼容而带来的体量膨胀等问题，选用了较为简约的模块化设计。RISC-V 的基础是 RISC-V 整型实现，具有基础但完整的指令功能，而后续其他的扩展指令则以模块

化的方式根据设计目标而进行可选式的扩展增添，这使得 RISC-V 更富通用性。在数据并行计算方面，RISC-V 提供了针对数字信号处理的 P 扩展和针对向量处理的 V 扩展。向量 V 扩展提供的向量体系结构不同于多媒体 SIMD 指令集扩展，向量扩展将向量长度与每个时钟周期可以进行的最大操作数分离，向量寄存器的大小由具体实现决定，程序根据使用的数据类型和宽度标记向量寄存器，不需要对不同类型的操作目标配置相应的指令，从而大幅减少向量指令的数量。向量架构能在较低功耗的情况下高效能地完成数据并行计算，并较好地加速数据密集型等应用。

RISC-V 指令集作为新兴指令集在 gem5 中的支持不像 x86、ARM 等老牌架构一样完整[7]，目前还在不断研究优化中。在行业中常使用处理器模拟器来研究新的架构特性和指令功能，并使用处理器模拟器拟合模拟实际情况，预评估产品性能。在 gem5 中，官方分支尚未完善对 RISC-V 向量扩展集的支持，研究构建 RISC-V 指令在 gem5 中支持与扩展有助于改善 RISC-V 指令集在处理器模拟器方面的空缺。

## 1.2 国内外研究现状

向量机活跃于 1975 年至 1990 年之间，在 1976 年，Seymour Cray 就在超级计算机 Cary-1 中使用了向量架构[8]，但由于向量架构需要大量访存带宽，硬件成本较高，逐渐被大规模并行计算机取代。

伴随着大数据计算等向量优势学科的飞速发展，向量架构又进入了大众的视野，RISC-V 在多数据处理方面提供了向量操作 V 扩展和组合单指令多数据 Packed SIMD 扩展，二者都用于加速多数据处理。P 扩展更趋向于处理特殊领域计算，常用于嵌入式 DSP 处理器或者 SIMD 压缩指令，V 扩展则用于高性能低功耗运算，在功能上基本包含 P 扩展的内容。

RISC-V 由于其开源、模块化设计、架构简单等特性，吸引了不少学校机构对其进行研究扩充，并涌现许多成果，如平头哥的玄铁系列处理器[9]，中科院的香山处理器[10]，上海交通大学的蓬莱可信执行环境[11]等。目前开发并提交到社区的模拟器共计 24 种，其中 gem5 模拟器主要用途是进行微架构模拟，gem5 官方分支对 RISC-V 指令集支持 RV64GC 指令集[12]，由于向量指令集还处于不断修订讨论之中，推出稳定版本的时间也较短，目前在官方各分支实现中都不包含 V 扩展。

Gem5 对于 RISC-V 的支持始于 2017 年，Alec Roelke 团队在系统调用模式对 RISC-V 单核方面的各类指令进行了实现和填充，包含整型指令集、乘除指令集、原子指令集、浮点指令集等，并结合外部插件模拟了实际的流程设计并给出版图布局和温度等信息，并与 Chisel 模拟器和 FPGA 仿真结果进行对比调整，使得性能统计上误差小于 10%[13]。证明了 gem5 对 RISC-V 指令集模拟的可行性与准确性，并给出了

gem5 中对 RISC-V 代码的最初支持版本。

到了 2018 年，康奈尔大学的 Tuan Ta 团队添加了多种系统调用支持，并补足了 Alec 团队中缺失的部分指令，并在 gem5 中实现了多核 RISC-V 指令系统支持[14]。Tuan Ta 在单核验证方面使用 RISC-V tool chain 工具编译，并专门编写向量应用来验证多核系统的正确性。

2021 年 6 月，华为研究中心的 Peter Yuen Ho Hin 团队增加了处理分配中断、实现物理内存检查机制用途的模块，并修复特权指令和 CPU 模型中的错误，在 gem5 全系统仿真模式方面添加了 RISC-V 支持。并成功在模拟器上运行 Linux 系统作为负载并运行了部分测试软件[15]。

目前，许多团队在官方分支上进行定制化的扩展研究，但是研究向量化方向的研究团队较少，在 2021 年 9 月底，RISC-V 向量扩展推出了 1.0 版本[16]，该版本是向量扩展的第一个稳定版本，且后续修改都将保持对 1.0 版本的兼容性，可以依照此版本进行工具链开发与功能模拟器的实现。

2021 年 10 月，CRISTÓBAL RAMÍREZ 等人在 gem5 模拟器加入了 RISC-V 向量架构模型，初步加入了 gem5 对 0.7 版本的向量扩展指令支持，并研究了不同矢量化应用程序实现的矢量化程度并对性能与期望性能进行了一定的对比评估[17]。

上述各团队对于 gem5 的扩充与配置，不断丰富 RISC-V 在 gem5 中的实现情况，但还是存在模拟器支持的规范版本较旧，存在规范定义变更，缺少状态寄存器及部分指令，不支持寄存器分组，不支持不可知策略，不支持异常检测等问题。

## 1.3 主要研究内容与论文组织安排

CRISTÓBAL RAMÍREZ 团队对向量架构进行了基础的构建，并提供了一套特制化程序用以评估向量化情况。本模拟器将针对上述问题，研究扩展该团队设计的 RISC-V 向量模拟器，分析该模拟器对于向量扩展指令集相关指令的实现情况与大概架构，对指令条目进行扩充实现，并基于该模拟器进行扩展优化，提高模拟器的通用性和可扩展性。

第一章：绪论。本章首先介绍体系结构模拟器的概念以及选择 RISC-V 向量指令集的原因。然后介绍了目前国内外对 RISC-V 发展近况及 gem5 模拟器对 RISC-V 指令集的支持实现情况。最后对本章内容以及论文组织结构进行了一定的说明。

第二章：模拟器及指令体系基础。本章在功能组件和部分实现方面介绍了本文使用到的体系结构模拟环境 gem5，介绍了体系结构模拟器的构成，RISC-V 向量扩展集的分类以及向量计算涉及的概念与计算方法。

第三章：模拟器结构设计。本章首先对 RISC-V 向量模拟器的总体结构进行介绍，

并对标量核的流水线改动，向量核的微架构，模块数据交互与功能划分等方面进行说明。

第四章：模拟器功能实现。本章对模拟器指令实现方法进行分析，说明了模拟器的译码功能实现方式，然后说明了对模拟器添加的指令类型及功能单元的实现方式，并说明了加速模拟器运行所做的改动优化，最后对模拟器进行异常检测添加。

第五章：模拟器测试。本章介绍了 gem5 二进制文件的编译构建，对模拟器使用的测试方式、测试软件以及测试内容与倾向进行说明，然后罗列测试结果并对结果进行简要分析。

第六章：总结与展望。本章对本文中的和成果进行总结，并对需要进一步研究的工作进行分析和展望。

# 第二章　模拟器和指令体系基础

## 2.1 模拟器基础

模拟器用以辅助集成电路设计可以追溯到 1980 年代[18]。体系结构模拟器是体系结构量化分析的重要手段，模拟器在开发不同时间周期具有不同的功能。模拟器可以辅助进行处理器微架构探索完善、芯片逻辑验证、硅后验证环境搭建、系统软件开发等工作[19]。

根据模拟的精细程度可将模拟器大致划分为功能模拟器和性能模拟器两类，前者模拟目标系统如指令语义等指令集体系结构；后者除了模拟指令功能，还可模拟流水线，分支预测等微体系结构[20]。模拟器根据模拟精度还可以划分为功能级别、时钟级别、寄存器传输级别、FPGA 级别仿真[21]，由表 2.1 可看出不同模拟精度级别对应的代码修改时间、仿真模拟所需时长、仿真结果准确度等区别。本模拟器的设计目标为时钟仿真级别的性能模拟器，会通过建立模型实现向量架构指令，并通过各种模块交互从而模拟微体系细节。时钟级仿真模拟器可在设计初期以较为均衡的性能特性，对设计进行预研，帮助产品迭代更新。

表2.1 不同仿真级别的耗时与性能对比表[14]

| | 代码修改耗时 | 模拟耗时 | 准确度 |
|---|---|---|---|
| 功能级仿真 | ++ | ++ | -- |
| 时钟级仿真 | + | + | - |
| 寄存器传输级仿真 | - | -- | ++ |
| 现场可编程门阵列级别仿真 | -- | ++ | ++ |

本次模拟器使用的环境工具 gem5 是一个时钟周期级别的模块化事件驱动的计算机系统模拟环境，提供许多不同的模型组件[22]。在 gem5 中，使用 tick 指代模拟器每次进行迭代的最小时间间隔，并使用 tick 配置模拟的精度与速度，tick 设定得越小，模拟器会更加精确，同时运行速度也会更慢。模拟器会根据事件的时间戳，在对应的流水线阶段按顺序执行事件，从而能够更加准确地模拟实际情况。

由图 2.1 可以清晰看出 gem5 的模块化特性。在 gem5 中，模型组件各自完成一部分系统功能实现，并通过端口接口进行连接与数据交互，各模型组件可以按照设计目标进行任意搭配。一个 gem5 系统可以包含 CPU 核、DRAM 模型、片上互联、高速缓存、I/O 设备等组件，还能根据需求添加 GPU 模型，硬盘控制器，PCI 总线等额外部件。在 CPU 模型的实现上，采用功能模型（ISA-specific models）与定时模型（CPU Models）共同实现对 CPU 核的模拟。内存模型的实现上也有传统缓存模型和 Ruby 缓

存模型可以选择。此外，Gem5 还整合了外部接口对接其他计算评估工具以获得更加准确的模拟数据。通过构建精确的模型与系统，对其进行调整修正，可以得到接近真实情况仿真结果。定制化修改并配置这些模型参数并使用 gem5 模拟模型之间的事件交互，可以实现对目标系统的构建，并基于此系统模型进行研究改进。

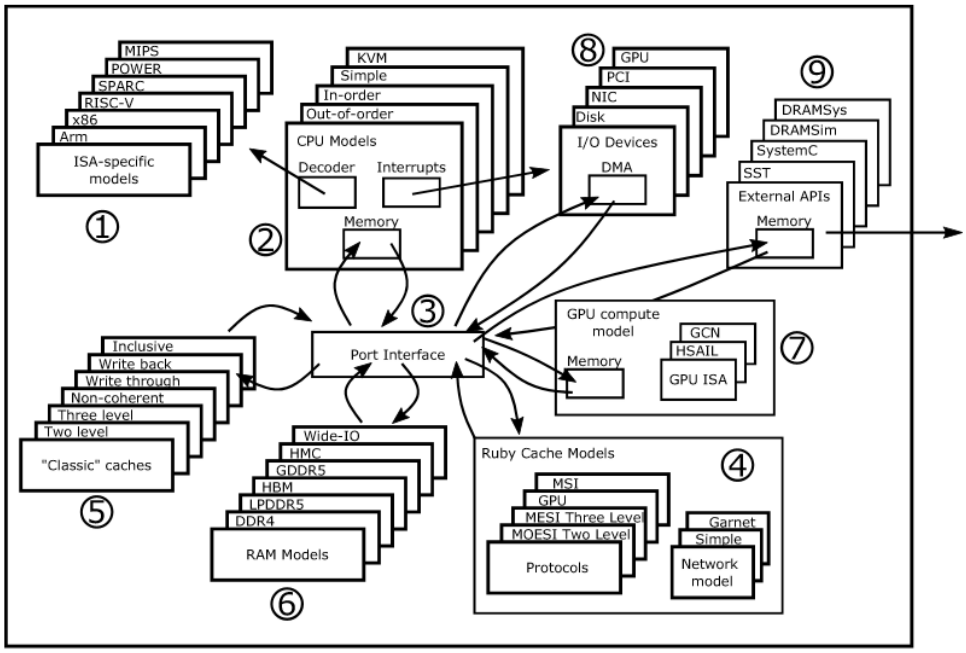


图2.1 gem5 模块连接示意图[22]

Gem5 含有两个模式，System call 系统调用模式和 Full System 全系统仿真模式。全系统仿真如图 2.2(a)所示，更注重系统调用层面，可以启动完整的基于 Linux 的操作系统并通过操作系统运行软件，还支持各种输入输出设备和外设，从而实现更高的仿真精度，但缺点是运行速度较慢。图 2.2(b)所示的系统调用模式则更加注重对 CPU 和内存系统的仿真，直接调用应用软件并简化地址翻译模型，运行速度较快。本模拟器将更加注重系统调用模式的具体实现。

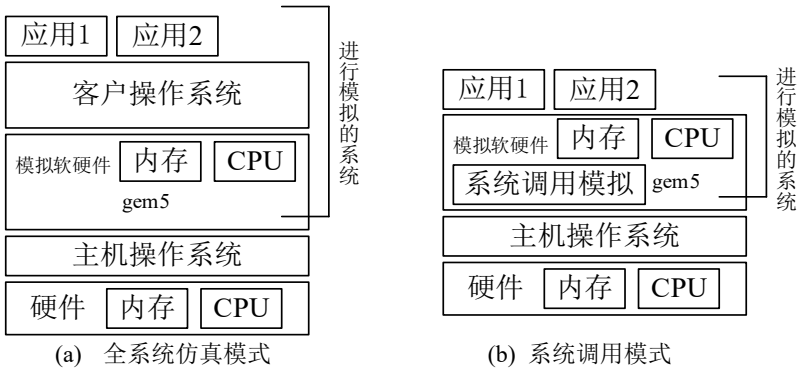


图2.2 gem5 的两个仿真模式示意[22]

图 2.3 展示了一个 gem5 中的简单系统，CPU 通过一级指令缓存与一级数据缓存分别与一二级缓存总线相连，并通过总线与二级缓存进行数据交互。二级缓存与系统总线相连，从而对挂载在总线上的各种模块进行交互。gem5 运行过程会将配置好的组件模型进行链接，然后调用系统负载，负载依照配置模式的不同可以是操作系统也可以是应用程序。根据执行程序的过程、结果、运行速度等数据，以及运行过程中可定制化的让 gem5 收集的统计数据，可以对指令功能和微架构的实现情况进行评估。在设计处理器的过程中，通过不断修改添加指令和调整微体系结构，运行符合目标应用场景的负载程序，针对性收集数据进行分析并反馈到架构设计配置进行迭代，可以实现对处理器的调整、优化与性能提升。

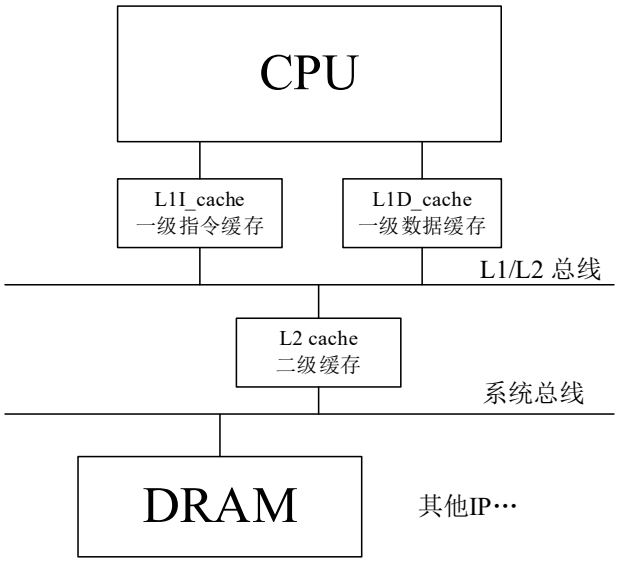


图2.3 简单的 gem5 系统示意图

对于 RISC-V V 处理器模拟器而言，关注重点是 CPU 核的设计，CPU 核心包含了指令功能实现以及对处理器微架构的模拟仿真。CPU 模型负责模拟处理器性能，ISA 指令集体系功能描述负责模拟处理器功能，二者通过动态指令类交互。CPU 模型包含寄存器等数据结构、处理器流水线、处理器行为等方面的配置定义，图 2.4 中包含一个简单的取指、译码、执行、访存、写回的五级流水线。动态指令类由静态指令类扩展而来，静态指令类包含了指令功能标记，源寄存器与目的寄存器编号，执行地址计算与内存访问的虚函数，译码指令方法等指令信息。动态指令除静态指令包含的信息之外还包含指令计数器和一些在运行时可修改设置的动态信息。Gem5 的特点是事件驱动，在 CPU 模型的不同流水阶段，定义的事件会顺序驱动结算。在译码阶段，当需要执行对应指令时，事件会驱动动态指令类将源数据和指令类型传递给 ISA，然后对 ISA 进行指令解析。在执行阶段，会调用指令相关的算术逻辑单元对指令进行计

算，然后将值通过动态指令传递回 CPU 模型，从而完成对指令的实现[20]。指令集体系结构使用 ISA 语言描述，然后使用 ISA 解析程序将指令进行替换解析，生成对应的指令类的声明与定义。

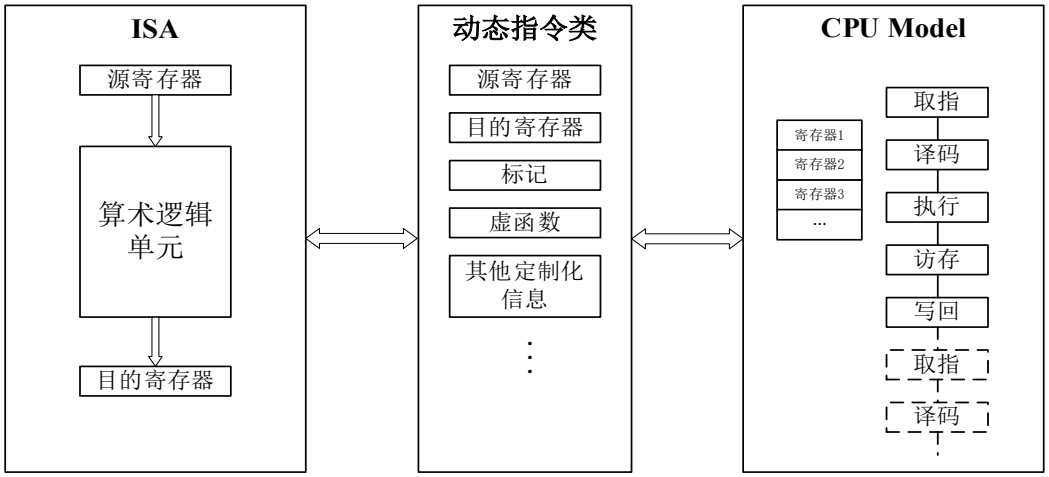


图2.4 ISA 与 CPU 交互示意图

对于标量指令而言，可以使用 ISA 语言描述行为从而模拟指令功能，然后在 CPU 模型中实现对微体系架构部分的模拟。由于向量指令功能复杂，指令的实现需要较长的时间周期，如果将向量指令实现与标量指令耦合在一起设计，在面积、功耗、频率等参数配置上二者会互相制约。因此，本模型采用将标量核与向量核解耦合的方式实现，通过构建新的模块对向量核的指令实现和微架构模拟进行仿真模拟。标量核与向量核之间使用接口交互，指令首先通过标量核初步译码，若指令是向量指令，则会先初步拆解指令，将指令的各种字段提取并打包，然后将译码结果发送到向量核进行执行处理。执行完毕之后，向量核将指令执行完毕的信息反馈给标量核。因此在指令描述方面，标量核实现向量指令译码，指令功能标记和指令类型分类，向量执行部分由向量核进行。

## 2.2 向量架构体系

向量架构的核心在于使用多通道提高同一时刻数据计算的带宽，从而加速指令计算速度。在向量架构体系中，向量结构收集在存储器中散布的数据元素集，将它们放在一些大型的顺序寄存器堆中，再对这些寄存器堆中的数据进行并行操作，最终将结果放回存储器[16]。以加法运算为例，假设加法器单元在一个时钟周期内可以进行一个加法操作，图 2.5(a)中计算宽度为十的向量加法需要十个时钟周期，而图 2.5(b)中由于使用了四个计算通道，十个元素交错排列在四个通道中，使得一个时钟周期可以同时进行四个加法运算，只需要三个时钟周期就可以完成宽度为十的向量加法计算。相

比于标量指令而言，向量指令可以实现单条指令对数据向量进行操作，从而实现对数十个独立数据元素进行寄存器与寄存器之间的操作，在面对大量重复类型数据计算时可以提升效率。由于指令需求量较少，一次可以执行多个元素之间的计算，因此计算效率较标量计算更快，但由于使用了多通道和多个计算单元，需要的面积比标量处理器 大。

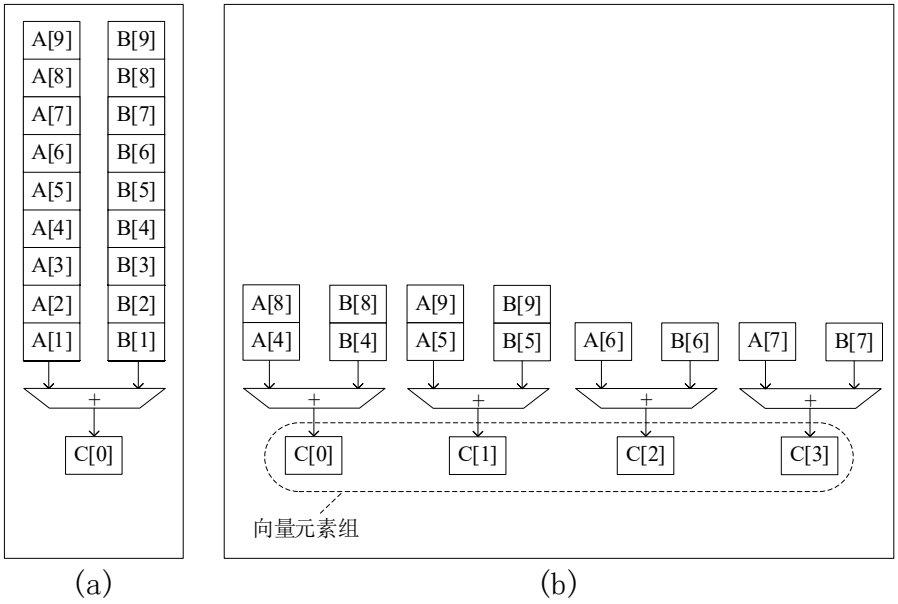


图2.5 加法指令单通道和多通道计算单元示意图

与传统 SIMD 技术相同，向量架构也是通过拓宽寄存器位宽实现数据并行计算，但向量寄存器使用的是可变长的寄存器，硬件会识别软件中定义的数据类型和计算总量，然后自动选择数据类型进行计算，在汇编层面中一条指令可以通过不同的状态寄存器配置实现对不同长度的数据进行处理。这个类型识别是动态隐式的，利于软件在不同向量位宽的处理器实现之间进行移植，使得软硬件解耦，对编程人员更加友好。传统的 SIMD 技术将寄存器位宽与指令编码绑定，同一类计算不同的数据类型需要使用多个指令编码，且在使用长数据的时候会削减一定的性能，在汇编层面上需要使用不同的指令才能进行不同数据类型计算，且编译好的软件可能需要进行修改重新编译链接才可在新的硬件实现上运行。

