清 华 大 学

综 合 论 文 训 练

题目：基于 Chisel 实现 RISC-V 指令集标量密码学扩展

系　　别：交叉信息研究院

专　　业：计算机科学与技术（计算机科学实验班）

姓　　名：郑鉉壬

指导教师：高鸣宇　助理教授

2022 年 6 月 25 日

中文摘要

密码学运算已经成为了现代应用中不可或缺的一部分，对密码学运算的优化也是现代应用性能优化的关注重点。相对于单独的软件或硬件优化，两者的协同优化往往会带来更好的适用性和扩展性，能在有限的硬件代价中最大化对软件的加速。这种协同优化通常被应用于硬件与软件间的桥梁——指令集的设计之上。

RISC-V 作为一个新兴的开放标准指令集架构，也在 2021 年底正式标准化了标量密码学扩展指令集。该扩展指令集中包括密码学常用的位运算、NIST 定义的高级加密标准（AES）和安全哈希函数（SHA256 和 SHA512）、商密定义的分组密码（SM4）和杂凑函数（SM3）等。利用这些扩展指令，现实世界大量软件中的密码学模块将获得潜在加速的机会。

Chisel 是一个基于 Scala 的硬件描述语言。它拥有高度参数化的特性，从而在电路设计方面表现出了极大的灵活性，为实现可配置的硬件提供了可能。与此同时，RISC-V 生态与 Chisel 联系紧密，使用 Chisel 实现相应单元能较为便捷地接入现有的 RISC-V 生态之中。

本文基于 Chisel 的参数化特性，实现了同时适用于 RISC-V 64 位和 32 位架构的支持标量密码学扩展指令集的硬件电路单元，并将其集成在 Rocket RISC-V 核心的流水线中；同时，利用上述标量密码学扩展指令，本文为 OpenSSL 编写了 AES、SM4、SM3 的优化软件实现。

在硬件设计中，本文实现的标量密码学扩展指令集电路单元可在 64 位架构中为 AES 和 SM4 带来最高 10 倍和 5 倍的性能提升，在 32 位架构中为 AES 和 SM4 带来最高 4 倍和 3.7 倍的性能提升，为 SHA256、SHA512 和 SM3 等哈希函数提供 1.5 倍到 2.9 倍的性能提升。其面积代价仅约为原有实现中的一个硬件乘法除法器。

在软件实现中，RISC-V 标量密码学指令集扩展涉及的算法同时利用了常规和扩展指令集。相较于独立硬件加速器的设计思路，扩展指令集在原理上最大化重复利用了已有的核心流水线运算单元，从而减少了面积消耗以及为完整算法执行提供了较小的延迟。上述结果共同体现了扩展指令集能在有限的硬件代价中提供可观的软件加速的优势，这种对资源的重复利用也能体现 RISC 的设计精神。

关键词：RISC-V 指令集架构；Chisel 硬件描述语言；硬件密码学加速

ABSTRACT

Cryptography has become an indispensable part of modern applications, and the acceleration of cryptographic operations has become the focus of performance optimization. Instead of optimizing software or hardware alone, hardware-software co-design often offers better applicability and extensibility, and could maximize the acceleration of the software within limited hardware budget. This kind of co-design can often be embodied by the design of instruction set architecture (ISA), the bridge between hardware and software.

As an emerging open-standard ISA, RISC-V has ratified the scalar cryptography extension in Autumn, 2021. This ISA extension contains bit-manipulation instructions for cryptography, special instructions for AES and SHA defined by NIST, and SM4 and SM3 defined by ShangMi. With these instructions, many software in the real world could be potentially accelerated.

Chisel is a Hardware Description Language (HDL) based on Scala. With its highly parameterizable features, it shows great flexibility in designing digital circuits, thus configurable hardware units could be designed by it. At the same time, the RISC-V ecosystem is closely connected with Chisel, therefore hardware units implemented in Chisel can be easily integrated into the existing ecosystem.

Based on the parameterizable features of Chisel, this thesis implemented a set of hardware units that support the scalar cryptography extension for both RISC-V 64 and RISC-V 32, and integrated them into the pipeline of the Rocket core. In the meantime, with the instructions from the scalar cryptography extension, this thesis programmed accelerated AES, SM4 and SM3 functions for OpenSSL, a widely used cryptography software library.

With the accelerated software functions, the hardware units can offer up to 10X and 5X speedup for AES and SM4, respectively, in RISC-V 64, and 4X and 3.7X in RISC-V 32. It can also offer 1.5X to 2.9X speedup for hash functions like SHA256, SHA512 and SM3. The hardware cost of these hardware units is merely comparable to one multiplier and divider in the original design.

Keywords: RISC-V ISA; Chisel HDL; Cryptographic hardware acceleration

目　　录

第 1 章　简介

密码学运算已经成为现代应用中不可或缺的一部分。无论是网络通信，例如互联网安全协议（IPSec）[1]、传输层安全性协议（TLS）[2] 以及被广泛应用[3-4]的超文本传输安全协议（HTTPS）[5]，还是本地存储，例如 Windows 操作系统全盘加密软件 BitLocker[6]、Linux 操作系统的硬盘加密软件 LUKS[7]，其中都有对称密码学、哈希函数、公钥密码学等大量密码学工具的参与。

大量的密码学操作为现代计算机带来了更多性能上的挑战。例如，相比于传统的超文本传输协议（HTTP）[8]，现代的超文本传输安全协议（HTTPS）[5] 在原有处理的基础上，还需要进行验证证书签名、密钥交换、对数据的加密与解密等操作，其中涉及到运算模式较复杂的对称密码学操作，例如 AES[9] 或 SM4[10] 中的 SBox，以及需要高精度大整数运算的公钥密码学操作，例如 RSA4096 需要对 4096 位大整数进行乘法取模[11]。

面对这一挑战，现有系统往往会在算法、软件乃至硬件等各个层面上进行优化。算法上，相对于使用 AES，部分系统[12]选择采用 ChaCha20[13] 作为其对称密钥学工具；相对于使用 RSA 作为公钥密码学的操作，现代系统[14]也在更加广泛地使用性能更为优越[15]的椭圆曲线密码学（ECC）[16]。软件上，常用密码学相关操作往往会被集成为高性能的密码学软件库[17]。此类软件库会利用硬件提供的通用特性进行软件上的优化，例如利用矢量化 SIMD 运算等。硬件上，现代硬件中也有密码学专用电路的出现。一种做法是实现专用密码学运算加速器[18]，作为现代计算机的扩展卡支持相关操作；另一种更为通用的做法是在 CPU 中加上专用电路[19]，并通过在软硬件接口——指令集架构——里添加相关密码学加速指令，将此专用电路提供给软件进行调用。

当前大多数主流指令集架构都已经添加了常用密码学运算加速指令。对于 x86 架构，Intel 公司提出了高级加密标准指令集（AES-NI）以对 AES 算法进行加速[20]。Intel 也提供了无进位乘积指令（CLMUL）[21]，主要用于 AES-GCM 的加速。对于 ARM 架构，ARMv8-A 指令集中也增加了密码学加速指令，处理 AES 和 SHA-2 的加速[22]。

不同于主流的商业化指令集架构，RISC-V 指令集架构自提出时即为开放架构的指令集[23]，而其开放特性使得其更易为人接受、使用、参与乃至贡献。与此同时，与 RISC-V 架构的开放属性相伴生，学术界与工业界实现了大量的开源处理

器[24-25]和开源软件工具链。这些共同构成了 RISC-V 的开放生态。

RISC-V 标量密码学扩展由社区讨论并贡献而来[26]。该扩展涵盖了密码学中常用的位运算，例如循环移位和无进位乘积；同时包含了 NIST 定义的高级加密标准（AES）[9]和安全哈希函数（SHA256 和 SHA512）[27]；以及商密定义的分组密码（SM4）[10]和杂凑函数（SM3）[28]。

Chisel 作为一个基于 Scala 的硬件描述语言，基于其高度参数化的特性，它为现有 RISC-V 开源处理器的实现提供了便捷的编程接口[24-25]。

本文基于 Chisel 的参数化特性，实现了同时适用于 RISC-V 64 位和 32 位架构的支持标量密码学扩展指令集的硬件电路单元，并将其集成在 Rocket RISC-V 核心的流水线中。现有的 RISC-V 标量密码学扩展指令集的硬件实现中，要么是通过 Verilog 实现，具备重复代码较多、难以维护等缺点[29]，要么只实现了 RISC-V 64 位架构的部分[25]。相对于现有实现，本文所提出的实现在设计上更为简洁，在功能上更为完备。在性能上，本文的电路实现可在 64 位架构中为 AES 和 SM4 带来最高 10 倍和 5 倍的性能提升，在 32 位架构中为 AES 和 SM4 带来最高 4 倍和 3.7 倍的性能提升，为 SHA256、SHA512 和 SM3 等哈希函数提供 1.5 倍到 2.9 倍的性能提升。其面积代价仅约为原有实现中的一个乘法除法器。

与此同时，利用上述硬件加速的标量密码学扩展指令，本文为 OpenSSL[17] 编写了 AES、SM4 和 SM3 的优化软件实现。相对于一些独立的只为 RISC-V 服务的密码学软件库[30-31]，基于 OpenSSL 实现 RISC-V 密码学软件库能为用户提供与其他指令集架构一致的接口，使得所有使用 OpenSSL 的软件（例如 Python、Nginx、OpenSSH 等，可达数百之多[32]）能无缝使用 RISC-V 标量密码学扩展，而不需要单独适配。

本文还详细讨论了 RISC-V 标量密码学指令集扩展的精巧设计带来的硬件与软件上的收益和影响。在硬件上，多个指令的数据通路能达到共享。例如 AES 的加密与解密过程是互逆的，运算顺序在直觉上并不一样。而在该扩展指令集的设计中，能做到加密与解密的运算顺序相同，从而在电路上合并数据通路，减小面积代价。与这种特殊的指令和电路相对应，在密码学软件中只需要做少量的处理，即可使用这些指令实现完整的密码学算法。相对于独立的硬件加速器设计思路（即将密码学算法全部交给独立的加速器进行运算），该扩展指令集在设计时利用了常规指令来进行算法中的其他普通运算，如异或和访存等。这样可以最大化重复利用已有的核心流水线运算单元，减少面积代价，以及为完整算法的执行提供较小的延迟。同时，这种利用常规指令的行为也体现了 RISC 的设计精神。

第 2 章　背景介绍

2.1　RISC-V 指令集架构

RISC-V 指令集架构由安德鲁·沃特曼（Andrew Waterman）等人在 2010 年附近提出。安氏在他的博士学位论文[23]中提到，现有的指令集都是商业指令集，并不开放，为芯片事业的发展带来了一定的阻碍；与此同时，现有的指令集由于其历史悠久，在发展过程中几经曲折，存在不少历史遗留问题。在此情况下，一个开放的新的指令集架构既能供更多人自由使用，也能避免较多的历史遗留问题。

该指令集基于加利福尼亚大学伯克利分校一直在发展的精简指令集（RISC）。精简指令集意味着单个指令只做最简单的操作，这与复杂指令集（CISC）相对应。加利福尼亚大学伯克利分校之前在精简指令集上一共作出了四个版本，分别是 RISC-I, RISC-II, SOAR 和 SPUR，而安氏的版本为第五版，所以该指令集取名为 RISC-V。RISC-V 也通常会被缩写成 RV。

他在文中提到，这个指令集在设计时既考虑到用于教育和科研，也考虑到适用应用于工业场景。为了满足这两种需求，RISC-V 指令集在设计时依据功能不同拆分为不同的扩展，其中 I 为基本整数指令集，包含指令较少，能支撑最基础的系统；其余还有诸如整数乘除法扩展（M），浮点数扩展（F 和 D），原子操作扩展（A），压缩指令扩展（C）等，足以供给通用计算机使用；而为了一些需求更为精细的场景，例如密码学加速，通过 RISC-V 社区的讨论和贡献[26]，最终确立了标量密码学扩展（Zk）。与此同时，考虑到不同场景所需的性能不同，处理器所需的位数也不相同，RISC-V 也设计了不同的指令集变种，为此定义了面向嵌入式领域的 RV32E 和 RV32I、面向现代高性能处理器的 RV64I 和为未来预留的 RV128I。

2.2　RISC-V 标量密码学扩展

RISC-V 标量密码学扩展（Zk）[26]作为一个 RISC-V 扩展，意在为 RISC-V 提供一定的密码学加速能力，在设计时要考虑到在最小化硬件设计的成本的同时覆盖大部分的应用场景。为此，该扩展并没有单独的寄存器，而是复用了处理器中已有的通用寄存器，同时，该扩展包含了密码学常用的位运算操作（命名为 Zbk）、NIST 定义的对称密码学和哈希函数（命名为 Zkn），以及商密定义的分组密码和杂凑函数（命名为 Zks）。

Zbk 扩展分为三个部分，一个是位操纵指令（命名为 Zbkb），包括循环移位、逻辑与否、逻辑或否、逻辑异或否、位打包、位调换顺序等指令；另一部分是无进位乘积指令（Zbkc）；还有一部分是逐字节/逐半字节排列（Zbkx），即按照其中一个寄存器的值逐个字节/半字节作为索引查询另一个寄存器的字节/半字节以得到结果。

Zkn 扩展分为两个部分，一个是 AES 的加密（Zkne）和解密（Zknd）指令集，由于 RV32I 和 RV64I 面向的场景不同，指令集在设计上有较大的差别，会有不同的硬件消耗和软件使用方式，这将在之后的章节详细讨论；另一部分是用于计算散列函数 SHA256 和 SHA512 的指令集（Zknh）。

Zks 扩展分为两个部分，一个是分组密码 SM4 的加密解密指令集（Zksed），由于 SM4 算法的特点，加密和解密共用同一个指令；另一部分是用于计算杂凑函数 SM3 指令集（Zksh）。

