计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
63(7): 1787−1802, 2026

面向 RISC-V 高性能乱序处理器的指令扩展框架

刘伯然　张承译　胡思鹏　赵　地　包云岗　孙凝晖
（处理器芯片全国重点实验室（中国科学院计算技术研究所）　北京　100190）
（中国科学院大学计算机科学与技术学院　北京　101408）
（liuboran_00@hotmail.co.jp）

Instruction Extension Framework for High-Performance Out-of-Order RISC-V Processors

Liu Boran, Zhang Chengyi, Hu Sipeng, Zhao Di, Bao Yungang, and Sun Ninghui
(*State Key Lab of Processors* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)
(*School of Computer Science and Technology*, *University of Chinese Academy of Sciences*, *Beijing* 101408)

Abstract　With the development of the RISC-V open-source ecosystem, driving domain-specific architectures (DSAs) with custom extension instructions based on RISC-V has become a widely recognized design approach. However, under traditional hardware description languages and design paradigms, integrating DSAs with processors and completing instruction design require significant manpower and time costs to understand processor design details and modify them. Currently, some work supports developers in accessing DSAs by implementing fixed data paths and interfaces in sequential processors. Yet, for high-performance out-of-order processors, there is no comprehensive solution available. We summarize the key issues that need to be addressed through practice when driving and integrating DSAs with out-of-order processors using extension instructions. We design a hardware programming framework for instruction extension targeted at high-performance out-of-order RISC-V processors, assisting developers in integrating DSAs and completing instruction extensions on out-of-order processors. The framework is implemented in Chisel, applying object-oriented and functional programming features to flexibly describe hardware and assist in completing signal connections, thereby saving developers' effort. Our evaluation shows that with the help of this framework, developers can save up to 1 000 lines of Chisel code effort, equivalent to over 10 000 lines of Verilog code effort. Moreover, compared with the traditional manual modification of pipelines, the framework can save developers over 9 000 lines of code reading work. The evaluation results indicate that the framework significantly reduces the manpower and time costs for developers and lowers the development threshold.

Key words　high-performance processor design; domain-specific architecture (DSA); instruction extension interface; RISC-V; out-of-order pipeline; hardware programming framework

摘　要　随着 RISC-V 生态的发展，基于 RISC-V 的自定义扩展指令来驱动领域专用结构（domain-specific architectures, DSA）成为了一种广受关注的设计方式。但在传统的硬件描述语言和设计范式下，因为需要了解处理器设计细节并对其进行修改，将 DSA 与处理器集成并完成指令设计需要耗费较高的人力和时间成本。目前，有部分工作通过在顺序处理器中实现固定的数据通路和接口来支持开发者接入 DSA，但对高性能乱序处理器尚无完善的解决方案。通过实践归纳出乱序处理器上以扩展指令驱动并集成 DSA

收稿日期：2025-06-17；**修回日期**：2026-03-04
基金项目：国家自然科学基金项目（62090020, 62090022）
This work was supported by the National Natural Science Foundation of China (62090020, 62090022).

时需要解决的关键问题。设计了一个面向 RISC-V 高性能乱序处理器指令扩展的硬件编程框架，来辅助开发者在乱序处理器上集成 DSA 并完成指令扩展。该框架基于 Chisel 实现，应用面向对象和函数式编程的特性灵活地描述硬件并辅助完成信号连接，帮助开发者节约了工作量。经过实验评估，在该框架的辅助下，开发者最多可以节约 1 000 行以上的 Chisel 代码工作量，等效的 Verilog 代码工作量达到了 10 000 行以上。并且，相比于传统人工修改流水线的方式，该框架可以为开发者省去 9 000 行以上的代码阅读工作。评估结果表明，该框架可以大幅降低开发者的人力和时间成本，并降低开发门槛。

关键词 高性能处理器设计；领域专用结构；指令扩展接口；RISC-V；乱序调度流水线；硬件编程框架

中图法分类号 TP391

DOI: 10.7544/issn1000-1239.202550477 **CSTR:** 32373.14.issn1000-1239.202550477

近年来，RISC-V 作为一个完全开源的指令集架构备受业界关注。随着其开源社区不断发展，现有的开源架构越来越先进，设计也变得愈加复杂。在社区中，业已开源了基于乱序流水线技术的高性能处理器设计，其中包括加州大学伯克利分校开源的 Boom[1] 和中国科学院计算技术研究所开源的香山处理器[2]，以及工业界的代表性作品——平头哥的 XuanTie 系列芯片[3]。

与此同时，随着互补金属氧化物半导体（complementary metal oxide semiconductor, CMOS）尺寸接近物理极限，为维持性能增长、提高能源效率和改善计算硬件资源利用率，在后摩尔时代，领域专用结构（domain specific architecture, DSA）设计的重要性也日益凸显[4]。在可预见的未来，如何更好地支持 RISC-V 生态和领域专用结构的结合成为整个社区必将面临的问题。

目前，随着 RISC-V 生态的迭代和发展，一些开发者已经尝试将开源的 RISC-V 架构设计与领域专用架构相融合，追求在特定的应用场景下以尽可能低的成本获得更佳的性能表现[5-6]。

在传统设计范式中，DSA 通常以外设的形式存在，并主要依赖于总线与处理器进行交互。然而，随着 RISC-V 指令集和架构设计的开源化，开发者现在有机会通过设计扩展指令的方式来驱动 DSA 进行计算。这种设计方法能够显著减少处理器与 DSA 之间的交互延迟，同时缩短数据传输路径。另一方面，由于对 DSA 的控制是通过指令完成的，这种方法显著提高了软件编程的灵活性，并为在 DSA 负载上应用编译优化方法提供了可能性。

然而，由于大部分的领域专用架构设计中多采用顺序执行来设计其内部的体系结构状态，同时也没有现代超标量处理器中被普遍应用的乱序调度和推测执行等机制，因此想要在开源的 RISC-V 高性能处理器上集成并以扩展指令驱动领域专用结构，就需要投入额外的工作——控制扩展指令的调度和分发结构来驱动领域专用结构能够与处理器协同并正确地完成计算。但是，众所周知，现代高性能处理器的设计与实现非常庞大且复杂，通过人工的方式来对高性能处理器和领域专用结构进行改写和集成不仅需要较大的工作量，同时也对设计者的知识储备提出了更高的要求，形成了较高的技术门槛。

正因此，目前的大部分基于领域专用结构设计的自定义扩展工作仍然更多地选择基于在如 Rocket-Chip[7], CVA6[8], PULPino[9] 这样的顺序处理器进行设计和集成。本文统计了 2016—2024 年期间基于 RISC-V 进行设计与实现的指令扩展工作，可以看到业界对该领域投入和关注是在持续增长的，如图 1 所示。然而经过调研，发现仅有 8% 的工作是基于乱序执行的处理器实现了领域专用结构的指令扩展和集成。这从侧面说明了该类工作在开发和设计上存在的门槛。

本文在一些早期的探索和实践的基础上，将基于高性能乱序处理器实现了领域专用结构的指令扩

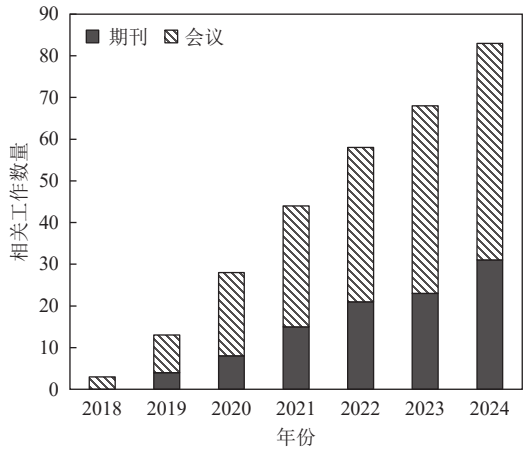


Fig. 1 Number of RISC-V custom instruction extension-related works

图 1 RISC-V 自定义指令扩展相关工作数量

展和集成开发工作流程划分成了如图 2 所示的 4 个主要阶段：1)理解处理器微结构的设计，在处理器中搭建提供的数据流信号和控制流信号接口；2)基于这些交互信号来进行 DSA 的硬件实现；3)对于与乱序调度模型不兼容的 DSA，需要设计额外的指令流调度机制来恢复部分上下文中的依赖信息；4)修改处理器设计，确保接口设计和调度机制正确工作，并完成组件连接，组成完整的系统。

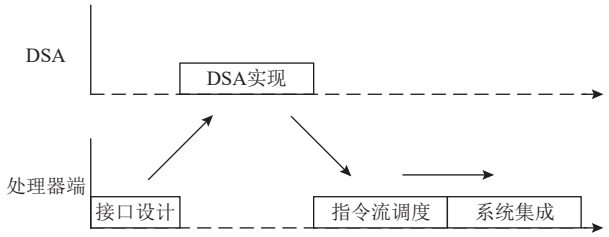


Fig. 2 General workflow for integrating and driving DSAs with extension instructions on general-purpose processors

图 2 在通用处理器上以扩展指令集成并驱动 DSA 的一般工作流程

在各类工作中，虽然在 DSA 的设计各不相同，但在处理器一侧进行的 3 个阶段中的工作是大多数开发者所面临的共性问题。将这 3 个阶段的工作尽可能实现自动化，一方面可以大幅缩减开发者的工作量，另一方面可以降低对开发者的知识储备要求。

