



RISC-V Cryptography Extensions

Volume II

Vector Instructions

Version v1.0.0, 05 October 2023:

Table of Contents

Colophon	1
Acknowledgments	2
1. Introduction	3
1.1. Intended Audience	3
1.2. Sail Specifications	4
1.3. Policies	4
1.4. Element Groups	5
1.5. Instruction Constraints	6
1.6. Vector-Scalar Instructions	7
1.7. Software Portability	8
2. Extensions Overview	10
2.1. Zvbb - Vector Basic Bit-manipulation	10
2.2. Zvbc - Vector Carryless Multiplication	11
2.3. Zvkb - Vector Cryptography Bit-manipulation	12
2.4. Zvkg - Vector GCM/GMAC	13
2.5. Zvkned - NIST Suite: Vector AES Block Cipher	14
2.6. Zvknh[ab] - NIST Suite: Vector SHA-2 Secure Hash	15
2.7. Zvkshed - ShangMi Suite: SM4 Block Cipher	16
2.8. Zvksh - ShangMi Suite: SM3 Secure Hash	17
2.9. Zvkn - NIST Algorithm Suite	18
2.10. Zvknc - NIST Algorithm Suite with carryless multiply	19
2.11. Zvkng - NIST Algorithm Suite with GCM	20
2.12. Zvks - ShangMi Algorithm Suite	21
2.13. Zvksc - ShangMi Algorithm Suite with carryless multiplication	22
2.14. Zvksg - ShangMi Algorithm Suite with GCM	23
2.15. Zvkt - Vector Data-Independent Execution Latency	24
2.15.1. All Zvbb instructions:	24
2.15.2. All Zvbc instructions	25
2.15.3. add/sub	25
2.15.4. add/sub with carry	25
2.15.5. compare and set	25
2.15.6. copy	25
2.15.7. extend	25
2.15.8. logical	25
2.15.9. multiply	26
2.15.10. multiply-add	26
2.15.11. Integer Merge	26
2.15.12. permute	26

2.15.13. shift	26
2.15.14. slide	27
3. Instructions	29
3.1. vaesdf.[vv,vs]	29
3.2. vaesdm.[vv,vs]	31
3.3. vaesef.[vv,vs]	33
3.4. vaesem.[vv,vs]	35
3.5. vaeskf1.vi	37
3.6. vaeskf2.vi	39
3.7. vaesz.vs	41
3.8. vandn.[vv,vx]	43
3.9. vbrev.v	45
3.10. vbrev8.v	46
3.11. vclmul.[vv,vx]	47
3.12. vclmulh.[vv,vx]	49
3.13. vclz.v	51
3.14. vcpop.v	52
3.15. vctz.v	53
3.16. vghsh.vv	54
3.17. vgmul.vv	56
3.18. vrev8.v	58
3.19. vrol.[vv,vx]	59
3.20. vror.[vv,vx,vi]	61
3.21. vsha2c[hl].vv	63
3.22. vsha2ms.vv	66
3.23. vsm3c.vi	69
3.24. vsm3me.vv	72
3.25. vsm4k.vi	75
3.26. vsm4r.[vv,vs]	78
3.27. vwsll.[vv,vx,vi]	81
4. Bibliography	83
5. Encodings	84
Appendix A: Crypto Vector Cryptographic Instructions	84
Appendix B: Vector Bitmanip and Carryless Multiply Instructions	85
Appendix C: Supporting Sail Code	89

Colophon

This document describes the Vector Cryptography extensions to the RISC-V Instruction Set Architecture.

This document is *Ratified*. No changes are allowed. Any desired or needed changes can be the subject of a follow-on new extension. Ratified extensions are never revised. For more information, see [here](#).



Copyright and licensure:

This work is licensed under a [Creative Commons Attribution 4.0 International License](#)



Document Version Information:

main @ 1769c2609bf4535632e0c0fd715778f212bb272e

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

Acknowledgments

Contributors to this specification (in alphabetical order) include:

Allen Baum, Barna Ibrahim, Barry Spinney, Ben Marshall, Derek Atkins, [Ken Dockser](#) (Editor), Markku-Juhani O. Saarinen, Nicolas Brunie, Richard Newell

We are all very grateful to the many other people who have helped to improve this specification through their comments, reviews, feedback and questions.

Chapter 1. Introduction

This document describes the proposed *vector* cryptography extensions for RISC-V. All instructions proposed here are based on the Vector registers. The instructions are designed to be highly performant, with large application and server-class cores being the main target. A companion document *Volume I: Scalar & Entropy Source Instructions*, describes cryptographic instruction proposals for smaller cores which do not implement the vector extension.

1.1. Intended Audience

Cryptography is a specialized 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 recognize 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 people's understanding of which aspects of the specification are particularly relevant to them, and which they may (safely!) ignore or 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 the motivations for the instructions we include, and be familiar with most of the algorithms and outside standards to which we refer.

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.

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

These people are responsible for ensuring the correct implementation of the extensions in hardware. No cryptography background is assumed. We expect them to identify interesting test cases from the specification. An understanding of their real-world usage will help with this.

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 [1]. Note that *Sail* refers to the specification language itself, and that there is a *model of RISC-V*, written using Sail.

It was our intention to include actual Sail code in this specification. However, the Vector Crypto Sail model needs the Vector Sail model as a basis on which to build. This Vector Cryptography extensions specification was completed before there was an approved RISC-V Vector Sail Model. Therefore, we don't have any Sail code to include in the instruction descriptions. Instead we have included Sail-like pseudo code. While we have endeavored to adhere to Sail syntax, we have taken some liberties for the sake of simplicity where we believe that that our intent is clear to the reader.



Where variables are concatenated, the order shown is how they would appear in a vector register from left to right. For example, an element group specified as `{a, b, e, f}` would appear in a vector register with `a` having the highest element index of the group and `f` having the lowest index of the group.

For the sake of brevity, our pseudo code does not include the handling of masks or tail elements. We follow the *undisturbed* and *agnostic* policies for masks and tails as described in the **RISC-V "V" Vector Extension** specification. Furthermore, the code does not explicitly handle overlap and SEW constraints; these are, however, explicitly stated in the text.

In many cases the pseudo code includes calls to supporting functions which are too verbose to include directly in the specification. This supporting code is listed in [Appendix C](#).

The [Sail Manual](#) is recommended reading in order to best understand the code snippets. Also, the [The Sail Programming Language: A Sail Cookbook](#) is a good reference that is in the process of being written.

For the latest RISC-V Sail model, refer to the formal model Github [repository](#).

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 vector crypto extensions will pick the more constrained but performant option. This fits a common pattern in other parts of the RISC-V specifications, where recommended (but not required) instruction sequences for performing particular tasks are given as an example, such that both hardware and software implementers can optimize for only a single use-case.
- The extensions will be designed to support *existing* standardized 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 vector cryptographic extensions standardization will be dealt with by future RISC-V vector cryptographic standard extensions.

- Historically, there has been some discussion [4] 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. All instructions shall be implemented with data-independent timing. That is, the latency of the execution of these instructions shall not vary with different input values.

1.4. Element Groups

Many vector crypto instructions operate on operands that are wider than elements (which are currently limited to 64 bits wide). Typically, these operands are 128- and 256-bits wide. In many cases, these operands are comprised of smaller operands that are combined (for example, each SHA-2 operand is comprised of 4 words). However, in other cases these operands are a single value (for example, in the AES round instructions, each operand is 128-bit block or round key).

We treat these operands as a vector of one or more *element groups* as defined in the [RISC-V Vector Element Groups](#) specification.

Each vector crypto instruction that operates on element groups explicitly specifies their three defining parameters: EGW, EGS, and EEW.

Instruction Group	Extension	EGW	EEW	EGS
AES	Zvkned	128	32	4
SHA256	zvknh[ab]	128	32	4
SHA512	zvknhb	256	64	4
GCM	Zvkg	128	32	4
SM4	Zvkse	128	32	4
SM3	Zvksh	256	32	8



- Element Group Width (**EGW**) - total number of bits in an element group
- Effective Element Width (**EEW**) - number of bits in each element
- Element Group Size (**EGS**) - number of elements in an element group

For all of the vector crypto instructions in this specification, **EEW=SEW**.



The required **SEW** for each cryptographic instruction was chosen to match what is typically needed for other instructions when implementing the targeted algorithm.

- A **Vector Element Group** is a vector of one or more element groups.
- A **Scalar Element Group** is a single element group.

Element groups can be formed across registers in implementations where **VLEN** < **EGW** by using an

LMUL>1.



Since the the **vector extension for application processors** requires a minimum of VLEN of 128, at most such implementations would require LMUL=2 to form the largest element groups in this specification.

However, implementations with a smaller VLEN, such as embedded designs, will requires a larger LMUL to form the necessary element groups. It is important to keep in mind that this reduces the number of register groups available such that it may be difficult or impossible to write efficient code for the intended cryptographic algorithms.

For example, an implementation with VLEN=32 would need to set LMUL=8 to create a 256-bit element group for SM3. This would mean that there would only be 4 register groups, 3 of which would be consumed by a single SM3 message-expansion instruction.

As with all vector instructions, the number of elements processed is specified by the vector length vl. The number of element groups operated upon is then vl/EGS. Likewise the starting element group is vstart/EGS. See Section 1.5 for limitations on vl and vstart for vector crypto instructions.

1.5. Instruction Constraints

The following is a quick reference for the various constraints of specific Vector Crypto instructions.

vl and vstart constraints

Since vl and vstart refer to elements, Vector Crypto instructions that use elements groups (See Section 1.4) require that these values are an integer multiple of the Element Group Size (EGS).

- Instructions that violate the vl or vstart requirements are *reserved*.

Instructions	EGS
vaes*	4
vsha2*	4
vgh*	4
vsm3*	8
vsm4*	4

LMUL constraints

For element-group instructions, LMUL*VLEN must always be at least as large as EGW, otherwise an *illegal instruction exception* is raised, even if vl=0.

Instructions	SEW	EGW
vaes*	32	128
vsha2*	32	128

Instructions	SEW	EGW
vsha2*	64	256
vgh*	32	128
vsm3*	32	256
vsm4*	32	128

SEW constraints

Some Vector Crypto instructions are only defined for a specific SEW. In such a case all other SEW values are *reserved*.

Instructions	Required SEW
vaes*	32
Zvknha: vsha2*	32
Zvknhb: vsha2*	32 or 64
vclmul[h]	64
vgh*	32
vsm3*	32
vsm4*	32

Source/Destination overlap constraints

Some Vector Crypto instructions have overlap constraints. Encodings that violate these constraints are *reserved*.

In the case of the .vs instructions defined in this specification, vs2 holds a 128-bit scalar element group. For implementations with $VLEN \geq 128$, vs2 refers to a single register. Thus, the vd register group must not overlap the vs2 register. However, in implementations where $VLEN < 128$, vs2 refers to a register group comprised of the number of registers needed to hold the 128-bit scalar element group. In this case, the vd register group must not overlap this vs2 register group.

Instruction	Register	Cannot Overlap
vaes*.vs	vs2	vd
vsm4r.vs	vs2	vd
vsha2c[hl]	vs1, vs2	vd
vsha2ms	vs1, vs2	vd
sm3me	vs2	vd
vsm3c	vs2	vd

1.6. Vector-Scalar Instructions

The RISC-V Vector Extension defines three encodings for Vector-Scalar operations which get their

scalar operand from a GPR or FP register:

- OPVX: Scalar GPR x register
- OPFVF: Scalar FP f register
- OPMVX: Scalar GPR x register

However, the Vector Extensions include Vector Reduction Operations which can also be considered Vector-Scalar operations because a scalar operand is provided from element 0 of vector register **vs1**. The vector operand is provided in vector register group **vs2**. These reduction operations all use the **.vs** suffix in their mnemonics. Additionally, the reduction operations all produce a scalar result in element 0 of the destination register, **vd**.

The Vector Crypto Extensions define Vector-Scalar instructions that are similar to these Vector Reduction Operations in that they get a scalar operand from a vector register. However, they differ in that they get a scalar element group (see [Section 1.4](#)) from **vs2** and they return *vector* results to **vd**, which is also a source vector operand. These Vector-Scalar crypto instructions also use the **.vs** suffix in their mnemonics.



We chose to use **vs2** as the scalar operand, and **vd** as the vector operand, so that we could use the **vs1** specifier as additional encoding bits for these instructions. This allows these instructions to have a much smaller encoding footprint, leaving more rooms for other instructions in the future.

These instructions enable a single key, specified as a scalar element group in **vs2**, to be applied to each element group of register group **vd**.



Scalar element groups will occupy at most a single register in application processors. However, in implementations where $VLEN < 128$, they will occupy 2 ($VLEN=64$) or 4 ($VLEN=32$) registers.



It is common for multiple AES encryption rounds (for example) to be performed in parallel with the same round key (e.g. in counter modes). Rather than having to first splat the common key across the whole vector group, these vector-scalar crypto instructions allow the round key to be specified as a scalar element group.

1.7. Software Portability

The following contains some guidelines that enable the portability of vector-crypto-based code to implementations with different values for **VLEN**

Application Processors

Application processors are expected to follow the V-extension and will therefore have $VLEN \geq 128$.

Since most of the *cryptography-specific* instructions have an **EGW**=128, nothing special needs to be done for these instructions to support implementations with **VLEN**=128.

However, the SHA-512 and SM3 instructions have an **EGW**=256. Implementations with **VLEN** = 128,

require that **LMUL** is doubled for these instructions in order to create 256-bit elements across a pair of registers. Code written with this doubling of **LMUL** will not affect the results returned by implementations with **VLEN** \geq 256 because **v1** controls how many element groups are processed. Therefore, we recommend that libraries that implement SHA-512 and SM3 employ this doubling of **LMUL** to ensure that the software can run on all implementation with **VLEN** \geq 128.

While the doubling of **LMUL** for these instructions is *safe* for implementations with **VLEN** \geq 256, it may be less optimal as it will result in unnecessary register pressure and might exact a performance penalty in some microarchitectures. Therefore, we suggest that in addition to providing portable code for SHA-512 and SM3, libraries should also include more optimal code for these instructions when **VLEN** \geq 256.

Algorithm	Instructions	VLEN	LMUL
SHA-512	vsha2*	64	v1/2
SM3	vsm3*	32	v1/4

Embedded Processors

Embedded processors will typically have implementations with **VLEN** $<$ 128. This will require code to be written with larger **LMUL** values to enable the element groups to be formed.

The **.vs** instructions require scalar element groups of **EGW**=128. On implementations with **VLEN** $<$ 128, these scalar element groups will necessarily be formed across registers. This is different from most scalars in vector instructions that typically consume part of a single register.

We recommend that different code be available for **VLEN**=32 and **VLEN**=64, as code written for **VLEN**=32 will likely be too burdensome for **VLEN**=64 implementations.

Chapter 2. Extensions Overview

The section introduces all of the extensions in the Vector Cryptography Instruction Set Extension Specification.

The [Zvknhb](#) and [Zvbc](#) Vector Crypto Extensions --and accordingly the composite extensions [Zvkn](#) and [Zvks](#)-- require a Zve64x base, or application ("V") base Vector Extension.

All of the other Vector Crypto Extensions can be built on *any* embedded (Zve*) or application ("V") base Vector Extension.

All *cryptography-specific* instructions defined in this Vector Crypto specification (i.e., those in [Zvkned](#), [Zvknh\[ab\]](#), [Zvkg](#), [Zvksed](#) and [Zvksh](#) but *not* [Zvbb](#), [Zvkb](#), or [Zvbc](#)) shall be executed with data-independent execution latency as defined in the [RISC-V Scalar Cryptography Extensions specification](#). It is important to note that the Vector Crypto instructions are independent of the implementation of the [Zkt](#) extension and do not require that [Zkt](#) is implemented.

This specification includes a [Zvkt](#) extension that, when implemented, requires certain vector instructions (including [Zvbb](#), [Zvkb](#), and [Zvbc](#)) to be executed with data-independent execution latency.

Detection of individual cryptography extensions uses the unified software-based RISC-V discovery method.



At the time of writing, these discovery mechanisms are still a work in progress.

2.1. [Zvbb](#) - Vector Basic Bit-manipulation

Vector basic bit-manipulation instructions.



This extension is a superset of the [Zvkb](#) extension.

Mnemonic	Instruction
vandn.[vv,vx]	Vector And-Not
vbrev.v	Vector Reverse Bits in Elements
vbrev8.v	Vector Reverse Bits in Bytes
vrev8.v	Vector Reverse Bytes
vclz.v	Vector Count Leading Zeros
vctz.v	Vector Count Trailing Zeros
vcpop.v	Vector Population Count
vrol.[vv,vx]	Vector Rotate Left
vrord.[vv,vx,vi]	Vector Rotate Right
vwsll.[vv,vx,vi]	Vector Widening Shift Left Logical

2.2. Zvbc - Vector Carryless Multiplication

General purpose carryless multiplication instructions which are commonly used in cryptography and hashing (e.g., Elliptic curve cryptography, GHASH, CRC).

These instructions are only defined for SEW=64.

Mnemonic	Instruction
vclmul.[vv,vx]	Vector Carry-less Multiply
vclmulh.[vv,vx]	Vector Carry-less Multiply Return High Half

2.3. Zvkb - Vector Cryptography Bit-manipulation

Vector bit-manipulation instructions that are essential for implementing common cryptographic workloads securely & efficiently.



