RISC-V Cryptography Extensions Volume I

Scalar & Entropy Source Instructions

Version 0.9.0, 04/2021

Table of Contents

Preface

Contributors to all versions of the specification in alphabetical order (please contact editors to suggest corrections):

Alexander Zeh, Andy Glew, Barry Spinney, Ben Marshall (Editor), Daniel Page, Derek Atkins, Ken Dockser, Markku-Juhani O. Saarinen, Nathan Menhorn, L Peter Deutsch, Richard Newell, Claire Wolf

Document Version Information:

dev/next-release @ a8cd3bb6e47be35741934b05543655e8fb2e1065

See github.com/riscv/riscv-crypto for more information.

Chapter 1. Introduction

This document describes the *scalar* cryptography extension for RISC-V. All instructions described herein use the general purpose `X` registers, and obey the 2-read-1-write register access constraint. These instructions are designed to be lightweight and suitable for `32` and `64` bit base architectures; from embedded IoT class cores to large, application class cores which do not implement a vector unit.

This document also describes the architectural interface to an Entropy Source, which can be used to generate cryptographic secrets. This is found in Chapter 4.

It also contains a mechanism allowing core implementers to provide *"Constant Time Execution"* guarantees in Chapter 5.

A companion document *Volume II: Vector Instructions*, describes instruction proposals which build on the RISC-V Vector Extension.

1.1. Intended Audience

Cryptography is a specialist subject, requiring people with many different backgrounds to cooperate in its secure and efficient implementation. Where possible, we have written this specification to be understandable by all, though we recognise that the motivations and references to algorithms or other specifications and standards may be unfamiliar to those who are not domain experts.

This specification anticipates being read and acted on by various people with different backgrounds. We have tried to capture these backgrounds here, with a brief explanation of what we expect them to know, and how it relates to the specification. We hope this aids peoples understanding of which aspects of the specificaiton are particularly relevent to them, which they may (safely!) ignore, and pass to a colleague.

Cryptographers and cryptographic software developers

These are the people we expect to write code using the instructions in this specification. They should understand fairly obviously the motivations for the instructions we include, and be familiar with most of the algorithms and outside standards which we refer to. We expect the sections on constant time exection (Chapter 5) and the entropy source (Chapter 4) to be chiefly understood with their help.

Computer architects

We do not expect architects to have a cryptography background. We nonetheless expect architects to be able to examine our instructions for implementation issues, understand how the instructions will be used in context, and advise on how best to fit the functionality the cryptographers want to the ISA interface.

Digital design engineers & micro-architects

These are the people who will implement the specification inside a core. Again, no cryptography expertise is assumed, but we expect them to interpret the specification and anticipate any hardware implementation issues. E.g., where high-frequency design considerations apply, or

where latency/area tradeoffs exist etc. In particular, they should be aware of the literature around efficiently implementing AES and SM4 SBoxes in hardware.

Verification engineers

Responsible for ensuring the correct implementation of the extension in hardware. No cryptography background is assumed. We hope they are able to identify interesting test cases from the specification, and knowing how the instructions are used in the real world. We do not expect verification engineers in this sense to be experts in entropy source design or certification, since this is a very specialised area. We do expect them however to identify all of the *architectural* test cases around the entropy source interface.

These are by no means the only people concerned with the specification, but they are the ones we considered most while writing it.

1.2. Sail Specifications

RISC-V maintains a formal model of the ISA specification, implemented in the Sail ISA specification language \cite{sail}. Note that *Sail* refers to the specification language itself, and that there is a *model of RISC-V*, written using Sail. It is not correct to refer to "the Sail model". This is ambiguous, given there are many models of different ISAs implemented using Sail. We refer to the Sail implementation of RISC-V as "the RISC-V Sail model".

The Cryptography extension uses inline Sail code snippets from the actual model to give canonical descriptions of instruction functionality. Each instruction is accompanied by its expression in Sail, and includes calls to supporting functions which are too verbose to include directly in the specificaiton. This supporting code is listed in Appendix D. The Sail Manual is recommended reading in order to best understand the code snippets.