进一步分析可知，基于高性能乱序处理器实现 DSA 指令扩展和集成需要解决以下 3 个关键问题：1)如何灵活快速地定义接口并在处理器中构造数据通路；2)超标量处理器的乱序指令流进入 DSA 后，如何维护各种指令间的依赖，避免指令冒险，使指令按照约定的方式执行；3)如何尽可能地使用编程工具来代替人工对处理器进行修改，连接并传递控制流和数据流信号到接口界面，并嵌入设计好的 DSA 模块，最终形成系统。

问题 1：经过长时间的技术迭代，在通用处理器中，特别是超标量乱序处理器的微结构设计中，其流水线的控制流信号是非常复杂的。除去指令译码之后得到的控制信号外及寄存器读取的数据信号，会伴生大量控制信号用于提升指令吞吐的调度机制，例如寄存器重命名、保留站指令唤醒、指令重排序等。因此，本工作需要从这些信号中鉴别出 DSA 需要哪些信号。相应地，还需要考虑当 DSA 需要的一些控制信号和数据信号超出了流水线内部原有信号的范畴应该如何处理。同时，面对各种不同规格的 DSA，如何使处理器端的接口设计能够最大程度地满足各种信号需求。

问题 2：如问题 1 所述，在处理器设计迭代的历史进程中，为了提升处理器性能，不断提升指令吞吐的过程中遇到了三大冒险问题，即结构冒险、数据冒险和控制冒险。在现代处理器中，寄存器重命名技术消除了假数据冒险，乱序多发射技术用于避免结构冒险和真数据冒险，重排序缓存结合流水线冲刷确保了控制冒险不会造成错误的执行结果。然而，在 DSA 设计中，其核心优势正是通过在特定应用场景下，根据其负载的局部性特点，通过软硬件协同设计，放弃了上述硬件调度机制，来释放额外的晶体管资源用于计算，以达成更高的计算效率[10]。因此，处理器中经过乱序调度的指令流不能直接传入 DSA 执行。它们必须被重新排序，并补充必要的上下文依赖，才能确保 DSA 的行为不存在冒险。

问题 3：在传统硬件设计流程下，想要在通用处理器的流水线上嵌入一个 DSA 模块，通常需要工程师阅读代码理解处理器的流水线结构，并手工构建数据通路和控制通路，将 DSA 接入到处理器中。而在当前的硬件设计领域，Chisel[11] 的出现标志着一个重要的转变。Chisel 以其更高的抽象描述能力结合 Scala 的函数式编程特性，使得开发者有机会探索更高层次和更抽象的硬件设计方法，从而代替工程师将 DSA 模块嵌入流水线中。

本文基于 RISC-V 高性能乱序处理器香山[2]，实现了一个用于集成指令扩展的硬件编程框架。

本文的贡献包括：

1)设计并定义了一组指令扩展接口，并且支持用户对部分的信号接口进行自定义。

2)基于面向对象的设计方法，抽象出了一套帮助用户集成指令扩展的硬件编程框架。用户仅需提供必要的信息和 DSA 实现，即可完成其在通用处理器上的集成。

3)为了解决 DSA 与乱序指令调度不兼容的问题，设计了专用的调度模块，来维护指令流的上下文信息，以保证 DSA 行为是正确的。

本文选取了 3 个设计实例以完成验证工作，其中包括 2 个第三方开发者设计的开源 DSA 实例及 1 个本团队独立开发的脉动阵列单元。经过测试和评估，在不同的框架配置下，本编程框架显著降低了工程师在集成过程中所需编写的代码量，例如在多种配置场景中，该框架能够帮助开发者节约 260~1 100 行的 Chisel 代码，以对应的 Verilog 代码来计算，节省代码量达到了 1 600~12 000 行。此外，本文选取了流水线

源代码阅读量作为衡量开发者负担的指标，在不同的配置选项下，该框架能助力开发者减少最多 4 800~9 300 行代码的阅读量，从而显著降低了开发的门槛。最后，本工作基于此框架集成了数个 DSA 设计实例，并对开发者在框架中需要填写的代码量进行了统计，以此证明框架对于不同设计的适用性。

1 问题背景及相关工作

1.1 处理器-DSA 指令接口

处理器与 DSA 的交互方法有多种形式，是将 DSA 作为处理器的设备，根据各类不同的协议来对特定地址空间的内容进行读写，以实现二者的信息交换和协同计算。共享存储交互、DMA 交互和中断机制，都是上述模式的一些典型实例。

近年来，基于 RISC-V 的指令集和处理器设计的开源化，以自定义的扩展指令来驱动 DSA 部件成为了一种新的设计模式。而在此之前，大多数商用处理设计细节和源代码并不公开，因此，除非经过授权，第三方开发者想要修改处理器以实现扩展指令几乎是无法做到的。

RISC-V 在设计指令集时预留了部分指令编码空间，以鼓励用户实现自定义指令。同时，在社区中，基于顺序通用处理器 Rocket-Chip 实现的 RoCC 指令扩展接口[7] 可以被用于集成和驱动 DSA。RoCC 成为此类接口设计工作的首个实例。随后，在学术界的几个开源处理器设计上也出现了一些作用相似接口设计[8-9]，这些接口设计根据各自的处理器设计特点，在 RoCC 的基础上对接口信号进行了裁撤或扩充。

但可以观察到，这些接口设计有着共同的局限性，其接口规格与处理器的硬件实现耦合紧密，一经确定便难以修改。对于 DSA 设计者，如需根据自身需求修改接口规格，就需要耗费人力和时间成本来理解处理器设计，并付出额外的工作量。如若不然，会导致 2 类问题：1）当处理器中提供的某些接口信号没有被使用，那么为实现这些接口而设计的信号通路和逻辑便会被浪费；2）当 DSA 需要的信号在接口中没有被满足，那么 DSA 工程师可能仍然要付出一些额外的工作来修改 DSA 设计以削足适履。

当面对高性能乱序处理器时，情况会变得更加复杂，对开发者的要求也会更加苛刻。而现有的工作中，尚未发现完善且针对性的解决方案。

因此，为了更加高效地利用硬件资源，需要在不同的场景下，令接口的设计有一定的自定义空间，并在此过程中避免引入额外的代码工作量。

1.2 自定义扩展指令的调度问题

在现代乱序处理器设计中，指令流中的依赖关系都在流水线的前置阶段被层层剥离，并交给专门的指令调度机制处理[12]。因此，乱序执行过程中产生的冒险行为对执行单元完全透明。换言之，执行单元通常无法获知指令流上下文中的依赖信息。

然而，DSA 之所以能够以更少的代价获得更高的计算能力，恰是针对特定应用场景下的负载特点，在硬件结构上进行定制设计，精简了计算过程中的调度机制，将节省的晶体管资源用于计算部件，以获取更高的计算效率。因此，在乱序流水线中，如何将指令流中必要的依赖关系正确地传递给 DSA，是 DSA 能够正确且高效执行计算的基础。

截至目前，在一些实现了指令扩展接口的处理器中，都根据自身特点采取了不同的措施来确保 DSA 获得的指令流中包含正确的指令依赖信息。

在 Rocket-Chip 中，扩展指令按照单周期处理器的执行方式执行，以确保传入的指令流的正确性，这种方法简单直接，但是略显低效。而在 CVA6 中，其为 DSA 前置了一个命名为 Sequencer[13] 的结构来缓存指令流，以确保 DSA 视角下指令流的正确性。在 BOOM 中，类似的结构被称为 Shim，由于到达这里的指令已经经过了乱序调度，因此该结构一定程度上维护了到达指令应有的顺序[14]。相似地，在开源的 Xuantie C910 中，引入了名为自定义指令管理单元的模块，专用于维护扩展指令存在的上下文关系。为方便叙述，本文将负责上述功能的逻辑结构称为“自定义指令调度单元”。

在这些工作中，自定义指令调度单元与处理器本身的架构设计高度相关，且与指令扩展接口耦合紧密。因此，指令扩展接口的灵活性受限，根据用户的需求来进行调整以增加或者修剪必要的控制信号的代价较大。

1.3 基于高级硬件描述语言的集成方法

截至当前，基于部分高级语言特性的硬件描述语言已经不在少数，其中比较有代表性的有 BlueSpec[15]，SystemC[16]，Spatial[17]，SystemVerilog[18] 等。

Chisel[11] 作为新兴的硬件描述语言，受到业界的广泛关注。Chisel 内嵌于 Scala，相较于上述其他的语言，抽象层次更高，可引入面向对象、函数式编程等高级编程语言的特性。因此，Chisel 获得了描述生成硬件结构的步骤和规则的能力。但是这样的能力非常依赖开发者对这些高级语言特性的理解和应用。

Twine[19] 基于 Chisel 设计了新的模块抽象，丰富了针对模块调度行为和接口的硬件抽象原语，来辅助小规模异构组件的设计。V++[20] 归纳了一些信号的组织模式，并以多写单读模型来辅助用户生成和连接数据通路。SpinalHDL[21] 提供了类似于 Chisel 中 DecoupledIO 的组件接口并赋予其一定的语义，来辅助模块之间的连接。BaseJump STL[22] 提供了带有模板的标准硬件组件库，但是缺乏组件之间的系统级自动化构造流程。这些工作的共同点在于：针对硬件的设计规模较小，多应用于流式计算单元的设计之间的构造和连接。

然而，针对高性能乱序处理器这样的复杂紧耦合结构设计中如何辅助工程师进行自动化的集成，上述工作并未给出可用的解决方案。同时，在目前常见的基于 Chisel 设计的各类处理器中，其描述风格更接近传统的“模块定义+硬件连线”的抽象形式。在该抽象形式下，几乎所有的连线和大部分的模块功能的描述仍然完全依赖人工实现。

