RISC-V Vector Extension for Wireless (Zvw)

Authors: Li Gaoshan

Version v0.0.0, 2024-11-14: Draft

Table of Contents

List of tables 1
List of listings 2
Copyright and license information 3
Contributors 4
1. Introduction 5
2. Basic operations definition 6
3. Registers 7
4. Vector Load and Store 8
4.1. vlfcA2B(vector load and format convert from format A to format B) 8
4.2. vsfcB2A(vector store and format convert from format B to format A) 8
5. Complex number format convert 10
5.1. vcpack(vector complex packing instruction) 10
5.2. vcunpackr(vector complex real unpacking instruction) 10
5.3. vcunpacki(vector complex image unpacking instruction) 10
6. Fix point dynamic scaling operations 12
6.1. vdsmul(vector dynamic scaling multiply) 12
6.2. vdsmacini(vector dynamic scaling MAC initialization) 12
6.3. vdsmac(vector dynamic scaling MAC) 13
6.4. vdsmaco(vector dynamic scaling MAC with final result output) 13
6.5. vlsb(vector leading sign bits instruction) 14
6.6. vconj(vector complex conjugate) 15
6.7. vdscmul (vector dynamic scaling complex multiply) 15
6.8. vdscmulj (vector dynamic scaling complex conjugate multiply) 16
6.9. vdscredsum(vector dynamic scaling complex reduced sum) 16
6.10. vdscmac(vector dynamic scaling complex MAC) 17
6.11. vdscmacj(vector dynamic scaling complex conjugate MAC) 18
6.12. vdscmaco(vector dynamic scaling complex MAC final result output) 18
6.13. vdscmacjo(vector dynamic scaling complex conjugate MAC with output) 19
6.14. vdscmacor(vector dynamic scaling complex MAC with widen output of real part) 20
6.15. vdscmacoi(vector dynamic scaling complex MAC with widen output of image part) 21
6.16. vdscmacjor(vector dynamic scaling complex conjugate MAC with widen output of real part) 22
6.17. vdscmacjoi(vector dynamic scaling complex conjugate MAC with widen output of image part) 23
6.18. vdscmulr(vector dynamic scaling complex multiply real part) 23
6.19. vdscmuli(vector dynamic scaling complex multiply image part) 24
6.20. vdscmuljr(vector dynamic scaling complex conjugate multiply real part) 25
6.21. vdscmulji(vector dynamic scaling complex conjugate multiply image part) 25
6.22. vdsredsum(vector dynamic scaling reduced sum) 26
6.23. vdsredsumn(vector dynamic scaling reduced sum for each n element) 26
6.24. vredmaxi(vector reduced max with index) 27
6.25. vredmini(vector reduced min with index) 27
7. Sequence generation operations 29
7.1. vpharot(vector phase rotate) 29
8. Inter-element operation instructions 30

8.1. vperm(vector permutation) 30
8.2. vfsl(vector funnel shift left) 30
8.3. vfsr(vector funnel shift right) 31
9. Fast non-linear operations 32
9.1. Introduction 32
9.2. Vector nonlinear parameter register 32
9.3. Instructions 32
9.3.1. vlnlp(vector load nonlinear parameter) 32
9.3.2. vnle(vector non-linear exponent calculation instruction) 33
9.3.3. vnlm(vector non-linear mantissa calculation instruction) 33
10. Apendix A 35
10.1. RVV & Zvw Encoding space 35
10.2. Zvw Instruction encoding 38
11. Fast non-linear instruction implemenation and paramters 43
Bibliography 44

List of tables

Table 1. VCSR bit fields table
Table 2. VNLCR bit fields table
Table 3. RVV overall table
Table 4. RVV Func3 encoding table
Table 5. Zvw Func6 encoding table part 1
Table 6. Zvw Func6 encoding table part 2
Table 7. Zvw Func6 encoding table part 3
Table 8. Zvw Func6 encoding table part 4
Table 9. Zvw encoding table
Table 10. Zvw load & store instruction encoding table

List of listings

This document is in the Development state

Expect potential changes. This draft specification is likely to evolve before it is accepted as a standard. Implementations based on this draft may not conform to the future standard.

Copyright and license information

Contributors

This RISC-V specification has been contributed to directly or indirectly by:

- **Li Gaoshan** <ligaoshan@cmiot.chinamobile.com>*
- Liang Shengdun <liangshengdun@cmiot.chinamobile.com>
- Li Ri <liri@cmiot.chinamobile.com>
- Cai yuehong <caiyuehong@cmiot.chinamobile.com>

Chapter 1. Introduction

This document outlines a specialized vector extension instruction set based on the RISC-V RVV 1.0 specification, tailored for wireless digital signal processing. This extension supplements the RVV instruction set and primarily addresses the demands of wireless digital signal processing.

Key additions include:

- Definitions for vector state register extension fields
- Complex dynamic scaling operation instructions
- Complex compressed format conversion instructions
- Vector register element inter-operation instructions
- Fixed-point dynamic scaling operation instructions
- Dynamic scaling multiply-accumulate operation instructions
- Unified fast nonlinear operation instructions

Currently, RISC-V Vector Working Group developed the RVV 1.0 vector extension, widely applied in AI and image processing. However, these basic instructions do not fully satisfy the requirements of wireless broadband signal processing, which involves handling complex numbers in 4G and 5G communications and demands high-precision fixed-point dynamic scaling. Therefore, additional and more effective vector extension instructions are necessary for wireless communication, covering a significant segment of vector signal processing needs, especially for 4G and 5G technologies.

The wireless signal processing extension instruction built on the RVV extension can accelerate advancements in 4G and 5G signal processing and facilitate the adoption of RISC-V in the wireless signal processing domain. The instruction set extends the RVV 1.0 specification to support high-bandwidth wireless communication signal processing. Termed the vector wireless broadband extension instruction set and represented by 'Zvw', it integrates into the basic 'v' extension. The Zvw extension adapts RVV instructions to better suit the high bandwidth, high-order modulation, and large throughput demands of wireless signal processing, particularly in 4G and 5G communications.

Currently, Zvw supports only RV32, with SEW variations limited to 8/16/32, where SEW=32 supports SC16. In SC16, real and imaginary parts are stored in adjacent vector register units (real in lower bits, imaginary in upper bits). For SC32, real and imaginary parts are stored in indexed units of different vector registers. The operational section addresses procedures on each vector processor element, adhering to LMUL and VL standards from the RVV specification. Operand objects are governed by SEW, execution on elements is controlled by vm, and default values for unprocessed elements by RVV-defined mask agnostic and tail agnostic rules.

Chapter 2. Basic operations definition

1. **Basic Operations**
   - int8: 8 bit signed char
   - int16: 16 bit signed half word
   - int32: 32 bit word
   - sc16: signed complex with 16bits image and 16bits real part, real part in low 16bit.
   - get_e: Get the exponent part in the complex compressed format
   - get_i: Get the imaginary part in the complex compressed format
   - get_r: Get the real part in the complex compressed format
   - zp16: Add 0 to 16 bits at the end
   - zp32: Add 0 to 32 bits at the end
   - sx16: sign extension to 16 bits
   - sx32: sign extension to 32 bits
   - min: Find the minimum
   - max: Find the maximum
   - max_i: Find the element position in the vector register where the maximum value is located
   - lsb: The number of bits of the high-order sign bit minus 1
   - cfp_pack: Put e, i and r into the compressed complex format according to the complex compressed format
   - vcpack: extract the high 16 bits of 32-bit real numbers and 32-bit imaginary numbers to form a complex number in SC16 format
   - sum: Sum
   - conj: Convert to the conjugate complex number of a negative number
   - alg_round_shift_right: Arithmetic right shift with vxrm mode rounding
   - alg_shift_left: Arithmetic left shift
   - shift_left: Logical left shift
   - clip_low_SEW: Clip the lowest SEW width bit number
   - round_clip_high_SEW: Saturate clip the high SEW width bit number with vxrm mode rounding
   - MIN_SEW: The minimum value of the signed number corresponding to the element of SEW bit width
   - MAX_SEW: The maximum value of the signed number corresponding to the element of SEW bit width
   - ACCREG[i]: The i-th accumulation register inside the MAC ALU, 80 bits, a total of 32 elements
   - MULSFTREG[i]: The i-th multiply shift register inside the MAC ALU, 5bits/element, 8*32 elements in total

Chapter 3. Registers

1. **VCSR** (Vector Control and Status Register)

   Add new fields to the vector control and status register (vcsr) to control the fixed-point multiplication with shift, fixed-point accumulation with shift, and funnel shift related instructions.