This Zvkb extension is a proper subset of the Zvbb extension. Zvkb allows for vector crypto implementations without incurring the the cost of implementing the additional bitmanip instructions in the Zvbb extension: `vbrev.v`, `vclz.v`, `vctz.v`, `vcpop.v`, and `vwsll.[vv,vx,vi]`.

Mnemonic	Instruction
<code>vandn.[vv,vx]</code>	Vector And-Not
<code>vbrev8.v</code>	Vector Reverse Bits in Bytes
<code>vrev8.v</code>	Vector Reverse Bytes
<code>vrol.[vv,vx]</code>	Vector Rotate Left
<code>vror.[vv,vx,vi]</code>	Vector Rotate Right

2.4. **Zvkg** - Vector GCM/GMAC

Instructions to enable the efficient implementation of GHASH_H which is used in Galois/Counter Mode (GCM) and Galois Message Authentication Code (GMAC).

All of these instructions work on 128-bit element groups comprised of four 32-bit elements.

GHASH_H is defined in the "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC" [3] (NIST Specification).



GCM is used in conjunction with block ciphers (e.g., AES and SM4) to encrypt a message and provide authentication. GMAC is used to provide authentication of a message without encryption.

To help avoid side-channel timing attacks, these instructions shall be implemented with data-independent timing.

The number of element groups to be processed is $v1/EGS$. $v1$ must be set to the number of $SEW=32$ elements to be processed and therefore must be a multiple of $EGS=4$.

Likewise, $vstart$ must be a multiple of $EGS=4$.

SEW	EGW	Mnemonic	Instruction
32	128	vghsh.vv	Vector GHASH Add-Multiply
32	128	vgmul.vv	Vector GHASH Multiply

2.5. Zvkned - NIST Suite: Vector AES Block Cipher

Instructions for accelerating encryption, decryption and key-schedule functions of the AES block cipher as defined in Federal Information Processing Standards Publication 197 [5]

All of these instructions work on 128-bit element groups comprised of four 32-bit elements.

For the best performance, it is suggested that these instruction be implemented on systems with $VLEN \geq 128$. On systems with $VLEN < 128$, element groups may be formed by concatenating 32-bit elements from two or four registers by using an $LMUL = 2$ and $LMUL = 4$ respectively.

To help avoid side-channel timing attacks, these instructions shall be implemented with data-independent timing.

The number of element groups to be processed is $v1/EGS$. $v1$ must be set to the number of $SEW=32$ elements to be processed and therefore must be a multiple of $EGS=4$.

Likewise, $vstart$ must be a multiple of $EGS=4$.

SEW	EGW	Mnemonic	Instruction
32	128	vaesef.[vv,vs]	Vector AES encrypt final round
32	128	vaesem.[vv,vs]	Vector AES encrypt middle round
32	128	vaesdf.[vv,vs]	Vector AES decrypt final round
32	128	vaesdm.[vv,vs]	Vector AES decrypt middle round
32	128	vaeskf1.vi	Vector AES-128 Forward KeySchedule
32	128	vaeskf2.vi	Vector AES-256 Forward KeySchedule
32	128	vaesz.vs	Vector AES round zero

2.6. **Zvknh[ab]** - NIST Suite: Vector SHA-2 Secure Hash

Instructions for accelerating SHA-2 as defined in FIPS PUB 180-4 Secure Hash Standard (SHS) [6]

SEW differentiates between SHA-256 (**SEW**=32) and SHA-512 (**SEW**=64).

- SHA-256: these instructions work on 128-bit element groups comprised of four 32-bit elements.
- SHA-512: these instructions work on 256-bit element groups comprised of four 64-bit elements.

SEW	EGW	SHA-2	Extension
32	128	SHA-256	Zvknha, Zvknhb
64	256	SHA-512	Zvknhb

- Zvknhb supports SHA-256 and SHA-512.
- Zvknha supports only SHA-256.

SHA-256 implementations with $VLEN < 128$ require $LMUL > 1$ to combine 32-bit elements from register groups to provide all four elements of the element group.

SHA-512 implementations with $VLEN < 256$ require $LMUL > 1$ to combine 64-bit elements from register groups to provide all four elements of the element group.

To help avoid side-channel timing attacks, these instructions shall be implemented with data-independent timing.

The number of element groups to be processed is $v1/EGS$. $v1$ must be set to the number of **SEW** elements to be processed and therefore must be a multiple of $EGS=4$.

Likewise, **vstart** must be a multiple of $EGS=4$.

Mnemonic	Instruction
vsha2ms.vv	Vector SHA-2 Message Schedule
vsha2c[hl].vv	Vector SHA-2 Compression

2.7. **Zvksed** - ShangMi Suite: SM4 Block Cipher

Instructions for accelerating encryption, decryption and key-schedule functions of the SM4 block cipher.

The SM4 block cipher is specified in 32907-2016: *{SM4} Block Cipher Algorithm* [2]

There are other various sources available that describe the SM4 block cipher. While not the final version of the standard, [RFC 8998 ShangMi \(SM\) Cipher Suites for TLS 1.3](#) is useful and easy to access.

All of these instructions work on 128-bit element groups comprised of four 32-bit elements.

To help avoid side-channel timing attacks, these instructions shall be implemented with data-independent timing.

The number of element groups to be processed is $v1/EGS$. $v1$ must be set to the number of $SEW=32$ elements to be processed and therefore must be a multiple of $EGS=4$.

Likewise, $vstart$ must be a multiple of $EGS=4$.

SEW	EGW	Mnemonic	Instruction
32	128	vsm4k.vi	Vector SM4 Key Expansion
32	128	vsm4r.[vv,vs]	SM4 Block Cipher Rounds

2.8. Zvksh - ShangMi Suite: SM3 Secure Hash

Instructions for accelerating functions of the SM3 Hash Function.

The SM3 secure hash algorithm is specified in *32905-2016: SM3 Cryptographic Hash Algorithm* [2]

There are other various sources available that describe the SM3 secure hash. While not the final version of the standard, [RFC 8998 ShangMi \(SM\) Cipher Suites for TLS 1.3](#) is useful and easy to access.

All of these instructions work on 256-bit element groups comprised of eight 32-bit elements.

Implementations with $VLEN < 256$ require $LMUL > 1$ to combine 32-bit elements from register groups to provide all eight elements of the element group.

To help avoid side-channel timing attacks, these instructions shall be implemented with data-independent timing.

The number of element groups to be processed is $v1/EGS$. $v1$ must be set to the number of $SEW=32$ elements to be processed and therefore must be a multiple of $EGS=8$.

Likewise, $vstart$ must be a multiple of $EGS=8$.

SEW	EGW	Mnemonic	Instruction
32	256	vsm3me.vv	SM3 Message Expansion
32	256	vsm3c.vi	SM3 Compression

2.9. Zvkn - NIST Algorithm Suite

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Zvkned	Zvkned
Zvknhb	Zvknhb
Zvkb	Zvkb
Zvkt	Zvkt



While Zvkg and Zvbc are not part of this extension, it is recommended that at least one of them is implemented with this extension to enable efficient AES-GCM.

2.10. **Zvkn** - NIST Algorithm Suite with carryless multiply

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Zvkn	Zvkn
Zvbc	Zvbc



This extension combines the NIST Algorithm Suite with the vector carryless multiply extension to enable AES-GCM.

2.11. **Zvkn**g - NIST Algorithm Suite with GCM

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Zvkn	Zvkn
Zvkg	Zvkg



This extension combines the NIST Algorithm Suite with the GCM/GMAC extension to enable high-performance AES-GCM.

2.12. **Zvks** - ShangMi Algorithm Suite

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Zvksed	Zvksed
Zvksh	Zvksh
Zvkb	Zvkb
Zvkt	Zvkt



While Zvkg and Zvbc are not part of this extension, it is recommended that at least one of them is implemented with this extension to enable efficient SM4-GCM.

2.13. **Zvks** - ShangMi Algorithm Suite with carryless multiplication

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Zvks	Zvks
Zvbc	Zvbc



This extension combines the ShangMi Algorithm Suite with the vector carryless multiply extension to enable SM4-GCM.

2.14. **Zvks**g - ShangMi Algorithm Suite with GCM

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Zvks	Zvks
Zvkg	Zvkg



This extension combines the ShangMi Algorithm Suite with the GCM/GMAC extension to enable high-performance SM4-GCM.

2.15. Zvkt - Vector Data-Independent Execution Latency

The Zvkt extension requires all implemented instructions from the following list to be executed with data-independent execution latency as defined in the [RISC-V Scalar Cryptography Extensions specification](#).

Data-independent execution latency (DIEL) applies to all *data operands* of an instruction, even those that are not a part of the body or that are inactive. However, DIEL does not apply to other values such as vl, vtype, and the mask (when used to control execution of a masked vector instruction). Also, DIEL does not apply to constant values specified in the instruction encoding such as the use of the zero register (x0), and, in the case of immediate forms of an instruction, the values in the immediate fields (i.e., imm, and uimm).

In some cases --- which are explicitly specified in the lists below --- operands that are used as control rather than data are exempt from DIEL.



DIEL helps protect against side-channel timing attacks that are used to determine data values that are intended to be kept secret. Such values include cryptographic keys, plain text, and partially encrypted text. DIEL is not intended to keep software (and cryptographic algorithms contained therein) secret as it is assumed that an adversary would already know these. This is why DIEL doesn't apply to constants embedded in instruction encodings.

It is important that the *values* of elements that are not in the body or that are masked off do not affect the execution latency of the instruction. Sometimes such elements contain data that also needs to be kept secret.

2.15.1. All Zvbb instructions:

- vandn.v[vx]
- vclz.v
- vcpop.v
- vctz.v
- vbrev.v
- vbrev8.v
- vrev8.v
- vrol.v[vx]
- vror.v[vxi]
- vwsll.[vv,vx,vi]



All Zvkb instructions are also covered by DIEL as they are a proper subset of Zvbb

2.15.2. All **Zvbc** instructions

- vclmul[h].v[vx]

2.15.3. **add/sub**

- v[r]sub.v[vx]
- vadd.v[ivx]
- vsub.v[vx]
- vwadd[u].[vw][vx]
- vwsb[u].[vw][vx]

2.15.4. **add/sub with carry**

- vadc.v[ivx]m
- vmadc.v[ivx][m]
- vmsbc.v[vx]m
- vsbc.v[vx]m

2.15.5. **compare and set**

- vmseq.v[vxi]
- vmsgt[u].v[xi]
- vmsle[u].v[xi]
- vmslt[u].v[xi]
- vmsne.v[ivx]

2.15.6. **copy**

- vmv.s.x
- vmv.v.[ivxs]
- vmv[1248]r.v

2.15.7. **extend**

- vsex.vf[248]
- vzext.vf[248]

2.15.8. **logical**

- vand.v[ivx]
- vm[n]or.mm

- vmand[n].mm
- vmnand.mm
- vmorn.mm
- vmx[n]or.mm
- vor.v[ivx]
- vxor.v[ivx]

2.15.9. multiply

- vmul[h].v[vx]
- vmulh[s]u.v[vx]
- vwmul.v[vx]
- vwmul[s]u.v[vx]

2.15.10. multiply-add

- vmacc.v[vx]
- vmadd.v[vx]
- vnmsac.v[vx]
- vnmsub.v[vx]
- vwmacc.v[vx]
- vwmacc[s]u.v[vx]
- vwmaccus.vx

2.15.11. Integer Merge

- vmerge.v[ivx]m

2.15.12. permute

In the **.vv** and **.xv** forms of the **vragather[ei16]** instructions, the values in **vs1** and **rs1** are used for control and therefore are exempt from DIEL.

- vrgather.v[ivx]
- vrgatherei16.vv

2.15.13. shift

- vnsr[al].w[ivx]
- vsll.v[ivx]
- vsr[al].v[ivx]

2.15.14. slide

- vslide1[up|down].vx
- vfslide1[up|down].vf

In the vslide[up|down].vx instructions, the value in `rs1` is used for control (i.e., slide amount) and therefore is exempt from DIEL.

- vslide[up|down].v[ix]

The following instructions are not affected by Zvkt:

- **All storage operations**
- **All floating-point operations**
- add/sub saturate
 - vsadd[u].v[ivx]
 - vssub[u].v[vx]
- clip
 - vnclip[u].w[ivx]
- compress
 - vcompress.vm
- divide
 - vdiv[u].v[vx]
 - vrem[u].v[vx]
- average
 - vaadd[u].v[vx]
 - vasub[u].v[vx]
- mask Op
 - vcpop.m
 - vfirst.m
 - vid.v
 - viota.m
 - vms[bio]f.m
- min/max
 - vmax[u].v[vx]
 - vmin[u].v[vx]
- Multiply-saturate
 - vsmul.v[vx]



- reduce
 - vredsum.vs
 - vwredsum[u].vs
 - vred[and | or | xor].vs
 - vred[min | max][u].vs
- shift round
 - vssra.v[ivx]
 - vssrl.v[ivx]
- vset
 - vsetivli
 - vsetvl[i]

Chapter 3. Instructions

3.1. vaesdf.[vv,vs]

Synopsis

Vector AES final-round decryption

Mnemonic

vaesdf.vv vd, vs2

vaesdf.vs vd, vs2

Encoding (Vector-Vector)

31	26	25	24	20	19	15	14	12	11	7	6	0
101000	1	vs2	00001	OPMVV	vd	OP-P						

Encoding (Vector-Scalar)

31	26	25	24	20	19	15	14	12	11	7	6	0
101001	1	vs2	00001	OPMVV	vd	OP-P						

Reserved Encodings

- **SEW** is any value other than 32
- Only for the **.vs** form: the **vd** register group overlaps the **vs2** scalar element group

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vd	input	128	4	32	round state
Vs2	input	128	4	32	round key
Vd	output	128	4	32	new round state

Description

A final-round AES block cipher decryption is performed.

The InvShiftRows and InvSubBytes steps are applied to each round state element group from **vd**. This is then XORed with the round key in either the corresponding element group in **vs2** (vector-vector form) or scalar element group in **vs2** (vector-scalar form).

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.

Operation

```
function clause execute (VAESDF(vs2, vd, suffix)) = {  
    if(LMUL*VLEN < EGW) then {
```

```

    handle_illegal(); // illegal instruction exception
    RETIRE_FAIL
} else {

    eg_len = (vl/EGS)
    eg_start = (vstart/EGS)

    foreach (i from eg_start to eg_len-1) {
        let keyelem = if suffix == "vv" then i else 0;
        let state : bits(128) = get_velem(vd, EGW=128, i);
        let rkey   : bits(128) = get_velem(vs2, EGW=128, keyelem);
        let sr     : bits(128) = aes_shift_rows_inv(state);
        let sb     : bits(128) = aes_subbytes_inv(sr);
        let ark    : bits(128) = sb ^ rkey;
        set_velem(vd, EGW=128, i, ark);
    }
    RETIRE_SUCCESS
}
}

```

Included in

[Zvkn](#), [Zvknc](#), [Zvkned](#), [Zvkng](#)

3.2. vaesdm.[vv,vs]

Synopsis

Vector AES middle-round decryption

Mnemonic

vaesdm.vv vd, vs2

vaesdm.vs vd, vs2

Encoding (Vector-Vector)

31	26	25	24	20	19	15	14	12	11	7	6	0
101000	1	vs2	00000	OPMVV	vd	OP-P						

Encoding (Vector-Scalar)

31	26	25	24	20	19	15	14	12	11	7	6	0
101001	1	vs2	00000	OPMVV	vd	OP-P						

Reserved Encodings

- **SEW** is any value other than 32
- Only for the **.vs** form: the **vd** register group overlaps the **vs2** scalar element group

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vd	input	128	4	32	round state
Vs2	input	128	4	32	round key
Vd	output	128	4	32	new round state

Description

A middle-round AES block cipher decryption is performed.

The InvShiftRows and InvSubBytes steps are applied to each round state element group from **vd**. This is then XORed with the round key in either the corresponding element group in **vs2** (vector-vector form) or the scalar element group in **vs2** (vector-scalar form). The result is then applied to the InvMixColumns step.

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.

Operation

```
function clause execute (VAESDM(vs2, vd, suffix)) = {
  if(LMUL*VLEN < EGW) then {
    handle_illegal(); // illegal instruction exception
    RETIRE_FAIL
  }
}
```

```

} else {

eg_len = (vl/EGS)
eg_start = (vstart/EGS)

foreach (i from eg_start to eg_len-1) {
  let keyelem = if suffix == "vv" then i else 0;
  let state : bits(128) = get_velem(vd, EGW=128, i);
  let rkey  : bits(128) = get_velem(vs2, EGW=128, keyelem);
  let sr    : bits(128) = aes_shift_rows_inv(state);
  let sb    : bits(128) = aes_subbytes_inv(sr);
  let ark   : bits(128) = sb ^ rkey;
  let mix   : bits(128) = aes_mixcolumns_inv(ark);
  set_velem(vd, EGW=128, i, mix);
}
RETIRE_SUCCESS
}
}

```

Included in

[Zvkn](#), [Zvknc](#), [Zvkned](#), [Zvkng](#)

3.3. vaesef.[vv,vs]

Synopsis

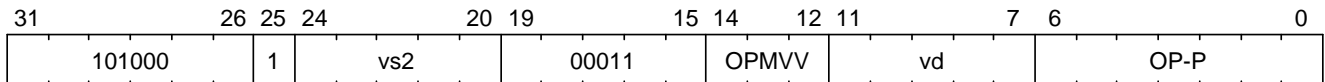
Vector AES final-round encryption

Mnemonic

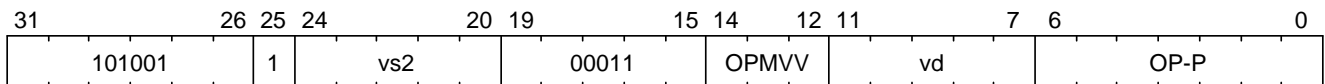
vaesef.vv vd, vs2

vaesef.vs vd, vs2

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Reserved Encodings

- **SEW** is any value other than 32
- Only for the **.vs** form: the **vd** register group overlaps the **vs2** scalar element group

Arguments

Register	Direction	EGW	EGS	EEW	Definition
vd	input	128	4	32	round state
vs2	input	128	4	32	round key
vd	output	128	4	32	new round state

Description

A final-round encryption function of the AES block cipher is performed.

