LoongArch Reference Manual

Volume II-a: Vector Instruction Extensions

V1.11

Reading Guide

This manual is Volume II-a of the LoongArch Reference Manual, covering the Vector Instruction Extensions.

Version History

Document Update Log	Document Title	LoongArch Reference Manual Volume II-a: Vector Instruction Extensions
	Version	1.11

Update History

Version	Update Details
1.00	Official release version.
1.10	Manual content refinement and corrections: 1） Section 1.2.3.1 [X]VFRSTPI.{B/H} description is incorrect, the position should be determined by the immediate value. 2） Section 1.2.5.3 [X]VFTINTRMH.L.S description is incorrect, should be "Results are rounded toward negative infinity". 3） Section 1.2.5.4 [X]VFFINT.S.L should be converted to single-precision numbers, [X]VFTINTL.D.W and [X]VFTINTH.D.W should converted to double-precision numbers. 4） Section 1.2.7.12 should be "Select the corresponding half-word/word/double-word element".ww 5） Section 1.2.7.11 adds that the result is undefined when the highest two bits of the index value are non-zero. 6） Section 1.2.4.6 contains errors in the descriptions of “vj/xk” and “vd/xk”. 7） In Section 2.2.33 and Section 2.2.34, "sum" is an incorrect term, it should be "x". 8） Throughout the document, "word integer" is uniformly adjusted to "integer word," and "double-word integer" is uniformly adjusted to "integer double-word." New content added for LoongArch V1.1: 1） Instructions for 128-bits SIMD operations: VFRECIPE.S, VFRECIPE.D, VFRSQRTE.S, and VFRSQRTE.D. 2） Instructions for 256-bits SIMD operations: XVFRECIPE.S, XVFRECIPE.D, XVFRSQRTE.S, and XVFRSQRTE.D.
1.11	Manual content refinement and corrections: 1） Correct the typos in Section 1.2.1.7, Section 1.2.1.8, Section 1.2.1.12, Section 1.2.2.4, Section 1.2.2.6, Section 1.2.2.8, Section 1.2.5.4 , Section 1.2.6.7, Section 1.2.7.12, and Section 1.2.7.14. 2） The descriptions of the "round to nearest, ties to even" operations in the instructions [X]VFRINTRNE.{S/D} in section 1.2.5.2 and [X]VFTINTRNE.{W.S/L.D}, [X]VFTINTRNE.W.D, and [X]VFTINTRNE{L/H}.L.S in section 1.2.5.3 are unified as "round to nearest". 3） Corrected the source operand element subscript in the pseudocode description of the [X]VPICKOD.{B/H/W/D} instruction in Section 1.2.7.9. 4） Corrected an error in immediate name in the virtual address description of the [X]VLDREPL.{W/D} instruction in Section 1.2.8.2. 5） Corrected the pseudocode definitions for Abs_Diff_S() and Abs_Diff_U() in Sections 2.2.10 and Sections 2.2.11 of Appendix A. 6） Corrected descriptions of operations involving saturationin Sections 2.2.26, Sections 2.2.27, Sections 2.2.30, and Sections 2.2.31 of Appendix A.

Table of Contents

1 Vector Extension Instructions 1
1.1 Vector Extension Instruction Programming Model 1
1.1.1 Vector Registers and Extended Vector Registers 1
1.1.2 Vector Element Data Types 1
1.1.3 Vector Floating-Point Operations 1
1.1.4 Endianness 1
1.1.5 Atomic 2
1.2 Overview of Vector Expansion Instructions 2
1.2.1 Vector Integer Arithmetic Instructions 2
1.2.2 Vector Bitwise Operation Instructions 51
1.2.3 Vector String Processing Instructions 93
1.2.4 Vector Floating-Point Operations Instructions 95
1.2.5 Vector Floating-Point Conversion Instructions 103
1.2.6 Vector Integer Comparison, Floating-Point Comparison, and Selection Instructions 114
1.2.7 Vector Shuffle and Mix Instructions 125
1.2.8 Vector Memory Access Instructions 140
2 Appendix A: Pseudocode Description of Function Definitions 142
2.1 Operator Definitions in Pseudocode 142
2.2 Pseudocode Description of Function 144
2.2.1 Saturation Arithmetic of Signed Integer Addition 144
2.2.2 Saturation Arithmetic of Unsigned Integer Addition 144
2.2.3 Saturation Arithmetic of Signed Integer Subtraction 145
2.2.4 Saturation Arithmetic of Unsigned Integer Subtraction 145
2.2.5 Signed Integer Maximum Operation 145
2.2.6 Signed Integer Minimum Operation 145
2.2.7 Unsigned Integer Maximum Operation 145
2.2.8 Unsigned integer Minimum Operation 146
2.2.9 Absolute Operation of Signed Integer 146
2.2.10 Absolute Operation of Difference between Signed Integers 146
2.2.11 Absolute Operation of Difference between Unsigned Integers 146
2.2.12 Logical Left Shift Operation 146
2.2.13 Logical Right Shift Operation 147
2.2.14 Arithmetic Right Shift Operation 147
2.2.15 Circular Right Shift Operation 147
2.2.16 Circular Left Shift Operation 147
2.2.17 Logical Left Shift with Zero Extension to Double Width 147
2.2.18 Logical Left Shift with Sign Extension to Double Width 148
2.2.19 Logical Right Shift with Rounding 148
2.2.20 Arithmetic Right Shift with Rounding 148
2.2.21 Logical Right Shift with Half Width 148
2.2.22 Arithmetic Right Shift with Half Width 148
2.2.23 Logical Right Shift with Rounding and Half Width 149
2.2.24 Arithmetic Right Shift with Rounding and Half Width 149
2.2.25 Logical Right Shift and Unsigned Saturation Arithmetic with Half Width 149
2.2.26 Logical Right Shift and Signed Saturation Arithmetic with Half Width 149

2.2.27 Arithmetic Right Shift and Unsigned Saturation Arithmetic with Half Width 150
2.2.28 Arithmetic Right Shift and Signed Saturation Arithmetic with Half Width 150
2.2.29 Logical Right Shift and Unsigned Saturation Arithmetic with Rounding and Half Width 150
2.2.30 Logical Right Shift and Signed Saturation Arithmetic with Rounding and Half Width 150
2.2.31 Arithmetic Right Shift and Unsigned Saturation Arithmetic with Rounding and Half Width 151
2.2.32 Arithmetic Right Shift and Signed Saturation Arithmetic with Rounding and Half Width 151
2.2.33 Signed Saturation Arithmetic up to Valid Bits 151
2.2.34 Unsigned Saturation Arithmetic up to Valid Bits 151
2.2.35 Operation of Counting the Number of Consecutive 1 Starting from Highest Bit 152
2.2.36 Operation of Counting the Number of Consecutive 0 Starting from Highest Bit 152
2.2.37 Operation of Counting the number of 1 152
2.2.38 Operation of Setting a Specific Bit to 0 in an Element 152
2.2.39 Operation of Setting a Specific Bit to 1 in an Element 152
2.2.40 Operation of Invert a Specified Bit in an Element 153
2.2.41 Operation of Converting Value Based on Sign Bit 153
2.2.42 Operation of Single-precision Floating-point Number Rounding 153
2.2.43 Operation of Double-precision Floating-point Number Rounding 153
2.2.44 Operation of Converting Floating-point Number to Signed Integer Word 154
2.2.45 Operation of Converting Floating-point Number to Signed Integer Double-word 154
2.2.46 Operation of Converting Floating-point Number to Unsigned Integer Word 154
2.2.47 Operation of Converting Floating-point Number to Unsigned Integer Double-word 154
2.2.48 Operation of Generating Value Based on Immediate 155
3 Appendix B: Instruction Code List 156

List of Table

Table 2-1 Explanation of Statement Keywords..... 142
Table 2-2 Explanation of Bit String Operator..... 143
Table 2-3 Explanation of Arithmetic Operators..... 143
Table 2-4 Explanation of Comparison Operator..... 143
Table 2-5 Explanation of Bitwise Operator..... 143
Table 2-6 Explanation of Logical Operator..... 144
Table 2-7 Operator Precedence..... 144

1 Vector Extension Instructions

The vector extension instructions of LoongArch comprise two parts: Vector Extension (Loongson SIMD Extension, abbreviated as LSX) and Advanced Vector Extension (Loongson Advanced SIMD Extension, abbreviated as LASX). Both extensions utilize SIMD instructions and have almost the same instruction functions. The main difference is vector width: LSX operates on 128-bit vectors while LASX operates on 256-bit vectors. LASX represents a further extension of LSX, meaning that implementing LASX necessitates the implementation of LSX.

The implementation of LSX or LASX is independent of whether the architecture is LA32 or LA64. Except for a few instructions that perform 64-bit data transfers between general-purpose registers and vector registers, which require implementation only on LA64 architectures, all other vector extension instructions can be implemented on both LA32 and LA64 architecture processors.

1.1 Vector Extension Instruction Programming Model

1.1.1 Vector Registers and Extended Vector Registers

LSX features 32 vector registers, denoted as v0 through v31. Each vector register has a 128-bit data width. From a software perspective, the data stored in each vector register is a 128-bit vector, with bit 0 located at the rightmost position and bit 127 at the leftmost position.

When vector instructions operate on vector registers, they always use 128 bits as the basic addressing unit. That means there is no situation where the high and low 64 bits of a 128-bit vector register, or four 32-bit registers, are addressed independently.

The lower half of each vector register overlaps with the corresponding floating-point register. That means when executing a basic floating-point instruction which updates floating-point register fN (N=0..31), the corresponding vN[63:0] are updated to the same value. However, note that the value of vN[127:64] is undefined at this point.

LASX contains 32 extended vector registers, denoted x0 through x31. Each extended vector register has a 256-bit data width. Extended vector registers are obtained by extending the upper bits of the vector registers. In other words, the lowest 128 bits of each extended vector register overlap with the corresponding vector register. This naturally implies that the lowest 64 bits of each extended vector register overlap with the corresponding floating-point register. When executing a basic floating-point instruction that updates floating-point register fN (N=0..31), the corresponding xN[63:0] are updated to the same value. However, note that xN[255:64] are undefined. When an LSX instruction updates vector register vN (N=0..31), the corresponding xN[127:0] are updated to the same value. However, note that the values of xN[255:128] are undefined at this point.

1.1.2 Vector Element Data Types

Vector instructions operate on vectors composed of elements of the same type. The vector element types supported by LSX and LASX include:

- Bit, abbreviated as b;
- Integer byte, 8 bits wide, abbreviated as B;
- Integer half-word, 16 bits wide, abbreviated as H;
- Integer word, 32 bits wide, abbreviated as W;
- Integer double word, 64 bits wide, abbreviated as L in floating-point type conversion instructions and as D in other instructions;
- Half-precision floating-point number, 16 bits wide, abbreviated as H;
- Single-precision floating-point number, 32 bits wide, abbreviated as S;
- Double-precision floating-point number, 64 bits wide, abbreviated as D.

When vectors composed of various elements are stored in vector registers, these elements are placed sequentially from the lowest bit to the highest bit within the vector register. In the following Section 1.2, when describing instruction functionality, we will use v.XX[i] to denote the i-th element of type XX in vector v. For example, vN.b[i] denotes the i-th bit element from low to high in the LSX vector register vN, while xvN.B[i] denotes the i-th byte element from low to high in the LASX vector register. The expression forms for the remaining vector element types follow this pattern and are not repeated here.

1.1.3 Vector Floating-Point Operations

The processing rules for vector floating-point elements are identical to those in the basic floating-point instructions.

Control and status records for vector floating-point operations reuse the FCSR register from basic floating-point instructions.

When a vector floating-point instruction generates a floating-point exception, it aggregates all floating-point exceptions generated by individual vector elements to produce the final floating-point exception result for that instruction.

1.1.4 Endianness

Vector data is also stored in little-endian format. When vector operands are stored in memory, their elements are arranged sequentially from the low-order memory address to the high-order memory address.

1.1.5 Atomic

For a single thread, in most cases, a single vector memory access instruction is always atomic, regardless of address alignment. A typical effect of this atomicity is that exceptions (including interrupts) either occur before a vector memory access instruction completes its full execution effect or after it completes its full execution effect. Exceptions cannot occur when a vector memory access instruction has only partially completed its execution effect. Another typical effect of single-threaded atomicity is that the storage operations of a vector storage instruction can only be completed entirely either before or after a barrier. It is impossible for different parts of an address-unaligned vector memory access instruction to have their memory operations distributed on both sides of a barrier.

The reason vector memory access instructions are described as providing atomicity for a single thread in "most cases" rather than "all cases" is that extremely rare scenarios can violate this atomicity. For example, an ECC (Error Correction Code) exception may occur when a vector store instruction spanning cache lines accesses the second cache line.

In multi-threaded, multi-core execution scenarios, unless the addresses of a vector memory access instruction are aligned, it is not guaranteed that the memory operations of that instruction will be observed as atomic by all observers on the same shared consistency domain. However, since the LoongArch employs a weak consistency model and vector memory access instructions are not used to construct barriers, a correct program will naturally ensure that memory operations from unaligned vector memory access instructions are observed atomically by all observers by inserting barriers.

1.2 Overview of Vector Expansion Instructions

1.2.1 Vector Integer Arithmetic Instructions

1.2.1.1 [X]VADD.{B/H/W/D/Q}, [X]VSUB.{B/H/W/D/Q}

Instruction Format:	vadd.b	vd, vj, vk	xvadd.b	xd, xj, xk
	vadd.h	vd, vj, vk	xvadd.h	xd, xj, xk
	vadd.w	vd, vj, vk	xvadd.w	xd, xj, xk
	vadd.d	vd, vj, vk	xvadd.d	xd, xj, xk
	vadd.q	vd, vj, vk	xvadd.q	xd, xj, xk
	vsub.b	vd, vj, vk	xvsub.b	xd, xj, xk
	vsub.h	vd, vj, vk	xvsub.h	xd, xj, xk
	vsub.w	vd, vj, vk	xvsub.w	xd, xj, xk
	vsub.d	vd, vj, vk	xvsub.d	xd, xj, xk
	vsub.q	vd, vj, vk	xvsub.q	xd, xj, xk

The [X]VADD.{B/H/W/D/Q} instruction adds the integer byte/half-word/word/double-word/quad-word elements in vector registers vj/xj to the corresponding integer byte/half-word/word/double-word/quad-word elements in vector register vk/xk. The result is written to the corresponding integer byte/half-word/word/double-word/quad-word element in vector register vd/xd.