   - *mulsft* (multiply shift) controls the right shift operation after the multiplier operation. The right shift operation is round shift, and the round mode is controlled by vxrm.
   - *accsft* (accumulated shift) controls the right shift operation of the accumulation operation. The shift operation is round shift, and the round mode is controlled by vxrm.
   - *fsft* configures the shift length of the funnel shift, so that the operation can realize flexible funnel shift function without increasing the source operand.

   Table 1. VCSR bit fields table

Bitfield	Name	Description
31:18	rsv	reserve
17:13	fsft	elements num. of funnel shift
12:8	accsft[12:8]	accumulation shift，0 ~ 31
7:3	mulsft[7:3]	multiply shift，0 ~ 31
2:1	vxrm[1:0]	rounding mode
0	vxsat	

2. **VNLCR** (Vector non-linear Control Register)

   This register is used to configure the function type and segmentation information of the nonlinear function. The register width is 32 bits, configured through the Zicsr extended instruction, address *0x00E*.

   - *FuncId* configures the type of the current nonlinear function;
   - *SegGruCnt* configures the number of segments of the nonlinear function. The maximum number of segments of the nonlinear function is 16.

   Table 2. VNLCR bit fields table

Bitfield	Name	Description
31:16	SegGruCnt	segments num. of non-linear function. 0：reciprocal; 1: square root; 2: reciprocal of square root; 3: log2; 4: arctan
15:0	funcId	non-linear function ID. 0: 4 segments; 1：8 segments; 2：12 segments; 3：16 segments

Chapter 4. Vector Load and Store

4.1. vlfcA2B(vector load and format convert from format A to format B)

1. **Grammar**

   vlfcb2h.v vd, rs1, vm
   vlfcb2w.v vd, rs1, vm
   vlfch2w.v vd, rs1, vm
   vlfcpa2c.v vd, rs1, vm
   vlfcpb2c.v vd, rs1, vm

2. **Purpose**

   Load elements from vector memory to vector registers, performing format conversion operations at the same time. When mask is 0 when vm is enabled, the address and register element index need to be increased at the same time.

3. **Description**

   rs1 stores the base address, and vd is the target register. The total number of elements does not exceed VLEN/SEW*LMUL. SEW is the target format width. The formats of A2B include:

   - ‘b2h’ is 8bit→16bit signed number conversion;
   - ‘b2w’ is 8bit→32bit signed number conversion;
   - ‘h2w’ is 16bit→32bit signed number conversion;
   - ‘pa2c’ is CFL(6/6/4)→SC16 format conversion；
   - ‘pb2c’ is CFL(7/7/2)→SC16 format conversion.

4. **Operation**

```
vd[i] = loadFormatConvert(void *rs1, A2B) {
  switch(A2B)
    case ‘b2h’
      return sx16(int8 rs1[i])
    case ‘b2w’
      return sx32(int8 rs1[i])
    case ‘h2w’
      return sx32(int16 rs1[i])
    case ‘pa2c’
      i = get_i(sc16 rs1[i])
      r = get_r(sc16 rs1[i])
      e = get_e(sc16 rs1[i])
      I = zp32( i )
      R = zp32( r )
      return vcpack(sra(I, e), sra(R, e))
    case 'pb2c'
      // support in future
}
```

4.2. vsfcB2A(vector store and format convert from format B to format A)

1. **Grammar**

   vsfch2b.v vs3, rs1, vm
   vsfcw2b.v vs3, rs1, vm
   vsfcw2h.v vs3, rs1, vm
   vsfcc2pa.v vs3, rs1, vm
   vsfcc2pb.v vs3, rs1, vm

2. **Purpose**

   Convert format of each element from the vector register and store it into vector memory. When mask is 0 with vm is enabled, the address of vector memroy and register element index need to increased at the same time.

3. **Description**

   rs1 stores the base address, and vs3 is the source register. The total number of elements does not exceed VLEN/SEW*LMUL. SEW is the target format width. The formats of B2A include:

   - ‘h2b’ is 16bit→8bit signed number conversion;
   - ‘w2b’ is 32bit→8bit signed number conversion;
   - ‘w2h’ is 32bit→16bit signed number conversion;
   - ‘c2pa’ is SC16→CFL(6/6/4) format conversion；
   - ‘c2pb’ is SC16→CFL(7/7/2) format conversion.

4. **Operation**

```
void storeFormatConvert(void *rs1, vs3[i], B2A) {
  switch(B2A)
    case ‘h2b’
      vmem(clip_low_8(int16 vs3[i]), (int8*)rs1[i])
    case ‘w2b’
      vmem(clip_low_8(int32 vs3[i]), (int8*)rs1[i])
    case ‘w2h’
      vmem(clip_low_16(int32 vs3[i]), (int16*)rs1[i])
    case ‘c2pa’
      I = vs3[i].i
      R = vs3[i].r
      num = max(abs(I), abs(R))
      exp = lsb(num)
      img = round_clip_high_6bit(I << exp)
      real = round_clip_high_6bit(R << exp)
      vmem(cfp_pack(exp, img, real), (int16*)rs1[i])
   case 'c2pb'
      // support in future
}
```

Chapter 5. Complex number format convert

5.1. vcpack(vector complex packing instruction)

1. **Grammar**

   vcpack.vv vd, vs2, vs1, vm

2. **Purpose**

   When SEW is 32bit, two 32-bit elements are saturated and truncated to the high 16 bits to form a complex number in SC16 format.

3. **Description**

   The elements of vs2 are used as real parts, and the elements of vs1 are used as imaginary parts. The elements in vs2 and vs1 are saturated and intercepted with the high 16 bits to form a complex number in SC16 format:

4. **Operation**

```
vd[i].r = round_clip_high_16(vs2[i]);
vd[i].i = round_clip_high_16(vs1[i]);
```

5.2. vcunpackr(vector complex real unpacking instruction)

1. **Grammar**

   vcunpackr.v vd, vs2, vm

2. **Purpose**

   When SEW is 32bit, the 16bit imaginary part of the SC16 complex number is converted into 32bit output.

3. **Description**

   The elements of vs2 are SC16 complex numbers. The 16-bit imaginary part is extracted and placed in the high 16 bits of vd, and the low 16 bits are padded with zeros.

4. **Operation**

```
vd[i] = ZP32(vs2[i].r);
```

5.3. vcunpacki(vector complex image unpacking instruction)

1. **Grammar**

   vcunpacki.v vd, vs2, vm

2. **Purpose**

   When SEW is 32bit, the 16bit imaginary part of the SC16 complex number is converted into 32bit output.

3. **Description**

   The elements of vs2 are SC16 complex numbers. The 16-bit imaginary part is extracted and placed in the high 16 bits of vd, and the low 16 bits are padded with zeros.

4. **Operation**

```
vd[i] = ZP32(vs2[i].i);
```

Chapter 6. Fix point dynamic scaling operations

6.1. vdsmul(vector dynamic scaling multiply)

1. **Grammar**

   vdsmul.vv vd, vs2, vs1, vm
   vdsmul.vs vd, vs2, vs1, vm

2. **Purpose**

   Two fixed-point numbers are multiplied, and the multiplied result is shifted according to the mulsft shift value for scaling, and the result bit width remains unchanged.

3. **Description**

   The signed fixed-point number of vs1 or vs1[0] and vs2 is multiplied. The intermediate result after the multiplication retains 2 times the bit width. After performing an arithmetic right shift according to the scaling requirements, the lower half bit width is retained and output to vd.

4. **Operation**

```
VV version
Tmp = vs1[i] * vs2[i]  // 2*SEW = SEW * SEW
Tmp = alg_round_shift_right(tmp, mulsft);
vd[i] = clip_low_SEW(tmp); // SEW = 2*SEW

VS version
Tmp = vs1[0] * vs2[i]  // 2*SEW = SEW * SEW
Tmp = alg_round_shift_right(tmp, mulsft);
vd[i] = clip_low_SEW(tmp); // SEW = 2*SEW
```

6.2. vdsmacini(vector dynamic scaling MAC initialization)

1. **Grammar**

   vdsmacini.v vs1, vm
   vdsmacini.s rs1， vm
   vdsmacini.i uimm, vm

2. **Purpose**

   Initializes the shift register value for each element of the multiply accumulator.

3. **Description**

   vs1, rs1 or uimm are stored as the shift value before the accumulator accumulates, and the multiplication shift value in the multiplication and accumulation is configured through this instruction.

4. **Operation**

```
V version
MULSFT[i] = vs[i];

S version
MULSFT[i] = rs1;

I version
MULSFT[i] = uimm;
```

6.3. vdsmac(vector dynamic scaling MAC)

1. **Grammar**

   vdsmac.vv vs2, vs1, vm
   vdsmac.vs vs2, vs1, vm

2. **Purpose**

   Calculate the multiplication and accumulation of two fixed-point numbers. The intermediate result of the multiplication and the final accumulation result can be dynamically shifted.