The SubBytes and ShiftRows steps are applied to each round state element group from **vd**. This is then XORed with the round key in either the corresponding element group in **vs2** (vector-vector form) or the scalar element group in **vs2** (vector-scalar form).

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.

Operation

```
function clause execute (VAESEF(vs2, vd, suffix) = {
  if(LMUL*VLEN < EGW) then {
    handle_illegal(); // illegal instruction exception
    RETIRE_FAIL
  } else {
```

```

eg_len = (vl/EGS)
eg_start = (vstart/EGS)

foreach (i from eg_start to eg_len-1) {
  let keyelem = if suffix == "vv" then i else 0;
  let state : bits(128) = get_velem(vd, EGW=128, i);
  let rkey  : bits(128) = get_velem(vs2, EGW=128, keyelem);
  let sb    : bits(128) = aes_subbytes_fwd(state);
  let sr    : bits(128) = aes_shift_rows_fwd(sb);
  let ark   : bits(128) = sr ^ rkey;
  set_velem(vd, EGW=128, i, ark);
}
RETIRE_SUCCESS
}
}

```

Included in

[Zvkn](#), [Zvknc](#), [Zvkned](#), [Zvkng](#)

3.4. vaesem.[vv,vs]

Synopsis

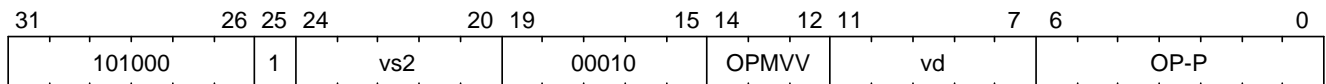
Vector AES middle-round encryption

Mnemonic

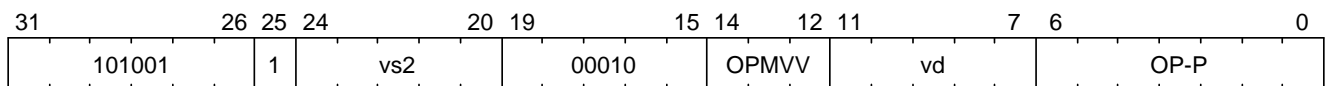
vaesem.vv vd, vs2

vaesem.vs vd, vs2

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Reserved Encodings

- **SEW** is any value other than 32
- Only for the **.vs** form: the **vd** register group overlaps the **vs2** scalar element group

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vd	input	128	4	32	round state
Vs2	input	128	4	32	Round key
Vd	output	128	4	32	new round state

Description

A middle-round encryption function of the AES block cipher is performed.

The SubBytes, ShiftRows, and MixColumns steps are applied to each round state element group from **vd**. This is then XORed with the round key in either the corresponding element group in **vs2** (vector-vector form) or the scalar element group in **vs2** (vector-scalar form).

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.

Operation

```
function clause execute (VAESEM(vs2, vd, suffix)) = {
  if(LMUL*VLEN < EGW) then {
    handle_illegal(); // illegal instruction exception
    RETIRE_FAIL
  } else {
```

```

eg_len = (vl/EGS)
eg_start = (vstart/EGS)

foreach (i from eg_start to eg_len-1) {
  let keyelem = if suffix == "vv" then i else 0;
  let state : bits(128) = get_velem(vd, EGW=128, i);
  let rkey  : bits(128) = get_velem(vs2, EGW=128, keyelem);
  let sb    : bits(128) = aes_subbytes_fwd(state);
  let sr    : bits(128) = aes_shift_rows_fwd(sb);
  let mix   : bits(128) = aes_mixcolumns_fwd(sr);
  let ark   : bits(128) = mix ^ rkey;
  set_velem(vd, EGW=128, i, ark);
}
RETIRE_SUCCESS
}
}

```

Included in

[Zvkn](#), [Zvknc](#), [Zvkned](#), [Zvkng](#)

3.5. vaeskf1.vi

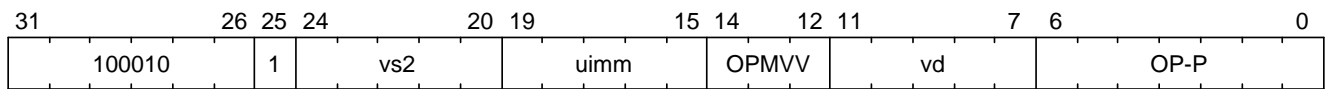
Synopsis

Vector AES-128 Forward KeySchedule generation

Mnemonic

vaeskf1.vi vd, vs2, uimm

Encoding



Reserved Encodings

- **SEW** is any value other than 32

Arguments

Register	Direction	EGW	EGS	EEW	Definition
uimm	input	-	-	-	Round Number (rnd)
Vs2	input	128	4	32	Current round key
Vd	output	128	4	32	Next round key

Description

A single round of the forward AES-128 KeySchedule is performed.

The next round key is generated word by word from the current round key element group in **vs2** and the immediately previous word of the round key. The least significant word is generated using the most significant word of the current round key as well as a round constant which is selected by the round number.

The round number, which ranges from 1 to 10, comes from **uimm[3:0]**; **uimm[4]** is ignored. The out-of-range **uimm[3:0]** values of 0 and 11-15 are mapped to in-range values by inverting **uimm[3]**. Thus, 0 maps to 8, and 11-15 maps to 3-7. The round number is used to specify a round constant which is used in generating the first round key word.

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.



We chose to map out-of-range round numbers to in-range values as this allows the instruction's behavior to be fully defined for all values of **uimm[4:0]** with minimal extra logic.

Operation

```
function clause execute (VAESKF1(rnd, vd, vs2)) = {  
    if(LMUL*VLEN < EGW) then {
```

```

    handle_illegal(); // illegal instruction exception
    RETIRE_FAIL
} else {

// project out-of-range immediates onto in-range values
if( (unsigned(rnd[3:0]) > 10) | (rnd[3:0] = 0)) then rnd[3] = ~rnd[3]

eg_len = (vl/EGS)
eg_start = (vstart/EGS)

let r : bits(4) = rnd-1;

foreach (i from eg_start to eg_len-1) {
    let CurrentRoundKey[3:0] : bits(128) = get_velem(vs2, EGW=128, i);
    let w[0] : bits(32) = aes_subword_fwd(aes_rotword(CurrentRoundKey[3])) XOR
        aes_decode_rcon(r) XOR CurrentRoundKey[0]
    let w[1] : bits(32) = w[0] XOR CurrentRoundKey[1]
    let w[2] : bits(32) = w[1] XOR CurrentRoundKey[2]
    let w[3] : bits(32) = w[2] XOR CurrentRoundKey[3]
    set_velem(vd, EGW=128, i, w[3:0]);
}
    RETIRE_SUCCESS
}
}

```

Included in

[Zvkn](#), [Zvknc](#), [Zvkned](#), [Zvkng](#)

3.6. vaeskf2.vi

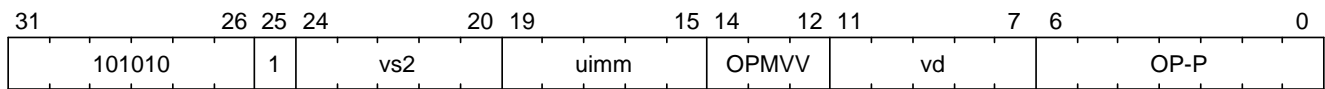
Synopsis

Vector AES-256 Forward KeySchedule generation

Mnemonic

vaeskf2.vi vd, vs2, uimm

Encoding



Reserved Encodings

- SEW is any value other than 32

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vd	input	128	4	32	Previous Round key
uimm	input	-	-	-	Round Number (rnd)
Vs2	input	128	4	32	Current Round key
Vd	output	128	4	32	Next round key

Description

A single round of the forward AES-256 KeySchedule is performed.

The next round key is generated word by word from the previous round key element group in **vd** and the immediately previous word of the round key. The least significant word of the next round key is generated by applying a function to the most significant word of the current round key and then XORing the result with the round constant. The round number is used to select the round constant as well as the function.

The round number, which ranges from 2 to 14, comes from **uimm[3:0]**; **uimm[4]** is ignored. The out-of-range **uimm[3:0]** values of 0-1 and 15 are mapped to in-range values by inverting **uimm[3]**. Thus, 0-1 maps to 8-9, and 15 maps to 7.

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.



We chose to map out-of-range round numbers to in-range values as this allows the instruction's behavior to be fully defined for all values of **uimm[4:0]** with minimal extra logic.

Operation

```
function clause execute (VAESKF2(rnd, vd, vs2)) = {
```

```

if(LMUL*VLEN < EGW) then {
    handle_illegal(); // illegal instruction exception
    RETIRE_FAIL
} else {

// project out-of-range immediates into in-range values
if((unsigned(rnd[3:0]) < 2) | (unsigned(rnd[3:0]) > 14)) then rnd[3] = ~rnd[3]

eg_len = (vl/EGS)
eg_start = (vstart/EGS)

foreach (i from eg_start to eg_len-1) {
    let CurrentRoundKey[3:0] : bits(128) = get_velem(vs2, EGW=128, i);
    let RoundKeyB[3:0] : bits(32) = get_velem(vd, EGW=128, i); // Previous round
key

    let w[0] : bits(32) = if (rnd[0]==1) then
        aes_subword_fwd(CurrentRoundKey[3]) XOR RoundKeyB[0];
    else
        aes_subword_fwd(aes_rotword(CurrentRoundKey[3])) XOR aes_decode_rcon((rnd>>1)
- 1) XOR RoundKeyB[0];
    w[1] : bits(32) = w[0] XOR RoundKeyB[1]
    w[2] : bits(32) = w[1] XOR RoundKeyB[2]
    w[3] : bits(32) = w[2] XOR RoundKeyB[3]
    set_velem(vd, EGW=128, i, w[3:0]);
}
    RETIRE_SUCCESS
}
}

```

Included in

[Zvkn](#), [Zvknc](#), [Zvkned](#), [Zvkng](#)

3.7. vaesz.vs

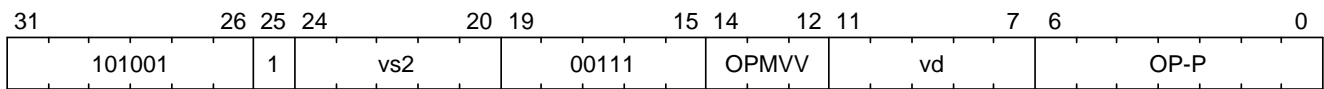
Synopsis

Vector AES round zero encryption/decryption

Mnemonic

vaesz.vs vd, vs2

Encoding (Vector-Scalar)



Reserved Encodings

- **SEW** is any value other than 32
- The **vd** register group overlaps the **vs2** register

Arguments

Register	Direction	EGW	EGS	EEW	Definition
vd	input	128	4	32	round state
vs2	input	128	4	32	round key
vd	output	128	4	32	new round state

Description

A round-0 AES block cipher operation is performed. This operation is used for both encryption and decryption.

There is only a **.vs** form of the instruction. **vs2** holds a scalar element group that is used as the round key for all of the round state element groups. The new round state output of each element group is produced by XORing the round key with each element group of **vd**.

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.



This instruction is needed to avoid the need to "splat" a 128-bit vector register group when the round key is the same for all 128-bit "lanes". Such a splat would typically be implemented with a **vrgather** instruction which would hurt performance in many implementations. This instruction only exists in the **.vs** form because the **.vv** form would be identical to the **vxor.vv vd, vs2, vd** instruction.

Operation

```
function clause execute (VAESZ(vs2, vd) = {  
  if(((vstart%EGS)<>0) | (LMUL*VLEN < EGW)) then {  
    handle_illegal(); // illegal instruction exception  
    RETIRE_FAIL  
  }
```

```

} else {

eg_len = (vl/EGS)
eg_start = (vstart/EGS)

foreach (i from eg_start to eg_len-1) {
  let state : bits(128) = get_velem(vd, EGW=128, i);
  let rkey  : bits(128) = get_velem(vs2, EGW=128, 0);
  let ark   : bits(128) = state ^ rkey;
  set_velem(vd, EGW=128, i, ark);
}
RETIRE_SUCCESS
}
}

```

Included in

[Zvkn](#), [Zvknc](#), [Zvkned](#), [Zvkng](#)

3.8. vandn.[vv,vx]

Synopsis

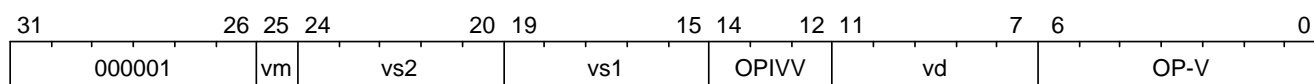
Bitwise And-Not

Mnemonic

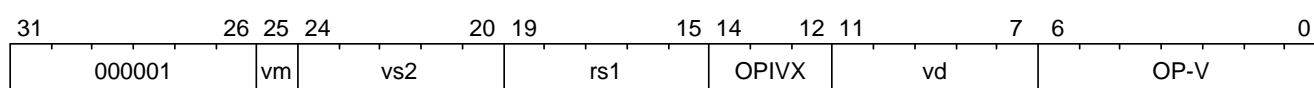
vandn.vv vd, vs2, vs1, vm

vandn.vx vd, vs2, rs1, vm

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Vector-Vector Arguments

Register	Direction	Definition
Vs1	input	Op1 (to be inverted)
Vs2	input	Op2
Vd	output	Result

Vector-Scalar Arguments

Register	Direction	Definition
Rs1	input	Op1 (to be inverted)
Vs2	input	Op2
Vd	output	Result

Description

A bitwise *and-not* operation is performed.

Each bit of **Op1** is inverted and logically ANDed with the corresponding bits in **vs2**. In the vector-scalar version, **Op1** is the sign-extended or truncated value in scalar register **rs1**. In the vector-vector version, **Op1** is **vs1**.



Note on necessity of instruction

This instruction is performance-critical to SHA3. Specifically, the Chi step of the FIPS 202 Keccak Permutation. Emulating it via 2 instructions is expected to have significant performance impact. The **.vv** form of the instruction is what is needed for SHA3; the **.vx** form was added for completeness.



There is no .vi version of this instruction because the same functionality can be achieved by using an inversion of the immediate value with the `vand.vi` instruction.

Operation

```
function clause execute (VANDN(vs2, vs1, vd, suffix)) = {
  foreach (i from vstart to vl-1) {
    let op1 = match suffix {
      "vv" => get_velem(vs1, SEW, i),
      "vx" => sext_or_truncate_to_sew(X(vs1))
    };
    let op2 = get_velem(vs2, SEW, i);
    set_velem(vd, EEW=SEW, i, ~op1 & op2);
  }
  RETIRE_SUCCESS
}
```

Included in

[Zvbb](#), [Zvkb](#), [Zvkn](#), [Zvknc](#), [Zvkng](#), [Zvks](#) [Zvksc](#), [Zvksg](#)

3.9. vbrev.v

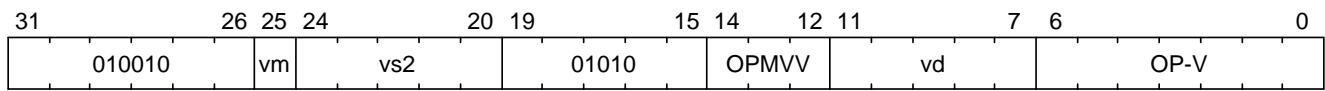
Synopsis

Vector Reverse Bits in Elements

Mnemonic

vbrev.v vd, vs2, vm

Encoding (Vector)



Arguments

Register	Direction	Definition
Vs2	input	Input elements
Vd	output	Elements with bits reversed

Description

A bit reversal is performed on the bits of each element.