因此，本文通过工程实践，归纳出在硬件集成过程中需要完成的工作，并探索如何尽可能利用高级语言特性降低系统集成过程中所要求的前置知识和工作量。

2 观察与设计关键点

本节将主要介绍本工作在实践中的观察和设计的关键点。

2.1 可配置指令扩展接口设计

如 1.1 节所述，定义处理器和 DSA 之间的接口设计规范，首先要了解处理器一端可提供的控制流和数据流信号，明确 DSA 所需要的控制流和数据流信号。在对图 1 中涉及的工作进行了调研和归纳后，通过观察发现，DSA 从处理器获取的信号可以被划分为 3 个部分：指令编码、数据信号和控制信号。

指令编码和数据信号几乎在所有的设计中是 DSA 所必需的。这其实很容易理解，以指令驱动的 DSA 必须获得指令编码才可以得知需要执行什么样的计算以及对哪些数据进行操作。而数据信号通常是指令需要读取的寄存器的值，以及访存信号通路。而控制信号的定义则需要进一步进行分析。观察发现，处理器和 DSA 之间的信号交互需要相应的握手信号，这部分在不同的设计之间是一致的。例如有些实现中包含了中断和异常信号，也有些 DSA 中需要一些自定义的控制信号来控制 DSA 内部的计算功能及读写行为。这部分信号可以由处理器生成，亦可由二次译码获得。

指令编码信号、数据信号和控制信号，在处理器中都构造了相应的通路，这与普通功能单元的输入输出信号高度一致。但自定义控制信号的需求对不同 DSA 而言各不相同。因此，本工作选择将这部分信号保留为一组可选的自定义信号束，其内容交由用户来进行定义。同时通过处理器的译码逻辑来根据指令生成对应的控制信号内容。

故本工作将接口设计为 2 个主要部分：一部分是通用信号束，这部分被定义为默认接口，这些接口在任何情况下都需要被实现。另一部分则是自定义信号束，开发者可以根据自身需要来定义这部分接口的内容。

接口设计的实现有 2 个基本设计要素：1）译码逻辑的扩增；2）数据通路的构建。本文将在 2.3 节和第 3 节中详细说明如何充分利用面向对象编程的特点，通过“继承+重构”的方式，形成编程范式，以完成设计要素的具体实现。

2.2 自定义指令调度单元设计

现代处理器之所以能够实现指令吞吐的大幅度提升，核心的思路是让指令尽可能并行执行，而非刻板地按照指令流指定的顺序执行。但逻辑层面，指令的上下文之间仍存在 3 类依赖关系：数据依赖、控制依赖和结构依赖。因此，若不对这些依赖进行处理，贸然地并行执行指令，会导致对应的 3 类冒险：数据冒险、控制冒险和结构冒险。现代处理器的乱序调度机制恰是为了规避或消除这些冒险行为，令处理器既能实现更高指令吞吐，也能保证其行为符合程序员预期。

乱序处理器中的各类调度机制在不同阶段将指令流中的依赖信息进行了剥离并交由对应的调度部件来进行维护，以确保指令流到达执行阶段时，指令可以被立即执行而不存在冒险。

下面将举例来说明本文的观察。图 3 是一段包含了 1 条分支指令和 4 条计算指令的指令流。图 3 中展示了这段指令流由取指令单元逐级传递至某个执行加法的功能单元中计算并最终由重排序缓存提交的过程。在取指令阶段，分支预测单元给出预测的跳转地址，并投机执行后续的指令，原本的控制依赖交给分支预测单元进行记录和维护。在重命名阶段，寄存器重命名单元用物理寄存器索引替换逻辑寄存器索引，原有的写后写和读后写这 2 种假数据依赖信息由重命名表进行记录和维护。在分派阶段，

写后读的真数据依赖转交保留站维护。在发射阶段,结构依赖也由保留站记录并维护,并将计算功能相同的指令发射到多个功能相同的执行单元上进行计算。最后,某个加法计算单元最终看到并进行计算的指令流片段仅有 1 条单独的指令。在提交阶段,完整的指令流与其内部的依赖关系会在由重排序缓存和前述的调度单元协同作用下被恢复并提交,同时完成对体系结构状态的更新。

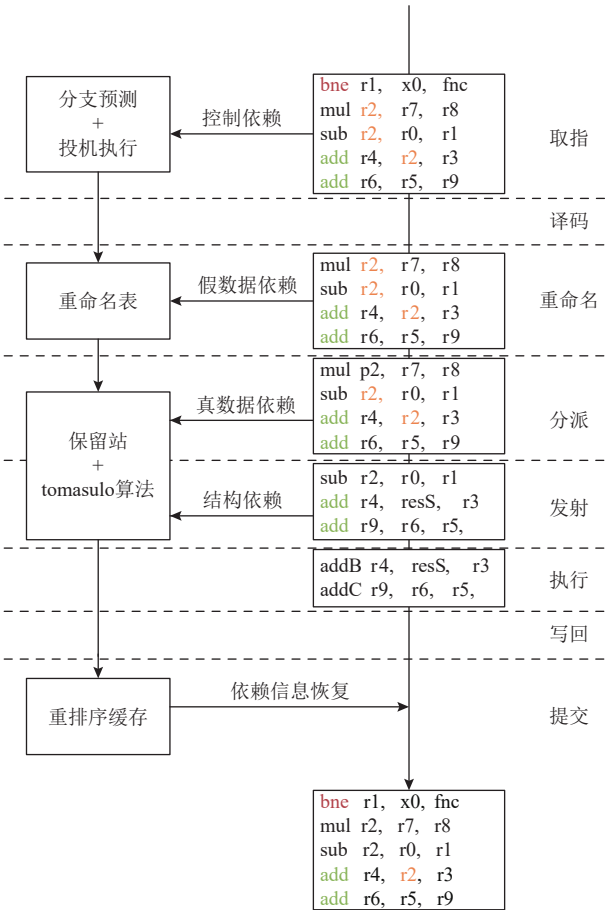


Fig. 3 Propagation of instruction stream context dependencies in out-of-order pipelines

图 3 指令流上下文的依赖关系在乱序流水线中的传递

然而对 DSA 来说,其内部逻辑通常不包含上述的调度机制,因此,需要在指令发送至 DSA 之前对指令流中的部分依赖信息进行必要的恢复,否则就会发生冒险。

在发送给 DSA 的指令流中,因为有可能引入存储结构,会产生的数据冒险有 2 种:读写通用寄存器产生的数据冒险与读写 DSA 内部寄存器所产生的数据冒险。DSA 部件的控制冒险是由普通的分支指令产生,因此需要与主流水线协同来保证指令在投机执行阶段不会产生写存储行为。DSA 部件的结构冒险源于其自身的硬件资源不足。接下来,本文对这 4 类冒险逐个分析:

1)结构冒险。结构依赖源于 DSA 内部计算资源不足,因此需要获知并记录 DSA 计算部件的繁忙状态来控制指令发射,以避免结构冒险。

2)控制冒险。控制冒险产生于取指令阶段产生的投机执行。而在取指令阶段就截获分支指令流以维护控制依赖是不现实的,因为必须经过译码后才能获知哪些指令是需要被发送到 DSA 中的指令。因此本工作中,分支预测得到的分支会被投机执行,在该分支上的指令被提交前,会核对分支预测的结果,当发现分支预测错误时,会进行流水线冲刷。对于已经执行并得到了计算结果的指令,我们暂时阻塞其写回行为,与重排序缓存进行通信,等待确认指令提交之后再完成写存储的操作,也确保了本工作的架构正确性不受分支预测算法的影响。

3)通用寄存器的数据冒险。这部分数据冒险源于扩展指令需要读写通用寄存器时可以复用流水线的调度机制,读寄存器行为由保留站进行维护,写寄存器的行为由写回逻辑和重排序缓存进行维护。因此,在指令进入 DSA 前需要与保留站交互以监听所需的操作数的值。当操作数就绪时,需记录操作数的值并更新操作数的就绪状态。写回通用寄存器时 DSA 将写回值和当前分配到的物理寄存器索引传递给写回逻辑进行后续的处理。

4)DSA 寄存器的数据冒险。由于一般的 DSA 中没有重命名机制,因此写后写、读后写和写后读 3 种冒险都需要被维护。因此,指令在执行前需要确认存在依赖的前序指令的提交和写回状态,以确保寄存器中的值有效。

2.3 基于面向对象编程的集成方法

乱序处理器流水线的结构复杂且组件之间的耦合度较高,对所有的流水线组件统一构造硬件原语并进行自动化的模块间连接是困难的。但在本工作的实践中发现,虽然流水线本身的组件繁多且结构复杂,但是将 DSA 集成到处理器的工作流程中涉及的组件和信号通路是有限的,若仅对这部分电路进行一定程度的抽象,那么自动化地完成模块生成和通路构建抽象是可行的。

而在面向对象的描述范式下,处理器内部的模块和信号通路分别被描述为类(class)的形式。因此,自然也可以通过面向对象中常用的继承、引用和重写等方式,来实现对已有的硬件结构的修改和扩展。因此,本工作尝试用面向对象体系结构设计方法[23]

来辅助开发者实现 DSA 的集成。

经过实践后，本文将 DSA 集成这一任务进一步地划分为 4 个部分：1)更新原有的译码逻辑以支持自定义的 RISC-V 指令；2)打通数据通路以获取所需的流水线信号；3)引入自定义指令调度单元实现扩展指令接口；4)接入 DSA 模块。下面将对 4 个部分任务进行详细介绍。