3. **Description**

   The signed fixed-point number of vs1 or vs1[0] and vs2 is multiplied. The intermediate result after the multiplication retains 2 times the bit width, and then performs arithmetic right shifting, and the result is accumulated with the value in the accumulation register.

4. **Operation**

```
VV version
Tmp = vs1[i] * vs2[i]  // 2*SEW = SEW * SEW
Tmp = alg_round_shift_right(tmp, MULSFT[i]);
ACCREG[i] += tmp;

VS version
Tmp = vs1[0] * vs2[i]  // 2*SEW = SEW * SEW
Tmp = alg_round_shift_right(tmp, MULSFT[i]);
ACCREG[i] += tmp;
```

6.4. vdsmaco(vector dynamic scaling MAC with final result output)

1. **Grammar**

   vdsmaco.vv vd, vs2, vs1, vm
   vdsmaco.vs vd, vs2, vs1,vm

2. **Purpose**

   The two fixed-point numbers perform a dynamic scaling accumulation operation, and then the accumulation result is output according to the final fixed-point shift configuration.

3. **Description**

   The signed fixed-point number of vs1 or vs1[0] and vs2 is multiplied. The intermediate result after the multiplication retains 2 times the bit width, and then performs arithmetic shifting. The result is accumulated with the value in the accumulation register. After the accumulation, the value in the accumulation register is The value is arithmetic shifted and then truncated and output to vd, and the accumulation register is cleared. This instruction is not controlled by LMUL.

4. **Operation**

```
VV version
Tmp = vs1[i] * vs2[i]  // 2*SEW = SEW * SEW
Tmp = alg_round_shift_right(tmp, MULSFT[i]);
ACCREG[i] += tmp;
vd[i] = clip_low_SEW(alg_round_shift_right(ACCREG[i], accsft)) // SEW
ACCREG[i] = 0；

VS version
Tmp = vs1[0] * vs2[i]  // 2*SEW = SEW * SEW
Tmp = alg_round_shift_right(tmp, MULSFT[i]);
ACCREG[i] += tmp;
vd[i] = clip_low_SEW(alg_round_shift_right(ACCREG[i], accsft)) // SEW
ACCREG[i] = 0；
```

6.5. vlsb(vector leading sign bits instruction)

1. **Grammar**

   vlsb.v vd, vs2, vm

2. **Purpose**

   Calculate the value of the number of leading sign bits minus 1, which is used for normalization operations of fixed-point numbers.

3. **Description**

   vs2 stores fixed-point values, calculates the number of leading sign bits of each fixed-point value minus one, and outputs this value to vd.

4. **Operation**

```
vd[i] = lsb(vs2[i]);
```

== Complex dynamic scaling operations

6.6. vconj(vector complex conjugate)

1. **Grammar**

   vconj.v vd, vs2, vm

2. **Purpose**

   Perform a conjugate transformation on a set of complex numbers.

3. **Description**

   vs2 stores a set of signed fixed-point complex numbers, performs conjugate transformation on them, and stores the result in vd.

4. **Operation**

```
vd[i] = conj(vs2[i])
```

6.7. vdscmul (vector dynamic scaling complex multiply)

1. **Grammar**

   vdscmul.vv vd, vs2, vs1, vm
   vdscmul.vs vd, vs2, vs1, vm

2. **Purpose**

   Bit-width-preserving dynamically scaled multiplication of two fixed-point complex numbers.

3. **Description**

   The signed fixed-point complex numbers in vs1 or rs1 and vs2 are multiplied. The intermediate result after the multiplication retains 2 times the bit width. After performing an arithmetic right shift according to the scaling requirements, the lower half bit width is retained and output to vd.

4. **Operation**

```
VV version
Tmp.r = vs1[i].r * vs2[i].r - vs1[i].i * vs2[i].i  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
Tmp.i = vs1[i].r * vs2[i].i + vs1[i].i * vs2[i].r  // SEW = SEW/2 * SEW/2 +
SEW/2 * SEW/2
vd[i].r = clip_low_SEW(alg_round_shift_right(Tmp.r, mulsft)) //SEW/2=SEW
vd[i].i = clip_low_SEW(alg_round_shift_right(Tmp.i, mulsft)) //SEW/2=SEW

VS version
Tmp.r = vs1[0].r * vs2[i].r - vs1[0].i * vs2[i].i  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
Tmp.i = vs1[0].r * vs2[i].i + vs1[0].i * vs2[i].r  // SEW = SEW/2 * SEW/2 +
SEW/2 * SEW/2
vd[i].r = clip_low_SEW(alg_round_shift_right(Tmp.r, mulsft)) //SEW/2=SEW
```

```
vd[i].i = clip_low_SEW(alg_round_shift_right(Tmp.i, mulsft)) //SEW/2=SEW
```

6.8. vdscmulj (vector dynamic scaling complex conjugate multiply)

1. **Grammar**

   vdscmulj.vv vd, vs2, vs1, vm
   vdscmulj.vs vd, vs2, vs1, vm

2. **Purpose**

   Bitwidth-constant complex dynamically scaled conjugate multiplication.

3. **Description**

   vs2 and the conjugate fixed-point complex numbers of vs1 are multiplied. The intermediate result retains 2 times the bit width. After performing an arithmetic right shift according to the scaling requirements, the lower half bit width is retained and output to vd.

4. **Operation**

```
VV version
Tmp.r = vs1[i].r * vs2[i].r + vs1[i].i * vs2[i].i  // SEW = SEW/2 * SEW/2 +
SEW/2 * SEW/2
Tmp.i = vs1[i].r * vs2[i].i - vs1[i].i * vs2[i].r  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
vd[i].r = clip_low_SEW(alg_round_shift_right(Tmp.r, mulsft)) //SEW/2=SEW
vd[i].i = clip_low_SEW(alg_round_shift_right(Tmp.i, mulsft)) //SEW/2=SEW

VS version
Tmp.r = vs1[0].r * vs2[i].r + vs1[0].i * vs2[i].i  // SEW = SEW/2 * SEW/2 +
SEW/2 * SEW/2
Tmp.i = vs1[0].r * vs2[i].i - vs1[0].i * vs2[i].r  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
vd[i].r = clip_low_SEW(alg_round_shift_right(Tmp.r, mulsft)) //SEW/2 = SEW
vd[i].i = clip_low_SEW(alg_round_shift_right(Tmp.i, mulsft)) //SEW/2 = SEW
```

6.9. vdscredsum(vector dynamic scaling complex reduced sum)

1. **Grammar**

   vdscredsum.v vd, vs2, vm

2. **Purpose**

   The real and imaginary parts of each complex element in the vector register are accumulated and shifted to obtain an output with constant width.

3. **Description**

The real and imaginary parts of the complex elements in vs2 are accumulated separately and the accumulated value is shifted and the low SEW bit is intercepted and output to vd[0]. When SEW is configured as 32bit, SC16 complex addition is performed.

4. **Operation**

```
Tmp.r = 0;
Tmp.i = 0;
For each i in vs2
Tmp.r += v0.b[i] == 0 ? 0 : vs2[i].r
Tmp.i += v0.b[i] == 0 ? 0 : vs2[i].i
Vd[0].r = clip_low_SEW(alg_round_shift_right(Tmp.r, accsft)) // SEW
Vd[0].i = clip_low_SEW(alg_round_shift_right(Tmp.i, accsft)) // SEW
```

6.10. vdscmac(vector dynamic scaling complex MAC)

1. **Grammar**

vdscmac.vv vs2, vs1, vm
vdscmac.vs vs2, vs1, vm

2. **Purpose**

Fixed-point complex dynamic scaling multiply-accumulate.

3. **Description**

The signed fixed-point complex numbers of vs2 and vs1 and vs1[0] are multiplied. The intermediate result after the multiplication retains 2 times the bit width, and then performs arithmetic shifting, and the result is accumulated with the value in the accumulation register. The first operand supports coming from vector register vs1 or general register rs1. General register rs1 stores a complex element, which is multiplied by the complex element in vector register vs2 respectively. When SEW is 32bit, SC16 complex multiplication operation is performed.