Operation

```
function clause execute (VBREV(vs2)) = {  
  
  foreach (i from vstart to vl-1) {  
    let input = get_velem(vs2, SEW, i);  
    let output : bits(SEW) = 0;  
    foreach (i from 0 to SEW-1)  
      let output[SEW-1-i] = input[i];  
    set_velem(vd, SEW, i, output)  
  }  
  RETIRE_SUCCESS  
}
```

Included in

[Zvbb](#)

3.10. vbrev8.v

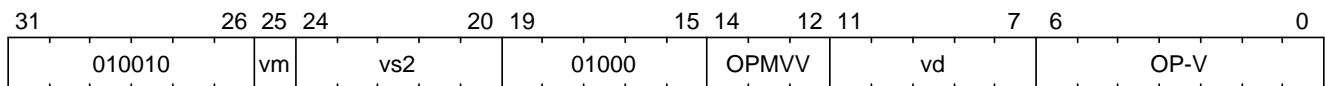
Synopsis

Vector Reverse Bits in Bytes

Mnemonic

vbrev8.v vd, vs2, vm

Encoding (Vector)



Arguments

Register	Direction	Definition
Vs2	input	Input elements
Vd	output	Elements with bit-reversed bytes

Description

A bit reversal is performed on the bits of each byte.



This instruction is commonly used for GCM when the zvk_g extension is not implemented. This byte-wise instruction is defined for all SEWs to eliminate the need to change SEW when operating on wider elements.

Operation

```
function clause execute (VBREV8(vs2)) = {  
    foreach (i from vstart to vl-1) {  
        let input = get_velem(vs2, SEW, i);  
        let output : bits(SEW) = 0;  
        foreach (i from 0 to SEW-8 by 8)  
            let output[i+7..i] = reverse_bits_in_byte(input[i+7..i]);  
        set_velem(vd, SEW, i, output)  
    }  
    RETIRE_SUCCESS  
}
```

Included in

[Zvbb](#), [Zvkb](#), [Zvkn](#), [ZvknC](#), [Zvkng](#), [Zvks](#) [ZvksC](#), [Zvksg](#)

3.11. vclmul.[vv,vx]

Synopsis

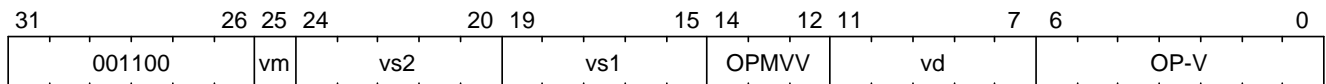
Vector Carry-less Multiply by vector or scalar - returning low half of product.

Mnemonic

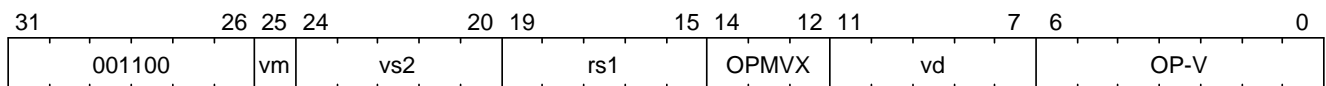
vclmul.vv vd, vs2, vs1, vm

vclmul.vx vd, vs2, rs1, vm

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Reserved Encodings

- **SEW** is any value other than 64

Arguments

Register	Direction	Definition
Vs1/Rs1	input	multiplier
Vs2	input	multiplicand
Vd	output	carry-less product low

Description

Produces the low half of 128-bit carry-less product.

Each 64-bit element in the **vs2** vector register is carry-less multiplied by either each 64-bit element in **vs1** (vector-vector), or the 64-bit value from integer register **rs1** (vector-scalar). The result is the least significant 64 bits of the carry-less product.



The 64-bit carryless multiply instructions can be used for implementing GCM in the absence of the **zvkg** extension. We do not make these instructions exclusive as the 64-bit carryless multiply is readily derived from the instructions in the **zvkg** extension and can have utility in other areas. Likewise, we treat other SEW values as reserved so as not to preclude future extensions from using this opcode with different element widths. For example, a future extension might define an **SEW=32** version of this instruction to enable **Zve32*** implementations to have vector carryless multiplication instructions.

Operation

```

function clause execute (VCLMUL(vs2, vs1, vd, suffix)) = {

  foreach (i from vstart to vl-1) {
    let op1 : bits (64) = if suffix == "vv" then get_velem(vs1,i)
                          else zext_or_truncate_to_sew(X(vs1));
    let op2 : bits (64) = get_velem(vs2,i);
    let product : bits (64) = clmul(op1,op2,SEW);
    set_velem(vd, i, product);
  }
  RETIRE_SUCCESS
}

function clmul(x, y, width) = {
  let result : bits(width) = zeros();
  foreach (i from 0 to (width - 1)) {
    if y[i] == 1 then result = result ^ (x << i);
  }
  result
}

```

Included in

[Zvbc](#), [Zvknc](#), [Zvksc](#)

3.12. vclmulh.[vv,vx]

Synopsis

Vector Carry-less Multiply by vector or scalar - returning high half of product.

Mnemonic

vclmulh.vv vd, vs2, vs1, vm

vclmulh.vx vd, vs2, rs1, vm

Encoding (Vector-Vector)

31	26	25	24	20	19	15	14	12	11	7	6	0
001101	vm	vs2	vs1	OPMVV	vd	OP-V						

Encoding (Vector-Scalar)

31	26	25	24	20	19	15	14	12	11	7	6	0
001101	vm	vs2	rs1	OPMVX	vd	OP-V						

Reserved Encodings

- **SEW** is any value other than 64

Arguments

Register	Direction	Definition
Vs1	input	multiplier
Vs2	input	multiplicand
Vd	output	carry-less product high

Description

Produces the high half of 128-bit carry-less product.

Each 64-bit element in the **vs2** vector register is carry-less multiplied by either each 64-bit element in **vs1** (vector-vector), or the 64-bit value from integer register **rs1** (vector-scalar). The result is the most significant 64 bits of the carry-less product.

Operation

```
function clause execute (VCLMULH(vs2, vs1, vd, suffix)) = {  
    foreach (i from vstart to vl-1) {  
        let op1 : bits (64) = if suffix == "vv" then get_velem(vs1,i)  
                               else zext_or_truncate_to_sew(X(vs1));  
        let op2 : bits (64) = get_velem(vs2, i);  
        let product : bits (64) = clmulh(op1, op2, SEW);  
        set_velem(vd, i, product);  
    }  
    RETIRE_SUCCESS
```

```
}  
  
function clmulh(x, y, width) = {  
  let result : bits(width) = 0;  
  foreach (i from 1 to (width - 1)) {  
    if y[i] == 1 then result = result ^ (x >> (width - i));  
  }  
  result  
}
```

Included in

[Zvbc](#), [Zvknc](#), [Zvksc](#)

3.13. vclz.v

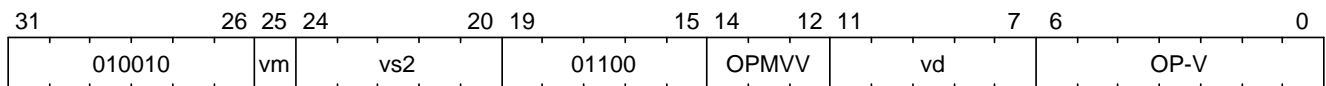
Synopsis

Vector Count Leading Zeros

Mnemonic

vclz.v vd, vs2, vm

Encoding (Vector)



Arguments

Register	Direction	Definition
Vs2	input	Input elements
Vd	output	Count of leading zero bits

Description

A leading zero count is performed on each element.

The result for zero-valued inputs is the value SEW.

Operation

```
function clause execute (VCLZ(vs2)) = {  
    foreach (i from vstart to vl-1) {  
        let input = get_velem(vs2, SEW, i);  
        for (j = (SEW - 1); j >= 0; j--)  
            if [input[j]] == 0b1 then break;  
        set_velem(vd, SEW, i, SEW - 1 - j)  
    }  
    RETIRE_SUCCESS  
}
```

Included in

[Zvbb](#)

3.14. vcpop.v

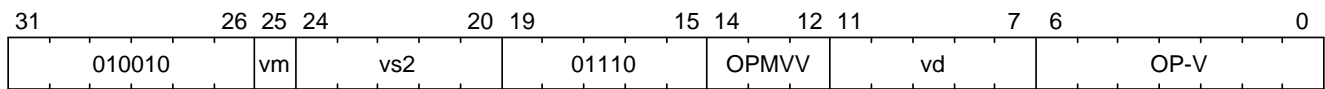
Synopsis

Count the number of bits set in each element

Mnemonic

vcpop.v vd, vs2, vm

Encoding (Vector)



Arguments

Register	Direction	Definition
Vs2	input	Input elements
Vd	output	Count of bits set

Description

A population count is performed on each element.

Operation

```
function clause execute (VCPop(vs2)) = {  
  
    foreach (i from vstart to vl-1) {  
        let input = get_velem(vs2, SEW, i);  
        let output : bits(SEW) = 0;  
        for (j = 0; j < SEW; j++)  
            output = output + input[j];  
        set_velem(vd, SEW, i, output)  
    }  
    RETIRE_SUCCESS  
}
```

Included in

[Zvbb](#)

3.15. vctz.v

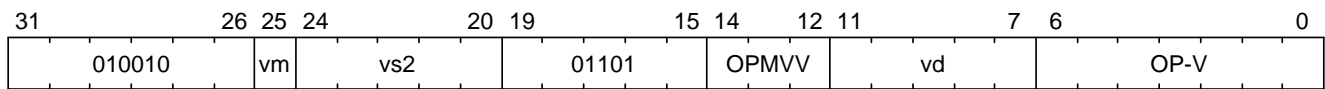
Synopsis

Vector Count Trailing Zeros

Mnemonic

vctz.v vd, vs2, vm

Encoding (Vector)



Arguments

Register	Direction	Definition
Vs2	input	Input elements
Vd	output	Count of trailing zero bits

Description

A trailing zero count is performed on each element.

Operation

```
function clause execute (VCTZ(vs2)) = {  
  
    foreach (i from vstart to vl-1) {  
        let input = get_velem(vs2, SEW, i);  
        for (j = 0; j < SEW; j++)  
            if [input[j]] == 0b1 then break;  
        set_velem(vd, SEW, i, j)  
    }  
    RETIRE_SUCCESS  
}
```

Included in

Zvbb

3.16. vghsh.vv

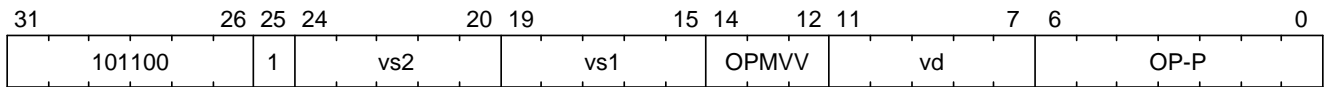
Synopsis

Vector Add-Multiply over GHASH Galois-Field

Mnemonic

vghsh.vv vd, vs2, vs1

Encoding



Reserved Encodings

- **SEW** is any value other than 32

Arguments

Register	Direction	EGW	EGS	SEW	Definition
Vd	input	128	4	32	Partial hash (Y_i)
Vs1	input	128	4	32	Cipher text (X_i)
Vs2	input	128	4	32	Hash Subkey (H)
Vd	output	128	4	32	Partial-hash (Y_{i+1})

Description

A single "iteration" of the GHASH_H algorithm is performed.

This instruction treats all of the inputs and outputs as 128-bit polynomials and performs operations over $\text{GF}[2]$. It produces the next partial hash (Y_{i+1}) by adding the current partial hash (Y_i) to the cipher text block (X_i) and then multiplying (over $\text{GF}(2^{128})$) this sum by the Hash Subkey (H).

The multiplication over $\text{GF}(2^{128})$ is a carryless multiply of two 128-bit polynomials modulo GHASH's irreducible polynomial ($x^{128} + x^7 + x^2 + x + 1$).

The operation can be compactly defined as $Y_{i+1} = ((Y_i \wedge X_i) \cdot H)$

The NIST specification (see [Zvkg](#)) orders the coefficients from left to right $x_0x_1x_2...x_{127}$ for a polynomial $x_0 + x_1u + x_2u^2 + ... + x_{127}u^{127}$. This can be viewed as a collection of byte elements in memory with the byte containing the lowest coefficients (i.e., 0,1,2,3,4,5,6,7) residing at the lowest memory address. Since the bits in the bytes are reversed, This instruction internally performs bit swaps within bytes to put the bits in the standard ordering (e.g., 7,6,5,4,3,2,1,0).

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.



We are bit-reversing the bytes of inputs and outputs so that the intermediate values are consistent with the NIST specification. These reversals are inexpensive to implement as they unconditionally swap bit positions and therefore do not

require any logic.



Since the same hash subkey **H** will typically be used repeatedly on a given message, a future extension might define a vector-scalar version of this instruction where **vs2** is the scalar element group. This would help reduce register pressure when **LMUL** > 1.

Operation

```
function clause execute (VGHSH(vs2, vs1, vd)) = {
  // operands are input with bits reversed in each byte
  if(LMUL*VLEN < EGW) then {
    handle_illegal(); // illegal instruction exception
    RETIRE_FAIL
  } else {

    eg_len = (v1/EGS)
    eg_start = (vstart/EGS)

    foreach (i from eg_start to eg_len-1) {
      let Y = (get_velem(vd,EGW=128,i)); // current partial-hash
      let X = get_velem(vs1,EGW=128,i); // block cipher output
      let H = brev8(get_velem(vs2,EGW=128,i)); // Hash subkey

      let Z : bits(128) = 0;

      let S = brev8(Y ^ X);

      for (int bit = 0; bit < 128; bit++) {
        if bit_to_bool(S[bit])
          Z ^= H

        bool reduce = bit_to_bool(H[127]);
        H = H << 1; // left shift H by 1
        if (reduce)
          H ^= 0x87; // Reduce using x^7 + x^2 + x^1 + 1 polynomial
      }

      let result = brev8(Z); // bit reverse bytes to get back to GCM standard ordering
      set_velem(vd, EGW=128, i, result);
    }
    RETIRE_SUCCESS
  }
}
```

Included in

[Zvkg](#), [Zvkng](#), [Zvksg](#)

3.17. vgmul.vv

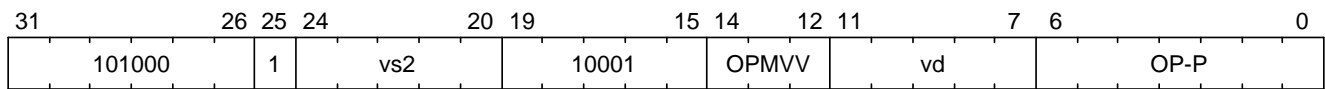
Synopsis

Vector Multiply over GHASH Galois-Field

Mnemonic

vgmul.vv vd, vs2

Encoding



Reserved Encodings

- **SEW** is any value other than 32

Arguments

Register	Direction	EGW	EGS	SEW	Definition
Vd	input	128	4	32	Multiplier
Vs2	input	128	4	32	Multiplicand
Vd	output	128	4	32	Product

Description

A GHASH_H multiply is performed.

This instruction treats all of the inputs and outputs as 128-bit polynomials and performs operations over $\text{GF}[2]$. It produces the product over $\text{GF}(2^{128})$ of the two 128-bit inputs.

The multiplication over $\text{GF}(2^{128})$ is a carryless multiply of two 128-bit polynomials modulo GHASH's irreducible polynomial ($x^{128} + x^7 + x^2 + x + 1$).

The NIST specification (see [Zvkg](#)) orders the coefficients from left to right $x_0x_1x_2\dots x_{127}$ for a polynomial $x_0 + x_1u + x_2u^2 + \dots + x_{127}u^{127}$. This can be viewed as a collection of byte elements in memory with the byte containing the lowest coefficients (i.e., 0,1,2,3,4,5,6,7) residing at the lowest memory address. Since the bits in the bytes are reversed, This instruction internally performs bit swaps within bytes to put the bits in the standard ordering (e.g., 7,6,5,4,3,2,1,0).

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.



We are bit-reversing the bytes of inputs and outputs so that the intermediate values are consistent with the NIST specification. These reversals are inexpensive to implement as they unconditionally swap bit positions and therefore do not require any logic.



Since the same multiplicand will typically be used repeatedly on a given message, a future extension might define a vector-scalar version of this instruction where

vs2 is the scalar element group. This would help reduce register pressure when **LMUL** > 1.



This instruction is identical to **vghsh.vv** with **vs1=0**. This instruction is often used in GHASH code. In some cases it is followed by an XOR to perform a multiply-add. Implementations may choose to fuse these two instructions to improve performance on GHASH code that doesn't use the add-multiply form of the **vghsh.vv** instruction.

Operation

```
function clause execute (VGMUL(vs2, vs1, vd)) = {
  // operands are input with bits reversed in each byte
  if(LMUL*VLEN < EGW) then {
    handle_illegal(); // illegal instruction exception
    RETIRE_FAIL
  } else {

    eg_len = (vL/EGS)
    eg_start = (vstart/EGS)

    foreach (i from eg_start to eg_len-1) {
      let Y = brev8(get_velem(vd,EGW=128,i)); // Multiplier
      let H = brev8(get_velem(vs2,EGW=128,i)); // Multiplicand
      let Z : bits(128) = 0;

      for (int bit = 0; bit < 128; bit++) {
        if bit_to_bool(Y[bit])
          Z ^= H

        bool reduce = bit_to_bool(H[127]);
        H = H << 1; // left shift H by 1
        if (reduce)
          H ^= 0x87; // Reduce using x^7 + x^2 + x^1 + 1 polynomial
      }

      let result = brev8(Z);
      set_velem(vd, EGW=128, i, result);
    }
    RETIRE_SUCCESS
  }
}
```

Included in

[Zvkg](#), [Zvkng](#), [Zvksg](#)

3.18. vrev8.v

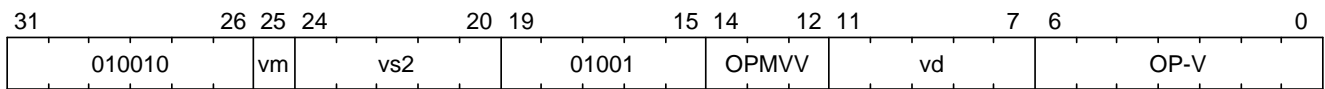
Synopsis

Vector Reverse Bytes

Mnemonic

vrev8.v vd, vs2, vm

Encoding (Vector)



Arguments

Register	Direction	Definition
Vs2	input	Input elements
Vd	output	Byte-reversed elements

Description

A byte reversal is performed on each element of **vs2**, effectively performing an endian swap.