VADD.B:

```
for i in range(16) :
    VR[vd].B[i] = VR[vj].B[i] + VR[vk].B[i]
```

VADD.H:

```
for i in range(8):
    VR[vd].H[i] = VR[vj].H[i] + VR[vk].H[i]
```

VADD.W:

```
for i in range(4):
    VR[vd].W[i] = VR[vj].W[i] + VR[vk].W[i]
```

VADD.D:

```
for i in range(2):
    VR[vd].D[i] = VR[vj].D[i] + VR[vk].D[i]
```

VADD.Q:

```
for i in range(1):
    VR[vd].Q[i] = VR[vj].Q[i] + VR[vk].Q[i]
```

XVADD.B:

```
for i in range(32):
    XR[xd].B[i] = XR[xj].B[i] + XR[xk].B[i]
```

XVADD.H:

```
for i in range(16):
    XR[xd].H[i] = XR[xj].H[i] + XR[xk].H[i]
```

XVADD.W:

```
for i in range(8):
    XR[xd].W[i] = XR[xj].W[i] + XR[xk].W[i]
```

XVADD.D:

```
for i in range(4):
    XR[xd].D[i] = XR[xj].D[i] + XR[xk].D[i]
```

XVADD.Q:

```
for i in range(2):
    XR[xd].Q[i] = XR[xj].Q[i] + XR[xk].Q[i]
```

The [X]VSUB.{B/H/W/D/Q} instruction subtracts the corresponding integer byte/half-word/word/double-word/quad-word elements in vector register vk/xk from the corresponding integer byte/half-word/word/double-word/quad-word elements in vector register vj/xj, writing the result into the corresponding integer byte/half-word/word/double-word/quad-word element in vector register vd/xd.

VSUB.B:

```
for i in range(16):
    VR[vd].B[i] = VR[vj].B[i] - VR[vk].B[i]
```

VSUB.H:

```
for i in range(8):
    VR[vd].H[i] = VR[vj].H[i] - VR[vk].H[i]
```

VSUB.W:

```
for i in range(4):
    VR[vd].W[i] = VR[vj].W[i] - VR[vk].W[i]
```

VSUB.D:

```
for i in range(2):
    VR[vd].D[i] = VR[vj].D[i] - VR[vk].D[i]
```

VSUB.Q:

```
for i in range(1):
    VR[vd].Q[i] = VR[vj].Q[i] - VR[vk].Q[i]
```

XVSUB.B:

```
for i in range(32):
    XR[xd].B[i] = XR[xj].B[i] - XR[xk].B[i]
```

XVSUB.H:

```
for i in range(16):
    XR[xd].H[i] = XR[xj].H[i] - XR[xk].H[i]
```

XVSUB.W:

```
for i in range(8):
    XR[xd].W[i] = XR[xj].W[i] - XR[xk].W[i]
```

XVSUB.D:

```
for i in range(4):
    XR[xd].D[i] = XR[xj].D[i] - XR[xk].D[i]
```

XVSUB.Q:

```
for i in range(2):
    XR[xd].Q[i] = XR[xj].Q[i] - XR[xk].Q[i]
```

1.2.1.2 [X]VADDI.{B/H/W/D}U, [X]VSUBI.{B/H/W/D}U

Instruction Format:				
	vaddi.bu	vd, vj, ui5	xvaddi.bu	xd, xj, ui5
	vaddi.hu	vd, vj, ui5	xvaddi.hu	xd, xj, ui5
	vaddi.wu	vd, vj, ui5	xvaddi.wu	xd, xj, ui5
	vaddi.du	vd, vj, ui5	xvaddi.du	xd, xj, ui5
	vsubi.bu	vd, vj, ui5	xvsubi.bu	xd, xj, ui5
	vsubi.hu	vd, vj, ui5	xvsubi.hu	xd, xj, ui5
	vsubi.wu	vd, vj, ui5	xvsubi.wu	xd, xj, ui5
	vsubi.du	vd, vj, ui5	xvsubi.du	xd, xj, ui5

The [X]The VADDI.{B/H/W/D}U instruction adds the integer byte/half-word/word/double-word elements in vector register vj/xj to the immediate value ui5, which is zero-extended to an integer byte/half-word/word/double-word. The result is written into the corresponding integer byte/half-word/word/double-word elements in vector register vd/xd.

VADDI.BU:

```
for i in range(16):
    VR[vd].B[i] = VR[vj].B[i] + ZeroExtend(ui5, 8)
```

VADDI.HU:

```
for i in range(8):
    VR[vd].H[i] = VR[vj].H[i] + ZeroExtend(ui5, 16)
```

VADDI.WU:

```
for i in range(4):
    VR[vd].W[i] = VR[vj].W[i] + ZeroExtend(ui5, 32)
```

VADDI.DU:

```
for i in range(2):
    VR[vd].D[i] = VR[vj].D[i] + ZeroExtend(ui5, 64)
```

XVADDI.BU:

```
for i in range(32):
    XR[xd].B[i] = XR[xj].B[i] + ZeroExtend(ui5, 8)
```

XVADDI.HU:

```
for i in range(16):
    XR[xd].H[i] = XR[xj].H[i] + ZeroExtend(ui5, 16)
```

XVADDI.WU:

```
for i in range(8):
    XR[xd].W[i] = XR[xj].W[i] + ZeroExtend(ui5, 32)
```

XVADDI.DU:

```
for i in range(4):
    XR[xd].D[i] = XR[xj].D[i] + ZeroExtend(ui5, 32)
```

The [X]VSUBI.{B/H/W/D}U instruction subtracts the immediate value ui5, zero-extended to an integer byte/half-word/word/double-word, from the integer byte/half-word/word/double-word elements in vector register vj/xj. The result is written to the corresponding integer byte/half-word/word/double-word elements in vector register vd/xd.

VSUBI.BU:

```
for i in range(16):
    VR[vd].B[i] = VR[vj].B[i] - ZeroExtend(ui5, 8)
```

VSUBI.HU:

```
for i in range(8):
    VR[vd].H[i] = VR[vj].H[i] - ZeroExtend(ui5, 16)
```

VSUBI.WU:

```
for i in range(4):
    VR[vd].W[i] = VR[vj].W[i] - ZeroExtend(ui5, 32)
```

VSUBI.DU:

```
for i in range(2):
    VR[vd].D[i] = VR[vj].D[i] - ZeroExtend(ui5, 64)
```

XVSUBI.BU:

```
for i in range(32):
    XR[xd].B[i] = XR[xj].B[i] - ZeroExtend(ui5, 8)
```

XVSUBI.HU:

```
for i in range(16):
    XR[xd].H[i] = XR[xj].H[i] - ZeroExtend(ui5, 16)
```

XVSUBI.WU:

```
for i in range(8):
    XR[xd].W[i] = XR[xj].W[i] - ZeroExtend(ui5, 32)
```

XVSUBI.DU:

```
for i in range(4):
    XR[xd].D[i] = XR[xj].D[i] - ZeroExtend(ui5, 64)
```

1.2.1.3 [X]VNEG.{B/H/W/D}

Instruction Format:	vneg.b	vd, vj	xvneg.b	xd, xj
	vneg.h	vd, vj	xvneg.h	xd, xj
	vneg.w	vd, vj	xvneg.w	xd, xj
	vneg.d	vd, vj	xvneg.d	xd, xj

The [X]VNEG.{B/H/W/D} instruction negates integer byte/half-word/word/double-word elements in vector register vj/xj, writing the result into the corresponding integer byte/half-word/word/double-word elements in vector register vd/xd.

VNEG.B:

```
for i in range(16):
    VR[vd].B[i] = -VR[vj].B[i]
```

VNEG.H:

```
for i in range(8):
    VR[vd].H[i] = -VR[vj].H[i]
```

VNEG.W:

```
for i in range(4):
    VR[vd].W[i] = -VR[vj].W[i]
```

VNEG.D:

```
for i in range(2):
    VR[vd].D[i] = -VR[vj].D[i]
```

XVNEG.B:

```
for i in range(32):
    XR[xd].B[i] = -XR[xj].B[i]
```

XVNEG.H:

```
for i in range(16):
    XR[xd].H[i] = -XR[xj].H[i]
```

XVNEG.W:

```
for i in range(8):
    XR[xd].W[i] = -XR[xj].W[i]
```

XVNEG.D:

```
for i in range(4):
    XR[xd].D[i] = -XR[xj].D[i]
```

1.2.1.4 [X]VSADD.{B/H/W/D}[U], [X]VSSUB.{B/H/W/D}[U]

Instruction Format:	vsadd.b	vd, vj, vk	xvsadd.b	xd, xj, xk
	vsadd.h	vd, vj, vk	xvsadd.h	xd, xj, xk
	vsadd.w	vd, vj, vk	xvsadd.w	xd, xj, xk

vsadd.d	vd, vj, vk	xvsadd.d	xd, xj, xk
vsadd.bu	vd, vj, vk	xvsadd.bu	xd, xj, xk
vsadd.hu	vd, vj, vk	xvsadd.hu	xd, xj, xk
vsadd.wu	vd, vj, vk	xvsadd.wu	xd, xj, xk
vsadd.du	vd, vj, vk	xvsadd.du	xd, xj, xk
vssub.b	vd, vj, vk	xvssub.b	xd, xj, xk
vssub.h	vd, vj, vk	xvssub.h	xd, xj, xk
vssub.w	vd, vj, vk	xvssub.w	xd, xj, xk
vssub.d	vd, vj, vk	xvssub.d	xd, xj, xk
vssub.bu	vd, vj, vk	xvssub.bu	xd, xj, xk
vssub.hu	vd, vj, vk	xvssub.hu	xd, xj, xk
vssub.wu	vd, vj, vk	xvssub.wu	xd, xj, xk
vssub.du	vd, vj, vk	xvssub.du	xd, xj, xk

The [X]VSADD.{B/H/W/D} instruction performs signed integer addition with saturation arithmetic on the integer byte/half-word/word/double-word elements in vector register vj/xj with the corresponding integer byte/half-word/word/double-word elements at the corresponding positions in vector registers vk/xk. The result is written into the corresponding integer byte/half-word/word/double-word elements in vector registers vd/xd.

VSADD.B:

```
for i in range(16):
    VR[vd].B[i] = SSat_Add_S(VR[vj].B[i], VR[vk].B[i])
```

VSADD.H:

```
for i in range(8):
    VR[vd].H[i] = SSat_Add_S(VR[vj].H[i], VR[vk].H[i])
```

VSADD.W:

```
for i in range(4):
    VR[vd].W[i] = SSat_Add_S(VR[vj].W[i], VR[vk].W[i])
```

VSADD.D:

```
for i in range(2):
    VR[vd].D[i] = SSat_Add_S(VR[vj].D[i], VR[vk].D[i])
```

XVSADD.B:

```
for i in range(32) :
    XR[xd].B[i] = SSat_Add_S(XR[xj].B[i], XR[xk].B[i])
```

XVSADD.H:

```
for i in range(16) :
    XR[xd].H[i] = SSat_Add_S(XR[xj].H[i], XR[xk].H[i])
```

XVSADD.W:

```
for i in range(8):
    XR[xd].W[i] = SSat_Add_S(XR[xj].W[i], XR[xk].W[i])
```

XVSADD.D:

```
for i in range(4):
    XR[xd].D[i] = SSat_Add_S(XR[xj].D[i], XR[xk].D[i])
```

The [X]VSADD.{B/H/W/D}U instruction performs unsigned integer addition with saturation arithmetic on the integer byte/half-word/word/double-word elements in vector register vj/xj with the corresponding integer byte/half-word/word/double-word elements at the corresponding positions in vector register vk/xk. The result is written into the corresponding integer byte/half-word/word/double-word elements in vector register vd/xd.

VSADD.BU:

```
for i in range(16):
    VR[vd].B[i] = USat_Add_U(VR[vj].B[i], VR[vk].B[i])
```

VSADD.HU:

```
for i in range(8):
    VR[vd].H[i] = USat_Add_U(VR[vj].H[i], VR[vk].H[i])
```

VSADD.WU:

```
for i in range(4):
    VR[vd].W[i] = USat_Add_U(VR[vj].W[i], VR[vk].W[i])
```

VSADD.DU:

```
for i in range(2):
    VR[vd].D[i] = USat_Add_U(VR[vj].D[i], VR[vk].D[i])
```

XVSADD.BU:

```
for i in range(32) :
    XR[xd].B[i] = USat_Add_U(XR[xj].B[i], XR[xk].B[i])
```

XVSADD.HU:

```
for i in range(16) :
    XR[xj].H[i] = USat_Add_U(XR[xk].H[i], XR[xj].H[i])
```

XVSADD.WU:

```
for i in range(8):
    XR[xd].W[i] = USat_Add_U(XR[xj].W[i], XR[xk].W[i])
```

XVSADD.DU:

```
for i in range(4):
    XR[xd].D[i] = USat_Add_U(XR[xj].D[i], XR[xk].D[i])
```

The [X]VSSUB.{B/H/W/D} instruction performs signed integer subtraction with saturation arithmetic on the integer byte/half-word/word/double-word elements in vector register vj/xj against the corresponding integer byte/half-word/word/double-word elements at the corresponding positions in vector registers vk/xk. The result is written into the corresponding integer byte/half-word/word/double-word elements in vector registers vd/xd.

VSSUB.B:

```
for i in range(16):
    VR[vd].B[i] = SSat_Sub_S(VR[vj].B[i], VR[vk].B[i])
```

VSSUB.H:

```
for i in range(8):
    VR[vd].H[i] = SSat_Sub_S(VR[vj].H[i], VR[vk].H[i])
```

VSSUB.W:

```
for i in range(4) :
    VR[vd].W[i] = SSat_Sub_S(VR[vj].W[i], VR[vk].W[i])
```

VSSUB.D:

```
for i in range(2):
    VR[vd].D[i] = SSat_Sub_S(VR[vj].D[i], VR[vk].D[i])
```

XVSSUB.B:

```
for i in range(32):
    XR[xd].B[i] = SSat_Sub_S(XR[xj].B[i], XR[xk].B[i])
```

XVSSUB.H:

```
for i in range(16) :
    XR[xd].H[i] = SSat_Sub_S(XR[xj].H[i], XR[xk].H[i])
```

XVSSUB.W:

```
for i in range(8) :
    XR[xd].W[i] = SSat_Sub_S(XR[xj].W[i], XR[xk].W[i])
```

XVSSUB.D:

```
for i in range(4):
    XR[xd].D[i] = SSat_Sub_S(XR[xj].D[i], XR[xk].D[i])
```

The [X]VSSUB.{B/H/W/D}U instruction performs unsigned integer subtraction with saturation arithmetic on the integer byte/half-word/word/double-

word elements in vector register vj/xj against the corresponding integer byte/half-word/word/double-word elements at the corresponding positions in vector register vk/xk. The result is written into the corresponding integer byte/half-word/word/double-word elements in vector register vd/xd.