与标量密码学扩展相对的，RISC-V 向量密码学扩展正在起步阶段，旨在为更高性能的密码学操作提供支持[26]。

2.3 RISC-V 标量密码学涉及的密码学运算

2.3.1 AES

AES[9] 是 NIST 定义的分组密码标准，在现代应用中广为使用。总的来说，AES 算法会对 128 位长的明文或密文用 128、192 或 256 位比特长的密钥进行加密或解密，得到密文或明文，这分别被称为 AES-128, AES-192 和 AES-256，具有不同的安全性和计算需求。

AES 加密由两个部分组成：密钥生成和多轮加密。密钥生成会根据输入的 128、192 或 256 位长密钥生成多轮的密钥，供之后的多轮加密使用；在每轮加密中，算法会对上一轮加密完成的 128 位状态进行逐字节 SBox 替换，行位移和列混合操作，最后再与该轮的密钥异或，得到新的状态。

在上述过程中，RISC-V 标量指令集扩展针对（逐字节 SBox 替换、行位移和列混合）操作定义了相应指令 aes32esmi 与 aes64esm，将三个操作融合在一个指令中执行，对于不需要列混合的加密轮，也定义了相应指令 aes32esi 和 aes64es；对于高性能的 RV64，该指令集扩展还对密钥生成定义了相应指令 aes64ks1i 和 aes64ks2。

AES 解密与 AES 加密互逆，直觉上只需要反向进行 AES 加密中的步骤，然而，直接反向进行会由于不能合并数据通路而带来较高的硬件代价。为了合并数据通路，有以下观察：

1. 一轮加密：（SBox 替换，行位移，列混合，轮密钥异或）的逆过程为一轮解密：（轮密钥异或，反向列混合，反向行位移，反向 SBox 替换）
2. SBox 替换与行位移可以互换顺序，反向 SBox 替换与反向行位移可以互换顺序
3. 由于异或的性质，可以对轮密钥本身做一次反向列混合，记为反向列混合轮密钥，从而（轮密钥异或，反向列混合）等于（反向列混合，与反向列混合轮密钥异或）
4. 两轮解密：[（轮密钥异或，反向列混合，反向行位移，反向 SBox 替换），（轮密钥异或，反向列混合，反向行位移，反向 SBox 替换）] 可以被重组为 [（反向列混合，与反向列混合轮密钥异或，反向 SBox 替换，反向行位移），（反向列混合，与反向列混合轮密钥异或，反向 SBox 替换，反向行位移）]，从而可以被重组为 [反向列混合，与反向列混合轮密钥异或，（反向 SBox 替换，反向行位移，反向列混合，与反向列混合轮密钥异或），反向 SBox 替换，反向行位移]
5. （反向 SBox 替换，反向行位移，反向列混合）可以与（SBox 替换，行位移，列混合）合并数据通路

在这个观察基础上，RISC-V 标量指令集扩展针对（反向 SBox 替换，反向行位移，反向列混合）操作定义了相应指令（aes32dsmi, aes64dsm），同时以上观察也能推广到不需要列混合的解密轮中，从而也定义了相应指令 aes32dsi 和 aes64ds；对于高性能的 RV64，除了已经定义的密钥生成指令 aes64ks1i 和 aes64ks2, 为了快速生成反向列混合轮密钥，还定义了 aes64im 指令；对于 RV32，为了生成反向列混合轮密钥，可以对轮密钥先进行一次 aes32esi，即（SBox 替换，行位移），后进行一次 aes32dsmi，即（反向 SBox 替换，反向行位移，反向列混合），最终结果为仅反向列混合。

2.3.2 SHA256/SHA512

SHA256/SHA512 是 NIST 定义的安全哈希函数 2 家族中的两个成员[27]，被广为使用。总的来说，SHA256/SHA512 的算法基本被基础指令集的运算所覆盖，只有其中的静态移位或静态循环移位并异或的运算（即 Sigma 算子和 Sum 算子）较为特殊，而这可以被硬件较快实现。

RISC-V 标量密码学指令集扩展分别对 Sigma 算子和 Sum 算子定义了相应的指令。

2.3.3 SM4

SM4[10] 是商密定义的分组密码标准。该算法的输入是 128 位比特的明文和 128 位比特的密钥，输出是 128 比特的密文。SM4 算法将 32 位比特称为一个字（word），故而也可以将输入看作四个字的明文和四个字的密钥，将输出看作四个字的密文。

SM4 加密由两个部分组成：多轮加密和密钥生成。多轮加密由轮函数 F 进行。轮函数 F 的定义是 $F(X_0, X_1, X_2, X_3, rk) = X_0 \oplus L(\tau(X_1 \oplus X_2 \oplus X_3 \oplus rk))$，其中 $\oplus$ 是异或，τ 是对输入的一个字中的各个字节进行 SM4 SBox 替换，L 是由循环移位组成的线性变换，X_i 和 rk 均为一个字。

加密过程是根据之前的四个字 $X_i, X_{i+1}, X_{i+2}, X_{i+3}$ 与轮密钥 rk_i 利用轮函数 F 生成新的字 $X_{i+4} = F(X_i, X_{i+1}, X_{i+2}, X_{i+3}, rk_i)$，重复 32 轮，直到生成 X_{35}，之后输出 $(X_{35}, X_{34}, X_{33}, X_{32})$。

密钥生成会根据输入的四个字的密钥生成 32 个字的轮密钥，在生成时，会使用另一个轮函数 F' 根据之前生成的四个字生成新的轮密钥，使用方法与加密中使用轮函数 F 的方法类似。F' 与 F 相比，只有 L' 线性变换与 L 不相同，其余过程均相同。在生成过程中，会用到一些常数 FK 与 CK，作用类似于加密过程中的轮密钥。

由于 SM4 算法的设计，其解密过程与加密过程相同，差别仅在于轮密钥的使用顺序与加密时相反，即第一轮解密使用的轮密钥为最后一轮加密时使用的轮密钥。

在 RISC-V 标量密码学指令集扩展中，SM4 加密解密指令是同一个指令，记为 sm4ed；该扩展也定义了密钥生成指令 sm4ks。

2.3.4 SM3

SM3[28] 是商密定义的杂凑函数。总的来说，SM3 的算法基本被基础指令集的运算所覆盖，只有其中的静态循环移位并异或操作（P0 变换和 P1 变换）较为特殊，而这可以被硬件较快实现。

RISC-V 标量密码学指令集扩展分别对 P0 变换和 P1 变换定义了相应的指令 sm3p0 和 sm3p1。

2.4 Chisel 硬件描述语言

Chisel[33] 是基于 Scala 的硬件描述语言。该语言由加利福尼亚大学伯克利分校开发，是使用 Scala 实现的软件库。与 Chisel 相伴随的还有电路灵活中间表达（Flexible Intermediate Representation for RTL, FIRRTL）用于表达电路，以及一系列综合工具链。

Chisel 的一个重要特点是能生成可参数化的电路。这是基于 Scala 中的类可以接受不同的初始化参数的特性实现的。在这种情况下，一份代码可以按照不同的场景（32 位处理器或 64 位处理器，是否启用某特性等）生成不同的电路，较为灵活。

Chisel 的另一个特点是可以使用 Scala 带来的高阶函数，例如 map, fold, reduce 等逻辑，这为表达电路带来了便利。

第 3 章　相关工作

3.1　riscv-crypto

riscv-crypto 是 RISC-V 标量密码学指令集扩展的官方仓库，在其 rtl 目录[29]下包含一份使用 Verilog 编写的 Zkn 与 Zks 的参考实现；Zbk 部分在 rocketchip PR#2906① 中。

该 Verilog 实现中重复代码较多，例如实例化了多个单元，在代码上仅有下标的差别，而在 Chisel 中这可以使用高阶函数便利表达；一些功能的高效实现也可以被集成成为语言的一部分，例如循环移位，而这是 Chisel 已经做到的；同时，在该实现中，RV32 和 RV64 的相同功能的模块代码并未合并成同一个模块，而是拆分成两个模块，而这是 Chisel 可以表达的。这些因素共同使得 Verilog 实现的维护难度较大。

该参考实现也尝试过向 Rocket 提交合并请求，即上文所述 rocketchip PR#2906，但是因为 Rocket 的开发语言是 Chisel 而没有合并。

在该实现中 SBox 是通过实时计算完成的。本文使用了该实现中的优化 SBox。

该仓库的 benchmarks 目录[30]下中有使用 intrinsic 和汇编实现的优化软件实现。由于其使用的接口是单独定义的接口，普通软件如果要使用该实现需要进行单独的适配，较为不便；该实现也仅仅适用于单个函数的性能评估，对于整体性能，例如 AES 各个操作模式的性能评估，还需要与 OpenSSL 等密码学库集成，而这是本文提供的软件实现能做到的。

3.2　香山

香山[25]是中国科学院计算技术研究所和鹏城实验室共同研发的开源高性能 RISC-V 64 位处理器，其中有使用 Chisel 编写的 RV64 标量密码学指令集扩展的电路实现。

相对于本文致力于减小面积而合并数据通路，香山在实现时更注重频率、在面积上考虑较少，故而在合并数据通路上并没有如同本文一样激进。例如本文中将 aes64dsm 和 aes64im 合并，共用同一个 MixColumn64，而香山则为两个指令单独实例化了 MixInv 模块；相对于本文中加密 SBox 与解密 SBox 共用一个 SBoxMid，

① https://github.com/chipsalliance/rocket-chip/pull/2906

香山在实现中分别实例化了完整的加密 SBox 和解密 SBox。

相对于本文中的单周期实现，香山在实现中插入了一些寄存器，将一个指令拆成多个周期执行。

相对于本文，香山的实现中没有 RV32 标量密码学扩展的实现。

3.3 riscv-isa-sim

riscv-isa-sim[34] 是 RISC-V 的官方指令集模拟器。该模拟器在软件上实现了标量密码学扩展的功能，是指令行为的黄金标准。

由于该模拟器的接口与电路实现仿真程序的接口相同，该模拟器可以用作对比和模糊测试。在本文的电路实现提交上游后，评审者使用了 riscv-isa-sim 与本文的电路实现进行模糊测试以验证电路实现的行为正确性，帮助本文的电路实现符合指令集标准。

3.4 riscv-arch-test

RISC-V 架构测试[35] 是 RISC-V 的官方测试工具。这里存放了对各个指令的单元测试，可以快速验证电路实现的基本功能。

本文的电路实现通过了架构测试中的 Zk 与 Zb 部分①。

① Zb 部分尚未合并，参考 https://github.com/riscv-non-isa/riscv-arch-test/pull/228

第 4 章　电路实现

本文在 Rocket 芯片生成器[24]的基础上实现 RISC-V 指令集标量密码学扩展。Rocket 是一个开源的基于 Chisel 的单片系统（System on Chip, SoC）生成器。作为模板，它可以根据不同的需求生成相应的单片系统，具备较大的灵活性。举个例子，它的同一套代码可以同时被用来生成 RV32 和 RV64 的核心；另一个例子是，它包含多达二十个布尔参数，可以选择是否开启某一芯片特性，例如是否开启虚拟化支持、是否开启原子指令支持、是否添加浮点运算单元等。本文遵循 Rocket 的思路，在实现标量密码学加速单元的同时，也会增加相应的布尔参数，将其变成用户可配置的功能。

本文的电路实现已经提交上游，在 rocket-chip PR#2950① 中，获得了较多的积极评审意见，同时本文作者也参与 Rocket Chip 工作组，为该实现最终合并进入上游听取上游的意见和建议。本文认为，仅仅开源密码学电路单元并不能让使用者利用该工作，从而沦为“看源”项目，与密码学电路单元配套的基础设施乃至工具链、工作流都需要开源与清晰的文档；与此同时，维护一个上游的分支并在分支中添加相应的单元不具备可维护性和可持续性，不仅维护成本高，使用者也需要在上游和分支实现中作出选择，出现问题时难以调试、苦不堪言；而将上游的分支称为自己的独创更不可取：上游提供的一系列自由开源软件，它们的名字里面虽然有 Free 和不作出保证（No Warranty），但这并不意味着免费和理所应当（taken for granted），上游也是人的劳动，并不是凭空而来，在上游缺乏功能或出现问题时，下游需要参与上游，而不是在摘取上游成果后不闻不顾；只有将自己的实现提交给上游，并参与上游的维护乃至成为相应的维护者，才能将一份工作的贡献最大化，真正能为生态添砖加瓦。

由于标量密码学扩展中的功能较多，本文在电路实现上分成了三个单元：NIST 密码单元，商密密码单元，算术位运算逻辑单元。NIST 密码单元实现了 NIST 定义的高级加密标准（AES）和安全哈希函数（SHA256 和 SHA512）；商密密码单元实现了商密定义的分组密码（SM4）和杂凑函数（SM3）；算术位运算逻辑单元（ABLU）基于 Rocket 中原有的算术逻辑单元（ALU），在实现原有基础指令集的基础上增加了位运算的支持，可以直接替换原有的算术逻辑单元。

① https://github.com/chipsalliance/rocket-chip/pull/2950

4.1 NIST 密码单元

4.1.1 AES 实现

在 AES 扩展指令设计时，对于 RV32 来说，其首要考虑的是面积和功耗，而对 RV64 来说，其首要考虑的是性能[19]。于是，在 RV32 中 AES 加密解密指令只会操作一个字节，只需要一个加密 SBox 和一个解密 SBox；而在 RV64 中 AES 加密解密指令可以同时操作八个字节，需要八个加密 SBox 和八个解密 SBox；RV64 还为密钥生成提供了专有指令，能在复用已有数据通路的情况下提供密钥生成的加速。