1)扩展译码逻辑。面向对象的描述范式下，译码器可以被构造为对象(object)并在其中定义了生成译码器的方法(method)。用户只需完成译码表的定义，调用该对象中对应的方法，即可得到一个符合该译码表规则的译码器。而译码表可以被定义为一个特定数据类型。每个元素是一个二元组，包含指令编码及其对应的译码信号。因此，要扩展原有译码表，本工作只需为新指令及其对应的译码信号构造一个新的译码表，并通过数组运算符将其与原有的译码表进行合并，再调用上述方法，即可生成扩展后的译码器。

2)构造数据通路。指令及其伴生的数据、控制流信号通常被称为微指令。在面向对象的描述范式下，被包裹在同一个 Bundle 对象的微指令被开发者在不同位置引用，并进行连接和信号传递。因此，须对传递微指令信号的 Bundle 类进行继承，并引入必要的控制信号束，最后重构原生类，即可方便地完成对数据通路的拓展。

3)引入自定义指令调度单元。本工作通过继承普通的功能单元模块，并在其中添加自定义指令调度单元，以确保能够获取标准的功能单元所需的微指令及操作数数据，并构造标准的输出路径。同时须对标准功能单元的输入输出进行扩展，以接收来自指令流的信号。同时，在这个指令调度单元的内侧，需要留出定义好的指令扩展接口，用于连接设计好的 DSA 模块。

4)接入 DSA 模块。由于处理器和 DSA 之间的接口规格是统一的，因此从处理器视角出发，所有 DSA 都是一个接口完全相同的黑盒，不同的是其内部的行为。因此，本工作定义了一个抽象类，并向其添加输入输出端，并预先完成该模块与自定义指令调度单元的连接和实例化。DSA 设计者可以在直接继承该抽象类，在内部调用和实例化 DSA 并通过接口信号与处理器完成交互。

2.4 小 结

本节中总结了实践和观察中获得的一些设计关键点，针对在高性能乱序处理器流水线中集成 DSA 时面对的 3 个关键问题做出了应对。

首先，本框架利用新一代硬件描述语言的面向对象特性，设计了可配置的信号通路和扩展接口，极大地简化了对既有流水线信号的修改和扩增工作。接着，构造了专用于加速模块的指令调度单元，以解决加速模块本身无法处理来自超标量处理器流水线的乱序指令流的问题。最后，面向对象描述范式完成了对 RISC-V 处理器指令扩展集成过程中所需模块及关键参数的描述，从而构造出完整的集成系统。

3 扩展指令编程框架的实现

基于第 2 节的观察和设计关键点，本工作基于新兴硬件描述语言的面向对象特性，以面向对象的硬件描述方法实现了一个在乱序处理器上辅助开发者快速集成指令扩展的硬件开发框架。该框架实现了指令扩展接口的定义及数据通路的构建，并允许开发者配置自定义控制信号及其生成规则。同时，该框架代替开发者定义了 DSA 功能单元的抽象类，并在流水线中完成了相应的连线和模块的嵌入，开发者仅需将 DSA 实例填入其中即可完成集成。

该框架主要包含 4 个部分：1)自定义指令译码规则；2)信号通路定义；3)DSA 功能单元实例；4)参数框架。整个框架的代码的结构如图 4 所示，共分为 2 个层次，分别描述 DSA 端和处理器端所需要的抽象结构。其中，ExtensionFrame 主要作用是继承整个框架所需的原生处理器模块，引用 DSAFrame 中给出的实例对原生模块进行重构，最终完成对原生模块的扩展和替换。而 DSAFrame 是面向 DSA 设计者的部分，用于设计者定义 DSA 所需要的额外的译码规则、控制信号及 DSA 模块的实现。

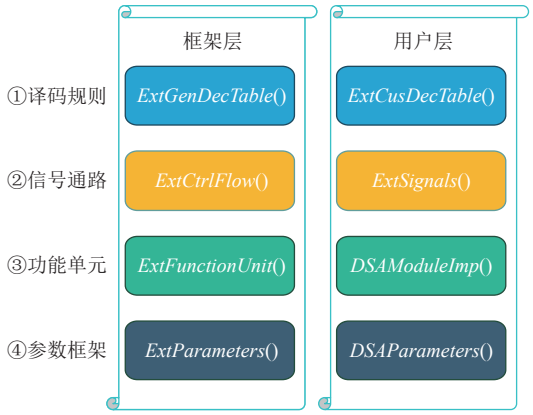


Fig. 4 Organizational structure of framework code

图 4 框架代码的组织结构

3.1 扩展指令译码逻辑生成

在处理器设计中，译码器的实现可以分为 3 个步骤。首先，需要定义出指令的译码表，罗列出需要实现的指令和对应的指令译码信号来作为译码器的输入和输出信号；接着，需要根据这些输入输出信号画出卡诺图，并得出输入输出信号之间的布尔映射的最简单表达形式；最后，根据该最简单表达形式生成对应的逻辑门电路，即最终的译码器结构。

在这个过程中，卡诺图的生成、表达式的简化和映射至逻辑门电路这 3 个环节与处理器设计或者译码表内容都是无关的。因此，在面向对象的设计范式下，以上过程可以被封装为方法供开发者调用。开发者只需提供格式化的译码表定义，即可通过该方法生成相应的译码部件。如此一来，开发者引入扩展指令时，只需定义扩展指令的译码表，并对原本的译码表完成扩充，将扩充后的译码表作为输入，调用译码器的生成方法，就可以得到包含扩展指令的处理器译码逻辑。

扩展指令的译码表定义分为 2 部分：1)扩展指令-通用译码信号的映射；2)扩展指令-扩展信号的映射。其完整的生成流程如图 5 所示。

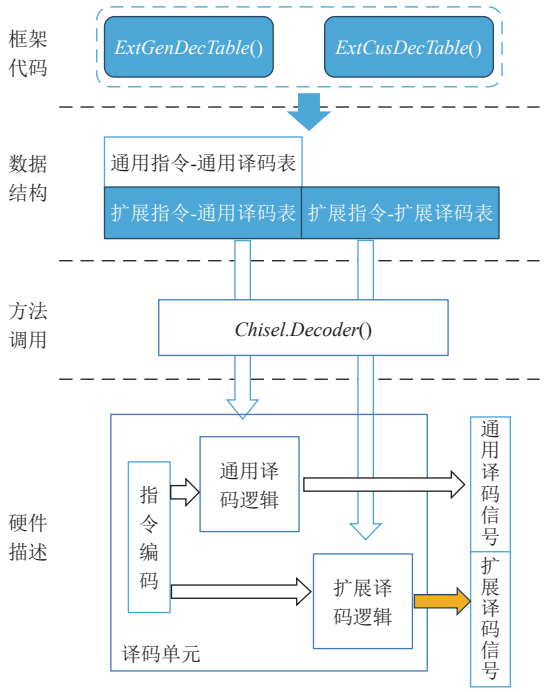


Fig. 5 Generation process of decode logic in the framework

图 5 框架中译码逻辑的生成流程

为了方便扩展，在面向对象的描述方法下指令译码表可以被定义为数组类型，每个数组元素是一个二元组，包含了指令编码和译码信号。同样地，扩展指令的译码表也可以定义成这样的数组，并通过数组间的运算与原有的译码表完成合并形成一张新的译码表。因此，扩展至指令-通用译码信号的映射可以通过这样的方式加入到原有的译码表中。而对于扩展指令-扩展译码信号的映射，本工作定义了一张新的译码表，并单独调用译码器生成方法，来生成一个独立的小型译码逻辑，从而实现扩展译码信号能够与通用译码信号同步生成。最后，通用译码逻辑的连线不需要任何的改动，但是扩展译码逻辑需要完成与扩展指令通路之间的连线。本工作代替开发者完成了这部分工作，因此，连线过程对开发者而言是完全透明的。

在以上流程中，框架为开发者完成大部分的扩展和连线工作，开发者仅需按照格式在框架中提供 2 组译码表中的映射关系即可完成对整个译码逻辑的修改和扩展。

3.2 信号通路定义

2.1 节和 2.3 节提到，信号通路的构造重点工作是定义出新的接口信号，并对原有的微指令中的控制信号的通路进行扩展，将所需的信号在流水线中向后传递。在通用处理器的流水线中，所有由译码阶段生成的控制信号，都可以被包含在同一个类中进行定义和实例化。微指令信号作为贯穿流水线始终的信号束，需要将上述控制信号类做为其父类进行继承。

本工作在 DSAFrame 中构造了一个抽象类，允许开发者在其中自由地定义所需的控制信号。这部分控制信号旨在将扩展译码逻辑产生的控制信号在流水线中向后传递，因此，这里的信号定义和排列顺序需要与扩展译码表实现一一对应。

在 ExtensionFrame 中，本工作构造一个新的控制信号类，并命名为 ExtCtrlFlow 类。ExtCtrlFlow 类会继承处理器原有的控制信号类，并引用 DSAFrame 中开发者自定义的控制信号定义，完成对已有的信号通路定义的扩展。最后，通过修改微指令类的继承路径，令其继承 ExtCtrlFlow 类，从而实现对流水线信号通路的修改和扩充。本节叙述的继承关系变化如图 6 所示。

3.3 DSA 功能单元实例

在现有的基于新型硬件描述语言实现的处理器核中，功能单元通常由一个抽象类进行定义，该类中会构造特定的参数并调用不同的方法，实例化出各种不同的功能单元的设计与实现。为了保证数据通路的完整性，并最大程度减少工作量，本工作复用了

