计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2019.20180771
56(1):35-48,2019

芯片敏捷开发实践:标签化 RISC-V

余子濠[1,2] 刘志刚[1,2] 李一苇[1,2] 黄博文[1] 王 卅[1,2] 孙凝晖[1,2] 包云岗[1,2]
[1](计算机体系结构国家重点实验室(中国科学院计算技术研究所) 北京 100190)
[2](中国科学院大学 北京 100049)
(yuzihao@ict.ac.cn)

Practice of Chip Agile Development: Labeled RISC-V

Yu Zihao[1,2], Liu Zhigang[1,2], Li Yiwei[1,2], Huang Bowen[1], Wang Sa[1,2], Sun Ninghui[1,2], and Bao Yungang[1,2]
[1](*State Key Laboratory of Computer Architecture* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)
[2](*University of Chinese Academy of Sciences*, *Beijing* 100049)

Abstract Current chip design projects require considerable manpower and time to carry out, and have certain risks. These conditions have limited the development of open-sourced chip design to some extent. To further reduce the threshold for chip development, research teams at University of California, Berkeley have designed the open ISA RISC-V. They also open-sourced the Rocket Chip project, the SoC implementation of RISC-V, and put forward Chisel, a new hardware construction language, for agile development. How do RISC-V, Rocket Chip and Chisel enable open-source chip agile development? With some case studies during the development of the Labeled RISC-V project led by the Institute of Computing Technology, Chinese Academy of Sciences, this article shows: 1) An open and active ISA ecosystem (such as RISC-V) is a necessary condition to promote chip innovation; 2) Chisel's features such as bulk connection, metaprogramming, object-oriented programming, and functional programming, can greatly reduce the amount of code and improve code maintainability; 3) Agile development can achieve an order of magnitude improvement in coding efficiency, while achieving comparable or even better performance, power consumption and area overhead than traditional hardware development models.

Key words RISC-V; Chisel; open-source; chip design; agile development

摘 要 随着开放指令集 RISC-V 的流行,开源芯片的概念逐渐进入人们的视野.但是目前的芯片设计项目需要投入相当的人力和时间才能开展,并且具有一定的风险,这些情况一定程度上限制了开源芯片的发展.为了进一步降低芯片开发的门槛,加州大学伯克利分校先后设计了开放指令集 RISC-V,开放了其 SoC 实现 Rocket Chip 的项目源码,并提出了一门面向敏捷开发的硬件构建语言 Chisel.RISC-V,

收稿日期:2018-11-17;**修回日期**:2018-12-03
基金项目:国家重点研发计划项目(2016YFB1000201);国家自然科学基金项目(61420106013, 61702480);中国科学院青年创新促进会(2013073)
This work was supported by the National Key Research and Development Program of China (2016YFB1000201), the National Natural Science Foundation of China (61420106013, 61702480), and the Youth Innovation Promotion Association of Chinese Academy of Sciences (2013073).
通信作者:包云岗(baoyg@ict.ac.cn)

Rocket Chip 和 Chisel 是如何赋能开源芯片敏捷开发?将基于中国科学院计算技术研究所的研究工作“标签化 RISC-V”项目开发过程中的若干案例,展示:1)开放又活跃的指令集生态(如 RISC-V)是推动芯片研发创新的必要条件;2)Chisel 的信号整体连接、元编程、面向对象编程以及函数式编程等特性可大幅缩减代码量,提升代码可维护性;3)敏捷开发能在编码效率提升一个数量级的同时,达到与传统硬件开发模式相当甚至更优的性能、功耗与面积.

关键词 RISC-V;Chisel;开源;芯片设计;敏捷开发

中图法分类号 TP302.1

芯片是信息技术的引擎,推动着人类社会的数字化、信息化与智能化.随着摩尔定律濒临终结,维持芯片技术创新面临挑战.开源芯片设计将是应对挑战的新思路.

如今芯片设计动辄需要上亿研发费用、投入上百人年,而一旦流片失败,将会浪费巨大的成本,只有少数企业才能承担相应的风险[1].反观互联网领域通过开源软件降低开发门槛,创造了繁荣的互联网产业.如果开源芯片设计能实现敏捷开发,将芯片设计门槛降低几个数量级——3~5 人的小团队在 3~4 个月内,只需几万元便能研制出一款有市场竞争力的芯片,就可以大大降低芯片开发的成本和风险,必将吸引大量人员投入芯片产业,重塑繁荣.

加州大学伯克利分校设计的开放指令集 RISC-V[2]朝着这个目标迈出了第 1 步,它有望像开源软件生态中的操作系统 Linux 那样,成为计算机芯片与系统创新的基石.但为了实现芯片的敏捷开发,只有 RISC-V 是远远不够的,还需要一个能够经过流片验证的 RISC-V SoC 开源设计,以及一门面向敏捷开发的硬件构建语言[3].因此在推广 RISC-V 的同时,伯克利研究团队也开放了 RISC-V 的 SoC 实现 Rocket Chip[4-5],以及面向敏捷开发的硬件构建语言 Chisel[6],期望通过这三驾马车率领芯片设计领域迈进敏捷开发的时代①.

本文希望探索这样一个问题:RISC-V,Rocket Chip 和 Chisel 是如何推动芯片敏捷开发的?为了探讨这个问题,我们将探讨 3 个方面的内容.

1) RISC-V 和 Rocket Chip 如何降低芯片开发的门槛?为了探索这个问题,我们介绍了 RISC-V 的开源开放理念,同时以中国科学院计算技术研究所开展的标签化 RISC-V[7-8](Labeled RISC-V)研究项目为案例,分别从设计开源度、定制灵活性、生态完整性以及社区活跃度 4 个方面,把开放不活跃的 SPARC V9[9]、活跃不开放的 MicroBlaze[10],与开放又活跃的 RISC-V 进行对比,揭示了一款不开放或者不活跃的指令集对芯片设计项目带来的限制,从而展示了 RISC-V 和 Rocket Chip 对降低芯片开发门槛的价值.

2) Chisel 如何对项目的敏捷开发提供帮助?为了回答这个问题,我们分别介绍了 Chisel 的信号整体连接、元编程、面向对象编程以及函数式编程的特性.同时我们以标签化 RISC-V 中的项目经验为例子,展现了这些特性如何帮助我们快速地开发项目需要的功能,并将这些特性与传统的硬件描述语言,包括 Verilog[11]和 SystemVerilog[12]进行比较,展示 Chisel 代码的简洁性、易读性以及易维护性,从而对项目的敏捷开发提供帮助.

3) 敏捷开发和传统开发相比,编码的效率和质量怎么样?为了展现 2 种开发模式的效率对比,我们分别使用 2 种模式来开发一个功能相同的二级缓存(L2 cache)模块.结果显示:与传统开发模式相比,敏捷开发模式的开发效率提升了一个数量级,同时编写出的代码可读性更好,更容易排除错误.为了展示 2 种开发模式的质量对比,进一步屏蔽设计的差异,我们将采用传统开发模式的 Verilog 代码翻译成功能相同的 Chisel 代码,并在 FPGA 流程上对翻译前后代码的性能、功耗、面积进行评估.结果显示:敏捷开发能达到与传统开发相当甚至更优的编码质量.

1 开放指令集的理念

2010 年,伯克利研究团队准备为接下来的一系列项目选择一款指令集.结合项目的需求,他们最后决定在 x86 和 ARM 中进行选择.但是选择 x86 是不可能的,首先它有知识产权问题,而且设计非常复杂.但 ARM 也是几乎不可能,它不仅和 x86 有同样的问题,而且当时并没有 64 位指令集的规范.这让伯克利研究团队感到困扰.

① RISC-V 和 Chisel 是由伯克利中 2 个不同的团队分别设计的

他们对当时指令集的状况进行了一些调研,发现即使是一个小小的 SoC 芯片,里面也包含很多处理器,包括应用处理器、图形处理器、图像处理器、视频 DSP、音频 DSP 等.但是这些大大小小的处理器使用的指令集可能都各不相同,比如应用处理器一般用 ARM 指令集,从不同厂商购买的 IP 核也许都会用自己私有的指令集,就连 SoC 厂商自己设计的核心,也可能会用自己的指令集.而这些都位于同一颗 SoC 芯片上的各种核心,都有自己独立的一套软件栈.