This element-wise endian swapping is needed for several cryptographic algorithms including SHA2 and SM3.

Operation

```
function clause execute (VREV8(vs2)) = {
  foreach (i from vstart to vl-1) {
    input = get_velem(vs2, SEW, i);
    let output : SEW = 0;
    let j = SEW - 1;
    foreach (k from 0 to (SEW - 8) by 8) {
      output[k..(k + 7)] = input[(j - 7)..j];
      j = j - 8;
    }
    set_velem(vd, SEW, i, output)
  }
  RETIRE_SUCCESS
}
```

Included in

[Zvbb](#), [Zvkb](#), [Zvkn](#), [Zvknc](#), [Zvkng](#), [Zvks](#) [Zvksc](#), [Zvksg](#)

3.19. vrol.[vv,vx]

Synopsis

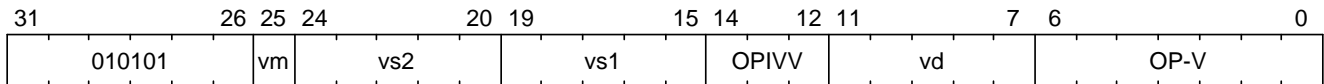
Vector rotate left by vector/scalar.

Mnemonic

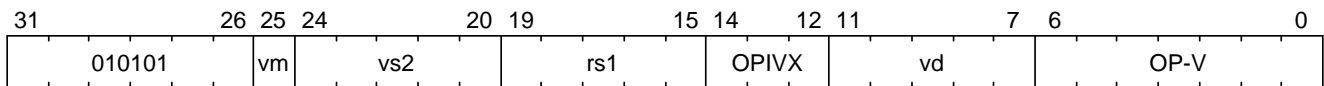
vrol.vv vd, vs2, vs1, vm

vrol.vx vd, vs2, rs1, vm

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Vector-Vector Arguments

Register	Direction	Definition
Vs1	input	Rotate amount
Vs2	input	Data
Vd	output	Rotated data

Vector-Scalar Arguments

Register	Direction	Definition
Rs1	input	Rotate amount
Vs2	input	Data
Vd	output	Rotated data

Description

A bitwise left rotation is performed on each element of **vs2**

The elements in **vs2** are rotated left by the rotate amount specified by either the cotrresponding elements of **vs1** (vector-vector), or integer register **rs1** (vector-scalar). Only the low log2(**SEW**) bits of the rotate-amount value are used, all other bits are ignored.



There is no immediate form of this instruction (i.e., **vrol.vi**) as the same result can be achieved by negating the rotate amount and using the immediate form of rotate right instruction (i.e., **vror.vi**).

Operation

```

function clause execute (VROL_VV(vs2, vs1, vd)) = {
    foreach (i from vstart to vl - 1) {
        set_velem(vd, EEW=SEW, i,
            get_velem(vs2, i) <<< (get_velem(vs1, i) & (SEW-1))
        )
    }
    RETIRE_SUCCESS
}

function clause execute (VROL_VX(vs2, rs1, vd)) = {
    foreach (i from vstart to vl - 1) {
        set_velem(vd, EEW=SEW, i,
            get_velem(vs2, i) <<< (X(rs1) & (SEW-1))
        )
    }
    RETIRE_SUCCESS
}

```

Included in

[Zvbb](#), [Zvkb](#), [Zvkn](#), [Zvknc](#), [Zvkng](#), [Zvks](#) [Zvksc](#), [Zvksg](#)

3.20. vror.[vv,vx,vi]

Synopsis

Vector rotate right by vector/scalar/immediate.

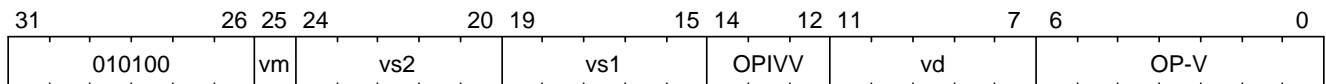
Mnemonic

vror.vv vd, vs2, vs1, vm

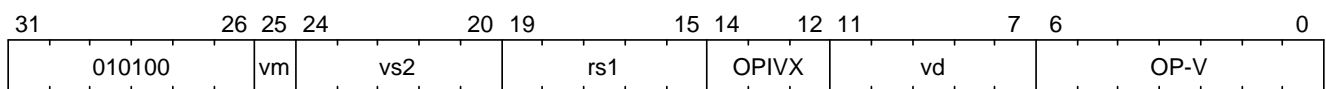
vror.vx vd, vs2, rs1, vm

vror.vi vd, vs2, uimm, vm

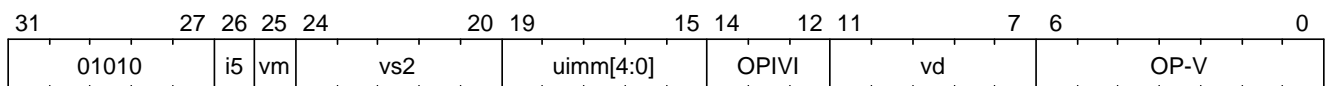
Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Encoding (Vector-Immediate)



Vector-Vector Arguments

Register	Direction	Definition
Vs1	input	Rotate amount
Vs2	input	Data
Vd	output	Rotated data

Vector-Scalar/Immediate Arguments

Register	Direction	Definition
Rs1/imm	input	Rotate amount
Vs2	input	Data
Vd	output	Rotated data

Description

A bitwise right rotation is performed on each element of **vs2**.

The elements in **vs2** are rotated right by the rotate amount specified by either the corresponding elements of **vs1** (vector-vector), integer register **rs1** (vector-scalar), or an immediate value (vector-immediate). Only the low $\log_2(\text{SEW})$ bits of the rotate-amount value are used, all other bits are ignored.

Operation

```
function clause execute (VROR_VV(vs2, vs1, vd)) = {
  foreach (i from vstart to vl - 1) {
    set_velem(vd, EEW=SEW, i,
      get_velem(vs2, i) >>> (get_velem(vs1, i) & (SEW-1))
    )
  }
  RETIRE_SUCCESS
}

function clause execute (VROR_VX(vs2, rs1, vd)) = {
  foreach (i from vstart to vl - 1) {
    set_velem(vd, EEW=SEW, i,
      get_velem(vs2, i) >>> (X(rs1) & (SEW-1))
    )
  }
  RETIRE_SUCCESS
}

function clause execute (VROR_VI(vs2, imm[5:0], vd)) = {
  foreach (i from vstart to vl - 1) {
    set_velem(vd, EEW=SEW, i,
      get_velem(vs2, i) >>> (imm[5:0] & (SEW-1))
    )
  }
  RETIRE_SUCCESS
}
```

Included in

[Zvbb](#), [Zvkb](#), [Zvkn](#), [Zvknc](#), [Zvkng](#), [Zvks](#) [Zvksc](#), [Zvksg](#)

3.21. vsha2c[hl].vv

Synopsis

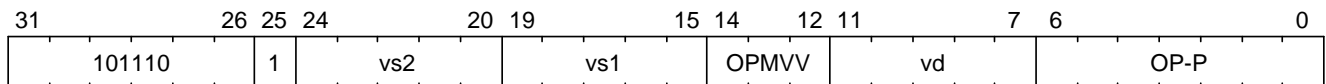
Vector SHA-2 two rounds of compression.

Mnemonic

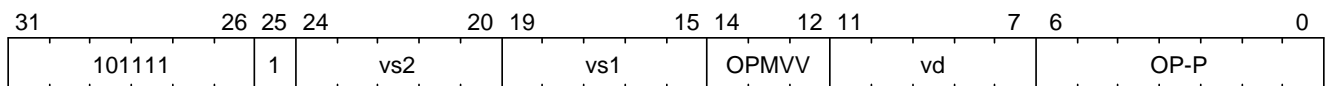
vsha2ch.vv vd, vs2, vs1

vsha2cl.vv vd, vs2, vs1

Encoding (Vector-Vector) High part



Encoding (Vector-Vector) Low part



Reserved Encodings

- **zvknha**: SEW is any value other than 32
- **zvknhb**: SEW is any value other than 32 or 64
- The **vd** register group overlaps with either **vs1** or **vs2**

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vd	input	4*SEW	4	SEW	current state {c, d, g, h}
Vs1	input	4*SEW	4	SEW	MessageSched plus constant[3:0]
Vs2	input	4*SEW	4	SEW	current state {a, b, e, f}
Vd	output	4*SEW	4	SEW	next state {a, b, e, f}

Description

- SEW=32: 2 rounds of SHA-256 compression are performed (**zvknha** and **zvknhb**)
- SEW=64: 2 rounds of SHA-512 compression are performed (**zvkhnb**)

Two words of **vs1** are processed with the 8 words of current state held in **vd** and **vs1** to perform two rounds of hash computation producing four words of the next state.



Note to software developers

The NIST standard (see [zvknh\[ab\]](#)) requires the final hash to be in big-endian byte ordering within SEW-sized words. Since this instruction treats all words as little-endian, software needs to perform an endian swap on the final output of this instruction after all of the message blocks have been processed.



The **vsha2ch** version of this instruction uses the two most significant message schedule words from the element group in **vs1** while the **vsha2cl** version uses the two least significant message schedule words. Otherwise, these versions of the instruction are identical. Having a high and low version of this instruction typically improves performance when interleaving independent hashing operations (i.e., when hashing several files at once).



Preventing overlap between **vd** and **vs1** or **vs2** simplifies implementation with **VLEN < EGW**. This restriction does not have any coding impact since proper implementation of the algorithm requires that **vd**, **vs1** and **vs2** each are different registers.

Operation

```
function clause execute (VSHA2c(vs2, vs1, vd)) = {
  if(LMUL*VLEN < EGW) then {
    handle_illegal(); // illegal instruction exception
    RETIRE_FAIL
  } else {

    eg_len = (vL/EGS)
    eg_start = (vstart/EGS)

    foreach (i from eg_start to eg_len-1) {
      let {a @ b @ e @ f} : bits(4*SEW) = get_velem(vs2, 4*SEW, i);
      let {c @ d @ g @ h} : bits(4*SEW) = get_velem(vd, 4*SEW, i);
      let MessageShedPlusC[3:0] : bits(4*SEW) = get_velem(vs1, 4*SEW, i);
      let {W1, W0} == VSHA2cl ? MessageSchedPlusC[1:0] : MessageSchedPlusC[3:2]; // 1
      vs h difference is the words selected

      let T1 : bits(SEW) = h + sum1(e) + ch(e,f,g) + W0;
      let T2 : bits(SEW) = sum0(a) + maj(a,b,c);
      h = g;
      g = f;
      f = e;
      e = d + T1;
      d = c;
      c = b;
      b = a;
      a = T1 + T2;

      T1 = h + sum1(e) + ch(e,f,g) + W1;
      T2 = sum0(a) + maj(a,b,c);
      h = g;
      g = f;
      f = e;
      e = d + T1;
      d = c;
```

```

    c = b;
    b = a;
    a = T1 + T2;
    set_velem(vd, 4*SEW, i, {a @ b @ e @ f});
}
RETIRE_SUCCESS
}
}

function sum0(x) = {
    match SEW {
        32 => rotr(x,2) XOR rotr(x,13) XOR rotr(x,22),
        64 => rotr(x,28) XOR rotr(x,34) XOR rotr(x,39)
    }
}

function sum1(x) = {
    match SEW {
        32 => rotr(x,6) XOR rotr(x,11) XOR rotr(x,25),
        64 => rotr(x,14) XOR rotr(x,18) XOR rotr(x,41)
    }
}

function ch(x, y, z) = ((x & y) ^ ((~x) & z))

function maj(x, y, z) = ((x & y) ^ (x & z) ^ (y & z))

function ROTR(x,n) = (x >> n) | (x << SEW - n)

```

Included in

[Zvkn](#), [Zvknc](#), [Zvkng](#), [zvknh\[ab\]](#)

3.22. vsha2ms.vv

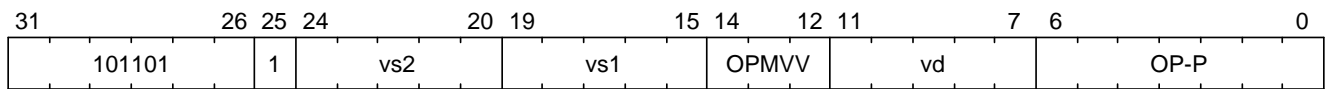
Synopsis

Vector SHA-2 message schedule.

Mnemonic

vsha2ms.vv vd, vs2, vs1

Encoding (Vector-Vector)



Reserved Encodings

- **zvknha**: SEW is any value other than 32
- **zvknhb**: SEW is any value other than 32 or 64
- The **vd** register group overlaps with either **vs1** or **vs2**

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vd	input	4*SEW	4	SEW	Message words {W[3], W[2], W[1], W[0]}
Vs2	input	4*SEW	4	SEW	Message words {W[11], W[10], W[9], W[4]}
Vs1	input	4*SEW	4	SEW	Message words {W[15], W[14], -, W[12]}
Vd	output	4*SEW	4	SEW	Message words {W[19], W[18], W[17], W[16]}

Description

- SEW=32: Four rounds of SHA-256 message schedule expansion are performed (**zvknha** and **zvknhb**)
- SEW=64: Four rounds of SHA-512 message schedule expansion are performed (**zvknhb**)

Eleven of the last 16 SEW-sized message-schedule words from **vd** (oldest), **vs2**, and **vs1** (most recent) are processed to produce the next 4 message-schedule words.



Note to software developers

The first 16 SEW-sized words of the message schedule come from the *message block* in big-endian byte order. Since this instruction treats all words as little endian, software is required to endian swap these words.

All of the subsequent message schedule words are produced by this instruction and therefore do not require an endian swap.



Note to software developers

Software is required to pack the words into element groups as shown above in the arguments table. The indices indicate the relate age with lower indices indicating

older words.



Note to software developers

The $\{W_{11}, W_{10}, W_9, W_4\}$ element group can easily be formed by using a vector vmerge instruction with the appropriate mask (for example with $v1=4$ and $4b0001$ as the 4 mask bits)

```
vmerge.vvm {W11, W10, W9, W4}, {W11, W10, W9, W8}, {W7, W6, W5, W4}, V0
```



Preventing overlap between vd and $vs1$ or $vs2$ simplifies implementation with $VLEN < EGW$. This restriction does not have any coding impact since proper implementation of the algorithm requires that vd , $vs1$ and $vs2$ each contain different portions of the message schedule.

Operation

```
function clause execute (VSHA2ms(vs2, vs1, vd)) = {
  // SEW32 = SHA-256
  // SEW64 = SHA-512
  if(LMUL*VLEN < EGW) then {
    handle_illegal(); // illegal instruction exception
    RETIRE_FAIL
  } else {

    eg_len = (v1/EGS)
    eg_start = (vstart/EGS)

    foreach (i from eg_start to eg_len-1) {
      {W[3] @ W[2] @ W[1] @ W[0]} : bits(EGW) = get_velem(vd, EGW, i);
      {W[11] @ W[10] @ W[9] @ W[4]} : bits(EGW) = get_velem(vs2, EGW, i);
      {W[15] @ W[14] @ W[13] @ W[12]} : bits(EGW) = get_velem(vs1, EGW, i);

      W[16] = sig1(W[14]) + W[9] + sig0(W[1]) + W[0];
      W[17] = sig1(W[15]) + W[10] + sig0(W[2]) + W[1];
      W[18] = sig1(W[16]) + W[11] + sig0(W[3]) + W[2];
      W[19] = sig1(W[17]) + W[12] + sig0(W[4]) + W[3];

      set_velem(vd, EGW, i, {W[19] @ W[18] @ W[17] @ W[16]});
    }
    RETIRE_SUCCESS
  }
}

function sig0(x) = {
  match SEW {
    32 => (ROTR(x,7) XOR ROTR(x,18) XOR SHR(x,3)),
    64 => (ROTR(x,1) XOR ROTR(x,8) XOR SHR(x,7));
  }
}
```

```

function sig1(x) = {
  match SEW {
    32 => (ROTR(x,17) XOR ROTR(x,19) XOR SHR(x,10),
    64 => ROTR(x,19) XOR ROTR(x,61) XOR SHR(x,6));
  }
}

function ROTR(x,n) = (x >> n) | (x << SEW - n)
function SHR (x,n) = x >> n

```

Included in

[Zvkn](#), [Zvknc](#), [Zvkng](#), [zvknh\[ab\]](#)

3.23. vsm3c.vi

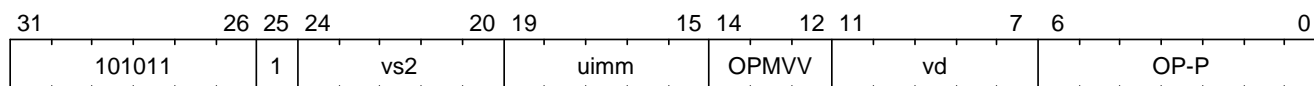
Synopsis

Vector SM3 Compression

Mnemonic

vsm3c.vi vd, vs2, uimm

Encoding



Reserved Encodings

- **SEW** is any value other than 32
- The **vd** register group overlaps with the **vs2** register group

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vd	input	256	8	32	Current state {H,G,F,E,D,C,B,A}
uimm	input	-	-	-	round number (rnds)
Vs2	input	256	8	32	Message words {-,-,w[5],w[4],-,-,w[1],w[0]}
Vd	output	256	8	32	Next state {H,G,F,E,D,C,B,A}

Description

Two rounds of SM3 compression are performed.