4. **Operation**

```
VV version
Tmp.r = vs1[i].r * vs2[i].r - vs1[i].i * vs2[i].i  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
Tmp.r = alg_round_shift_right(Tmp.r, MULSFT[i]);
Tmp.i = vs1[i].r * vs2[i].i + vs1[i].i * vs2[i].r  // SEW = SEW/2 * SEW/2 +
SEW/2 * SEW/2
Tmp.i = alg_round_shift_right(Tmp.i, MULSFT[i]);
ACCREG[i].r += Tmp.r;
ACCREG[i].i += Tmp.i;

VS version
Tmp.r = vs1[0].r * vs2[i].r - vs1[0].i * vs2[i].i  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
Tmp.r = alg_round_shift_right(Tmp.r, MULSFT[i]);
Tmp.i = vs1[0].r * vs2[i].i + vs1[0].i * vs2[i].r  // SEW = SEW/2 * SEW/2 +
SEW/2 * SEW/2
Tmp.i = alg_round_shift_right(Tmp.i, MULSFT[i]);
ACCREG[i].r += Tmp.r;
ACCREG[i].i += Tmp.i;
```

6.11. vdscmacj(vector dynamic scaling complex conjugate MAC)

1. **Grammar**

   vdscmacj.vv vs2, vs1, vm
   vdscmacj.vs vs2, vs1, vm

2. **Purpose**

   Fixed-point complex dynamically scaled conjugate multiplication with constant bit width.

3. **Description**

   The conjugate multiplication of vs2 and vs1, vs1[0], the intermediate result after the multiplication retains 2 times the bit width, and then performs an arithmetic right shift, and the result is accumulated with the value in the accumulation register. When SEW is 32bit, SC16 complex multiplication operation is performed.

4. **Operation**

```
VV version
Tmp.r = vs1[i].r * vs2[i].r + vs1[i].i * vs2[i].i  // SEW = SEW/2 * SEW/2 +
SEW/2 * SEW/2
Tmp.r = alg_round_shift_right(Tmp.r, MULSFT[i]);
Tmp.i = vs1[i].r * vs2[i].i - vs1[i].i * vs2[i].r  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
Tmp.i = alg_round_shift_right(Tmp.i, MULSFT[i]);
ACCREG[i].r += Tmp.r;
ACCREG[i].i += Tmp.i;

VS version
Tmp.r = vs1[0].r * vs2[i].r + vs1[0].i * vs2[i].i  // SEW = SEW/2 * SEW/2 +
SEW/2 * SEW/2
Tmp.r = alg_round_shift_right(Tmp.r, MULSFT[i]);
Tmp.i = vs1[0].r * vs2[i].i - vs1[0].i * vs2[i].r  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
Tmp.i = alg_round_shift_right(Tmp.i, MULSFT[i]);
ACCREG[i].r += Tmp.r;
ACCREG[i].i += Tmp.i;
```

6.12. vdscmaco(vector dynamic scaling complex MAC final result output)

1. **Grammar**

vdscmaco.vv vd, vs2, vs1, vm
vdscmaco.vs vd, vs2, vs1, vm

2. **Purpose**

   Dynamic scaling of fixed-point complex numbers with constant bit width multiply and accumulate and output the final accumulation result.

3. **Description**

   The complex numbers of vs2 and vs1 and vs1[0] are multiplied. The intermediate result after the multiplication retains 2 times the bit width, and then performs an arithmetic right shift, and the result is accumulated with the value in the accumulation register. Arithmetically shift the real part and imaginary part of the value in the accumulation register, truncate and output them to vd, and clear the accumulation register. When SEW is 32bit, the complex number of SC16 is output.

4. **Operation**

```
VV version
Tmp.r = vs1[i].r * vs2[i].r - vs1[i].i * vs2[i].i  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
Tmp.r = alg_round_shift_right(Tmp.r, MULSFT[i]);
Tmp.i = vs1[i].r * vs2[i].i + vs1[i].i * vs2[i].r  // SEW = SEW/2 * SEW/2 +
SEW/2 * SEW/2
Tmp.i = alg_round_shift_right(Tmp.i, MULSFT[i]);
ACCREG[i].r += Tmp.r;
ACCREG[i].i += Tmp.i;
vd[i].r = clip_low_SEW(alg_round_shift_right(ACCREG[i].r, accsft)) // SEW
vd[i].i = clip_low_SEW(alg_round_shift_right(ACCREG[i].i, accsft)) // SEW
ACCREG[i] = 0;

VS version
Tmp.r = vs1[0].r * vs2[i].r - vs1[0].i * vs2[i].i  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
Tmp.r = alg_round_shift_right(Tmp.r, MULSFT[i]);
Tmp.i = vs1[0].r * vs2[i].i + vs1[0].i * vs2[i].r  // SEW = SEW/2 * SEW/2 +
SEW/2 * SEW/2
Tmp.i = alg_round_shift_right(Tmp.i, MULSFT[i]);
ACCREG[i].r += Tmp.r;
ACCREG[i].i += Tmp.i;
vd[i].r = clip_low_SEW(alg_round_shift_right(ACCREG[i].r, accsft)) // SEW
vd[i].i = clip_low_SEW(alg_round_shift_right(ACCREG[i].i, accsft)) // SEW
ACCREG[i] = 0;
```

6.13. vdscmacjo(vector dynamic scaling complex conjugate MAC with output)

1. **Grammar**

   vdscmacjo.vv vd, vs2, vs1, vm
   vdscmacjo.vs vd, vs2, vs1, vm

2. **Purpose**

   Fixed-point complex number dynamic scaling conjugate multiply and accumulate with constant bit width and output the final accumulation result.

3. **Description**

   The complex conjugate of vs1/vs1[0] and vs2 is multiplied. The intermediate result after the multiplication retains 2 times the bit width, and then performs an arithmetic right shift, and the result is accumulated with the value in the accumulation register. Arithmetically shift the real part and imaginary part of the value in the accumulation register, truncate and output them to vd, and clear the accumulation register. When SEW is 32bit, the complex number of SC16 is output.

4. **Operation**

```
VV version
Tmp.r = vs1[i].r * vs2[i].r + vs1[i].i * vs2[i].i  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
Tmp.r = alg_round_shift_right(Tmp.r, MULSFT[i]);
Tmp.i = vs1[i].r * vs2[i].i - vs1[i].i * vs2[i].r  // SEW = SEW/2 * SEW/2 +
SEW/2 * SEW/2
Tmp.i = alg_round_shift_right(Tmp.i, MULSFT[i]);
ACCREG[i].r += Tmp.r;
ACCREG[i].i += Tmp.i;
vd[i].r = clip_low_SEW(alg_round_shift_right(ACCREG[i].r, accsft)) // SEW
vd[i].i = clip_low_SEW(alg_round_shift_right(ACCREG[i].i, accsft)) // SEW
ACCREG[i] = 0;

VS version
Tmp.r = vs1[0].r * vs2[i].r + vs1[0].i * vs2[i].i  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
Tmp.r = alg_round_shift_right(Tmp.r, MULSFT[i]);
Tmp.i = vs1[0].r * vs2[i].i - vs1[0].i * vs2[i].r  // SEW = SEW/2 * SEW/2 +
SEW/2 * SEW/2
Tmp.i = alg_round_shift_right(Tmp.i, MULSFT[i]);
ACCREG[i].r += Tmp.r;
ACCREG[i].i += Tmp.i;
vd[i].r = clip_low_SEW(alg_round_shift_right(ACCREG[i].r, accsft)) // SEW
vd[i].i = clip_low_SEW(alg_round_shift_right(ACCREG[i].i, accsft)) // SEW
ACCREG[i] = 0;
```

6.14. vdscmacor(vector dynamic scaling complex MAC with widen output of real part)

1. **Grammar**

   vdscmacor.vv vd, vs2, vs1, vm
   vdscmacor.vs vd, vs2, vs1, vm

2. **Purpose**

Dynamically scale complex multiplication and accumulation and output the real part with twice the bit width.

3. **Description**

   The complex numbers of vs2 and vs1 and vs1[0] are multiplied. The intermediate result after the multiplication retains 2 times the bit width, and then performs an arithmetic right shift, and the result is accumulated with the value in the accumulation register. Perform arithmetic shifts on the real part in the accumulation register, intercept the low bits of the real part, and output them to vd, and clear the real part to zero. When SEW is configured as 32bit, the real part of the output is 32bit. This command is invalid when SEW is configured as other values.

4. **Operation**

```
VV version
Tmp.r = vs1[i].r * vs2[i].r - vs1[i].i * vs2[i].i  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
Tmp.r = alg_round_shift_right(Tmp.r, MULSFT[i]);
ACCREG[i].r += Tmp.r;
vd[i] = clip_low_SEW(alg_round_shift_right(ACCREG[i].r, accsft)) // SEW
ACCREG[i].r = 0;

VS version
Tmp.r = vs1[0].r * vs2[i].r - vs1[0].i * vs2[i].i  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
Tmp.r = alg_round_shift_right(Tmp.r, MULSFT[i]);
ACCREG[i].r += Tmp.r;
vd[i] = clip_low_SEW(alg_round_shift_right(ACCREG[i].r, accsft)) // SEW
ACCREG[i].r = 0;
```

6.15. vdscmacoi(vector dynamic scaling complex MAC with widen output of image part)

1. **Grammar**

   vdscmacoi.vv vd, vs2, vs1, vm
   vdscmacoi.vs vd, vs2, vs1, vm

2. **Purpose**

   Dynamically scale complex multiplication and accumulation and output the imaginary part with twice the bit width.