但是,我们真的需要这么多不同的指令集吗?为了回答这个问题,伯克利研究团队又调研了开源软件的状况,发现和指令集的情况大不相同.如表 1 所示,开源软件中的不同领域,都有一套开放的标准,在这套开放的标准之下,既有开放自由的实现,也有私有的实现.虽然总体上私有实现的效果更优,但开放自由的社区也非常活跃.然而,以私有实现主导的指令集领域却毫无生机.

Table 1 Summary of Open Software Standards

表 1 开放软件和标准小结[13]

Field	Standard	Free, Open Implementation	Proprietary Implementation
Networking	Ethernet, TCP/IP	Many	Many
OS	Posix	Linux, FreeBSD	M/S Windows
Compilers	C	GCC, LLVM	Intel Icc, ARMcc
Databases	SQL	MySQL, PostgresSQL	Oracle 12C, M/S DB2
Graphics	OpenGL	Mesa3D	M/S DirectX
ISA	None	None	x86, ARM, IBM360

如果有一款开放自由的指令集,大家都可以用它来做任何事情,会怎么样呢?借鉴软件领域的发展状况,伯克利研究团队认为,指令集体系结构作为软硬件接口的一种标准,不应该像 x86 和 ARM 等指令集那样需要授权才能使用,而应该开放(open)出来让大家自由(free)使用,这样才能塑造指令集领域的繁荣生态[14].于是伯克利研究团队发起了一个持续 3 个月的暑期项目,目标是从零开始设计一款新的指令集,并将其彻底开放.

这套新的指令集被命名为 RISC-V(读作 RISC-Five).2011 年 5 月第 1 版 RISC-V 指令集正式发布.

实际上,第 1 版 RISC-V 发布后并未受到关注,也未取得预期反响,反而备受多方质疑.一方面,很多学术界人士认为 RISC-V 指令集毫无技术创新;另一方面,工业界对于这种由学术界推出的新指令集也是持观望态度.开放指令集的理念和意义并未得到广泛认可.

面对各方质疑,伯克利研究团队采取了 3 项措施:

1) 研究团队设计并实现了一个基于 RISC-V 指令集的顺序执行 64 位处理器核心(代号为 Rocket),并对相应的 SoC 设计 Rocket Chip 进行开源.随后伯克利研究团队又推出了开源的乱序执行核心 BOOM(Berkeley out-of-order machine)[15-16].这两款开源的微结构设计打破了学术界长期缺乏可用芯片原型的困境,很快吸引了学术界的广泛关注.

2) 伯克利研究团队在 2015 年成立非盈利组织 RISC-V 基金会(RISC-V Foundation),旨在凝聚全世界的力量一起共同构建开放、合作的软硬件社区,打造 RISC-V 生态系统.

3) 伯克利研究团队从 2015 年开始组织举办 RISC-V 技术研讨会,鼓励企业和科研机构在研讨会上分享 RISC-V 相关的工作和研究,从而传播 RISC-V 开放指令集的理念和意义,大大地加深了人们对 RISC-V 的了解[17].

伯克利研究团队实施的这一套组合拳,很快让 RISC-V 成长为一个开放又活跃的社区.

2 标签化 RISC-V 项目中的指令集选择

关于 RISC-V 开源开放的理念对学术研究的意义,我们在开展标签化体系结构这一科研项目的过程中也深有体会.标签化 RISC-V 是一个基于 RISC-V 指令集的标签化体系结构的实现,它基于 RISC-V 指令集的开源实现 Rocket Chip,添加了标签化体系结构相关的功能.标签化体系结构的主要思想是通过标签向底层硬件传播软件信息,向硬件添加身份识别、区分服务、性能调控等新功能.标签化体系结构有 4 点特性[18]:

1) 细粒度对象.所有访存和 I/O 请求都带上标签.

2) 语义关联.标签和软件实体(包括虚拟机、进程、线程或变量)进行关联.

3) 携带传播.标签随着请求一同在整个系统中传播.

4) 可编程控制逻辑.对携带不同标签的不同请求进行区分化处理.

基于这些特性,标签化体系结构的其中一个应用场景,是保证关键应用性能的同时,提高系统的资源利用率[19].具体地,我们可以通过标签向底层硬件传递访存和 I/O 请求的身份信息,这一身份信息

会随着请求在整个系统中传播.这样以后,硬件共享资源的控制器(如末级缓存控制器、内存控制器等)就可以识别收到的一个请求来源于哪个应用,若发现请求来源于关键应用,则优先对其进行处理,并保证其使用的资源不受非关键应用的干扰;若发现请求来源于非关键应用,则对其提供尽力而为的服务.

原则上,标签化体系结构可以实现在任意一款指令集之上.在项目前期,我们也曾经在多款指令集中进行选择和尝试,最终体会到 RISC-V 的开放活跃对学术研究的意义.

2.1 开放不活跃的 SPARC V9

我们一开始在 2013 年 7 月选择了 OpenSPARC T1[20]来开展项目.它是一个 8 核 32 线程的 64 位工业级微结构实现,采用 SPARC V9 指令集,在 2006 年通过 GPL 协议开源.我们本来以为这款工业级的开源实现可以给项目带来性能上的保障,但是在开展过程中,我们遇到了 4 个挑战:

1) 工具不再维护,所依赖的库版本较老,难以寻找与其适配的开发环境;

2) 缺少详细设计文档,工程中注释较少,阅读并理解工程中的 Verilog 代码较为困难;

3) 生态不完善,难以运行真实应用;

4) 社区不再活跃,难以寻求帮助.

我们原计划将 OpenSPARC T1 的实现从 8 核 32 线程裁剪为单核单线程,再开展标签化的研究.但由于上述挑战,裁剪的定制工作非常困难,进行了约半年时间却未有明显进展,最后无奈放弃 OpenSPARC T1 这款开源设计.这说明,只开放但不活跃的指令集,虽然能让研究者使用,但定制的难度很高,要基于一个不活跃指令集开展项目研究还是很困难.

2.2 活跃不开放的 MicroBlaze

之后我们尝试了赛灵思(Xilinx)的 MicroBlaze 架构,它是一款主要面向嵌入式领域的处理器.与 OpenSPARC T1 相比,MicroBlaze 的微结构设计、生态和社区都由赛灵思维护.2009 年 6 月,MicroBlaze 作为第 1 款软核 CPU 架构并入 Linux 内核主线;3 个月后,MicroBlaze 的 GNU 工具链也开始并入相应的主线仓库[21]并一直在演进,而 GCC 则从 4.6 发行版开始支持 MicroBlaze[22].在赛灵思论坛的 MicroBlaze 板块,开发者踊跃提问,并且能很快得到赛灵思工作人员的答复.这表明,MicroBlaze 的生态和社区状况非常健康,这也是我们项目选择 MicroBlaze 的一个重要原因.

我们虽然在开发过程中也会遇到困难,但借助社区的力量,我们很快就解决了困难,构建出一套可以运行 Linux 的系统.然后我们在 MicroBlaze 上进行了 4 个方面的扩展,实现了标签化体系结构:

1) 细粒度对象.在核外添加标签寄存器.

2) 语义关联.标签寄存器中存放虚拟机的标识.

3) 携带传播.把标签寄存器中存放的标签连接到 AXI 总线的 USER 域中,通过 AXI 总线将标签传播到整个系统中.

4) 可编程控制逻辑.向赛灵思系统缓存(Xilinx system cache[23])模块的替换算法添加基于标签路划分的功能,并添加可编程的缓存控制逻辑模块,对路划分的参数进行编程,来实现缓存容量的隔离;同时在内存控制器前的数据通路中添加基于标签的令牌桶模块,并添加可编程的内存控制逻辑模块,对令牌桶的参数进行编程,来实现内存带宽的隔离.

虽然我们成功在 MicroBlaze 上实现了标签化体系结构,但在实现过程中,我们仍然遇到了表 2 中的困难.