如背景介绍中所述，RISC-V 标量密码学扩展指令在设计时即考虑到硬件设计上的如何合并数据通路。相对于时延，本文在实现电路时更为关注于减小密码学单元所占用的面积，从而致力于合并多个指令的数据通路。如图 4.1 所示，RV32 的所有 AES 加速指令均可合并到同一条数据通路中：对于需要列混合的指令 aes32esmi 和 aes32dsmi，该数据通路在 SBox 和 MixColumn8 处可以根据控制信号进行正向或反向的运算，从而合并两者；对于不需要列混合的指令 aes32esi 和 aes32dsi，如图 4.1(c) 所示，它们只需要根据控制信号直接选择来自 SBox 的输出，即可复用已有的数据通路。

在 RV64 中，除了较为独立的仅对输入异或的密钥生成指令 aes64ks2，其余的所有指令均可合并在同一条数据通路中。如图 4.2 所示，对于加密与解密指令 aes64esm, aes64dsm, aes64es, aes64ds，其合并逻辑与 RV32 类似，在此不再赘述；对于密钥生成指令 aes64ks1i，该指令需要对输入中的四个字节进行 SBox 替换，相对于新增四个加密 SBox, 本文在实现时重复利用已有的八个加密 SBox，由表 4.1 可知单个 SBox 占用的面积较多，所以复用已有的 SBox 能带来可观的面积减小；对于反向列混合轮密钥生成指令 aes64im，该指令可以重复利用已有的 MixColumn64 模块中的反向部分。

由于 RV32 与 RV64 的差异，我们将一个指令的多个操作拆分为多个小模块，利用 Chisel 的灵活性快速构建数据通路。RV32 和 RV64 均会对输入数据进行逐字节 SBox 替换、行位移和列混合等操作。对于逐字节 SBox 替换，我们将加密与解密的 SBox 合并，参考图 4.3(a) 可知，加密与解密 SBox 可以共享一个 SBoxMid 模块[36-37]，相较与单独实例化加密 SBox 和解密 SBox，这种合并能节省面积；对于行位移，由于 RV32 中只操作一个字节，所以需要动态循环位移实现相应效果，即图 4.1 中由 bs 信号控制的 Byte Select 和 RotateRight，而在 RV64 中，由于精妙的指令设计，RV64 只需要从将输入的 rs1 与 rs2 两个寄存器中各 8 字节数据拼接成

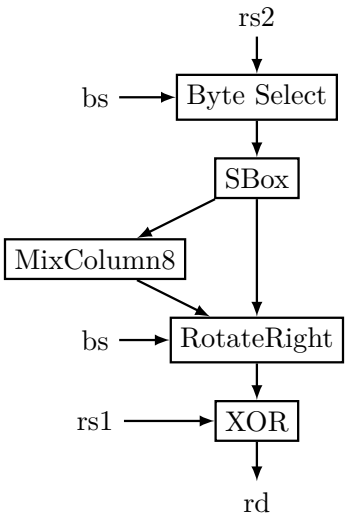
代码 4.1　MixColumn32 的 map 与 reduce

```
// 将 32 位输入拆分成四个字节，各个字节带上索引
io.out := asBytes(io.in).zipWithIndex.map({
  // 字节，索引
  case (b, i) => {
    val m = Module(new MixColumn8(enc))
    // 将字节送入 MixColumn8 中
    m.io.in := b
    // 将 MixColumn8 的输出根据索引静态循环移位，成为返回值
    m.io.out.rotateLeft(i * 8)
  }
}).reduce(_ ^ _)
// 将所有返回值异或
```

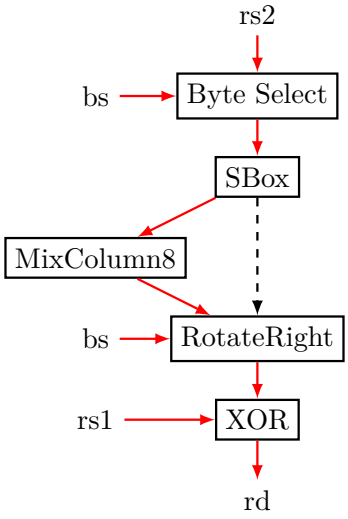
16 字节数据（即一个完整的 AES 状态），并从中静态选取八个字节，于是我们实现了名为 ShiftRows 的模块来进行相应操作，图 4.2(b) 和图 4.2(c) 中均用到了该模块；对于列混合，我们实现了 MixColumn8，MixColumn32 和 MixColumn64 三个模块，后者由前者组合而成。如图 4.3(b) 所示，本文在 MixColumn8 中同时处理正向和反向列混合，图中的 $\otimes$ 是 GFMul，在下一段中详细介绍；图 4.3(c) 中的 MixColumn32 会将输入的 4 字节数据分发给四个 MixColumn8 实例，并将各实例输出的 4 字节数据异或在一起，参考代码 4.1，该实现用到了 Scala 提供的 map 和 reduce 高阶函数，相较于 Verilog，这种写法更为简洁明了；图 4.3(d) 中的 MixColumn64 则是将输入的 8 字节数据分发给两个 MixColumn32 实例，并将结果拼接在一起。

对于 GFMul，本文利用了 Chisel 的参数化特性快捷生成高效电路。GFMul 是定义在伽罗瓦域上的乘法，在八位数据上进行运算。本文注意到 MixColumn8 模块中均为将输入乘以常数，这可以在电路上优化从而得到高效电路。参考图 4.3(b) 可知，共有 6 个常数，相对于针对各个常数编写相应模块，使用 Chisel 可以直接得到参数化模板，在调用时再生成相应高效电路。如代码 4.2 所示，GFMul 模块接受一个 Int 类型的 y 参数，相对于用于描述电路的 UInt 类型，y 参数需要在编译时确定，故而在模块中的 VecInit 时，可以在编译时确定是否包含某一电路，从而生成高效电路。这来自一个观察，即该伽罗瓦域上的加法等价于异或，则 $x \otimes y = (x \otimes (8 * y_3))\hat{}(x \otimes (4 * y_2))\hat{}(x \otimes (2 * y_1))\hat{}(x \otimes (1 * y_0))$，其中 y_i 是 y 参数的各个比特位。这里也用到了 Scala 提供的 reduce 高阶函数，简洁明了。

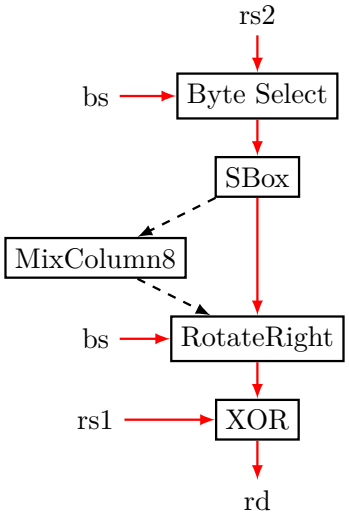
在 SBox 的实现上，本文尝试了两种方法，一种是使用 ROM，另一种是实时计算[36-37]，结果发现后者能在大幅减小面积的同时不造成延时上的问题。如表 4.1 所示，对 RV64 综合后可知，其中 NIST 密码单元中配置了 8 个加密解密 SBox，相



(a) 完整数据通路

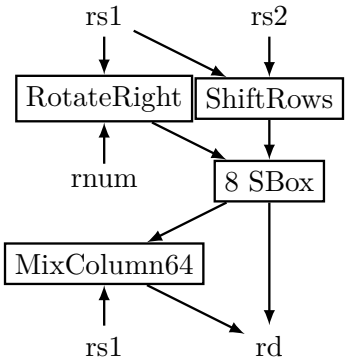


(b) aes32esmi, aes32dsmi

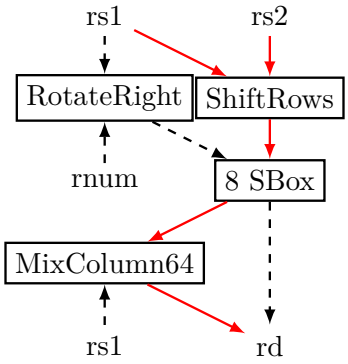


(c) aes32esi, aes32dsi

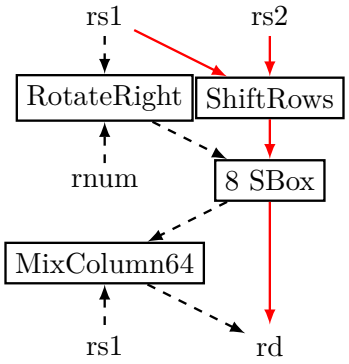
图 4.1　RISC-V 32 的 AES 数据通路



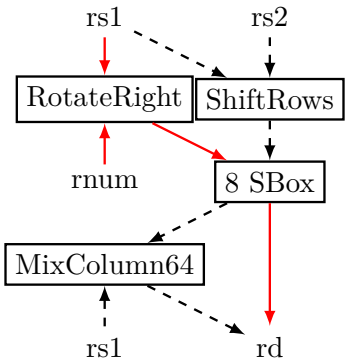
(a) 完整数据通路



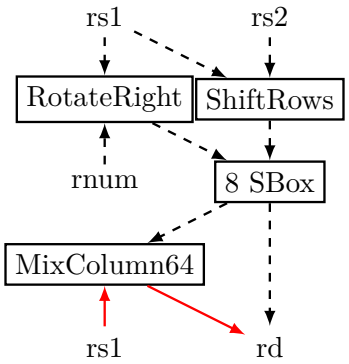
(b) aes64esm, aes64dsm



(c) aes64es, aes64ds

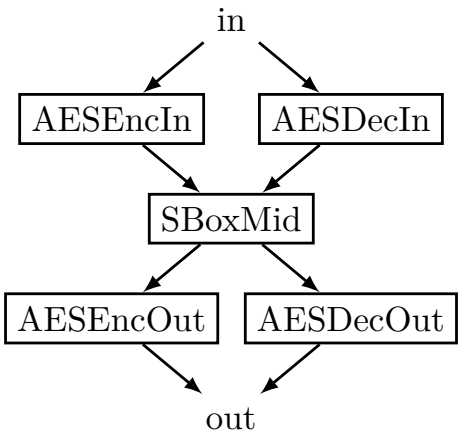


(d) aes64ks1i

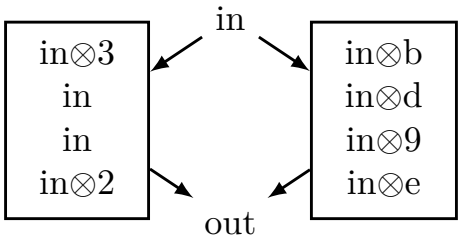


(e) aes64im

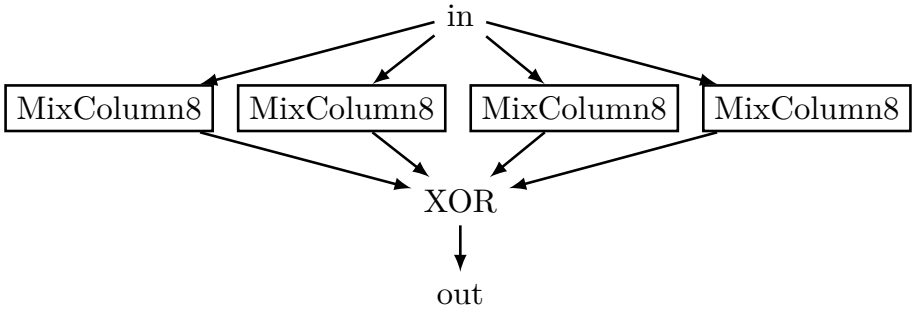
图 4.2　RISC-V 64 的 AES 数据通路



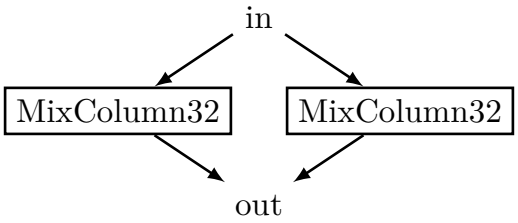
(a) AES 加密与解密 SBox 可以合并



(b) 正向与反向 MixColumn8 可以合并



(c) 由 MixColumn8 搭建的 MixColumn32



(d) 由 MixColumn32 搭建的 MixColumn64

图 4.3　RV32 与 RV64 共有的 AES 单元

代码 4.2　GFMul 的 Chisel 参数化实现

```
class GFMul(y: Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })

  // x*f(x) = 2*in in GF
  def xt(in: UInt): UInt = (in << 1)(7,0) ^
    Mux(in(7), 0x1b.U(8.W), 0x00.U(8.W))
  // 4*in in GF
  def xt2(in: UInt): UInt = xt(xt(in))
  // 8*in in GF
  def xt3(in: UInt): UInt = xt(xt2(in))

  require(y != 0)
  io.out := VecInit(
    (if ((y & 0x1) != 0) Seq(   (io.in)) else Nil) ++
    (if ((y & 0x2) != 0) Seq( xt(io.in)) else Nil) ++
    (if ((y & 0x4) != 0) Seq(xt2(io.in)) else Nil) ++
    (if ((y & 0x8) != 0) Seq(xt3(io.in)) else Nil)
  ).reduce(_ ^ _)
}
```

对于整个 Rocket 核的大小，如果在 ROM 实现下，NIST 密码单元能占据 25% 的面积，而在实时计算的实现中，NIST 密码单元能减少到与乘法除法器相类似的面积；对 RV32 综合后可知，由于 NIST 密码单元只配置了 1 个加密解密 SBox，面积也能大幅减小。

表 4.1　不同 SBox 实现的面积对比

模块名	面积（ROM）	面积（实时运算）	面积（实时运算，RV32）
Rocket 核	80570	67377	36346
乘法除法器	8015	8015	3107
NIST 密码单元	19378	6804	1829
单个 AES 加密解密 SBox	1899	323	335
商密密码单元	1471	709	707
单个 SM4 加密解密 SBox	934	172	172