3. **Description**

   Multiply the complex numbers of vs2 and vs1 and vs1[0], perform an arithmetic shift on the imaginary part in the accumulation register, intercept the low bits of the imaginary part and output it to vd, and clear the imaginary part to zero. When SEW is configured for 32bit, the imaginary part of the output is 32bit. This command is invalid when SEW is configured to other values.

4. **Operation**

```
VV version
Tmp.i = vs1[i].r * vs2[i].i + vs1[i].i * vs2[i].r  // SEW = SEW/2 * SEW/2 +
SEW/2 * SEW/2
Tmp.i = alg_round_shift_right(Tmp.i, MULSFT[i]);
ACCREG[i].i += Tmp.i;
vd[i] = clip_low_SEW(alg_round_shift_right(ACCREG[i].i, accsft)) // SEW
ACCREG[i].i = 0；

VS version
Tmp.i = vs1[0].r * vs2[i].i + vs1[0].i * vs2[i].r  // SEW = SEW/2 * SEW/2 +
SEW/2 * SEW/2
Tmp.i = alg_round_shift_right(Tmp.i, MULSFT[i]);
ACCREG[i].i += Tmp.i;
vd[i] = clip_low_SEW(alg_round_shift_right(ACCREG[i].i, accsft)) // SEW
ACCREG[i].i = 0；
```

6.16. vdscmacjor(vector dynamic scaling complex conjugate MAC with widen output of real part)

1. **Grammar**

   vdscmacjor.vv vd, vs2, vs1, vm
   vdscmacjor.vs vd, vs2, vs1, vm

2. **Purpose**

   Dynamically scale complex conjugate multiply and accumulate and output the real part with twice the bit width.

3. **Description**

   The complex conjugate of vs1/vs1[0] and vs2 is multiplied. The intermediate result after the multiplication retains 2 times the bit width, and then performs an arithmetic right shift, and the result is accumulated with the value in the accumulation register. Perform arithmetic shifts on the real part in the accumulation register, intercept the low bits of the real part, and output them to vd, and clear the real part to zero. When SEW is configured as 32bit, the real part of the output is 32bit. This command is invalid when SEW is configured as other values.

4. **Operation**

```
VV version
Tmp.r = vs1[i].r * vs2[i].r + vs1[i].i * vs2[i].i  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
Tmp.r = alg_round_shift_right(Tmp.r, MULSFT[i]);
ACCREG[i].r += Tmp.r;
vd[i] = clip_low_SEW(alg_round_shift_right(ACCREG[i].r, accsft)) // SEW
ACCREG[i].r = 0；

VS version
Tmp.r = vs1[0].r * vs2[i].r + vs1[0].i * vs2[i].i  // SEW = SEW/2 * SEW/2 -
```

```
SEW/2 * SEW/2
Tmp.r = alg_round_shift_right(Tmp.r, MULSFT[i]);
ACCREG[i].r += Tmp.r;
vd[i] = clip_low_SEW(alg_round_shift_right(ACCREG[i].r, accsft)) // SEW
ACCREG[i].r = 0;
```

6.17. vdscmacjoi(vector dynamic scaling complex conjugate MAC with widen output of image part)

1. **Grammar**

   vdscmacjoi.vv vd, vs2, vs1, vm
   vdscmacjoi.vs vd, vs2, vs1, vm

2. **Purpose**

   Dynamically scale complex conjugate multiply and accumulate and output the imaginary part with twice the bit width.

3. **Description**

   Multiply the complex conjugate of vs1/vs1[0] and vs2, perform an arithmetic shift on the imaginary part in the accumulation register, intercept the low bits of the imaginary part and output it to vd, and clear the imaginary part to zero. When SEW is configured for 32bit, the imaginary part of the output is 32bit. This command is invalid when SEW is configured to other values.

4. **Operation**

```
VV version
Tmp.i = vs1[i].r * vs2[i].i - vs1[i].i * vs2[i].r  // SEW = SEW/2 * SEW/2 +
SEW/2 * SEW/2
Tmp.i = alg_round_shift_right(Tmp.i, MULSFT[i]);
ACCREG[i].i += Tmp.i;
vd[i] = clip_low_SEW(alg_round_shift_right(ACCREG[i].i, accsft)) // SEW
ACCREG[i].i = 0;

VS version
Tmp.i = vs1[0].r * vs2[i].i - vs1[0].i * vs2[i].r  // SEW = SEW/2 * SEW/2 +
SEW/2 * SEW/2
Tmp.i = alg_round_shift_right(Tmp.i, MULSFT[i]);
ACCREG[i].i += Tmp.i;
vd[i] = clip_low_SEW(alg_round_shift_right(ACCREG[i].i, accsft)) // SEW
ACCREG[i].i = 0;
```

6.18. vdscmulr(vector dynamic scaling complex multiply real part)

1. **Grammar**

   vdscmulr.vv vd, vs2, vs1, vm

vdscmulr.vs vd, vs2, vs1, vm

2. **Purpose**

   Fixed-point complex dynamically scaled multiplication, outputting twice the bit-width real part.

3. **Description**

   vs2 multiply the conjugate of vs1/vs1[0]. The real part of the multiplied result retains 2 times the bit width. It performs an arithmetic right shift according to the scaling requirements and then outputs it to vd. When SEW is configured as 32bit, the SC16 complex multiplication operation is performed and the 32bit real part is output.

4. **Operation**

```
VV version
Tmp.r = vs1[i].r * vs2[i].r - vs1[i].i * vs2[i].i  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
vd[i] = clip_low_SEW(alg_round_shift_right(Tmp.r, mulsft)) //SEW

VS version
Tmp.r = vs1[0].r * vs2[i].r - vs1[0].i * vs2[i].i  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
vd[i] = clip_low_SEW(alg_round_shift_right(Tmp.r, mulsft)) //SEW
```

6.19. vdscmuli(vector dynamic scaling complex multiply image part)

1. **Grammar**

   vdscmuli.vv vd, vs2, vs1, vm
   vdscmuli.vs vd, vs2, vs1, vm

2. **Purpose**

   Fixed-point complex number dynamic scaling multiplication, output twice the bit width imaginary part.

3. **Description**

   vs2 multiply the conjugate of vs1/vs1[0]. The imaginary part of the multiplied result retains 2 times the bit width. It performs an arithmetic right shift according to the scaling requirements and then outputs it to vd. When SEW is configured as 32bit, the SC16 complex multiplication operation is performed and the 32bit imaginary part is output.

4. **Operation**

```
VV version
Tmp.i = vs1[i].r * vs2[i].i + vs1[i].i * vs2[i].r  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
vd[i] = clip_low_SEW(alg_round_shift_right(Tmp.i, mulsft)) //SEW

VS version
```

```
Tmp.i = vs1[0].r * vs2[i].i + vs1[0].i * vs2[i].r  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
vd[i] = clip_low_SEW(alg_round_shift_right(Tmp.i, mulsft)) //SEW
```

6.20. vdscmuljr(vector dynamic scaling complex conjugate multiply real part)

1. **Grammar**

   vdscmuljr.vv vd, vs2, vs1, vm
   vdscmuljr.vs vd, vs2, vs1, vm

2. **Purpose**

   Dynamically scaled conjugate multiplication of fixed-point complex numbers, outputting twice the bit-width real part.

3. **Description**

   vs2 multiply the conjugate of vs1/vs1[0]. The real part of the conjugate multiplication result retains 2 times the bit width, and is output to vd after performing an arithmetic right shift according to the scaling requirements. When SEW is configured as 32bit, the complex conjugate multiplication operation of SC16 is performed, and the real part of 32bit is output.

4. **Operation**

```
VV version
Tmp.r = vs1[i].r * vs2[i].r + vs1[i].i * vs2[i].i  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
vd[i] = clip_low_SEW(alg_round_shift_right(Tmp.r, mulsft)) //SEW

VS version
Tmp.r = vs1[0].r * vs2[i].r + vs1[0].i * vs2[i].i  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
vd[i] = clip_low_SEW(alg_round_shift_right(Tmp.r, mulsft)) //SEW
```

6.21. vdscmulji(vector dynamic scaling complex conjugate multiply image part)

1. **Grammar**

   vdscmulji.vv vd, vs2, vs1, vm
   vdscmulji.vs vd, vs2, vs1, vm

2. **Purpose**

   Dynamically scaled conjugate multiplication of fixed-point complex numbers, outputting twice the bit width of the imaginary part.