Table 2 Summary of Challenges About MicroBlaze

表 2 使用 MicroBlaze 的挑战总结

Cause	Solution	Example
Complexity	Spend more time	Hard to read the source code of cache module
Closed source	Bypass	AXI data width converter cuts off the propagation of label
	No solution, but acceptable	Low performance of core
	No solution	No SMP OS; Can not tapeout

表 2 中,第 1 种困难属于模块复杂度较高,同时缺少相关资料帮助理解该模块的细节.例如赛灵思系统缓存模块的代码约 6 万行,我们需要在其中添加基于标签路划分的替换算法,但由于缺少详细设计文档和注释,理解代码较为困难.这种困难可以通过投入足够时间来克服.我们安排了一位工程师专门阅读赛灵思系统缓存模块的代码,花费了将近半年时间才完成替换算法的添加,虽然最终成功实现这一功能,但这也一定程度上也影响了从想法到原型的周期(time to prototype).

第 2 种困难是代码不开源带来的限制,虽然对工程实现有一定影响,但可付出少量代价绕开.例如 AXI 数据宽度转换器会截断 USER 域[24],导致基于 USER 域实现的标签无法继续往后传播.但由于

相关代码不开源,我们无法修改转换器的代码,只能在项目中小心地规划模块的顺序,避免在标签传播的通路上使用这个转换器;在必须使用转换器的情况下,只能额外将标签接入到转换器的下游模块中来让标签继续传播.另一个例子是由于 MicroBlaze 处理器不开源,我们无法在处理器内部添加寄存器来实现进程级标签,只能把标签寄存器放置在处理器外面,让操作系统通过 I/O 的方式来设置进程级标签.虽然这个解决方案会带来一定的性能开销,但至少绕开了无法修改处理器的限制.

第 3 种困难同样是代码不开源带来的限制,虽然没有解决方案,但相关的研究工作还是可以继续开展.例如 MicroBlaze 处理器主要面向嵌入式场景,性能一般.我们在测试后发现,在运行云计算典型应用(如 Memcached)时,软件管理的 TLB 在缺页频繁时带来了约 20%的性能开销,但由于处理器代码不开源,我们无法对相关设计进行改进.不过对于我们的项目来说,这一性能开销还是可以接受.

第 4 种困难也是代码不开源带来的限制,但却无法解决,会直接限制相关研究工作无法开展.例如 MicroBlaze 无法运行多核操作系统,这是因为为了运行多核操作系统,我们需要一种多核核间通信的方式,例如核间中断(inter-processor interrupt,IPI),还需要一种内存同步和通信的方式.要实现这些功能,就需要对处理器内部进行改动[25],但由于处理器代码不开源,我们无法对其进行改进,使其支持多核操作系统.另一个例子是流片,不开源的处理器自然也无法流片.

因此,不开放的指令集和微结构实现,会对相关研究工作在项目周期、工程实现和性能表现等方面带来颇多影响,甚至会使得部分颠覆式前沿研究无法开展.为了解决这些问题,就需要有一款开放的指令集以及相应的开源微结构实现.

2.3 开放又活跃的 RISC-V

有了 SPARC V9 和 MicroBlaze 的经历,我们意识到需要有一个开放又活跃的指令集来支撑项目的开展.事实上,期间我们还因为 ARM 的活跃社区和生态而调研过基于 ARM 的解决方案,然而 ARM 的开放性比 MicroBlaze 还弱,而且需要支付高额的授权费用,一般的研究项目无法承担,最终还是放弃了 ARM 的选择.最后我们选择了 RISC-V,依托其开放的理念及活跃的生态,在其微结构实现 Rocket Chip 上成功解决了在 MicroBlaze 上遇到的问题:

1) 实现简单.Rocket Chip 的二级缓存模块代码约 1 000 行,即使缺少详细设计文档和注释,我们花费约 3 天时间就成功实现了之前在赛灵思系统缓存上实现的功能,效率提高了 50 倍.

2) 修改灵活.我们通过 Rocket Chip 上使用的开放总线协议 TileLink[26] 来传播标签,我们在 TileLink 总线上灵活地添加一组信号来专门传播标签,在 TileLink 总线适配器(如数据宽度转换器等)不能自动传播标签时,可以灵活地修改适配器的硬件代码来实现标签的传播;此外,我们可以灵活地在 Rocket 核心中添加一个新的控制状态寄存器(control status register, CSR)来存放进程级标签的信息,操作系统可以通过 CSR 指令高效地进行设置.

3) 性能较高.RISC-V 的 TLB 为硬件管理,性能比 MicroBlaze 高,而且若性能不能满足项目需要,可以对处理器自由进行修改.

4) 支持多核.RISC-V 丰富的生态已经支持多核操作系统的运行.

5) 允许流片.Rocket Chip 的代码全部开源,RISC-V 也无需授权费用,可以进行流片.

表 3 总结了我们项目对不同指令集进行的尝试.其中,基于 SPARC V9 指令集的 OpenSPARC T1 的微结构设计虽然开源,但由于代码难以阅读,对设计进行定制化非常困难,而且软件生态的支持非常有限,社区也不活跃,对科研项目来说难以使用.MicroBlaze 的生态和社区都由赛灵思进行维护,活跃度很好,但其设计不开源,只能对其进行很有限的配置,虽然总体上能满足一些嵌入式研究工作的需求,但对于性能要求更高或者对芯片有所创新的研究工作,MicroBlaze 还是难以胜任.RISC-V 则结合了两者的优点,首先开放的理念吸引了大批企业和个人帮助其建设生态,社区非常活跃;其次 RISC-V 有一款开源的微结构设计 Rocket Chip,无论是指令集的模块化,还是微结构的设计细节,大家都可以根据各自的需求对其进行灵活的个性化定制,而且可以直接将设计进行流片.

Table 3 Summary of Different ISA

表 3 不同 ISA 的特点总结

Characteristics	SPARC V9	MicroBlaze	RISC-V
Open-sourcing	√√√	√	√√√
Flexibility	√	√	√√√
Ecosystem	√	√√	√√√
Community	√	√√√	√√√

正是这些好处给芯片相关的科研工作带来了很多崭新的机会:与模拟器相比,在一个可综合的平台上进行验证的工作将会更有说服力,同时距离真正芯片的实现也更近;而对工业界来说,这些好处也同样受到青睐.因此,开放活跃的指令集和相应开源的微结构设计,是迈向开源芯片设计的第1步.

3 基于 Chisel 的敏捷开发

在学术界,目前大多数微结构相关的研究还是在模拟器上进行,这是因为基于 FPGA 的工作周期一般都比较长,例如我们的标签化 RISC-V 研究需要花费约半年的时间来完成 FPGA 原型系统的构建,基于真实芯片设计的研究工作则鲜有听闻.在工业界,芯片的开发周期长达 2～3 年,其中设计和验证工作需要花费 1～2 年,投片需要花费约 1 年.如果流片失败,投入的时间和精力将会付诸东流,风险相当大[1].因此,如果有办法加快芯片设计的效率,实现芯片的敏捷开发,那么将会对学术研究和芯片产业带来巨大的影响.

伯克利的另一个研究团队在 2010 年的时候已经考虑到这方面的问题了,他们在 2012 年的 DAC 会议上发表了一门新的编程语言 Chisel[6] 来进行硬件的敏捷开发.Chisel 的主要目标是减少项目中的重复代码,提高代码密度,从而提升开发效率、代码的可读性和易维护性.编写 Chisel 代码后,用 Chisel 编译器将其编译成底层的 Verilog 代码(网表),可用于标准的 ASIC 和 FPGA 流程.需要说明的是,虽然 Chisel 支持传统硬件描述语言不具备的很多高级特性,但 Chisel 还是一门硬件构建语言,而不是高层次综合语言.硬件构建语言用于描述电路具体如何构建,而高层次综合则用于描述算法的流程.

3.1 信号整体连接

Chisel 丰富的类型系统使得我们可以很容易地修改类型的定义,在不修改引用该类型的代码(如通过该类型来定义的信号)的情况下,可以轻松对该类型的信号进行全局修改.此外,整体连接运算符"<>"会根据类型的定义,把相同类型的 2 组信号的相应成员信号一一连接,从而省去重复而且易出错的连线代码.在工程项目中,我们一般会在代码中的不同位置定义同种类型的信号,比如总线信号.在这种情况下,信号整体连接的特性能大幅减少项目中的重复代码.