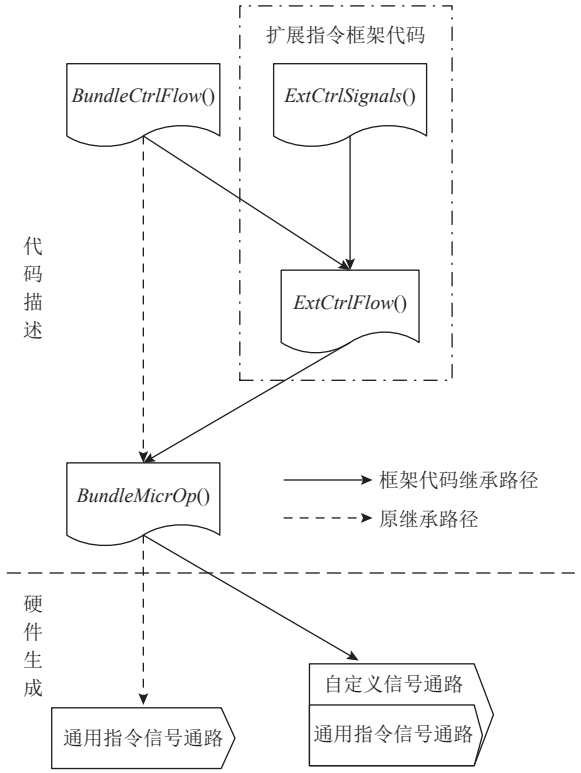


Fig. 6 Definition and generation of custom signal path

图 6 自定义信号通路的定义及生成

这样的定义方式，复用了功能单元的模板及其参数系统，并构造了完整的数据流和控制流通路。

因此，本工作在 ExtensionFrame 中继承了处理器的功能单元的抽象框架。并且，为还原乱序调度后丢失的指令上下文信息（1.2 节和 2.2 节所述），本工作在该框架中实例化了自定义指令调度单元（第 4 节中会详述该模块的实现）来消除冒险，以避免 DSA 执行单元的错误行为。

此外，本工作在 DSAFrame 中构造了一个 DSA 抽象类，在该抽象类中，模块的 I/O 部分的定义完全遵循指令扩展接口（2.1 节所述）设计规范。DSA 设计者需要继承这一抽象类，在其中嵌入设计好的 DSA，并手动完成对应接口信号的连接。最终，这个内嵌了 DSA 的抽象类会被 ExtensionFrame 中的功能单元抽象框架引用并在其内部完成实例化来完成信号的连接。上述流程如图 7 所示。在这个过程中，需要 DSA 设计者人工完成的步骤仅有：1）在抽象类中嵌入 DSA 实现；2）完成抽象类内接口的连线。

3.4 参数框架

这里的参数框架构造主要是用于构造和补齐功能单元抽象类所必须的参数。由此保证功能单元框架能够正确生成和例化，并在其前后搭建出完整的数据通路。

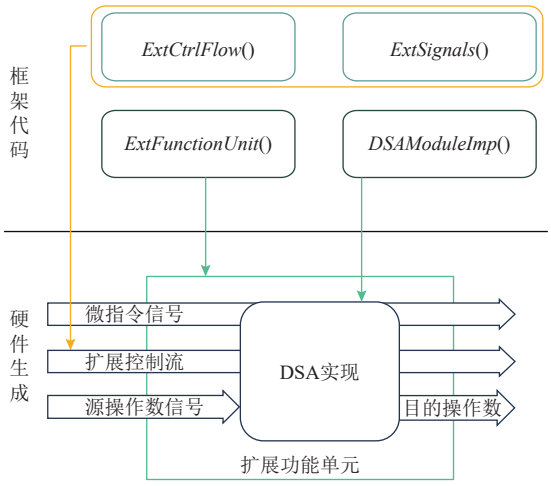


Fig. 7 The programming framework inherits native functional units and embeds DSA

图 7 编程框架继承原生功能单元并嵌入 DSA

当前版本的参数框架仅仅是为了保证整个流程可以正常工作而做出的工程实现。其中预留了 2 个可变参数用于指明：1）DSA 内部是否有寄存器；2）访存接口的宽度。

关于访存接口的实现，本框架复用了 Ld/St 指令原有的大部分设计与数据通路，与通用访存指令一同参与调度并与 L1Cache 交互。DSA 专用指令在译码阶段会被添加标记位，以确保从 L1Cache 获取访存数据后会通过流水线访存单元转发至 DSA 的自定义寄存器，而不影响通用寄存器状态。访存相关的各类冒险行为和一致性等问题则由通用访存模块进行维护。其中，DSA 内部有寄存器时，需要引入扩展指令调度单元，并完整地例化指令调度单元与流水线之间的信号交互通路。而访存宽度一方面直接用于访存接口的例化，另一方面会按照比例增加处理器中访存单元的数目。剩余的大部分参数为常量，并且其赋值暂时无法被修改。但是随着本工作的进一步迭代，在今后的工作中将尝试通过调整参数协商机制来实现更灵活的数据通路扩展和接口定义。

4 扩展指令调度单元

2.2 节中详述过，若 DSA 内部存在自定义存储结构，那么将 DSA 集成在乱序处理器的流水线中时，必须对指令流上下文的依赖关系进行恢复。因此，本文对 DSA 中可能会产生的冒险进行了分析，明确了指令调度单元需要解决的 4 类冒险：结构冒险、控制冒险、通用寄存器数据冒险和 DSA 寄存器数据冒险。

基于上述的观察和分析，本工作实现了一个指令调度模块。该模块的主要构成是一个指令调度表，指令调度表要通过与流水线中各阶段的调度部件进行交互，以确保本框架能够维护上述冒险对应的4类依赖，来避免错误的执行状态。

调度表与整个流水线之间的交互主要有以下3个部分构成：1)由流水线的分派单元顺序获得指令流及对应的微指令，确保能够维护完整的结构依赖和DSA寄存器中的数据依赖；2)与保留站交互以取得通用寄存器的读数和更新扩展指令的操作数状态，以确保通用寄存器中的数据依赖；3)与重排序缓存进行交互，并接入流水线冲刷信号，以确保错误分支预测错误导致的控制冒险不会改变体系结构状态。图8完整表述了调度表获取指令流及依赖信息的路径。

基于需要被维护的依赖关系，本工作构造了指令调度表。表1列举了调度表中关键的交互和其对应的含义。需要特别注意的是，在这个调度表中维护的数据依赖主要是DSA寄存器之间的数据依赖。而表中对通用寄存器值的获取和就绪状态的更新全部通过保留站获取。

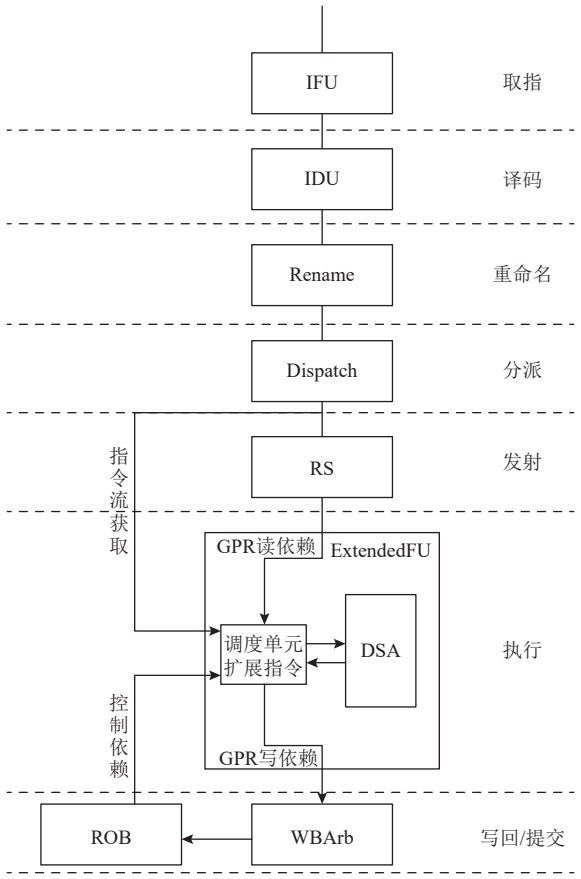


Fig. 8 The instruction scheduling unit retrieves instruction stream and dependency information from corresponding pipeline stages

图8 指令调度单元从对应流水线阶段获取指令流及依赖信息

Table 1 Signals Required for Maintaining Dependency Relationship

表1 维护依赖关系所需要的信号

调度所需交互信号	状态标记	维护的依赖关系
指令虚地址	-	无
DSA执行单元索引	-	结构依赖
DSA空闲状态	True/False	
第1个源寄存器的索引	-	数据依赖
rs1值就绪	True/False	
第2个源寄存器的索引	-	
rs2值就绪	True/False	
目的寄存器的索引	-	
目的寄存器值就绪状态	True/False	
目的寄存器写使能	-	控制依赖
指令执行状态	等待/发射/写回/提交	

注：“-”代表该处的值为具体的索引值而非状态标记。

指令调度表是一个循环队列。首先从分派阶段获取所有已经抵达的扩展指令，并顺序对其进行入队操作，相应地，调度表会将尾指针指向最后一条入队的指令的末尾并对所有的指令中的操作数状态进行初始化。

通用寄存器和DSA寄存器的数据依赖维护都通过3个寄存器的就绪状态字段来进行，但对应的指令状态更新条件略有不同。所有来自通用寄存器的操作数的就绪状态都会被初始化为“未就绪”。待保留站获取到对应寄存器的值并发射到这里时，会同步更新寄存器值并将对应的就绪状态置为“有效”。