3. **Description**

vs2 multiply the conjugate of vs1/vs1[0]. The imaginary part of the conjugate multiplication result retains 2 times the bit width, and is output to vd after performing an arithmetic right shift according to the scaling requirements. When SEW is configured as 32bit, the complex conjugate multiplication operation of SC16 is performed and the 32bit imaginary part is output.

4. **Operation**

```
VV version
Tmp.i = vs1[i].i * vs2[i].r - vs1[i].r * vs2[i].i  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
vd[i] = clip_low_SEW(alg_round_shift_right(Tmp.i, mulsft)) //SEW

VS version
Tmp.i = vs1[0].i * vs2[i].r - vs1[0].r * vs2[i].i  // SEW = SEW/2 * SEW/2 -
SEW/2 * SEW/2
vd[i] = clip_low_SEW(alg_round_shift_right(Tmp.i, mulsft)) //SEW
```

== Dynamic scaling Reduced operation

6.22. vdsredsum(vector dynamic scaling reduced sum)

1. **Grammar**

vdsredsum.v vd, vs2, vm

2. **Purpose**

Each element in the vector register is accumulated and then shifted to the right dynamically, and the output width remains unchanged.

3. **Description**

The elements in vs2 are accumulated and the accumulated value is dynamically shifted to the right by round arithmetic and then the low SEW bit is intercepted and output to vd[0].

4. **Operation**

```
Tmp = 0;
For each i in vs2
Tmp += v0.b[i] == 0 ? 0 : vs2[i]
Vd[0] = clip_low_SEW(alg_round_shift_right(tmp, accsft)) // SEW
```

6.23. vdsredsumn(vector dynamic scaling reduced sum for each n element)

1. **Grammar**

vdsredsumn.vs vd, vs2, rs1, vm
vdsredsumn.vi vd, vs2, uimm, vm

2. **Purpose**

Every n elements in the vector register are accumulated and then round shifted before output, where n is the power of an integer of 2.

3. **Description**

   Every 2^rs1 elements in vs2 are accumulated and then do algorithm round right shift before output to vd.

4. **Operation**

```
VS version
vd[j] = clip_low_SEW(alg_round_shift_right(sum(vs2[j * 2^rs1 .... j * 2^rs1
+ 2^rs1 - 1]), accsft))

VI version
vd[j] = clip_low_SEW(alg_round_shift_right(sum(vs2[j * 2^uimm .... j *
2^uimm + 2^uimm - 1]), accsft))
```

6.24. vredmaxi(vector reduced max with index)

1. **Grammar**

   vredmaxi.vv vd, vs2, vs1, vm

2. **Purpose**

   Find the maximum value in the vector register and its corresponding index value and output it to the first and second elements in the target register.

3. **Description**

   vs2 saves the input value, vs1 saves the index value corresponding to each input value, finds the maximum value in vs2 and writes it into vd[0], and its corresponding index value is written into vd[1]. If the maximum value appears in multiple locations, follow The first maximum value from low to high index position is written into vd, and the value in vs2 and the index value in vs1 are set to the signed minimum value corresponding to SEW.

4. **Operation**

```
vd[0] = max[vs2];
i = max_i[vs2]; // First maximum value searched from lowest element
vd[1] = vs1[i];
vs1[i] = MIN_SEW;
vs2[i] = MIN_SEW;
```

6.25. vredmini(vector reduced min with index)

1. **Grammar**

   vredmini.vv vd, vs2, vs1, vm

2. **Purpose**

   Find the minimum value in the vector register and its corresponding index value and output it to the first and second elements in the target register.

3. **Description**

   vs2 saves the input value, vs1 saves the index value corresponding to each input value, finds the minimum value in vs2 and writes it into vd[0], and its corresponding index value is written into vd[1]. If the minimum value appears in multiple locations, follow The first minimum value of the element index position from low to high is written into vd, and the value in vs2 and the index value in vs1 are set to the signed maximum value corresponding to SEW.

4. **Operation**

```
vd[0] = min[vs2];
i = min_i[vs2]; //First minimum value searched from lowest element
vd[1] = vs1[i];
vs1[i] = MAX_SEW;
vs2[i] = MAX_SEW;
```

Chapter 7. Sequence generation operations

7.1. vpharot(vector phase rotate)

1. **Grammar**

   vpharot.s vd, rs1
   vpharot.v vd, vs2

2. **Purpose**

   Generates a sequence of unit complex numbers with a specified phase.

3. **Description**

   The low 16 bits of rs1 specify the initial phase value of the rotation sequence, the high 16 bits of rs1 specify the step size of the phase rotation, and the SC16 phase rotation sequence is generated and output to vd. Or each element phase input value of vs2 generates the corresponding complex number output to vd. The real and imaginary parts are the fixed-point values of Q16.15.

4. **Operation**

```
S version
vd[i].i = sin(rs1_l + i * rs1_h) // SEW
vd[i].r = cos(rs1_l + i * rs1_h) // SEW

V version
vd[i].i = sin(vs2[i]) // SEW
vd[i].r = cos(vs2[i]) // SEW
```

Chapter 8. Inter-element operation instructions

8.1. vperm(vector permutation)

1. **Grammar**

   vperm.vi vd, vs2, uimm

2. **Purpose**

   According to the element sorting mode predefined by the customer, some elements are taken out from the elements of the source vector register, the elements are reordered according to a certain defined sorting mode and output to the destination vector register.

3. **Description**

   The original data is stored in vs2. uimm selects a predefined sorting mode and takes out elements from vs2 according to the new element sorting mode and places them in new positions in vd. The values at the X index position are filled with 0. The pattern index starts counting from 0.

   Patterns e.g.

   P0：012...31 → 012X345X678X......X 21 22 23X
   P1: 012345 → 012345XX......X X 18 19 20 21 22 23 X X
   P2: 01 → 01010101010101XXXXXXXXXXXXXXXXX
   P3: 01234567 → 0011223344556677XXXXXXXXXXXXXXXX
   P4: 012 → 012012012012XXXXXXXXXXXXXXXXXXXX
   P5: 0123 → 000111222333XXXXXXXXXXXXXXXXXXXX
   P6: 0123 → 0123012301230123XXXXXXXXXXXXXXXX
   P7: 0123 → 0000111122223333XXXXXXXXXXXXXXXX
   P8: 0123 ... 31 → 31 3210

4. **Operation**

```
vd[i] = vs2[PATTEN[uimm][i]]
```

8.2. vfsl(vector funnel shift left)

1. **Grammar**

   vfsl.vv vd, vs2, vs1

2. **Purpose**

   The elements of two vector registers are funnel shifted left.

3. **Description**

   vs2 and vs1 form a new vector with twice the length from left to right. All elements are sequentially shifted left by fsht bits, left shifted out and discarded. Finally, the left half of the elements are intercepted to form a new vector and output to vd.

4. **Operation**

```
vd = vs2[(31- fsft) ... 0] : vs1[31 ... (31 - fsft + 1)]
```

8.3. vfsr(vector funnel shift right)

1. **Grammar**

   vfsr.vv vd, vs2, vs1

2. **Purpose**

   The elements of two vector registers are funnel shifted right.

3. **Description**

   vs2 and vs1 form a new vector with twice the length from left to right. All elements are shifted to the right by fsht bits in order, and the right side is shifted out and discarded. Finally, the right half of the elements are intercepted to form a new vector and output to vd.

4. **Operation**

```
vd = vs2[(fsft - 1) ... 0] : vs1[31 ... fsft]
```

Chapter 9. Fast non-linear operations

9.1. Introduction

In the fast nonlinear instructions vs2 store the fixed-point value in Q32.n format， vs1 store the fix point value n of the fix point format, n belongs to [0, 31]. vd stores the scaling shift value or the mantissa value of the fix point nonlinear output.

The segmentation of the nonlinear function is represented by five parameters A, B, S, SE and T. A represents the slope of the straight line, B represents the bias of the straight line, S represents the mantissa value of the starting position of the straight line on X-axis, SE is the exponent value value of the starting position of the straight line on X-axis, T is the nonlinear function type.

4 segments form a segment group. The vector nonlinear function needs to perform a segmented search for each element in the vector. If each element is searched for all segments at the same time, it consumes too much hardware resources, and the number of segments can change in scenarios with different accuracy requirements. In order to reduce hardware resources and make linear segmentation settings more flexible, linear segments need to be grouped into K segment groups, each group has 4 segments. Each instruction only scans one segment group at a time. More accuracy needs more segment groups, with more segment groups it needs to call the instruction multiple times, each call only scans one of the segment group.