VSSUB.BU:

```
for i in range(16):
    VR[vd].B[i] = USat_Sub_U(VR[vj].B[i], VR[vk].B[i])
```

VSSUB.HU:

```
for i in range(8):
    VR[vd].H[i] = USat_Sub_U(VR[vj].H[i], VR[vk].H[i])
```

VSSUB.WU:

```
for i in range(4):
    VR[vd].W[i] = USat_Sub_U(VR[vj].W[i], VR[vk].W[i])
```

VSSUB.DU:

```
for i in range(2):
    VR[vd].D[i] = USat_Sub_U(VR[vj].D[i], VR[vk].D[i])
```

XVSSUB.BU:

```
for i in range(32) :
    XR[xd].B[i] = USat_Sub_U(XR[xj].B[i], XR[xk].B[i])
```

XVSSUB.HU:

```
for i in range(16) :
    XR[xd].H[i] = USat_Sub_U(XR[xj].H[i], XR[xk].H[i])
```

XVSSUB.WU:

```
for i in range(8):
    XR[xd].W[i] = USat_Sub_U(XR[xj].W[i], XR[xk].W[i])
```

XVSSUB.DU:

```
for i in range(4):
    XR[xd].D[i] = USat_Sub_U(XR[xj].D[i], XR[xk].D[i])
```

1.2.1.5 [X]VHADDW.{H.B/W.H/D.W/Q.D/HU.BU/WU.HU/DU.WU/QU.DU}, [X]VHSUBW.{ H.B/W.H/D.W/Q.D/HU.BU/WU.HU/DU.WU/QU.DU }

Instruction Format:				
	vhaddw.h.b	vd, vj, vk	xvhaddw.h.b	xd, xj, xk
	vhaddw.w.h	vd, vj, vk	xvhaddw.w.h	xd, xj, xk
	vhaddw.d.w	vd, vj, vk	xvhaddw.d.w	xd, xj, xk
	vhaddw.q.d	vd, vj, vk	xvhaddw.q.d	xd, xj, xk
	vhaddw.hu.bu	vd, vj, vk	xvhaddw.hu.bu	xd, xj, xk
	vhaddw.wu.hu	vd, vj, vk	xvhaddw.wu.hu	xd, xj, xk
	vhaddw.du.wu	vd, vj, vk	xvhaddw.du.wu	xd, xj, xk
	vhaddw.qu.du	vd, vj, vk	xvhaddw.qu.du	xd, xj, xk
	vhsubw.h.b	vd, vj, vk	xvhsubw.h.b	xd, xj, xk
	vhsubw.w.h	vd, vj, vk	xvhsubw.w.h	xd, xj, xk
	vhsubw.d.w	vd, vj, vk	xvhsubw.d.w	xd, xj, xk
	vhsubw.q.d	vd, vj, vk	xvhsubw.q.d	xd, xj, xk
	vhsubw.hu.bu	vd, vj, vk	xvhsubw.hu.bu	xd, xj, xk
	vhsubw.wu.hu	vd, vj, vk	xvhsubw.wu.hu	xd, xj, xk
	vhsubw.du.wu	vd, vj, vk	xvhsubw.du.wu	xd, xj, xk
	vhsubw.qu.du	vd, vj, vk	xvhsubw.qu.du	xd, xj, xk

The [X]VHADDW.{H/W/D/Q}.{B/H/W/D} instruction performs signed integer addition on the byte/half-word/word/double-word elements at each odd position in vector register vj/xj with the corresponding byte/half-word/word/double-word elements in vector register vk/xk. The result is sign-extended to double the width and written into vector register vd/xd.

VHADDW.H.B:

```
for i in range(8) :
    VR[vd].H[i] = SignExtend(signed(VR[vj].B[2i+1]) + signed(VR[vk].B[2i]),16)
```

VHADDW.W.H:

```
for i in range(4):
    VR[vd].W[i] = SignExtend(signed(VR[vj].H[2i+1]) + signed(VR[vk].H[2i]), 32)
```

VHADDW.D.W:

```
for i in range(2) :
    VR[vd].D[i] = SignExtend(signed(VR[vj].W[2i+1]) + signed(VR[vk].W[2i]),64)
```

VHADDW.Q.D:

```
for i in range(1):
    VR[vd].Q[i] = SignExtend(signed(VR[vj].D[2i+1]) + signed(VR[vk].D[2i]), 128)
```

XVHADDW.H.B:

```
for i in range(16):
    XR[xd].H[i] = SignExtend(signed(XR[xj].B[2i+1]) + signed(XR[xk].B[2i]), 16)
```

XVHADDW.W.H:

```
for i in range(8) :
    XR[xd].W[i] = SignExtend(signed(XR[xj].H[2i+1]) + signed(XR[xk].H[2i]),32)
```

XVHADDW.D.W:

```
for i in range(4) :
    XR[xd].D[i] = SignExtend(signed(XR[xj].W[2i+1]) + signed(XR[xk].W[2i]),64)
```

XVHADDW.Q.D:

```
for i in range(2) :
    XR[xd].Q[i] = SignExtend(signed(XR[xj].D[2i+1]) + signed(XR[xk].D[2i]),128)
```

The [X]VHADDW.{H/W/D/Q}U.{B/H/W/D}U instruction performs unsigned integer addition on the byte/half-word/word/double-word elements at each odd position in vector register vj/xj to the corresponding byte/half-word/word/double-word elements in vector register vk/xk. The result is zero-extended to double the width and written into vector register vd/xd.

VHADDW.HU.BU:

```
for i in range(8):
    VR[vd].H[i] = ZeroExtend(unsigned(VR[vj].B[2i+1]) + unsigned(VR[vk].B[2i]), 16)
```

VHADDW.WU.HU:

```
for i in range(4):
    VR[vd].W[i] = ZeroExtend(unsigned(VR[vj].H[2i+1]) + unsigned(VR[vk].H[2i]), 32)
```

VHADDW.DU.WU:

```
for i in range(2):
    VR[vd].D[i] = ZeroExtend(unsigned(VR[vj].W[2i+1]) + unsigned(VR[vk].W[2i]), 64)
```

VHADDW.QU.DU:

```
for i in range(1):
    VR[vd].Q[i] = ZeroExtend(unsigned(VR[vj].D[2i+1]) + unsigned(VR[vk].D[2 i]), 128)
```

XVHADDW.HU.BU:

```
for i in range(16) :
    XR[xd].H[i] = ZeroExtend(unsigned(XR[xj].B[2i+1]) + unsigned(XR[xk].B[2 i]), 16)
```

XVHADDW.WU.HU:

```
for i in range(8):
    XR[xd].W[i] = ZeroExtend(unsigned(XR[xj].H[2i+1]) + unsigned(XR[xk].H[2i]), 32)
```

XVHADDW.DU.WU:

```
for i in range(4):
    XR[xd].D[i] = ZeroExtend(unsigned(XR[xj].W[2i+1]) + unsigned(XR[xk].W[2i]), 64)
```

XVHADDW.QU.DU:

```
for i in range(2):
    XR[xd].Q[i] = ZeroExtend(unsigned(XR[xj].D[2i+1]) + unsigned(XR[xk].D[2i]), 128)
```

The [X]VHSUBW.{H/W/D/Q}.{B/H/W/D} instruction subtracts the byte/half-word/word/double-word elements at each odd position in vector register vk/xk from the corresponding byte/half-word/word/double-word elements in vector register vj/xj at the corresponding even positions. The result is sign-extended to double the width and written into vector register vd/xd.

VHSUBW.H.B:

```
for i in range(8):
    VR[vd].H[i] = SignExtend(signed(VR[vj].B[2i+1]) - signed(VR[vk].B[2i]), 16)
```

VHSUBW.W.H:

```
for i in range(4):
    VR[vd].W[i] = SignExtend(signed(VR[vj].H[2i+1]) - signed(VR[vk].H[2i]), 32)
```

VHSUBW.D.W:

```
for i in range(2):
    VR[vd].D[i] = SignExtend(signed(VR[vj].W[2i+1]) - signed(VR[vk].W[2i]), 64)
```

VHSUBW.Q.D:

```
for i in range(1):
    VR[vd].Q[i] = SignExtend(signed(VR[vj].D[2i+1]) - signed(VR[vk].D[2i]), 128)
```

XVHSUBW.H.B:

```
for i in range(16):
    XR[xd].H[i] = SignExtend(signed(XR[xj].B[2i+1]) - signed(XR[xk].B[2i]), 16)
```

XVHSUBW.W.H:

```
for i in range(8):
    XR[xd].W[i] = SignExtend(signed(XR[xj].H[2i+1]) - signed(XR[xk].H[2i]), 32)
```

XVHSUBW.D.W:

```
for i in range(4):
    XR[xd].D[i] = SignExtend(signed(XR[xj].W[2i+1]) - signed(XR[xk].W[2i]), 64)
```

XVHSUBW.Q.D:

```
for i in range(2):
    XR[xd].Q[i] = SignExtend(signed(XR[xj].D[2i+1]) - signed(XR[xk].D[2i]), 128)
```

The [X]VHSUBW.{H/W/D/Q}U.{B/H/W/D}U instruction subtracts the byte/half-word/word/double-word elements at each odd position in vector register vk/xk from the corresponding byte/half-word/word/double-word elements in vector register vj/xj as unsigned integers. The result is zero-extended to double the width and written into vector register vd/xd.

VHSUBW.HU.BU:

```
for i in range(8):
    VR[vd].H[i] = ZeroExtend(unsigned(VR[vj].B[2i+1]) - unsigned(VR[vk].B[2i]), 16)
```

VHSUBW.WU.HU:

```
for i in range(4):
    VR[vd].W[i] = ZeroExtend(unsigned(VR[vj].H[2i+1]) - unsigned(VR[vk].H[2i]), 32)
```

VHSUBW.DU.WU:

```
for i in range(2):
    VR[vd].D[i] = ZeroExtend(unsigned(VR[vj].W[2i+1]) - unsigned(VR[vk].W[2i]), 64)
```

VHSUBW.QU.DU:

```
for i in range(1):
    VR[vd].Q[i] = ZeroExtend(unsigned(VR[vj].D[2i+1]) - unsigned(VR[vk].D[2i]), 128)
```

XVHSUBW.HU.BU:

```
for i in range(16) :
    XR[xd].H[i] = ZeroExtend(unsigned(XR[xj].B[2i+1]) - unsigned(XR[xk].B[2i]), 16)
```

XVHSUBW.WU.HU:

```
for i in range(8):
    XR[xd].W[i] = ZeroExtend(unsigned(XR[xj].H[2i+1]) - unsigned(XR[xk].H[2i]), 32)
```

XVHSUBW.DU.WU:

```
for i in range(4):
    XR[xd].D[i] = ZeroExtend(unsigned(XR[xj].W[2i+1]) - unsigned(XR[xk].W[2i]), 64)
```

XVHSUBW.QU.DU:

```
for i in range(2):
    XR[xd].Q[i] = ZeroExtend(unsigned(XR[xj].D[2i+1]) - unsigned(XR[xk].D[2i]), 128)
```

1.2.1.6 [X]VADDW{EV/OD}.{H.B/W.H/D.W/Q.D}, [X]VSUBW{EV/OD}.{H.B/W.H/D.W/Q.D}

Instruction Format:				
	vaddwev.h.b	vd, vj, vk	xvaddwev.h.b	xd, xj, xk
	vaddwev.w.h	vd, vj, vk	xvaddwev.w.h	xd, xj, xk
	vaddwev.d.w	vd, vj, vk	xvaddwev.d.w	xd, xj, xk
	vaddwev.q.d	vd, vj, vk	xvaddwev.q.d	xd, xj, xk
	vaddwod.h.b	vd, vj, vk	xvaddwod.h.b	xd, xj, xk
	vaddwod.w.h	vd, vj, vk	xvaddwod.w.h	xd, xj, xk
	vaddwod.d.w	vd, vj, vk	xvaddwod.d.w	xd, xj, xk
	vaddwod.q.d	vd, vj, vk	xvaddwod.q.d	xd, xj, xk
	vsubwev.h.b	vd, vj, vk	xvsubwev.h.b	xd, xj, xk
	vsubwev.w.h	vd, vj, vk	xvsubwev.w.h	xd, xj, xk
	vsubwev.d.w	vd, vj, vk	xvsubwev.d.w	xd, xj, xk
	vsubwev.q.d	vd, vj, vk	xvsubwev.q.d	xd, xj, xk
	vsubwod.h.b	vd, vj, vk	xvsubwod.h.b	xd, xj, xk
	vsubwod.w.h	vd, vj, vk	xvsubwod.w.h	xd, xj, xk
	vsubwod.d.w	vd, vj, vk	xvsubwod.d.w	xd, xj,xk
	vsubwod.q.d	vd, vj, vk	xvsubwod.q.d	xd, xj, xk

The [X]VADDW{EV/OD}.{H.B/W.H/D.W/Q.D} The instruction adds the signed byte/halfword/word/doubleword elements at even/odd positions in vector register vj/xj to the corresponding signed byte/halfword/word/doubleword elements at even/odd positions of vector register vk/xk, after sign-extends them to double the width. The result is written to the corresponding half-word/word/double-word/quad-word elements in vector register vd/xd.