代码 4.3　SHA256 与 SHA512 加速指令的 Chisel 实现

```
val inb = io.rs1(31,0)
val sha256sig0 = sext(inb.rotateRight(7)
  ^ inb.rotateRight(18) ^ (inb >>  3))
// sha256sig1, sha256sum0, sha256sum1 等与上述实现类似

val sha512sig0 = if (xLen == 32) {
  val sha512sig0_rs1  = (io.rs1 >> 1 )
    ^ (io.rs1 >> 7) ^ (io.rs1 >>  8)
  val sha512sig0_rs2h = (io.rs2 << 31)
    ^ (io.rs2 << 24)
  val sha512sig0_rs2l = sha512sig0_rs2h ^ (io.rs2 << 25)
  sha512sig0_rs1 ^
    Mux(io.hl, sha512sig0_rs2h, sha512sig0_rs2l)
} else {
  require(xLen == 64)
  io.rs1.rotateRight(1)
    ^ io.rs1.rotateRight(8) ^ io.rs1 >> 7
}
// sha512sig1, sha512sum0, sha512sum1 等与上述实现类似
```

4.1.2　SHA256/SHA512 实现

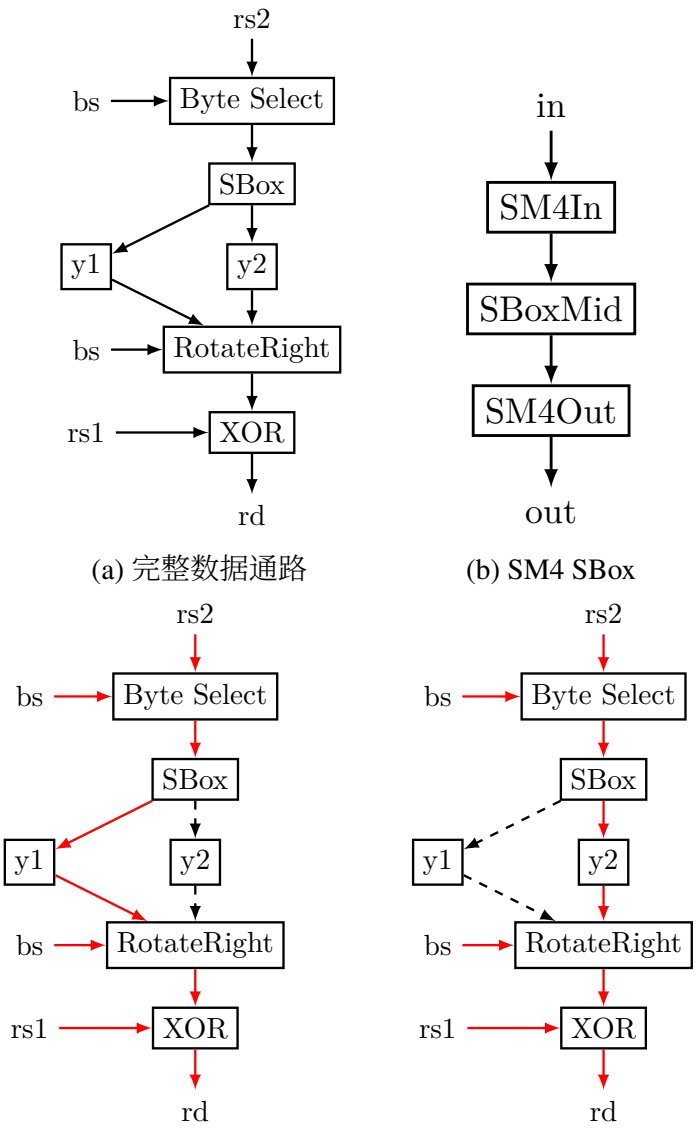
RISC-V 标量密码学指令集扩展对 SHA256/SHA512 的 Sigma 算子和 Sum 算子定义了相应的指令。Sigma 算子和 Sum 算子均为对输入进行静态位移或静态循环循环位移并异或的算子。如代码 4.3 所示，所有算子均可使用 Chisel 提供的静态循环移位、静态移位和异或等功能实现该电路；可以注意到，对于 SHA256，两个架构可以共享 SHA256 的代码，SHA512 则需要根据不同的架构生成不同的电路。

4.2　商密密码单元

4.2.1　SM4 实现

对于 SM4，RISC-V 标量密码学指令集扩展为 RV32 和 RV64 两个架构定义的指令均相同, 为加密解密指令 sm4ed 与密钥生成指令 sm4ks。由于算法的特性, SM4 的加密与解密使用同一个 SBox，所以 sm4ed 指令可以同时用于加密与解密。SM4 的实现与 RV32 中的 AES 指令集较为类似，单个指令只操作一个字节；即使对于 RV64，SM4 指令也只操作一个字节，这与 RV64 中 AES 指令可以同时操作八个字节不同。

如图 4.4(a) 所示，与实现 AES 的思路类似，sm4ed 与 sm4ks 也可以合并数据通路。图中的 y1 和 y2 是因为加密解密和密钥生成所使用的线性变换 L 与 L' 不同



(a) 完整数据通路　(b) SM4 SBox

(c) sm4ed　(d) sm4ks

图 4.4　SM4 数据通路与 SM4 SBox

代码 4.4　SM3 加速指令的 Chisel 实现

```
val r1 = io.rs1(31,0)
val sm3 = sext(Mux(io.zks_fn === ZKS.FN_SM3P0,
  r1 ^ r1.rotateLeft(9) ^ r1.rotateLeft(17),
  r1 ^ r1.rotateLeft(15) ^ r1.rotateLeft(23)))
```

(参考背景介绍)，其余数据通路均可合并。

由于 SM4 的加密与解密使用同一个 SBox，于是在电路实现中 SBox 占用的面积较小。如表 4.1 所示，同时用于加密与解密的 SM4 SBox 的面积是同时用于加密与解密的 AES SBox 的一半。

可以注意到，SM4 的数据通路（图 4.4(a)）与 RV32 中的 AES 32 的数据通路（图 4.1(a)）由于共享 Byte Select，RotateRight 和 XOR 等单元，可以合并成一条数据通路，SM4 的 SBox（图 4.4(b)）与 AES SBox（图 4.3(a)）由于共享 SBoxMid 也可以合并成一个 SBox。故而在实现 RV32 核心时，可以将其进一步合并，达到 RV32 上占用面积最小的目标。在本文中，由于 NIST 密码单元与商密单元是单独的两个单元，且在 Rocket 配置中可以只开启其中一个单元，在写法上较为不便，本文并没有合并；如果要实现同时开启两个单元并合并数据通路的 RV32 电路并追求面积最小，只需要将 SM4 的代码填入 RV32 的 AES 代码中并增加一层 Mux，较为简便。

4.2.2　SM3 实现

RISC-V 标量密码学指令集扩展对 SM3 的 P0 变换和 P1 变换定义了相应的指令。P0 变换和 P1 变换均为对输入进行静态位移或静态循环循环位移并异或的算子。如代码 4.4 所示，所有算子均可使用 Chisel 提供的静态循环移位、静态移位和异或等功能实现该电路。

4.3　算术位运算逻辑单元

RISC-V 标量密码学指令集扩展对常见的位运算也有支持，而这些可以被合并到现有的数据通路中。现有的 ALU 为了最小化面积，将加法和减法通过同一个加法器实现，这利用了 $a-b=a+(\sim b)+1$ 这一性质，从而在这里引入了一个对 b 的取反运算；与此同时，基础指令集中也定义了逻辑运算 AND/OR/XOR，即 $a\&b$, $a|b$ 和 $a\hat{}b$。RISC-V 标量密码学指令集扩展在此基础上定义了 ANDN/ORN/XNOR，即 $a\&(\sim b)$, $a|(\sim b)$, $a\hat{}(\sim b)$。本文复用已有的减法中的对 b 取反（图 4.6 中的 in2_inv）

和逻辑运算的逻辑门实现了这三个指令，而这只需要付出增加少量面积和时延的代价。

经过一定的权衡，本文将一些常用的位运算操作合并进入 ALU，并将新的单元称为 ABLU。参考表 4.2 与表 4.3，在一定的面积代价下，在原有 ALU 支持的 14 条指令的基础上，ABLU 能支持多达 48 条指令。如图 4.5 与图 4.6 所示，ALU 与 ABLU 中均有加法器、位移器、逻辑单元和比较单元；为了支持更多的操作，本文提取了 48 条指令的数据路径，将其合并进入已有的数据路径中，最终效果如图 4.6 所示。

表 4.2　ALU 与 ABLU 的面积对比

模块名	面积	面积（RV32）
ALU	1721	791
ABLU	4309	1953
xperm/clmul	7612	2008

表 4.3　ALU 与 ABLU 的功能对比

模块名	支持指令数量
ALU	14
ABLU	48

由于 ABLU 中的指令较多，合并数据路径的方法并不相同，本文在此只简单介绍循环位移的合并方法。在 ALU 中，由于动态位移电路占用面积较大，对于左移和右移单独实现数据通路并不经济，观察到只需要将输入调转比特顺序（reverse），左移即可用右移实现，只需要在输出时再次调转比特顺序即可获得正确的输出，于是如图 4.5 中的 shin_r 到 shin 与 shout_r 到 shout 所示，这一条数据通路可以同时实现右移和左移；这里的一个细节是用右移实现左移而不是用左移实现右移，因为右移有算术右移和逻辑右移两种，而左移只有逻辑左移一种，较容易用逻辑右移实现。ABLU 需要支持循环左移和循环右移，于是遵照 ALU 中的思路，将循环左移转变为循环右移，从而复用已有的调转比特顺序的电路，这可以参考图 4.6 中的 rotate_r 和 shift_r。事实上，在桶式移位器（Barrel Shifter）中，循环右移和右移也能合并数据路径，而 Chisel 暂未支持该功能。另一个可以合并的通路数据是在 RV64 中循环右移低 32 位数据（RORW），直觉上这不能使用 64 位循环右移器实现，

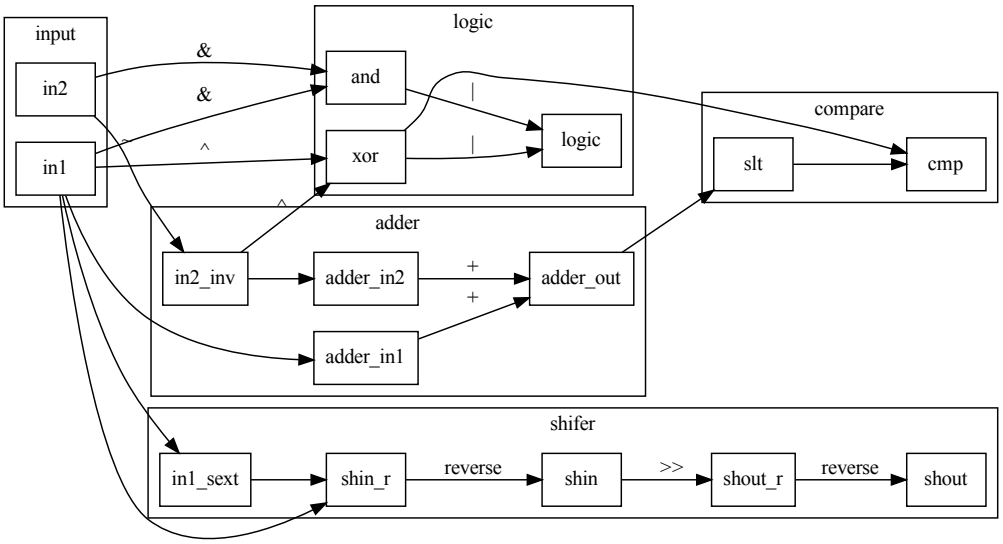


图 4.5　ALU 数据通路

但如果将输入的低 32 位数据（31,0）复制到高位（63,32），即图 4.6 中的 in1_rotate，则可以利用已有的 64 位循环右移器，而不需要新增一个 32 位循环右移器。

对于特殊的运算 xperm （逐字节/逐半字节排列）和 clmul（无进位乘法），本文将其单独拆分成 BitmanipCrypto 模块，从表 4.2 中可知其占用面积较高，这是预期的现象[26,38]。xperm 需要对于输入的每个字节或每半个字节进行操作，clmul 需要做对每一位控制乘法。如代码 4.5 所示，相对于 Verilog 写法中需要多份重复代码来实现该功能[29]，Chisel 实现中 xperm 和 clmul 可以利用 map 和 reduce 等高阶函数，简介明了。

