Chapter 36. RISC-V Cryptography Extensions Volume III: Additional Vector Instructions

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.

36.1. 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.*

36.1.1. `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`.

This extension covers two gaps of `Zvbc`:

- allowing vector implementations with smaller `ELEN=32` (e.g. implementations selecting `Zve32*`) to implement some support for vector carry-less multiplication (this is not allowed by `Zvbc` which requires `ELEN >= 64`)
- for implementations which have `ELEN >= 64`: allowing more efficient implementations of algorithms relying on 32-bit carry less multiplications. The list of such algorithms includes the CLM-based folding algorithm used to compute the widespread 32-bit CRCs (e.g. ethernet CRC). This technique can already be implemented with `Zvbc` but only half the 64-bit multiplication is

exploited. Selecting `Zvbc32e` only allows implementations to save area while providing identical performance on those algorithms.

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`.

Note

> The extra cost of supporting `Zvbc32e` on top of `Zvbc` should be minimal, as the hardware required to implement the instructions in `Zvbc32e` is a subset of the hardware required to implement `Zvbc’s instructions.

Mnemonic	Instruction
`vclmul.[vv,vx]`	Vector Carry-less Multiply
`vclmulh.[vv,vx]`	[insns-vclmulh-32e]

36.1.2. `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`.

One of the key use cases for the vector instructions `vghsh.vv` and `vgmul.vv` defined in Zvkg is to speed-up GCM cipher mode for a single stream by computing the GHASH algorithm for multiple blocks of the same message in parallel. The parallel processing accumulates and multiplies multiple blocks of the message by the same power of H (encryption of `0` by the cipher key). The power being equal to the number of blocks processed in parallel. The processing completes by reducing the parallel accumulators into a single output tag. With `Zvkg` only, a full vector register was required to old the multiple copies of the power of H. `Zvkgs` reduces the size of the vector register group needed for powers of H: it just needs to contain a 128-bit wide element group, freeing some vector registers. This exploits the same scalar element group broadcast mechanism used in other instructions defined in the vector crypto extensions (e.g. `vaesem.vs` from Zvkned).

SEW	EGW	Mnemonic	Instruction
32	128	`vghsh.vs`	Vector-Scalar GHASH Add-Multiply
32	128	`vgmul.vs`	[insns-vgmul-vs]

36.2. Instructions

36.2.1. 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, Zvbc

36.3. 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

36.4. 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
100011	1	vs2	vs1	OPMVV	vd	OP-VE

Reserved Encodings

- `SEW` is any value other than 32
- the `vd` register group overlaps the `vs2` scalar element group

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 $x_0x_1x_2...x_{127}$ 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.

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 Hinit = 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
    // Since H is destroyed by the inner loop it must be reset
    // on every element-group iteration (even if loop independent)
    let H = Hinit;

    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

36.5. vgmul.vs

Synopsis

Vector-Scalar Multiply over GHASH Galois-Field

Mnemonic

vgmul.vs vd, vs2

Encoding (Vector-Scalar)

31 26	25	24 20	19 15	14 12	11 7	6 0
101001	1	vs2	10001	OPMVV	vd	OP-VE

Reserved Encodings

- `SEW` is any value other than 32
- the `vd` register group overlaps the `vs2` scalar element group

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 Hinit = 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;
    // Since H is destroyed by the inner loop it must be reset
    // on every element-group iteration (even if loop independent)
    let H = Hinit;

    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

36.6. Encodings

Appendix A: Additional Vector Cryptographic Instructions

OP-VE (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			
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			

funct6					funct6				funct6			
101100					101100	V		vghsh	101100			
101101					101101	V		vsha2ms	101101			
101110					101110	V		vsha2ch	101110			
101111					101111	V		vsha2cl	101111			

Table 75. VAES.vv and VAES.vs encoding space

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