1.3. Policies

In creating this proposal, we tried to adhere to the following policies:

- Where there is a choice between: 1) supporting diverse implementation strategies for an algorithm or 2) supporting a single implementation style which is more performant / less expensive; the crypto extension will pick the more constrained but performant option. This fits a common pattern in other parts of the RISC-V specification, where recommended (but not required) instruction sequences for performing particular tasks are given as an example, such that both hardware and software implementers can optimise for only a single use-case.
- The extension will be designed to support *existing* standardised cryptographic constructs well. It will not try to support proposed standards, or cryptographic constructs which exist only in academia. Cryptographic standards which are settled upon concurrently with, or after the RISC-V cryptographic extension standardisation will be dealt with by future additions to, or versions of, the RISC-V cryptographic standard extension. It is anticipated that the NIST Lightweight Cryptography contest, and the NIST Post-Quantum Cryptography contest may be dealt with this way, depending on timescales.
- Historically, there has been some discussion \cite{LSYRR:04} on how newly supported operations in general purpose computing might enable new bases for cryptographic algorithms.

The standard will not try to anticipate new useful low level operations which *may* be useful as building blocks for future cryptographic constructs.

- Regarding side-channel countermeasures: Where relevant, proposed instructions must aim to remove the possibility of any timing side-channels. For side-channels based on power or electro-magnetic (EM) measurements, the extension will not aim to support countermeasures which are implemented above the ISA abstraction layer. Recommendations will be given where relevant on how micro-architectures can implement instructions in a power/EM side-channel resistant way.

Chapter 2. Implementation Profiles

All instructions in the scalar cryptography extension are grouped into *functional sets* and *feature sets*. Functional sets are very fine grained, and are constructed around specific algorithms, standards requirements, or small logical groupings of instructions. Feature sets are more coarse grained, and are what cores are expected to implement. The letters used to construct Functional Set names are explained in Table 1.

The Feature Sets for instructions exclusive to the scalar cryptography extension are listed in Table Table 2. All Bit-manipulation instructions used by the scalar cryptography extension (Section 3.1) are always included in the feature sets listed in Table Table 2. Exceptions are RV64 only instructions, which are not included in RV32 based implementations.

Table 1. Explanation of the functional sets.

Functional Set	Description
`Zkg`	Constant time carry-less multiply for Galois/Counter Mode.
`Zkb`	Bitmanip subset included in the scalar cryptography extension, minus those in Zk.
`Zkr`	Entropy source for seeding random number generators.
`Zkne`	NIST AES Encryption Instructions.
`Zknd`	NIST AES Decryption Instructions.
`Zknh`	NIST SHA2 Hash function instructions.
`Zksed`	SM4 Instructions.
`Zksh`	SM3 Hash function instructions.

Table 2. Explanation of the feature strings used to refer to the functional sets.

Feature Set	Description
`K`	The default scalar cryptography extension, short for `Zkn_Zkr`
`Zkn`	NIST algorithm suite. Short for `Zkne_Zknd_Zknh_Zkg_Zkb`.
`Zks`	ShangMi (SM) algorithm suite. Short for `Zksed_Zksh_Zkg_Zkb`.

Encryption and decryption instructions are separated into different functional groups because some popular use cases (e.g., Galois/Counter Mode in TLS 1.3, among others) do not require decryption functionality. The NIST and ShangMi algorithms suites are separated because their usefulness is heavily dependent on the countries a device is expected to operate in. NIST ciphers are a part of most standardised internet protocols, while ShangMi ciphers are required for use in China.

Presence of the cryptography extension in any form is indicated by bit `10` of the MISA CSR. i.e. bit `K`, because `C` is taken and `K` is for *Kappa*, the first letter of the ancient Greek word *kruptós*, meaning *hidden*. Detection of fine-grained functionality uses the mechanisms defined by the tech-config RISC-V Task Group. At the time of writing, these mechanisms are still being defined.

