



RISC-V Cryptography Extensions
Volume III
Additional Vector Instructions

Version v0.0.4, 6 February 2024:

Table of Contents

Colophon	1
Acknowledgments	2
1. Introduction	3
2. Extensions Overview	4
2.1. Zvbc32e - Vector Carryless Multiplication	4
2.2. Zvkgs - Vector-Scalar GCM/GMAC	5
3. Instructions	6
3.1. <code>vclmul.[vv,vx]</code>	6
3.2. <code>vclmulh.[vv,vx]</code>	8
3.3. <code>vghsh.vs</code>	10
3.4. <code>vgmul.vs</code>	13
4. Bibliography	15
5. Encodings	16
Appendix A: Additional Vector Cryptographic Instructions	16

Colophon

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

This document is *Discussion Document*. Assume everything can change. This document is not complete yet and was created only for the purpose of conversation outside of the document. For more information, see [here](#).



Copyright and licensure:

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



Document Version Information:

HEAD @ 4ae2021a989ddc90fda47fd4db721c3d850ea322

See github.com/riscv/riscv-crypto/doc/vector-extra for more information.

Acknowledgments

Contributors to this specification (in alphabetical order) include:

Eric Biggers, Ken Dockser, 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 *additional vector* cryptography extensions for RISC-V. Those extensions extend the *vector* cryptography extensions for RISC-V, providing additional features not mandatory for a high performance implementation but which can help further improve the efficiency some algorithms (e.g. CRC, AES-GCM). All instructions proposed here are based on the Vector registers.

Chapter 2. Extensions Overview

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

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

As the instructions defined in this specification might be used to implement cryptographic primitives they may be implemented with data-independent execution latencies as defined in the [RISC-V Scalar Cryptography Extensions specification](#).

If **Zvkt** is implemented, all the instructions from **Zvbc32e** (`vclmul[h].[vv,vx]`) shall be executed with data-independent execution latency as

Whether **Zvkt** is implemented or not, all instructions from **Zvkgs** (`vgmul.vs, vghsh.vs`) shall 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. **Zvbc32e** - 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=32**. **Zvbc32e** can be supported when **ELEN >=32**.

Note

The extension **Zvbc32e** is independent from **Zvbc** which defines the same instructions for **SEW=64**. When **ELEN >=64** both extensions can be combined to have `vclmul.v[vx]` and `vclmulh.v[vx]` defined for both **SEW=32** and **SEW=64**.

Mnemonic	Instruction
<code>vclmul.[vv,vx]</code>	Vector Carry-less Multiply
<code>vclmulh.[vv,vx]</code>	[insns-vclmulh-32e]

2.2. Zvkgs - Vector-Scalar GCM/GMAC

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

Zvkgs depends on Zvkg. It extends the existing vghsh.vv and vgmul.vv instructions with new vector-scalar variants: vghsh.vs and vgmul.vs.

The instructions inherit the constraints defined in Zvkg:

- element group size (EGS) is 4
- data independent execution timing
- vl/vstart must be multiples of EGS=4multiple constraints

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 vl/EGS. vl 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.vs	Vector-Scalar GHASH Add-Multiply
32	128	vgmul.vs	Vector GHASH Multiply

Chapter 3. Instructions

3.1. vclmul.[vv,vx]

Synopsis

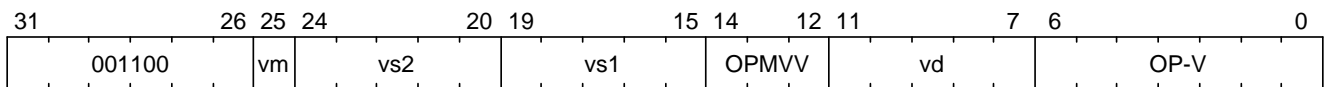
Vector Carry-less Multiply by vector or scalar - returning low half of product.

Mnemonic

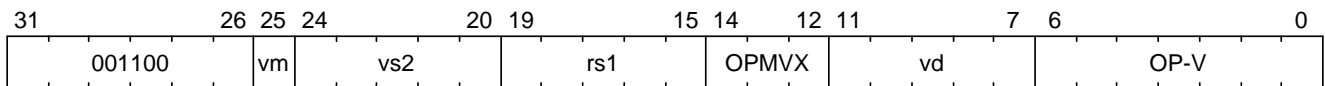
vclmul.vv vd, vs2, vs1, vm

vclmul.vx vd, vs2, rs1, vm

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Reserved Encodings

- SEW is any value other than 32 (*Zvbc32e* only)
- SEW is any value other than 64 (*Zvbc* only)
- SEW is any value other than 32 or 64 (*Zvbc* and *Zvbc32e*)

Arguments

Register	Direction	Definition
vs1/rs1	input	multiplier
vs2	input	multiplicand
vd	output	lower part of carry-less multiply



vclmul instruction was initially defined in *Zvbc* with only SEW=64-bit support, this page describes how the specification is extended in *Zvbc32e* to support SEW=32 bits.

Description

Produces the low half of $2*SEW$ -bit carry-less product.