在标签化 RISC-V 项目中,为了在 TileLink 总线上添加标签并实现携带传播,我们只需要添加 4 行 Chisel 代码.图 1 中以"+"为行首的代码为添加的代码.具体地,添加的代码首先定义一个宽度为 tlDsidBits 的新信号 *dsid*,然后把这个新信号加入到 TileLink 协议的元数据中.在类型系统的作用下,项目中定义的所有 TileLink 总线都会自动带上这个新成员信号 *dsid*.此外,由于项目中使用"<>"来连接 TileLink 总线,因此"<>"也会自动将新成员信号 *dsid* 连接起来.我们无需编写额外的代码,就已经实现了标签在 TileLink 总线上的携带传播了.

```
+trait HasDsid extends HasTileLinkParameters{
+val dsid = UInt(width = tlDsidBits)
+}
  class AcquireMetadata(implicit p: Parameters) extends
    ClientToManagerChannel
  with HasCacheBlockAddress
+with HasDsid
  with HasClientTransactionId
  with HasTileLinkBeatId

  /* An example of using "<>" */
  core(0).out <> arbiter.in(0)
```

Fig. 1 Implemention of labels over TileLink

图 1 在 TileLink 总线上添加标签的代码实现

这个例子展示了敏捷开发中的一个常见现象:需求变更时,可以对项目快速进行修改以实现新的需求.使用 Verilog 语言进行开发时,若要对总线的成员信号进行改动,工程师只能对项目中所有用到总线的模块端口逐一进行改动,同时还需要手工添加或移除相应成员信号的 assign 语句,这对 Verilog 工程师来说是一件非常麻烦的事情.SystemVerilog 的 interface 特性一定程度上也能实现类似 Chisel 中信号整体连接的功能,但它仍然有一些局限性,例如 modport 不能嵌套定义,使得我们无法从不同的 interface 中将相同的部分进一步抽象出来.Chisel 中的信号整体连接特性比 SystemVerilog 还要强大,这使得我们可以进一步减少重复的代码,实现"一改全改"的效果,从而提升项目的开发效率.

3.2 元编程

Chisel 支持基于 Scala 的元编程,可以借助 Scala 的特性抽象出多份相似的 Chisel 代码的共性部分,我们只需要维护一份共性代码,具体的 Chisel 代码可以通过对共性代码实例化得到,从而进一步减少冗余的代码.一个常见的例子是使用模板来实现队列原型.无论队列中的元素是何种类型,队列本身的

功能都是一样的.使用模板可以将队列本身的功能抽象成一个带类型参数的队列原型,然后在实例化时给出队列元素类型就可以得到一个该种元素类型的队列.通过这种方式,我们不必分别实现不同元素类型的队列,只需要维护一份队列原型即可.

图 2 展示了 Chisel 使用模板类 Queue 来抽象出队列原型的一个例子.行①表示这个模板类接收队列元素 *gen* 和队列项数 *entries* 两个参数,其中队列元素类型并未事先确定.行②定义模块的输入输出端口.行③定义队列的存储单元,它们均与队列元素类型有关.行④和行⑤分别定义队列的读写指针.剩余代码未列出,但其实现与传统硬件描述语言非常类似.有了 Queue 模板类,我们就可以通过它来定义各种元素类型的队列了.

```
① class Queue[T<:Data](gen:T,val entries: Int) extends
    Module(){
②   val io = IO(new QueueIO(gen, entries))
③   val ram = Mem(entries, gen)
④   val enq_ptr = Counter(entries)
⑤   val deq_ptr = Counter(entries)
⋮
㊸ }
```

Fig. 2 Example of template class in Chisel

图 2 Chisel 模板类例子

在标签化 RISC-V 项目中,在 TileLink 总线中添加标签之后,一个需要考虑的问题是,项目中有一些对 TileLink 请求进行缓冲的队列.我们希望标签能够随着请求一同经过队列,以实现携带传播的效果,这也许要求我们修改相关的代码.幸运的是,元编程的功能已经自动实现了这一效果.这是因为我们通过扩充 TileLink 元数据类型的方式加入标签,本质上修改了 TileLink 类型,但上述队列原型可适用于各种元素类型,从而无需修改相关的代码.而为了在 Verilog 中实现这一功能,针对代码中的所有相关队列,我们要么增加队列元素的宽度,要么增加一个新队列来专门对标签进行缓冲,同时还需要维护标签和请求的对应关系,十分繁琐.SystemVerilog 也支持模板,但相应的代码是不可综合的,更多的是用在测试激励的编写中.

3.3 面向对象编程

Chisel 可以使用面向对象编程的特性来提高硬件开发的效率,在这里我们介绍继承和重载.

3.3.1 继 承

面向对象编程中的继承特性允许我们将一些共同的代码特性通过一个父类抽象出来,达到减少冗余代码的效果,同时层次化的类型系统还可以使类型检查的过程更加严格,从而降低代码出错的可能性.具体地,我们在实现一个模块时,只需要从父类继承,就可以让该模块自动拥有父类定义的所有特性,从而避免在不同的模块中重复实现这些特性.

在硬件项目中,不少模块之间都有一些共同的特性,例如 cache 的不同替换算法都需要读出并更新历史状态,而对于带有总线接口的模块,总线相关的参数也非常相似.在这些情况下,使用继承特性可以有效地节省项目的代码量.

在标签化 RISC-V 项目中,我们需要编写一个生成随机地址的 TileLink 负载发生器,来对基于标签的令牌桶模块进行压力测试.图 3 展示了该负载发生器的 Chisel 实现.其中,行①定义了模块的名称 TileLinkTrafficGenerator,它从 TLModule 类继承.行②~⑤定义了该模块的输入输出端口,包括一组 TileLink 主端口 *out* 以及一个从令牌桶模块传过来的使能信号 *traffic_enable*.模块内部的具体实现未列出,但整个模块的代码量只有 20 行.

```
① class TileLinkTrafficGenerator(implicit p: Parameters) extends
    TLModule()(p){
②   val io = IO(new Bundle{
③     val out = new ClientTileLinkIO
④     val traffic_enable = Bool().asInput
⑤   })
⋮
⑳ }
```

Fig. 3 Example of inheritance in Chisel

图 3 Chisel 继承例子

从 TLModule 模块继承后,负载发生器模块会自动拥有 TLModule 所拥有的所有特性,这些特性是具有 TileLink 接口的模块所共同拥有的,包括总线的大量参数,如地址位宽、数据位宽等.这样,我们就可以在行③通过 1 行 Chisel 代码直接定义一个 TileLink 的主端口,而无需额外显式指定大量的总线参数,让代码功能一目了然,容易维护.相比之下,传统的硬件描述语言则难以实现类似的效果,例如若使用 Verilog 编写,就需要 40 行代码来声明 TileLink 主端口,然后还需要编写另外 40 行代码对端口中的每一个信号进行赋值操作,加上模块内部的具体实现,共需要约 200 行代码来编写此模块,是 Chisel 代码的 10 倍.在这种情况下,Verilog 代码就难以一目了然了.而 SystemVerilog 虽然支持继承,但功能有限,而且相应的代码不可综合[27],无法对硬件构建提供帮助.

3.3.2 重 载

面向对象编程的另一个特性是重载,重载可以提升代码的可读性.首先,Chisel 支持以函数的方式来对电路进行抽象(包括模块实例化),以达到复用的效果,同时可以把函数返回值当做相应电路的输出,直接作为 Chisel 表达式的一部分,而无需额外定义函数的输出信号.此外,Chisel 支持函数名重载,允许定义带有不同参数的多个同名函数,而且可以设置缺省参数,在调用函数时,Chisel 会在多个函数定义中自动选择类型匹配的函数定义.运算符重载则是把运算符当作一个特殊的函数,可以根据运算符左右两侧的变量类型来决定运算符的具体行为.