VADDWEV.H.B:

```
for i in range(8):
    VR[vd].H[i] = SignExtend(VR[vj].B[2i], 16) + SignExtend(VR[vk].B[2i], 16)
```

VADDWEV.W.H:

```
for i in range(4):
    VR[vd].W[i] = SignExtend(VR[vj].H[2i], 32) + SignExtend(VR[vk].H[2i], 32)
```

VADDWEV.D.W:

```
for i in range(2):
    VR[vd].D[i] = SignExtend(VR[vj].W[2i], 64) + SignExtend(VR[vk].W[2i], 64)
```

VADDWEV.Q.D:

```
for i in range(1):
    VR[vd].Q[i] = SignExtend(VR[vj].D[2i], 128) + SignExtend(VR[vk].D[2i], 128)
```

VADDWOD.H.B:

```
for i in range(8):
    VR[vd].H[i] = SignExtend(VR[vj].B[2i+1], 16) + SignExtend(VR[vk].B[2i+1], 16)
```

VADDWOD.W.H:

```
for i in range(4) :
    VR[vd].W[i] = SignExtend(VR[vj].H[2i+1], 32) + SignExtend(VR[vk].H[2i+1], 32)
```

VADDWOD.D.W:

```
for i in range(2):
    VR[vd].D[i] = SignExtend(VR[vj].W[2i+1], 64) + SignExtend(VR[vk].W[2i+1], 64)
```

VADDWOD.Q.D:

```
for i in range(1):
    VR[vd].Q[i] = SignExtend(VR[vj].D[2i+1], 128) + SignExtend(VR[vk].D[2i+1], 128)
```

XVADDWEV.H.B:

```
for i in range(16):
    XR[vd].H[i] = SignExtend(XR[vj].B[2i], 16) + SignExtend(XR[vk].B[2i], 16)
```

XVADDWEV.W.H:

```
for i in range(8):
    XR[vd].W[i] = SignExtend(XR[vj].H[2i], 32) + SignExtend(XR[vk].H[2i], 32)
```

XVADDWEV.D.W:

```
for i in range(4):
    XR[vd].D[i] = SignExtend(XR[vj].W[2i], 64) + SignExtend(XR[vk].W[2i], 64)
```

XVADDWEV.Q.D:

```
for i in range(2):
    XR[vd].Q[i] = SignExtend(XR[vj].D[2i], 128) + SignExtend(XR[vk].D[2i], 128)
```

XVADDWOD.H.B:

```
for i in range(16):
    XR[vd].H[i] = SignExtend(XR[vj].B[2i+1], 16) + SignExtend(XR[vk].B[2i+1], 16)
```

XVADDWOD.W.H:

```
for i in range(8):
    XR[vd].W[i] = SignExtend(XR[vj].H[2i+1], 32) + SignExtend(XR[vk].H[2i+1], 32)
```

XVADDWOD.D.W:

```
for i in range(4):
    XR[vd].D[i] = SignExtend(XR[vj].W[2i+1], 64) + SignExtend(XR[vk].W[2i+1], 64)
```

XVADDWOD.Q.D:

```
for i in range(2):
    XR[vd].Q[i] = SignExtend(XR[vj].D[2i+1], 128) + SignExtend(XR[vk].D[2i+1], 128)
```

The [X]VSUBW{EV/OD}.{H.B/W.H/D.W/Q.D} instruction subtracts signed byte/half-word/word/double-word elements at even/odd positions in vector register vk/xk from the corresponding signed byte/halfword/word/doubleword elements at even/odd positions in vector register vj/xj. The result is written to the corresponding half-word/word/double-word/quad-word elements in vector register vd/xd.

VSUBWEV.H.B:

```
for i in range(8):
    VR[vd].H[i] = SignExtend(VR[vj].B[2i], 16) - SignExtend(VR[vk].B[2i], 16)
```

VSUBWEV.W.H:

```
for i in range(4):
    VR[vd].W[i] = SignExtend(VR[vj].H[2i], 32) - SignExtend(VR[vk].H[2i], 32)
```

VSUBWEV.D.W:

```
for i in range(2):
    VR[vd].D[i] = SignExtend(VR[vj].W[2i], 64) - SignExtend(VR[vk].W[2i], 64)
```

VSUBWEV.Q.D:

```
for i in range(1):
    VR[vd].Q[i] = SignExtend(VR[vj].D[2i], 128) - SignExtend(VR[vk].D[2i], 128)
```

VSUBWOD.H.B:

```
for i in range(8):
    VR[vd].H[i] = SignExtend(VR[vj].B[2i+1], 16) - SignExtend(VR[vk].B[2i+1], 16)
```

VSUBWOD.W.H:

```
for i in range(4):
    VR[vd].W[i] = SignExtend(VR[vj].H[2i+1], 32) - SignExtend(VR[vk].H[2i+1], 32)
```

VSUBWOD.D.W:

```
for i in range(2):
    VR[vd].D[i] = SignExtend(VR[vj].W[2i+1], 64) - SignExtend(VR[vk].W[2i+1], 64)
```

VSUBWOD.Q.D:

```
for i in range(1):
    VR[vd].Q[i] = SignExtend(VR[vj].D[2i+1], 128) - SignExtend(VR[vk].D[2i+1], 128)
```

XVSUBWEV.H.B:

```
for i in range(16):
    XR[vd].H[i] = SignExtend(XR[vj].B[2i], 16) - SignExtend(XR[vk].B[2i], 16)
```

XVSUBWEV.W.H:

```
for i in range(8):
    XR[vd].W[i] = SignExtend(XR[vj].H[2i], 32) - SignExtend(XR[vk].H[2i], 32)
```

XVSUBWEV.D.W:

```
for i in range(4):
    XR[vd].D[i] = SignExtend(XR[vj].W[2i], 64) - SignExtend(XR[vk].W[2i], 64)
```

XVSUBWEV.Q.D:

```
for i in range(2):
    XR[vd].Q[i] = SignExtend(XR[vj].D[2i], 128) - SignExtend(XR[vk].D[2i], 128)
```

XVSUBWOD.H.B:

```
for i in range(16) :
    XR[vd].H[i] = SignExtend(XR[vj].B[2i+1], 16) - SignExtend(XR[vk].B[2i+1], 16)
```

XVSUBWOD.W.H:

```
for i in range(8):
    XR[vd].W[i] = SignExtend(XR[vj].H[2i+1], 32) - SignExtend(XR[vk].H[2i+1], 32)
```

XVSUBWOD.D.W:

```
for i in range(4):
    XR[vd].D[i] = SignExtend(XR[vj].W[2i+1], 64) - SignExtend(XR[vk].W[2i+1], 64)
```

XVSUBWOD.Q.D:

```
for i in range(2):
    XR[vd].Q[i] = SignExtend(XR[vj].D[2i+1], 128) - SignExtend(XR[vk].D[2i+1], 128)
```

1.2.1.7 [X]VADDW{EV/OD}.{H.B/W.H/D.W/Q.D}U, [X]VSUBW{EV/OD}.{H.B/W.H/D.W/Q.D}U

Instruction Format:	vaddwev.h.bu	vd, vj, vk	xvaddwev.h.bu	xd, xj, xk
	vaddwev.w.hu	vd, vj, vk	xvaddwev.w.hu	xd, xj, xk
	vaddwev.d.wu	vd, vj, vk	xvaddwev.d.wu	xd, xj, xk
	vaddwev.q.du	vd, vj, vk	xvaddwev.q.du	xd, xj, xk

vaddwod.h.bu	vd, vj, vk	xvaddwod.h.bu	xd, xj, xk
vaddwod.w.hu	vd, vj, vk	xvaddwod.w.hu	xd, xj, xk
vaddwod.d.wu	vd, vj, vk	xvaddwod.d.wu	xd, xj, xk
vaddwod.q.du	vd, vj, vk	xvaddwod.q.du	xd, xj, xk
vsubwev.h.bu	vd, vj, vk	xvsubwev.h.bu	xd, xj, xk
vsubwev.w.hu	vd, vj, vk	xvsubwev.w.hu	xd, xj,xk
vsubwev.d.wu	vd, vj, vk	xvsubwev.d.wu	xd, xj, xk
vsubwev.q.du	vd, vj, vk	xvsubwev.q.du	xd, xj, xk
vsubwod.h.bu	vd, vj, vk	xvsubwod.h.bu	xd, xj, xk
vsubwod.w.hu	vd, vj, vk	xvsubwod.w.hu	xd, xj, xk
vsubwod.d.wu	vd, vj, vk	xvsubwod.d.wu	xd, xj, xk
vsubwod.q.du	vd, vj, vk	xvsubwod.q.du	xd, xj, xk

The [X]VADDW{EV/OD}.{H.B/W.H/D.W/Q.D}U instruction adds unsigned byte/half-word/word/double-word elements at even/odd positions in vector register vj/xj to the corresponding unsigned byte/half-word/word/double-word elements at even/odd positions in vector register vk/xk, after zero-extending to double the width. The result is written into the corresponding half-word/word/double-word/quad-word elements in vector register vd/xd.

VADDWEV.H.BU:

```
for i in range(8):
    VR[vd].H[i] = ZeroExtend(VR[vj].B[2i], 16) + ZeroExtend(VR[vk].B[2i], 16)
```

VADDWEV.W.HU:

```
for i in range(4):
    VR[vd].W[i] = ZeroExtend(VR[vj].H[2i], 32) + ZeroExtend(VR[vk].H[2i], 32)
```

VADDWEV.D.WU:

```
for i in range(2):
    VR[vd].D[i] = ZeroExtend(VR[vj].W[2i], 64) + ZeroExtend(VR[vk].W[2i], 64)
```

VADDWEV.Q.DU:

```
for i in range(1):
    VR[vd].Q[i] = ZeroExtend(VR[vj].D[2i], 128) + ZeroExtend(VR[vk].D[2i], 128)
```

VADDWOD.H.BU:

```
for i in range(8) :
    VR[vd].H[i] = ZeroExtend(VR[vj].B[2i+1], 16) + ZeroExtend(VR[vk].B[2i+1], 16)
```

VADDWOD.W.HU:

```
for i in range(4):
    VR[vd].W[i] = ZeroExtend(VR[vj].H[2i+1], 32) + ZeroExtend(VR[vk].H[2i+1], 32)
```

VADDWOD.D.WU:

```
for i in range(2):
    VR[vd].D[i] = ZeroExtend(VR[vj].W[2i+1], 64) + ZeroExtend(VR[vk].W[2i+1], 64)
```

VADDWOD.Q.DU:

```
for i in range(1):
    VR[vd].Q[i] = ZeroExtend(VR[vj].D[2i+1], 128) + ZeroExtend(VR[vk].D[2i+1], 128)
```

XVADDWEV.H.BU:

```
for i in range(16) :
    XR[vd].H[i] = ZeroExtend(XR[vj].B[2i], 16) + ZeroExtend(XR[vk].B[2i], 16)
```

XVADDWEV.W.HU:

```
for i in range(8):
    XR[vd].W[i] = ZeroExtend(XR[vj].H[2i], 32) + ZeroExtend(XR[vk].H[2i], 32)
```

XVADDWEV.D.WU:

```
for i in range(4):
    XR[vd].D[i] = ZeroExtend(XR[vj].W[2i], 64) + ZeroExtend(XR[vk].W[2i], 64)
```

XVADDWEV.Q.DU:

```
for i in range(2):
    XR[vd].Q[i] = ZeroExtend(XR[vj].D[2i], 128) + ZeroExtend(XR[vk].D[2i], 128)
```

XVADDWOD.H.BU:

```
for i in range(16) :
    XR[vd].H[i] = ZeroExtend(XR[vj].B[2i+1], 16) + ZeroExtend(XR[vk].B[2i+1], 16)
```

XVADDWOD.W.HU:

```
for i in range(8):
    XR[vd].W[i] = ZeroExtend(XR[vj].H[2i+1], 32) + ZeroExtend(XR[vk].H[2i+1], 32)
```

XVADDWOD.D.WU:

```
for i in range(4):
    XR[vd].D[i] = ZeroExtend(XR[vj].W[2i+1], 64) + ZeroExtend(XR[vk].W[2i+1], 64)
```

XVADDWOD.Q.DU:

```
for i in range(2):
    XR[vd].Q[i] = ZeroExtend(XR[vj].D[2i+1], 128) + ZeroExtend(XR[vk].D[2i+1], 128)
```

The [X]VSUBW{EV/OD}.{H.B/W.H/D.W/Q.D}U instruction subtracts signed byte/half-word/word/double-word elements at even/odd positions in vector register vk/xk from the corresponding even/odd positions of signed byte/half-word/word/double-word elements in vector register vj/xj, after zero-extending to double the width. The result is written to the corresponding half-word/word/double-word/quad-word elements in vector register vd/xd.

VSUBWEV.H.BU:

```
for i in range(8) :
    VR[vd].H[i] = ZeroExtend(VR[vj].B[2i], 16) - ZeroExtend(VR[vk].B[2i], 16)
```

VSUBWEV.W.HU:

```
for i in range(4):
    VR[vd].W[i] = ZeroExtend(VR[vj].H[2i], 32) - ZeroExtend(VR[vk].H[2i], 32)
```

VSUBWEV.D.WU:

```
for i in range(2):
    VR[vd].D[i] = ZeroExtend(VR[vj].W[2i], 64) - ZeroExtend(VR[vk].W[2i], 64)
```

VSUBWEV.Q.DU:

```
for i in range(1):
    VR[vd].Q[i] = ZeroExtend(VR[vj].D[2i], 128) - ZeroExtend(VR[vk].D[2i], 128)
```

VSUBWOD.H.BU:

```
for i in range(8) :
    VR[vd].H[i] = ZeroExtend(VR[vj].B[2i+1], 16) - ZeroExtend(VR[vk].B[2i+1], 16)
```

VSUBWOD.W.HU:

```
for i in range(4) :
    VR[vd].W[i] = ZeroExtend(VR[vj].H[2i+1], 32) - ZeroExtend(VR[vk].H[2i+1], 32)
```

VSUBWOD.D.WU:

```
for i in range(2):
    VR[vd].D[i] = ZeroExtend(VR[vj].W[2i+1], 64) - ZeroExtend(VR[vk].W[2i+1], 64)
```

VSUBWOD.Q.DU:

```
for i in range(1):
    VR[vd].Q[i] = ZeroExtend(VR[vj].D[2i+1], 128) - ZeroExtend(VR[vk].D[2i+1], 128)
```

XVSUBWEV.H.BU:

```
for i in range(16) :
    XR[vd].H[i] = ZeroExtend(XR[vj].B[2i], 16) - ZeroExtend(XR[vk].B[2i], 16)
```

XVSUBWEV.W.HU:

```
for i in range(8):
    XR[vd].W[i] = ZeroExtend(XR[vj].H[2i], 32) - ZeroExtend(XR[vk].H[2i], 32)
```