The current state of eight 32-bit words is read in as an element group from **vd**. Eight 32-bit message words are read in as an element group from **vs2**, although only four of them are used. All of the 32-bit input words are byte-swapped from big endian to little endian. These inputs are processed somewhat differently based on the round group (as specified in rnds), and the next state is generated as an element group of eight 32-bit words. The next state of eight 32-bit words are generated, swapped from little endian to big endian, and are returned in an eight-element group.

The round number is provided by the 5-bit **rnds** unsigned immediate. Legal values are 0 - 31 and indicate which group of two rounds are being performed. For example, if rnds=1, then rounds 2 and 3 are being performed.



The round number is used in the rotation of the constant as well to inform the behavior which differs between rounds 0-15 and rounds 16-63.



The endian byte swapping of the input and output words enables us to align with the SM3 specification without requiring that software perform these swaps.



Preventing overlap between **vd** and **vs2** simplifies implementation with **VLEN < EGW**.

This restriction does not have any coding impact since proper implementation of the algorithm requires that **vd** and **vs2** each are different registers.

Operation

```
function clause execute (VSM3C(rnds, vs2, vd)) = {
  if(LMUL*VLEN < EGW) then {
    handle_illegal(); // illegal instruction exception
    RETIRE_FAIL
  } else {

    eg_len = (vL/EGS)
    eg_start = (vstart/EGS)

    foreach (i from eg_start to eg_len-1) {

      // load state
      let {Hi @ Gi @ Fi @ Ei @ Di @ Ci @ Bi @ Ai} : bits(256) : bits(256) = (get_velem(vd,
256, i));
      //load message schedule
      let {u_w7 @ u_w6 @ w5i @ w4i @ u_w3 @ u_w2 @ w1i @ w0i} : bits(256) =
(get_velem(vs2, 256, i));
      // u_w inputs are unused

      // perform endian swap
      let H : bits(32) = rev8(Hi);
      let G : bits(32) = rev8(Gi);
      let F : bits(32) = rev8(Fi);
      let E : bits(32) = rev8(Ei);
      let D : bits(32) = rev8(Di);
      let C : bits(32) = rev8(Ci);
      let B : bits(32) = rev8(Bi);
      let A : bits(32) = rev8(Ai);

      let w5 = : bits(32) rev8(w5i);
      let w4 = : bits(32) rev8(w4i);
      let w1 = : bits(32) rev8(w1i);
      let w0 = : bits(32) rev8(w0i);

      let x0 :bits(32) = w0 ^ w4; // W'[0]
      let x1 :bits(32) = w1 ^ w5; // W'[1]

      let j = 2 * rnds;
      let ss1 : bits(32) = ROL32(ROL32(A, 12) + E + ROL32(T_j(j), j % 32), 7);
      let ss2 : bits(32) = ss1 ^ ROL32(A, 12);
      let tt1 : bits(32) = FF_j(A, B, C, j) + D + ss2 + x0;
      let tt2 : bits(32) = GG_j(E, F, G, j) + H + ss1 + w0;
      D = C;
      let : bits(32) C1 = ROL32(B, 9);
      B = A;
```

```

let A1 : bits(32) = tt1;
H = G;
let G1 : bits(32) = ROL32(F, 19);
F = E;
let E1 : bits(32) = P_0(tt2);

j = 2 * rnds + 1;
ss1 = ROL32(ROL32(A1, 12) + E1 + ROL32(T_j(j), j % 32), 7);
ss2 = ss1 ^ ROL32(A1, 12);
tt1 = FF_j(A1, B, C1, j) + D + ss2 + x1;
tt2 = GG_j(E1, F, G1, j) + H + ss1 + w1;
D = C1;
let C2 : bits(32) = ROL32(B, 9);
B = A1;
let A2 : bits(32) = tt1;
H = G1;
let G2 = : bits(32) ROL32(F, 19);
F = E1;
let E2 = : bits(32) P_0(tt2);

// Update the destination register - swap back to big endian
let result : bits(256) = {rev8(G1) @ rev8(G2) @ rev8(E1) @ rev8(E2) @ rev8(C1) @
rev8(C2) @ rev8(A1) @ rev8(A2)};
set_velem(vd, 256, i, result);
}

RETIRE_SUCCESS
}
}

function FF1(X, Y, Z) = ((X) ^ (Y) ^ (Z))
function FF2(X, Y, Z) = (((X) & (Y)) | ((X) & (Z)) | ((Y) & (Z)))

function FF_j(X, Y, Z, J) = (((J) <= 15) ? FF1(X, Y, Z) : FF2(X, Y, Z))

function GG1(X, Y, Z) = ((X) ^ (Y) ^ (Z))
function GG2(X, Y, Z) = (((X) & (Y)) | ((~(X)) & (Z)))
.
function GG_j(X, Y, Z, J) = (((J) <= 15) ? GG1(X, Y, Z) : GG2(X, Y, Z))

function T_j(J) = (((J) <= 15) ? (0x79CC4519) : (0x7A879D8A))

function P_0(X) = ((X) ^ ROL32((X), 9) ^ ROL32((X), 17))

```

Included in

[Zvks](#), [Zvksc](#), [Zvksg](#), [Zvksh](#)

3.24. vsm3me.vv

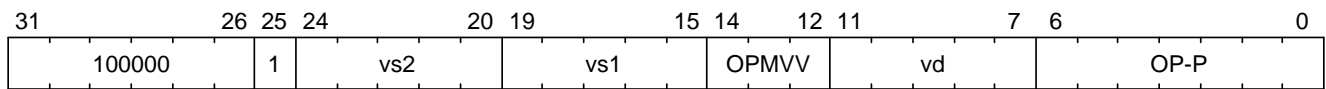
Synopsis

Vector SM3 Message Expansion

Mnemonic

vsm3me.vv vd, vs2, vs1

Encoding



Reserved Encodings

- **SEW** is any value other than 32
- The **vd** register group overlaps with the **vs2** register group.

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vs1	input	256	8	32	Message words W[7:0]
Vs2	input	256	8	32	Message words W[15:8]
Vd	output	256	8	32	Message words W[23:16]

Description

Eight rounds of SM3 message expansion are performed.

The sixteen most recent 32-bit message words are read in as two eight-element groups from **vs1** and **vs2**. Each of these words is swapped from big endian to little endian. The next eight 32-bit message words are generated, swapped from little endian to big endian, and are returned in an eight-element group.



The endian byte swapping of the input and output words enables us to align with the SM3 specification without requiring that software perform these swaps.



Preventing overlap between **vd** and **vs2** simplifies implementations with **VLEN < EGW**. This restriction should not have any coding impact since the algorithm requires these values to be preserved for generating the next 8 words.

Operation

```
function clause execute (VSM3ME(vs2, vs1)) = {  
  if(LMUL*VLEN < EGW) then {  
    handle_illegal(); // illegal instruction exception  
    RETIRE_FAIL  
  } else {
```

```

eg_len = (vl/EGS)
eg_start = (vstart/EGS)

foreach (i from eg_start to eg_len-1) {
    let w[7:0] : bits(256) = get_velem(vs1, 256, i);
    let w[15:8] : bits(256) = get_velem(vs2, 256, i);

    // Byte Swap inputs from big-endian to little-endian
    let w15 = rev8(w[15]);
    let w14 = rev8(w[14]);
    let w13 = rev8(w[13]);
    let w12 = rev8(w[12]);
    let w11 = rev8(w[11]);
    let w10 = rev8(w[10]);
    let w9 = rev8(w[9]);
    let w8 = rev8(w[8]);
    let w7 = rev8(w[7]);
    let w6 = rev8(w[6]);
    let w5 = rev8(w[5]);
    let w4 = rev8(w[4]);
    let w3 = rev8(w[3]);
    let w2 = rev8(w[2]);
    let w1 = rev8(w[1]);
    let w0 = rev8(w[0]);

    // Note that some of the newly computed words are used in later invocations.
    let w[16] = ZVKSH_W(w0 @ w7 @ w13 @ w3 @ w10);
    let w[17] = ZVKSH_W(w1 @ w8 @ w14 @ w4 @ w11);
    let w[18] = ZVKSH_W(w2 @ w9 @ w15 @ w5 @ w12);
    let w[19] = ZVKSH_W(w3 @ w10 @ w16 @ w6 @ w13);
    let w[20] = ZVKSH_W(w4 @ w11 @ w17 @ w7 @ w14);
    let w[21] = ZVKSH_W(w5 @ w12 @ w18 @ w8 @ w15);
    let w[22] = ZVKSH_W(w6 @ w13 @ w19 @ w9 @ w16);
    let w[23] = ZVKSH_W(w7 @ w14 @ w20 @ w10 @ w17);

    // Byte swap outputs from little-endian back to big-endian
    let w16 : Bits(32) = rev8(W[16]);
    let w17 : Bits(32) = rev8(W[17]);
    let w18 : Bits(32) = rev8(W[18]);
    let w19 : Bits(32) = rev8(W[19]);
    let w20 : Bits(32) = rev8(W[20]);
    let w21 : Bits(32) = rev8(W[21]);
    let w22 : Bits(32) = rev8(W[22]);
    let w23 : Bits(32) = rev8(W[23]);

    // Update the destination register.
    set_velem(vd, 256, i, {w23 @ w22 @ w21 @ w20 @ w19 @ w18 @ w17 @ w16});
}
RETIRE_SUCCESS

```

```

}
}

function P_1(X) ((X) ^ ROL32((X), 15) ^ ROL32((X), 23))

function ZVKSH_W(M16, M9, M3, M13, M6) = \
(P1( (M16) ^ (M9) ^ ROL32((M3), 15) ) ^ ROL32((M13), 7) ^ (M6))

```

Included in

[Zvks](#), [Zvksc](#), [Zvksg](#), [Zvksh](#)

3.25. vsm4k.vi

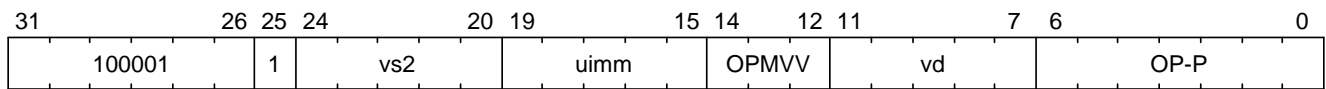
Synopsis

Vector SM4 KeyExpansion

Mnemonic

vsm4k.vi vd, vs2, uimm

Encoding



Reserved Encodings

- **SEW** is any value other than 32

Arguments

Register	Direction	EGW	EGS	EEW	Definition
uimm	input	-	-	-	Round group (rnd)
Vs2	input	128	4	32	Current 4 round keys rK[0:3]
Vd	output	128	4	32	Next 4 round keys rK'[0:3]

Description

Four rounds of the SM4 Key Expansion are performed.

Four round keys are read in as a 4-element group from **vs2**. Each of the next four round keys are generated by iteratively XORing the last three round keys with a constant that is indexed by the Round Group Number, performing a byte-wise substitution, and then performing XORs between rotated versions of this value and the corresponding current round key.

The Round group number (**rnd**) comes from **uimm[2:0]**; the bits in **uimm[4:3]** are ignored. Round group numbers range from 0 to 7 and indicate which group of four round keys are being generated. Round Keys range from 0-31. For example, if **rnd**=1, then round keys 4, 5, 6, and 7 are being generated.



Software needs to generate the initial round keys. This is done by XORing the 128-bit encryption key with the system parameters: FK[0:3]

Table 1. System Parameters

FK	constant
0	A3B1BAC6
1	56AA3350
2	677D9197

FK	constant
3	B27022DC



Implementation Hint

The round constants (CK) can be generated on the fly fairly cheaply. If the bytes of the constants are assigned an incrementing index from 0 to 127, the value of each byte is equal to its index multiplied by 7 modulo 256. Since the results are all limited to 8 bits, the modulo operation occurs for free:

$$\begin{aligned} B[n] &= n + 2n + 4n; \\ &= 8n + \sim n + 1; \end{aligned}$$

Operation

```
function clause execute (vsm4k(uimm, vs2)) = {
  if(LMUL*VLEN < EGW) then {
    handle_illegal(); // illegal instruction exception
    RETIRE_FAIL
  } else {

    eg_len = (vl/EGS)
    eg_start = (vstart/EGS)

    let B : bits(32) = 0;
    let S : bits(32) = 0;
    let rk4 : bits(32) = 0;
    let rk5 : bits(32) = 0;
    let rk6 : bits(32) = 0;
    let rk7 : bits(32) = 0;
    let rnd : bits(3) = uimm[2:0]; // Lower 3 bits

    foreach (i from eg_start to eg_len-1) {
      let (rk3 @ rk2 @ rk1 @ rk0) : bits(128) = get_velem(vs2, 128, i);

      B = rk1 ^ rk2 ^ rk3 ^ ck(4 * rnd);
      S = sm4_subword(B);
      rk4 = ROUND_KEY(rk0, S);

      B = rk2 ^ rk3 ^ rk4 ^ ck(4 * rnd + 1);
      S = sm4_subword(B);
      rk5 = ROUND_KEY(rk1, S);

      B = rk3 ^ rk4 ^ rk5 ^ ck(4 * rnd + 2);
      S = sm4_subword(B);
      rk6 = ROUND_KEY(rk2, S);

      B = rk4 ^ rk5 ^ rk6 ^ ck(4 * rnd + 3);
```

```

    S = sm4_subword(B);
    rk7 = ROUND_KEY(rk3, S);

    // Update the destination register.
    set_velem(vd, EGW=128, i, (rk7 @ rk6 @ rk5 @ rk4));
}
RETIRE_SUCCESS
}
}

val round_key : bits(32) -> bits(32)
function ROUND_KEY(X, S) = ((X) ^ ((S) ^ ROL32((S), 13) ^ ROL32((S), 23)))

// SM4 Constant Key (CK)
let ck : list(bits(32)) = [|
    0x00070E15, 0x1C232A31, 0x383F464D, 0x545B6269,
    0x70777E85, 0x8C939AA1, 0xA8AFB6BD, 0xC4CBD2D9,
    0xE0E7EEF5, 0xFC030A11, 0x181F262D, 0x343B4249,
    0x50575E65, 0x6C737A81, 0x888F969D, 0xA4ABB2B9,
    0xC0C7CED5, 0xDCE3EAF1, 0xF8FF060D, 0x141B2229,
    0x30373E45, 0x4C535A61, 0x686F767D, 0x848B9299,
    0xA0A7AEB5, 0xBCC3CAD1, 0xD8DFE6ED, 0xF4FB0209,
    0x10171E25, 0x2C333A41, 0x484F565D, 0x646B7279
|]
};

```

Included in

[Zvks](#), [Zvksc](#), [Zvksed](#), [Zvksg](#)

3.26. vsm4r.[vv,vs]

Synopsis

Vector SM4 Rounds

Mnemonic

vsm4r.vv vd, vs2

vsm4r.vs vd, vs2

Encoding (Vector-Vector)

31	26	25	24	20	19	15	14	12	11	7	6	0
101000	1	vs2	10000	OPMVV	vd	OP-P						

Encoding (Vector-Scalar)

31	26	25	24	20	19	15	14	12	11	7	6	0
101001	1	vs2	10000	OPMVV	vd	OP-P						

Reserved Encodings

- **SEW** is any value other than 32
- Only for the **.vs** form: the **vd** register group overlaps the **vs2** register

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vd	input	128	4	32	Current state X[0:3]
Vs2	input	128	4	32	Round keys rk[0:3]
Vd	output	128	4	32	Next state X'[0:3]

Description

Four rounds of SM4 Encryption/Decryption are performed.

The four words of current state are read as a 4-element group from 'vd' and the round keys are read from either the corresponding 4-element group in **vs2** (vector-vector form) or the scalar element group in **vs2** (vector-scalar form). The next four words of state are generated by iteratively XORing the last three words of the state with the corresponding round key, performing a byte-wise substitution, and then performing XORs between rotated versions of this value and the corresponding current state.



In SM4, encryption and decryption are identical except that decryption consumes the round keys in the reverse order.



For the first four rounds of encryption, the *current state* is the plain text. For the first four rounds of decryption, the *current state* is the cipher text. For all subsequent rounds, the *current state* is the *next state* from the previous four

rounds.