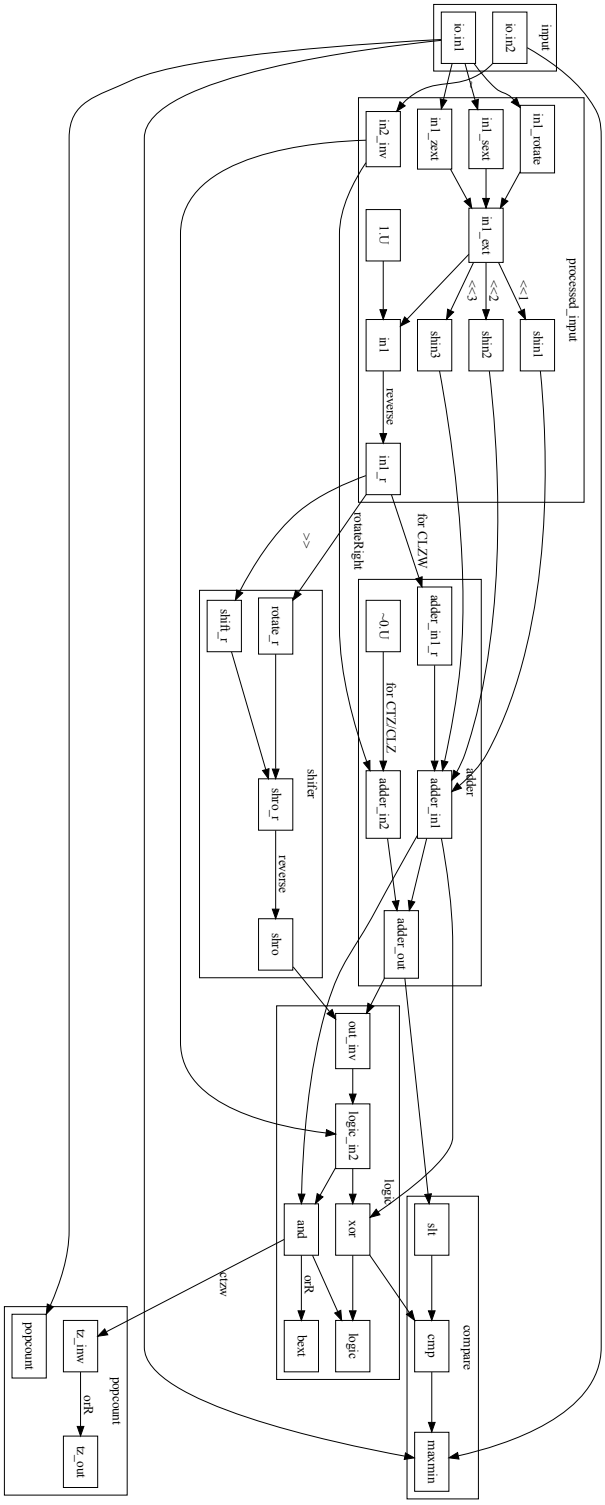


图 4.6　ABLU 数据通路

代码 4.5　xperm 和 clmul 的 Chisel 实现

```
val xperm8 = VecInit(rs2_bytes.map(
  // x 是 rs2 的一个字节
  // 当 x 大于 xLen/8 时，索引溢出，返回 0
  // 否则在 rs1 的字节们中查询
  x => Mux(x(7,log2Ceil(xLen/8)).orR, 0.U(8.W), rs1_bytes(x))
// 拼接所有查询结果
).toSeq).asUInt

val xperm4 = VecInit(rs2_nibbles.map(
  // x 是 rs2 的一个半字节
  // 当 x 大于 xLen/4 时，索引溢出，返回 0
  // 否则在 rs1 的半字节们中查询
  x => if (xLen == 32)
    Mux(x(3,log2Ceil(xLen/4)).orR, 0.U(4.W), rs1_nibbles(x))
  else {
    require(xLen == 64)
    // 由于 x 不会大于 16，故不会溢出
    // 直接在 rs1 的半字节们中查询
    rs1_nibbles(x)
  }
).toSeq).asUInt

// rs1 乘以 rs2，按照课本上的经典乘法
// 即为 rs2 的每个比特控制静态位移后的 rs1
// 之后用异或累加，异或为伽罗瓦域中的加法
val clmul = clmul_rs2.asBools.zipWithIndex.map({
  // rs2 的一个比特，该比特的位数
  // 左移 i 位，如果比特 b 是 1 则采用，否则为 0
  case (b, i) => Mux(b, clmul_rs1 << i, 0.U)
  // 使用高阶函数 reduce 累加
}).reduce(_ ^ _)(xLen-1,0)
```

第 5 章　软件实现

指令集设计的精妙之处往往需要从硬件和软件两个角度展示。在上一章节中，本文展示了不同指令对数据通路的复用（其中以 AES 的解密为典型例子），也展示了对 Chisel 模块的复用；在这一章节中，本文将说明多个指令如何联合起来达到不同的作用，这里既有一个指令的多个用途，也有加速指令与普通指令之间的联合作用。这即是上一章节的延伸，也体现了 RISC 的设计精神。

本章节详细讨论了为 OpenSSL 编写的优化软件实现[①②③④]的设计细节，从软件使用方式上展示了该扩展指令集为了硬件友好而带来的软件上的影响。

5.1　AES

5.1.1　单轮加密与解密

在 AES 的多轮加密中，每一轮的输入和输出均为 16 字节，本文将这 16 字节称为 AES 状态。在每一轮中，需要对 AES 状态进行逐字节 SBox 替换，行位移，列混合和轮密钥异或，从而生成新的状态。由于输入字节较多，该指令集在设计时并没有考虑将一轮加密通过一个指令完成，也没有将各个步骤拆分成单独的指令，而是将多个步骤融合在一个指令中，单个指令只处理一个状态的一部分。

在 RISC-V 64 位架构中，两个寄存器即可储存完整的 AES 状态，按照 RISC 指令经典的 “两个源地址、一个目标地址” 的结构，加速指令在完整获取一个 AES 状态后，从中选出状态的一半（这种选取即为行位移），对半个状态做逐字节 SBox 替换和列混合，生成异或之前的新 AES 状态的一半；由于行位移的特性，仅仅调换两个源寄存器的顺序即可生成完整的异或之前的新 AES 状态；轮密钥异或则可以使用常规的异或指令，这可以体现加速指令与普通指令的联合作用。该过程如代码 5.1 所示。

RISC-V 32 位架构则需要四个寄存器才能储存一个完整状态。与 RISC-V 64 位架构不同，由于列混合的类似于矩阵乘法的特性，相对于 64 位架构中的内积方式，32 位架构采用了外积的方式，即在轮密钥的基础上，通过逐次累加（即异或）旧

① RISC-V 64 AES 实现：https://github.com/openssl/openssl/pull/18197
② RISC-V 32 AES 实现：https://github.com/openssl/openssl/pull/18308
③ 同时适用于 RISC-V 64 和 RISC-V 32 的 SM4 实现：https://github.com/openssl/openssl/pull/18285
④ SM3 优化：https://github.com/openssl/openssl/pull/18287

代码 5.1　RISC-V 64 的一轮 AES 加密

```
# $Q0 与 $Q1 中储存一个 AES 状态，生成异或前的新 AES 状态的一半
aes64esm $Q2,$Q0,$Q1
# 调换源寄存器顺序，生成异或前的新 AES 状态的另一半
aes64esm $Q3,$Q1,$Q0
# 使用常规指令访存和异或
ld       $T0,0($KEYP)
ld       $T1,8($KEYP)
xor      $Q0,$Q2,$T0
xor      $Q1,$Q3,$T1
# $Q0 与 $Q1 中储存一个新的 AES 状态
```

代码 5.2　RISC-V 32 的一轮 AES 加密

```
# aes32esmi4 $key,$s0,$s1,$s2,$s3 是以下过程的缩写
# 该过程将 AES 状态的 4 个字节外积到 $key 上
aes32esmi $key,$key,$s0,0
aes32esmi $key,$key,$s1,1
aes32esmi $key,$key,$s2,2
aes32esmi $key,$key,$s3,3

# T0~T4 读取轮密钥
lw      $T0,0($KEYP)
lw      $T1,4($KEYP)
lw      $T2,8($KEYP)
lw      $T3,12($KEYP)
# Q0~Q3 中储存一个 AES 状态
aes32esmi4 $T0,$Q0,$Q1,$Q2,$Q3
aes32esmi4 $T1,$Q1,$Q2,$Q3,$Q0
aes32esmi4 $T2,$Q2,$Q3,$Q0,$Q1
aes32esmi4 $T3,$Q3,$Q0,$Q1,$Q2
# T0~T3 中储存一个新的 AES 状态
```

AES 状态与常数的乘积（即 GFMul 和 MixColumn8），逐步更新轮密钥以得到新 AES 状态。单个 aes32esmi 指令只对选取旧 AES 状态中的一个字节，经过 SBox 替换和列混合后得到外积，之后通过异或更新轮密钥，逐步累加成新状态。行位移则体现在多个指令之间：aes32esmi4 中只选取完整状态 s0, s1, s2, s3 中的四个字节以更新同一个轮密钥。该过程如代码 5.2 所示。

对于不需要列混合的最后一轮加密，只需要将 aes64esm 替换为 aes64es、将 aes32esmi 替换为 aes32esi 。这种巧合在指令设计时即被考虑到：对于 64 位架构，行位移所选择的旧状态的一半恰好是新状态的一半；对于 32 位架构，在不需要外积时，只需要更新轮密钥的一个字节。对于仅仅进行轮密钥异或的首轮加密，两种

代码 5.3　RISC-V 64 的 AES-128 轮密钥生成

```
# T0 与 T1 中储存旧轮密钥
# ks1i 负责进行 SBox 替换与轮常数异或
aes64ks1i    $T2,$T1,$rnum
# ks2 负责更新轮密钥
aes64ks2     $T0,$T2,$T0
aes64ks2     $T1,$T0,$T1
# T0 与 T1 中储存新轮密钥

# 如果需要反向列混合轮密钥，则需要添加如下语句
# 将 T0 与 T1 反向列混合至 T2，储存 T2
# T0 与 T1 将继续用于轮密钥生成
aes64im      $T2,$T0
sd           $T2,0($KEYP)
aes64im      $T2,$T1
sd           $T2,8($KEYP)
```

架构均采用常规的异或指令。

参考背景介绍可知，在合并数据路径的同时，软件解密方式与加密方式也较为相似，可以以此类推。

5.1.2　密钥生成

密钥生成意在将通过输入的 16 字节（AES-128）或 24 字节（AES-192）或 32 字节（AES-256）密钥扩展为多轮密钥，供每轮加密、解密使用。对于 RISC-V 64 位架构的加密轮密钥生成，除了上一章节提到的合并数据路径外，aes64ks1i 和 aes64ks2 的设计较为直观，在此不做详细展开；对于解密轮密钥生成，尽管对于一般的实现而言，加密轮密钥和解密轮密钥是相等的，但参考背景介绍可知，RISC-V 标量密码学指令集扩展中解密算法使用的是反向列混合轮密钥而不是轮密钥，故而需要在生成加密轮密钥后单独对轮密钥进行 aes64im。该过程如代码 5.3 所示。

RISC-V 32 位架构中并没有单独的密钥生成加速指令，然而由于 32 位架构中的寄存器位宽为 32 位，恰好能匹配上密钥生成操作中使用的位宽，故而可以复用异或和循环移位等指令；对于密钥生成中所需要的 SBox 替换，则可以直接使用没有列混合的 aes32esi，复用其中 SBox 的部分；为了生成反向列混合轮密钥，在生成加密轮密钥后，进行一次 aes32esi，即 SBox 替换和行位移，之后进行一次 aes32dsmi，即反向 SBox 替换，反向行位移和反向列混合，最终结果为仅反向列混合，故而能得到反向列混合轮密钥。该过程如代码 5.4 所示。

代码 5.4　RISC-V 32 的 AES-128 轮密钥生成

```
# fwdsbox4 $rd,$rs 是以下过程的缩写
# 该过程讲 $rs 各个字节进行 SBox 替换后储存到 $rd 上
aes32esi $rd,$rd,$rs,0
aes32esi $rd,$rd,$rs,1
aes32esi $rd,$rd,$rs,2
aes32esi $rd,$rd,$rs,3

# invm4 $rd,$tmp,$rs 是以下过程的缩写
# 该过程进行 fwdsbox4 后再进行反向 fwdsbox4 并进行反向列混合
# 结果为仅仅反向列混合
li         $tmp,0
li         $rd,0
fwdsbox4   $tmp,$rs
aes32dsmi  $rd,$rd,$tmp,0
aes32dsmi  $rd,$rd,$tmp,1
aes32dsmi  $rd,$rd,$tmp,2
aes32dsmi  $rd,$rd,$tmp,3

# T0~T3 储存旧轮密钥
# 使用 T4 储存轮常数
li         $T4,$rcon[$rnum]
# 因为异或满足结合律和交换律，可以先将
# T0 与轮常数异或，之后再与 SBox 替换结果异或
xor        $T0,$T0,$T4
# 使用 T4 储存循环位移后的 T3
rori       $T4,$T3,8
# T0 与 SBox 替换后的 T4 异或
fwdsbox4 $T0,$T4
# 更新 T1~T3
xor        $T1,$T1,$T0
xor        $T2,$T2,$T1
xor        $T3,$T3,$T2
# T0~T3 储存新轮密钥

# 如果需要反向列混合轮密钥，则需要添加如下语句
# T0~T3 将继续用于轮密钥生成
invm4      $T5,$T4,$T0
sw         $T5,0($KEYP)
invm4      $T5,$T4,$T1
sw         $T5,4($KEYP)
invm4      $T5,$T4,$T2
sw         $T5,8($KEYP)
invm4      $T5,$T4,$T3
sw         $T5,12($KEYP)
```

5.2 SM4

RISC-V 标量密码学指令集扩展对 RV32 与 RV64 定义的 SM4 加速指令相同，与此同时 SM4 算法是自然定义在 32 位数据上运算的。在这种条件下，可以预期 RV64 的实现与 RV32 实现基本相同。为此，相对于为 RV32 和 RV64 分别提供汇编实现，为了提高可维护性，本文为 OpenSSL 编写的 SM4 优化汇编实现可以同时被 RV32 和 RV64 使用。

为了实现这种复用，本文使用了以下两个技巧：一、在 RV64 中，寄存器储存的有效数据只有 32 位，以模拟 32 位的情况；二、在汇编实现的函数内不使用被调用者保存（callee-saved）寄存器，避免在函数调用时需要保存和恢复相应寄存器，而这种保存和恢复是与架构相关的（在 RV32 中是 SW 和 LW，在 RV64 中是 SD 和 LD）