XVSUBWEV.D.WU:

```
for i in range(4) :
    XR[vd].D[i] = ZeroExtend(XR[vj].W[2i], 64) - ZeroExtend(XR[vk].W[2i], 64)
```

XVSUBWEV.Q.DU:

```
for i in range(2):
    XR[vd].Q[i] = ZeroExtend(XR[vj].D[2i], 128) - ZeroExtend(XR[vk].D[2i], 128)
```

XVSUBWOD.H.BU:

```
for i in range(16) :
    XR[vd].H[i] = ZeroExtend(XR[vj].B[2i+1], 16) - ZeroExtend(XR[vk].B[2i+1], 16)
```

XVSUBWOD.W.HU:

```
for i in range(8):
    XR[vd].W[i] = ZeroExtend(XR[vj].H[2i+1], 32) - ZeroExtend(XR[vk].H[2i+1], 32)
```

XVSUBWOD.D.WU:

```
for i in range(4):
    XR[vd].D[i] = ZeroExtend(XR[vj].W[2i+1], 64) - ZeroExtend(XR[vk].W[2i+1], 64)
```

XVSUBWOD.Q.DU:

```
for i in range(2):
    XR[vd].Q[i] = ZeroExtend(XR[vj].D[2i+1], 128) - ZeroExtend(XR[vk].D[2i+1], 128)
```

1.2.1.8 [X]VADDW{EV/OD}.{H.BU.B/W.HU.H/D.WU.W/Q.DU.D}

Instruction Format:	vaddwev.h.bu.b	vd, vj, vk	xvaddwev.h.bu.b	xd, xj, xk
	vaddwev.w.hu.h	vd, vj, vk	xvaddwev.w.hu.h	xd, xj, xk
	vaddwev.d.wu.w	vd, vj, vk	xvaddwev.d.wu.w	xd, xj, xk
	vaddwev.q.du.d	vd, vj, vk	xvaddwev.q.du.d	xd, xj, xk
	vaddwod.h.bu.b	vd, vj, vk	xvaddwod.h.bu.b	xd, xj, xk
	vaddwod.w.hu.h	vd, vj, vk	xvaddwod.w.hu.h	xd, xj, xk
	vaddwod.d.wu.w	vd, vj, vk	xvaddwod.d.wu.w	xd, xj, xk
	vaddwod.q.du.d	vd, vj, vk	xvaddwod.q.du.d	xd, xj, xk

The [X]VADDW{EV/OD}.{H.BU.B/W.HU.H/D.WU.W/Q.DU.D} instruction adds unsigned byte/half-word/word/double-word elements in the even/odd positions of vector register vj/xj to signed byte/half-word/word/double-word elements in the corresponding even/odd positions of vector register vk/xk, after signed-extending to double the width. The result is written to the half-word/word/double-word/quad-word elements in vector register vd/xd.

VADDWEV.H.BU.B:

```
for i in range(8):
    VR[vd].H[i] = ZeroExtend(VR[vj].B[2i], 16) + SignExtend(VR[vk].B[2i], 16)
```

VADDWEV.W.HU.H:

```
for i in range(4):
    VR[vd].W[i] = ZeroExtend(VR[vj].H[2i], 32) + SignExtend(VR[vk].H[2i], 32)
```

VADDWEV.D.WU.W:

```
for i in range(2):
    VR[vd].D[i] = ZeroExtend(VR[vj].W[2i], 64) + SignExtend(VR[vk].W[2i], 64)
```

VADDWEV.Q.DU.D:

```
for i in range(1):
    VR[vd].Q[i] = ZeroExtend(VR[vj].D[2i], 128) + SignExtend(VR[vk].D[2i], 128)
```

VADDWOD.H.BU.B:

```
for i in range(8) :
    VR[vd].H[i] = ZeroExtend(VR[vj].B[2i+1], 16) + SignExtend(VR[vk].B[2i+1], 16)
```

VADDWOD.W.HU.H:

```
for i in range(4):
    VR[vd].W[i] = ZeroExtend(VR[vj].H[2i+1], 32) + SignExtend(VR[vk].H[2i+1], 32)
```

VADDWOD.D.WU.W:

```
for i in range(2):
    VR[vd].D[i] = ZeroExtend(VR[vj].W[2i+1], 64) + SignExtend(VR[vk].W[2i+1], 64)
```

VADDWOD.Q.DU.D:

```
for i in range(1):
    VR[vd].Q[i] = ZeroExtend(VR[vj].D[2i+1], 128) + SignExtend(VR[vk].D[2i+1], 128)
```

XVADDWEV.H.BU.B:

```
for i in range(16) :
    XR[vd].H[i] = ZeroExtend(XR[vj].B[2i], 16) + SignExtend(XR[vk].B[2i], 16)
```

XVADDWEV.W.HU.H:

```
for i in range(8):
    XR[vd].W[i] = ZeroExtend(XR[vj].H[2i], 32) + SignExtend(XR[vk].H[2i], 32)
```

XVADDWEV.D.WU.W:

```
for i in range(4):
    XR[vd].D[i] = ZeroExtend(XR[vj].W[2i], 64) + SignExtend(XR[vk].W[2i], 64)
```

XVADDWEV.Q.DU.D:

```
for i in range(2):
    XR[vd].Q[i] = ZeroExtend(XR[vj].D[2i], 128) + SignExtend(XR[vk].D[2i], 128)
```

XVADDWOD.H.BU.B:

```
for i in range(16) :
    XR[vd].H[i] = ZeroExtend(XR[vj].B[2i+1], 16) + SignExtend(XR[vk].B[2i+1], 16)
```

XVADDWOD.W.HU.H:

```
for i in range(8):
    XR[vd].W[i] = ZeroExtend(XR[vj].H[2i+1], 32) + SignExtend(XR[vk].H[2i+1], 32)
```

XVADDWOD.D.WU.W:

```
for i in range(4):
    XR[vd].D[i] = ZeroExtend(XR[vj].W[2i+1], 64) + SignExtend(XR[vk].W[2i+1], 64)
```

XVADDWOD.Q.DU.D:

```
for i in range(2):
    XR[vd].Q[i] = ZeroExtend(XR[vj].D[2i+1], 128) + SignExtend(XR[vk].D[2i+1], 128)
```

1.2.1.9 [X]VAVG.{B/H/W/D}[U], [X]VAVGR.{B/H/W/D}[U]

Instruction Format:				
	vavg.b	vd, vj, vk	xvavg.b	xd, xj, xk
	vavg.h	vd, vj, vk	xvavg.h	xd, xj, xk
	vavg.w	vd, vj, vk	xvavg.w	xd, xj, xk
	vavg.d	vd, vj, vk	xvavg.d	xd, xj, xk
	vavg.bu	vd, vj, vk	xvavg.bu	xd, xj, xk
	vavg.hu	vd, vj, vk	xvavg.hu	xd, xj, xk
	vavg.wu	vd, vj, vk	xvavg.wu	xd, xj, xk
	vavg.du	vd, vj, vk	xvavg.du	xd, xj, xk
	vavgr.b	vd, vj, vk	xvavgr.b	xd, xj, xk
	vavgr.h	vd, vj, vk	xvavgr.h	xd, xj, xk
	vavgr.w	vd, vj, vk	xvavgr.w	xd, xj, xk

vavgr.d	vd, vj, vk	xvavgr.d	xd, xj, xk
vavgr.bu	vd, vj, vk	xvavgr.bu	xd, xj, xk
vavgr.hu	vd, vj, vk	xvavgr.hu	xd, xj, xk
vavgr.wu	vd, vj, vk	xvavgr.wu	xd, xj, xk
vavgr.du	vd, vj, vk	xvavgr.du	xd, xj, xk

The [X]VAVG.{B/H/W/D} instruction performs an average operation on the signed integer byte/half-word/word/double-word elements in vector register vj/xj and the corresponding signed integer byte/half-word/word/double-word elements at the corresponding positions in vector register vk/xk. The results are written to the byte/half-word/word/double-word elements at the corresponding positions in vector register vd/xd.

VAVG.B:

```
for i in range(16) :
    VR[vd].B[i] = (signed(VR[vj].B[i]) + signed(VR[vk].B[i])) / 2
```

VAVG.H:

```
for i in range(8):
    VR[vd].H[i] = (signed(VR[vj].H[i]) + signed(VR[vk].H[i])) / 2
```

VAVG.W:

```
for i in range(4) :
    VR[vd].W[i] = (signed(VR[vj].W[i]) + signed(VR[vk].W[i])) / 2
```

VAVG.D:

```
for i in range(2):
    VR[vd].D[i] = (signed(VR[vj].D[i]) + signed(VR[vk].D[i])) / 2
```

XVAVG.B:

```
for i in range(32) :
    XR[xd].B[i] = (signed(XR[xj].B[i]) + signed(XR[xk].B[i])) / 2
```

XVAVG.H:

```
for i in range(16):
    XR[xd].H[i] = (signed(XR[xj].H[i]) + signed(XR[xk].H[i])) / 2
```

XVAVG.W:

```
for i in range(8) :
    XR[xd].W[i] = (signed(XR[xj].W[i]) + signed(XR[xk].W[i])) / 2
```

XVAVG.D:

```
for i in range(4):
    XR[xd].D[i] = (signed(XR[xj].D[i]) + signed(XR[xk].D[i])) / 2
```

The [X]VAVG.{BU/HU/WU/DU} instruction performs an average operation on the unsigned integer byte/half-word/word/double-word elements in vector register vj/xj and the corresponding unsigned integer byte/half-word/word/double-word elements at the corresponding positions in vector register vk/xk. The results are written to the byte/half-word/word/double-word elements at the corresponding positions in vector register vd/xd.

VAVG.BU:

```
for i in range(16) :
    VR[vd].B[i] = (unsigned(VR[vj].B[i]) + unsigned(VR[vk].B[i])) / 2
```

VAVG.HU:

```
for i in range(8):
    VR[vd].H[i] = (unsigned(VR[vj].H[i]) + unsigned(VR[vk].H[i])) / 2
```

VAVG.WU:

```
for i in range(4) :
    VR[vd].W[i] = (unsigned(VR[vj].W[i]) + unsigned(VR[vk].W[i])) / 2
```

VAVG.DU:

```
for i in range(2):
    VR[vd].D[i] = (unsigned(VR[vj].D[i]) + unsigned(VR[vk].D[i])) / 2
```

XVAVG.BU:

```
for i in range(32) :
    XR[xd].B[i] = (unsigned(XR[xj].B[i]) + unsigned(XR[xk].B[i])) / 2
```

XVAVG.HU:

```
for i in range(16):
    XR[xd].H[i] = (unsigned(XR[xj].H[i]) + unsigned(XR[xk].H[i])) / 2
```

XVAVG.WU:

```
for i in range(8):
    XR[xd].W[i] = (unsigned(XR[xj].W[i]) + unsigned(XR[xk].W[i])) / 2
```

XVAVG.DU:

```
for i in range(4):
    XR[xd].D[i] = (unsigned(XR[xj].D[i]) + unsigned(XR[xk].D[i])) / 2
```

The [X]VAVGR.{B/H/W/D} instruction performs an average operation with rounding on signed integer byte/half-word/word/double-word elements in vector register vj/xj and the corresponding signed integer byte/half-word/word/double-word elements at the corresponding positions in vector register vk/xk. The results are written to the byte/half-word/word/double-word elements at the corresponding positions in vector register vd/xd.

VAVGR.B:

```
for i in range(16) :
    VR[vd].B[i] = (signed(VR[vj].B[i]) + signed(VR[vk].B[i])+1) / 2
```

VAVGR.H:

```
for i in range(8):
    VR[vd].H[i] = (signed(VR[vj].H[i]) + signed(VR[vk].H[i])+1) / 2
```

VAVGR.W:

```
for i in range(4) :
    VR[vd].W[i] = (signed(VR[vj].W[i]) + signed(VR[vk].W[i])+1) / 2
```

VAVGR.D:

```
for i in range(2):
    VR[vd].D[i] = (signed(VR[vj].D[i]) + signed(VR[vk].D[i])+1) / 2
```

XVAVGR.B:

```
for i in range(32) :
    XR[xd].B[i] = (signed(XR[xj].B[i]) + signed(XR[xk].B[i])+1) / 2
```

XVAVGR.H:

```
for i in range(16):
    XR[xd].H[i] = (signed(XR[xj].H[i]) + signed(XR[xk].H[i])+1) / 2
```

XVAVGR.W:

```
for i in range(8):
    XR[xd].W[i] = (signed(XR[xj].W[i]) + signed(XR[xk].W[i])+1) / 2
```

XVAVGR.D:

```
for i in range(4):
    XR[xd].D[i] = (signed(XR[xj].D[i]) + signed(XR[xk].D[i])+1) / 2
```

The [X]VAVGR.{B/H/W/D}U instruction performs an average operation with rounding on unsigned integer byte/half-word/word/double-word elements in vector register vj/xj and the corresponding unsigned integer byte/half-word/word/double-word elements at the corresponding positions in vector register vk/xk. The results are written to the byte/half-word/word/double-word elements at the corresponding positions in vector register vd/xd.

VAVGR.BU:

```
for i in range(16):
    VR[vd].B[i] = (unsigned(VR[vj].B[i]) + unsigned(VR[vk].B[i]) + 1) / 2
```

VAVGR.HU:

```
for i in range(8):
    VR[vd].H[i] = (unsigned(VR[vj].H[i]) + unsigned(VR[vk].H[i]) + 1) / 2
```

VAVGR.WU:

```
for i in range(4) :
    VR[vd].W[i] = (unsigned(VR[vj].W[i]) + unsigned(VR[vk].W[i])+1) / 2
```

VAVGR.DU:

```
for i in range(2):
    VR[vd].D[i] = (unsigned(VR[vj].D[i]) + unsigned(VR[vk].D[i]) + 1) / 2
```

XVAVGR.BU:

```
for i in range(32) :
    XR[xd].B[i] = (unsigned(XR[xj].B[i]) + unsigned(XR[xk].B[i]) + 1) / 2
```

XVAVGR.HU:

```
for i in range(16):
    XR[xd].H[i] = (unsigned(XR[xj].H[i]) + unsigned(XR[xk].H[i]) + 1) / 2
```

XVAVGR.WU:

```
for i in range(8):
    XR[xd].W[i] = (unsigned(XR[xj].W[i]) + unsigned(XR[xk].W[i]) + 1) / 2
```

XVAVGR.DU:

```
for i in range(4):
    XR[xd].D[i] = (unsigned(XR[xj].D[i]) + unsigned(XR[xk].D[i]) + 1) / 2
```

1.2.1.10 [X]VABSD.{B/H/W/D}[U]

Instruction Format:				
	vabsd.b	vd, vj, vk	xvabsd.b	xd, xj, xk
	vabsd.h	vd, vj, vk	xvabsd.h	xd, xj, xk
	vabsd.w	vd, vj, vk	xvabsd.w	xd, xj, xk
	vabsd.d	vd, vj, vk	xvabsd.d	xd, xj, xk
	vabsd.bu	vd, vj, vk	xvabsd.bu	xd, xj, xk
	vabsd.hu	vd, vj, vk	xvabsd.hu	xd, xj, xk
	vabsd.wu	vd, vj, vk	xvabsd.wu	xd, xj, xk
	vabsd.du	vd, vj, vk	xvabsd.du	xd, xj, xk

The [X]VABSD.{B/H/W/D} instruction performs the absolute value of the signed integer difference between each byte/half-word/word/double-word elements in vector register vj/xj with the corresponding byte/half-word/word/double-word element in vector register vk/xk. The result is written to the corresponding byte/half-word/word/double-word elements in vector register vd/xd.