在标签化 RISC-V 项目中,我们需要在访存通路上添加一个延迟器来凸显二级缓存的效果,从而测试基于标签的路划分实现是否有效.图 4 的阴影部分展示了新增的代码.其中,原代码通过函数 *AXI4RAM*(),*AXI4Buffer*(),*AXI4Fragmenter*() 和 *AXI4MasterNode*(),分别实例化了 AXI4 接口的 SRAM、缓冲器、分片器以及主节点,并将它们的 AXI4 主从端口依次连接起来.此处项目对运算符":="进行了重载①,让运算符两侧节点的 AXI4 主从端口通过"<>"进行整体连接.为了在通路上添加延迟器,我们只需要添加 *AXI4Delayer*() 的调用,来实例化一个延迟为 150 周期的延迟器,并通过重载后的":="接入到访存通路中,代码功能一目了然,容易维护.相比之下,Verilog 和 SystemVerilog 中的函数、任务和运算符均不支持重载,无法实现上述效果,只能使用 module 来实例化这些模块,从而引入大量的连线代码.

```
val node = AXI4MasterNode(List(edge.master))
val sram = LazyModule(new AXI4RAM(…))
sram.node := AXI4Buffer() := AXI4Fragmenter() :=
  AXI4Delayer(0, 150) := node
```

Fig. 4 Example of overloading in Chisel

图 4 Chisel 重载例子

3.4 函数式编程

Chisel 支持使用函数式编程的特性来描述电路,可以编写更紧凑、可读性更好的代码.首先,Chisel 使用"容器"(collection)来抽象电路元素,容器中可以是信号、寄存器、端口、模块、映射等,或者是这些元素的复合.然后,Chisel 使用 map 算子对容器中的对象进行批量操作,操作可以是连接、归约、算术和逻辑运算、选择、实例化、函数调用、计算新映射等,或者是这些操作的复合,操作结果返回一个新容器.通过容器和 map 算子的组合,我们可以轻松地通过少量代码描述复杂电路.

在标签化 RISC-V 项目中,我们需要把 3.3.1 节中介绍的负载发生器接入到 Rocket Chip 中,来测试标签化令牌桶在极限情况下的效果,接入方式如图 5 所示.具体地,我们希望在第 2 到 n 个核心的数据通路中分别添加一个 2 选 1 的 TileLink crossbar,并接入相应的负载发生器,而第 1 个核心的数据通路则保持不变,从而达到让 $n-1$ 个负载发生器与第 1 个核心竞争 L1toL2 Network 模块入口带宽的效果.此外,我们还希望 n 是可配置的,取值可以是 2,4,8 等.

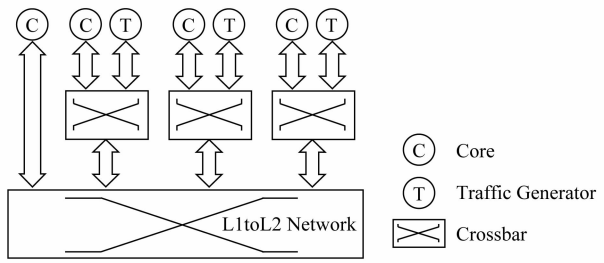


Fig. 5 Diagram of connecting traffic generators

图 5 负载发生器连接示意图

我们可以通过图 6 所示的 Chisel 代码实现这一功能.其中,行①的 *cachedPortsBefore Generator* 表示每个核心的 TileLink 主端口的集合,*take*(1)表示取出其中第 1 个核心的端口.行②的 *drop*(1)表示取出除去第 1 个核心之外剩余其他核心(即第 2 到 n 个核心)的端口,map 算子表示对这些端口迭代进行行③~⑨的操作.具体地,首先为每个端口取一个别名 p(行③),行④实例化一个负载发生器模块;行⑤将令牌桶的使能信号接入负载发生器;行⑥实例化一个 2 选 1 的 TileLink crossbar;行⑦将迭代中的端口 p 和负载发生器模块的主端口组织成一个列表,然后通过"<>"将 2 个端口分别接入到 crossbar 的 2 个输入口;行⑧将 crossbar 的输出口作为一次迭代操作的结果.这样,行②~⑨的结果就是第 2 到 n 个核心的数据通路中 crossbar 的输出口,连同第 1 个核心的端口构成的集合,取名为 *cachedPorts*.行⑩通过"<>"将 *cachedPorts* 接入到 L1toL2 Network 模块中,就能实现图 5 的效果.

① 实际上是 diplomacy[28] 语言的功能

```
① val cachedPorts = cachedPortsBeforeGenerator.take(1)++
②  cachedPortsBeforeGenerator.drop(1).map{
③    case p => {
④     val trafficGenerator = Module(new TileLinkTrafficGenerator()(p_alter))
⑤     trafficGenerator.io.traffic_enable := io.traffic_enable
⑥     val trafficGeneratorArb = Module(new ClientTileLinkIOArbiter(2)(p_alter))
⑦     trafficGeneratorArb.io.in <> List(p,trafficGenerator.io.out)
⑧     trafficGeneratorArb.io.out
⑨   }}
⑩ l1tol2net.io.clients_cache <> cachedPorts
```

Fig. 6 Example of functional programming in Chisel

图 6 Chisel 函数式编程例子

这个例子说明了 Chisel 确实是一门硬件构建语言,而不是高层次综合语言:容器中的对象和 map 算子的操作,都是可综合电路中的概念. Chisel 只是使用高级特性来方便地描述电路,而不是通过它们来描述算法. 而 Verilog 虽然有 for 和 generate 特性,但它们只能基于整数进行迭代,与 Chisel 中可以对任意对象批量进行任意操作的功能相比,Verilog 的迭代功能就非常有限了. 例如上述例子在 map 算子中使用了"<>"运算符,能实现进一步减少重复代码的效果,而 Verilog 和 SystemVerilog 均不支持函数式编程,难以实现类似效果.

3.5 开发语言对比小结

表 4 总结了 Chisel 和传统的硬件描述语言(Verilog 和 SystemVerilog)之间的对比. Chisel 支持信号整体连接,可以将类型相同的 2 组信号的相应成员信号自动连接起来;SystemVerilog 的 interface 特性也可以实现类似的效果,但有一定的局限性,如 modport 不能嵌套定义;而 Verilog 则不支持此功能,其类型(wire 和 reg)不具备语义,因此只能逐个信号地定义和连接. Chisel 支持元编程,包括模板类和模板函数,可以把功能近似但类型不同的类或函数进行抽象,进一步实现代码重用的效果;SystemVerilog 也支持模板的部分功能,但相应代码不可综合,无法对硬件构建提供帮助,而 Verilog 则不支持模板. Chisel 支持面向对象编程,通过继承可以使子类自动带上父类的特性,从而减少重复的代码,层次化的类型系统也可以使类型检查更加严格,降低代码出错的可能性,同时重载可以提升代码的可读性;SystemVerilog 不支持重载,虽然支持继承的部分功能,但和模板类似,相应的代码是不可综合的;而类型系统较弱的 Verilog 则完全不支持面向对象编程. Chisel 还支持函数式编程,可以将任意电路对象作为容器的元素,并通过 map 算子对这些对象批量进行任意操作,从而实现用少量代码描述复杂电路的效果;而 SystemVerilog 和 Verilog 均不支持函数式编程.

Table 4 Summary of Different Hardware Description Languages

表 4 不同硬件描述语言的特性总结

Feature	Chisel	SystemVerilog	Verilog
Bulk Connection	Yes	Yes, but with limitation	No
Meta-programming	Yes	Partially support and non-synthesizable	No
Object-oriented Programming	Yes	Partially support and non-synthesizable	No
Functional Programming	Yes	No	No

和传统的硬件描述语言相比,Chisel 的这些高级特性可以大大减少项目中的冗余代码,提高项目的开发效率,同时高密度的代码也提高了可读性,使得项目更容易维护. 正是 Chisel 语言的这些特性,使得它成为硬件敏捷开发的利器.

4 敏捷开发案例评估

我们将通过项目中的 2 个开发案例,分别从编码效率(开发耗时和代码量)和编码质量(性能、功耗和面积)这 2 方面,对分别以 Chisel 和 Verilog 为代表的敏捷开发模式和传统开发模式进行对比.

4.1 Chisel 与 Verilog 编码效率对比

我们团队曾经由于项目需要,期望尽快实现一个简单的共享二级缓存. 该二级缓存无需实现一致性协议的功能,只需要具有缓存功能即可,但需要集成到标签化 RISC-V 项目中并正确运行. 团队中的 2 人分别进行独立开发,具体情况如表 5 所示.

参与开发的其中一位是团队中的工程师,他在标签化项目的早期阅读并理解过 OpenSPARC T1

的二级缓存源代码,也修改过赛灵思系统缓存,在其中成功添加基于标签的路划分功能,具有丰富的缓存设计经验.这位工程师使用传统开发模式,选择Verilog语言来开发这个二级缓存,并决定从零开始搭建测试环境,不复用任何代码.他主要开发了6周,编写了约1 700行有效代码.遗憾的是,截至本文投稿为止,他开发的二级缓存模块仍然无法在标签化RISC-V项目上成功运行.