5.2.1 四轮加密与解密

参考背景介绍可知，SM4 使用 F 轮函数进行一轮加密或解密。为了方便表达，本文将四轮加密或解密考虑为一个基本单元，从而每个基本单元的输入为四个字，输出也为四个字，所需轮密钥为四个字。

轮函数 F 运算过程中所需的 τ 运算是对输入的四个字节进行逐字节 SBox 替换，而 sm4ed 指令只能对一个字节做 SBox，故需要通过四条指令来做到完整的 τ 运算，本文记为 sm4ed4；sm4ed4 也同时做了 L 线性变换和与 X_0 异或，故 sm4ed4 等价于轮函数 F 。该过程如代码 5.5 所示。

对于一个基本单元，本文的实现能让储存输入的寄存器在经过运算后恰好储存输出；与此同时，本文的实现重复利用了第 1 轮和第 3 轮产生的 XOR，减少了运算量。

5.2.2 四轮密钥生成

与四轮加密或解密类似，本文将四轮密钥生成考虑为一个基本单元，从而每个基本单元的输入为四个字，输出也为四个字。

轮函数 F' 运算过程中所需的 τ 运算是对输入的四个字节进行逐字节 SBox 替换，而 sm4ks 指令只能对一个字节做 SBox，故需要通过四条指令来做到完整的 τ 运算，本文记为 sm4ks4；sm4ks4 也同时做了 L' 线性变换和与 X_0 异或，故 sm4ks4 等价于轮函数 F' 。该过程如代码 5.6 所示。

与四轮加密或解密类似，四轮密钥生成也能保证输出刚好在储存输入的寄存

代码 5.5　SM4 的四轮加密或解密

```
# sm4ed4 $rd,$rs 是以下过程的缩写
sm4ed   $rd,$rd,$rs,0
sm4ed   $rd,$rd,$rs,1
sm4ed   $rd,$rd,$rs,2
sm4ed   $rd,$rd,$rs,3

# Q0~Q3 储存着输入的四个字
# T0~T3 储存着轮密钥的四个字
# T 和 XOR 均为临时寄存器
# 第 1 轮
xor     $XOR,$Q2,$Q3 # X2 ^ X3
xor     $T,$Q1,$T0 # X1 ^ K0
xor     $T,$T,$XOR # X1 ^ X2 ^ X3 ^ K0
sm4ed4  $Q0,$T # X0 ^ F(T) = X4

# 第 2 轮
# 重复利用 XOR
xor     $T,$Q0,$T1 # X4 ^ K1
xor     $T,$T,$XOR # X2 ^ X3 ^ X4 ^ K1
sm4ed4  $Q1,T # X1 ^ F(T) = X5

# 第 3 轮
xor     $XOR,$Q0,$Q1 # X4 ^ X5
xor     $T,$Q3,$T2 # X3 ^ K2
xor     $T,$T,$XOR # X3 ^ X4 ^ X5 ^ K2
sm4ed4  $Q2,T # X2 ^ F(T) = X6

# 第 4 轮
# 重复利用 XOR
xor     $T,$Q2,$T3 # X6 ^ K3
xor     $T,$T,$XOR # X4 ^ X5 ^ X6 ^ K3
sm4ed4  $Q3,T # X3 ^ F(T) = X7
# Q0~Q3 储存着输出的四个字
```

代码 5.6　SM4 的四轮密钥生成

```
# sm4ks4 $rd,$rs 是以下过程的缩写
sm4ks    $rd,$rd,$rs,0
sm4ks    $rd,$rd,$rs,1
sm4ks    $rd,$rd,$rs,2
sm4ks    $rd,$rd,$rs,3

# Q0~Q3 储存着输入的四个字
# T0~T3 储存着常数轮密钥的四个字
# T 和 XOR 均为临时寄存器
# 第 1 轮
xor      $XOR,$Q2,$Q3 # X2 ^ X3
xor      $T,$Q1,$T0 # X1 ^ K0
xor      $T,$T,$XOR # X1 ^ X2 ^ X3 ^ K0
sm4ks4   $Q0,$T # X0 ^ F(T) = X4

# 第 2 轮
# 重复利用 XOR
xor      $T,$Q0,$T1 # X4 ^ K1
xor      $T,$T,$XOR # X2 ^ X3 ^ X4 ^ K1
sm4ks4   $Q1,T # X1 ^ F(T) = X5

# 第 3 轮
xor      $XOR,$Q0,$Q1 # X4 ^ X5
xor      $T,$Q3,$T2 # X3 ^ K2
xor      $T,$T,$XOR # X3 ^ X4 ^ X5 ^ K2
sm4ks4   $Q2,T # X2 ^ F(T) = X6

# 第 4 轮
# 重复利用 XOR
xor      $T,$Q2,$T3 # X6 ^ K3
xor      $T,$T,$XOR # X4 ^ X5 ^ X6 ^ K3
sm4ks4   $Q3,T # X3 ^ F(T) = X7
# Q0~Q3 储存着输出的四个字
```

器中，以及复用 XOR 运算结果。

5.3　SM3 与哈希函数

由于 RISC-V 标量密码学指令集扩展中均只对 SHA256, SHA512 和 SM3 的部分操作设计了特殊指令，相对于提供一个完整的汇编实现，通过内联汇编的方式改变部分算子、从而复用已有的大部分代码更具可维护性。

对于 SM3，只需要将 P0 变换和 P1 变换在架构支持时内联汇编进入 C 语言实现的函数中。如代码 5.7 所示，在支持 Zksh 扩展且没有关闭汇编支持时，P0 和 P1

代码 5.7　SM3 的内联加速指令汇编

```
#ifndef PEDANTIC
# if defined(__GNUC__) && __GNUC__>=2 && \
     !defined(OPENSSL_NO_ASM) && !defined(
        OPENSSL_NO_INLINE_ASM)
#  if defined(__riscv_zksh)
#   define P0(x) ({ MD32_REG_T ret;          \
                    asm ("sm3p0 %0, %1" \
                    : "=r"(ret)           \
                    : "r"(x)); ret;       })
#   define P1(x) ({ MD32_REG_T ret;          \
                    asm ("sm3p1 %0, %1" \
                    : "=r"(ret)           \
                    : "r"(x)); ret;       })
#  endif
# endif
#endif

# define ROTATE(a,n) \
  (((a)<<(n))|(((a)&0xffffffff)>>(32-(n))))
#ifndef P0
# define P0(X) (X ^ ROTATE(X, 9) ^ ROTATE(X, 17))
#endif
#ifndef P1
# define P1(X) (X ^ ROTATE(X, 15) ^ ROTATE(X, 23))
#endif
```

的宏将会展开成内联 SM3 加速指令汇编；如果没有相关支持，则展开成纯软件实现。

SHA256 和 SHA512 加速指令的使用方式与上述方式类似，此处不做展开。

第 6 章　性能评估

本文使用之前章节中的电路实现和软件实现进行性能评估。对于电路实现，对于 RV64 和 RV32 两种架构，本文使用启用标量密码学扩展的默认配置，在 xc7k325tffg900-2 FPGA 上以 100MHz 运行；对于软件实现，本文使用增加了软件优化实现的 OpenSSL，评估方式为 openssl speed -elapsed -evp algorithm，其中 algorithm 随着所需评估算法的改变而改变。

本章节将展示主要的评估结果。完整数据在提交上游时附带的链接中[①]。

6.1　AES

OpenSSL 中有 AES 的不同实现。其中有与架构无关的使用 C 语言的实现，使用 RV64I 的汇编实现 openssl#17640[②]，使用 RV64 ZKN 加速指令的汇编实现 openssl#18197[③]，使用 RV32 ZKN 加速指令的汇编实现 openssl#18308[④]。在这四个实现中，后两者由本文贡献。

与此同时，AES-GCM 操作模式的运算方式较为特殊。GCM 模式需要运算较为复杂的 CLMUL 操作，而这可以被在 ZBKC 中的 CLMUL 指令加速。在 OpenSSL 中，CLMUL 有与架构无关的使用 C 语言的实现，也有使用 RV64 ZBKC 加速指令的汇编实现 openssl#17640[⑤]。两者均由 OpenSSL 提供。

在 64 位架构中，openssl#18197 能提供最高 10 倍、最低 2 倍的加速比。最高值由 ECB 模式提供，如图 6.1(a) 所示。最低值由 GCM 模式提供，如图 6.1(b) 所示。可以注意到 GCM 模式的运算瓶颈在 CLMUL 上，而并不是在 AES 上，因为如果同时加速 AES 和 CLMUL，GCM 模式也可以得到高达 10 倍的加速比，如图 6.1(c) 所示，故而该最低值并不能代表平均水平。对于常用的模式 CBC 和 CTR 来说，openssl#18197 可以达到 4 到 8 倍的加速比，如图 6.1(d) 和图 6.1(e) 所示。

① AES 评估数据：https://gist.github.com/ZenithalHourlyRate/7b5175734f87acb73d0bbc53391d7140
SM4 评估数据：https://gist.github.com/ZenithalHourlyRate/12b60891cfc70f97b5fb357f458d6a41
SM3 评估数据：https://github.com/openssl/openssl/pull/18287
SHA256 和 SHA512 评估数据：https://github.com/openssl/openssl/pull/16638#issuecomment-1121281173
ChaCha20 评估数据：https://github.com/openssl/openssl/pull/18289
3DES 评估数据：https://github.com/openssl/openssl/pull/18290
② https://github.com/openssl/openssl/pull/17640
③ https://github.com/openssl/openssl/pull/18197
④ https://github.com/openssl/openssl/pull/18308
⑤ https://github.com/openssl/openssl/pull/17640

在 32 位架构中，openssl#18308 能提供最高 4 倍、最低 1.5 倍的加速比。与 32 位架构相同，最高值由 ECB 模式提供，如图 6.2(a) 所示；最低值由 GCM 模式提供，如图 6.2(b) 所示。对于常用的模式 CBC 和 CTR 来说，openssl#18308 可以达到 2 到 3 倍的加速比，如图 6.2(c) 和图 6.2(d) 所示。

6.2 SM4

OpenSSL 中有 SM4 的不同实现。其中有与架构无关的使用 C 语言的实现和使用同时适用与 RV64 和 RV32 的使用 ZKS 加速指令的汇编实现 openssl#18285[①]。后者由本文贡献。

在 64 位架构中，openssl#18285 能提供最高 5 倍，最低 2 倍的加速比。最高值由使用 CLMUL 加速的 GCM 模式提供，如图 6.3(a) 所示。最低值由 GCM 模式提供，如图 6.3(b) 所示。对于常用的模式 CBC 和 CTR 来说，openssl#18285 可以达到 2 到 4 倍的加速比，如图 6.3(c) 和图 6.3(d) 所示。

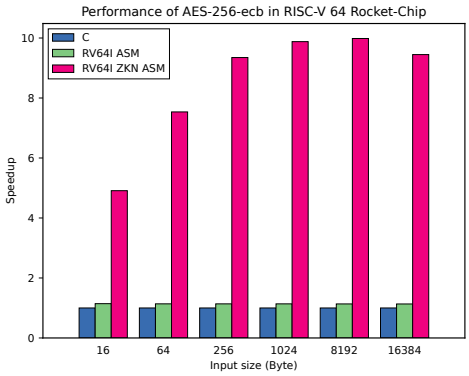
在 32 位架构中，openssl#18285 能提供最高 3.7 倍，最低 1.5 倍的加速比。最高值由 ECB 模式提供，如图 6.4(a) 所示。最低值由 GCM 模式提供，如图 6.4(b) 所示。对于常用的模式 CBC 和 CTR 来说，openssl#18285 可以达到 2 到 3.5 倍的加速比，如图 6.4(c) 和图 6.4(d) 所示。

6.3 哈希函数

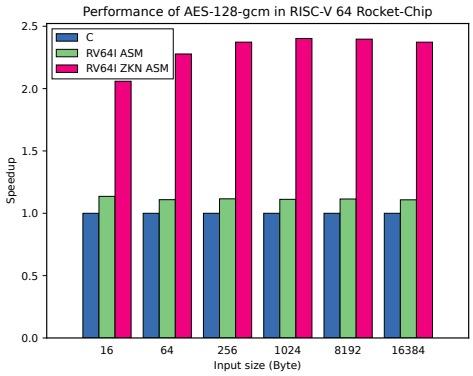
对于 SHA256, SHA512 和 SM3 等哈希函数，可以通过使用内联汇编的方式，只在一些特定操作中使用加速指令，从而不需要完整重新实现。在这种情况下，OpenSSL 提供了与架构无关的使用 C 语言的实现，内联 SHA256 加速指令的实现 openssl#16710[②]，内联 SHA512 加速指令的实现 openssl#16638[③]和内联 SM3 加速指令的实现 openssl#18287[④]。最后一个实现由本文贡献。

如图 6.5 所示，内联加速指令为哈希函数能提供 1.5 倍到 2.9 倍的加速比。

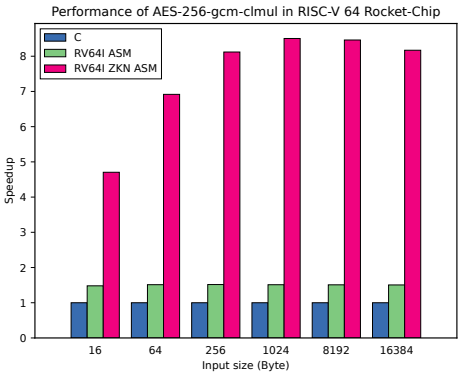
① https://github.com/openssl/openssl/pull/18285
② https://github.com/openssl/openssl/pull/16710
③ https://github.com/openssl/openssl/pull/16638
④ https://github.com/openssl/openssl/pull/18287