Some example GCC `-march=` strings:

- `rv32ik` - Implement the 32-bit NIST feature set (Zkn, the entropy source feature set (Zkr) and all of the 32-bit Bit-manipulation instructions used by the scalar cryptography extension listed in Section 3.1.
- `rv64ik` - As above, but implementing the 64-bit version of the NIST feature set, and additional 64-bit instructions from the Bit-manipulation subset.
- `rv64i_Zks_Zkr` - Implement the Entropy Source instructions, the 64-bit Bit-manipulation instructions, and the XLEN independent ShangMi suite instructions.
- `rv64i_Zkne_Zknh` - Implement only the 64-bit NIST AES Encryption and hash function instructions.

Table 3. Feature sets for instructions in the scalar cryptography extension.

Instruction	Functional Set	Zkn (RV32)	Zkn (RV64)	Zks (RV32)	Zks (RV64)	Zkr
aes32dsi	Zknd	x				
aes32dsmi	Zknd	x				
aes32esi	Zkne	x				
aes32esmi	Zkne	x				
aes64ds	Zknd		x			
aes64dsm	Zknd		x			
aes64es	Zkne		x			
aes64esm	Zkne		x			
aes64im	Zknd		x			
aes64ks1i	Zkne		x			
aes64ks2	Zkne		x			
sha256sig0	Zknh	x	x			
sha256sig1	Zknh	x	x			
sha256sum0	Zknh	x	x			
sha256sum1	Zknh	x	x			
sha512sig0h	Zknh	x				
sha512sig0l	Zknh	x				
sha512sig1h	Zknh	x				
sha512sig1l	Zknh	x				
sha512sum0r	Zknh	x				
sha512sum1r	Zknh	x				
sha512sig0	Zknh	x				

Instruction	Functional Set	Zkn (RV32)	Zkn (RV64)	Zks (RV32)	Zks (RV64)	Zkr
sha512sig1	Zknh	x				
sha512sum0	Zknh	x				
sha512sum1	Zknh	x				
sm3p0	Zksh			x	x	
sm3p1	Zksh			x	x	
sm4ed	Zksed			x	x	
sm4ks	Zksed			x	x	
pollentropy	Zkr					x
getnoise	Zkr					x
clmul, clmulh	Zkg	x	x	x	x	
xperm.n, xperm.b	Zkb	x	x	x	x	
ror, rol, rori	Zkb	x	x	x	x	
rorw, rolw, roriw	Zkb		x		x	
andn, orn, xnor	Zkb	x	x	x	x	
pack, packu, packh	Zkb	x	x	x	x	
packw, packuw	Zkb		x		x	
rev.b, rev8 (grevi)	Zkb	x	x	x	x	
rev8.w (grevi)	Zkb		x		x	
zip (shfli)	Zkb	x		x		
unzip (unshfli)	Zkb	x		x		

Chapter 3. Scalar Cryptography Extension

As per the RISC-V Cryptographic Extensions Task Group charter: *"The committee will also make ISA extension proposals for lightweight scalar instructions for 32 and 64 bit machines that improve the performance and reduce the code size required for software execution of common algorithms like AES and SHA and lightweight algorithms like PRESENT and GOST."*

For context, some of these instructions have been developed based on academic work at the University of Bristol as part of the XCrypto project \cite{MPP:19}, and work by Paris Telecom on acceleration of lightweight block ciphers \cite{TGMGD:19}.

3.1. Bitmanip Instructions for Cryptography: ZKb

todo - convert to asciidoc.

3.2. Scalar AES Instructions

todo - convert to asciidoc.

3.3. Scalar SHA2 Instructions

todo - convert to asciidoc.

3.4. Scalar SM3 Instructions

todo - convert to asciidoc.

3.5. Scalar SM4 Instructions

todo - convert to asciidoc.

Chapter 4. Entropy Source

Needs converting from LaTeX

Chapter 5. Data Independent Execution Latency Subset: ZKt

todo - merge in existing asciidoc document found here.

Chapter 6. Bibliography

This is a placeholder file while bibliography is being implemented.

Appendix A: Instruction Encodings

Needs converting from LaTeX

Appendix B: Entropy Source: Rationale and Discussion

Needs converting from LaTeX

Appendix C: Supplementary Materials

Needs converting from LaTeX

Appendix D: Supporting Sail Code

Needs converting from LaTeX