向量架构是将来处理器在多数据发展的一个方向，在计算机体系结构量化方法一书中，第六版修订增添了 RISC-V 的向量架构相关内容[23]。一个向量扩展架构通常包含以下几个部分：标量寄存器、向量寄存器、向量功能单元、向量存取单元等，图 2.6 所示为一个包含标量架构的 RV64V 架构表示，其中标量寄存器可以提供数据给向量计算单元，以实现标量与向量之间的运算。RV64V 具有 32 个 64 位宽的向量寄存器，

每个向量寄存器都能够储存一个向量，所有的向量寄存器与标量寄存器都具有足够的读写端口与计算单元相连。功能计算单元是流水线化的，每个周期都能进行一个新的操作，可以进行标量计算和向量计算，且存在检测指令冲突的控制单元。加载和存储单元同样也是流水线化的，用于从主内存中存取计算所需的数据，在初始化之后，每个周期都可以获取与带宽宽度大小一致的数据量。

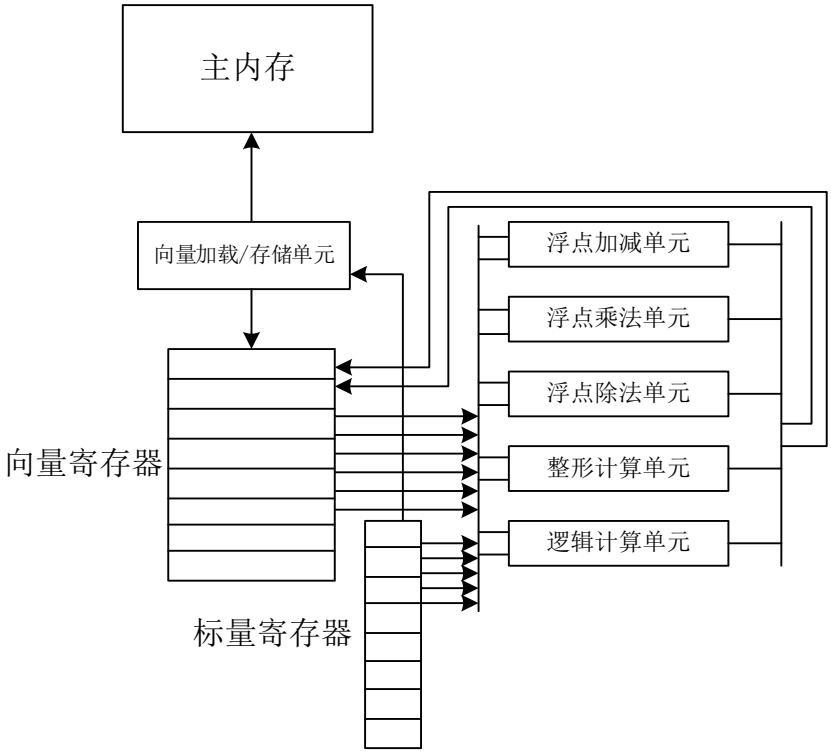


图2.6 RV64V 向量架构示意图[23]

## 2.3 向量指令集

RISC-V 向量 V 扩展为 RISC-V 许多标量指令添加了对应的向量操作[16]。向量扩展添加了 32 个附带状态位 VLEN 的向量寄存器 v0~v31，还添加了 7 个名为 vstart、vxsat、vxrm、vcsr、vl、vlenb、vtype 的状态寄存器。状态寄存器的功能如表 2.2 所示。其中 vtype 为向量类型寄存器，使用向量配置指令更新，包含 vill、vma、vta、vsew、vlmul 等字段，其中 vsew 用于标记当前元素宽度，vlmul 用于标记本次计算需要使用到的寄存器数量。大多数向量指令可以进行掩码操作来屏蔽向量寄存器中的部分元素从而实现条件执行，vta 和 vma 的值用以选定掩码不受干扰或者掩码不可知策略，从而控制掩码操作下对应元素的行为。vill 则用于标记向量配置指令发送了非法参数值。以下将会对向量扩展的指令类型进行简单介绍，然后在 2.4 小节中对向量计算进行说明。

表2.2 向量状态寄存器定义与功能示意表

| | |
|---|---|
| vstart | 定义向量执行第一个元素索引，与异常恢复相关 |
| vxsat | 表示输出结果进行饱和截位 |
| vxrm | 配置定点舍入模式，如四舍五入模式，取整模式等 |
| vcsr | 打包 vxsat 和 vxrm 两个状态寄存器，可以加速对上下文的保存或恢复 |
| vl | 定义向量指令执行向量元素个数 |
| vlenb | 储存向量长度对应的字节数 |
| vtype | 向量类型寄存器，包含向量循环计算相关参数，使用向量配置指令更新 |

RISC-V 的向量扩展主要分为三类指令，向量配置指令、向量访存指令以及向量算术计算指令。RISC-V 向量扩展将配置型指令和计算指令划分为 OP-V（1010111B）型，而访存指令操作码使用浮点类型指令格式 LOAD-FP（0000111B）和 STORE-FP（0100111B）型进行扩展，以下将分别介绍。

RISC-V 在向量扩展方面不同于其他架构的实现方式，指令的数据类型与长度跟指令的操作码分割，使用对应的状态寄存器标记，在进行计算之前，若指令计算与旧配置发生了变动，编译器会使用指令进行调整。这可以节约指令编码，使得向量集更加精简，主要依靠向量配置指令进行实现。

向量指令使用 14-12 位的 FUNCT3 字段来将 OP-V 类型的指令划分为几种子类指令，分别为 OPIVV、OPFVV、OPMVV、OPIVI、OPIVX、OPFVF、OPMVX、向量配置指令八种，其中 OP 代表 OP-V 类型，而后接着的字母代表操作数据类型，I 代表整数操作、F 代表浮点操作、M 代表特殊操作，如归约计算、平均计算、位扩展操作、类型转换等。最后面的两个字母代表两个源操作数来源，其中 V 代表向量寄存器、I 代表立即数、X 代表整数寄存器、F 代表浮点寄存器。

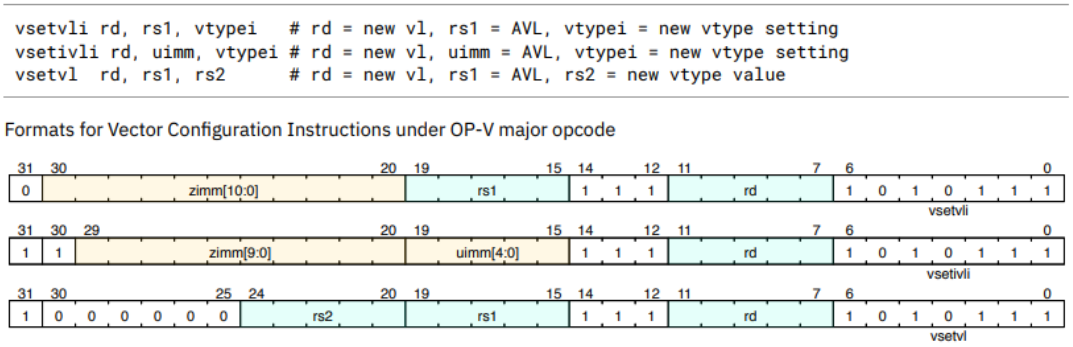


图2.7 向量配置指令格式示意图[16]

向量配置指令只有 vsetvli、vsetivli、vsetvl 三种，向量配置指令会根据有效向量

长度 AVL 的值，配置向量长度 vl 的值，并写入到 rd 寄存器中，并根据 vtype 配置状态寄存器以规定后续向量计算时向量元素的识别方式，从而对向量计算循环进行配置。指令类型分属 OP-V 型，指令如图 2.7 所示。

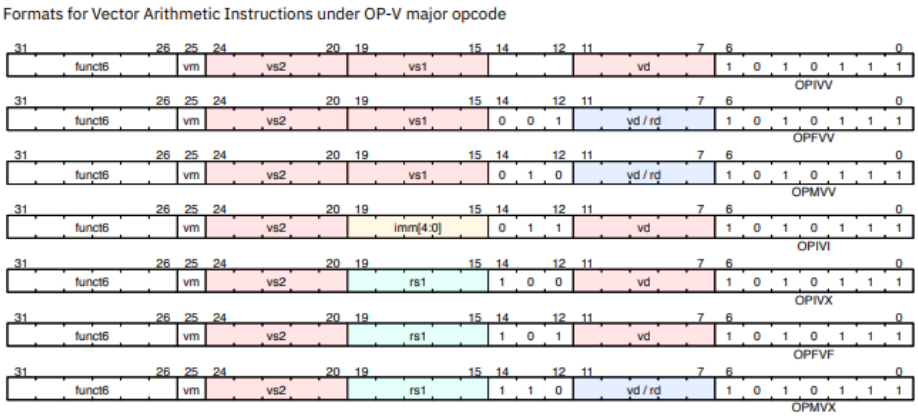


图2.8 向量算术指令格式示意图[16]

向量算术指令指令格式如图 2.8 所示。向量算术指令存在一套命名规则，即 v+目标类型种类+计算操作名称+有无符号.第一个源向量类型+第二个源向量类型。目标类型种类可以是普通型，拓宽型 w，紧缩型 n，掩码型 m 四种，拓宽型即目的寄存器的元素宽度是源寄存器元素宽度的 2 倍，缩宽型即目的寄存器的元素宽度是源寄存器元素宽度的 1/2，掩码型代表将写入谓词寄存器。操作名称包含算术计算 add、sub、madc 等，比较 seq、sne、slt 等，位操作 zext、sext、and、or 等，复合操作 macc、nmsac 等，寄存器合并操作 merge，归约计算 redsum、redmax 等，浮点转换操作 fcvt、fwcvt 等。符号类型默认为有符号运算，无符号用 unsigned 中的 u 替代。第一个源向量寄存器类型可以是向量 v，宽向量 w，掩码 m。第二个源向量寄存器类型可以是向量 v，标量 s，浮点寄存器 f，立即数 i，无符号立即数 u，掩码 m。

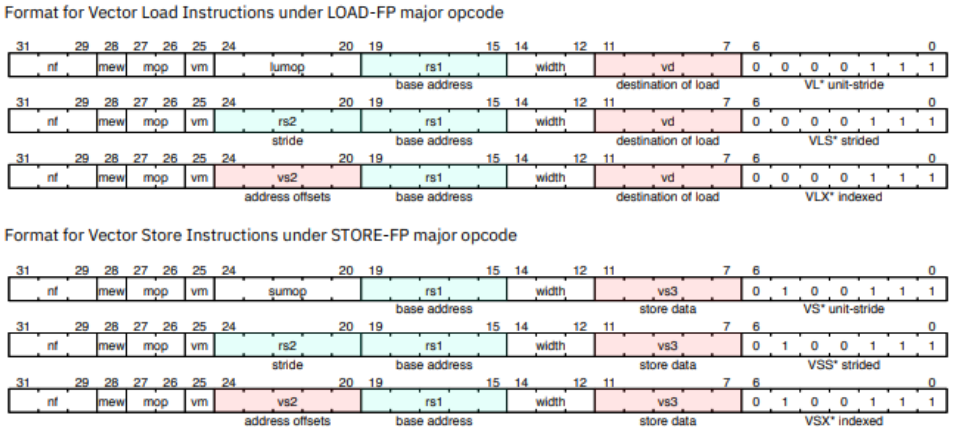


图2.9 向量访存指令示意图[16]

向量访存操作指令格式如图 2.9 所示。向量存储指令使用 27-26 位即 mop 字段来识别存取类型，存取操作分为单位跨步寻址、定长跨步寻址、索引寻址。24-20 位的 LUMOP 字段用来指定单位跨步寻址的寻址模式。

向量单位跨步寻址为存指令 vld 及逆操作 vst，指令会按照提供的基地址开始以单位跨步操作访问在存储单元中连续的元素，获取向量长度寄存器 vl 中规定的元素数量。对于多维数据，访问不一定是顺序的，如二维数组行优先序进行存储，对列元素进行顺序访问。向量跨步寻址为 vlds 及逆操作 vsts，通过跨步数据传输进行实现，由提供的基地址开始访问第一个元素，然后按照给定标量的字节偏移访问后续元素，直至 vl 个元素为止。向量索引寻址为 vldx 和逆操作 vstx，通过额外提供一个向量，将向量中每个元素内容添加到基址中以计算出所需元素的有效位置，之后将地址序列发送到内存以获得目标元素向量。

## 2.4 向量计算

在向量计算过程，需要根据向量指令类型使用对应向量状态寄存器中的值指示向量的计算方式，以下将对向量计算与各寄存器功能进行介绍。

向量计算过程中，对于向量寄存器中的元素，按照向量元素索引分为几类。预启动元素 prestart，指索引小于 vstart 寄存器初始值的元素。主体元素 body 包含活跃元素 active 和非活跃元素 inactive 两类，指的是索引大于 vstart 寄存器值且小于向量长度 vl 部分的元素。活跃元素指向量指令执行期间，在当前向量长度范围内掩码寄存器 v0 中对应元素索引位的值为 1 的元素，活跃元素可以引发异常并更新目标向量寄存器组。非活跃元素指向量指令执行期间，在当前向量长度范围内但是掩码寄存器 v0 对应元素索引位为 0 的元素。尾部元素指超出当前向量长度设置的元素。其中，预启动元素，非活跃元素和尾部元素不引发异常也不更新目标向量寄存器组，详细范围划分见图 2.10。

```
for element index x
prestart(x) = (0 <= x < vstart)
body(x)     = (vstart <= x < vl)
tail(x)     = (vl <= x < max(VLMAX,VLEN/SEW))
mask(x)     = unmasked || v0.mask[x] == 1
active(x)   = body(x) && mask(x)
inactive(x) = body(x) && !mask(x)
```

图2.10 向量循环元素分类

一次向量计算会对当前循环中的活跃元素进行计算并更新目的向量寄存器对应

的元素位，如果没有额外配置设置，非活跃元素和尾部元素需要采取 undistrubed 不受干扰策略，即目标向量中对应索引中的元素值在计算时不受影响，而 agnostic 不可知策略允许将这部分元素值保持之前的值不变或者被重写为 1，是否进行重写可以是不确定的。

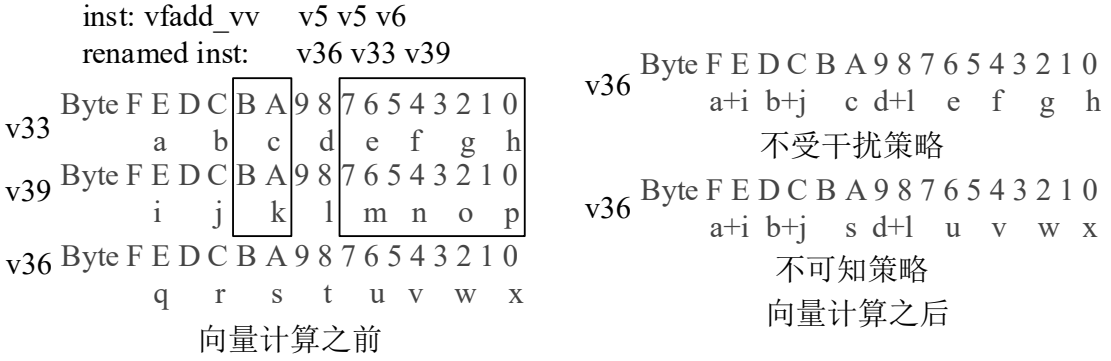


图2.11 向量计算策略示意图

寄存器 vta 为 Vector Tail Agnostic 定义了尾部元素的处理策略，寄存器 vma 为 Vector Mask Agnostic 定义了非活跃元素的处理策略。对于带有向量寄存器重命名的设计而言，设置不可知策略可以忽略无效元素内容，从而提升计算效率。如图 2.11 所示，向量计算之前使用方框圈出元素为被掩码不进行计算的元素，若使用不受干扰策略，目的寄存器中经过功能单元计算之后，不进行更新的元素需要保持原来的值，对于带有重命名策略的设计需要根据重命名映射表，查到原有的映射关系，再把这部分元素的值先读出然后写到重命名后的向量寄存器对应位置上，即需要将方框圈出的元素从 v33 寄存器中读取出，然后写入到 v36 寄存器中。使用不可知策略则允许这部分值为 v36 中寄存器对应元素值或者置 1。对于后续的指令而言没有用处的元素可以通过设置不可知策略避免浪费性能与功耗搬运数据。若是后续计算对尾部元素或者非活跃元素有需求，则可以通过设定 vta 和 vma 的值，使得设计去查找映射表，将元素值保留。尾部元素的掩码操作不受 vta 影响被视为使用不可知策略，具体的策略配置如表 2.3 所示。

表2.3 向量元素策略配置表

| vta | vma | 尾部元素 | 非活跃元素 |
|---|---|---|---|
| 0 | 0 | 不受干扰策略 | 不受干扰策略 |
| 0 | 1 | 不受干扰策略 | 不可知策略 |
| 1 | 0 | 不可知策略 | 不受干扰策略 |
| 1 | 1 | 不可知策略 | 不可知策略 |

向量配置指令可以配置对应的向量寄存器，从而配置向量循环的各类参数。其中vtype 向量寄存器中包含两个重要变量，VSEW 和 VLMUL。VSEW 为向量选择元素宽 Vector Selected Element Width，用来指定当前向量寄存器中用于进行操作的元素位宽，目的寄存器的元素有效位宽也可以根据指令类型与 VSEW 计算得出。VLMUL 为向量寄存器组系数 Vector Register Group Multiplier，用来表示寄存器分组，可以为 1/8、1/4、1/2、1、2、4、8，代表将指定系数倍的寄存器打包分组，使得一条指令可以对打包完毕的寄存器组整体进行操作。通过 SEW 和 LMUL 配合，可以确定在当前配置下，可以操作的向量最大元素数量 Vector Max Length，也就是 VLMAX 的值，计算公式为：

$$VLMAX = \frac{VLEN}{SEW} * LMUL \tag{2-1}$$

如果向量寄存器长度为 128 位，当 SEW 设置为 32 位，LMUL 设置为 4，此时进行计算需要选定四个向量寄存器进行合并分组，一条向量指令最大可以计算 16 个元素，每个元素宽为 32 位，实际计算使用的向量元素数由配置指令设定，并储存到状态寄存器 vl 中。如图 2.12 所示，RISC-V 默认采用的是小端格式[24]，可以看到元素优先从向量寄存器的低位开始填充。同理，当 LMUL 小于 1 时则代表使用向量寄存器中的低位部分进行计算操作。此时，从低位开始填充寄存器，高位依照 vta 规定填充对应数值。

VLEN=128b, SEW=32b, LMUL=4

| Byte | F E D C | B A 9 8 | 7 6 5 4 | 3 2 1 0 |
|---|---|---|---|---|
| v4*n | 3 | 2 | 1 | 0 |
| v4*n+1 | 7 | 6 | 5 | 4 |
| v4*n+2 | B | A | 9 | 8 |
| v4*n+3 | F | E | D | C |

图2.12 选择向量长度与寄存器系数的配置示意图

向量支持元素混合宽度操作，这样可以在不改变计算元素个数 vl 的情况下操作多个数据位宽不同的向量进行计算，从而减少指令配置下 vl 的更新频率，如图 2.13 所示。前述 LMUL 允许小于 1 是为了在混合位宽操作下提高向量寄存器利用率，在这种情况下，仅需要同倍数增大或者缩减 SEW 和 LMUL，即可更新 vtype 且不改变 vl 的值。

```
Example VLEN=128b, with SEW/LMUL=16

Byte      F E D C B A 9 8 7 6 5 4 3 2 1 0
vn        - - - - - - - - 7 6 5 4 3 2 1 0  SEW=8b, LMUL=1/2

vn          7   6   5   4   3   2   1   0  SEW=16b, LMUL=1

v2*n            3       2       1       0  SEW=32b, LMUL=2
v2*n+1          7       6       5       4

v4*n                    1               0  SEW=64b, LMUL=4
v4*n+1                  3               2
v4*n+2                  5               4
v4*n+3                  7               6
```

图2.13 混合位宽向量计算

向量的所有操作数都包含一个有效元素宽度 Effective Element Width 和一个有效寄存器组系数 Effective Register Group Multiplier，在一般情况下，有效元素宽度与选择元素宽度相等，有效寄存器组系数与向量寄存器组系数相等，意味着进行向量指令计算之后元素个数和位宽不变。对于向量拓宽或者缩宽操作来说，向量目的向量可能有效元素宽度是源向量的两倍或一半，此时也应该保证计算前后元素个数一致，即：

$$\frac{EEW}{EMUL}=\frac{SEW}{LMUL} \tag{2-2}$$

绝大多数的向量指令支持掩码操作，被掩码的元素操作不产生异常。使用 25 位字段 vm 标记，当 vm 为 0 时表示为对向量指令进行掩码操作，vm 为 1 时则不对指令进行掩码操作。在向量计算时，采用向量寄存器 v0 保存向量中元素的掩码值，v0 中对应源向量中的元素索引位的值若为 1，则代表允许该元素进行向量操作，计算结果并更新目的向量寄存器，若对应元素位中的值为 0，则代表该元素被屏蔽，不进行向量操作，计算结果值由上述 vta 和 vma 配置的策略决定。

当向量需要进行四舍五入或取整时，使用寄存器 xrm 规定舍入方式，在 xrm 为 0x0 时，为 round-to-nearest-up，向距离近的方向进行舍入，当与两边的距离相等则向上舍入。xrm 为 0x1 时，为 round-to-nearest-even，向距离近的方向进行舍入，当与两边距离相等时向偶数方向舍入。xrm 为 0x2 时为 round-down，向下舍入，取移位后的值。xrm 为 0x3 时为 round-to-odd，舍入到奇数方向。例如，对于源操作数 v，需要截掉 d 位数据，则最终结果应为（v>>d）+r，r 为使用 xrm 规定模式计算出的值，计算方式具体见表 2.4。

表2.4 舍入模式与舍入增量计算方式示意表

| vxrm[1:0] | | 缩写 | 舍入模式 | 舍入增量，r |
|---|---|---|---|---|
| 0 | 0 | rnu | 舍入最近方向，相等则向上舍入 | v[d-1] |
| 0 | 1 | rne | 舍入最近方向，相等向偶数舍入 | v[d-1] & (v[d-2:0]≠0 \| v[d]) |
| 1 | 0 | rdn | 向下舍入 | 0 |
| 1 | 1 | rod | 舍入到奇数方向 | !v[d] & v[d-1:0]≠0 |