VABSD.B:

```
for i in range(16):
    VR[vd].B[i] = Abs_Diff_S(VR[vj].B[i], VR[vk].B[i])
```

VABSD.H:

```
for i in range(8):
    VR[vd].H[i] = Abs_Diff_S(VR[vj].H[i], VR[vk].H[i])
```

VABSD.W:

```
for i in range(4):
    VR[vd].W[i] = Abs_Diff_S(VR[vj].W[i], VR[vk].W[i])
```

VABSD.D:

```
for i in range(2):
    VR[vd].D[i] = Abs_Diff_S(VR[vj].D[i], VR[vk].D[i])
```

XVABSD.B:

```
for i in range(32) :
    XR[xd].B[i] = Abs_Diff_S(XR[xj].B[i], XR[xk].B[i])
```

XVABSD.H:

```
for i in range(16) :
    XR[xd].H[i] = Abs_Diff_S(XR[xj].H[i], XR[xk].H[i])
```

XVABSD.W:

```
for i in range(8) :
    XR[xd].W[i] = Abs_Diff_S(XR[xj].W[i], XR[xk].W[i])
```

XVABSD.D:

```
for i in range(4):
    XR[xd].D[i] = Abs_Diff_S(XR[xj].D[i], XR[xk].D[i])
```

The [X]VABSD.{B/H/W/D}U instruction performs the absolute value of the unsigned integer difference between each byte/half-word/word/double-word elements in vector register vj/xj with the corresponding byte/half-word/word/double-word element in vector register vk/xk. The result is written to the corresponding byte/half-word/word/double-word elements in vector register vd/xd.

VABSD.BU:

```
for i in range(16):
    VR[vd].B[i] = Abs_Diff_U(VR[vj].B[i], VR[vk].B[i])
```

VABSD.HU:

```
for i in range(8):
    VR[vd].H[i] = Abs_Diff_U(VR[vj].H[i], VR[vk].H[i])
```

VABSD.WU:

```
for i in range(4):
    VR[vd].W[i] = Abs_Diff_U(VR[vj].W[i], VR[vk].W[i])
```

VABSD.DU:

```
for i in range(2):
    VR[vd].D[i] = Abs_Diff_U(VR[vj].D[i], VR[vk].D[i])
```

XVABSD.BU:

```
for i in range(32) :
    XR[xd].B[i] = Abs_Diff_U(XR[xj].B[i], XR[xk].B[i])
```

XVABSD.HU:

```
for i in range(16) :
    XR[xj].H[i] = Abs_Diff_U(XR[xk].H[i], XR[xj].H[i])
```

XVABSD.WU:

```
for i in range(8):
    XR[xd].W[i] = Abs_Diff_U(XR[xj].W[i], XR[xk].W[i])
```

XVABSD.DU:

```
for i in range(4) :
    XR[xd].D[i] = Abs_Diff_U(XR[xj].D[i], XR[xk].D[i])
```

1.2.1.11 [X]VADDA.{B/H/W/D}

Instruction Format:				
	vadda.b	vd, vj, vk	xvadda.b	xd, xj, xk
	vadda.h	vd, vj, vk	xvadda.h	xd, xj, xk
	vadda.w	vd, vj, vk	xvadda.w	xd, xj, xk
	vadda.d	vd, vj, vk	xvadda.d	xd, xj, xk

The [X]VADDA.{B/H/W/D} instruction adds the absolute values of each signed integer byte/half-word/word/double-word element in vector register vj/xj to the corresponding signed integer byte/half-word/word/double-word element at the corresponding position in vector register vk/xk. The result is written to the

byte/half-word/word/double-word elements at the corresponding position in vector register vd/xd.

VADDA.B:

```
for i in range(16):
    VR[vd].B[i] = Abs(VR[vj].B[i]) + Abs(VR[vk].B[i])
```

VADDA.H:

```
for i in range(8):
    VR[vd].H[i] = Abs(VR[vj].H[i]) + Abs(VR[vk].H[i])
```

VADDA.W:

```
for i in range(4):
    VR[vd].W[i] = Abs(VR[vj].W[i]) + Abs(VR[vk].W[i])
```

VADDA.D:

```
for i in range(2):
    VR[vd].D[i] = Abs(VR[vj].D[i]) + Abs(VR[vk].D[i])
```

XVADDA.B:

```
for i in range(32) :
    XR[xd].B[i] = Abs(XR[xj].B[i]) + Abs(XR[xk].B[i])
```

XVADDA.H:

```
for i in range(16):
    XR[xd].H[i] = Abs(XR[xj].H[i]) + Abs(XR[xk].H[i])
```

XVADDA.W:

```
for i in range(8):
    XR[xd].W[i] = Abs(XR[xj].W[i]) + Abs(XR[xk].W[i])
```

XVADDA.D:

```
for i in range(4):
    XR[xd].D[i] = Abs(XR[xj].D[i]) + Abs(XR[xk].D[i])
```

1.2.1.12 [X]VMAX[I].{B/H/W/D}[U], [X]VMIN[I].{B/H/W/D}[U]

Instruction Format:				
	vmax.b	vd, vj, vk	xvmax.b	xd, xj, xk
	vmax.h	vd, vj, vk	xvmax.h	xd, xj, xk
	vmax.w	vd, vj, vk	xvmax.w	xd, xj, xk
	vmax.d	vd, vj, vk	xvmax.d	xd, xj, xk
	vmaxi.b	vd, vj, si5	xvmaxi.b	xd, xj, si5
	vmaxi.h	vd, vj, si5	xvmaxi.h	xd, xj, si5
	vmaxi.w	vd, vj, si5	xvmaxi.w	xd, xj, si5
	vmaxi.d	vd, vj, si5	xvmaxi.d	xd, xj, si5
	vmax.bu	vd, vj, vk	xvmax.bu	xd, xj, xk
	vmax.hu	vd, vj, vk	xvmax.hu	xd, xj, xk
	vmax.wu	vd, vj, vk	xvmax.wu	xd, xj, xk
	vmax.du	vd, vj, vk	xvmax.du	xd, xj, xk
	vmaxi.bu	vd, vj, ui5	xvmaxi.bu	xd, xj, ui5
	vmaxi.hu	vd, vj, ui5	xvmaxi.hu	xd, xj, ui5
	vmaxi.wu	vd, vj, ui5	xvmaxi.wu	xd, xj, ui5
	vmaxi.du	vd, vj, ui5	xvmaxi.du	xd, xj, ui5
	vmin.b	vd, vj, vk	xvmin.b	xd, xj, xk
	vmin.h	vd, vj, vk	xvmin.h	xd, xj, xk
	vmin.w	vd, vj, vk	xvmin.w	xd, xj, xk
	vmin.d	vd, vj, vk	xvmin.d	xd, xj, xk
	vmini.b	vd, vj, si5	xvmini.b	xd, xj, si5
	vmini.h	vd, vj, si5	xvmini.h	xd, xj, si5
	vmini.w	vd, vj, si5	xvmini.w	xd, xj, si5
	vmini.d	vd, vj, si5	xvmini.d	xd, xj, si5

vmin.bu	vd, vj, vk	xvmin.bu	xd, xj, xk
vmin.hu	vd, vj, vk	xvmin.hu	xd, xj, xk
vmin.wu	vd, vj, vk	xvmin.wu	xd,xj, xk
vmin.du	vd, vj, vk	xvmin.du	xd, xj, xk
vmini.bu	vd, vj, ui5	xvmini.bu	xd, xj, ui5
vmini.hu	vd, vj, ui5	xvmini.hu	xd, xj, ui5
vmini.wu	vd, vj, ui5	xvmini.wu	xd, xj, ui5
vmini.du	vd, vj, ui5	xvmini.du	xd, xj, ui5

The [X]VMAX.{B/H/W/D} instruction performs signed integer maximum operations on each byte/half-word/word/double-word element of vector register vj/xj and the corresponding byte/half-word/word/double-word element in vector register vk/xk. The results are written into the byte/half-word/word/double-word elements in vector register vd/xd.

VMAX.B:

```
for i in range(16) :
    VR[vd].B[i] = Max_S(VR[vj].B[i], VR[vk].B[i])
```

VMAX.H:

```
for i in range(8):
    VR[vd].H[i] = Max_S(VR[vj].H[i], VR[vk].H[i])
```

VMAX.W:

```
for i in range(4):
    VR[vd].W[i] = Max_S(VR[vj].W[i], VR[vk].W[i])
```

VMAX.D:

```
for i in range(2):
    VR[vd].D[i] = Max_S(VR[vj].D[i], VR[vk].D[i])
```

XVMAX.B:

```
for i in range(32) :
    XR[xd].B[i] = Max_S(XR[xj].B[i], XR[xk].B[i])
```

XVMAX.H:

```
for i in range(16) :
    XR[xd].H[i] = Max_S(XR[xj].H[i], XR[xk].H[i])
```

XVMAX.W:

```
for i in range(8):
    XR[xd].W[i] = Max_S(XR[xj].W[i], XR[xk].W[i])
```

XVMAX.D:

```
for i in range(4):
    XR[xd].D[i] = Max_S(XR[xj].D[i], XR[xk].D[i])
```

The [X]VMAXI.{B/H/W/D} instruction performs signed integer maximum operation on each byte/half-word/word/double-word element of vector register vj/xj and immediate value si5, which is signed-extended to byte/half-word/word/double-word size. The result is written into the byte/half-word/word/double-word elements of vector register vd/xd.

VMAXI.B:

```
for i in range(16):
    VR[vd].B[i] = Max_S(VR[vj].B[i], SignExtend(si5, 8))
```

VMAXI.H:

```
for i in range(8):
    VR[vd].H[i] = Max_S(VR[vj].H[i], SignExtend(si5, 16))
```

VMAXI.W:

```
for i in range(4):
    VR[vd].W[i] = Max_S(VR[vj].W[i], SignExtend(si5, 32))
```

VMAXI.D:

```
for i in range(2):
    VR[vd].D[i] = Max_S(VR[vj].D[i], SignExtend(si5, 64))
```

XVMAXI.B:

```
for i in range(32) :
    XR[xd].B[i] = Max_S(XR[xj].B[i], SignExtend(si5, 8))
```

XVMAXI.H:

```
for i in range(16):
    XR[xd].H[i] = Max_S(XR[xj].H[i], SignExtend(si5, 16))
```

XVMAXI.W:

```
for i in range(8):
    XR[xd].W[i] = Max_S(XR[xj].W[i], SignExtend(si5, 32))
```

XVMAXI.D:

```
for i in range(4):
    XR[xd].D[i] = Max_S(XR[xj].D[i], SignExtend(si5, 64))
```

The [X]VMAX.{B/H/W/D}U instruction performs unsigned integer maximum operations on each byte/half-word/word/double-word element of vector register vj/xj and the corresponding byte/half-word/word/double-word element in vector register vk/xk. The results are written into the byte/half-word/word/double-word elements in vector register vd/xd.

VMAX.BU

```
for i in range(16) :
    VR[vd].B[i] = Max_U(VR[vj].B[i], VR[vk].B[i])
```

VMAX.HU

```
for i in range(8):
    VR[vd].H[i] = Max_U(VR[vj].H[i], VR[vk].H[i])
```

VMAX.WU

```
for i in range(4):
    VR[vd].W[i] = Max_U(VR[vj].W[i], VR[vk].W[i])
```

VMAX.DU

```
for i in range(2):
    VR[vd].D[i] = Max_U(VR[vj].D[i], VR[vk].D[i])
```

XVMAX.BU

```
for i in range(32):
    XR[xd].B[i] = Max_U(XR[xj].B[i], XR[xk].B[i])
```

XVMAX.HU

```
for i in range(16):
    XR[xd].H[i] = Max_U(XR[xj].H[i], XR[xk].H[i])
```

XVMAX.WU

```
for i in range(8):
    XR[xd].W[i] = Max_U(XR[xj].W[i], XR[xk].W[i])
```

XVMAX.DU

```
for i in range(4):
    XR[xd].D[i] = Max_U(XR[xj].D[i], XR[xk].D[i])
```

The [X]VMAXI.{B/H/W/D}U instruction performs unsigned integer maximum operations on each byte/half-word/word/double-word element of vector register vj/xj and the immediate value ui5, which is zero-extended to the corresponding byte/half-word/word/double-word size. The results are written into the byte/half-word/word/double-word elements of vector register vd/xd.

VMAXI.BU

```
for i in range(16) :
    VR[vd].B[i] = Max_U(VR[vj].B[i], ZeroExtend(ui5, 8))
```

VMAXI.HU

```
for i in range(8):
    VR[vd].H[i] = Max_U(VR[vj].H[i], ZeroExtend(ui5, 16))
```

VMAXI.WU

```
for i in range(4):
    VR[vd].W[i] = Max_U(VR[vj].W[i], ZeroExtend(ui5, 32))
```

VMAXI.DU

```
for i in range(2):
    VR[vd].D[i] = Max_U(VR[vj].D[i], ZeroExtend(ui5, 64))
```

XVMAXI.BU

```
for i in range(32) :
    XR[xd].B[i] = Max_U(XR[xj].B[i], ZeroExtend(ui5, 8))
```

XVMAXI.HU

```
for i in range(16):
    XR[xd].H[i] = Max_U(XR[xj].H[i], ZeroExtend(ui5, 16))
```

XVMAXI.WU

```
for i in range(8):
    XR[xd].W[i] = Max_U(XR[xj].W[i], ZeroExtend(ui5, 32))
```

XVMAXI.DU

```
for i in range(4):
    XR[xd].D[i] = Max_U(XR[xj].D[i], ZeroExtend(ui5, 64))
```

The [X]VMIN.{B/H/W/D} instruction performs signed integer minimum operations on each byte/half-word/word/double-word element of vector register vj/xj and the corresponding byte/half-word/word/double-word element in vector register vk/xk. The results are written into the byte/half-word/word/double-word elements of vector register vd/xd.