对于目的寄存器为通用寄存器的指令，待源操作数就绪并且对应的DSA空闲即可将指令对应的操作数和控制信号通过扩展指令接口发送给DSA并将状态从“等待”更新为“发射”。DSA运算完成后传回计算结果。得到计算结果后该指令的微指令信号和写回值会被一起发送回流水线，通过标准的功能单元输出完成后续的写回和提交操作。当该指令在重排序缓存中被提交时，在调度表中的状态也会更新为“提交”，并同步更新“目的寄存器就绪状态”为真，此时该指令可以被退休。

当操作数索引了DSA寄存器，需要在表中寻找其与前序指令的目的寄存器的索引依赖关系。若有依赖，则将对应的就绪状态初始化为“False”，前序指

令完成了目的寄存器状态更新时，本条指令的源操作数的就绪状态才会被更新为“True”；若未发现依赖，则就绪状态被初始化为“True”。

目的寄存器为 DSA 寄存器的指令，指令状态保持“等待”，直到重排序缓存中该指令提交后，才会同步地将该指令在调度表中的指令状态更新为“提交”，并将指令转发给 DSA。指令的“提交”状态会维持一段时间，直至 DSA 向指令调度单元发出功能单元已空闲的信号，表示该指令已经计算完成，此时才会更新目的寄存器就绪状态为“True”，此时指令才会从表中出队。

结构依赖的维护已经包含在上述的流程中，调度单元在表中获取 DSA 执行单元的索引，查看对应执行单元的空闲状态并实时更新。控制依赖的处理主要由指令执行状态和流水线冲刷信号共同维护。当发现当前的指令流位于错误分支路径时，流水线中会产生冲刷信号。调度单元接收到信号后，会将调度表中所有非“提交”状态的指令进行冲刷，并将调度表的尾指针回滚至最后一个状态为“提交”的指令。

至此，本工作通过指令调度表完成了对 2 类数据依赖、结构依赖和控制依赖的维护和指令的调度。

5 评估结果

引言提到，基于 RISC-V 高性能乱序处理器设计并实现由扩展指令驱动的领域专用架构的完整工作流程目前仍然高度依赖设计者人工进行设计、开发与集成。本工作旨在降低开发者在此过程中的开发门槛及实际工作量。因此，评估工作重点从表征本框架为用户节约的工作量的角度来进行。

本节将首先展示不同的场景下，本框架帮助开发者完成的代码量，以证明本框架可以代替开发者完成的工作量；然后本文统计了传统的设计流程中，为了正确地修改相关部件设计以实现指令扩展，原本需要开发者亲自阅读并理解的流水线源代码的代码总量，以说明本框架能够降低此类工作的开发门槛。随后本文选取了 3 个不同的第三方 DSA 设计实例，通过本框架集成在处理器上，并统计了该过程中开发者需要手动添加或修改的代码行数，以证明框架面对不同的 DSA 实例能够灵活高效地完成其与处理器的集成。最后，为了证明经过框架集成的 DSA 切实可用，在模拟器上针对其中一个实例进行了数据计算和测试，得到了符合预期的结果。

5.1 参数配置

本文工作实现的框架是基于香山处理器完成的。本实验所采用的香山处理器架构配置如表 2 所示。此外，本文的评估采用了 4 类框架配置，每类配置包含的具体内容如表 3 所示。

Table 2 Microarchitectural Configuration of Processor

表 2 处理器微结构配置

部件	配置
分支预测器	16 KB TAGE-SC+ITTAGE+RAS+4 KB BTB
1 级数据缓存	128 KB，8 路数据缓存
1 级指令缓存	128 KB，8 路指令缓存
取指宽度	每周期 8×4B 指令
译码重命名宽度	每周期 6 条指令
ROB/LQ/SQ	256/80/64
物理寄存器堆	192 整数；192 浮点
执行单元	Int: 4×ALU，2×MDU，1×Misc Mem: 2×Ld AGU，2×St AGU Float: 4×FMA，2×Misc

Table 3 Framework Configurations for Evaluation

表 3 用于评估的框架配置

编号	配置内容
配置 1	DSA
配置 2	DSA+自定义控制信号
配置 3	DSA+自定义控制信号+指令调度
配置 4	DSA+自定义控制信号+指令调度+访存接口

5.2 评 估

首先，为证明本框架的可用性，在该框架的辅助下，本文集成了一个用于加速不同数据精度下的卷积计算的指令扩展 RV-CVP，并对其加速效果进行了评估。本文针对 2b, 4b, 8b, 16b 这 4 种精度，以通用指令的总周期数衡量，对同等规模的数据进行卷积计算，测试得到了加速比，结果如图 9 所示。并且在不同情况下，对系统执行的指令总数也进行了比较，结果如图 10 所示。结果显示，RV-CVP 最高可以获得 2.27~10.14 倍于通用指令的加速比，并能够大幅减少指令总数，证明了框架的可用性。

为了评估框架在不同场景下代替开发者完成的工作量，本文借鉴了 Twine[19]，选取了 Chisel 代码行数和根据其生成的 Verilog 代码行数这 2 个指标进行了评估，以表征本框架替代用户完成的工作量。

框架代码嵌入在香山处理器的流水线中，根据配置参数选择性生效和实例化，不能简单地将所有这些代码同等视之。因此，在不同的框架配置下，本

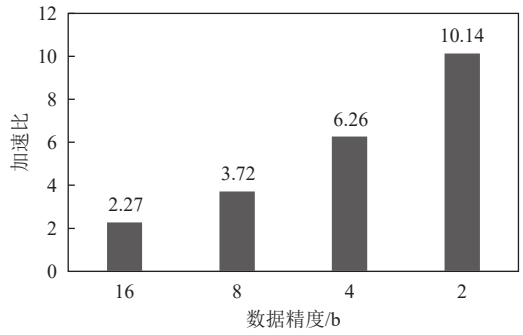


Fig. 9 Acceleration ratio of RV-CVP instruction extension

图 9 RV-CVP 指令扩展的加速比

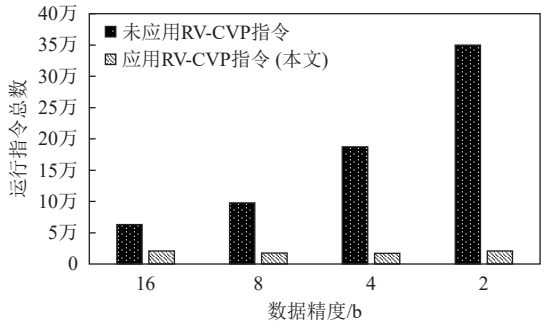


Fig. 10 The total number of instructions executed for convolution calculations with different precisions

图 10 不同精度卷积计算所运行的指令数

文统计了 Chisel 框架代码生效的行数, 以及这些生效的 Chisel 代码生成的 Verilog 代码的行数。

在框架代码中, 由于有一些参数和参数的协商逻辑是为了调整不同配置下不同代码的生效方式, 在人工实现的工作流程下, 这部分代码是不必要的, 因此本文在统计生效代码时没有将这部分代码列入任何一种配置中去。

如图 11 所示, 通过统计可以看出, 框架在较复杂 DSA 设计要求下, 在集成环节可以为用户省去超过 1 000 行的 Chisel 代码工作量。如果换成 Verilog 实现, 则最多可以节约 10 000 行左右的工作量。

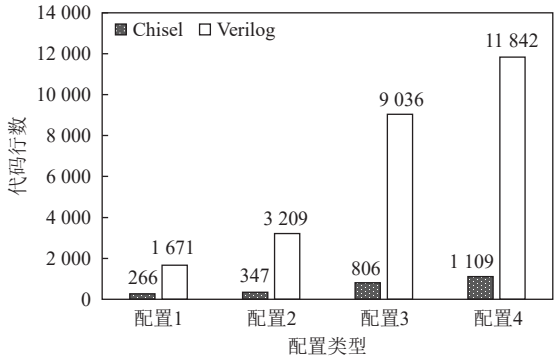


Fig. 11 Code workload saved by the framework under different experimental configurations

图 11 不同实验配置下框架节约的代码工作量

值得注意的是, 由于 Verilog 代码是根据 Chisel 生成的, 为了使后端工程师更好地调试和理解, 其代码风格并非采用了最有效率的编程方式(例如有时会将一个 Bundle 内的信号定义完全扁平化地拆分)。因此, 当人工进行同样的实现时, 可能存在可以被优化的空间。

在处理器上实现指令扩展驱动的 DSA, 需要花费相当的时间成本来阅读流水线源代码, 并理解流水线的工作原理, 这些开发成本无法直接体现在代码量上。因此, 本文希望能够通过间接的方式来表征这些隐性的工作量和开发成本。

阅读 Chisel 描述的流水线源代码时, 除了需要搞清楚代码模块本身描述的硬件逻辑, 还需要明白各抽象之间的调用关系及参数的传递路径。为此, 本文选取了修改流水线代码涉及的文件数目、模块数目及这些模块的代码总行数 3 个变量。

在基于 Chisel 开发的处理器中, 通常在模块间使用了跨层次的参数机制来方便硬件逻辑的复用和生成, 这有时会使得多种实体之间的调用关系趋于复杂。因此, 统计出流水线代码中需要修改的节点所涉及的文件数量和模块数量, 可以从侧面表征帮助开发者厘清 Chisel 中的抽象继承关系和参数传递所付出的努力。这些模块中包含的代码总行数可以间接表征阅读并理解这些流水线中相关模块具体的工作方式和原理所需要花费的时间。