## 2.5 本章小结

本章简要描述了模拟器的构成与优劣势，简要介绍了模拟工具 gem5 的特点。其次对向量架构体系的实现方式与基础构成，RISC-V 向量扩展集的指令分类与向量循环计算定义等基础概念进行了说明。RISC-V 作为发展迅速的指令集体系结构，由于其开源、简洁等良好特性受到许多开发团队的欢迎，而 gem5 作为研究体系结构的常用工具而言对 RISC-V 指令集的支持还在不断扩展中，本模拟器将实现 RISC-V 向量扩展集在 gem5 中的实现以及向量核的微体系模拟。

# 第三章　模拟器结构设计

## 3.1 模拟器结构总览

本模拟器是基于 CRISTÓBAL RAMÍREZ 团队设计的时钟仿真级别的性能模拟器的更新优化。整体系统的一种搭建情况如图 3.1 所示。

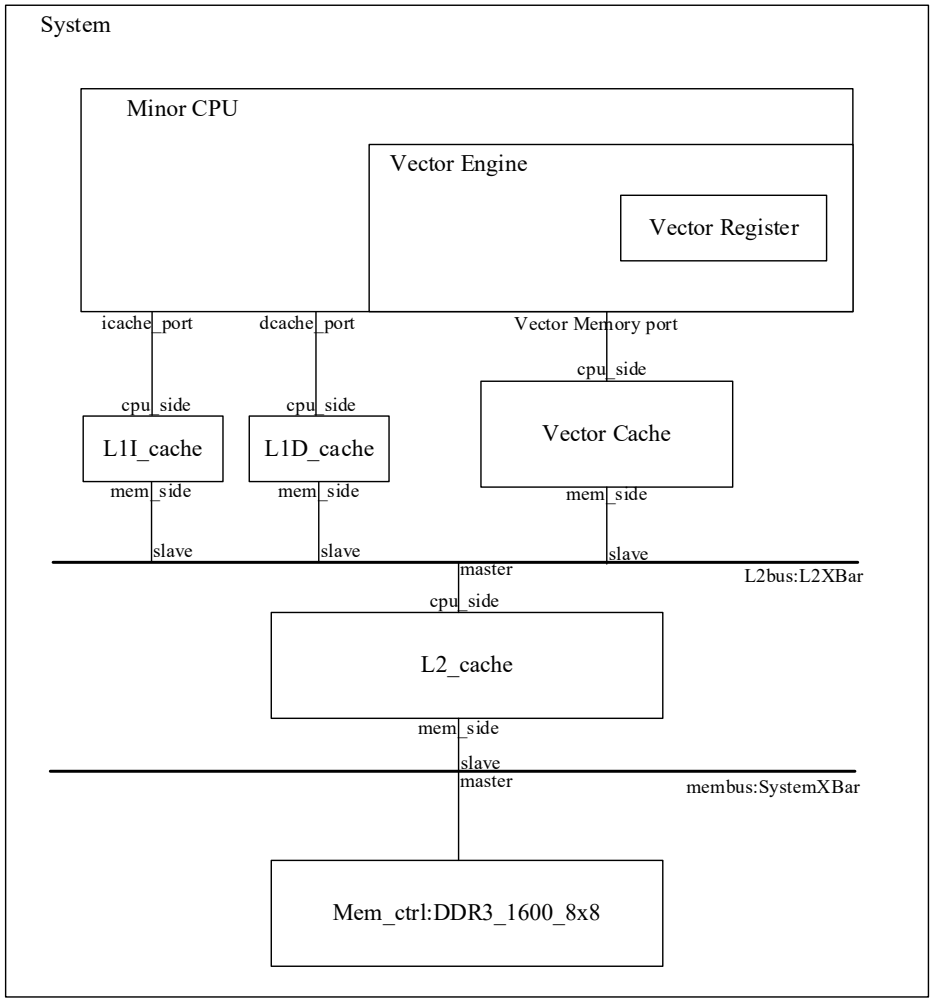


图3.1 模拟器的一种搭建形式示意图

在生成二进制文件后，可以通过更改配置文件进行组件选择、连接方式、缓存大小、缓存级数、内存一致性策略等配置，模拟器核心实现为 CPU 核部分，不同于传统 SIMD 将标量核与向量执行单元紧密耦合设计方案，本模拟器的设计采用标量核与向量核结构解耦合的方式执行。解耦指的是使用多个指令流将指令分类并发送给对应运算单元执行[25]。在本实现中，将指令流拆分到三个不同的指令流当中：标量指令、向量计算指令和向量访存指令，三者并行指令。这能够在一定程度上消除内存延迟带

来的影响，且允许标量计算提前执行。在选择较小的向量寄存器长度的情况下，标量核与向量核耦合在一起实现可以提高设计的整体性，在一定程度上可以将目标设计得更加紧凑，提升计算效率并降低延迟。但耦合设计会限制向量的最大长度、标量核频率等性能指标。将标量核与向量执行单元解耦可以提高设计的灵活度，还可对不同的计算单元设计不同的动态电压，进行更加细化的低功耗设计，在不进行向量计算时可关闭向量核，从而节省能耗。

CPU 核包含 ISA 指令功能描述与 CPU 模型两部分，二者之间可以任意搭配。Gem5 工具提供了几种 CPU 模型，分别为顺序执行且无流水线配置的 Simple 模型，可细分为访存请求发出即可返回的 AtomicSimple 模型与模拟访存延迟的 TimingSimple 模型；顺序执行且包含流水线配置的 Minor 模型；乱序执行且包含流水线的 Out of order 模型。

本模拟器使用的标量核通过更改 gem5 提供的 Minor CPU 模型进行实现，在指令流水线中的不同阶段修改事件实现对向量集的性能模拟，并向 ISA 模块增添 RISC-V 向量指令译码代码、模板与构造函数等定义进行功能模拟。在标量核部分实现对标量指令的译码执行，以及对向量指令的译码标记与分发，向量核部分通过定义新的模块，接收来自标量核的指令数据并分析执行。在向量核的各个模块中会包含微体系结构与向量指令功能实现。

标量核与向量核并行执行，二者通过接口连接并传递指令数据，接口如图 3.2 所示。接口为 VectorEngineInterface，包含以下函数，requestGrant 函数使得标量核能够对向量引擎请求发送新的指令，向量引擎判断向量核指令队列容量、重命名存在空闲表项且物理寄存器空间充足，则许可标量核进行指令分发。在得到向量引擎的许可之后，标量核使用 sendCommand 函数传输动态指令到向量引擎进行分发，动态指令中包含了标量核译码的指令信息。Bussy 函数则可以告知标量核目前向量核处于忙碌的状态。reqAppVectorLength 与指令配置相关。

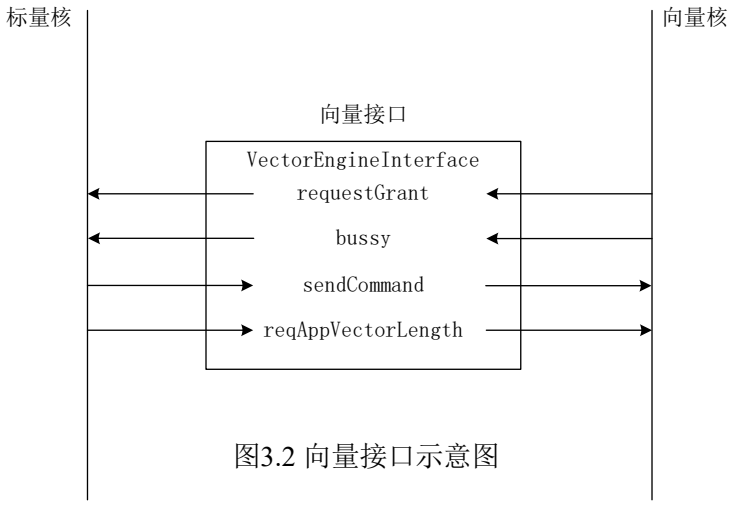


图3.2 向量接口示意图

## 3.2 标量核的设计与实现

Alec Roelke 自 2016 年在 gem5 中对 RISC-V 指令集的进行了初步支持实现[13]，许多团队基于该实现不断完善 RISC-V 与相关扩展集，与此同时 gem5 工具本身也在不断更新迭代中。Gem5 工具本身附带的模型较为细致，通过增添修改工具提供的 Minor CPU 模型可以满足向量设计中标量核的需求，在 RISC-V 对应 ISA 指令功能描述译码部分将向量指令内容填充进去，使得标量核负责初始指令的取码与译码，并按照指令类型将指令分发到对应的流水线当中，指令流水线分为标量指令、向量计算指令和向量访存指令三种。由于向量和标量相互解耦合，不希望向量指令出现冒险中断而导致状态回退，因此如果向量指令有标量源操作数需求，标量核会在准备完成向量核所需源操作数之后，才把相应的指令发送到向量核心进行进一步计算。除了向量配置指令等少部分需要操作向量寄存器的指令之外，大多数向量指令对于标量核来说相当于空操作（NOP）。下面将分别介绍对 Minor CPU 模型方面的改动。

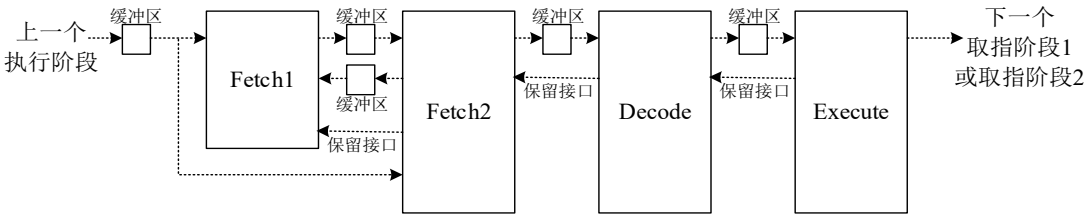


图3.3 Minor CPU 模型流水线示意图

Gem5 工具提供的 Minor CPU 模型的流水线如图 3.3 所示。Minor 模型的流水线主要包含四个阶段，各个阶段间使用先入先出缓存连接，以对级间延迟进行建模[26]。第一个阶段为 Fetch1 取指第一阶段，负责接受来自前段执行或者后续 Fetch2 的更改需求与预测信息以选定指令流地址，然后从指令缓存获取对应指令缓存行，并传递给 Fetch2 进行指令的分析分解。Fetch1 包含请求和传输两个队列以处理转换行提取地址和适应内存提取请求，且可以通过保留接口获取 Fetch2 输入的缓存状态，在 Fetch2 阶段对应的输入缓冲中存在空间时初始化内存取值请求，Fetch2 与 Fetch1 之间的反向缓存用来承载分支预测内容。第二个阶段为 Fetch2 取指第二阶段，Fetch2 包含分支预测机制，会从输入缓存中获取来自 Fetch1 的数据，迭代拆分缓存数据，打包为独立的指令向量以传递给 Decode 阶段，预先处理指令并打包为指令向量有利于提前发现错误并中止指令进程。第三个阶段为 Decode 译码阶段，译码阶段获取 Fetch2 阶段打包的指令向量，并将这些指令分解为微指令并将他们打包到输出指令向量中。第四个阶段为 Execute 执行阶段，Execute 包含指令执行和内存访问机制，并通过功能单元先入先出流水线建模确定指令运行需要的周期。

需要对流水线做出一定增改，以对指令进行处理分类，从而解耦标量核与向量核。对 Minor 模型的解耦更改主要体现在 Execute 阶段，Execute 阶段处理分支与预测信息，执行访存操作，处理中断信息，进行指令分发与提交等操作。

对 Execute 阶段的修改主要体现在指令分发 issue 与指令提交 commit 两部分，Execute 阶段大致执行流程如图 3.4 所示，虚线部分为添加修改部分。首先配置输入输出数据和分支数据信息，检测中断，若发送中断则响应中断分支跳转到下一个取指阶段对中断进行处理，否则检测当前指令状态，将执行完毕的指令提交，然后发送新的指令到相应的功能单元中，推进功能单元流水线执行，检查功能单元执行情况，若功能单元未执行完毕则重新激活执行阶段，最后将未全部使用的数据重新提交到输入缓存中。指令分发用于确认功能单元状态以及进行数据有效地址计算等预处理，指令提交则用于将指令发送到功能单元并等待处理结果提交指令。

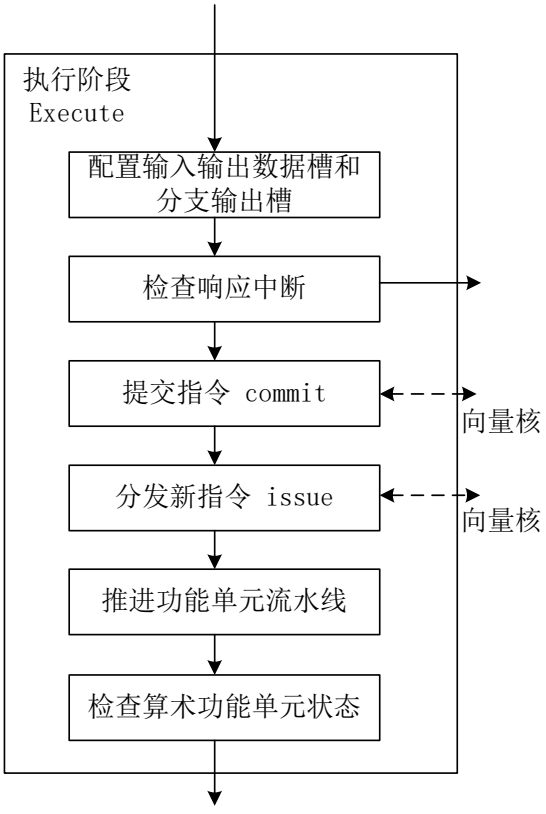


图3.4 指令流水示意图

Execute::inFlightInsts 队列中会按指令分发顺序存储所有计划执行的指令。分发 issue 事件会访问指令功能单元获得单元状态信息，若功能单元空闲可以执行指令，则对指令进行标记，算术计算指令而言加入到 Execute::inFlightInsts 队列尾，访存指令除了加入 Execute::inFlightInsts 队列之外还会额外加入到 Execute::inFUMemInsts 队列，且访存操作会在执行之前计算有效地址。

提交 commit 事件会将指令分发到对应的执行单元中，检查 Execute::inFlightInsts 队列的队首指令的执行状态来决定是否提交指令，若指令已在对应的功能单元执行完毕则可将指令从队列中弹出，根据执行状态选择提交或撤销指令结果。对于访存指令会检查 Execute::inFUMemInsts 队首将指令提交到访存队列 Load Store Queue 进行处理并从 Execute::inFUMemInsts 队列中弹出，在访存指令完成之后进行标记，等待成为 Execute::inFlightInsts 队列队首从而进行弹出处理。分支、中断、错误都会导致提交事件中止并转向下一个取指阶段进行处理。

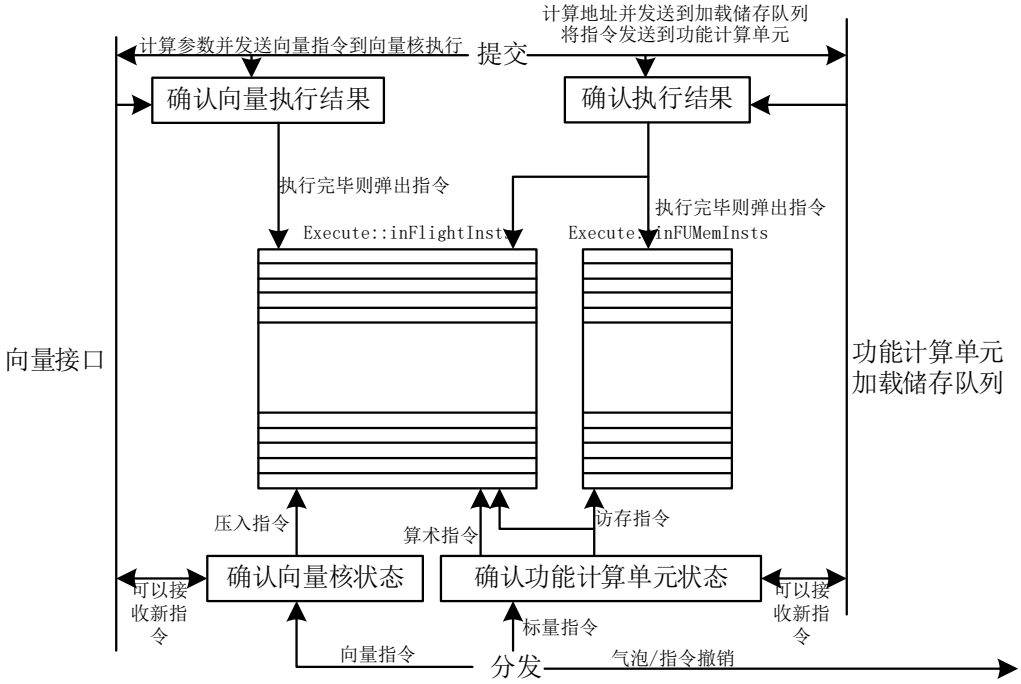


图3.5 执行阶段更改后流程示意图

如图 3.5 所示，增添对向量指令集的支持需要对 Minor 模型的 Execute 阶段中 issue 与 commit 两个事件进行增添，使得标量核可以判断指令是否为向量指令，然后将向量指令打包发送到向量核进行处理。通过修改译码 isa 描述，可以在静态指令类中添加标记 flag，对向量指令进行区分。在 issue 事件通过判断指令是否为向量指令，对于向量指令通过标量核与向量核的接口确认向量核的状态，并在向量核可以接受指令且不存在内存障碍时将指令添加到 Execute::InFlightInsts 队列中。commit 事件则会在标量源数据准备完毕之后，将指令进行初步计算后通过向量接口发送到向量核执行，并检测向量核执行状态，若该指令在向量核中执行完毕，再根据指令状态判断进行提交指令或者撤销指令。对于向量计算指令或者向量访存指令，不断循环等待向量核返回指令结果，当向量指令计算完毕在向量核中完成提交时，会反馈给标量核标记指令完成，此时标量核检查 Execute::InFlightInsts 队首进行指令的提交操作，然后将指令

从 Execute::InFlightInsts 队列中弹出并继续执行下一个流水线阶段，若是向量配置等与状态寄存器相关的指令，则还需更新相关联的状态寄存器。

## 3.3 向量核的设计与实现

本模拟器中，向量核与标量核拆分实现，这可以解除向量核在面积、频率、低功耗设计等方面与标量核之间的绑定，从而减少向量核设计方面的诸多限制。由于向量核与标量核之间是解耦的，标量核无法控制向量核的进程，因此向量核的设计上应当尽可能减小错误与冒险的产生。向量指令分发到处理单元的顺序与数据准备顺序不一定是同步的，为了降低流水线的停顿，向量核可以采用动态调度的方式进行优化流水线，指令顺序进入，乱序执行，顺序提交。动态调度包含寄存器重命名，发射队列和重排序缓存三个结构[27]。寄存器重命名用于分析指令相关性，将逻辑寄存器映射到物理寄存器上，发射队列负责判断向量执行单元和指令源数据状态，将数据准备完毕的指令提交给向量执行单元进行执行，重排序缓存检测向量执行状态，储存目的寄存器重命名前对应的物理寄存器编号，按照指令接收顺序提交计算结果。整体向量核实现见图 3.6。在向量核的各个模块都包含状态检测单元，可以通过向量接口模块进行状态信息整合然后反馈给标量核，下面将分模块对向量核进行介绍说明。

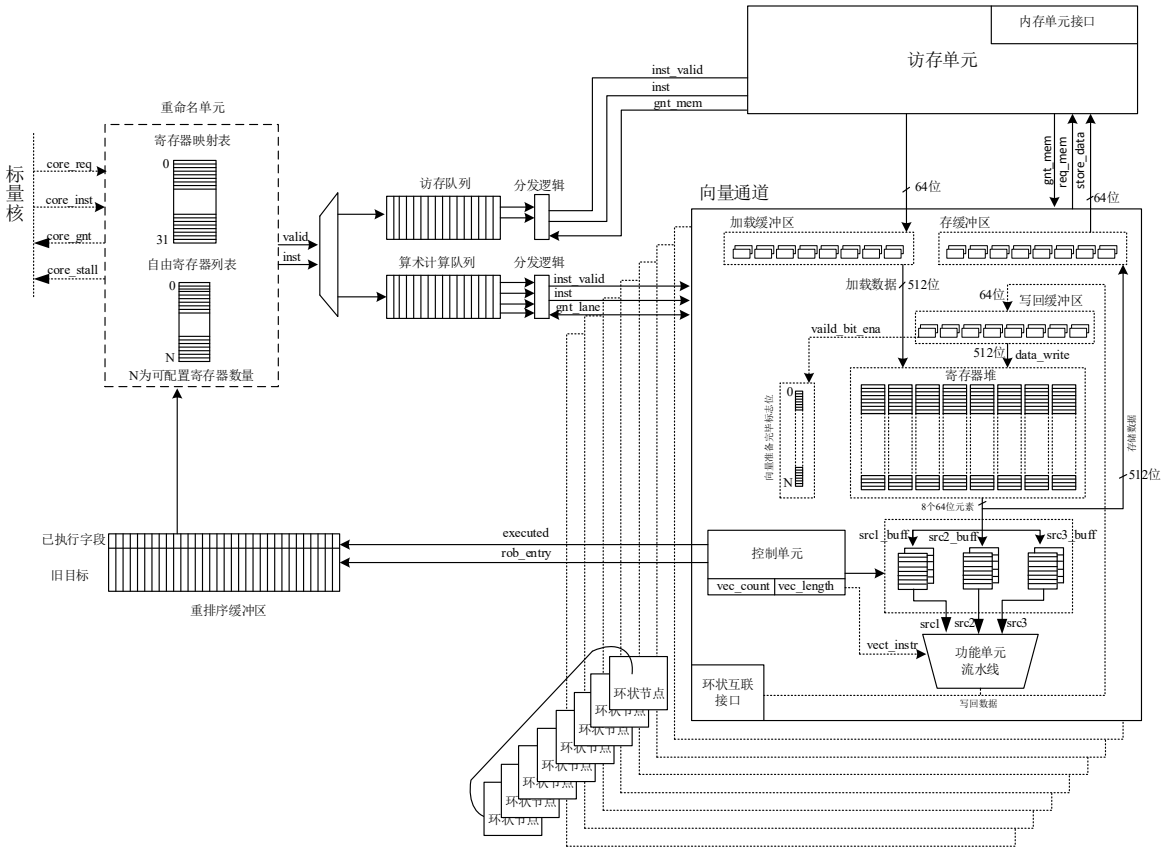


图3.6 向量核结构示意图[17]