Operation

```
function clause execute (VSM4R(vd, vs2)) = {
  if(LMUL*VLEN < EGW) then {
    handle_illegal(); // illegal instruction exception
    RETIRE_FAIL
  } else {

    eg_len = (vl/EGS)
    eg_start = (vstart/EGS)

    let B : bits(32) = 0;
    let S : bits(32) = 0;
    let rk0 : bits(32) = 0;
    let rk1 : bits(32) = 0;
    let rk2 : bits(32) = 0;
    let rk3 : bits(32) = 0;
    let x0 : bits(32) = 0;
    let x1 : bits(32) = 0;
    let x2 : bits(32) = 0;
    let x3 : bits(32) = 0;
    let x4 : bits(32) = 0;
    let x5 : bits(32) = 0;
    let x6 : bits(32) = 0;
    let x7 : bits(32) = 0;

    let keyelem : bits(32) = 0;

    foreach (i from eg_start to eg_len-1) {
      keyelem = if suffix == "vv" then i else 0;
      {rk3 @ rk2 @ rk1 @ rk0} : bits(128) = get_velem(vs2, EGW=128, keyelem);
      {x3 @ x2 @ x1 @ x0} : bits(128) = get_velem(vd, EGW=128, i);

      B = x1 ^ x2 ^ x3 ^ rk0;
      S = sm4_subword(B);
      x4 = sm4_round(x0, S);

      B = x2 ^ x3 ^ x4 ^ rk1;
      S = sm4_subword(B);
      x5 = sm4_round(x1, S);

      B = x3 ^ x4 ^ x5 ^ rk2;
      S = sm4_subword(B);
      x6 = sm4_round(x2, S);

      B = x4 ^ x5 ^ x6 ^ rk3;
      S = sm4_subword(B);
      x7 = sm4_round(x3, S);
```

```

    set_velem(vd, EGW=128, i, (x7 @ x6 @ x5 @ x4));

}
RETIRE_SUCCESS
}
}

val sm4_round : bits(32) -> bits(32)
function sm4_round(X, S) = \
  ((X) ^ ((S) ^ ROL32((S), 2) ^ ROL32((S), 10) ^ ROL32((S), 18) ^ ROL32((S), 24)))

```

Included in

[Zvks](#), [Zvksc](#), [Zvk sed](#), [Zvksg](#)

3.27. vwsll.[vv,vx,vi]

Synopsis

Vector widening shift left logical by vector/scalar/immediate.

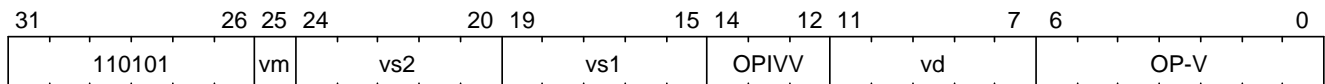
Mnemonic

vwsll.vv vd, vs2, vs1, vm

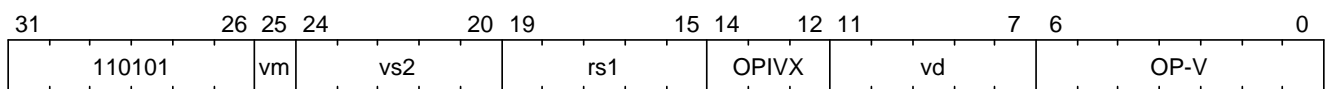
vwsll.vx vd, vs2, rs1, vm

vwsll.vi vd, vs2, uimm, vm

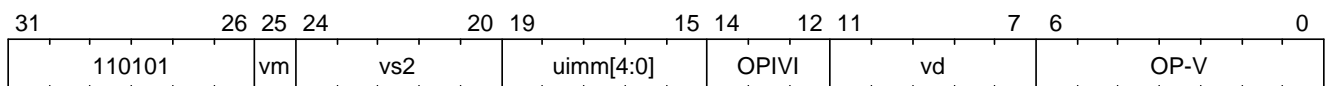
Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Encoding (Vector-Immediate)



Vector-Vector Arguments

Register	Direction	Definition
Vs1	input	Shift amount
Vs2	input	Data
Vd	output	Shifted data

Vector-Scalar/Immediate Arguments

Register	Direction	EEW	Definition
Rs1/imm	input	SEW	Shift amount
Vs2	input	SEW	Data
Vd	output	2*SEW	Shifted data

Description

A widening logical shift left is performed on each element of **vs2**.

The elements in **vs2** are zero-extended to 2***SEW** bits, then shifted left by the shift amount specified by either the corresponding elements of **vs1** (vector-vector), integer register **rs1** (vector-scalar), or an immediate value (vector-immediate). Only the low $\log_2(2*\text{SEW})$ bits of the shift-amount value are used, all other bits are ignored.

Operation

```
function clause execute (VWSSL_VV(vs2, vs1, vd)) = {
  foreach (i from vstart to vl - 1) {
    set_velem(vd, EEW=2*SEW, i,
      get_velem(vs2, i) << (get_velem(vs1, i) & ((2*SEW)-1))
    )
  }
  RETIRE_SUCCESS
}

function clause execute (VWSSL_VX(vs2, rs1, vd)) = {
  foreach (i from vstart to vl - 1) {
    set_velem(vd, EEW=2*SEW, i,
      get_velem(vs2, i) << (X(rs1) & ((2*SEW)-1))
    )
  }
  RETIRE_SUCCESS
}

function clause execute (VWSSL_VI(vs2, uimm[4:0], vd)) = {
  foreach (i from vstart to vl - 1) {
    set_velem(vd, EEW=2*SEW, i,
      get_velem(vs2, i) << (uimm[4:0] & ((2*SEW)-1))
    )
  }
  RETIRE_SUCCESS
}
```

Included in

[Zvbb](#)

Chapter 4. Bibliography

- [1] “SAIL ISA Specification Language.” [Online]. Available: github.com/remis-project/sail.
- [2] “GB/T 32907-2016: SM4 Block Cipher Algorithm.” Also GM/T 0002-2012. Standardization Administration of China, Aug. 2016, [Online]. Available: www.gmbz.org.cn/upload/2018-04-04/1522788048733065051.pdf.
- [3] M. Dworkin, “Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC.” NIST Special Publication SP 800-38D, Nov. 2007, [Online]. Available: doi.org/10.6028/NIST.SP.800-38D.
- [4] R. B. Lee, Z. J. Shi, Y. L. Yin, R. L. Rivest, and M. J. B. Robshaw, “On permutation operations in cipher design,” in *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004.*, 2004, vol. 2, pp. 569–577.
- [5] NIST, “Advanced Encryption Standard (AES).” Federal Information Processing Standards Publication FIPS 197, Nov. 2001, [Online]. Available: doi.org/10.6028/NIST.FIPS.197.
- [6] NIST, “Secure Hash Standard (SHS).” Federal Information Processing Standards Publication FIPS 180-4, Aug. 2015, [Online]. Available: doi.org/10.6028/NIST.FIPS.180-4.

Chapter 5. Encodings

Appendix A: Crypto Vector Cryptographic Instructions

OP-P (0x77) Crypto Vector instructions except Zvbb and Zvbc

Integer					Integer				FP			
funct3					funct3				funct3			
OPIVV	V				OPMVV	V			OPFVV	V		
OPIVX		X			OPMVX		X		OPFVF		F	
OPIVI			I									

funct6					funct6			funct6			
100000					100000	V	vsm3me	100000			
100001					100001	V	vsm4k.vi	100001			
100010					100010	V	vaesfk1.vi	100010			
100011					100011			100011			
100100					100100			100100			
100101					100101			100101			
100110					100110			100110			
100111					100111			100111			
101000					101000	V	VAES.vv	101000			
101001					101001	V	VAES.vs	101001			
101010					101010	V	vaesfk2.vi	101010			
101011					101011	V	vsm3c.vi	101011			
101100					101100	V	vghsh	101100			
101101					101101	V	vsha2ms	101101			
101110					101110	V	vsha2ch	101110			
101111					101111	V	vsha2cl	101111			

Table 2. VAES.vv and VAES.vs encoding space

vs1	
00000	vaesdm
00001	vaesdf
00010	vaesem
00011	vaesef
00111	vaesz
10000	vsm4r
10001	vgmul

Appendix B: Vector Bitmanip and Carryless Multiply Instructions

OP-V (0x57) **Zvbb**, **Zvkb**, and **Zvbc** Vector instructions **in bold**

Integer				Integer				FP			
funct3				funct3				funct3			
OPIVV	V			OPMVV	V			OPFVV	V		
OPIVX		X		OPMVX		X		OPFVF		F	
OPIVI			I								

funct6					funct6					funct6			
000000	V	X	I	vadd	000000	V		vredsum	000000	V	F	vfadd	
000001	V	X		vandn	000001	V		vredand	000001	V		vfredusum	
000010	V	X		vsub	000010	V		vredor	000010	V	F	vfsub	
000011		X	I	vrsb	000011	V		vredxor	000011	V		vfredosum	
000100	V	X		vminu	000100	V		vredminu	000100	V	F	vfmin	
000101	V	X		vmin	000101	V		vredmin	000101	V		vfredmin	
000110	V	X		vmaxu	000110	V		vredmaxu	000110	V	F	vfmax	
000111	V	X		vmax	000111	V		vredmax	000111	V		vfredmax	
001000					001000	V	X	vaaddu	001000	V	F	vfsgnj	
001001	V	X	I	vand	001001	V	X	vaadd	001001	V	F	vfsgnjn	
001010	V	X	I	vor	001010	V	X	vasubu	001010	V	F	vfsgnjx	
001011	V	X	I	vxor	001011	V	X	vasub	001011				
001100	V	X	I	vrgather	001100	V	X	vclmul	001100				
001101					001101	V	X	vclmulh	001101				

funct6				funct6				funct6				
001110		X	I	vslideup	001110		X	vslide1up	001110		F	vfslide1up
001110	V			vrgatherei16								
001111		X	I	vslidedown	001111		X	vslide1down	001111		F	vfslide1down

funct6					funct6					funct6				
010000	V	X	I	vadc	010000	V			VWXUNARY0	010000	V			VWFUNARY0
					010000		X		VRXUNARY0	010000		F		VRFUNARY0
010001	V	X	I	vmadc	010001					010001				
010010	V	X		vsbc	010010	V			VXUNARY0	010010	V			VFUNARY0
010011	V	X		vmsbc	010011					010011	V			VFUNARY1
010100	V	X		vror	010100	V			VMUNARY0	010100				
010101	V	X		vrol	010101					010101				
01010x			I	vror										
010110					010110					010110				
010111	V	X	I	vmerge/vmv	010111	V			vcompress	010111		F		vfmerge/vfmv
011000	V	X	I	vmseq	011000	V			vmandn	011000	V	F		vmfeq
011001	V	X	I	vmsne	011001	V			vmand	011001	V	F		vmfle
011010	V	X		vmsltu	011010	V			vmor	011010				
011011	V	X		vmslt	011011	V			vmxor	011011	V	F		vmflt
011100	V	X	I	vmsleu	011100	V			vmorn	011100	V	F		vmfne
011101	V	X	I	vmsle	011101	V			vmnand	011101		F		vmfgt
011110		X	I	vmsgtu	011110	V			vmnor	011110				
011111		X	I	vmsgt	011111	V			vmxnor	011111		F		vmfge

funct6					funct6					funct6				
100000	V	X	I	vsaddu	100000	V	X	vdivu	100000	V	F	vfdiv		
100001	V	X	I	vsadd	100001	V	X	vdiv	100001		F	vfrdiv		
100010	V	X		vssubu	100010	V	X	vremu	100010					
100011	V	X		vssub	100011	V	X	vrem	100011					
100100					100100	V	X	vmulhu	100100	V	F	vfmul		
100101	V	X	I	vsll	100101	V	X	vmul	100101					
100110					100110	V	X	vmulhsu	100110					
100111	V	X		vsmul	100111	V	X	vmulh	100111		F	vfrsub		
			I	vmv<nr>r										

funct6					funct6					funct6				
101000	V	X	I	vsrl	101000					101000	V	F		vfmadd
101001	V	X	I	vsra	101001	V	X		vmadd	101001	V	F		vfnmadd
101010	V	X	I	vssrl	101010					101010	V	F		vfmsub
101011	V	X	I	vssra	101011	V	X		vnmsub	101011	V	F		vfnmsub
101100	V	X	I	vnsrl	101100					101100	V	F		vfmacc
101101	V	X	I	vnsra	101101	V	X		vmacc	101101	V	F		vfnmacc
101110	V	X	I	vnclipu	101110					101110	V	F		vfmsac
101111	V	X	I	vnclip	101111	V	X		vnmsac	101111	V	F		vfnmsac

funct6					funct6					funct6				
110000	V			vwredsumu	110000	V	X		vwaddu	110000	V	F		vfwadd
110001	V			vwredsum	110001	V	X		vwadd	110001	V			vfwredusum
110010					110010	V	X		vwsubu	110010	V	F		vfwsu
110011					110011	V	X		vwsu	110011	V			vfwredosum
110100					110100	V	X		vwaddu.w	110100	V	F		vfwadd.w
110101	V	X	I	vwsll	110101	V	X		vwadd.w	110101				
110110					110110	V	X		vwsu.w	110110	V	F		vfwsu.w
110111					110111	V	X		vwsu.w	110111				
111000					111000	V	X		vwmulu	111000	V	F		vfwmul
111001					111001					111001				
111010					111010	V	X		vwmulsu	111010				
111011					111011	V	X		vwmul	111011				
111100					111100	V	X		vwmaccu	111100	V	F		vfwmacc
111101					111101	V	X		vwmacc	111101	V	F		vfwnmacc
111110					111110		X		vwmaccus	111110	V	F		vfwmsac
111111					111111	V	X		vwmaccsu	111111	V	F		vfwnmsac

Table 3.
VXUNARY0
encoding space

vs1	
00010	vzext.vf8
00011	vsext.vf8
00100	vzext.vf4
00101	vsext.vf4
00110	vzext.vf2
00111	vsext.vf2
01000	vbrev8
01001	vrev8
01010	vbrev
01100	vclz
01101	vctz
01110	vcpop

Appendix C: Supporting Sail Code

This section contains the supporting Sail code referenced by the instruction descriptions throughout the specification. The [Sail Manual](#) is recommended reading in order to best understand the supporting code.

```
// include:.../extern/sail-riscv/model/riscv_types_kext.sail[lines=12..-1]

/* Auxiliary function for performing GF multiplicaiton */
val xt2 : bits(8) -> bits(8)
function xt2(x) = {
  (x << 1) ^ (if bit_to_bool(x[7]) then 0x1b else 0x00)
}

val xt3 : bits(8) -> bits(8)
function xt3(x) = x ^ xt2(x)

/* Multiply 8-bit field element by 4-bit value for AES MixCols step */
val gfmul : (bits(8), bits(4)) -> bits(8)
function gfmul( x, y) = {
  (if bit_to_bool(y[0]) then x else 0x00) ^
  (if bit_to_bool(y[1]) then xt2(x) else 0x00) ^
  (if bit_to_bool(y[2]) then xt2(xt2(x)) else 0x00) ^
  (if bit_to_bool(y[3]) then xt2(xt2(xt2(x))) else 0x00)
}

/* 8-bit to 32-bit partial AES Mix Colum - forwards */
val aes_mixcolumn_byte_fwd : bits(8) -> bits(32)
function aes_mixcolumn_byte_fwd(so) = {
  gfmul(so, 0x3) @ so @ so @ gfmul(so, 0x2)
}

/* 8-bit to 32-bit partial AES Mix Colum - inverse*/
val aes_mixcolumn_byte_inv : bits(8) -> bits(32)
function aes_mixcolumn_byte_inv(so) = {
  gfmul(so, 0xb) @ gfmul(so, 0xd) @ gfmul(so, 0x9) @ gfmul(so, 0xe)
}

/* 32-bit to 32-bit AES forward MixColumn */
val aes_mixcolumn_fwd : bits(32) -> bits(32)
function aes_mixcolumn_fwd(x) = {
  let s0 : bits (8) = x[ 7.. 0];
  let s1 : bits (8) = x[15.. 8];
  let s2 : bits (8) = x[23..16];
  let s3 : bits (8) = x[31..24];
  let b0 : bits (8) = xt2(s0) ^ xt3(s1) ^ (s2) ^ (s3);
  let b1 : bits (8) = (s0) ^ xt2(s1) ^ xt3(s2) ^ (s3);
  let b2 : bits (8) = (s0) ^ (s1) ^ xt2(s2) ^ xt3(s3);
  let b3 : bits (8) = xt3(s0) ^ (s1) ^ (s2) ^ xt2(s3);
```

```

    b3 @ b2 @ b1 @ b0 /* Return value */
}

/* 32-bit to 32-bit AES inverse MixColumn */
val aes_mixcolumn_inv : bits(32) -> bits(32)
function aes_mixcolumn_inv(x) = {
    let s0 : bits (8) = x[ 7.. 0];
    let s1 : bits (8) = x[15.. 8];
    let s2 : bits (8) = x[23..16];
    let s3 : bits (8) = x[31..24];
    let b0 : bits (8) = gfmul(s0, 0xE) ^ gfmul(s1, 0xB) ^ gfmul(s2, 0xD) ^ gfmul(s3,
0x9);
    let b1 : bits (8) = gfmul(s0, 0x9) ^ gfmul(s1, 0xE) ^ gfmul(s2, 0xB) ^ gfmul(s3,
0xD);
    let b2 : bits (8) = gfmul(s0, 0xD) ^ gfmul(s1, 0x9) ^ gfmul(s2, 0xE) ^ gfmul(s3,
0xB);
    let b3 : bits (8) = gfmul(s0, 0xB) ^ gfmul(s1, 0xD) ^ gfmul(s2, 0x9) ^ gfmul(s3,
0xE);
    b3 @ b2 @ b1 @ b0 /* Return value */
}

val aes_decode_rcon : bits(4) -> bits(32)
function aes_decode_rcon(r) = {
    match r {
        0x0 => 0x00000001,
        0x1 => 0x00000002,
        0x2 => 0x00000004,
        0x3 => 0x00000008,
        0x4 => 0x00000010,
        0x5 => 0x00000020,
        0x6 => 0x00000040,
        0x7 => 0x00000080,
        0x8 => 0x0000001b,
        0x9 => 0x00000036,
        0xA => 0x00000000,
        0xB => 0x00000000,
        0xC => 0x00000000,
        0xD => 0x00000000,
        0xE => 0x00000000,
        0xF => 0x00000000
    }
}

/* SM4 SBox - only one sbox for forwards and inverse */
let sm4_sbox_table : list(bits(8)) = [
0xD6, 0x90, 0xE9, 0xFE, 0xCC, 0xE1, 0x3D, 0xB7, 0x16, 0xB6, 0x14, 0xC2, 0x28,
0xFB, 0x2C, 0x05, 0x2B, 0x67, 0x9A, 0x76, 0x2A, 0xBE, 0x04, 0xC3, 0xAA, 0x44,
0x13, 0x26, 0x49, 0x86, 0x06, 0x99, 0x9C, 0x42, 0x50, 0xF4, 0x91, 0xEF, 0x98,
0x7A, 0x33, 0x54, 0x0B, 0x43, 0xED, 0xCF, 0xAC, 0x62, 0xE4, 0xB3, 0x1C, 0xA9,
0xC9, 0x08, 0xE8, 0x95, 0x80, 0xDF, 0x94, 0xFA, 0x75, 0x8F, 0x3F, 0xA6, 0x47,
0x07, 0xA7, 0xFC, 0xF3, 0x73, 0x17, 0xBA, 0x83, 0x59, 0x3C, 0x19, 0xE6, 0x85,