Table 5 Case Study of Implementing an L2 Cache

表5 L2 Cache开发案例对比

Metric	Engineer	Undergraduate
Experience	Skilled with OpenSPARC T1 and Xilinx System Cache	Finish assignment of CPU design with 9 months of Chisel development experiment
Language	Verilog	Chisel
Time	6 weeks	3 days
Effect	Still can not boot Linux	Can boot SMP Linux with Ethernet in DMA mode
Lines of Code	~1700	~350
Code Reuse	No	Yes

参与开发的另外一位人员是团队中的大四本科实习生,他做过CPU课程设计,并有9个月的Chisel开发经验,但从未设计过二级缓存.这位本科生使用敏捷开发模式,选择Chisel语言来开发这个二级缓存,并使用Chisel标准库来帮助设计,同时也复用标签化RISC-V项目的测试环境.经过了2天的设计和仿真验证,他就写出了一个可以支持多核Linux启动的二级缓存,有效代码量约350行,只有工程师编写代码的1/5.不过这个二级缓存一开始并不支持不完整的突发读写,导致DMA模式的以太网模块不能正确工作.一周后团队将这个情况反馈给他,他又额外花了一天时间改动了约50行有效代码[29],添加了对不完整突发读写的支持,并进行仿真验证,最终成功支持DMA模式的以太网模块正确工作.

这个案例充分展示了敏捷开发在项目中的优势:运用语言的各种高级特性、复用标准库中已经经过验证的模块来编写易读、易维护、高密度的代码,可以大大提升项目开发的效率.具体在这个案例中,敏捷开发模式的效率是传统开发模式的14倍!由于没有进行代码和环境的复用,这位工程师表示他花费了一半的时间在构建测试环境和编写并测试基本元件(如RAM、队列、仲裁器等)中,而使用Chisel的本科生并没有在这些事情上花费任何时间.不过即使把工程师花费的一半时间排除在效率比较的范围之外,敏捷开发模式的效率仍然是传统开发模式的7倍.实际上代码重用是敏捷开发的一个基本理念,代码的重用率越高,项目开发的效率就越高.

有趣的是,这位工程师后来提到,当时为了编写一份端口数量可配置的总线连接代码,他在generate特性的基础上运用了一些特殊技巧实现了这一功能,编写了约250行Verilog代码.但是工程师在编写过程中,由于连线和数字下标太多,并且需要顾及总线握手协议,他曾经因疏忽而导致2个连线错误的bug,花了约3天时间才发现并修复它们.这位工程师还表示,这部分代码的可读性其实并不好,使用Verilog实现这一功能实在太繁琐了,即使在代码中有相应注释,他在一周后也不能马上理解他使用的特殊技巧是如何工作的了.相比之下,若使用Chisel来实现类似功能,我们只需要编写2行代码即可,可读性好,而且几乎不会出现错误.此外,本科生实际上也是在一周后重新回头阅读并修改自己编写的二级缓存,但他仍然在一天内成功修复了问题,这说明代码的可读性对项目维护来说是非常重要的.

4.2 Chisel与Verilog编码质量对比

考虑上述案例展示的2个设计,虽然它们的需求是一致的,但不同的开发人员可能会采用不同的实现方式,导致编码质量的可比性不强.为了进一步对比Chisel和Verilog的编码质量,我们找到团队中的另一位没有Chisel开发经验的大四实习本科生,让他来把上述Verilog代码中的部分关键模块翻译成功能等价的Chisel代码.我们在项目中提供了一些测试,用于验证翻译结果的等价性.

4.2.1 逐句翻译

由于这位本科生一开始并没有Chisel的开发经验,他需要从零开始学习Chisel,并选择最简单的翻译方式:逐句翻译,而不使用Chisel的高级特性.这位本科生表示,他一开始觉得Chisel代码比较难读懂,但是学习并逐句翻译Chisel代码的过程中,他也逐渐感受到Chisel的方便之处,例如丰富的标准库、方便的数据类型系统及其转化机制、简洁的时序逻辑编码风格等,这些特性让他对Chisel有了新的认识.

我们用Chisel编译器把翻译后的Chisel代码编译成Verilog代码(网表),然后对其以及工程师编写的Verilog代码分别进行评估.我们在Vivado 2017.01中,使用xc7v2000tfhg1716-1型号的FPGA,

在 125 MHz 的时钟频率下进行评估,结果如表 6 中第 2,3 列所示.为了评估设计的性能,我们展示了时序报告中的最差负时序余量(worst negative slack,WNS),并将其换算成可运行的最高时钟频率,结果显示 Verilog 和 Chisel 最高分别可运行在 135.814 MHz 和 136.388 MHz 的时钟频率下,性能非常接近.而两者的功耗则分别为 0.770 W 和 0.749 W,和 Verilog 相比,Chisel 的功耗节省了 2.73%.而为了展示面积开销,我们给出 2 份设计各自消耗的查找表(lookup table, LUT)和触发器(flip-flop, FF)数量.我们对 LUT 的消耗分成逻辑和存储 2 部分来统计,和 Verilog 代码相比,Chisel 代码多消耗了 13.14%的 LUT 逻辑,但节省了 29.62%的 LUT 存储,这是因为 Chisel 标准库提供了 RAM 相关的基本元件,它与工程师实现的 RAM 被 Vivado 分别映射成 RAM64M 和 RAM64X1D 这 2 种不同的原语[30],其中 RAM64M 会消耗更多的 LUT 逻辑,但节约大量 LUT 存储.对于 FF,Chisel 比 Verilog 节省了 14.72%,这是因为 Chisel 编译器对代码进行了更进一步的优化.从代码量上看,即使是逐句翻译,Chisel 也比 Verilog 节约了 23.95%的代码量,这是因为:1)和 Verilog 的 generate 特性相比,基于 Scala 的元编程特性编写出的代码更加紧凑;2)在 Chisel 中编写时序逻辑无须像 Verilog 那样声明 always 块;3)Chisel 代码不会产生锁存器(latch),无须像 Verilog 那样补全 if 和 switch 的所有分支.

Table 6 PPA and LoC Comparison of Chisel and Verilog

表 6 Chisel 和 Verilog 的性能、功耗、面积和对比

Metric	Verilog	Chisel	Chisel-opt
WNS/ns	0.637	0.668	1.511
Max Period/MHz	135.814	136.388	154.107
Power/W	0.770	0.749	0.749
LUT Logic	5676	6422	2594
LUT RAM	1796	1264	1492
FF	4266	3638	747
Lines of Code	618	470	155

逐句翻译方式的整体评估结果说明,使用 Chisel 开发不但节省了代码,编码质量也和 Verilog 非常接近,在部分指标上甚至优于 Verilog.

4.2.2 使用 Chisel 高级特性

这位本科生对 Chisel 上手之后,我们让他使用 Chisel 的高级特性对代码进行重构,包括使用 Chisel 标准库来实例化 RAM 和队列等基本元件,同时使用第 3 节提到的信号整体连接、元编程、面向对象编程和函数式编程来节省代码量.在使用这些高级特性的过程中,这位本科生逐渐认识到 Chisel 的更多好处,他表示:使用 Chisel 标准库之后,因为大部分异步通信被统一封装为 Decoupled 模板类,为了能使用"<>"进行信号整体连接,自然就会尝试从 Bundle 类继承来定义模块端口,进而也会考虑使用函数式编程中的 map 算子对 Vec 数组进行简便的连接.这些好处让他体会到"Chisel 高级特性的一以贯之,相辅相成".

我们对重构之后的 Chisel 代码进行评估,结果如表 6 中第 4 列(Chisel-opt)所示.令人惊讶的是,和 Verilog 相比,Chisel-opt 的 LUT 逻辑节省了 54.30%,而 FF 则节省了 82.49%!这是因为和工程师编写的基本元件相比,Chisel 标准库提供的基本元件更成熟,能使用更少的资源实现相同的功能.由于资源的大幅节省,Chisel-opt 的性能也得到了进一步的提升.具体地,可运行的最高时钟频率提升到 154.107 MHz,与 Verilog 相比提升了 13.47%.除此之外,Chisel 的高级特性使得 Chisel-opt 的代码量更加精简,和 Verilog 相比,节省了 74.92%的代码量.