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



The 32-bit carryless multiply instructions can be used for implementing GCM in the absence of the *zvkg* extension. In particular for implementation with *ELEN=32*

where `Zvkg` cannot be implemented. It can also be used to speed-up CRC evaluation.

Operation

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

[Zvbc32e](#)

3.2. vclmulh.[vv,vx]

Synopsis

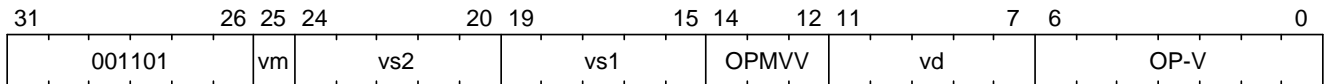
Vector Carry-less Multiply by vector or scalar - returning high half of product.

Mnemonic

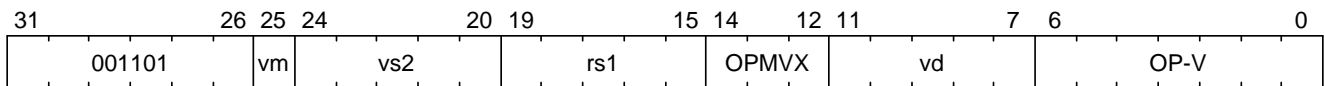
vclmulh.vv vd, vs2, vs1, vm

vclmulh.vx vd, vs2, rs1, vm

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Reserved Encodings

- SEW is any value other than 64 (Zvbc only)
- SEW is any value other than 32 (Zvbc32e only)
- SEW is any value other than 32 or 64 (Zvbc32e and Zvbc)

Arguments

Register	Direction	Definition
vs1/rs1	input	multiplier
vs2	input	multiplicand
vd	output	upper part of carry-less multiply



vclmulh instruction was initially defined in Zvbc, this page describes how the specification is extended in Zvbc32e to support SEW=32 bits.

Description

Produces the high half of $2 \cdot \text{SEW}$ -bit carry-less product.

Each SEW-bit element in the vs2 vector register is carry-less multiplied by either each SEW-bit element in vs1 (vector-vector), or the SEW-bit value from integer register rs1 (vector-scalar). The result is the most significant SEW 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 (SEW) = if suffix == "vv" then get_velem(vs1,i)
                      else zext_or_truncate_to_sew(X(vs1));
let op2 : bits (SEW) = get_velem(vs2, i);
let product : bits (SEW) = c1mulh(op1, op2, SEW);
set_velem(vd, i, product);
}
RETIRE_SUCCESS
}

function c1mulh(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

[Zvbc32e](#)

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.

Operation

```
function clause execute (VGHSHVS(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)

    // H is component to all element groups
    let helem = 0;
    let H = brev8(get_velem(vs2, EGW=128, helem)); // Hash subkey

    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 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

Zvkgs

3.4. vgmul.vs

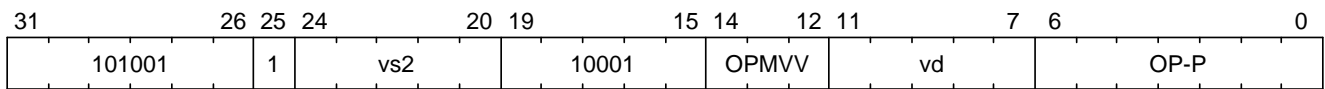
Synopsis

Vector-Scalar Multiply over GHASH Galois-Field

Mnemonic

vgmul.vs vd, vs2

Encoding (Vector-Scalar)



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.

The multipliers are read as 4-element groups from 'vd', the multiplicands subkeys are read from the scalar element group in **vs2**. The resulting products are written as 4-element groups into **vd**.

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 $x_0 + x_1u + x_2 u^2 + \dots + x_{127}u^{127}$. 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.



Similarly to how the instruction `vgmul.vv` is identical to `vghsh.vv` with the value of `vs1` register being 0, the instruction `vgmul.vs` is identical to `vghsh.vs` with the value of `vs1` being 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, suffix)) = {
  // 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)
    // H multiplicand is constant for all loop iterations
    let helem = 0;
    let H = brev8(get_velem(vs2,EGW=128, helem)); // Multiplicand

    foreach (i from eg_start to eg_len-1) {
      let Y = brev8(get_velem(vd,EGW=128,i)); // Multiplier
      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

[Zvkgs](#)

Chapter 4. Bibliography

Chapter 5. Encodings

Appendix A: Additional Vector Cryptographic Instructions

OP-P (0x77) Additional Vector Crypto instructions, including Zvkgs, except Zvbb and Zvbc The new/modified encoding are in bold and underlined.

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	V	<u>vghsh.vs</u>		100011			
100100				100100				100100			
100101				100101				100101			
100110				100110				100110			
100111				100111				100111			
101000				101000	V	VAES.vv		101000			
101001				101001	V	<u>VAES.vs</u>		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 1. VAES.vv and VAES.vs encoding space

vs1	
00000	vaesdm
00001	vaesdf
00010	vaesem
00011	vaesef
00111	vaesz
10000	vsm4r
10001	<i>vgmul</i>