### 3.3.1 寄存器重命名与重排序缓存

向量核采用的是指令乱序执行，指令在流水线中重叠执行时，部分指令之间存在相关性，可能会影响计算的结果。在指令执行过程中，前后指令的数据发生冲突的情况称为数据冒险。数据冒险分为先写后读相关（Read After Write）、先写后写相关（Write After Write）、先读后写相关（Write After Read）三种情况[28]，数据相关的示意如图 3.7 所示，其中发生数据相关的皆为 V1 寄存器，指令 A 在流水线执行顺序上先于指令 B。先写后读相关即当靠前的指令 A 写入寄存器，后续指令 B 需要读取同一寄存器，但指令 A 还未将结果写入寄存器，导致指令 B 读取寄存器旧值，引发逻辑错误。先写后写相关即靠前的指令 A 写入寄存器，而指令 B 也写入同一寄存器，但如果指令 B 提前将结果写入寄存器，然后指令 A 再将结果写入寄存器，导致寄存器中存储的是旧值，从而发生逻辑错误。先读后写相关即靠前的指令 A 读取寄存器，后续指令 B 写入同一寄存器，若指令 B 先将结果写入寄存器，从而导致指令 A 读取的是新值，从而发生逻辑错误。数据冒险会导致在下一个时钟周期中后续指令无法执行或者计算错误，最简单的解决办法是插入空气泡或者空操作将发生数据冒险的指令延后，不过这样会降低指令密度从而降低处理器性能。先写后读是无法回避的数据相关，在向量通道中的寄存器堆中，每个向量都带有一个合法标志位来标记该向量是否已经准备完毕，发射队列会检测该标记，如果指令相关性是先写后读，则该指令不会进行发射计算，从而避免先写后读的数据冲突。其他两类则可以通过寄存器重命名消除指令相关性，从而提升流水线效率。

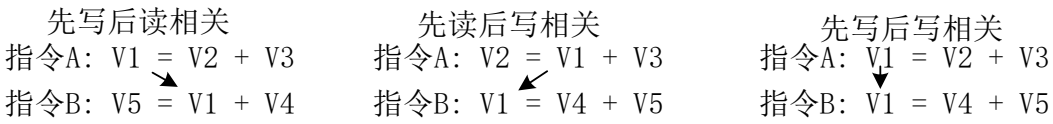


图3.7 数据相关性示意图

寄存器重命名和重排序缓存是在 CPU 设计流程中为了消除流水线中的名称依赖类型的数据冒险的常用技术，二者在向量核中的 dispatch 事件中进行实现，具体结构如图 3.8 所示。

寄存器重命名结构位于向量核的入口，由一个空闲物理寄存器列表和一个寄存器映射表构成。空闲寄存器列表包含所有未被使用的物理寄存器编号，寄存器映射表储存着逻辑寄存器与物理寄存器的映射关系，物理寄存器编号为 0-31，逻辑寄存器编号从 32 开始标记。重排序缓存则包含已执行字段标记，指令目标寄存器对应的重命名之前的逻辑寄存器编号组等信息，使用双指针记录待提交的指令，头指针指向需要检查执行状态然后进行提交操作的指令，后续指令按向量核接收顺序存放在头指针后并

使用尾指针标记最新到达向量核的指令。本实现中，物理寄存器的数据是储存在寄存器堆中的。

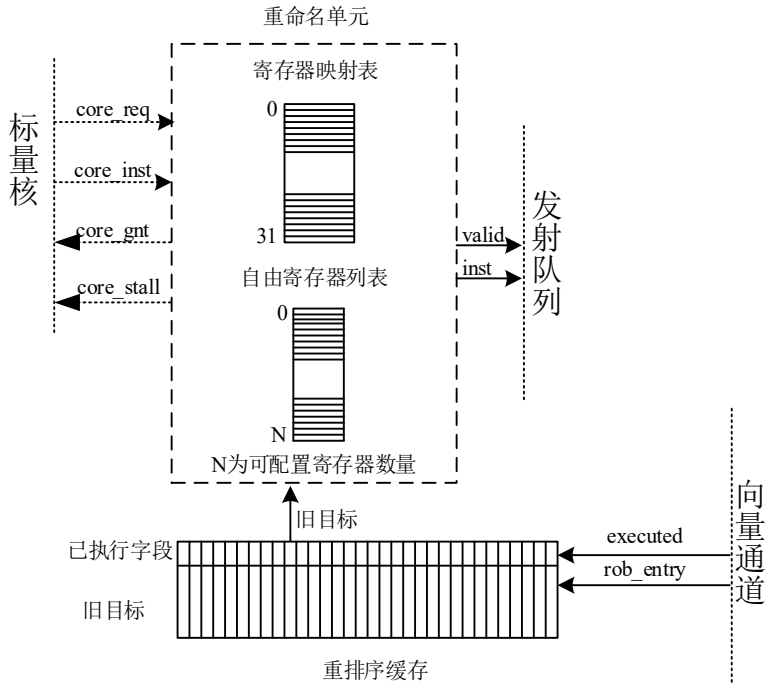


图3.8 向量重命名单元与重排序缓存结构示意图

初始化时，将寄存器映射表中的所有逻辑寄存器映射到对应序号的逻辑寄存器上，代表未对其进行向量重命名操作。当指令从标量核发送到向量核中时，对指令类型、源有效元素宽 EEW、目的有效元素宽 EEW、源寄存器组系数 EMUL、目的寄存器组系数 EMUL 进行计算检查，若指令类型为拓宽或者缩宽类型且需要对源向量寄存器覆盖的情况，还需要使用断言保证当目的 EEW 小于源 EEW 时，覆盖的源向量是源向量寄存器中最低编号的寄存器；当目的 EEW 大于源 EEW 时，覆盖的源向量是目的向量寄存器中最高编号的寄存器。根据计算出目的寄存器数量，从空闲寄存器列表中获得寄存器组编号并将其从空闲寄存器列表中移除，进行寄存器重命名操作并将编号储存到动态指令类中，同时将映射关系写入到寄存器映射表中，再根据指令类型，将寄存器堆中对应位置的合法标志位置 0，代表该操作向量还未准备完毕，任何以该向量为源操作数的指令应该等待其计算或者准备完毕才能被发射到功能单元中。同时，重排序缓存在尾指针对应表项中，将指令对应目标寄存器组的重命名前对应的逻辑寄存器组编号储存在旧目标中，然后将已执行字段置为 0，代表该指令还未执行完毕，并将指令对应重排序表项等信息附加到动态指令类中。当指令执行完毕之后，数据通道会将重排序缓存中对应表项的已执行字段置位，意味着指令已经准备完成可以进行提交操作。头指针会周期性检查该字段，若为可提交状态，则提交指令，且同时检查

后续是否需要使用到该寄存器组内部数据，即检查是否有寄存器标记其为旧目标寄存器，若该寄存器组后续不再使用则释放占用的物理寄存器，将物理寄存器重新添加到空闲寄存器列表当中，然后头指针将指向下一条指令，继续循环检测字段状态。对目的寄存器进行重命名操作可以解决先读后写数据冲突，寄存器重命名流程如图 3.9 所示。重命名中加入了异常状态检测，只有将目标寄存器写入掩码值的指令或储存归约指令的标量结果才可以将目标寄存器组与掩码寄存器 v0 重叠，否则报告异常。

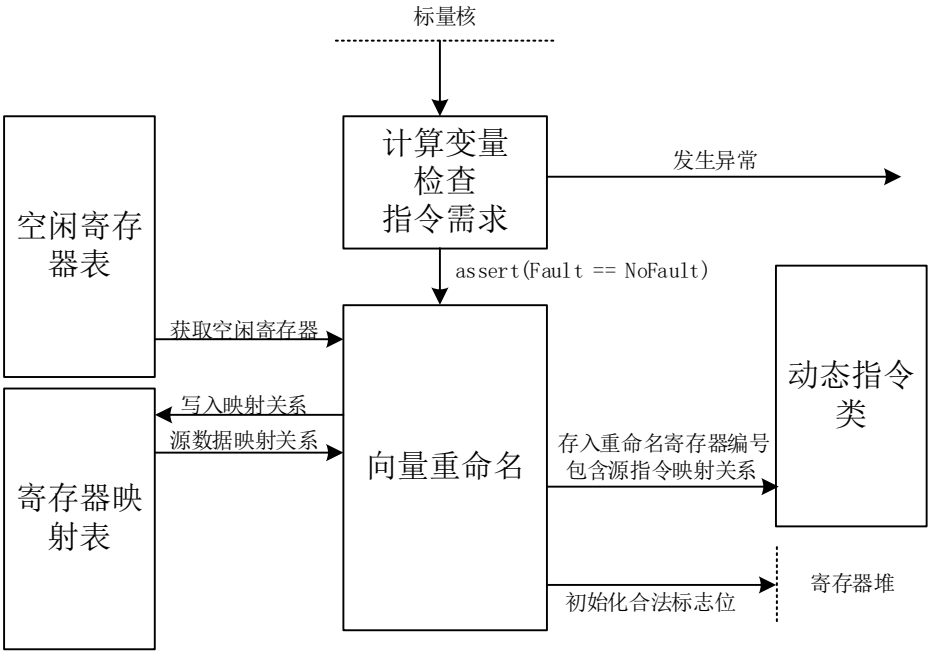


图3.9 寄存器重命名流程示意图

重命名单元具有检查是否更新寄存器映射表的结构，当发生先写后写数据冲突时，重排序缓存中的目的寄存器映射的物理寄存器来源于较旧的指令中的目的寄存器，如图 3.10 所示，向量寄存器 v5 发生了先写后写数据冲突，第一条指令将寄存器 v5 映射为 v2，v5 计算之前的数据储存在 v37 中，第二条指令将 v5 映射为 v36，旧目标的编号为第一条指令的目标寄存器编号 v2，由于目前标量核的实现是顺序执行的，所以向量指令的重命名操作也是顺序的，所以重命名操作可以直接更新映射表，若第一二条指令之间需要使用到 v5 的值，查询映射表得到的也是储存旧值的 v2 寄存器，而最新的数据根据映射表其实储存在 v33 寄存器中，从而避免了先写后写数据冲突。

```
194153000: system.cpu.ve_interface.vector_engine: inst: vfadd_vv v5 v5 v6          PC 0x1112C
194153000: system.cpu.ve_interface.vector_engine: renamed inst: vfadd_vv v2 v37 v39   old_dst v37        PC 0x1112C
194174000: system.cpu.ve_interface.vector_engine: inst: vfadd_vv v5 v5 v3          PC 0x11130
194174000: system.cpu.ve_interface.vector_engine: renamed inst: vfadd_vv v36 v2 v5   old_dst v2        PC 0x11130
194211000: system.cpu.ve_interface.vector_engine: inst: vfadd_vv v5 v5 v4          PC 0x1113C
194211000: system.cpu.ve_interface.vector_engine: renamed inst: vfadd_vv v33 v36 v38   old_dst v36        PC 0x1113C
```

图3.10 先写后写重命名示意图

### 3.3.2 发射队列与向量运算单元

当指令完成重命名操作之后，向量核会根据指令的类型将指令分发到访存发射队列和算术发射队列当中。发射队列在向量核的 dispatch 事件中实现，负责对指令源数据进行准备确认，对向量执行单元进行状态确认并将准备完毕的指令分发到对应执行单元中，具体结构如图 3.11 所示。

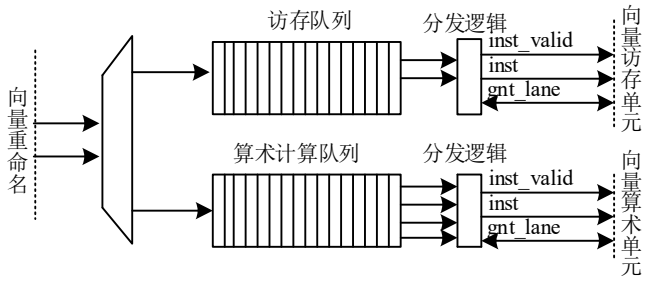


图3.11 向量发射队列结构示意图

向量运算的操作数据需要通过访存指令从内存中读取，由于命中概率不同，数据准备需要的时间也有长有短。向量运算单元一次只能执行一条向量指令，为了提升向量核效率，避免较前的指令数据依赖未准备完毕而引起流水线阻塞，向量核允许指令乱序执行。发射队列会不断循环检测指令数据，检查寄存器堆中源寄存器对应的合法标志位，当所有需求的数据都准备完毕，且指令对应的向量执行单元空闲，可以接收新指令时，将准备好的指令发送到对应单元进行处理，并将指令从队列中去除。向量算术计算指令和向量加载指令可以乱序发射、乱序执行，但向量存储操作需要按顺序进行发射执行。访存指令与算术指令处理的结构不同，所以发射队列也相应地分为两个队列：访存队列与算术队列，对应的向量执行单元为向量访存单元和向量算术计算单元。向量访存单元是用来获取算术队列指令计算所需用到的数据，并将向量计算结果保存到主内存当中的模块；算术计算单元则是用来分析向量算术指令并计算指令结果的模块。

指令经由访存队列发射之后到达向量访存单元，结构如图 3.12 所示，向量访存单元通过总线与缓存进行连接，可以配置为一级缓存、二级缓存、内存等，不同的连接对象具有不同的命中率和数据读取速度。向量访存单元会根据标量核译码的指令信息，分析访存指令类型，根据存储类型对向量通道中的加载缓存或者储存缓存进行操作。由向量扩展集存储向量的指令类型可得，向量访存单元需支持单位跨步寻址、跨步寻址、索引（聚集/分散）寻址模式。一旦储存单元接收到了访存的指令、基址、向量长度、跨步等信息，存储单元会计算出所有的请求地址并将其放入到一个先入先出队列中。然后流水线式地发送访存请求，然而由于数据命中问题，存储单元可以按照

不同顺序应答以避免阻塞。当向量成功完成指令后反馈指令完成，并接收来自访存队列的下一条指令。加载和存储会将数据暂时存放在加载缓存区和存储缓存区中，当加载缓存区中的数据与寄存器堆行大小一致时进行写入寄存器堆，当存储缓存区中的数据与系统缓存行大小一致时写回到内存中。加载操作会将加载数据对应的寄存器堆中数据的首地址放入到动态指令类中，以便向量通道计算时将元素加载到对应的数据缓存中，通过物理寄存器编号可以计算确定该寄存器在寄存器堆中的地址，再根据元素编号与有效元素宽度可以确认该元素在寄存器堆中存放的地址。

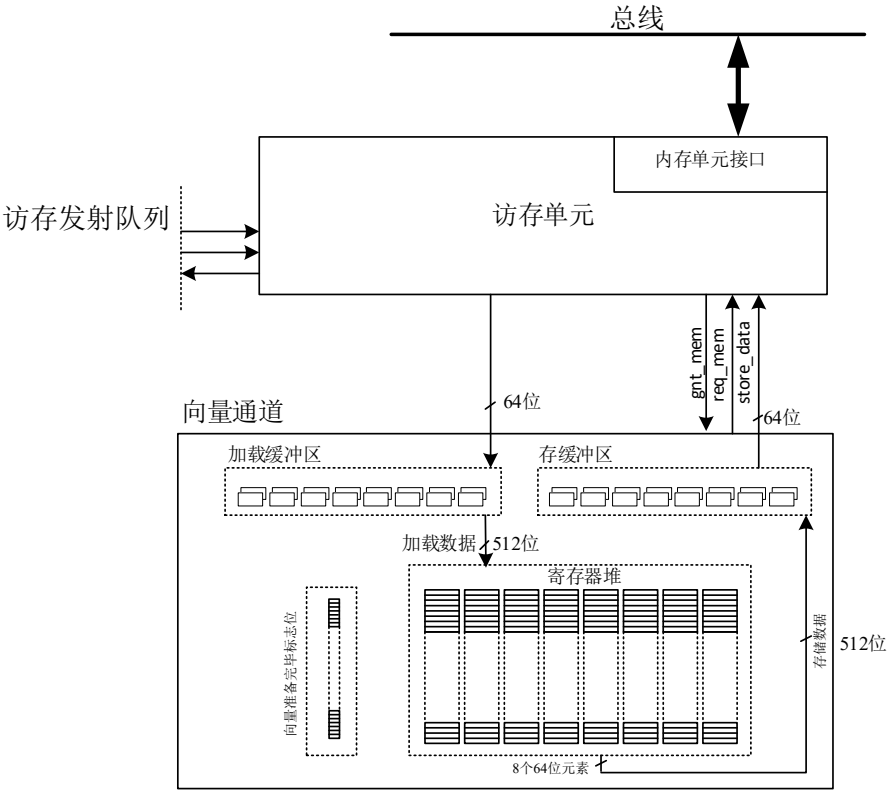


图3.12 访存单元与寄存器队列示意图

由于向量计算需要的访存需求量大，设计上端口可以跟多级别的内存层级相连，本模拟器配置了多种连接方式，测试时将向量访存单元跟 L2 缓存连接。如果跟较小的 L1 缓存相连，则存取指令会发生较多的缓存缺失的情况，需要频繁向下调取数据。而对于向量指令而言，由于处理元素足够多，所以可以接受较大的内存访问延迟。

一个完整的向量引擎包含多个向量通道，向量通道也是向量并行化计算的体现。一个向量通道包含加载缓存、储存缓存、写回缓存、向量寄存器堆、数据缓存、向量计算功能单元等部分。向量通道与存储单元之间使用端口互联以便访存指令从内存单元存储数据。在多通道的情况下，每个通道操作处理向量的一部分元素，向量寄存器的元素以交错的方式排列在各通道上，计算时各通道之间异步进行，共同处理一条向

量指令指定的操作。如图 3.13 所示，图中缓存行大小为 64 个 8 位元素即为 512 位，配置最大向量长度 MVL 为 256 位，向量通道数为 8，可以在图中看到所有元素在通道中交错排布的样子。每个读操作会返回一个缓存堆行大小的数据，分配到所有的向量通道中，图示缓存行大小为 512 位，则每个通道每次读取可以获得 64 位的数据，缓存中每个数字代表一个 8 位元素，是向量核配置的最小元素宽，当需要使用更大的元素进行计算时，则使用多个 8 位组合放置该元素。加载和存储操作都是流水线化且独立的，对于加载操作而言，内存可以不按顺序对访存请求进行响应，并将数据暂存在加载缓存中，当缓存收集到寄存器堆行大小的数据后，会将寄存器堆的合法标志位置 1，意味着该数据已准备完毕。

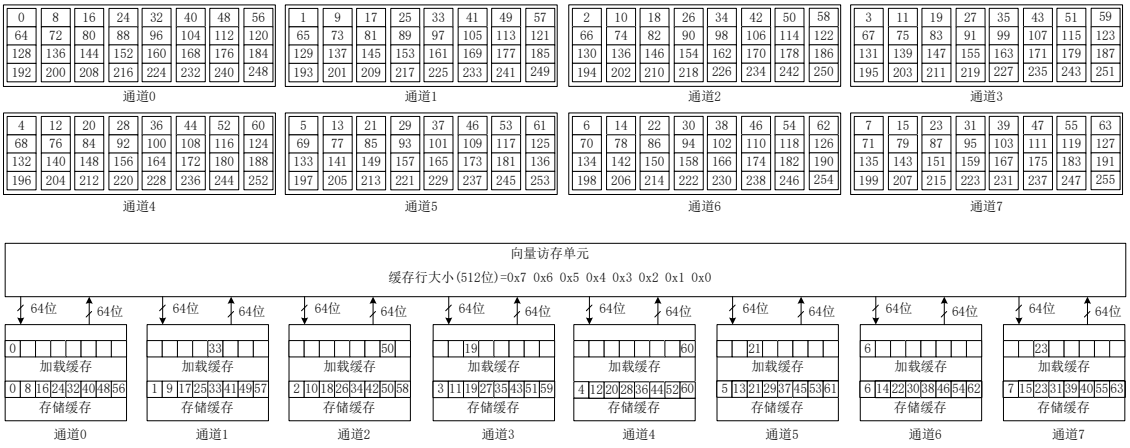


图3.13 向量数据交错排列示意图[17]

为了降低数据存取对计算效率的影响，应当保证存储单元与计算核心之间数据交互的效率，可以通过增加端口与互联线的方式提高数据吞吐速度，但由于互联线的提高会影响系统复杂度，限制最大操作频率，提高版图面积且大幅提高成本，在计算元素较大的向量的情况下，几个周期的数据存储属于可以接受的范围，本模拟器每个向量通道使用的读取宽度均为 64 位。

指令相关的数据都准备完毕且向量算术计算单元空闲时，算术计算指令经由算术计算队列发射之后到达向量算术计算单元，结构如图 3.14 所示，计算流程如图 3.15 所示。向量通道中寄存器堆与数据缓存相连，此时向量通道会根据动态指令类中的信息，按对应元素位宽 SEW、向量元素数 vl 从寄存器堆中获取指定数量的元素，然后将数据按序暂存在数据缓存中，并根据 EMUL、EEW 等数据调用相应的向量功能单元进行数据计算，同时会标记对寄存器的分组。模拟器会按找向量元素数 vl 的值分配的向量通道，若有效元素宽度为 16 位，那在向量最大宽度配置为 64 位的情况下，8 通道一个周期可以完成两个向量寄存器构成的寄存器组的计算。计算完毕的数据会暂时存储在写回缓存中。当写回缓存中得到与寄存器堆行大小相同的数据时，执行写

回寄存器堆的操作。向量算术计算指令执行结束时，会通过字段反馈标记目的寄存器相关的数据源准备完毕，此时等待该指令结果作为源数据的指令允许进行发射。而寄存器堆中的数据可以通过向量写命令通过存储队列将计算结果写回内存单元中。对于拓宽或缩宽这类改变有效元素宽度的数据，会在写回缓存通过识别动态指令类中重命名阶段载入的信息，然后写回到寄存器堆中对应的寄存器组中。