有趣的是,这位工程师一开始看到 Chisel-opt 的评估数据时,并不相信这一结果,甚至怀疑是本科生的代码编写错误,导致 Vivado 对代码进行了非预期的优化,才使得评估结果大幅优于 Verilog.但 Chisel-opt 确实通过了工程师亲自编写的所有仿真测试,工程师不得不相信 Chisel-opt 确实可以正确工作.但当他看到 Chisel-opt 生成可读性较差的网表级 Verilog 代码时,仍然不敢相信,并先后提出"这样子生成的 Verilog 代码,最后的结果应该是挺差劲的才对"、"也许是 Chisel 生成的代码太乱了,Vivado 正巧才匹配上一些复杂的原语"、"Chisel 也许对各种型号的 FPGA 原语有专门的优化"等各种猜测.工程师对 Vivado 报告进行详细的分析,最后发现是自己在 RAM 中实现的一个性能优化特性消耗了大量的 LUT 逻辑,把这个特性去掉之后,Verilog 消耗的 LUT 逻辑就减少到与 Chisel-opt 相当的水平,但这会给当前的 Verilog 设计带来额外的延迟,使得仿真测试的性能数据反而不如 Chisel-opt.不过关于 FF 的大量差异,工程师最后仍然找不到可以令他信服的原因.经过这次详细分析,工程师最后也不得不相信,"使用 Chisel 开发并不会引入明显的资源开销","除非 FPGA 工程师直接调用原语,不然正常情况下只会跟 Chisel 的资源消耗持平,

或者反而消耗更多资源”,甚至表示“如果 ASIC 也是这样的趋势,Chisel 肯定是下一代 HDL 的强力竞争者”.

这个案例很好地展现了一个敏捷开发的例子:一个本科生的 Chisel 新手,可以在更短的时间内编写更少的代码,编码质量就能达到和工程师相当的水平,甚至还可以超越工程师.即使编码质量与传统开发有 20%的差距,敏捷开发仍然展现了其节省人力和时间的价值:能快速构建一个可以工作的原型,对项目开展来说是非常有意义的.从这点来看,敏捷开发确实大大降低了硬件开发的门槛.

5 改进与展望

诚然,当前这些项目也有不足之处.例如由于 Rocket Chip 的迭代速度过快,基于 Rocket Chip 的研究项目需要花费一定的精力来跟进主线的功能,若跟进后发现主线的改动与研究项目的内容有所冲突,还需要花费额外的精力去解决这些冲突.我们曾经多次参加 RISC-V 国际研讨会,会上不少同行都表示对此感到困扰.同时 Rocket Chip 项目的版本管理目前也有待完善,截止到本文投稿为止,Rocket Chip 在 github 上的开源仓库虽然受到广泛关注,但仍然没有发布任何稳定的发行版本[31],这对于初次接触 Rocket Chip 项目、想寻找一个稳定版本的开发者来说并不友好.

相对来说,对 Chisel 表达建议的声音就更多了.首先 Chisel 的学习曲线比较陡峭,不少硬件开发者比较少接触面向对象编程,对其编程思维在硬件开发中的应用缺少清晰的认识,而函数式编程对硬件开发者来说就更是遥不可及了.然而关于 Chisel 语言的资料也较为缺乏,不少开发者因为找不到合适的学习资料而处于观望状态,面对充满 Chisel 高级特性的 Rocket Chip 项目更是望而却步.此外,目前 EDA 工具对 Chisel 的支持并不完善,由于 Chisel 生成的 Verilog 代码属于低层次的网表级别,并不是给硬件工程师阅读的,让他们使用 EDA 工具对这样的 Verilog 代码进行验证和调试是非常困难的.

事实上,对开源芯片设计来说,开源 EDA 工具也是一个很重要的部分.我们对目前的开源 EDA 工具进行了简单的调研,发现开源 EDA 工具面临 3 个挑战:1)目前开源的器件库较少且工艺较老,如 Qflow 工具支持的最先进器件库为 0.18 μm[32],这限制了芯片的实用性;2)开源 EDA 工具的可靠性相对较弱,遇到问题时也不易寻求帮助,这是因为这些工具未被广泛使用,社区较小;3)目前缺少完整高效、灵活可配置且简单易用的开源 EDA 工具链.

但是,芯片敏捷开发的趋势是不可阻挡的.据了解,最近一两年越来越多大型 IT 企业传出了拥抱 RISC-V 的消息[33-34],甚至一些初创企业也尝试使用 Chisel 进行项目开发,并表示效果不错.因此只要给予充分的时间,上述问题是可以解决的.甚至上述部分问题已经被列入了 DARPA 牵头的电子复兴计划[35]将要解决的问题列表中.

本文中分享的案例侧重于提高前端的编码效率,但这只是敏捷开发所倡导的其中一项原则.伯克利研究团队曾经根据自己的经验提出了硬件敏捷开发宣言[3],包括:1)优先开发未完成但容易改造的原型,而非构建功能齐全却难以扩展的模型;2)优先组建灵活协作的团队,而非强调各司其职的分工;3)优先完善工具和生成器,而非改进独立的设计实例;4)优先拥抱变化,而非遵循计划.事实上,传统的观点认为硬件开发需要使用传统的开发模型,是因为硬件开发的特点与软件开发有较大差异.这些差异包括硬件的设计和验证周期更长、完整的硬件设计流程需要更多更专业的技能(总体架构设计、微结构设计、前端 RTL 设计和验证、后端物理设计和验证)、硬件成品无法进行频繁的更新等.但伯克利研究团队认为,正是因为这些差异,我们更需要借鉴软件开发的经验来提升硬件开发的效率.例如通过高效的工具和生成器来实现代码复用,缩短硬件开发周期,节省验证成本;培养全栈的工程团队来节省沟通成本;通过管理多个未完成原型的版本来拥抱需求的变化.这些做法背后的原理,和软件敏捷开发的理念是非常相似的.

伯克利研究团队的实践已经展示了敏捷开发的惊人效果:他们在 5 年时间内进行了 11 次投片[3],平均每 5~6 个月完成一款芯片的设计,与传统的 2 年设计一款芯片相比,设计效率提高了 4~5 倍.最近睿思芯科也公布了他们设计的第 1 款芯片,他们使用 RISC-V 相关的基础设施进行芯片的敏捷开发,从零开始到设计完成,只用了 7 个月的时间[36],是传统芯片设计效率的 3 倍.

这些例子说明了,通过敏捷开发来将芯片设计门槛降低几个数量级,是有可能实现的.我们即将迎来芯片设计的黄金时代,希望更多的有识之士能加入到这股潮流之中.

6 总 结

本文通过分享标签化 RISC-V 的项目经验,分别从设计开源度、定制灵活性、生态完整性以及社区活跃度 4 个方面,对 SPARC V9,MicroBlaze 和 RISC-V 三种指令集进行比较,说明了开放又活跃的指令集及其开源的微结构设计,是芯片敏捷开发的必要条件.同时本文也介绍了 Chisel 中有利于敏捷开发的特性,将这些特性与传统的硬件描述语言进行比较,展示了 Chisel 代码的简洁性、易读性以及易维护性,从而对项目的敏捷开发提供帮助.最后,本文还将敏捷开发和传统开发的效率和质量进行对比,结果显示:敏捷开发能在编码效率提升一个数量级的同时,达到与传统硬件开发模式相当甚至更优的性能、功耗与面积.