VMIN.B:

```
for i in range(16) :
    VR[vd].B[i] = Min_S(VR[vj].B[i], VR[vk].B[i])
```

VMIN.H:

```
for i in range(8):
    VR[vd].H[i] = Min_S(VR[vj].H[i], VR[vk].H[i])
```

VMIN.W:

```
for i in range(4):
    VR[vd].W[i] = Min_S(VR[vj].W[i], VR[vk].W[i])
```

VMIN.D:

```
for i in range(2):
    VR[vd].D[i] = Min_S(VR[vj].D[i], VR[vk].D[i])
```

XVMIN.B:

```
for i in range(32) :
    XR[xd].B[i] = Min_S(XR[xj].B[i], XR[xk].B[i])
```

XVMIN.H:

```
for i in range(16):
    XR[xd].H[i] = Min_S(XR[xj].H[i], XR[xk].H[i])
```

XVMIN.W:

```
for i in range(8):
    XR[xd].W[i] = Min_S(XR[xj].W[i], XR[xk].W[i])
```

XVMIN.D:

```
for i in range(4):
    XR[xd].D[i] = Min_S(XR[xj].D[i], XR[xk].D[i])
```

The [X]VMINI.{B/H/W/D} instruction performs signed integer minimum operations on each byte/half-word/word/double-word elements of vector register vj/xj and the immediate value si5 which is signed-extended to byte/half-word/word/double-word size. The results are written into the byte/half-word/word/double-word elements of vector registers vd/xd.

VMINI.B:

```
for i in range(16) :
    VR[vd].B[i] = Min_S(VR[vj].B[i], SignExtend(si5, 8))
```

VMINI.H:

```
for i in range(8):
    VR[vd].H[i] = Min_S(VR[vj].H[i], SignExtend(si5, 16))
```

VMINI.W:

```
for i in range(4):
    VR[vd].W[i] = Min_S(VR[vj].W[i], SignExtend(si5, 32))
```

VMINI.D:

```
for i in range(2):
    VR[vd].D[i] = Min_S(VR[vj].D[i], SignExtend(si5, 64))
```

XVMINI.B:

```
for i in range(32):
    XR[xd].B[i] = Min_S(XR[xj].B[i], SignExtend(si5, 8))
```

XVMINI.H:

```
for i in range(16) :
    XR[xd].H[i] = Min_S(XR[xj].H[i], SignExtend(si5, 16))
```

XVMINI.W:

```
for i in range(8):
    XR[xd].W[i] = Min_S(XR[xj].W[i], SignExtend(si5, 32))
```

XVMINI.D:

```
for i in range(4):
    XR[xd].D[i] = Min_S(XR[xj].D[i], SignExtend(si5, 64))
```

The [X]VMIN.{B/H/W/D}U instruction performs unsigned integer minimum operations on each integer byte/half-word/word/double-word element of vector register vj/xj and the corresponding byte/half-word/word/double-word element in vector register vk/xk. The result is written into the byte/half-word/word/double-word elements of vector register vd/xd.

VMIN.BU

```
for i in range(16):
    VR[vd].B[i] = Min_U(VR[vj].B[i], VR[vk].B[i])
```

VMIN.HU

```
for i in range(8):
    VR[vd].H[i] = Min_U(VR[vj].H[i], VR[vk].H[i])
```

VMIN.WU

```
for i in range(4):
    VR[vd].W[i] = Min_U(VR[vj].W[i], VR[vk].W[i])
```

VMIN.DU

```
for i in range(2):
    VR[vd].D[i] = Min_U(VR[vj].D[i], VR[vk].D[i])
```

XVMIN.BU

```
for i in range(32):
    XR[xd].B[i] = Min_U(XR[xj].B[i], XR[xk].B[i])
```

XVMIN.HU

```
for i in range(16):
    XR[xd].H[i] = Min_U(XR[xj].H[i], XR[xk].H[i])
```

XVMIN.WU

```
for i in range(8):
    XR[xd].W[i] = Min_U(XR[xj].W[i], XR[xk].W[i])
```

XVMIN.DU

```
for i in range(4):
    XR[xd].D[i] = Min_U(XR[xj].D[i], XR[xk].D[i])
```

The [X]VMINI.{B/H/W/D}U instruction performs unsigned integer minimum operations on each byte/half-word/word/double-word element of vector register vj/xj and the immediate value ui5, which is zero-extended to byte/half-word/word/double-word size. The results are written into the byte/half-word/word/double-word elements of vector register vd/xd.

VMINI.BU

```
for i in range(16) :
    VR[vd].B[i] = Min_U(VR[vj].B[i], ZeroExtend(ui5, 8))
```

VMINI.HU

```
for i in range(8):
    VR[vd].H[i] = Min_U(VR[vj].H[i], ZeroExtend(ui5, 16))
```

VMINI.WU

```
for i in range(4):
    VR[vd].W[i] = Min_U(VR[vj].W[i], ZeroExtend(ui5, 32))
```

VMINI.DU

```
for i in range(2):
    VR[vd].D[i] = Min_U(VR[vj].D[i], ZeroExtend(ui5, 64))
```

XVMINI.BU

```
for i in range(32) :
    XR[xd].B[i] = Min_U(XR[xj].B[i], ZeroExtend(ui5, 8))
```

XVMINI.HU

```
for i in range(16):
    XR[xd].H[i] = Min_U(XR[xj].H[i], ZeroExtend(ui5, 16))
```

XVMINI.WU

```
for i in range(8):
    XR[xd].W[i] = Min_U(XR[xj].W[i], ZeroExtend(ui5, 32))
```

XVMINI.DU

```
for i in range(4):
    XR[xd].D[i] = Min_U(XR[xj].D[i], ZeroExtend(ui5, 64))
```

1.2.1.13 [X]VMUL.{B/H/W/D}, [X]VMUH.{B/H/W/D}[U]

Instruction Format:	vmul.b	vd, vj, vk		xvmul.b	xd, xj, xk
	vmul.h	vd, vj, vk		xvmul.h	xd, xj, xk
	vmul.w	vd, vj, vk		xvmul.w	xd, xj, xk
	vmul.d	vd, vj, vk		xvmul.d	xd, xj, xk

vmuh.b	vd, vj, vk	xvmuh.b	xd, xj, xk
vmuh.h	vd, vj, vk	xvmuh.h	xd, xj, xk
vmuh.w	vd, vj, vk	xvmuh.w	xd, xj, xk
vmuh.d	vd,vj, vk	xvmuh.d	xd, xj, xk
vmuh.bu	vd, vj, vk	xvmuh.bu	xd, xj, xk
vmuh.hu	vd, vj, vk	xvmuh.hu	xd, xj, xk
vmuh.wu	vd, vj, vk	xvmuh.wu	xd, xj, xk
vmuh.du	vd, vj, vk	xvmuh.du	xd, xj, xk

The [X]VMUL.{B/H/W/D} instruction multiplies each integer byte/half-word/word/double-word element of vector register vj/xj with the corresponding integer byte/half-word/word/double-word element of vector register vk/xk. The lower half of the product is written into the corresponding byte/half-word/word/double-word elements of vector register vd/xd.

VMUL.B:

```
for i in range(16):
    product = VR[vj].B[i] * VR[vk].B[i]
    VR[vd].B[i] = product[7:0]
```

VMUL.H:

```
for i in range(8):
    product = VR[vj].H[i] * VR[vk].H[i]
    VR[vd].H[i] = product[15:0]
```

VMUL.W:

```
for i in range(4):
    product = VR[vj].W[i] * VR[vk].W[i]
    VR[vd].W[i] = product[31:0]
```

VMUL.D:

```
for i in range(2):
    product = VR[vj].D[i] * VR[vk].D[i]
    VR[vd].D[i] = product[63:0]
```

XVMUL.B:

```
for i in range(32):
    product = XR[xj].B[i] * XR[xk].B[i]
    XR[xd].B[i] = product[7:0]
```

XVMUL.H:

```
for i in range(16):
    product = XR[xj].H[i] * XR[xk].H[i]
    XR[xd].H[i] = product[15:0]
```

XVMUL.W:

```
for i in range(8):
    product = XR[xj].W[i] * XR[xk].W[i]
    XR[xd].W[i] = product[31:0]
```

XVMUL.D:

```
for i in range(4):
    product = XR[xj].D[i] * XR[xk].D[i]
    XR[xd].D[i] = product[63:0]
```

The [X]VMUH.{B/H/W/D} instruction multiplies each signed integer byte/half-word/word/double-word element of vector register vj/xj with the corresponding signed integer byte/half-word/word/double-word element at the corresponding position in vector register vk/xk. The upper half of the product is written into the byte/half-word/word/double-word elements at the corresponding position in vector register vd/xd.

VMUH.B:

```
for i in range(16):
    product = signed(VR[vj].B[i]) * signed(VR[vk].B[i])
    VR[vd].B[i] = product[15:8]
```

VMUH.H:

```
for i in range(8):
    product = signed(VR[vj].H[i]) * signed(VR[vk].H[i])
    VR[vd].H[i] = product[31:16]
```

VMUH.W:

```
for i in range(4):
    product = signed(VR[vj].W[i]) * signed(VR[vk].W[i])
    VR[vd].W[i] = product[63:32]
```

VMUH.D:

```
for i in range(2):
    product = signed(VR[vj].D[i]) * signed(VR[vk].D[i])
    VR[vd].D[i] = product[127:64]
```

XVMUH.B:

```
for i in range(32):
    product = signed(XR[xj].B[i]) * signed(XR[xk].B[i])
    XR[xd].B[i] = product[15:8]
```

XVMUH.H:

```
for i in range(16):
    product = signed(XR[xj].H[i]) * signed(XR[xk].H[i])
    XR[xd].H[i] = product[31:16]
```

XVMUH.W:

```
for i in range(8):
    product = signed(XR[xj].W[i]) * signed(XR[xk].W[i])
    XR[xd].W[i] = product[63:32]
```

XVMUH.D:

```
for i in range(4):
    product = signed(XR[xj].D[i]) * signed(XR[xk].D[i])
    XR[xd].D[i] = product[127:64]
```

The [X]VMUH.{B/H/W/D}U instruction multiplies each unsigned integer byte/half-word/word/double-word element of vector register vj/xj with the corresponding unsigned integer byte/half-word/word/double-word element at the corresponding position in vector register vk/xk. The upper half of the product is written into the byte/half-word/word/double-word elements at the corresponding position in vector register vd/xd.

VMUH.BU:

```
for i in range(16) :
    product = unsigned(VR[vj].B[i]) * unsigned(VR[vk].B[i])
    VR[vd].B[i] = product[15:8]
```

VMUH.HU:

```
for i in range(8):
    product = unsigned(VR[vj].H[i]) * unsigned(VR[vk].H[i])
    VR[vd].H[i] = product[31:16]
```

VMUH.WU:

```
for i in range(4):
    product = unsigned(VR[vj].W[i]) * unsigned(VR[vk].W[i])
    VR[vd].W[i] = product[63:32]
```

VMUH.DU:

```
for i in range(2):
    product = unsigned(VR[vj].D[i]) * unsigned(VR[vk].D[i])
    VR[vd].D[i] = product[127:64]
```

XVMUH.BU:

```
for i in range(32):
    product = unsigned(XR[xj].B[i]) * unsigned(XR[xk].B[i])
    XR[xd].B[i] = product[15:8]
```

XVMUH.HU:

```
for i in range(16):
    product = unsigned(XR[xj].H[i]) * unsigned(XR[xk].H[i])
    XR[xd].H[i] = product[31:16]
```

XVMUH.WU:

```
for i in range(8):
    product = unsigned(XR[xj].W[i]) * unsigned(XR[xk].W[i])
    XR[xd].W[i] = product[63:32]
```

XVMUH.DU:

```
for i in range(4):
    product = unsigned(XR[xj].D[i]) * unsigned(XR[xk].D[i])
    XR[xd].D[i] = product[127:64]
```

1.2.1.14 [X]VMULW{EV/OD}.{H.B/W.H/D.W/Q.D}[U], [X]VMULW{EV/OD}.{H.BU.B/W.HU.H/D.WU.W/Q.DU.D}

Instruction format:				
	vmulwev.h.b	vd, vj, vk	xvmulwev.h.b	xd, xj, xk
	vmulwev.w.h	vd, vj, vk	xvmulwev.w.h	xd, xj, xk
	vmulwev.d.w	vd, vj, vk	xvmulwev.d.w	xd, xj, xk
	vmulwev.q.d	vd, vj, vk	xvmulwev.q.d	xd, xj, xk
	vmulwod.h.b	vd, vj, vk	xvmulwod.h.b	xd, xj, xk
	vmulwod.w.h	vd, vj, vk	xvmulwod.w.h	xd, xj, xk
	vmulwod.d.w	vd, vj, vk	xvmulwod.d.w	xd, xj, xk
	vmulwod.q.d	vd, vj, vk	xvmulwod.q.d	xd, xj, xk
	vmulwev.h.bu	vd, vj, vk	xvmulwev.h.bu	xd, xj, xk
	vmulwev.w.hu	vd, vj, vk	xvmulwev.w.hu	xd, xj, xk
	vmulwev.d.wu	vd, vj, vk	xvmulwev.d.wu	xd, xj, xk
	vmulwev.q.du	vd, vj, vk	xvmulwev.q.du	xd, xj, xk
	vmulwod.h.bu	vd, vj, vk	xvmulwod.h.bu	xd, xj, xk
	vmulwod.w.hu	vd, vj, vk	xvmulwod.w.hu	xd, xj, xk
	vmulwod.d.wu	vd, vj, vk	xvmulwod.d.wu	xd, xj, xk
	vmulwod.q.du	vd, vj, vk	xvmulwod.q.du	xd, xj, xk
	vmulwev.h.bu.b	vd, vj, vk	xvmulwev.h.bu.b	xd, xj, xk
	vmulwev.w.hu.h	vd, vj, vk	xvmulwev.w.hu.h	xd, xj, xk
	vmulwev.d.wu.w	vd, vj, vk	xvmulwev.d.wu.w	xd, xj, xk
	vmulwev.q.du.d	vd, vj, vk	xvmulwev.q.du.d	xd, xj, xk
	vmulwod.h.bu.b	vd, vj, vk	xvmulwod.h.bu.b	xd, xj, xk
	vmulwod.w.hu.h	vd, vj, vk	xvmulwod.w.hu.h	xd, xj, xk
	vmulwod.d.wu.w	vd, vj, vk	xvmulwod.d.wu.w	xd, xj, xk
	vmulwod.q.du.d	vd, vj, vk	xvmulwod.q.du.d	xd, xj, xk

The [X]VMULW{EV/OD}.{H.B/W.H/D.W/Q.D} instruction multiplies the signed integer byte/half-word/word/double-word elements at even/odd positions in vector registers vj/xj with the corresponding signed integer byte/half-word/word/double-word elements at even/odd positions in vector registers vk/xk. The double-width product is written into the corresponding half-word/word/double-word/quad-word positions of vector register vd/xk.