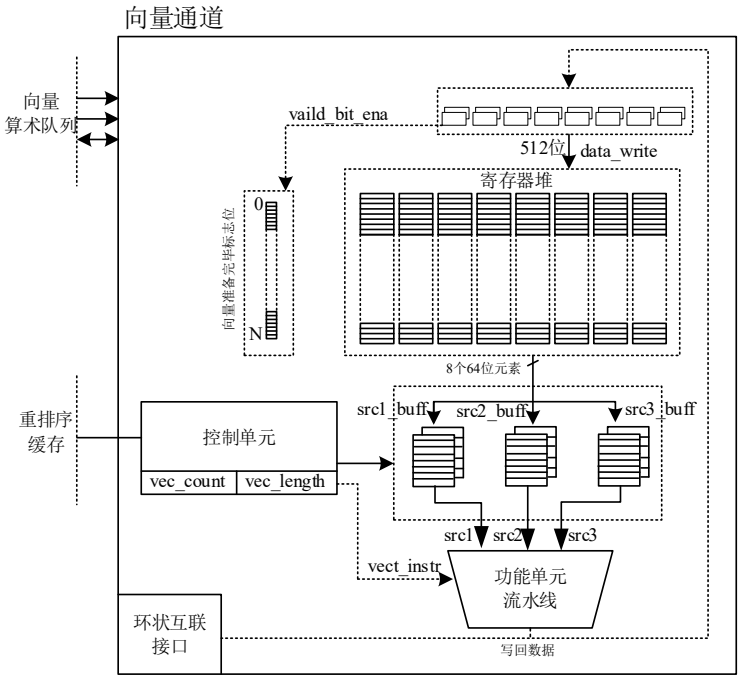


图3.14 算术计算单元示意图

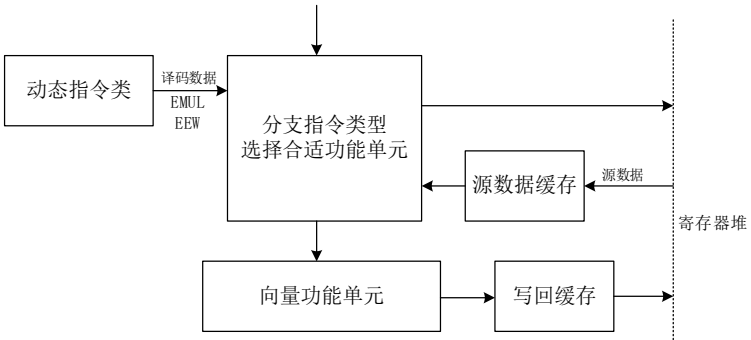


图3.15 算术计算单元流程示意图

不可知策略的实现也添加在向量通道中，在生成动态指令类时，会将 vta 和 vma 的值加入到动态指令类中，当算术计算单元查询指令类别时，若对尾部元素或者非活跃元素进行了不可知策略的配置，则在获取源数据时不会查询旧目标，也不调用对应功能函数，直接将对应进行掩码操作的元素置为 1。

### 3.3.3 通道间互联

不同的向量通道之间指令是异步运行的，在加载时将数据交错载入到各通道中，存储时也按照通道交错地将数据填入储存单元中。在向量指令集中具有需要使用到向量通道之间交互的指令，如在向量寄存器中将一段元素滑动偏移的 slide 操作，将向量元素进行归约运算的 reduction 操作等。这些操作设计的指令要求不同向量通道之间进行数据通信，所以需要建立互联网络以便实现数据的交换。互联网络的拓扑结构有很多种，如线性网、超立方体网、环型网、循环移数网、全连接网、二叉树网等[29]。由于各通道之间关系平等，目前实现了环型网络结构和全连接型网络结构两种，如图 3.16 所示。

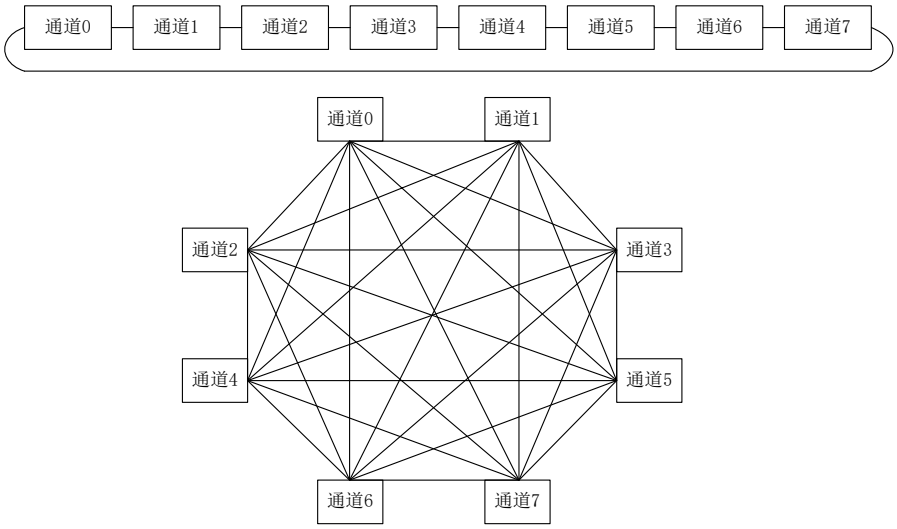


图3.16 环型互联网络和全连接互联网络示意图

交错型网络中不同的通道之间都存在双向连接，因此仅需一个时钟周期就可以实现任意两个通道之间的元素交换，这对密集使用通道间通讯的应用友好，但是通道数增多会导致互联设计的复杂度、实现成本和面积需求的增加。环状网络中一个通道只与相邻的两个通道双向连接，这样进行一次元素移动需要经由多个通道传递数据，耗费多个时钟周期，但是更加节省面积、成本低也易于设计实现。本文测试配置采取的是环状互联作为通道通讯结构。

## 3.4 本章小结

本章对 gem5 中的模拟器结构进行了介绍与说明，由于 gem5 是模块化的，所以对于各个部件都可以根据实际设计的目标要求进行配置更改，对于处理器模拟器而言最重要的部分为处理器核部分。处理器核包含 CPU 模型与 ISA 指令功能描述两个方

面，前者实现性能模拟，后者实现功能模拟。本章简要介绍了 gem5 中提供的 CPU 模型，并选取顺序流水 Minor CPU 模型，改动其流水线中 Execute 阶段中的事件，使得标量核在 Execute 阶段对向量核进行指令分发与反馈接收。本章的后半部分对向量核的结构与微体系实现进行了说明阐述，配置了重命名单元来解决向量指令的数据冲突问题，使用发射队列检测指令数据和执行单元的准备状态，并介绍了向量访存指令和向量算术指令的功能实现单元，最后介绍了向量核中不同通道之间的交互方式。

# 第四章　模拟器功能实现

CRISTÓBAL RAMÍREZ 团队所设计的模拟器是基于向量扩展集 0.7 版本进行设计配置，实现了向量配置操作、算术运算操作、逻辑操作、移位操作、比较操作、归约操作、滑动操作等基础指令，且仅支持 LMUL 小于 1 的情况。本章将分析原模拟器实现的指令功能，并对比 RISC-V 向量集稳定版本 1.0 版需要实现的指令类型，添加更多的指令支持，并完成对模拟器进行配置设置。

## 4.1 向量指令译码实现

模拟器在 Minor CPU 模型中定义了指令的流水线，然后在各阶段中使用事件对阶段内发生的数据交换和计算过程进行建模与时序预估，从而实现对 CPU 性能的模拟。在 CPU 功能实现方面，则在流水线的 Decode 阶段定义，半自动生成指令集功能代码。Gem5 通过使用领域特定语言描述指令集中各个指令功能和译码函数，文件后缀为 isa，在编译过程中通过分析不同字段，对指令进行分类拆分，调用 python 脚本对 isa 的词法和语法进行分析，使用对应格式和模板将指令行为描述转换，生成包含指令定义和译码函数的 c++文件，从而对指令实现译码与功能实现，具体流程如图 4.1 所示。通过对指令译码模块代码的增添，可以对模拟器加上向量指令的译码支持。

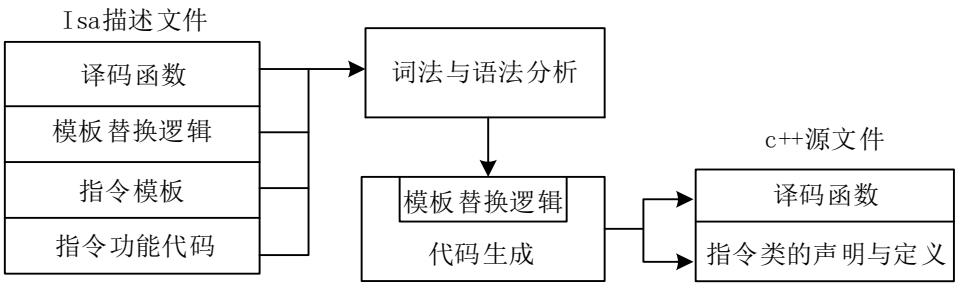


图4.1 Gem5 半自动生成指令功能代码示意图[30]

图 4.2 展示了部分 decoder.isa 文件中的内容，其中字段的定义是放在 bitfield.isa 中的。译码过程是通过逐步查询指令中相应字段的值，从而对指令进行分类。

由于在设计上标量核与向量核之间为解耦设计，所以向量指令的功能单元 Functional Unit 将定义在向量核中，标量指令在 isa 描述文件中定义指令译码方式和功能单元的指令实现，例如图 4.2 中的浮点加载字指令 flw，就在指令 flw 后定义了该指令的功能实现，该指令为浮点读取内存指令。对于向量指令而言，向量指令需要在标量核中定义译码方式将指令信息拆分储存在静态指令类中，所以需要在

decoder.isa 中定义向量指令进行占位，但不具体定义向量指令的功能实现，使得 Decode 阶段可以识别向量指令的指令类型与名称，通过定义对应的模板、格式，添加向量核区分指令类型和计算所需的标记信息，方便标量核在 Execute 阶段的 issue 事件和 commit 事件中识别指令，对指令进行译码并发送到向量核进行执行。向量核再根据标记信息区分应该将指令发往向量访存单元还是向量算术计算单元，选取对应长度的元素与功能实现单元进行后续计算处理。

```
// The RISC-V ISA decoder
//

decode QUADRANT default Unknown::unknown() {
    0x3: decode OPCODE {
        0x01: decode FUNCT3 {
            format Load {
                0x2: flw({{
                    STATUS status = xc->readMiscReg(MISCREG_STATUS);
                    if (status.fs == FPUStatus::OFF)
                        fault = make_shared<IllegalInstFault>("FPU is off",
                                                              machInst);

                    Fd_bits = (uint64_t)Mem_uw;
                }}, inst_flags=FloatMemReadOp);
                0x3: fld({{
                    STATUS status = xc->readMiscReg(MISCREG_STATUS);
                    if (status.fs == FPUStatus::OFF)
                        fault = make_shared<IllegalInstFault>("FPU is off",
                                                              machInst);

                    Fd_bits = Mem;
                }}, inst_flags=FloatMemReadOp);
            }
            0x0: decode MOP {
                0x0:VectorMemLoadOp::vlb_v();
                0x2:VectorMemLoadOp::vlsb_v();
                0x3:VectorMemLoadOp::vlxb_v();
            }
```

图4.2 Gem5 中对指令的译码示意图

对于向量扩展集而言，常用的指令字段如图 4.3 所示。

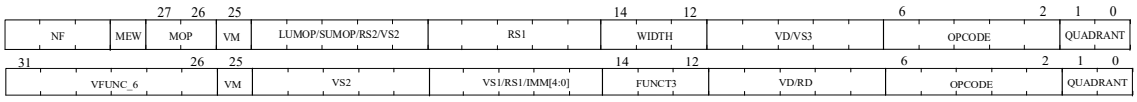


图4.3 向量存储和算术指令字段示意图

RISC-V 指令长度可通过低位的值进行快速区分，其中 16 位压缩指令 QUADRANT 字段不等于 0x3，高于 16 位的指令 QUADRANT 字段为 0x3。OPCODE 字段用于区分 32 位指令的功能，向量加载指令选择 0x1，向量存储指令选择 0x9，由 WIDTH 字段区分指令的操作数据位宽，再通过 MOP 字段区分向量加载或存储的方式。同样 QUADRANT 字段选择 0x3，OPCODE 字段选择 0x15，根据 FUNCT3 字段

区分子向量类型，依此为 OPIVV、OPFVV、OPMVV、OPIVI、OPIVX、OPFVF、OPMVX 与向量配置指令类。然后根据 FUNCT6 字段区分具体的指令类型，具体选择过程如图 4.4 和图 4.5 所示。

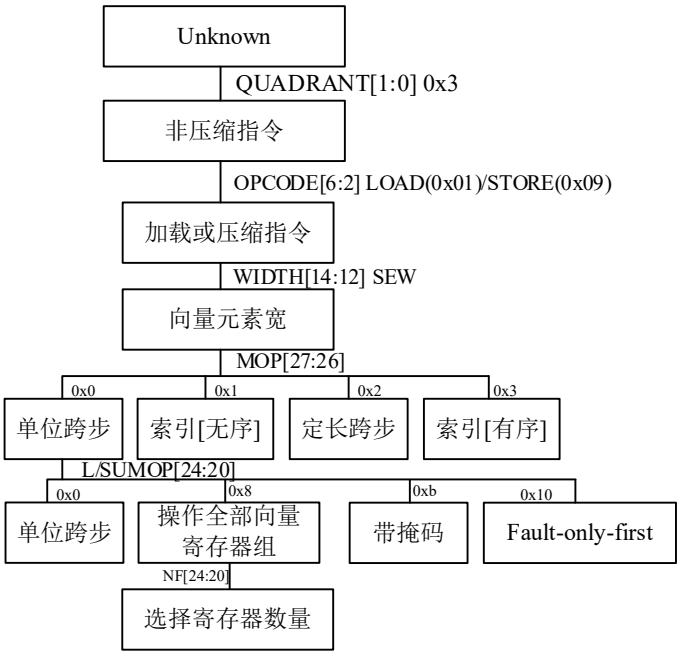


图4.4 向量访存指令的译码流程示意图

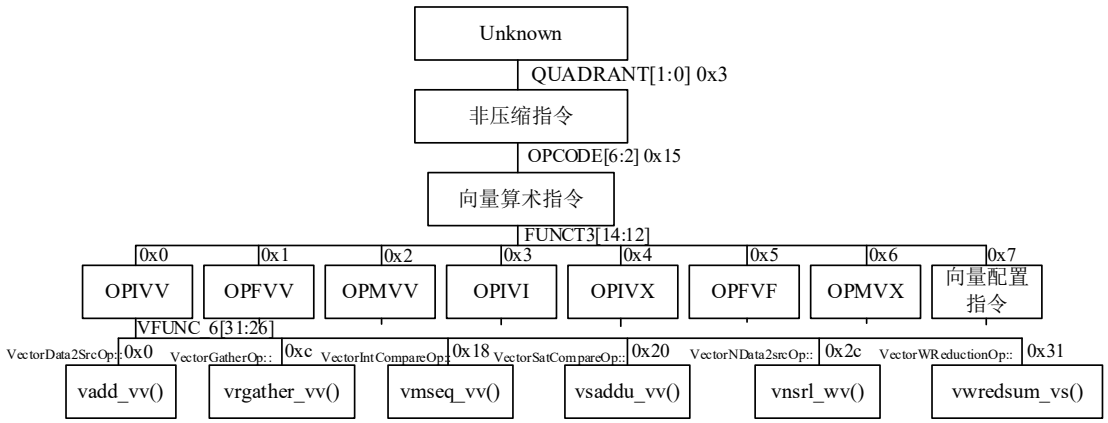


图4.5 向量算术部分指令的译码流程示意图

在 decoder.isa 中，可以对指令定义其类型，并对类型定义模板和格式，在译码时，gem5 会对同一类型的指令套用相同的模板和构造函数生成译码函数和指令类的声明和定义。由图 4.5 可以看出，在译码过程中，会给所有向量指令分配一个指令类型，以便脚本能够根据指令类型去找到对应的格式和构造函数，向量指令集根据指令功能可以分为几种类型的操作指令，如向量配置操作 VectorConfigOp、向量类型转换操作 VectorConvertFPToIntOp、向量数据计算操作 VectorData2SrcOp、向量比较操作

VectorIntCompareOp 等。

上述的操作类型还有很多，通过定义操作类型将指令按照计算方式归类划分，并在 isa 描述文件中定义模板和构造函数，从而添加对向量指令的译码实现。使用函数 opClass()可以获取指令的操作类型，在译码时可以将所属的操作类型、指令字段、源操作数编号、目的操作数编号、模板中定义的标志位等数据提取之后储存在静态指令类中。

## 4.2 向量功能单元实现

在 gem5 中的 Minor CPU 或者 O3 CPU 模型流水线的译码阶段，会根据模板产生类声明与定义，这些类可以通过功能单元进行执行从而实现指令功能[31]。在 Minor CPU 模型的配置文件中，定义了流水线中可以使用的所有功能单元，还定义了这些类的发射耗时 issueLat 和执行耗时 opLat，使用较为准确的延迟信息有助于模拟器对指令运行时长进行精确估算。在 Minor CPU 模型的执行阶段，会检查操作类型和选择相应的功能单元，所以对于 4.1 小节所添加的指令操作，需要在 Minor CPU 模型的配置文件中进行添加，如图 4.6 所示。

```
169 class MinorDefaultMemFU(MinorFU):
170     opClasses = minorMakeOpClassSet(['MemRead', 'MemWrite', 'FloatMemRead',
171                                      'FloatMemWrite'])
172     timings = [MinorFUTiming(description='Mem',
173         srcRegsRelativeLats=[1], extraAssumedLat=2)]
174     opLat = 1
175
176 class MinorDefaultVectorFU(MinorFU):
177     opClasses = minorMakeOpClassSet([
178     "VectorArith1Src", "VectorArith2Src", "VectorArith3Src",
179     "VectorMaskLogical", "VectorReduction", "VectorConvertIntToFP",
180     "VectorConvertFPToInt", "VectorWConvertFPToInt", "VectorWConvertIntToFP",
181     "VectorWConvertFPToFP", "VectorNConvertFPToInt", "VectorNConvertIntToFP",
182     "VectorNConvertFPToFP", "VectorFPCompare", "VectorIntCompare",
183     "VectorSlideUp", "VectorSlideDown", "VectorMemoryLoad",
184     "VectorMemoryStore", "VectorConfig", "VectorToScalar",
185     "VectorSatCompute", "VectorGatherOp", "VectorNData2SrcOp",
186     "VectorWReductionOp"])
187
188 class MinorDefaultFUPool(MinorFUPool):
189     funcUnits = [MinorDefaultIntFU(), MinorDefaultIntFU(),
190         MinorDefaultIntMulFU(), MinorDefaultIntDivFU(),
191         MinorDefaultFloatSimdFU(), MinorDefaultPredFU(),
192         MinorDefaultMemFU(), MinorDefaultMiscFU(),
193         MinorDefaultVectorFU()]
```

图4.6 添加向量操作示意图

对于指令的操作延迟信息，较为规范的做法是在向量操作类中进行定义，如图 4.6 所示中的 MinorDefaultMemFU 一样将指令按分类定义延迟更便于指令延迟配置和管理。本模拟器暂时没有完成这部分的统一规范，在向量单元的 inst_latency.hh 文件中

根据指令名称对延迟进行定义，在向量通道中进行耗时计算，如图 4.7 所示。

```
void
Datapath::get_instruction_latency()
{
    std::string operation = insn->getName();

    /*************************************************************************
     * Floating point Operations
     ************************************************************************/
    if ((operation == "vfadd_vv") | (operation == "vfadd_vf")) {
        Oplatency           = 4;
    }

    if ((operation == "vfsub_vv") | (operation == "vfsub_vf")) {
        Oplatency           = 4;
    }

    if ((operation == "vfmul_vv") | (operation == "vfmul_vf")) {
        Oplatency           = 3;
    }

    if ((operation == "vfdiv_vv") ||(operation == "vfdiv_vf")) {
        Oplatency           = 14;
    }
```

图4.7 指令延迟定义

通过向量功能单元定义向量指令的功能实现，由第三章可知功能单元定义在数据通道中，每个数据通道进行数据计算时会根据指令类型调用相关的功能单元，功能单元区分元素宽度，数据通道会根据动态指令类中的信息识别指令，并调用相应位宽的功能单元进行处理。对于新增添的指令，也需要定义不同指令位宽的功能单元，如对于 OPIVV 向量整型操作，就需要定义 64 位的 long int、32 位的 int、16 位的 int16、8 位的 int8 四种类型的功能单元，具体实现定义如图 4.8 所示。

```
#include "debug/Datapath.hh"

int
Datapath::compute_float_fp_comp_op(float Aitem, float Bitem,
    RiscvISA::VectorStaticInst* insn)
{
    int Ditem=0;
    std::string operation = insn->getName();
    numFP32_operations = numFP32_operations.value() + 1; // number of 32-bit FP operations

    if ((operation == "vmfeq_vv") || (operation == "vmfeq_vf")) {
        Ditem = (Bitem == Aitem) ? 1 : 0;
        DPRINTF(Datapath,"WB Instruction = %f == %f  = %d  \n",
            Bitem,Aitem, Ditem);
    }

    if ((operation == "vmfne_vv") || (operation == "vmfne_vf")) {
        Ditem = (Bitem != Aitem) ? 1 : 0;
        DPRINTF(Datapath,"WB Instruction = %f != %f  = %d  \n",
            Bitem,Aitem, Ditem);
    }

    if ((operation == "vmflt_vv") || (operation == "vmflt_vf")) {
        Ditem = (Bitem < Aitem) ? 1 : 0;
        DPRINTF(Datapath,"WB Instruction = %f < %f  = %d  \n",
            Bitem,Aitem, Ditem);
```

图4.8 功能单元定义

数据通道在进行指令处理的时候首先分析向量选择元素宽度 SEW，将数据从寄

存器堆中按照 SEW 宽度按序放入数据缓存队列中，然后根据指令操作类型计算操作延迟。在选定功能单元之前，从各数据缓存队列的队首获取元素，传递给对应的功能单元进行结果计算。代码定义上使用 Aitem 保存 VS1 字段对应的操作数，可以是向量寄存器中的元素、立即数或标量寄存器元素；Bitem 保存 VS2 字段对应的操作数，为向量寄存器中的元素；Ditem 保存 VD 字段对应的目的操作数，用于暂存计算结果；Dstitem 保存目的寄存器重命名映射的向量寄存器中对应位置的元素，当使用不受干扰策略时，不进行向量计算的元素需要保持对应位置的元素值不变，对于这类型的指令需要传入该值并判断掩码寄存器；Mitem 保存掩码寄存器 v0 中储存的对应元素的掩码值，当指令需要进行掩码判断或者如进位加法指令一样需要用到 v0 的值的时候将 v0 向量寄存器对应元素位进行加载。

## 4.3 对模拟器的修改配置

### 4.3.1 状态寄存器增添

对于状态寄存器而言，vl、vtype、vlenb 这三个寄存器是用户可读不可写的，vlenb 为固定值，是向量寄存器长度 Vector Register Length 的字节数，对于特定的配置实现来说是固定的，用于简化某些需要向量寄存器以字节计数的场景计算，vl、vtype 两个状态寄存器是通过向量配置指令进行配置，在执行向量配置指令时会提取配置指令的值并更新状态寄存器。

```
435     CSR_TDATA3 = 0x7A3,
436     CSR_DCSR = 0x7B0,
437     CSR_DPC = 0x7B1,
438     CSR_DSCRATCH = 0x7B2,
439
440     CSR_VSTART      = 0x008,
441     CSR_VXSAT       = 0x009,
442     CSR_VXRM        = 0x00A,
443     CSR_VCSR        = 0x00F,
444     CSR_VL          = 0xC20,
445     CSR_VTYPE       = 0xC21,
446     CSR_VLENB       = 0xC22
447 };
```