统计结果如图 12 和图 13 所示, 在不同场景下, 即便是要集成一个最简单的 DSA 模块(配置 1), 需要阅读并理解的代码也接近 5 000 行。同时, 这些代码散布在 16 个文件的共计 37 个模块中。而在更复杂一些的场景(配置 4)下, 这些数字几乎翻了 1 倍。

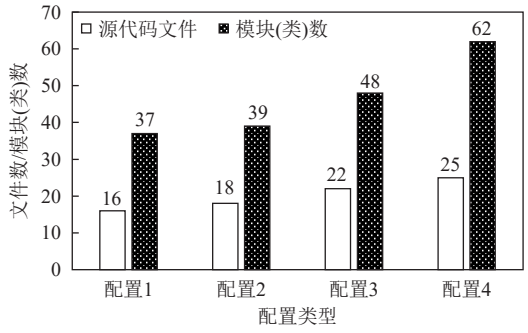


Fig. 12 Number of files and modules in the source code that developers need to read without the assistance of the framework

图 12 无编程框架辅助时开发者需阅读的代码文件和模块数

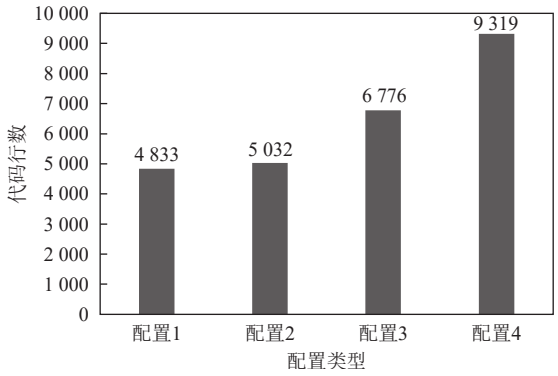


Fig. 13 Number of source code lines that developers need to read without the assistance of the programming framework

图 13 无编程框架辅助时开发者需要阅读的源代码行数

遗憾的是，本文难以将这部分努力用工作时长来表示，因为这取决于开发者的知识储备。习惯了面向对象和函数式编程的开发者来说，可能花费较少的时间就能完成这些工作，而对于这方面背景较薄弱的开发者，该过程就会比较漫长。

上述评估结果说明，本框架不仅代替开发者完成数目可观的代码编写量，同时，可以使开发者免于阅读理解大量的处理器源代码，从而为开发者节约开发时间并降低了开发门槛。

为了证明该框架面对不同类型 DSA 时有较广泛的适用性，本文在框架的辅助下将 3 个不同的设计实例集成到香山处理器上，并进行了评估。本文选取的 3 个实例为：用于可变精度卷积计算加速的 RV-CVP 自定义扩展，共有 12 条指令；用于加速脉冲神经网络的 RV-SNN 自定义扩展，共有 9 条指令；用于加速矩阵乘法的脉动阵列扩展 RV-SA，共 3 条指令来完成访存和矩阵乘法计算。

本工作对这些 DSA 设计进行有限的修改，以使其可以适用于本框架的 4 类配置，最后在本框架的辅助下完成集成。本文统计了此过程中开发者需要亲自在框架中编写的总代码行数，结果如图 14 所示。框架面对不同类型 DSA 时，开发者需要定义的代码量都显著少于同场景下手动集成所需要完成的代码量，证明了框架在不同场景下都可以很好地帮助开发者降低开发成本。对于不同的 DSA，开发者在框架中编写的代码量的不同主要来自于该 DSA 对应的扩展指令数量。为说明这一点，本文对配置 4 下的统计数据进行了更细致的划分。从图 15 可以看出，对不同的指令集，完成信号连接和参数修正所需要编写的代码量基本相同，而译码表和自定义信号束的定义部分会由于扩展指令数量的不同有明显改变。

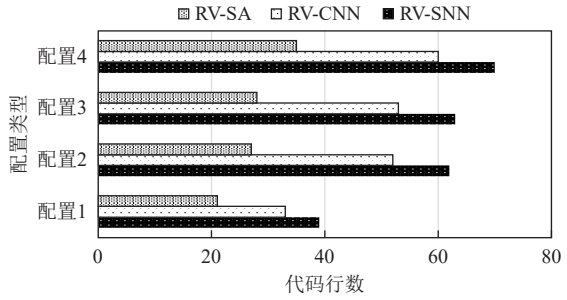


Fig. 14 Number of code lines required to be written in the framework for three instances under four types of configurations

图 14 3 个实例在 4 类配置下需要在框架中编写的代码行数

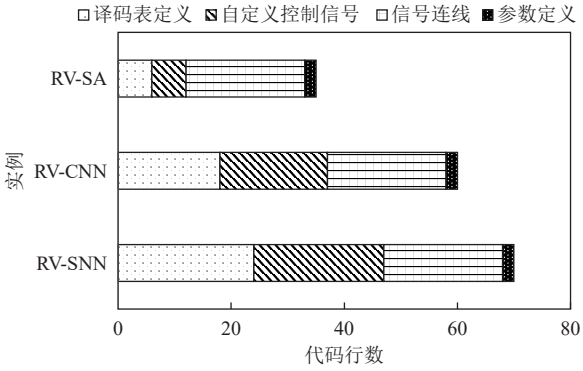


Fig. 15 Number of code lines written by developers within the framework according to different design elements

图 15 将开发者在框架内编写的不同设计要素的代码行数

需要补充说明的是，针对每个实例，并不是所有的配置都能保证集成后系统的正确性。由于 RV-SNN 扩展中包含不与内存直接交互的自定义寄存器，因此在配置 3 和配置 4 下可以正确运行。RV-SA 扩展包含可以与内存交互的寄存器，因此在配置 4 下可以正确运行。RV-CVP 扩展则直接对通用寄存器进行操作，因此在配置 2~4 下皆可正确工作。最后，本文为证明该框架的实用性，在 FPGA 上进行了综合仿真并获取了相应的实验数据。通用处理器配置不变，采用的 FPGA 型号为 Vertex VU19P。

本框架选择以 RV-SA 搭配 4 类配置进行综合，并统计了每类配置下框架本身的 FPGA 资源占用，结果如图 16 所示。由于配置 1 的资源占用非常少，在图中几乎不可见。从实际数据上看，指令调度表和访存接口的引入会占用较多的资源，但这样的数据表现是因为本实验搭载的通用处理器本身的调度机制和访存设计较为复杂。这也侧面证明了本框架在面对高性能处理器集成 DSA 这一场景时，确实为开

发者节约了大量的开发成本。对于本身的调度机制较为简单的处理器,例如没有重排序缓存或者没有推测执行,框架中的调度表可以被部分裁剪甚至直接改用不包含调度表的框架配置来辅助集成。

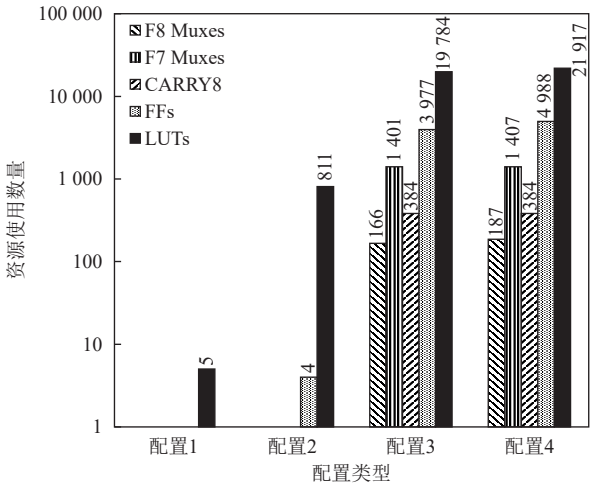


Fig. 16 Resource consumption of framework across different configurations

图 16 在不同配置下框架的资源占用情况

最后,我们将框架本身消耗的资源与通用处理器中用于指令调度所消耗的资源做了对比,结果见图 17。我们的对比对象是通用高性能处理器用于指令调度的核心模块的资源占用与配置 4 下的框架代码的资源占用情况。可见框架代码虽然在配置 3~4 的场景下占用资源较多,但是相较于对应场景下的处理器而言,这些资源占用是完全可以接受的。

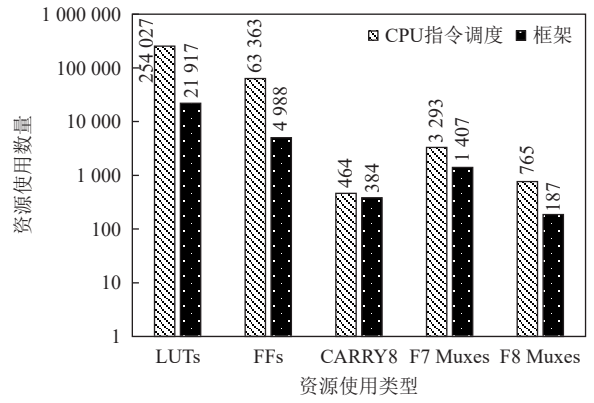


Fig. 17 Comparison of resource occupancy between software frameworks and CPU instruction schedulers