The starting index of the segment group is random. Each time when vnlm instruction is called, the segment group index is incremented by one. After calling K times, the entire segment range is covered.

9.2. Vector nonlinear parameter register

The segmentation information of the nonlinear function is stored through the parameter configuration registers vnlcr0 and vnlcr1. The length of vnlcr0 and vnlcr1 is 1024bits. This register is configured through the vlnlp instruction.

- **vnlpr0 1024bits (vector non-linear paramter register 0)**

R_NL_A[16]	R_NL_B[16]

- **vnlpr1 1024bits (vector non-linear paramter register 1)**

R_NL_SEG_EXP[16]	R_NL_SEG[16]

R_NL_SEG[16]: 16 nonlinear function segment points registers
R_NL_SEG_EXP[16]: 16 nonlinear function segment points exponent registers (required for some nonlinear functions)
R_NL_A[16]: 16 nonlinear function segement slope registers
R_NL_B[16]: 16 nonlinear function segement offset registers

9.3. Instructions

9.3.1. vlnlp(vector load nonlinear parameter)

1. **Grammar**

   vlnlp.s rs1

2. **Purpose**

   Load nonlinear function parameter configuration from vector memory to nonlinear function parameter registers vnlpr0 and vnlpr1

3. **Description**

   Load configuration from vector memory address rs1 to vnlpr0 and vnlpr1, a total of 2048bits.

4. **Operation**

```
vnlpr0 = *rs1;
vnlpr1 = *(rs1 + 128);
```

9.3.2. vnle(vector non-linear exponent calculation instruction)

1. **Grammar**

   vnle.vv vd, vs2, vs1, vm
   vnle.vs vd, vs2, vs1, vm

2. **Purpose**

   Computes the exponential part of the output value of a nonlinear function of an input vector.

3. **Description**

   vs2 stores the fixed-point value, vs1 or vs1[0] stores the fixed-point value n. According to the function type configured in the nonlinear control register, the corresponding exponential result E is calculated, and the result is placed in vd.

4. **Operation**

```
VV version
x[i], E[i] = NORM(R_NL_TYPE, vs2[i], vs1[i])
x[i], s[i], E[i] = preproc(R_NL_TYPE, x[i], E[i])
vd[i] = ExpAdjust(R_NL_TYPE, E[i])

VS version
x[i], E[i] = NORM(R_NL_TYPE, vs2[i], vs1[0])
x[i], s[i], E[i] = preproc(R_NL_TYPE, x[i], E[i])
vd[i] = ExpAdjust(R_NL_TYPE, E[i])
```

9.3.3. vnlm(vector non-linear mantissa calculation instruction)

1. **Grammar**

   vnlm.vv vd, vs2, vs1, vm
   vnlm.vs vd, vs2, vs1, vm

2. **Purpose**

   Computes the linear portion of the output of a nonlinear function of the input vector.

3. **Description**

   vs2 stores the fixed-point value, vs1 or vs1[0] stores the fixed-point value n. According to the function type configured in the nonlinear control register and the segmented parameter configuration, the segmented nonlinear result is calculated. Finally, the results are put into vd after fix point format conversion.

4. **Operation**

```
VV version
x[i], E[i] = NORM(R_NL_TYPE, vs2[i], vs1[i])
x[i], s[i], E[i] = preproc(R_NL_TYPE, x[i], E[i])
segIdx[i] = findSeg(x[i], R_SEG[N])  // N is segment group ID
If (segIdx[i] is valid)
{
tmp = alg_round_shift_right((A[segIdx[i]] * x[i]), n) + B[segIdx[i]];
Vd[i] = postproc(R_NL_TYPE, tmp)
}
N =(N+1) mod K;  //K is the max segments group number

VS version
x[i], E[i] = NORM(R_NL_TYPE, vs2[i], vs1[0])
x[i], s[i], E[i] = preproc(R_NL_TYPE, x[i], E[i])
segIdx[i] = findSeg(x[i], R_SEG[N]) // N is segment group ID
If (segIdx[i] is valid)
{
tmp = alg_round_shift_right((A[segIdx[i]] * x[i]), n) + B[segIdx[i]];
Vd[i] = postproc(R_NL_TYPE, tmp)
}
N =(N+1) mod K; //K is the max segments group number
```

Chapter 10. Apendix A

10.1. RVV & Zvw Encoding space

- Zvw reuses the extension space of RVV (Func7 encoding space 1010111)

Table 3. RVV overall table

inst[4:2]	000	001	010	011	100	101	110	111(> 32b)
inst[6:5]								
00	LOAD	LOAD-FP	custom-0	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	custom-1	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	**RVV**	custom-2/rv128	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	custom-3/rv128	≥80b

- Zvw follow the same Func3 encoding law of RVV

Table 4. RVV Func3 encoding table

funct3[2:0]			Category	Operands	Type of scalar operand
0	0	0	OPIVV	vector-vector	N/A
0	0	1	OPFVV	vector-vector	N/A
0	1	0	OPMVV	vector-vector	N/A
0	1	1	OPIVI	vector-imm	imm[4:0]
1	0	0	OPIVX	vector-scalar	GPR x register rs1
1	0	1	OPFVF	vector-scalar	FP f register rs1
1	1	0	OPMVX	vector-scalar	GPR x register rs1
1	1	1	OPCFG	scalars-imms	GPR x register rs1 & rs2/imm

- Zvw Func6 encoding space in RVV

Table 5. Zvw Func6 encoding table part 1

funct6					funct6				funct6			
000000	V	X	I	vadd	000000	V		vredsum	000000	V	F	vfadd
000001	V	X		***vdsmac***	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

funct6					funct6				funct6			
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		***vpharot***	001100			
001101		X	I	***vdsredsumn***	001101	V		***vdscredsum***	001101			
001110		X	I	vslideup	001110		X	vslide1up	001110		F	vfslide1up
001110	V			vrgatherei16								
001111		X	I	vslidedown	001111		X	vslide1down	001111		F	vfslide1down

Table 6. Zvw Func6 encoding table part 2

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	V		***vcunpackr/i***	010001			
010010	V	X		vsbc	010010	V		VXUNARY0	010010	V		VFUNARY0
010011	V	X		vmsbc	010011	V		***vredmaxi***	010011	V		VFUNARY1
010100	V	X		***vdsmul***	010100	V		VMUNARY0	010100			
010101	V	X	I	***vdsmacini***	010101	V		***vdsredsum***	010101			
010110				***vperm***	010110	V		***vredmini***	010110			
010111	V	X	I	vmerge/vmv	010111	V		vcompress	010111		F	vfmerge/vfmv

funct6					funct6				funct6			
011000	V	X	I	vmseq	011000	V		vmandnot	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		vmornot	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

Table 7. Zvw Func6 encoding table part 3

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	V			***vfsl***	100100	V	X	vmulhu	100100	V	F	vfmul
100101	V	X	I	vsll	100101	V	X	vmul	100101			
100110	V			***vfsr***	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	V		***vcpack***	101000	V	F	vfmadd
101001	V	X	I	vsra	101001	V	X	vmadd	101001	V	F	vfnmadd
101010	V	X	I	vssrl	101010	V	X	***vdsmaco***	101010	V	F	vfmsub
101011	V	X	I	vssra	101011	V	X	vnmsub	101011	V	F	vfnmsub
101100	V	X	I	vnsrl	101100	V	X	***vnle***	101100	V	F	vfmacc
101101	V	X	I	vnsra	101101	V	X	vmacc	101101	V	F	vfnmacc
101110	V	X	I	vnclipu	101110	V		***vnlm***	101110	V	F	vfmsac
101111	V	X	I	vnclip	101111	V	X	vnmsac	101111	V	F	vfnmsac

Table 8. Zvw Func6 encoding table part 4

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	V	X		*vdscmul*	110010	V	X	vwsubu	110010	V	F	vfwsub
110011	V	X		*vdscmulj*	110011	V	X	vwsub	110011	V		vfwredosum
110100	V	X		*vdscmac(j)*	110100	V	X	vwaddu.w	110100	V	F	vfwadd.w
110101	V	X		*vdscmacjoi*	110101	V	X	vwadd.w	110101			
110110	V	X		*vdscmaco*	110110	V	X	vwsubu.w	110110	V	F	vfwsub.w
110111	V	X		*vdscmacor*	110111	V	X	vwsub.w	110111			
111000	V	X		*vdscmacoi*	111000	V	X	vwmulu	111000	V	F	vfwmul
111001	V	X		*vdscmulr*	111001	V	X	*vdsmul*	111001			
111010	V	X		*vdscmuli*	111010	V	X	vwmulsu	111010			
111011	V	X		*vdscmuljr*	111011	V	X	vwmul	111011			
111100	V	X		*vdscmulji*	111100	V	X	vwmaccu	111100	V	F	vfwmacc
111101	V			*vconj*	111101	V	X	vwmacc	111101	V	F	vfwnmacc
111110	V	X		*vdscmacjor*	111110		X	vwmaccus	111110	V	F	vfwmsac
111111	V	X		*vdscmacjo*	111111	V	X	vwmaccsu	111111	V	F	vfwnmsac