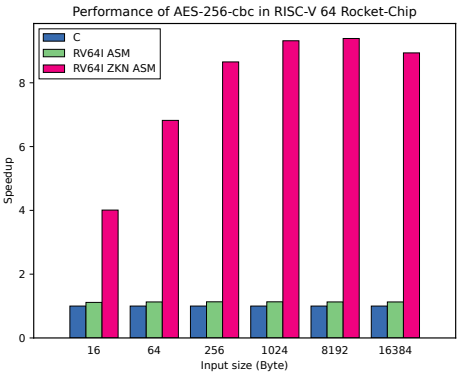
(a) AES-256-ECB



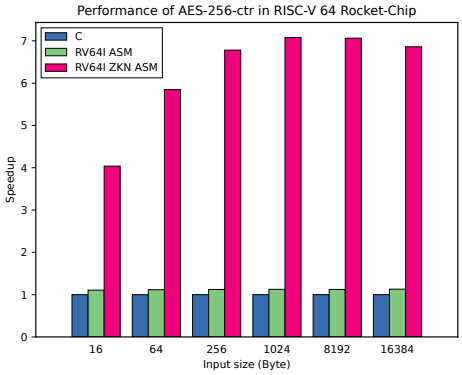
(b) AES-128-GCM



(c) AES-256-GCM-CLMUL

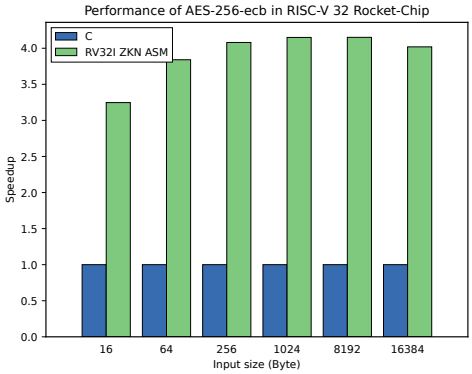


(d) AES-256-CBC

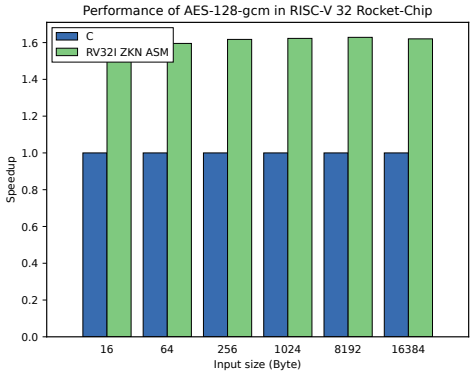


(e) AES-256-CTR

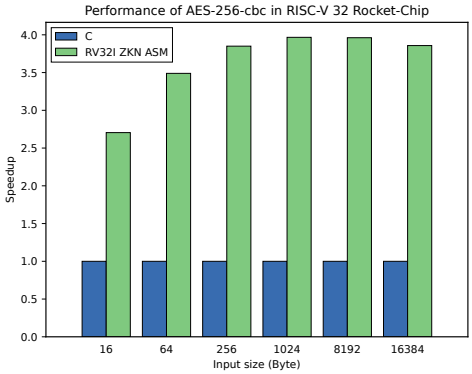
图 6.1　RISC-V 64 Rocket 上 AES 的加速比



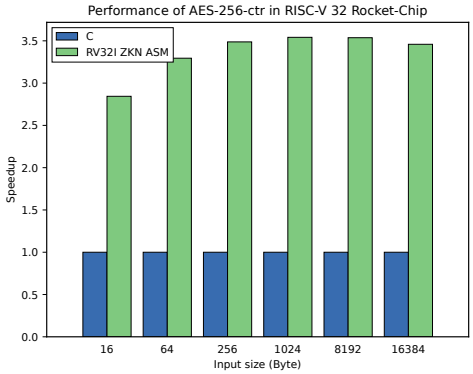
(a) AES-256-ECB



(b) AES-128-GCM



(c) AES-256-CBC



(d) AES-256-CTR

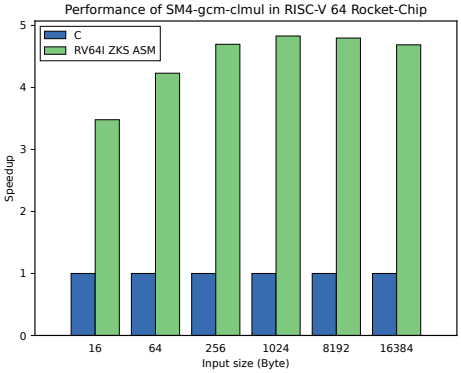
图 6.2　RISC-V 32 Rocket 上 AES 的加速比

6.4　ChaCha20 与 3DES

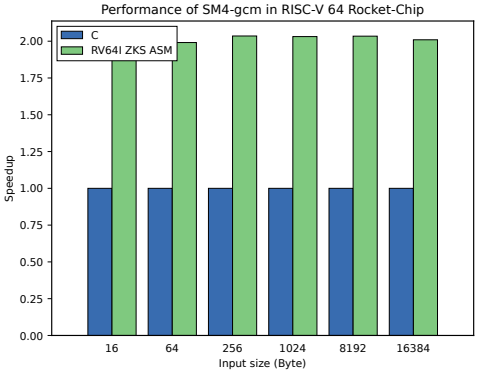
在 RISC-V 标量密码学指令集扩展中，除了特定算法的加速指令，常用的位运算指令也可以被一些密码学算法使用，其中一个指令为循环移位。对 ChaCha20 和 3DES 等算法，OpenSSL 可以通过内联循环移位汇编的方式提供加速。在这种情况下，OpenSSL 提供了与架构无关的使用 C 语言的实现和内联循环移位指令的 ChaCha20 实现 openssl#18289①，内联循环移位指令的 DES 实现 openssl#18290②。这三个实现中的后两者由本文贡献。

如图 6.6(a) 所示，内联循环移位能为 ChaCha20 提供 1.3 倍到 1.7 倍的加速比。如图 6.6(b) 所示，内联循环移位能为 DES 提供 1.16 倍的加速比。

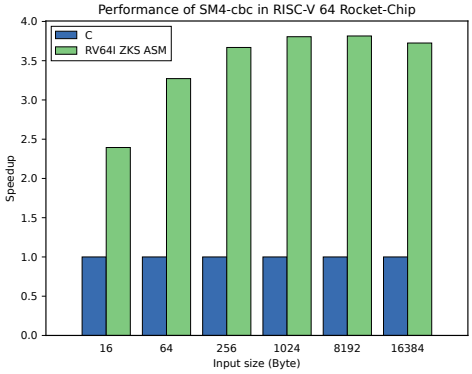
① https://github.com/openssl/openssl/pull/18289
② https://github.com/openssl/openssl/pull/18290



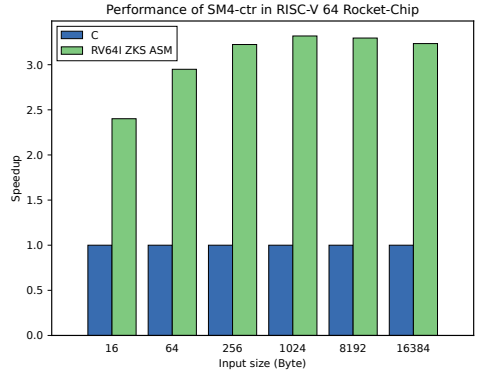
(a) SM4-GCM-CLMUL



(b) SM4-GCM

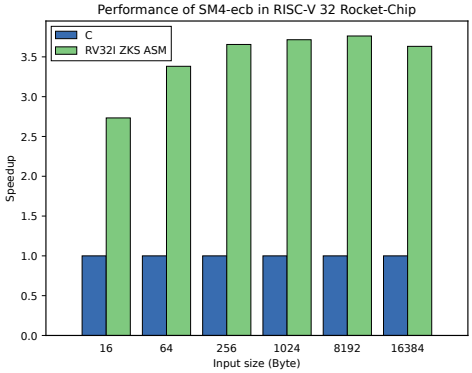


(c) SM4-CBC

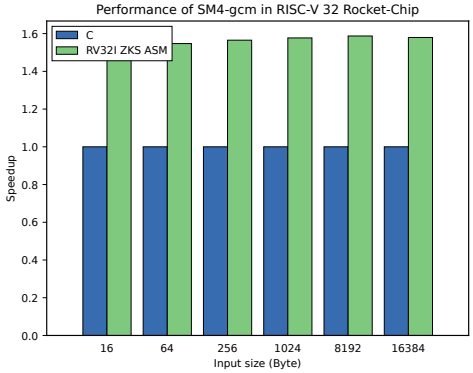


(d) SM4-CTR

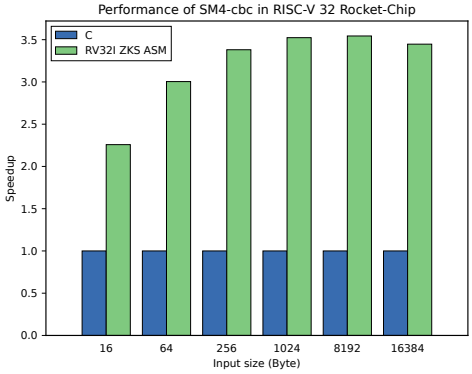
图 6.3　RISC-V 64 Rocket 上 SM4 的加速比



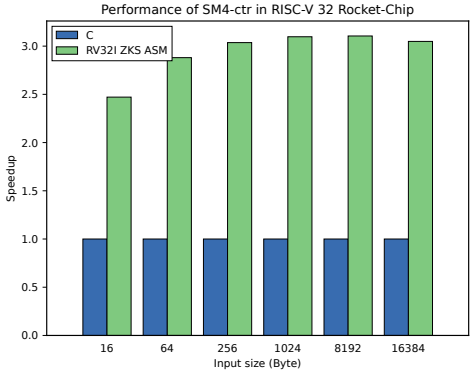
(a) SM4-ECB



(b) SM4-GCM

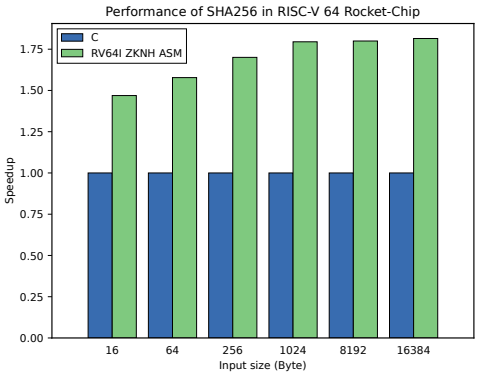


(c) SM4-CBC

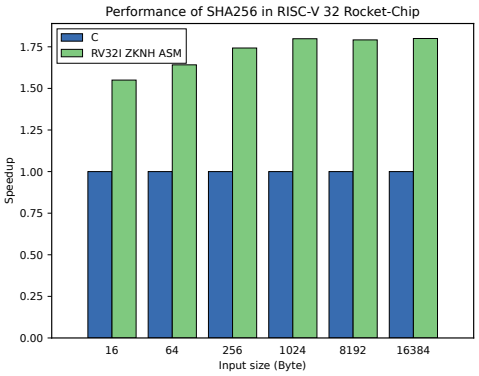


(d) SM4-CTR

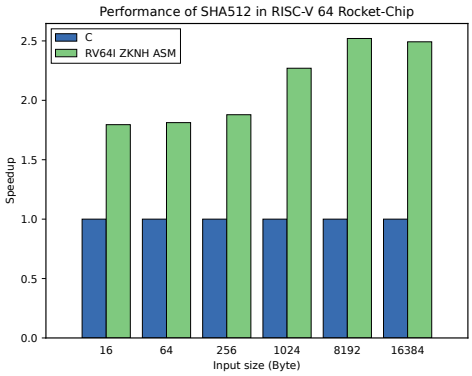
图 6.4　RISC-V 32 Rocket 上 SM4 的加速比



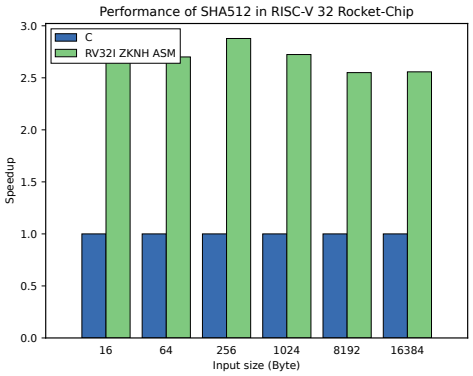
(a) SHA256, RISC-V 64



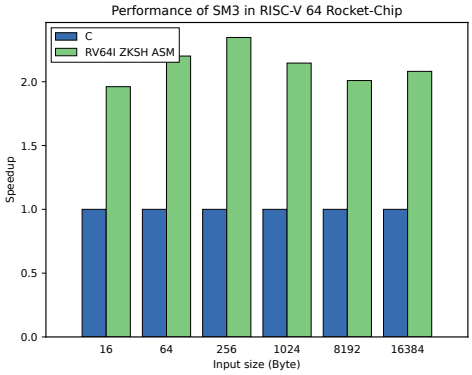
(b) SHA256, RISC-V 32



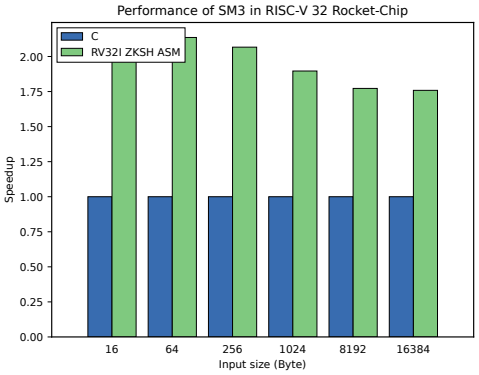
(c) SHA512, RISC-V 64



(d) SHA512, RISC-V 32

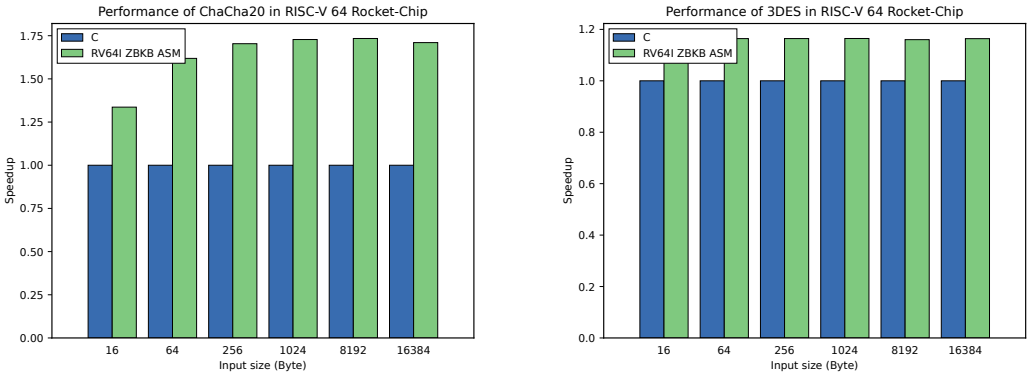


(e) SM3, RISC-V 64



(f) SM3, RISC-V 32

图 6.5　Rocket 上哈希函数的加速比



(a) ChaCha20　　(b) 3DES

图 6.6　RISC-V 64 Rocket 上 ChaCha20 和 3DES 的加速比

第 7 章　结论

本文基于 Chisel 的参数化特性，为 Rocket 实现了同时适用于 RISC-V 64 位和 32 位架构的支持标量密码学指令集扩展的硬件电路单元。本文注重于最小化硬件电路占用的面积，故而致力于合并多个指令的数据通路，最终分别将 AES 和 SM4 的多个指令合并为一个数据通路，并将位运算单元合并进入已有的算术逻辑单元中。本文也指出，SM4 的数据通路可以与 AES 的数据通路进一步合并。