```

```

0x4F, 0xA8, 0x68, 0x6B, 0x81, 0xB2, 0x71, 0x64, 0xDA, 0x8B, 0xF8, 0xEB, 0x0F,
0x4B, 0x70, 0x56, 0x9D, 0x35, 0x1E, 0x24, 0x0E, 0x5E, 0x63, 0x58, 0xD1, 0xA2,
0x25, 0x22, 0x7C, 0x3B, 0x01, 0x21, 0x78, 0x87, 0xD4, 0x00, 0x46, 0x57, 0x9F,
0xD3, 0x27, 0x52, 0x4C, 0x36, 0x02, 0xE7, 0xA0, 0xC4, 0xC8, 0x9E, 0xEA, 0xBF,
0x8A, 0xD2, 0x40, 0xC7, 0x38, 0xB5, 0xA3, 0xF7, 0xF2, 0xCE, 0xF9, 0x61, 0x15,
0xA1, 0xE0, 0xAE, 0x5D, 0xA4, 0x9B, 0x34, 0x1A, 0x55, 0xAD, 0x93, 0x32, 0x30,
0xF5, 0x8C, 0xB1, 0xE3, 0x1D, 0xF6, 0xE2, 0x2E, 0x82, 0x66, 0xCA, 0x60, 0xC0,
0x29, 0x23, 0xAB, 0x0D, 0x53, 0x4E, 0x6F, 0xD5, 0xDB, 0x37, 0x45, 0xDE, 0xFD,
0x8E, 0x2F, 0x03, 0xFF, 0x6A, 0x72, 0x6D, 0x6C, 0x5B, 0x51, 0x8D, 0x1B, 0xAF,
0x92, 0xBB, 0xDD, 0xBC, 0x7F, 0x11, 0xD9, 0x5C, 0x41, 0x1F, 0x10, 0x5A, 0xD8,
0x0A, 0xC1, 0x31, 0x88, 0xA5, 0xCD, 0x7B, 0xBD, 0x2D, 0x74, 0xD0, 0x12, 0xB8,
0xE5, 0xB4, 0xB0, 0x89, 0x69, 0x97, 0x4A, 0x0C, 0x96, 0x77, 0x7E, 0x65, 0xB9,
0xF1, 0x09, 0xC5, 0x6E, 0xC6, 0x84, 0x18, 0xF0, 0x7D, 0xEC, 0x3A, 0xDC, 0x4D,
0x20, 0x79, 0xEE, 0x5F, 0x3E, 0xD7, 0xCB, 0x39, 0x48

```

```
[ ]
```

```

let aes_sbox_fwd_table : list(bits(8)) = [ |
0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe,
0xd7, 0xab, 0x76, 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4,
0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, 0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7,
0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, 0x04, 0xc7, 0x23, 0xc3,
0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, 0x09,
0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3,
0x2f, 0x84, 0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe,
0x39, 0x4a, 0x4c, 0x58, 0xcf, 0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, 0x51, 0xa3, 0x40, 0x8f, 0x92,
0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, 0xcd, 0x0c,
0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19,
0x73, 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14,
0xde, 0x5e, 0x0b, 0xdb, 0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2,
0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, 0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5,
0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, 0xba, 0x78, 0x25,
0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86,
0xc1, 0x1d, 0x9e, 0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e,
0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, 0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42,
0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16

```

```
[ ]
```

```

let aes_sbox_inv_table : list(bits(8)) = [ |
0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81,
0xf3, 0xd7, 0xfb, 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e,
0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb, 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23,
0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e, 0x08, 0x2e, 0xa1, 0x66,
0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25, 0x72,
0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65,
0xb6, 0x92, 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46,
0x57, 0xa7, 0x8d, 0x9d, 0x84, 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a,
0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06, 0xd0, 0x2c, 0x1e, 0x8f, 0xca,
0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b, 0x3a, 0x91,
0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6,

```

```

0x73, 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8,
0x1c, 0x75, 0xdf, 0x6e, 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f,
0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b, 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2,
0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4, 0x1f, 0xdd, 0xa8,
0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,
0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93,
0xc9, 0x9c, 0xef, 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb,
0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61, 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6,
0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d
|]

```

```

/* Lookup function - takes an index and a list, and retrieves the
 * x'th element of that list.
 */

```

```

val sbbox_lookup : (bits(8), list(bits(8))) -> bits(8)
function sbbox_lookup(x, table) = {
  match (x, table) {
    (0x00, t0::tn) => t0,
    (  y, t0::tn) => sbbox_lookup(x - 0x01, tn)
  }
}

```

```

/* Easy function to perform a forward AES SBox operation on 1 byte. */

```

```

val aes_sbbox_fwd : bits(8) -> bits(8)
function aes_sbbox_fwd(x) = sbbox_lookup(x, aes_sbbox_fwd_table)

```

```

/* Easy function to perform an inverse AES SBox operation on 1 byte. */

```

```

val aes_sbbox_inv : bits(8) -> bits(8)
function aes_sbbox_inv(x) = sbbox_lookup(x, aes_sbbox_inv_table)

```

```

/* AES SubWord function used in the key expansion
 * - Applies the forward sbbox to each byte in the input word.
 */

```

```

val aes_subword_fwd : bits(32) -> bits(32)
function aes_subword_fwd(x) = {
  aes_sbbox_fwd(x[31..24]) @
  aes_sbbox_fwd(x[23..16]) @
  aes_sbbox_fwd(x[15.. 8]) @
  aes_sbbox_fwd(x[ 7.. 0])
}

```

```

/* AES Inverse SubWord function.
 * - Applies the inverse sbbox to each byte in the input word.
 */

```

```

val aes_subword_inv : bits(32) -> bits(32)
function aes_subword_inv(x) = {
  aes_sbbox_inv(x[31..24]) @
  aes_sbbox_inv(x[23..16]) @
  aes_sbbox_inv(x[15.. 8]) @
  aes_sbbox_inv(x[ 7.. 0])
}

```

```

/* Easy function to perform an SM4 SBox operation on 1 byte. */
val sm4_sbox : bits(8) -> bits(8)
function sm4_sbox(x) = sbox_lookup(x, sm4_sbox_table)

val aes_get_column : (bits(128), nat) -> bits(32)
function aes_get_column(state,c) = (state >> (to_bits(7, 32 * c)))[31..0]

/* 64-bit to 64-bit function which applies the AES forward sbox to each byte
 * in a 64-bit word.
 */
val aes_apply_fwd_sbox_to_each_byte : bits(64) -> bits(64)
function aes_apply_fwd_sbox_to_each_byte(x) = {
  aes_sbox_fwd(x[63..56]) @
  aes_sbox_fwd(x[55..48]) @
  aes_sbox_fwd(x[47..40]) @
  aes_sbox_fwd(x[39..32]) @
  aes_sbox_fwd(x[31..24]) @
  aes_sbox_fwd(x[23..16]) @
  aes_sbox_fwd(x[15.. 8]) @
  aes_sbox_fwd(x[ 7.. 0])
}

/* 64-bit to 64-bit function which applies the AES inverse sbox to each byte
 * in a 64-bit word.
 */
val aes_apply_inv_sbox_to_each_byte : bits(64) -> bits(64)
function aes_apply_inv_sbox_to_each_byte(x) = {
  aes_sbox_inv(x[63..56]) @
  aes_sbox_inv(x[55..48]) @
  aes_sbox_inv(x[47..40]) @
  aes_sbox_inv(x[39..32]) @
  aes_sbox_inv(x[31..24]) @
  aes_sbox_inv(x[23..16]) @
  aes_sbox_inv(x[15.. 8]) @
  aes_sbox_inv(x[ 7.. 0])
}

/*
 * AES full-round transformation functions.
 */

val getbyte : (bits(64), int) -> bits(8)
function getbyte(x, i) = (x >> to_bits(6, i * 8))[7..0]

val aes_rv64_shiftrows_fwd : (bits(64), bits(64)) -> bits(64)
function aes_rv64_shiftrows_fwd(rs2, rs1) = {
  getbyte(rs1, 3) @
  getbyte(rs2, 6) @
  getbyte(rs2, 1) @
  getbyte(rs1, 4) @

```

```

    getbyte(rs2, 7) @
    getbyte(rs2, 2) @
    getbyte(rs1, 5) @
    getbyte(rs1, 0)
}

val aes_rv64_shiftrows_inv : (bits(64), bits(64)) -> bits(64)
function aes_rv64_shiftrows_inv(rs2, rs1) = {
    getbyte(rs2, 3) @
    getbyte(rs2, 6) @
    getbyte(rs1, 1) @
    getbyte(rs1, 4) @
    getbyte(rs1, 7) @
    getbyte(rs2, 2) @
    getbyte(rs2, 5) @
    getbyte(rs1, 0)
}

/* 128-bit to 128-bit implementation of the forward AES ShiftRows transform.
 * Byte 0 of state is input column 0, bits 7..0.
 * Byte 5 of state is input column 1, bits 15..8.
 */
val aes_shift_rows_fwd : bits(128) -> bits(128)
function aes_shift_rows_fwd(x) = {
    let ic3 : bits(32) = aes_get_column(x, 3);
    let ic2 : bits(32) = aes_get_column(x, 2);
    let ic1 : bits(32) = aes_get_column(x, 1);
    let ic0 : bits(32) = aes_get_column(x, 0);
    let oc0 : bits(32) = ic3[31..24] @ ic2[23..16] @ ic1[15.. 8] @ ic0[ 7.. 0];
    let oc1 : bits(32) = ic0[31..24] @ ic3[23..16] @ ic2[15.. 8] @ ic1[ 7.. 0];
    let oc2 : bits(32) = ic1[31..24] @ ic0[23..16] @ ic3[15.. 8] @ ic2[ 7.. 0];
    let oc3 : bits(32) = ic2[31..24] @ ic1[23..16] @ ic0[15.. 8] @ ic3[ 7.. 0];
    (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}

/* 128-bit to 128-bit implementation of the inverse AES ShiftRows transform.
 * Byte 0 of state is input column 0, bits 7..0.
 * Byte 5 of state is input column 1, bits 15..8.
 */
val aes_shift_rows_inv : bits(128) -> bits(128)
function aes_shift_rows_inv(x) = {
    let ic3 : bits(32) = aes_get_column(x, 3); /* In column 3 */
    let ic2 : bits(32) = aes_get_column(x, 2);
    let ic1 : bits(32) = aes_get_column(x, 1);
    let ic0 : bits(32) = aes_get_column(x, 0);
    let oc0 : bits(32) = ic1[31..24] @ ic2[23..16] @ ic3[15.. 8] @ ic0[ 7.. 0];
    let oc1 : bits(32) = ic2[31..24] @ ic3[23..16] @ ic0[15.. 8] @ ic1[ 7.. 0];
    let oc2 : bits(32) = ic3[31..24] @ ic0[23..16] @ ic1[15.. 8] @ ic2[ 7.. 0];
    let oc3 : bits(32) = ic0[31..24] @ ic1[23..16] @ ic2[15.. 8] @ ic3[ 7.. 0];
    (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}

```

```

/* Applies the forward sub-bytes step of AES to a 128-bit vector
 * representation of its state.
 */
val aes_subbytes_fwd : bits(128) -> bits(128)
function aes_subbytes_fwd(x) = {
  let oc0 : bits(32) = aes_subword_fwd(aes_get_column(x, 0));
  let oc1 : bits(32) = aes_subword_fwd(aes_get_column(x, 1));
  let oc2 : bits(32) = aes_subword_fwd(aes_get_column(x, 2));
  let oc3 : bits(32) = aes_subword_fwd(aes_get_column(x, 3));
  (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}

/* Applies the inverse sub-bytes step of AES to a 128-bit vector
 * representation of its state.
 */
val aes_subbytes_inv : bits(128) -> bits(128)
function aes_subbytes_inv(x) = {
  let oc0 : bits(32) = aes_subword_inv(aes_get_column(x, 0));
  let oc1 : bits(32) = aes_subword_inv(aes_get_column(x, 1));
  let oc2 : bits(32) = aes_subword_inv(aes_get_column(x, 2));
  let oc3 : bits(32) = aes_subword_inv(aes_get_column(x, 3));
  (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}

/* Applies the forward MixColumns step of AES to a 128-bit vector
 * representation of its state.
 */
val aes_mixcolumns_fwd : bits(128) -> bits(128)
function aes_mixcolumns_fwd(x) = {
  let oc0 : bits(32) = aes_mixcolumn_fwd(aes_get_column(x, 0));
  let oc1 : bits(32) = aes_mixcolumn_fwd(aes_get_column(x, 1));
  let oc2 : bits(32) = aes_mixcolumn_fwd(aes_get_column(x, 2));
  let oc3 : bits(32) = aes_mixcolumn_fwd(aes_get_column(x, 3));
  (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}

/* Applies the inverse MixColumns step of AES to a 128-bit vector
 * representation of its state.
 */
val aes_mixcolumns_inv : bits(128) -> bits(128)
function aes_mixcolumns_inv(x) = {
  let oc0 : bits(32) = aes_mixcolumn_inv(aes_get_column(x, 0));
  let oc1 : bits(32) = aes_mixcolumn_inv(aes_get_column(x, 1));
  let oc2 : bits(32) = aes_mixcolumn_inv(aes_get_column(x, 2));
  let oc3 : bits(32) = aes_mixcolumn_inv(aes_get_column(x, 3));
  (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}

/* Performs the word rotation for AES key schedule
 */

```

```

val aes_rotword : bits(32) -> bits(32)
function aes_rotword(x) = {
  let a0 : bits (8) = x[ 7.. 0];
  let a1 : bits (8) = x[15.. 8];
  let a2 : bits (8) = x[23..16];
  let a3 : bits (8) = x[31..24];
  (a0 @ a3 @ a2 @ a1) /* Return Value */
}

val brev : bits(SEW) -> bits(SEW)
function brev(x) = {
  let output : bits(SEW) = 0;
  foreach (i from 0 to SEW-8 by 8)
    output[i+7..i] = reverse_bits_in_byte(input[i+7..i]);
  output /* Return Value */
}

val reverse_bits_in_byte : bits(8) -> bits(8)
function reverse_bits_in_byte(x) = {
  let output : bits(8) = 0;
  foreach (i from 0 to 7)
    output[i] = x[7-i]);
  output /* Return Value */
}

val rev8 : bits(SEW) -> bits(SEW)
function rev8(x) = {      // endian swap
  let output : bits(SEW) = 0;
  let j = SEW - 1;
  foreach (k from 0 to (SEW - 8) by 8) {
    output[k..(k + 7)] = x[(j - 7)..j];
    j = j - 8;
  }
  output /* Return Value */
}
RETIRE_SUCCESS

val rol32 : bits(32) -> bits(32)
function ROL32(x,n) = (X << N) | (X >> (32 - N))

val sm4_subword : bits(32) -> bits(32)
function sm4_subword(x) = {
  sm4_sbox(x[31..24]) @
  sm4_sbox(x[23..16]) @
  sm4_sbox(x[15.. 8]) @
  sm4_sbox(x[ 7.. 0])
}

```