图4.9 增添向量计算状态寄存器索引

本模拟器添加了 vstart、vxsat、vxrm、vcsr、vlenb 等用户可读可写状态寄存器，增添的寄存器索引如图 4.9 所示，需要在模拟器添加对状态寄存器的读写方式，并对向量核添加访问向量状态寄存器的方式。新添加的状态寄存器与部分向量计算指令相关，vstart 会在异常发生的时候进行更新，将其置为异常发生时执行的元素索引，便

于异常处理之后进行恢复，vxsat 与饱和计算相关，当向量饱和计算指令发生饱和截取时需要更新饱和状态寄存器的值，vxrm 为向量舍入方式寄存器，储存着向量发生舍入时的舍入模式，vcsr 包含 xrm 与 vxsar 两个状态寄存器的值。

其中 vxrm 可以通过状态寄存器写立即数 csrwi 指令写入，而由于 csrwi 为状态寄存器写立即数操作，指令分属标量核处理的范围，需要对标量核与向量核之间增添交互手段。当对 vxrm 状态寄存器修改时通知向量核配置对应的值，由于向量指令乱序执行，向量执行时再检查 vxrm 状态寄存器可能获得到错误的值导致运算错误，需要检查写立即数指令与向量指令的先后关系，较为麻烦繁琐。目前添加的实现为将 vxrm 的值在译码时添加到静态指令类中，使得指令执行顺序不影响舍入方式。而对于向量饱和计算这类需要向量核对状态寄存器进行更新的指令而言，构造类时需要添加指针 ExecContextPtr& xc，并对功能单元进行更改，从而在发生饱和截位时更新 vxsat 的值。

### 4.3.2 向量指令增添

本模拟器在译码模块与功能单元上增添了许多指令，以下将分类描述。

带进借位加减法，这类指令由于编码限制，进位输入和借位输出必须来自隐式 v0 寄存器，这类指令编码时为带掩码的指令，即 vm 字段为 0，但是计算时会对所有元素进行操作，需要在数据通道对这类指令传入 Mitem 作为操作数，并调用对应的功能计算单元。

向量加减平均指令，这类指令对向量进行加减操作，然后右移一位并按照状态寄存器 vxrm 中的值进行舍入，这类指令则需要将指令发送时对应的状态寄存器 vxrm 的值添加到指令类中。

向量饱和计算，这类指令会在计算溢出时取边界最大值或最小值，这部分计算的功能单元会在计算出现饱和时配置对应 vxsat 状态寄存器。

寄存器聚集指令 vrgather.vv，这类指令会将 vs1 向量寄存器作为索引，提取出 vs2 向量寄存器中的值，这种类型的指令需要采用向量通道间的互联网络对数据进行处理，对于需要使用到通道互联或者计算结果需要写入到标量核的指令而言，功能单元是定义在向量通道中的。由于使用的是环状互联，因此聚集指令执行时间较长。

向量移动指令，这类指令会复制向量寄存器或者向量寄存器组到目的寄存器。对于需要移动标量寄存器与浮点寄存器时，需要使用 ExecContextPtr& xc 进行寄存器配置。

向量类型转换指令，使用强制类型转换完成功能单元的定义，并按照指令类型进行舍入操作。

向量拓宽与缩宽指令，原模拟器没有完成对寄存器分组的设定配置，对于计算指令强制限定 LMUL 的值不能超过一个向量长度，计算时按照最大向量长度 Vector

Maximum Length 进行计算，每个周期计算通道数的值并循环查询计算通道是否空闲，以最大化使用向量计算资源。本模拟器在向量通道中初步添加了寄存器分组，即实现选择元素宽度 SEW 和向量寄存器系数 LMUL 配置与计算，并添加缩宽计算指令和拓宽计算指令，已在第三章中大致进行说明。基础算术计算通常包含拓宽版本，而缩宽指令更多涉及舍入，移位等操作。

对于向量拓宽指令和向量缩宽指令而言，源寄存器和目的寄存器的有效元素宽度不一致，如果向量结果操作数需要占用多个寄存器，则默认占用编号较低的向量寄存器。在发生拓宽和缩宽的情况下，应保证式 2-2 成立，即有效寄存器组系数应该按比例变大或者缩小。对于拓宽指令，会在寄存器重命名阶段计算 EMUL，然后对需求的寄存器组进行重命名操作并将物理寄存器编号写入到动态指令类中便于后续处理。

对于上述需要更改状态寄存器的指令，会在标量核的提交阶段对状态寄存器进行写入更新，而需求状态寄存器数值的指令，会等待前方指令更新完毕状态寄存器后再被发射到向量核中。

### 4.3.3 优化运算速度

由于向量核实现使用的向量重命名，逻辑寄存器会按照寄存器映射表映射到物理寄存器上。如果使用默认的不受干扰策略，不进行向量操作的元素需要目的寄存器查询映射表并将相应位置的元素数据从相应映射的旧目标寄存器中复制到新的物理寄存器对应位置上，从而实现目的寄存器中进行掩码操作的元素值不变。这对于掩码屏蔽的元素而言会浪费一定的性能与功耗。通过实现 vta 和 vma 标定的对向量元素的策略，可以对不关心的元素进行忽略，可以减少指令计算时间。原模拟器在向量通道中对于需要进行掩码操作的指令，会查询指令映射表并提取目的寄存器重命名的映射寄存器对应位置元素值命名为 Dstitem，在对应位置不进行向量计算时将结果赋值为 Dstitem，通过实现 vta 和 vma 的配置与设定，可以基于 vta 和 vma 的值，在向量通道调用向量功能单元之前进行判定，对于后续不需要使用到的非活跃元素和尾部元素直接将对应的值覆盖为 1，省去查询原值这个过程，从而提升计算效率。

### 4.3.4 添加异常检测

原模拟器没有定义任何异常，将指令交由向量访存单元与向量算术计算单元执行之后直接返回 NoFault，本模拟器将初步添加异常检测，通过断言对异常情况进行判定，若出现异常，则会停止模拟并进行错误汇报。发生以下情况会引发异常。

当非写入掩码值指令或者归约操作的标量结果试图以掩码寄存器时，会引发异常。

当前向量寄存器组系数仅支持 1/8、1/4、1/2、1、2、4、8，当有效向量寄存器组系数超过这个范围时，例如对 LMUL 为 1/8 的数据进行缩宽操作时，EMUL 将为 1/16，

这种情况会引发异常。在加载和存储指令中，如果传入的有效元素宽度不支持，则也会引发异常。

当指令尝试执行 vtype 中 vill 字段为 1 的指令时，会引发异常。

当 vxrm 寄存器中舍入模式无效时，会引发异常。

目前版本只添加了对非法指令的检测与判定，对于指令的恢复处理应当是在指令发生异常或者中断时，将指向当前正在执行计算的向量寄存器指针覆盖*epc，元素索引存入 vstart 状态寄存器，对异常和中断进行处理后从中断点恢复继续执行。但本模拟器暂时没有实现对发生异常情况进行处理的代码，发生异常时会报告错误并停止程序的模拟执行。测试用例是采用工具链生成的，不会包含上述非法情况。

## 4.4 本章小结

处理器核包含 CPU 模型与 ISA 指令功能描述两个方面，本章首先对 ISA 指令功能的实现方式进行了说明介绍，通过指令译码和指令功能单元定义处理源操作数的方式。通过修改 isa 描述文件，在译码阶段加入向量指令译码支持，并在向量通道中编写功能单元定义，加入向量指令的执行单元，从而对添加的指令进行功能实现。

本章后半部分说明了对模拟器的各种功能增添修改，包括对状态寄存器以及对应的向量指令的增加，不可知策略的实现，异常检测的初步添加等。

# 第五章 模拟器测试

## 5.1 Gem5 文件编译与调试标志

Gem5 工具利用 python 脚本对模拟的系统进行参数配置，将参数传递给 c++代码从而完成对模拟器模型的构造。当完成模拟器代码的修改配置之后，使用开源自动化构建工具 SCons[32]对其进行编译，可以生成 gem5 二进制文件。二进制文件具有以下几种参数选择，对应不同的优化程度和调试状态，如表 5.1 所示。

表5.1 Gem5 二进制文件类型表

| | |
|---|---|
| gem5.debug | 保证变量不被优化，包含调试标志，模拟速度较慢 |
| gem5.opt | 打开程序优化，保留部分调试标志，模拟调试速度较平衡 |
| gem5.fast | 打开程序优化，关闭调试，编译优化程度最高，速度较快 |
| gem5.prof | 搭配 gprof 分析工具使用 |
| gem5.perf | 使用 google perftools 分析工具使用 |

按照测试需求选择合适的二进制文件类型，gem5.debug 常用于开发和测试阶段，gem5.fast 常用于完成功能验证后对二进制文件进行模拟实验，gem5.opt 则介于两者之间，而 prof 和 perf 需要搭配分析工具从而提升 gem5 性能。本次选择 gem5.opt 对所配置的代码进行搭建。SCons 工具会读取 gem5 的脚本文件并解析指令，根据配置脚本的内容完成模拟系统的构建和配置，根据脚本文件中的指令生成 gem5 的二进制文件 gem5.opt。此时 gem5.opt 可启用模拟环境，通过传入 python 配置文件，gem5 会根据配置文件中的信息创建模拟对象、模拟参数和模拟系统。Python 配置文件中会配置目标系统的各模块参数与连接方式，并作为参数传入 gem5.opt 中。

可以进行传参配置的参数与功能如表 5.2 所示使用命令传参可以快速堆配置数据进行改动，连接关系可以手动对 python 配置文件进行修改。

在默认设定下，系统缓存行大小为 64 位，l1i_cache 大小为 32kB，l1d_cache 大小为 32kB，l2_cache 大小为 256kB，标量核 cpu 主频率为 2GHz，向量核频率为 1GHz，重命名寄存器配置为 40 个，访存队列缓存长 32，算术队列缓存长 32，重命名缓存长 32，通道设置为 8，最大向量长度位宽为 16384 位（最大可同时执行 256 个 64 位的元素），且向量内存单元与 l2_cache 相连，各组件之间的默认连接关系如图 3.1 所示。保持其他值不变，通过控制变量，使用不同配置的模拟器测试向量应用，可以得到不同情况下应用的执行速度与优化情况。

表5.2 向量模拟器参数配置表

| | |
|---|---|
| --cmd | 配置工作负载 |
| --output | 配置输出文件 |
| --cache_line_size | 配置系统缓存行大小，默认值 64 |
| --l1i_size | 一级指令缓存大小，默认值 32kB |
| --l1d_size | 一级数据缓存大小，默认值 32kB |
| --l2_size | 二级缓存大小，默认值 256kB |
| --mem_size | 内存大小，默认值 2GB |
| --cpu_clk | 标量核处理器频率，默认值 2GHz |
| --renamed_regs | 向量物理寄存器数，默认值 40 个 |
| --VRF_line_size | 每个向量通道中寄存器堆行大小，默认值 8 |
| --OoO_queues | 向量核是否乱序执行，默认乱序执行 |
| --mem_queue_size | 向量访存队列大小，默认值 32 |
| --arith_queue_size | 向量算术队列大小，默认值 32 |
| --rob_size | 重命名缓存大小，默认值 64 |
| --vector_clk | 向量核频率，默认值 1GHz |
| --v_lanes | 向量通道数，默认值 8 |
| --max_vl | 最大向量长度位宽，默认值 16384（256 个 64 位元素） |

表5.3 常用的调试标志及功能示意表

| | |
|---|---|
| VectorRename | 输出重命名操作信息 |
| ReorderBuffer | 输出重定向模块信息 |
| VectorInst | 输出向量指令汇编 |
| MemUnitWriteTiming | 输出访存单元加载信息 |
| MemUnitReadTiming | 输出访存单元读取信息 |
| VectorMemUnit | 输出访存相关地址信息 |
| InstQueueInst | 输出算术队列指令 |
| InstQueueRenInst | 输出算术队列重命名指令 |
| VectorLane | 输出向量通道执行过程信息 |
| Datapath | 输出数据通道数据处理和功能单元调用等信息 |

选择的 gem5 二进制类型为 opt，可以通过传入调试标志在测试结果中显示相应的调试信息，在 SCons 配置文件 SConscript 中添加调试标志，并在需要打印调试标记

处使用 DPRINTF(调试标志, 打印信息)将调试信息打印出，向量核中定义的常用调试标志及功能如表 5.3 所示，调试标志定义如图 5.1 所示，调试信息输出方式如图 5.2 所示，以 VectorInst 为例，在执行 gem5 时，传入--debug-flag=VectorInst，可以见到如图 5.3 所示的指令提示信息。

```
37 Source('vector_engine_interface.cc')
38 Source('vector_engine.cc')
39
40 DebugFlag('VectorEngineInterface')
41 DebugFlag('VectorEngine')
42 DebugFlag('VectorEngineInfo')
43 DebugFlag('VectorInst')
```

图5.1 调试标志定义代码示意图

```
283
284     if (insn.arith1Src()) {
285         DPRINTF(VectorInst,"inst: %s %s%d v%d %s            PC 0x%X\n",
286             insn.getName(),reg_type,insn.vd(),insn.vs2(),masked,*(uint64_t*)&p
    c );
287         DPRINTF(VectorRename,"renamed inst: %s %s%d v%d %s  old_dst v%d     "
288             "    PC 0x%X\n",insn.getName(),reg_type,PDst,Pvs2,mask_ren.str(),
289             POldDst,*(uint64_t*)&pc);
290     }
```

图5.2 调试信息输出方式示意图

```
4595300000: system.cpu.ve_interface.vector_engine: inst: vfmerge_vf v6 v6 f0 v0.m       PC 0x1077C
4595337000: system.cpu.ve_interface.vector_engine: inst: vfsub_vv v6 v16 v6           PC 0x10780
4595371000: system.cpu.ve_interface.vector_engine: inst: vfmul_vf v16 v6 f2           PC 0x10784
4595410000: system.cpu.ve_interface.vector_engine: inst: vfmul_vf v22 v6 f1           PC 0x10788
4595447000: system.cpu.ve_interface.vector_engine: inst: vfsub_vv v16 v11 v16          PC 0x1078C
4595484000: system.cpu.ve_interface.vector_engine: inst: vfsub_vv v16 v16 v22          PC 0x10790
4595521000: system.cpu.ve_interface.vector_engine: inst: vfmul_vv v22 v16 v16          PC 0x10794
4595558000: system.cpu.ve_interface.vector_engine: inst: vfmadd_vv v17 v18 v16          PC 0x10798
4595592000: system.cpu.ve_interface.vector_engine: inst: vfmadd_vv v17 v19 v16          PC 0x1079C
4595629000: system.cpu.ve_interface.vector_engine: inst: vfmadd_vv v17 v21 v16          PC 0x107A0
4595665000: system.cpu.ve_interface.vector_engine: inst: vfmadd_vv v17 v20 v16          PC 0x107A4
4595701000: system.cpu.ve_interface.vector_engine: inst: vfmadd_vv v17 v5 v16          PC 0x107A8
4595738000: system.cpu.ve_interface.vector_engine: inst: vfmadd_vv v17 v16 v22          PC 0x107AC
4595775000: system.cpu.ve_interface.vector_engine: inst: vfadd_vf v5 v17 f0           PC 0x107B0
4595811000: system.cpu.ve_interface.vector_engine: inst: vfcvt_x_f_v v6 v6             PC 0x107B4
4595850000: system.cpu.ve_interface.vector_engine: inst: vadd_vx v6 v6 x11           PC 0x107B8
4595889000: system.cpu.ve_interface.vector_engine: inst: vsll_vi v6 v6  23           PC 0x107BC
```

图5.3 传递调试标志输出调试信息示意图

## 5.2 测试编译软件

搭建好模拟器环境并配置好连接关系之后，需要在模拟器上执行应用程序来测试模拟器的功能与性能，用编程语言编写的程序需要通过软件翻译转变为机器语言才能被模拟器识别。

GNU Compiler Collection(GCC)是第一款支持 RISC-V 指令的编译器[33]。Riscv-gnu-toolchain 可以完成 RISC-V 与 C 或者 C++语言的交叉编译[34]，将测试用程序转化

为 RISC-V 模拟器可以识别的可执行文件。伴随着向量扩展集的不断修订，工具链也初步对向量扩展集进行了一定的支持。通过编译器对 c++代码进行交叉编译，可以得到 RISC-V 指令的可执行文件，使用配置的模拟器进行测试，可以获得具有参考价值的测试结果。Riscv-gnu-toolchain 具有许多分支，本次测试采用的是 rvv_next 分支，编译使用的指令集为 rv64gcv，即 RV64IMAFDCV，其中 I 为整数扩展为基础指令集，M 为乘除扩展，A 为标准原子扩展，F 和 D 为单双精度浮点扩展，C 为压缩扩展，V 为向量扩展。

Riscv-gnu-toolchain 编译完成后会在安装路径下产生许多 riscv64-unknown-linux-gnu 文件，如图 5.4 所示。其中 riscv64-unknown-linux-gnu-g++文件即是用于将 c++文件转换为 riscv 可执行文件的，riscv64-unknown-linux-gnu-objdump 则是用来将 riscv 可执行文件反汇编成指令，以后续将介绍的测试软件 streamcluster 为例，反汇编结果如图 5.5 所示。

```
qemu-riscv32                      riscv64-unknown-linux-gnu-g++         riscv64-unknown-linux-gnu-gdb           riscv64-unknown-linux-gnu-objdump
qemu-riscv64                      riscv64-unknown-linux-gnu-gcc         riscv64-unknown-linux-gnu-gdb-add-index riscv64-unknown-linux-gnu-ranlib
riscv64-unknown-linux-gnu-addr2line riscv64-unknown-linux-gnu-gcc-12.0.1 riscv64-unknown-linux-gnu-gfortran     riscv64-unknown-linux-gnu-readelf
riscv64-unknown-linux-gnu-ar      riscv64-unknown-linux-gnu-gcc-ar      riscv64-unknown-linux-gnu-gprof         riscv64-unknown-linux-gnu-run
riscv64-unknown-linux-gnu-as      riscv64-unknown-linux-gnu-gcc-nm      riscv64-unknown-linux-gnu-ld            riscv64-unknown-linux-gnu-size
riscv64-unknown-linux-gnu-c++     riscv64-unknown-linux-gnu-gcc-ranlib  riscv64-unknown-linux-gnu-ld.bfd        riscv64-unknown-linux-gnu-strings
riscv64-unknown-linux-gnu-c++filt riscv64-unknown-linux-gnu-gcov        riscv64-unknown-linux-gnu-lto-dump      riscv64-unknown-linux-gnu-strip
riscv64-unknown-linux-gnu-cpp     riscv64-unknown-linux-gnu-gcov-dump   riscv64-unknown-linux-gnu-nm
riscv64-unknown-linux-gnu-elfedit riscv64-unknown-linux-gnu-gcov-tool   riscv64-unknown-linux-gnu-objcopy
riscv@/opt/RISCV/bin:
```

图5.4 编译后主路径文件

```
bin/streamcluster_vector:     file format elf64-littleriscv


Disassembly of section .text:

00000000000103e0 <__cxa_bad_cast>:
   103e0:   1141                    addi    sp,sp,-16
   103e2:   4521                    li  a0,8
   103e4:   e406                    sd  ra,8(sp)
   103e6:   3f36a0ef            jal ra,7afd8 <__cxa_allocate_exception>
   103ea:   00127797            auipc   a5,0x127
   103ee:   02e7b783            ld  a5,46(a5) # 137418 <_GLOBAL_OFFSET_TABLE_+0xac8>
   103f2:   07c1                    addi    a5,a5,16
   103f4:   e11c                    sd  a5,0(a0)
   103f6:   00127617            auipc   a2,0x127
   103fa:   c6a63603            ld  a2,-918(a2) # 137060 <_GLOBAL_OFFSET_TABLE_+0x710>
   103fe:   00127597            auipc   a1,0x127
   10402:   4025b583            ld  a1,1026(a1) # 137800 <_GLOBAL_OFFSET_TABLE_+0xeb0>
   10406:   4c2040ef            jal ra,148c8 <__cxa_throw>

000000000001040a <__cxa_bad_typeid>:
```

图5.5 streamcluster 程序反汇编示意图

## 5.3 测试方法

测试划分为指令功能测试与模拟器性能测试两部分，前者测试单条指令功能实现是否完全，后者测试在对应工作负载下模拟器的运行性能差别。

### 5.3.1 指令功能测试

指令功能测试是通过编写指令功能相关的汇编指令，使用 riscv-gnu-toolchain 转换为可执行程序并执行，通过比对执行结果与预期结果是否一致来确认向量指令功能是否正确实现。

通过工具链新增指令具有很多方式，对应的工作量和汇编友好程度不同，最基础的方式是直接添加机器码，需要记住对应机器码的二进制含义，如在图 5.5 中，ld a5, 46（a5）对应的机器码为 02e7b783，使用.word 0x02e7b783 则可定义这条指令，这种方式较为低效。另一种方式是借助编译工具进行修改，使得可以在 c 程序中内嵌汇编或者直接编写汇编程序，并通过工具链转换为模拟器可识别的可执行文件[35]。C++的内嵌汇编模板如下，汇编语句模板为必须填写部分，输出、输入、破坏描述部分为可选部分，asm 为 GCC 关键字 asm 的宏定义，而 volatile 则指示编译器不对内联汇编进行优化，对于向量寄存器而言，可以采用类似的方式进行编写测试。

asm volatile (汇编语句模板 ： 输出部分 ： 输入部分 ： 破坏描述部分)

例如，对于一个简单的浮点加法运算，可以编写以下测试语句。

asm volatile ("fadd.d %0,%1,%2" : "=f" (fd) : "f" (fs1), "f" (fs2))

通过提取配置 fs1 和 fs2 寄存器的值，然后执行语句之后查看 fd 的值是否为预期值，从而判断指令是否实现正确，可以通过预设 fs1 和 fs2 寄存器中的值，计算结果并使用相等跳转语句 beq 来比较模拟器实现与预期值之间的区别，从而跳转选择测试成功分支或者测试失败分支[36]。

```
19 RVTEST_CODE_BEGIN
20
21
22   li t0, -1
23   vsetvli t1, t0, e16,m4,ta,ma
24   la a2, tdat
25   vle16.v v8, (a2)
26
27   vsetvli t1, t0, e16,m4,ta,ma
28   vle16.v v4, (a2)
29   la a2, tdat+8
30
31   vsetvli t1, t0, e16,m4,ta,ma
32   vle16.v v12, (a2)
33
34
35   li t0, 32
36   vsetvli t1, t0, e16,m4,ta,ma
37   vadd.vv v4, v8, v12
38
39   li t0, -1
40   vsetvli t1, t0, e16,m4,ta,ma
41   la a1, res
42   vse16.v v4, (a1)
43
44   TEST_CASE(2, t0, 0x2000afff8, ld t0, 0(a1); addi a1, a1, 8)
45   TEST_CASE(3, t0, 0xfffff000f006ffff, ld t0, 0(a1); addi a1, a1, 8)
46   TEST_CASE(4, t0, 0xefffff000ffff, ld t0, 0(a1); addi a1, a1, 8)
```

