RISC-V Cryptography Extensions Volume III

Additional Vector Instructions

Version v0.0.5, March 7th 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. vclmul.[vv,vx] . . . 6

  3.2. vclmulh.[vv,vx] . . . 8

  3.3. vghsh.vs . . . 10

  3.4. vgmul.vs . . . 13

4. Bibliography . . . 15

5. Encodings . . . 16

  Appendix A: Vector Carryless Multiply Instructions . . . 16

  Appendix B: Additional Vector Cryptographic Instructions . . . 19

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.

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, Liana Koleva, 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. Those extensions aim at either enabling some use cases (e.g. carry-less multiply on 32-bit vector implementations) or enabling more efficient implementations of 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. In particular `Zvbc32e` allows `Zve32*` implementations to support vector carry-less multiplication.

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.

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

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=4

All of these instructions work on 128-bit element groups comprised of four 32-bit elements, in element group parlance EGS=4, EGW=128 and the instructions are only defined for SEW=32.

To help avoid side-channel timing attacks, these instructions shall always 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)

31 — 26	25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
001100	vm	vs2	vs1	OPMVV	vd	OP-V

Encoding (Vector-Scalar)

31 — 26	25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
001100	vm	vs2	rs1	OPMVX	vd	OP-V

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)

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 (`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*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) = 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

Zvbc32e, Zvbc

3.3. vghsh.vs

Synopsis

Vector-Scalar Add-Multiply over GHASH Galois-Field

Mnemonic

vghsh.vs vd, vs2, vs1

Encoding (Vector-Scalar)

31 26	25	24 20	19 15	14 12	11 7	6 0
101100	1	vs2	vs1	OPMVV	vd	OP-P

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.

The previous partial hashes are read as 4-element groups from `vd`, the cipher texts are read as 4-element groups from `vs1` and the hash subkeys are read from the scalar element group in `vs2`. The resulting partial hashes are writen as 4-element groups into `vd`.

This instruction treats all of the input and output element groups as 128-bit polynomials and performs operations over 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 $GF(2^{128})$) this sum by the Hash Subkey (H).

The multiplication over $GF(2^{128})$ 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 $x_0x_1x_2 \ldots x_{127}$ for a polynomial $x_0 + x_1u + x_2 u^2 + \ldots + x_{127}u^{127}$. 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.

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 common 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

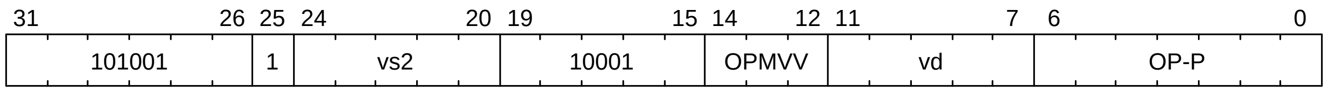
Synopsis

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 GF[2]. It produces the product over GF(2^{128}) of the two 128-bit inputs.

The multiplication over 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 $x_0x_1x_2...x_{127}$ for a polynomial $x_0 + x_1u + x_2 u^2 + ... + x_{127}u^{127}$. 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.

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 common 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: Vector Carryless Multiply Instructions

OP-V (0x57) **Zvbc32e** 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	vrsub	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			
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

funct6					funct6				funct6			
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								
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	vfwsub
110011					110011	V	X	vwsub	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	vwsubu.w	110110	V	F	vfwsub.w
110111					110111	V	X	vwsub.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 1. 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 B: Additional Vector Cryptographic Instructions

OP-P (0x77) Vector Crypto instructions, including `Zvkgs`, except `Zvbb` and `Zvbc`. The new/modified encodings are 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**			
100000					100000	V		vsm3me	100000			
100001					100001	V		vsm4k.vi	100001			
100010					100010	V		vaesfk1.vi	100010			
100011					100011	V		***vghsh.vs***	100011			
100100					100100				100100			
100101					100101				100101			
100110					100110				100110			

funct6					funct6				funct6			
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***