同时，本文使用汇编语言为 OpenSSL 提供了 AES、SM4 和 SM3 的优化软件实现。本文详细讨论了这些软件优化的实现细节，展示了如何使用加速指令实现高效而完整的密码学算法。

本文利用上述的软件实现对上述的电路实现进行性能评估，在 AES 上最高可以达到十倍的加速比，在其他密码学算法上也取得了显著的加速。

本文也详细讨论了 RISC-V 标量密码学指令集扩展的设计。在电路实现和软件实现的两个章节中，本文从硬件和软件的两个角度展示了硬件和软件的桥梁——指令集架构——能如何影响硬件设计和软件编写，从而带来不同的优势和代价。

本文的所有实现都提交了上游并得到了积极的回应，预期会成为上述软件的一部分。由于 Rocket 和 OpenSSL 影响广大，随着 RISC-V 开放生态的不断发展，本文作者期待本文的工作能走进千家万户。

插图索引

表格索引

参考文献

[1] FRANKEL S, KRISHNAN S. Ip security (ipsec) and internet key exchange (ike) document roadmap: 6071[R/OL]. RFC Editor, 2011. http://www.rfc-editor.org/rfc/rfc6071.txt.

[2] DIERKS T, RESCORLA E. The transport layer security (tls) protocol version 1.2: 5246[R/OL]. RFC Editor, 2008. http://www.rfc-editor.org/rfc/rfc5246.txt.

[3] Chromium Blog. A safer default for navigation: Https[EB/OL]. [2022-03-20]. https://blog.chromium.org/2021/03/a-safer-default-for-navigation-https.html.

[4] Electronic Frontier Foundation. Https everywhere[EB/OL]. [2022-03-20]. https://www.eff.org/https-everywhere.

[5] RESCORLA E. Http over tls: 2818[R/OL]. RFC Editor, 2000. http://www.rfc-editor.org/rfc/rfc2818.txt.

[6] Microsoft Corporation. Bitlocker[EB/OL]. [2022-03-20]. https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview.

[7] Clemens Fruhwirth. Luks1 on-disk format specification version 1.2.3[EB/OL]. [2022-03-20]. https://gitlab.com/cryptsetup/cryptsetup/-/wikis/LUKS-standard/on-disk-format.pdf.

[8] FIELDING R T, GETTYS J, MOGUL J C, et al. Hypertext transfer protocol – http/1.1: 2616[R/OL]. RFC Editor, 1999. http://www.rfc-editor.org/rfc/rfc2616.txt.

[9] DWORKIN M, BARKER E, NECHVATAL J, et al. Advanced encryption standard (aes)[M/OL]. Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, 2001. DOI: https://doi.org/10.6028/NIST.FIPS.197.

[10] 国家密码管理局. SM4 分组密码算法[Z]. 2012.

[11] JONSSON J, KALISKI B. Public-key cryptography standards (pkcs) #1: Rsa cryptography specifications version 2.1: 3447[R/OL]. RFC Editor, 2003. http://www.rfc-editor.org/rfc/rfc3447.txt.

[12] RESCORLA E. The transport layer security (tls) protocol version 1.3: 8446[R]. RFC Editor, 2018.

[13] NIR Y, LANGLEY A. Chacha20 and poly1305 for ietf protocols: 7539[R/OL]. RFC Editor, 2015. http://www.rfc-editor.org/rfc/rfc7539.txt.

[14] HODGES J, MANDYAM G, JONES M. Registries for web authentication (webauthn): 8809[R]. RFC Editor, 2020.

[15] BAFANDEHKAR M, YASIN S M, MAHMOD R, et al. Comparison of ecc and rsa algorithm in resource constrained devices[C/OL]//2013 International Conference on IT Convergence and Security (ICITCS). 2013: 1-3. DOI: 10.1109/ICITCS.2013.6717816.

[16] KOBLITZ N. Elliptic curve cryptosystems[J]. Mathematics of Computation, 1987, 48(177): 203-209.

[17] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS[EB/OL]. 2003. www.openssl.org.

[18] ZHANG Y, WANG S, ZHANG X, et al. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture[M/OL]. IEEE Press, 2021: 416–428. https://doi.org/10.1109/ISCA52012.2021.00040.

[19] MARSHALL B, NEWELL G R, PAGE D, et al. The design of scalar AES Instruction Set Extensions for RISC-V[J/OL]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2021: 109-136[2022-01-22]. https://tches.iacr.org/index.php/TCHES/article/view/8729. DOI: 10.46586/tches.v2021.i1.109-136.

[20] Intel Corporation. Intel® aes new instructions (intel® aes-ni)[EB/OL]. [2022-03-20]. https://www.intel.com/content/www/us/en/architecture-and-technology/advanced-encryption-standard-aes/data-protection-aes-general-technology.html.

[21] Shay Gueron and Michael E. Kounavis. Intel carry-less multiplication instruction and its usage for computing the gcm mode[EB/OL]. [2022-03-20]. https://www.intel.com/content/dam/develop/external/us/en/documents/clmul-wp-rev-2-02-2014-04-20.pdf.

[22] LIMITED A. Arm architecture reference manual for a-profile architecture[Z]. 2011.

[23] WATERMAN A. Design of the risc-v instruction set architecture: UCB/EECS-2016-1[D/OL]. EECS Department, University of California, Berkeley, 2016. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html.

[24] Chipsalliance. Rocket chip generator[EB/OL]. [2022-03-20]. https://github.com/chipsalliance/rocket-chip.

[25] OpenXiangshan. Xiangshan[EB/OL]. [2022-03-20]. https://github.com/OpenXiangShan/XiangShan.

[26] RISC-V International. Risc-v cryptography extensions volume i scalar & entropy source instructions[EB/OL]. [2022-03-20]. https://github.com/riscv/riscv-crypto/releases/download/v1.0.1-scalar/riscv-crypto-spec-scalar-v1.0.1.pdf.

[27] HANSEN T, 3RD D E E. Request for comments: number 4634 US Secure Hash Algorithms (SHA and HMAC-SHA)[M/OL]. RFC Editor, 2006. https://www.rfc-editor.org/info/rfc4634. DOI: 10.17487/RFC4634.

[28] 国家密码管理局. SM3 密码杂凑算法[Z]. 2012.

[29] Ben Marshall. Risc-v crypto rtl[EB/OL]. [2022-03-20]. https://github.com/riscv/riscv-crypto/tree/master/rtl.

[30] Ben Marshall. Risc-v crypto benchmarks[EB/OL]. [2022-05-25]. https://github.com/riscv/riscv-crypto/tree/master/benchmarks.

[31] Markku-Juhani O. Saarinen. Algorithm tests for risc-v crypto extension[EB/OL]. [2022-05-25]. https://github.com/rvkrypto/rvkrypto-fips.

[32] Arch Linux. Arch linux - openssl[EB/OL]. [2022-05-25]. https://archlinux.org/packages/core/x86_64/openssl/.

[33] University of California, Berkeley. Chisel/firrtl[EB/OL]. [2022-03-20]. https://www.chisel-lang.org/.

[34] riscv-software-src. Spike risc-v isa simulator[EB/OL]. [2022-03-20]. https://github.com/riscv-software-src/riscv-isa-sim.

[35] riscv-non-isa. Risc-v architecture test[EB/OL]. [2022-04-24]. https://github.com/riscv-non-isa/riscv-arch-test.

[36] NYBERG K. Differentially uniform mappings for cryptography[C]//HELLESETH T. Advances in Cryptology — EUROCRYPT '93. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994: 55-64.

[37] BOYAR J, PERALTA R. A small depth-16 circuit for the aes s-box[C]//GRITZALIS D, FURNELL S, THEOHARIDOU M. Information Security and Privacy Research. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012: 287-298.

[38] RISC-V International. Risc-v bit-manipulation isa-extensions[EB/OL]. [2022-04-24]. https://github.com/riscv/riscv-bitmanip/releases/download/1.0.0/bitmanip-1.0.0.pdf.

致　　谢

我是一个不善于表达情感的人，尤其是在这种表达感谢的时候。所以说，正文其实已经完成了几天，但致谢一直都没头绪怎么下笔。其实，如果只是单纯罗列还是很好罗列的，只是不知道怎么组织语言罢了。

这篇毕业论文的题目在去年就有了头绪，是在一次吃饭的时候，和一些相关人士聊到一个项目能不能有一个 RISC-V 的移植版本，当时我就觉得非常 exciting，但大家都知道，做事嘛，依赖着别人来做是不成的，得自己做；当时也机缘巧合，缺一些东西，也刚新出了一些东西，思前想后，决定自己下场干一把。

当时我是在中科院软件所 PLCT 实验室的 Arch Linux 组做一些打包工作，我们戏称为开源社区综合实践，毕竟发行版嘛，是整个生态的最终端，所以打包的工作就是和上游打架。当时做得还算成功吧（这里就要感谢最好的本科生教育了，以及感谢高鸣宇导师给出的自由度，让我有足够多闲暇的时间自己折腾），从而对 RISC-V 有了一定的了解。打包固然重要，但带来的是一种持续的维护感，是细水长流，并没有像开发东西那样能给我带来快感（所以我很敬佩肥猫，苦行僧一样非常禁欲）；再者，我已经够多需要维护的业务了，再多一些可能就比较同质化，所以需要一些新的感官上的刺激，这时候我就找到了上面的相关人士之一，Sequencer，来看看能不能做一些开发的工作。

从去年开始，其实都一直是在积累经验（好吧，就是摸鱼），干一些杂活，比如对 Chisel 的了解啦，对 RISC-V 的了解啦，对 Sequencer 写电路的理念的了解啦，今年立项以后才开始真正发力，然后发现真的可以非常迅速，也真的可以非常刺激，比如一个晚上就能做出来大新闻，然后就可以带来很大的 impact。

当然，这里也感到，尽管一个晚上能干完开发，但追踪上游和做成生态到最后真正能有 impact，还是很累很需要时间的。发论文也只是一个开头，但真正落地和走进千家万户，还有一段距离。指令集也是类似的，大家都在说 RISC-V 指令集本身，其实指令集本身是平凡的，但与之配套的生态的建设是困难的。自主或者自由的指令集并不代表什么，任何一个经过良好培训的本科生都能一个晚上做到，但要把自主或者自由的生态搞得红红火火，从 CPU 到工具链到发行版，那就不是一个晚上能做完的了。

这里还是很感谢《大教堂与集市》这本书，大概是高中的时候读过的了吧，当

时就告诉我大教堂这条道路最终是行不通的，这么久下来，我也确实感到这条路非常难走，因为现代计算机系统的复杂度远远不是一个人或者一个公司就能搞定的；集市这种形容确实非常精妙，我在零散的给上游贡献的过程中，比如内核，确实感到自己只是集市的一部分，随来随走，并不持续，只是在有兴趣的时候来参与一下，但并不想完全成为维护者。大家总说开源拖拉机，但这辆破车就是在大量像我这样的零散而随机的贡献下行驶起来的，并成为了现代文明的基石。

还是讲回毕设本身吧。我干开源的活其实有个习惯，喜欢把过程发到群里让群友围观我干活，边吐槽边写，所以在这个过程中，自然得到了愿意（这个“愿意”是最宝贵的）围观我的群友的大量帮助，比如德拉姆，比如 Sequencer，比如 cyyself，这里还有一百多个群友，就不一一罗列了。

在 FPGA 实验平台的搭建上，要特别感谢 cyyself 的大力支持；在电路实现的测试上，要特别感谢 Phantom1003 给出的模糊测试；这里也要感谢 Andrew Waterman 在开工作会议时候给出的一些方向上的指引，以及 Sequencer、Jerry Zhao、OpenSSL 基金会的工作人员在评审代码时给出的宝贵意见；同时也要感谢高鸣宇导师对本毕业论文的监督与修改。

最后，最需要感谢的还是自己的干活本身。