参 考 文 献

[1] Olofsson A. Intelligent design of electronic assets (IDEA) & posh open source hardware (POSH)[R]. Mountain View, CA: DARPA, 2017

[2] Waterman A, Lee Y, Patterson D, et al. The RISC-V instruction set manual, volume I: User-level ISA, version 2.1[R]. Berkeley, CA: University of California, Berkeley, 2016

[3] Lee Y, Waterman A, Cook H, et al. An agile approach to building RISC-V microprocessors[J]. IEEE Micro, 2016, 2(36): 8-20

[4] Asanović K, Avizienis R, Bachrach J, et al. The Rocket Chip generator, UCB/EECS-2016-17 [R]. Berkeley, CA: University of California, Berkeley, 2016

[5] Terpstra W, Waterman A, Cook H, et al. Rocket Chip generator github repository[EB/OL]. [2018-11-23]. https://github.com/freechipsproject/rocket-chip

[6] 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

[7] Yu Zihao, Huang Bowen, Ma Jiuyue, et al. Labeled RISC-V: A new perspective on software-defined architecture[C] // Proc of the 1st Workshop on Computer Architecture Research with RISC-V. 2017 [2018-11-23]. https://github.com/carrv/carrv.github.io/blob/master/2017/papers/yu-labeled_riscv-carrv2017.pdf

[8] Yu Zihao, Wang Huizhe, Liu Zhigang, et al. Labeled RISC-V github repository[EB/OL]. [2018-11-23]. https://github.com/LvNA-system/labeled-RISC-V

[9] Weaver D, Germond T. The SPARC architecture manual, version 9, SAV09R1459912 [R]. San Jose: SPARC International, Inc., 1994

[10] Xilinx Inc.. MicroBlaze Processor Reference Guide: UG984 (v2018.2)[EB/OL]. (2018-06-21)[2018-11-23]. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug984-vivado-microblaze-ref.pdf

[11] IEEE Computer Society. IEEE Standard for Verilog Hardware Description Language[S]. Piscataway, NJ: IEEE, 2006

[12] IEEE Computer Society. IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language[S]. Piscataway, NJ: IEEE, 2005

[13] Asanović K. Instruction sets want to be free! [EB/OL]. [2018-11-23]. https://riscv.org/wp-content/uploads/2017/05/Mon1045-Why-RISC-V-ISAs-Want-to-be-Free.pdf

[14] Asanović K, Patterson D. Instruction sets should be free: The case for RISC-V, UCB/EECS-2014-146 [R]. Berkeley, CA: University of California, Berkeley, 2014

[15] Celio C, Chiu P, Nikolic B, et al. BOOM v2: An open-source out-of-order RISC-V core, UCB/EECS-2017-157 [R]. Berkeley, CA: University of California, Berkeley, 2017

[16] Celio C. BOOM github repository[EB/OL]. [2018-11-23]. https://github.com/riscv-boom/riscv-boom

[17] Bao Yungang. Some thoughts on India's adoption of RISC-V as its national instruction set [J]. Communications of the CCF, 2018, 14(1): 38-44 (in Chinese)
(包云岗. 关于 RISC-V 成为印度国家指令集的一些看法[J]. 中国计算机学会通讯, 2018, 14(1): 38-44)

[18] Bao Yungang, Wang Sa. Labeled von neumann architecture for software-defined cloud[J]. Journal of Computer Science and Technology, 2017, 32(2): 219-223

[19] Ma Jiuyue, Sui Xiufeng, Sun Ninghui, et al. Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (PARD)[C]// Proc of the 20th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2015: 131-143

[20] Oracle Inc.. OpenSPARC T1 and specifications [EB/OL]. [2018-11-23]. https://www.oracle.com/technetwork/systems/opensparc/opensparc-t1-page-1444609.html

[21] Wikipedia Community. MicroBlaze [EB/OL]. (2018-11-01)[2018-11-23]. https://en.wikipedia.org/wiki/MicroBlaze

[22] GCC team. GCC 4.6 release series changes, new features, and fixes[EB/OL]. [2018-11-23]. https://gcc.gnu.org/gcc-4.6/changes.html#microblaze

[23] Xilinx Inc.. System Cache v4.0, LogiCORE IP Product Guide (PG118)[EB/OL]. (2017-04-05)[2018-11-23]. https://www.xilinx.com/support/documentation/ip_documentation/system_cache/v4_0/pg118-system-cache.pdf

[24] Xilinx Inc.. LogiCORE IP AXI Interconnect v2.0, Product Guide for Vivado Design Suite (PG059)[EB/OL]. (2013-03-20)[2018-11-23]. https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_0/pg059-axi-interconnect.pdf

[25] Matthews E, Shannon L, Fedorova A. Shared memory multicore MicroBlaze system with SMP Linux support [J]. ACM Transactions on Reconfigurable Technology and Systems, 2016, 9(4): 26:1-26:22

[26] SiFive Inc.. SiFive TileLink Specification, Version 1.7 [EB/OL]. (2017-08-21) [2018-11-23]. https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf

[27] Stack Overflow Community. What SystemVerilog features should be avoided in synthesis? [EB/OL]. [2018-11-23]. https://stackoverflow.com/questions/20312810/what-system verilog-features-should-be-avoided-in-synthesis

[28] Cook H, Terpstra W, Lee Y. Diplomatic design patterns: A TileLink case study [C] // Proc of the 1st Workshop on Computer Architecture Research with RISC-V. 2017 [2018-11-23]. https://github.com/carrv/carrv.github.io/blob/master/2017/papers/cook-diplomacy-carrv2017.pdf

[29] Liu Zhigang. Fix burst length support in L2 cache [EB/OL]. [2018-11-23]. https://github.com/LvNA-system/labeled-RISC-V/commit/3f6a6f623e627dab076601bbbd82e616e7dea71e

[30] Xilinx Inc.. Vivado Design Suite 7 Series FPGA Libraries Guide (UG953) [EB/OL]. (2012-07-25) [2018-11-23]. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug953-vivado-7series-libraries.pdf

[31] Terpstra W, Waterman A, Cook H, et al. Rocket Chip Generator github repository release page [EB/OL]. [2018-11-23]. https://github.com/freechipsproject/rocket-chip/releases

[32] Open Circuit Design. Qflow 1.1: An open-source digital synthesis flow [EB/OL]. (2017-08-24) [2018-11-23]. http://opencircuitdesign.com/qflow/index.html

[33] Song Wei, Huang Bowei, Guo Xiongfei. RISC-V Bi-weekly Report [EB/OL]. (2017-12-07) [2018-11-23]. https://cnrv.io/bi-week-rpts/2017-12-07 (in Chinese)
(宋威, 黄柏玮, 郭雄飞. RISC-V 双周简报 [EB/OL]. (2017-12-07) [2018-11-23]. https://cnrv.io/bi-week-rpts/2017-12-07)

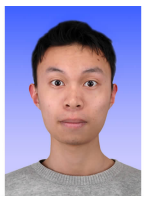
[34] Song Wei, Huang Bowei, Guo Xiongfei. RISC-V Bi-weekly Report [EB/OL]. (2018-01-18) [2018-11-23]. https://cnrv.io/bi-week-rpts/2018-01-18 (in Chinese)
(宋威, 黄柏玮, 郭雄飞. RISC-V 双周简报 [EB/OL]. (2018-01-18) [2018-11-23]. https://cnrv.io/bi-week-rpts/2018-01-18)

[35] DARPA. DARPA electronics resurgence initiative [EB/OL]. [2018-11-23]. https://www.darpa.mil/work-with-us/electronics-resurgence-initiative

[36] RISC-V Community. Sohu Article: OURS Technology Released 64-bit AI SoC Based on RISC-V [EB/OL]. [2018-11-23]. https://riscv.org/2018/11/sohu-article-pygmy-soc/

Yu Zihao, born in 1991. PhD candidate in the Institute of Computing Technology, Chinese Academy of Sciences. His main research interests include computer architecture and operating system.

Liu Zhigang, born in 1995. Master candidate in the Institute of Computing Technology, Chinese Academy of Sciences. His main research interests include computer architecture and operating system. (liuzhigang@ict.ac.cn)



Li Yiwei, born in 1996. Undergraduate student in the University of Chinese Academy of Sciences. His main research interests include computer architecture and machine learning. (liyiwei15@mails.ucas.ac.cn)

Huang Bowen, born in 1992. Assistant engineer in the Institute of Computing Technology, Chinese Academy of Sciences. His main research interests include micro-architecture, especially cache subsystem. (huangbowen@ict.ac.cn)

Wang Sa, born in 1986. PhD, assistant professor. Member of CCF. His main research interests include operating systems, distributed systems, and virtual machines. (wangsa@ict.ac.cn)

Sun Ninghui, born in 1968. Professor and PhD supervisor. His main research interests include computer architecture, high performance computing and distributed operating system. (snh@ict.ac.cn)

Bao Yungang, born in 1980. Professor and PhD supervisor. Member of CCF. His main research interests include computer architecture, operating system and system performance modeling and evaluation.