图5.6 指令功能测试示意图

RISC-V 基金会提供了对 RISC-V 指令集功能验证使用的自测试用例，使用汇编语言编写且可以使用所有 RISC-V 常见的汇编指令。代码搭载于 github 的 riscv-software-src/riscv-tests 仓库。其中存在用于向量测试的分支，由于向量指令中大部分指令是标量指令并行化计算的体现，所以在测试用例中，可以编写向量指令的汇编代码，并提前使用标量计算相同的内容并保存结果，然后将二者进行对比，若一致则说明向量指令实现正确，向量加法的一个测试用例代码如图 5.6 所示，首先调用 vsetvli 配置向量计算的各参数，将 SEW 设为 16 位、LMUL 设定为 4、同时开启尾部不可知策略和掩码不可知策略，然后向 a2 中加载入基址，将 v4、v8、v12 都按单位跨步加载方式载入 16 位元素，调用 vadd.vv 进行代码计算，最后向 v4 中载入正确结果然后进行比对。

### 5.3.2 模拟器性能测试

模拟器性能测试是通过 riscv-gnu-toolchain 编译注重多数据计算的程序代码，对比不同配置下指令执行的速度来对向量指令计算性能与优化程度进行评估。

选用的测试应用分别为 streamcluster、particalfilter、canneal、swaptions、pathfinder、blacksholes[37-42]等。Steamcluster 是一个计算来自输入流数据的聚类算法程序，常用于研究分类问题和数据挖掘。Particalfilter 是通过寻找状态空间中随机样本，近似表示概率密度函数的用来计算粒子滤波的程序。Canneal 是用来模拟具备缓存感知的退火优化芯片设计的路由成本的程序。Swaptions 是用来交换期权组合的定价，计算固定债务、浮动债务和期权互换的工具。Pathfinder 是用来处理包含重叠向量路径的工作负载的程序。Blacksholes 是使用 Black-Scholes 偏微分方程对期权等金融衍生品价格进行数学建模分析求解的应用。以上应用可以将部分标量计算操作使用向量运算进行替代。

使用 riscv-gnu-toolchain 工具并设置目标为 rv64fcv 编译、链接出可执行文件，这个情况下生成的可执行文件中含有向量指令，传入参数-DUSE_RISCV_VECTOR 重新编译、链接出可执行文件，这个情况生成的可执行文件中不含向量指令，传入-O 可以使得工具对程序进行优化，这个情况生成的可执行文件为优化后的带向量指令。配置不同向量长度的模拟器，并将这几个版本的可执行文件作为模拟器的工作负载，记录执行所耗费时间，进行对比可以获得向量扩展架构对程序的优化比。

通常而言，测试平台不同，配置方式不同，计算耗费的时长也不同。通过运行应用标量版本和向量版本的可执行文件，通过调节不同的配置参数，使用模拟器进行仿真，并依照应用向量需求、运算结果之间的相互关系，大致进行结果分析。

## 5.4 测试结果与分析

通过表 5.4 可以看出各待测应用在向量使用长度范围、内存访问寻址类型、向量算术计算指令类型、内部互联指令使用类型以及与标量核数据交互密集程度等方面的使用情况。通过调整配置使用不同模拟器对测试程序进行测试，从而总结规律。

表5.4 测试程序与应用范围测试表

| 应用程序 | | Blackscholes | Canneal | Particle Filter | Pathfinder | Streamcluster | Swaptions |
|---|---|---|---|---|---|---|---|
| 向量长度 | 短 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | 中 | ✓ | ✓ | ✓ | ✓ | | ✓ |
| | 长 | ✓ | | ✓ | ✓ | | ✓ |
| 访存方式 | 单位步幅寻址 | ✓ | | ✓ | ✓ | | ✓ |
| | 索引寻址 | ✓ | ✓ | | | ✓ | |
| 向量通道 | 算术计算 | | ✓ | ✓ | ✓ | ✓ | ✓ |
| | 掩码计算 | ✓ | | ✓ | | ✓ | ✓ |
| 内部互联 | 滑动操作 | ✓ | | | ✓ | | |
| | 归约计算 | | ✓ | | | ✓ | ✓ |
| 是否与标量核通信 | | | ✓ | ✓ | | ✓ | |

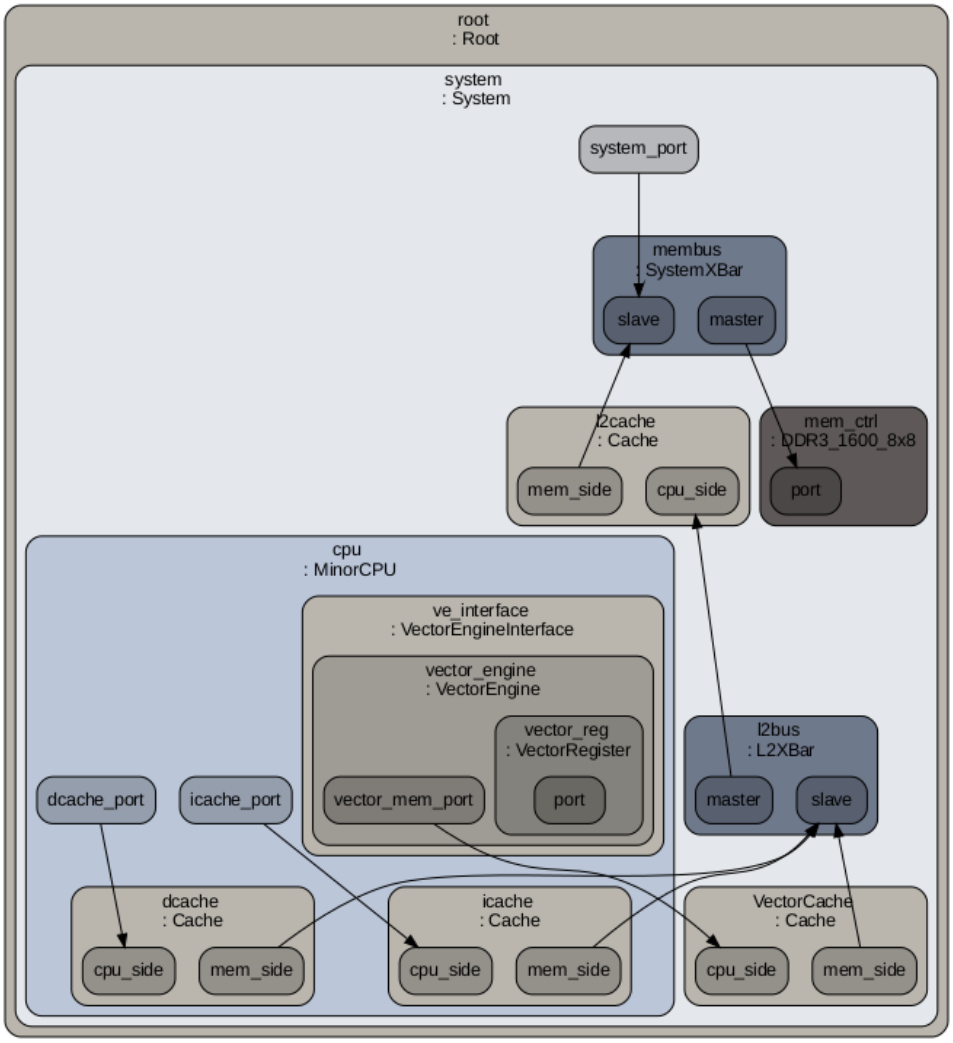


图5.7 模拟器连接示意图

表5.5 Blackscholes 测试情况结果表

| | blackscholes | | | | | |
|---|---|---|---|---|---|---|
| 向量通道数 | 一通道 | | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
| 运行速度(s) | 166.36 | 153.79 | 150.47 | 149.1 | 148.26 | 148.68 |
| 向量加速比 | 1.217 | 1.316 | 1.345 | 1.358 | 1.365 | 1.362 |
| 总指令数 | 47084388 | 46544933 | 46269967 | 46137273 | 46069521 | 46035649 |
| 向量通道数 | 二通道 | | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
| 运行速度(s) | 149.56 | 143.26 | 141.06 | 140.05 | 136.49 | 138.76 |
| 向量加速比 | 1.354 | 1.413 | 1.435 | 1.445 | 1.483 | 1.459 |
| 总指令数 | 47084290 | 46545544 | 46272543 | 46132996 | 46068666 | 46034986 |
| 向量通道数 | 四通道 | | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
| 运行速度(s) | 144.74 | 138.51 | 134.82 | 133.34 | 132.55 | 132.15 |
| 向量加速比 | 1.399 | 1.462 | 1.502 | 1.518 | 1.527 | 1.532 |
| 总指令数 | 47090232 | 46545856 | 46272365 | 46137024 | 46065202 | 46035047 |
| 向量通道数 | 八通道 | | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
| 运行速度(s) | 142 | 139.6 | 133.22 | 131.81 | 130.83 | 129.92 |
| 向量加速比 | 1.426 | 1.450 | 1.520 | 1.536 | 1.547 | 1.558 |
| 总指令数 | 47089675 | 46544755 | 46273980 | 46131928 | 46064616 | 46034016 |
| 向量算术指令数 | 640000 | 320000 | 160000 | 80000 | 40000 | 20000 |
| 向量配置指令数 | 64000 | 32000 | 16000 | 8000 | 4000 | 2000 |
| 向量访存指令数 | 22400 | 11200 | 5600 | 2800 | 1400 | 700 |
| 标量指令数 | 46357988 | 46181733 | 46088367 | 46046473 | 46024121 | 46012949 |
| 向量操作 | 10009600 | 10009600 | 10009600 | 10009600 | 10009600 | 10009600 |
| 向量通道数 | 标量 | | | | | |
| 最大向量长度(bit) | 0 | | | | | |
| 运行速度(s) | 202.44 | | | | | |
| 向量加速比 | 1.000 | | | | | |
| 总指令数 | 70994112 | | | | | |

表 5.4 列出了 blackscholes 应用在不同配置情况下的 gem5 运行时间、并给出了向量加速比，总指令数等信息。可以看出，向量运行的速度随着指令通道的增加而变快，相同通道而言，由于 blackscholes 应用支持长向量的计算，所以对于较大的最大向量长度而言，也可以发挥模拟器的功效。应用总体加速时间与向量指令操作跟标量指令数的比相关，表中可得，向量指令配置的操作占总操作指令的 17.5%~17.8%之间，因此向量加速比也会受到这个值的制约。运算结果如图 5.8 和图 5.9 所示。

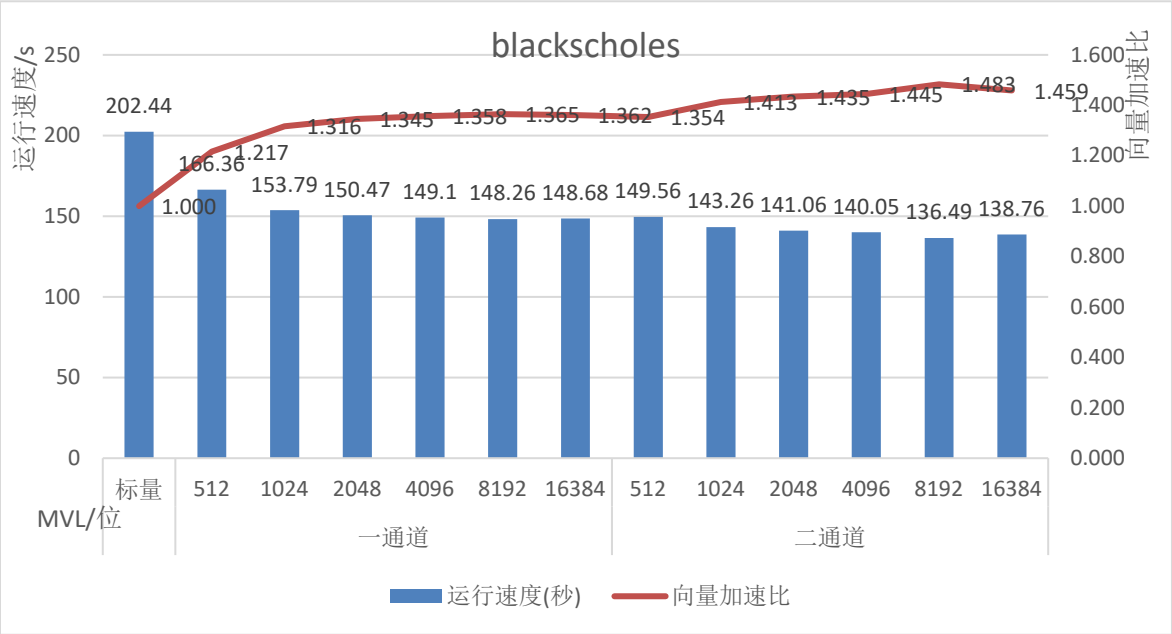


图5.8 Blackscholes 一二通道数据示意图

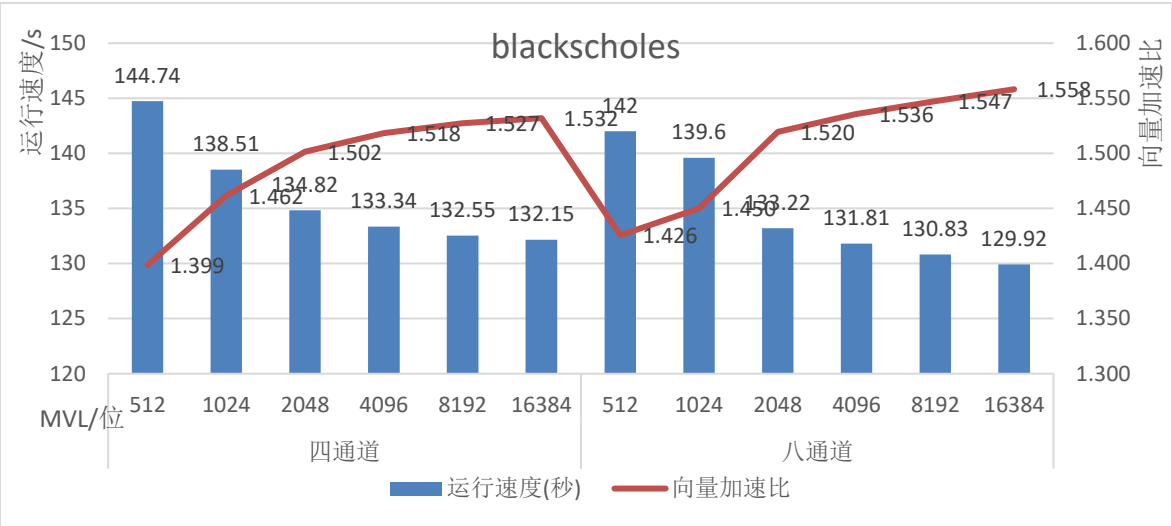


图5.9 Blackscholes 三四通道数据示意图

表5.6 Canneal 测试情况结果表

| | canneal | | | | | |
|---|---|---|---|---|---|---|
| | 一通道 | | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
| 运行速度(s) | 23.53 | 24.04 | 35.73 | 37.63 | 55.79 | 95.38 |
| 总指令数 | 4346898 | 4196420 | 4189918 | 4195146 | 4194738 | 4195999 |
| 向量算术计算指令数 | 179918 | 138630 | 137894 | 137894 | 137894 | 137894 |
| 向量配置指令数 | 133613 | 118130 | 117854 | 117854 | 117854 | 117854 |
| 向量访存指令数 | 56378 | 40895 | 40619 | 40619 | 40619 | 40619 |
| 标量指令数 | 3976989 | 3898765 | 3893551 | 3898779 | 3898371 | 3899632 |
| 向量操作 | 2155460 | 2428264 | 2708728 | 3270392 | 4393720 | 6640376 |
| | 八通道 | | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
| 运行速度(s) | 18.87 | 18.85 | 19.51 | 23.67 | 26.2 | 36.59 |
| 总指令数 | 4341931 | 4196035 | 4194171 | 4188913 | 4194111 | 4190009 |
| 向量算术计算指令数 | 179918 | 138630 | 137894 | 137894 | 137894 | 137894 |
| 向量配置指令数 | 133613 | 118130 | 117854 | 117854 | 117854 | 117854 |
| 向量访存指令数 | 56378 | 40895 | 40619 | 40619 | 40619 | 40619 |
| 标量指令数 | 3972022 | 3898380 | 3897804 | 3892546 | 3897744 | 3893642 |
| 向量操作 | 2155460 | 2428264 | 2708728 | 3270392 | 4393720 | 6640376 |
| | 标量 | | | | | |
| 运行速度 | 15.2 | | | | | |
| 标量指令数 | 5318076 | | | | | |

对于 canneal，由于 canneal 的计算需要使用到向量通道中的内部互联，由表 5.6 可以看到如果使用较小的向量数据集，随着 MVL 的增大，向量算术计算指令数不变，而由于向量归约计算会对整个向量进行计算，此时向量的算术操作数量（即向量调用算术功能单元的次数）会随着 MVL 的增大而随之增大，此时较大的 MVL 配置反而会拖慢向量计算，从而降低指令优化速度。如果向量应用使用向量化的指令不多，使用向量架构并不能起到加速计算的效果，可能还会导致负优化。

Swaptions 应用的测试情况如表 5.7 所示，可以看到，该应用相比于 blackscholes 向量操作占总体操作的比例更高，大致为 79.5%~ 93.4%，因此可以看到加速效果比 blackscholes 应用更加良好。随着 MVL 的增大，标量指令数量如基址加载等需求逐渐降低，因此整体运算速度更快，但在 MVL 到达 8192 之后优化程度有限。

表5.7 Swaptions 测试情况结果表

| | swaptions | | | | | |
|---|---|---|---|---|---|---|
| | 一通道 | | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
| 运行速度(s) | 174.88 | 144.9 | 136.69 | 125.2 | 120.52 | 119.53 |
| 向量加速比 | 0.742 | 0.896 | 0.950 | 1.037 | 1.077 | 1.086 |
| 总指令数 | 10001181 | 5768630 | 3668833 | 2622567 | 2091840 | 1831229 |
| | 二通道 | | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
| 运行速度(s) | 132.87 | 101.28 | 88.32 | 78.45 | 74.27 | 73.91 |
| 向量加速比 | 0.977 | 1.282 | 1.470 | 1.655 | 1.748 | 1.756 |
| 总指令数 | 10000545 | 5767812 | 3667515 | 2622459 | 2092066 | 1831357 |
| | 四通道 | | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
| 运行速度(s) | 105.48 | 77.93 | 63.22 | 54.06 | 50.6 | 48.93 |
| 向量加速比 | 1.231 | 1.666 | 2.053 | 2.401 | 2.565 | 2.653 |
| 总指令数 | 10000810 | 5768957 | 3668541 | 2615664 | 2090979 | 1830659 |
| | 八通道 | | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
| 运行速度(s) | 95.1 | 66.6 | 52.8 | 42.66 | 39.09 | 38.84 |
| 向量加速比 | 1.365 | 1.949 | 2.459 | 3.043 | 3.321 | 3.342 |
| 总指令数 | 9994479 | 5768944 | 3667975 | 2621753 | 2091275 | 1829321 |
| 向量算术计算指令数 | 3022848 | 1511424 | 755712 | 377856 | 188928 | 94464 |
| 向量配置指令数 | 310280 | 155144 | 77576 | 38792 | 19400 | 9704 |
| 向量访存指令数 | 440320 | 220160 | 110080 | 55040 | 27520 | 13760 |
| 标量指令数 | 6221031 | 3882216 | 2724607 | 2150065 | 1855427 | 1711393 |
| 向量操作数 | 24175616 | 24175104 | 24174848 | 24174720 | 24174656 | 24174624 |
| | 标量 | | | | | |
| 运行速度(s) | 129.81 | | | | | |
| 向量加速比 | 1.000 | | | | | |
| 总指令数 | 47081435 | | | | | |

表5.8 Pathfinder 测试情况结果表

| | pathfinder | | | | |
|---|---|---|---|---|---|
| | 一通道 | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 |
| 运行速度(s) | 455.69 | 357.01 | 288.69 | 270.14 | 257.86 |
| 向量加速比 | 1.754 | 2.239 | 2.769 | 2.9595 | 3.100 |
| 总指令数 | 32143154 | 18731286 | 12026428 | 8674296 | 6998184 |
| 标量指令数 | 24827954 | 15073686 | 10197628 | 7759896 | 6540984 |
| | 二通道 | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 |
| 运行速度(s) | 384.5 | 282.04 | 221.36 | 196.62 | 185.57 |
| 向量加速比 | 2.079 | 2.835 | 3.612 | 4.066 | 4.308 |
| 总指令数 | 32145292 | 18731133 | 12026687 | 8672758 | 6995938 |
| 标量指令数 | 24830092 | 15073533 | 10197887 | 7758358 | 6538738 |
| | 四通道 | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 |
| 运行速度(s) | 360.6 | 246.12 | 178.99 | 162.78 | 149.49 |
| 向量加速比 | 2.217 | 3.248 | 4.467 | 4.911 | 5.348 |
| 总指令数 | 32143127 | 18731654 | 12025060 | 8673779 | 6997669 |
| 标量指令数 | 24827927 | 15074054 | 10196260 | 7759379 | 6540469 |
| | 八通道 | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 |
| 运行速度(s) | 341.81 | 235.05 | 167.01 | 150.27 | 134.77 |
| 向量加速比 | 2.339 | 3.401 | 4.787 | 5.320 | 5.932 |
| 总指令数 | 32143697 | 18732458 | 12025407 | 8672973 | 6997835 |
| 标量指令数 | 24828497 | 15074858 | 10196607 | 7758573 | 6540635 |
| 向量操作 | 39014400 | 39014400 | 39014400 | 39014400 | 39014400 |
| 向量算术计算指令数 | 4064000 | 2032000 | 1016000 | 508000 | 254000 |
| 向量配置指令数 | 812800 | 406400 | 203200 | 101600 | 50800 |
| 向量访存指令数 | 2438400 | 1219200 | 609600 | 304800 | 152400 |
| | 标量 | | | | |
| 运行速度(s) | 799.47 | | | | |
| 总指令数 | 342214240 | | | | |