图 17 软件框架与 CPU 指令调度单元资源占用情况对比

6 讨 论

本节旨在探讨将本框架的设计方法应用于其他的 RISC-V 处理器的可行性,以及当前版本的框架设计中存在的局限性。

本框架的设计方法主要解决了 3 个关键问题:1)实现可灵活扩展的信号通路;2)帮助 DSA 乱序指令流中的各类依赖;3)利用面向对象范式实现快速集成。从方法论上来看,只要处理器设计时所采用的硬件描述语言具备面向对象的特性,采用本工作的方法实现一套类似的编程框架是完全可行的。同时,本工作中,指令调度表针对不同的指令依赖关系都给出了对应的调度机制。因此对于不同类型的处理器或 DSA,可以通过有选择地简化相应的调度机制来实现不同处理器-DSA 组合的适配。因此,本框架的设计方法,在上述条件下是可以适用于香山处理器以外其他 RISC-V 处理器的。同时,随着越来越多的硬件描述语言具备更高级的语言特性,本工作的适用范围也会越来越广泛。

当前版本的框架设计也尚存一些局限性。在目前的实现中,对于 DSA 所需要的一些基本控制信号,可以通过译码表的扩展来进行生成和传递,但无法在流水线中对这些信号进行任何前置的逻辑处理。例如中断和异常这样需要在处理器一端设计相应的逻辑进行处理的信号,暂时无法被纳入到框架中。这使得自定义控制信号的能力在具体的实现中会受到一定程度的限制。

另外,在当前版本中,本工作的参数框架有部分内容是为了适配香山处理器内部模块的参数化机制而完成的机械化补全。对此,有 2 种未来的改进思路:1)将参数机制对框架的用户完全透明化,进一步精简框架中需要开发者手动指定的内容;2)将参数框架的后台机制进一步充实,引入自动化的参数调节机制,根据用户提供的参数自动调整框架及接口配置以实现整个系统的优化。

目前的框架仅能支持非特权级指令的扩展,不支持任何特权指令相关的实现。对于是否需要引进这方面的功能,以及如何适配现有的框架,还需要进一步的讨论和探究。

最后,本工作的方法虽然是基于香山处理器做出了实现,但是从理论上讲,只要处理器所用的硬件描述语言包含面向对象特性,都可以基于本工作的方法论实现类似的开发框架。并且,本工作中,通过实现调度表来帮助 DSA 处理通用处理器指令流中可能存在的各类依赖问题。对于调度机制较为简单的处理器,在对本框架进行裁剪的基础上实现一套简化的版本,以适配更简单的流水线设计是理论可行的。

7 总 结

本文对现有的 RISC-V 自定义指令扩展工作进行了调研。并通过前期的实践，归纳出指令集扩展硬件实现流程的 4 个基本阶段，提炼出可以用工具替代人工完成的部分工作流程。

基于上述分析，本文提出了一个基于 RISC-V 乱序处理器的自定义指令扩展框架。该框架基于香山处理器进行了实现，利用新型硬件描述语言的面向对象特性，辅助开发者将设计好的 DSA 集成在通用的高性能乱序处理器中，并以扩展指令的方式驱动其进行计算。

经过多方面的评估，该框架在不同的 DSA 配置下，可以大幅度节约开发者在将 DSA 集成到处理器这一工作上所花费的人力成本和时间成本，降低开发门槛。

作者贡献声明：刘伯然完成了论文的撰写和部分的代码实现工作；张承译完成了部分的代码实现和实验数据；胡思鹏提供了部分的实验数据；赵地、包云岗、孙凝晖提供了实验和开发思路，给予工作支持和指导意见。

参 考 文 献

[1] Asanovic K, Patterson D A, Celio C. The Berkeley out-of-order machine (Boom): An industry-competitive, synthesizable, parameterized RISC-V processor[R]. Berkeley: University of California at Berkeley, 2015

[2] Xu Yinan, Yu Zihao, Tang Dan, et al. Towards developing high performance RISC-V processors using agile methodology[C]//Proc of the 55th IEEE/ACM Int Symp on Microarchitecture (MICRO). Piscataway, NJ: IEEE, 2022: 1178−1199

[3] Chen Chen, Xiang Xiaoyang, Liu Chang, et al. XuanTie−910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension: Industrial product[C]//Proc of the 47th ACM/IEEE Annual Int Symp on Computer Architecture (ISCA). Piscataway, NJ: IEEE, 2020: 52−64

[4] Hameed R, Qadeer W, Wachs M, et al. Understanding sources of inefficiency in general-purpose chips[C]//Proc of the 37th Annual Int Symp on Computer Architecture. New York: ACM, 2010: 37−47

[5] Liu Chang, Wu Yanjun, Wu Jingzheng, et al. Survey on RISC-V system architecture research[J]. Jornal of Software, 2021, 32(12): 3992−4024 (in Chinese)
(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)

[6] Cui Enfang, Li Tianzheng, Wei Qian. RISC-V instruction set architecture extensions: A survey[J]. IEEE Access, 2023, 11: 24696−24711

[7] Asanovic K, Avizienis R, Bachrach J, et al. The rocket chip generator, UCB/EECS-2016-17[R]. Berkeley: EECS Department, University of California, Berkeley, 2016

[8] Zaruba F, Benini L. The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology[J]. IEEE Transactions on Very Large Scale Integration Systems, 2019, 27(11): 2629−2640

[9] Traber A, Zaruba F, Stucki S, et al. PULPino: A small single-core RISC-V SoC[C]//Proc of the 3rd RISCV Workshop. Zurich: RISC-V International, 2016: 15

[10] Leiserson C E, Thompson N C, Emer J S, et al. There's plenty of room at the top: What will drive computer performance after Moore's law?[J]. Science: 2020, 368(6495): eaam9744

[11] Bachrach J, Vo H, Richards B, et al. Chisel: Constructing hardware in a scala embedded language[C]//Proc of the 49th Annual Design Automation Conf. New York: ACM, 2012: 1216−1225

[12] Hennessy J L, Patterson D A. Computer Architecture: A Quantitative Approach[M]. San Francisco: Morgan Kaufmann Publishers Inc, 2011

[13] OpenHW Group. CVA6 RISC-V CPU[DB/OL]. [2026-02-11]. https://github.com/openhwgroup/cva6

[14] UC Berkley. BOOM: The Berkeley out-of-order RISC-V processor [DB/OL]. [2026-02-11]. https://github.com/riscv-boom/riscv-boom

[15] Nikhil R. Bluespec system Verilog: Efficient, correct RTL from high level specifications[C]//Proc of the 2nd ACM and IEEE Int Conf on Formal Methods and Models for Co-Design(MEMOCODE'04). Piscataway, NJ: IEEE, 2004: 69−70

[16] Panda P R. SystemC: A modeling platform supporting multiple design abstractions[C]//Proc of the 14th Int Symp on Systems synthesis. New York: ACM, 2001: 75−80

[17] Koeplinger D, Feldman M, Prabhakar R, et al. Spatial: A language and compiler for application accelerators[C]//Proc of the 39th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2018: 296−311

[18] Sutherland S, Davidmann S, Flake P. SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling[M]. Berlin: Springer, 2006

[19] Chen Shibo, Fisseha Y, Jeannin J B, et al. Twine: A Chisel extension for component-level heterogeneous design[C]//Proc of the 25th Design, Automation & Test in Europe Conf & Exhibition (DATE). Piscataway, NJ: IEEE, 2022: 466−471

[20] McGeer P C, Cheng S T, Meyer M J, et al. Hardware design language for the design of integrated circuits: US Patent 006421808B1[P]. 2002-07-16

[21] SpinalHDL. SpinalHDL[DB/OL]. [2026-02-11]. https://github.com/SpinalHDL/SpinalHDL

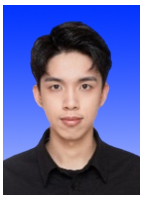
[22] Taylor M B. BaseJump STL: SystemVerilog needs a standard template library for hardware design[C]//Proc of the 55th Annual Design

Automation Conf. New York: ACM, 2018: 1−6

[23] Bao Yungang, Chang Yisong, Han Yinhe, et al. Agile design of processor chips: Issues and challenges[J]. Journal of Computer Research and Development, 2021, 58(6): 1131−1145 (in Chinese)
(包云岗, 常轶松, 韩银和, 等. 处理器芯片敏捷设计方法: 问题与挑战[J]. 计算机研究与发展, 2021, 58(6): 1131−1145)



Liu Boran, born in 1992. PhD. His main research interests include computer architecture and domain-specific architecture.
刘伯然, 1992 年生。博士。主要研究方向为计算机系统结构、领域专用结构。



Zhang Chengyi, born in 2000. Master. His main research interests include integrated circuit design and cache microarchitecture.
张承译, 2000 年生。硕士。主要研究方向为集成电路设计、缓存微结构。



Hu Sipeng, born in 1998. Master. His main research interest includes computer architecture.
胡思鹏, 1998 年生, 硕士。主要研究方向为计算机系统结构。



Zhao Di, born in 1978. PhD, associate professor. Member of CCF. His main interests include computer architecture and artificial intelligence.
赵　地, 1978 年生。博士, 副研究员。CCF 会员。主要研究方向为计算机系统结构、人工智能。



Bao Yungang, born in 1980. PhD, professor. Member of CCF. His main research interests include datacenter architecture, agile design methodology of processor chips, and ecosystem of open-source processor chips.
包云岗, 1980 年生。博士, 研究员。CCF 会员。主要研究方向为数据中心系统结构、处理器芯片敏捷设计方法论、开源处理器芯片生态。



Sun Ninghui, born in 1968. PhD. Academician of Chinese Academy of Engineering. Fellow of CCF. His main research interests include computer architecture and high performance computing.
孙凝晖, 1968 年生。博士。中国工程院院士。CCF 会士。主要研究方向为计算机系统结构、高性能计算。