10.2. Zvw Instruction encoding

- Zvw Instruction encoding

Table 9. Zvw encoding table

Inst. format	Func6	vm	vs2	vs1/rs1/imm	Func3	vd/vs3	Func7
vlsb.v vd, vs2, vm	010100	vm	vs2	00000	000	vd	1010111
vdsmul.vv vd, vs2, vs1, vm	111001	vm	vs2	vs1	010	vd	1010111
vdsmul.vs vd, vs2, vs1, vm	111001	vm	vs2	vs1	110	vd	1010111

Inst. format	Func6	vm	vs2	vs1/rs1/imm	Func3	vd/vs3	Func7
vdsmacini.v vs2, vm	010101	vm	vs2	00000	000	00000	1010111
vdsmacini.s rs1, vm	010101	vm	00000	rs1	100	00000	1010111
vdsmacini.i uimm, vm	010101	vm	00000	imm	011	00000	1010111
vdsmac.vv vs2, vs1, vm	000001	vm	vs2	vs1	000	00000	1010111
vdsmac.vs vs2, vs1, vm	000001	vm	vs2	vs1	100	00000	1010111
vdsmaco.vv vd,vs2, vs1,vm	101010	vm	vs2	vs1	010	vd	1010111
vdsmaco.vs vd,vs2, vs1, vm	101010	vm	vs2	vs1	110	vd	1010111
vcpack.vv vd, vs2, vs1, vm	101000	vm	vs2	vs1	010	vd	1010111
vcunpackr.v vd, vs2, vm	010001	vm	vs2	00000	010	vd	1010111
vcunpacki.v vd, vs2, vm	010001	vm	vs2	00001	010	vd	1010111
vconj.v vd, vs2, vm	111101	vm	vs2	00000	000	vd	1010111
vdscmul.vv vd, vs2, vs1, vm	110010	vm	vs2	vs1	000	vd	1010111
vdscmul.vs vd, vs2, vs1, vm	110010	vm	vs2	vs1	100	vd	1010111
vdscmulj.vv vd, vs2, vs1, vm	110011	vm	vs2	vs1	000	vd	1010111
vdscmulj.vs vd, vs2, vs1, vm	110011	vm	vs2	vs1	100	vd	1010111
vdscredsum.v vd, vs2, vm	001101	vm	vs2	00000	010	vd	1010111
vdscmac.vv vs2, vs1, vm	110100	vm	vs2	vs1	000	00000	1010111
vdscmac.vs vs2, vs1, vm	110100	vm	vs2	vs1	100	00000	1010111
vdscmacj.vv vs2, vs1, vm	110100	vm	vs2	vs1	000	00001	1010111
vdscmacj.vs vs2, vs1, vm	110100	vm	vs2	vs1	100	00001	1010111

Inst. format	Func6	vm	vs2	vs1/rs1/imm	Func3	vd/vs3	Func7
vdscmaco.vv vd, vs2, vs1, vm	110110	vm	vs2	vs1	000	vd	1010111
vdscmaco.vs vd, vs2, vs1, vm	110110	vm	vs2	vs1	100	vd	1010111
vdscmacjo.vv vd, vs2, vs1, vm	111111	vm	vs2	vs1	000	vd	1010111
vdscmacjo.vs vd, vs2, vs1, vm	111111	vm	vs2	vs1	100	vd	1010111
vdscmacor.vv vd,vs2, vs1, vm	110111	vm	vs2	vs1	000	vd	1010111
vdscmacor.vs vd,vs2, vs1, vm	110111	vm	vs2	vs1	100	vd	1010111
vdscmacjor.vv vd,vs2, vs1, vm	111110	vm	vs2	vs1	000	vd	1010111
vdscmacjor.vs vd,vs2, vs1, vm	111110	vm	vs2	vs1	100	vd	1010111
vdscmacoi.vv vd, vs2, vs1, vm	111000	vm	vs2	vs1	000	vd	1010111
vdscmacoi.vs vd, vs2, vs1, vm	111000	vm	vs2	vs1	100	vd	1010111
vdscmacjoi.vv vd, vs2, vs1, vm	110101	vm	vs2	vs1	000	vd	1010111
vdscmacjoi.vs vd, vs2, vs1, vm	110101	vm	vs2	vs1	100	vd	1010111
vdscmulr.vv vd, vs2, vs1, vm	111001	vm	vs2	vs1	000	vd	1010111
vdscmulr.vs vd, vs2, vs1, vm	111001	vm	vs2	vs1	100	vd	1010111
vdscmuli.vv vd, vs2, vs1, vm	111010	vm	vs2	vs1	000	vd	1010111
vdscmuli.vs vd, vs2, vs1, vm	111010	vm	vs2	vs1	100	vd	1010111

Inst. format	Func6	vm	vs2	vs1/rs1/imm	Func3	vd/vs3	Func7
vdscmuljr.vv vd, vs2, vs1, vm	111011	vm	vs2	vs1	000	vd	1010111
vdscmuljr.vs vd, vs2, vs1, vm	111011	vm	vs2	vs1	100	vd	1010111
vdscmulji.vv vd, vs2, vs1, vm	111100	vm	vs2	vs1	000	vd	1010111
vdscmulji.vs vd, vs2, vs1, vm	111100	vm	vs2	vs1	100	vd	1010111
vdsredsum.v vd, vs2, vm	010101	vm	vs2	00000	010	vd	1010111
vdsredsumn.vs vd, vs2, rs1, vm	001101	vm	vs2	rs1	100	vd	1010111
vdsredsumn.vi vd, vs2, uimm, vm	001101	vm	vs2	imm	011	vd	1010111
vredmaxi.vv vd, vs2, vs1, vm	010011	vm	vs2	vs1	010	vd	1010111
vredmini.vv vd, vs2, vs1, vm	010110	vm	vs2	vs1	010	vd	1010111
vpharot.s vd,rs1	001100	1	00000	rs1	110	vd	1010111
vpharot.v vd,vs2	001100	1	vs2	00000	010	vd	1010111
vperm.vi vd, vs2, uimm	010110	1	vs2	imm	011	vd	1010111
vfsl.vv vd, vs2, vs1	100100	1	vs2	vs1	000	vd	1010111
vfsr.vv vd, vs2, vs1	100110	1	vs2	vs1	000	vd	1010111
vnle.vv vd, vs2, vs1, vm	101100	vm	vs2	vs1	010	vd	1010111
vnle.vs vd, vs2, vs1, vm	101100	vm	vs2	vs1	110	vd	1010111
vnlm.vv vd, vs2, vs1, vm	101110	vm	vs2	vs1	010	vd	1010111
vnlm.vs vd, vs2, vs1, vm	101110	vm	vs2	vs1	110	vd	1010111

- **Zvw Load & Store Instruction encoding**

Table 10. Zvw load & store instruction encoding table

Inst. format	nf-mew-mop	vm	lumop/sumop	rs1	width	vd/vs3	Func7
vlfcb2h.v vd, rs1, vm	000000	vm	10001	rs1	101	vd	0000111
vlfcb2w.v vd, rs1, vm	000000	vm	10010	rs1	110	vd	0000111
vlfch2w.v vd, rs1, vm	000000	vm	10011	rs1	110	vd	0000111
vlfcpa2c.v vd, rs1, vm	000000	vm	10100	rs1	110	vd	0000111
vlfcpb2c.v vd, rs1, vm	000000	vm	10101	rs1	110	vd	0000111
vsfch2b.v vs3, rs1, vm	000000	vm	10001	rs1	101	vs3	0100111
vsfcw2b.v vs3, rs1, vm	000000	vm	10010	rs1	110	vs3	0100111
vsfcw2h.v vs3, rs1, vm	000000	vm	10011	rs1	110	vs3	0100111
vsfcc2pa.v vs3, rs1, vm	000000	vm	10100	rs1	110	vs3	0100111
vsfcc2pb.v vs3, rs1, vm	000000	vm	10101	rs1	110	vs3	0100111
vlnlp.s rs1	000000	1	11000	rs1	110	00000	0000111

.

Chapter 11. Fast non-linear instruction implemenation and paramters

Bibliography