表5.9 Stream cluster 测试情况结果表

| | steamcluster | | | | | |
|---|---|---|---|---|---|---|
| | 一通道 | | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
| 运行速度(s) | 247.37 | 227.69 | 227.2 | 267.94 | 359.92 | 557.57 |
| 向量加速比 | 1.534 | 1.666 | 1.670 | 1.416 | 1.054 | 0.681 |
| 总指令数 | 35189425 | 30436287 | 28065901 | 26878116 | 26880114 | 26879795 |
| | 二通道 | | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
| 运行速度(s) | 211.69 | 183.62 | 183.34 | 195.06 | 249.62 | 363.83 |
| 向量加速比 | 1.792 | 2.066 | 2.070 | 1.945 | 1.520 | 1.043 |
| 总指令数 | 35187659 | 30436215 | 28065962 | 26874524 | 26879973 | 26879760 |
| | 四通道 | | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
| 运行速度(s) | 196.34 | 168.36 | 156.89 | 161.85 | 195.66 | 255.33 |
| 向量加速比 | 1.933 | 2.254 | 2.418 | 2.344 | 1.939 | 1.486 |
| 总指令数 | 35189158 | 30439291 | 28061273 | 26878692 | 26879130 | 26873679 |
| | 八通道 | | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
| 运行速度(s) | 186.69 | 157.51 | 144.84 | 148.71 | 169.5 | 211.03 |
| 向量加速比 | 2.032 | 2.409 | 2.620 | 2.551 | 2.239 | 1.798 |
| 总指令数 | 35183399 | 30434990 | 28061148 | 26878514 | 26878687 | 26879248 |
| 向量算术计算指令数 | 2077566 | 1286222 | 890550 | 692714 | 692714 | 692714 |
| 向量配置指令数 | 1483094 | 1087422 | 889586 | 790668 | 790668 | 790668 |
| 向量访存指令数 | 1582688 | 791344 | 395672 | 197836 | 197836 | 197836 |
| 标量指令数 | 30040051 | 27270002 | 25885340 | 25197296 | 25197469 | 25198030 |
| 向量操作 | 30083392 | 34843776 | 44364544 | 63406080 | 76005888 | 1.01E+08 |
| | 标量 | | | | | |
| 运行速度(s) | 379.43 | | | | | |
| 总指令数 | 138170187 | | | | | |

选取八通道的 streamcluster、swaptions、pathfinder 应用向量操作占总操作比例与向量加速比对比如图 5.10 所示。

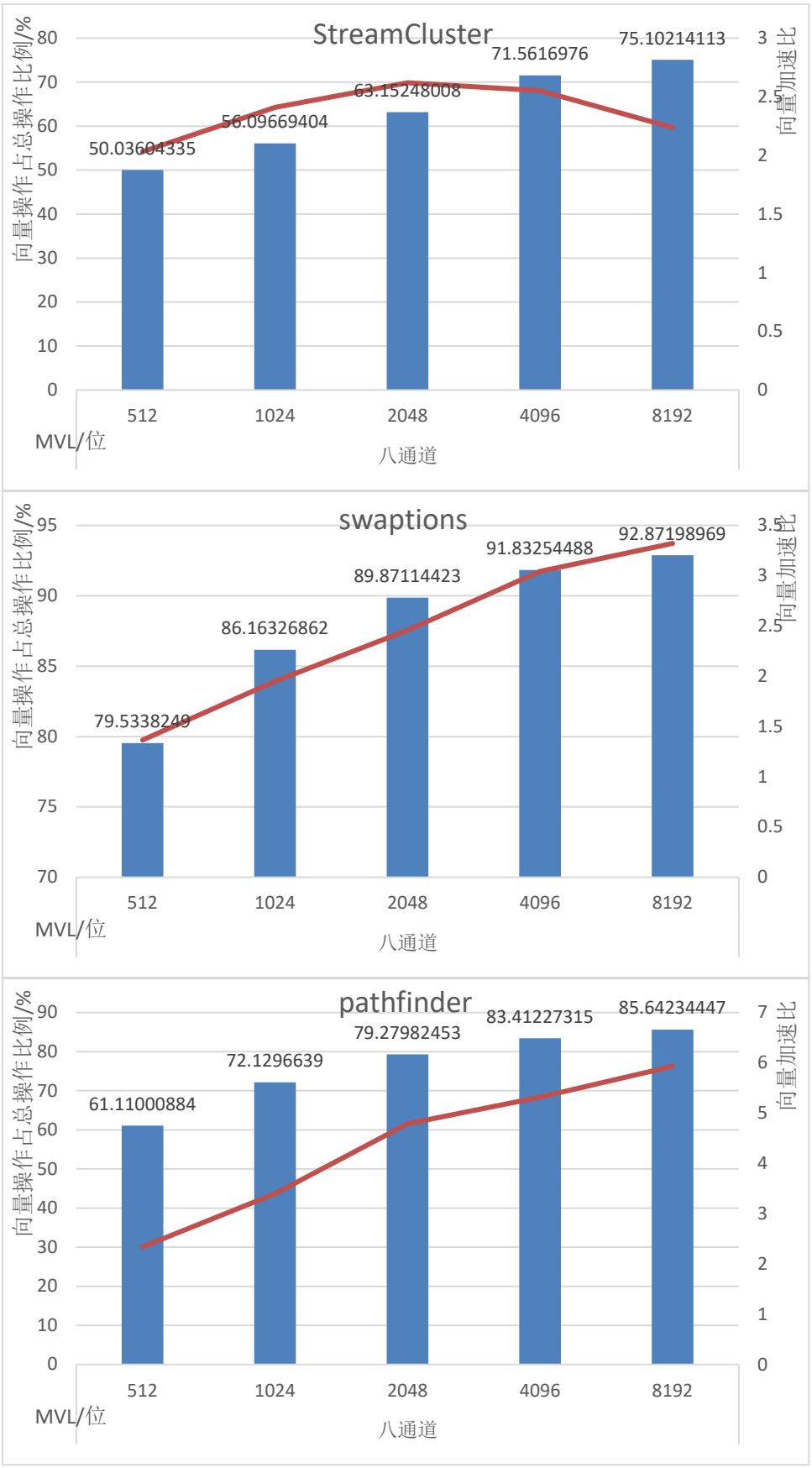


图5.10 八通道配置下向量操作占总操作比率与向量加速比对比图

由图 5.10 可以看到，大部分情况下，向量操作占总操作比例与向量运算加速的效果大致是相同的变化方向的，随着 MVL 的增大，向量操作占总操作的比重会上升，向量处理相同计算量和数据存取操作的速度要快于标量计算，因此如果应用支持长向量操作，那么在面对长的 MVL 时能够较好地利用向量核，从而使得总体指令数下降，而 streamcluster 与 canneal 一致，在 MVL 设定为 2048 位时还能够利用向量核进行优化，MVL 大于 2048 时，由于选定的向量计算不需要使用到更大的空间，向量执行所需的指令数没有下降，而由于 streamcluster 需要使用到归约计算，所以向量操作反而随着 MVL 的增多的变多。

表5.10 Particle Filter 测试情况结果表

| | Particle Filter | | | | | |
|---|---|---|---|---|---|---|
| | 一通道 | | | | | |
| 最大向量长度(bit) | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
| 运行速度(s) | 351.89 | 316.42 | 298.14 | 288.66 | 283.69 | 282.19 |
| 总指令数 | 75476015 | 68931064 | 65646108 | 64020197 | 63190240 | 62732149 |
| 向量算术计算指令数 | 3852015 | 1926071 | 963099 | 481611 | 240867 | 120499 |
| 向量配置指令数 | 925881 | 462973 | 231519 | 115791 | 57927 | 28997 |
| 向量访存指令数 | 9344 | 4672 | 2336 | 1168 | 584 | 292 |
| 标量指令数 | 70688775 | 66537348 | 64449154 | 63421627 | 62890862 | 62582361 |
| 向量操作 | 16113904 | 16114176 | 16114720 | 16115776 | 16117888 | 16122368 |
| | 二通道 | | | | | |
| 运行速度(s) | 298.1 | 266.93 | 249.48 | 242.7 | 234.02 | 234.71 |
| | 四通道 | | | | | |
| 运行速度(s) | 271.9 | 242.19 | 223.68 | 214.63 | 209.8 | 209.29 |
| | 八通道 | | | | | |
| 运行速度(s) | 262.09 | 227.98 | 209.88 | 201.32 | 197.48 | 196.56 |
| | 标量 | | | | | |
| 运行速度(s) | 234.21 | | | | | |
| 标量操作数 | 89508339 | | | | | |

可以看到，对于 particle filter 而言，首先向量操作占比不高，在低通道的实现中，较小的最大向量长度配置下出现了负优化。是由于 Particle filter 这个应用的执行的算

术计算包含带掩码取对数、余弦计算、平方根计算等特殊操作，向量核需要频繁跟标量核进行通信。这种情况下对标量变量的需求高，而标量核会暂缓发射向量指令给向量核直至标量依赖数据准备完毕，因此对于低通道短最大向量长度出现了负优化的情况，随着通道数和最大向量长度的提升，向量指令执行效率变高，从而运行速度变快。

## 5.5 总结分析

在算术计算方面，向量架构通过设置多向量通道并行化处理数据，提高一条指令包含的元素操作数量，从而降低总指令数，在算术操作次数变化不大的情况下提升计算效率。在访存方面，向量架构能够通过大的访存带宽，在一个周期内获得多个元素的值，通过加快数据的准备速度来提升运行效率。

由于只有向量操作能够在向量核中进行并行化计算提升计算效率，因此向量架构对应用的加速受到应用中能够进行向量化操作的指令数占整体指令操作数的比例影响。通常而言，对于相同的操作，越多指令能够向量化执行，则使用向量核进行加速的表现越好。向量应用如果支持长的 vl 向量元素长度，则加大最大向量长度 MVL 的配置可以使得向量配置和向量计算所需的指令减少，从而加速向量计算的效率；如果不支持长的 vl 向量元素长度，则加大最大向量长度 MVL 的配置对向量计算效率影响不大。增大最大向量长度会影响能够进行向量化操作的指令占比。Blackscholes 虽然支持长的向量计算，但是由于向量指令占总操作指令比率较低，因此在将 MVL 扩大到 4096 位后，并不能明显减少总指令数，因此优化效果增加得不明显。而对于 swaptions 应用，在八通道下，向量总指令数随着 MVL 的扩大而变少，减少了向量配置操作数量，能够向量化的操作由 79.53%到 93.4%不等，使得一条向量算术操作能够进行更多元素操作，对应的最大向量加速比也随着向量化程度的上涨而上升。

对于不同的应用而言，优化程度需要根据向量操作所用到的操作类型进行分析。对于同一个应用而言，加大向量通道数能够体现向量计算并行化的特点，使得一个周期进行多个指令操作，从而加速指令的计算速度。对于 canneal 而言，不支持较长的向量长度 vl 配置，增大 MVL 无法降低指令总数，由于操作需要使用到归约计算，归约计算与最大的 MVL 相关，增大 MVL 会使得不必要的操作变多，从而降低计算效率，产生负优化。而 streamcluster 虽然也需要使用到归约运算，但是由于支持长的向量长度 vl，在稍微增大 MVL 的配置时，向量操作对指令操作的优化占主导，因此向量加速比上升，而随着继续增大 MVL，归约操作增加的向量操作的负优化占主导，因此加速比下降。

## 5.6 本章小结

本章介绍了 gem5 工具的编译及调试方法、RISC-V 交叉编译工具 Riscv-gnu-toolchain 及模拟器指令功能测试和模拟器性能测试的方式，通过使用不同的配置设置模拟器，并针对不同的向量应用进行结果测试，对结果分析得出模拟器各方面性能与应用侧重之间的关系与影响。

在实际设计中，我们可以通过对目标应用环境进行总结提炼并进行反汇编分析，通过模拟器对应用执行环境和目标等数据进行模拟从而选择最合适的配置进行设置。

# 第六章 总结与展望

## 6.1 总结

本文基于 gem5 工具完成了 RISC-V 向量扩展指令集的处理器模拟器的配置实现，通过对 gem5 中模型的修改扩充，添加对应模块、功能实现，从而完成处理器整体的设计，本文的主要工作包括：

（1）通过修改 gem5 提供的功能流水模型，在 Decode 阶段添加了向量指令的译码识别和分类代码，在执行阶段改动事件实现标量核和向量核分离实现。

（2）对向量核的微体系结构、数据交换等方面进行了研究介绍，对于标量核实现的是指令顺序执行，向量核实现的是乱序执行。通过向量重命名和寄存器堆中合法标志位解决了指令执行时数据冲突的问题，并将向量指令区分为访存指令和算术计算指令两类，分别定义了对应发射队列和功能处理单元。

（3）在模拟器中加入了异常检测代码，用断言对发生异常的情况进行限制。

（4）实现了寄存器分组，使得模拟器支持 LMUL 大于 1 的向量计算，并且完成了部分拓宽指令和缩宽指令的支持，添加了许多指令。

（5）添加了状态寄存器，并使得向量核能访问和修改向量状态寄存器，并在向量通道中实现了不可知策略，提高了的运算效率。

（6）搭建了处理器模拟器，并使用 riscv-gnu-toolchain 工具对指令功能和处理器模拟器的性能实现进行测试验证，并初步给出了分析结果，对于不同的向量应用而言，向量扩展集和相关架构对运行速度的提升有不同的表现，对于使用大向量的应用，使用向量核能够以向量操作占总操作的百分比正相关程度优化整体运行速度，向量化程度越高，总体运行加速越快，pathfinder 使用向量核最高能达到 5.93 倍的标量运行速度。

## 6.2 展望

本文在研究 RISC-V 向量扩展集处理器模拟器的过程中，还存在许多需要改进和提高的地方。

（1）仅使用 Minor CPU 模型扩展。向量核与标量核的设计为了尽可能的减少错误的发生，简化向量指令中标量源数据的判定，因此使用顺序模型使得标量核发送向量指令的时候可以较少考虑一些情况，因此标量指令是顺序计算的，后续可以对 Out of Order 模型进行考虑研究，使得标量核也支持乱序执行，这种情况下还需要对重命

名单元更新映射表增加判断，以能正确处理先写后写相关依赖。

（2）对于异常和错误支持仅停留在检测部分，没有完成对自陷（trap）发生，产生中断，处理异常然后恢复的代码支持，这种情况下需要更改重命名缓存，使得重命名储存旧目标内容值而非编号以便进行数据恢复。

（3）在向量通道中，功能单元的实现是按照数据带宽分别定义的，一个功能单元需要编写多种位宽的类定义，不利于模拟器的扩展位宽和添加功能实现等方面的运行，后续可以对这方面进行研究修改。

（4）指令延迟是定义在向量通道中，且对于每类指令都进行单独定义，这样不利于后续的指令添加和延迟信息维护，后续可以将这部分更改添加到指令操作类中处理，需要更加细致化地对向量指令进行分类。

（5）指令实现不完全，没有完全实现向量扩展集中的全部向量指令。

（6）模拟器校准通常分为两种办法，一是与 RTL 进行校准，二是与真实硬件进行校准，由于指令推出的时间较短，没有对模拟器的准确度进行校准配置。

# 参考文献


[1] Dennis D K, Priyam A, Virk S S, et al. Single cycle RISC-V micro architecture processor and its FPGA prototype. [C] // 2017 7th International Symposium on Embedded Computing and System Design (ISED). IEEE, 2017: 1-5.

[2] Herdt V, Große D, Pieper P, et al. RISC-V based virtual prototype: An extensible and configurable platform for the system-level[J]. Journal of Systems Architecture, 2020, 109: 101756.

[3] Abulila A, Hajj I E, Jung M, et al. Asynchronous Persistence with ASAP[J]. arXiv preprint arXiv: 2302.13394, 2023.

[4] Medina R,Kein J, Ansaloni G, et al.System-Level Exploration of In-Package Wireless Communication for Multi-Chiplet Platforms[C] // Proceedings of the 28th Asia and South Pacific Design Automation Conference. 2023: 561-566.

[5] Shim J E, Kang M, Han T H. NCDE: In-Network Caching for Directory Entries to Expedite Data Access in Tiled-chip Multiprocessors[J]. IEEE Access, 2023.

[6] 张晨曦, 王志英, 沈立等. 计算机系统结构教程[M]. 清华大学出版社, 2009.

[7] Binkert N , Beckmann B , Black G , et al. The gem5 simulator[J]. Acm Sigarch Computer Architecture News, 2011, 39(2):1-7.

[8] Russell R M. The CRAY-1 computer system[J]. Communications of the ACM, 1978, 21(1): 63-72.

[9] 平头哥发布全新 RISC-V 处理器[J].中国集成电路,2021,30(06):21.

[10] Xiang Shan-doc.UCAS&ICT,PCL.2021.https:/Igithub.com/Open Xiang Shan/XiangShan-doc

[11] Feng EH,Lu X,Du D,et al.Scalable memory protection in the PENGLAl enclave.In:Proc.of the 15thUSENIX Symp.on Operating Systems Design and lmplementation(OSDI 2021).2021.275-294.https://ipads.se.sjtu.edu.cn/zh/publications/Feng OSD121-preprint.pdf

[12] 刘畅,武延军,吴敬征等.RISC-V 指令集架构研究综述[J].软件学报,2021,32(12):3992-4024.DOI:10.13328/j.cnki.jos.006490.

[13] Alec Roelke and Mircea R. Stan. RISC5: Implementing the RISC-V ISA in gem5 [C/OL]. In Proceedings of Computer Architecture Research with RISC-V, Boston, Massachusetts USA, October 14, 2017 (CARRV'17), 7 pages. https://carrv.github.io/2017/papers/roelke-risc5-carrv2017.pdf

[14] Tuan Ta, Lin Cheng, and Christopher Batten. Simulating Multi-Core RISC-V Systems in gem5 [C/OL]. In Proceedings of Computer Architecture Research with RISC-V, Los Angeles, CA, USA, June 2, 2018 (CARRV'17). https://www.csl.cornell.edu/~cbaten/pdfs/ta-gem5-riscv-carrv2018.

pdf

[15] Peter Yuen Ho Hin, Xiongfei Liao, Jin Cui. Supporting RISC-V Full System Simulation in gem5 [C/OL]. In Proceedings of Computer Architecture Research with RISC-V (CARRV '21). ACM, New York, NY, USA, 6 pages. DOI:10. 1145. https://carrv.github.io/2021/papers/CARRV2021 paper_7_Yuen.pdf

[16] Alon Amid, Krste Asanovic, Allen Baum, et al. RISC-V "V" Vector Extension Version 1.0 [EB/OL]. [2022-09-03]. https://github.com/riscv/riscv-v-spec/releases/tag/v1.0

[17] Cristóbal Ramírez, César Hernandez, Oscar Palomar, et al. A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures [J]. ACM Transactions on Architecture and Code Optimization 17, 4, Article 38 (October 2020), 29 pages.

[18] Mukherjee S S, Adve S V, Austin T, et al. Performance simulation tools.[J]. Computer, 2022,35(2):38-39.

[19] 张乾龙,侯锐,杨思博等.体系结构模拟器在处理器设计过程中的作用[J].计算机研究与发展,2019,56(12):2702-2719.

[20] 刘晓燕. 一种 RISC 处理器指令集模拟器的设计与实现[D].国防科学技术大学,2014.

[21] Papaphilippou P, Kelly P H J, Luk W. Extending the RISC-V ISA for exploring advanced reconfigurable SIMD instructions[J]. arXiv preprint arXiv:2106.07456, 2021.

[22] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, et al. The gem5 Simulator: Version 20.0+ A new era for the open-source computer architecture simulator [J]. arXiv:2007.03152, 2020.

[23] Hennessy John L, Patterson D A . Computer Architecture: A Quantitative Approach 6th Edition [M]. US ,Morgan Kaufmann Publishers, 2019

[24] Editors Andrew Waterman and Krste Asanović. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 20191213[EB/OL]. [2022-02-12]. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

[25] Smith J E. Decoupled access/execute computer architectures. [J]. ACM SIGARCH Computer Architecture News, 1982, 10(3): 112-119.

[26] Andrew Bardsley. Minor CPU Model. [EB/OL]. [2023-01-26] https://www.gem5.org/documentation/general_docs/cpu_models/minor_cpu

[27] Steven G , Christianson B , Collins R , et al. A superscalar architecture to exploit instruction level parallelism[J]. Microprocessors & Microsystems, 1997, 20(7):391-400.

[28] 姚永斌. 超标量处理器设计[M]. 北京, 清华大学出版社, 2014.

[29] 郑纬民, 汤志忠. 计算机系统结构(第 2 版)[M]. 清华大学出版社, 1998.

[30] 赵紫微,涂航,刘芹等.针对 gem5 指令集实现及其功能测试的自动代码生成[J/OL].计算机研究与发展:1-13[2023-03-01]. http://kns.cnki.net/kcms/detail/11.1777.TP.20220818.1449.002.

html.

[31] Rogers S, Slycord J, Raheja R, et al. Scalable LLVM-based accelerator modeling in gem5[J]. IEEE Computer Architecture Letters, 2019, 18(1): 18-21.

[32] SCons Foundation. SCons: A software construction tool [EB/OL]. [2022-01-14]. https://scons.org/

[33] Poorhosseini M, Nebel W, Grüttner K. A compiler comparison in the risc-v ecosystem. [C]//2020 International Conference on Omni-layer Intelligent Systems (COINS). IEEE, 2020: 1-6.

[34] Kito-cheng et al. riscv-collab/Riscv-gnu-toolchain [OL]. [2022-02-02] https://github.com/riscv-collab/riscv-gnu-toolchain

[35] 严迎建,杨志峰,任方.面向专用指令集处理器设计的软硬件协同验证[J].计算机工程, 2010, 36(06):241-243.

[36] Louis M S, Azad Z, Delshadtehrani L, et al. Towards deep learning using tensorflow lite on risc-v. [C] Third Workshop on Computer Architecture Research with RISC-V (CARRV). 2019, 1: 6.

[37] M. A. Goodrum, M. J. Trotter, A. Aksel, et al. Parallelization of Particle Filter Algorithms [C]. In Proceedings of the 3rd Workshop on Emerging Applications and Many-core Architecture (EAMA), in conjunction with the IEEE/ACM International Symposium on Computer Architecture (ISCA), June 2010.

[38] S. Che, M. Boyer, J. Meng, et al."Rodinia: A Benchmark Suite for Heterogeneous Computing" [C]. IEEE International Symposium on Workload Characterization, Oct 2009.

[39] J. Meng and K. Skadron. "Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs." [C]. In Proceedings of the 23rd Annual ACM International Conference on Supercomputing (ICS), June 2009.

[40] L.G. Szafaryn, K. Skadron and J. Saucerman. "Experiences Accelerating MATLAB Systems Biology Applications."[C]. In Workshop on Biomedicine in Computing (BiC) at the International Symposium on Computer Architecture (ISCA), June 2009.

[41] M. Boyer, D. Tarjan, S. T. Acton, et al. "Accelerating Leukocyte Tracking using CUDA: A Case Study in Leveraging Manycore Coprocessors."[C]. In Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), May 2009.

[42] S. Che, M. Boyer, J. Meng, et al. "A Performance Study of General Purpose Applications on Graphics Processors using CUDA". [C] Journal of Parallel and Distributed Computing, Elsevier, June 2008


西安电子科技大学
XIDIAN UNIVERSITY

地址：西安市太白南路2号
邮编：710071
网址：www.xidian.edu.cn