VMULWEV.H.B:

```
for i in range(8):
    product = signed(VR[vj].B[2i]) * signed(VR[vk].B[2i])
    VR[vd].H[i] = product[15:0]
```

VMULWEV.W.H:

```
for i in range(4):
    product = signed(VR[vj].H[2i]) * signed(VR[vk].H[2i])
    VR[vd].W[i] = product[31:0]
```

VMULWEV.D.W:

```
for i in range(2):
    product = signed(VR[vj].W[2i]) * signed(VR[vk].W[2i])
    VR[vd].D[i] = product[63:0]
```

VMULWEV.Q.D:

```
for i in range(1):
    product = signed(VR[vj].D[2i]) * signed(VR[vk].D[2i])
    VR[vd].Q[i] = product[127:0]
```

VMULWOD.H.B:

```
for i in range(8):
    product = signed(VR[vj].B[2i+1]) * signed(VR[vk].B[2i+1])
    VR[vd].H[i] = product[15:0]
```

VMULWOD.W.H:

```
for i in range(4):
    product = signed(VR[vj].H[2i+1]) * signed(VR[vk].H[2i+1])
    VR[vd].W[i] = product[31:0]
```

VMULWOD.D.W:

```
for i in range(2):
    product = signed(VR[vj].W[2i+1]) * signed(VR[vk].W[2i+1])
    VR[vd].D[i] = product[63:0]
```

VMULWOD.Q.D:

```
for i in range(1):
    product = signed(VR[vj].D[2i+1]) * signed(VR[vk].D[2i+1])
    VR[vd].Q[i] = product[127:0]
```

XVMULWEV.H.B:

```
for i in range(16):
    product = signed(XR[vj].B[2i]) * signed(XR[vk].B[2i])
    VR[vd].H[i] = product[15:0]
```

XVMULWEV.W.H:

```
for i in range(8):
    product = signed(XR[vj].H[2i]) * signed(XR[vk].H[2i])
    VR[vd].W[i] = product[31:0]
```

XVMULWEV.D.W:

```
for i in range(4):
    product = signed(XR[vj].W[2i]) * signed(XR[vk].W[2i])
    VR[vd].D[i] = product[63:0]
```

XVMULWEV.Q.D:

```
for i in range(2):
    product = signed(XR[vj].D[2i]) * signed(XR[vk].D[2i])
    VR[vd].Q[i] = product[127:0]
```

XVMULWOD.H.B:

```
for i in range(16):
    product = signed(XR[vj].B[2i+1]) * signed(XR[vk].B[2i+1])
    VR[vd].H[i] = product[15:0]
```

XVMULWOD.W.H:

```
for i in range(8) :
    product = signed(XR[vj].H[2i+1]) * signed(XR[vk].H[2i+1])
    VR[vd].W[i] = product[31:0]
```

XVMULWOD.D.W:

```
for i in range(4):
    product = signed(XR[vj].W[2i+1]) * signed(XR[vk].W[2i+1])
    VR[vd].D[i] = product[63:0]
```

XVMULWOD.Q.D:

```
for i in range(2):
    product = signed(XR[vj].D[2i+1]) * signed(XR[vk].D[2i+1])
    VR[vd].Q[i] = product[127:0]
```

The [X]VMULW{EV/OD}.{H.B/W.H/D.W/Q.D}U instruction multiplies the unsigned integer byte/half-word/word/double-word elements at even/odd positions in vector registers vj/xj with the corresponding unsigned integer byte/half-word/word/double-word elements at even/odd positions in vector registers vk/xk. The double-width product is written into the corresponding half-word/word/double-word/quad-word positions in vector register vd/xk.

VMULWEV.H.BU:

```
for i in range(8):
    product = unsigned(VR[vj].B[2i]) * unsigned(VR[vk].B[2i])
    VR[vd].H[i] = product[15:0]
```

VMULWEV.W.HU:

```
for i in range(4):
    product = unsigned(VR[vj].H[2i]) * unsigned(VR[vk].H[2i])
    VR[vd].W[i] = product[31:0]
```

VMULWEV.D.WU:

```
for i in range(2):
    product = unsigned(VR[vj].W[2i]) * unsigned(VR[vk].W[2i])
    VR[vd].D[i] = product[63:0]
```

VMULWEV.Q.DU:

```
for i in range(1):
    product = unsigned(VR[vj].D[2i]) * unsigned(VR[vk].D[2i])
    VR[vd].Q[i] = product[127:0]
```

VMULWOD.H.BU:

```
for i in range(8) :
    product = unsigned(VR[vj].B[2i+1]) * unsigned(VR[vk].B[2i+1])
    VR[vd].H[i] = product[15:0]
```

VMULWOD.W.HU:

```
for i in range(4) :
    product = unsigned(VR[vj].H[2i+1]) * unsigned(VR[vk].H[2i+1])
    VR[vd].W[i] = product[31:0]
```

VMULWOD.D.WU:

```
for i in range(2):
    product = unsigned(VR[vj].W[2i+1]) * unsigned(VR[vk].W[2i+1])
    VR[vd].D[i] = product[63:0]
```

VMULWOD.Q.DU:

```
for i in range(1):
    product = unsigned(VR[vj].D[2i+1]) * unsigned(VR[vk].D[2i+1])
    VR[vd].Q[i] = product[127:0]
```

XVMULWEV.H.BU:

```
for i in range(16) :
    product = unsigned(XR[vj].B[2i]) * unsigned(XR[vk].B[2i])
    VR[vd].H[i] = product[15:0]
```

XVMULWEV.W.HU:

```
for i in range(8):
    product = unsigned(XR[vj].H[2i]) * unsigned(XR[vk].H[2i])
    VR[vd].W[i] = product[31:0]
```

XVMULWEV.D.WU:

```
for i in range(4):
    product = unsigned(XR[vj].W[2i]) * unsigned(XR[vk].W[2i])
    VR[vd].D[i] = product[63:0]
```

XVMULWEV.Q.DU:

```
for i in range(2):
    product = unsigned(XR[vj].D[2i]) * unsigned(XR[vk].D[2i])
    VR[vd].Q[i] = product[127:0]
```

XVMULWOD.H.BU:

```
for i in range(16):
    product = unsigned(XR[vj].B[2i+1]) * unsigned(XR[vk].B[2i+1])
    VR[vd].H[i] = product[15:0]
```

XVMULWOD.W.HU:

```
for i in range(8) :
    product = unsigned(XR[vj].H[2i+1]) * unsigned(XR[vk].H[2i+1])
    VR[vd].W[i] = product[31:0]
```

XVMULWOD.D.WU:

```
for i in range(4):
    product = unsigned(XR[vj].W[2i+1]) * unsigned(XR[vk].W[2i+1])
    VR[vd].D[i] = product[63:0]
```

XVMULWOD.Q.DU:

```
for i in range(2):
    product = unsigned(XR[vj].D[2i+1]) * unsigned(XR[vk].D[2i+1])
    VR[vd].Q[i] = product[127:0]
```

The [X]VMULW{EV/OD}.{H.BU.B/W.HU.H/D.WU.W/Q.DU.D} instruction multiplies the unsigned integer byte/half-word/word/double-word elements at even/odd positions in vector register vj/xj with the signed integer byte/half-word/word/double-word elements at the corresponding even/odd positions in vector register vk/xk. The double-width product is written into the corresponding half-word/word/double-word/quad-word positions in vector register vd/xk.

VMULWEV.H.BU.B:

```
for i in range(8):
    product = unsigned(VR[vj].B[2i]) * signed(VR[vk].B[2i])
    VR[vd].H[i] = product[15:0]
```

VMULWEV.W.HU.H:

```
for i in range(4):
    product = unsigned(VR[vj].H[2i]) * signed(VR[vk].H[2i])
    VR[vd].W[i] = product[31:0]
```

VMULWEV.D.WU.W:

```
for i in range(2):
    product = unsigned(VR[vj].W[2i]) * signed(VR[vk].W[2i])
    VR[vd].D[i] = product[63:0]
```

VMULWEV.Q.DU.D:

```
for i in range(1):
    product = unsigned(VR[vj].D[2i]) * signed(VR[vk].D[2i])
    VR[vd].Q[i] = product[127:0]
```

VMULWOD.H.BU.B:

```
for i in range(8):
    product = unsigned(VR[vj].B[2i+1]) * signed(VR[vk].B[2i+1])
    VR[vd].H[i] = product[15:0]
```

VMULWOD.W.HU.H:

```
for i in range(4) :
    product = unsigned(VR[vj].H[2i+1]) * signed(VR[vk].H[2i+1])
    VR[vd].W[i] = product[31:0]
```

VMULWOD.D.WU.W:

```
for i in range(2):
    product = unsigned(VR[vj].W[2i+1]) * signed(VR[vk].W[2i+1])
    VR[vd].D[i] = product[63:0]
```

VMULWOD.Q.DU.D:

```
for i in range(1):
    product = unsigned(VR[vj].D[2i+1]) * signed(VR[vk].D[2i+1])
    VR[vd].Q[i] = product[127:0]
```

XVMULWEV.H.BU.B:

```
for i in range(16) :
    product = unsigned(XR[vj].B[2i]) * signed(XR[vk].B[2i])
    VR[vd].H[i] = product[15:0]
```

XVMULWEV.W.HU.H:

```
for i in range(8) :
    product = unsigned(XR[vj].H[2i]) * signed(XR[vk].H[2i])
    VR[vd].W[i] = product[31:0]
```

XVMULWEV.D.WU.W:

```
for i in range(4):
    product = unsigned(XR[vj].W[2i]) * signed(XR[vk].W[2i])
    VR[vd].D[i] = product[63:0]
```

XVMULWEV.Q.DU.D:

```
for i in range(2):
    product = unsigned(XR[vj].D[2i]) * signed(XR[vk].D[2i])
    VR[vd].Q[i] = product[127:0]
```

XVMULWOD.H.BU.B:

```
for i in range(16) :
    product = unsigned(XR[vj].B[2i+1]) * signed(XR[vk].B[2i+1])
    VR[vd].H[i] = product[15:0]
```

XVMULWOD.W.HU.H:

```
for i in range(8) :
    product = unsigned(XR[vj].H[2i+1]) * signed(XR[vk].H[2i+1])
    VR[vd].W[i] = product[31:0]
```

XVMULWOD.D.WU.W:

```
for i in range(4):
    product = unsigned(XR[vj].W[2i+1]) * signed(XR[vk].W[2i+1])
    VR[vd].D[i] = product[63:0]
```

XVMULWOD.Q.DU.D:

```
for i in range(2):
    product = unsigned(XR[vj].D[2i+1]) * signed(XR[vk].D[2i+1])
    VR[vd].Q[i] = product[127:0]
```

1.2.1.15 [X]VMADD.{B/H/W/D}, [X]VMSUB.{B/H/W/D}

Instruction Format:				
	vmadd.b	vd, vj, vk	xvmadd.b	xd, xj, xk
	vmadd.h	vd, vj, vk	xvmadd.h	xd, xj, xk
	vmadd.w	vd, vj, vk	xvmadd.w	xd, xj, xk
	vmadd.d	vd, vj, vk	xvmadd.d	xd, xj, xk
	vmsub.b	vd, vj, vk	xvmsub.b	xd, xj, xk
	vmsub.h	vd, vj, vk	xvmsub.h	xd, xj, xk
	vmsub.w	vd, vj, vk	xvmsub.w	xd, xj, xk
	vmsub.d	vd, vj, vk	xvmsub.d	xd, xj, xk

The [X]VMADD.{B/H/W/D} instruction multiplies each byte/half-word/word/double-word element in vector register vj/xj with the corresponding byte/half-word/word/double-word element in vector register vk/xk. The lower half of the resulting product is accumulated into the corresponding byte/half-word/word/double-word elements in vector register vd/xd.

VMADD.B:

```
for i in range(16) :
    product = VR[vj].B[i] * VR[vk].B[i]
    VR[vd].B[i] += product[7:0]
```

VMADD.H:

```
for i in range(8):
    product = VR[vj].H[i] * VR[vk].H[i]
    VR[vd].H[i] += product[15:0]
```

VMADD.W:

```
for i in range(4):
    product = VR[vj].W[i] * VR[vk].W[i]
    VR[vd].W[i] += product[31:0]
```

VMADD.D:

```
for i in range(2):
    product = VR[vj].D[i] * VR[vk].D[i]
    VR[vd].D[i] += product[63:0]
```

XVMADD.B:

```
for i in range(32):
    product = XR[xj].B[i] * XR[xk].B[i]
    XR[xd].B[i] += product[7:0]
```

XVMADD.H:

```
for i in range(16):
    product = XR[xj].H[i] * XR[xk].H[i]
    XR[xd].H[i] += product[15:0]
```

XVMADD.W:

```
for i in range(8):
    product = XR[xj].W[i] * XR[xk].W[i]
    XR[xd].W[i] += product[31:0]
```

XVMADD.D:

```
for i in range(4):
    product = XR[xj].D[i] * XR[xk].D[i]
    XR[xd].D[i] += product[63:0]
```

The [X]VMSUB.{B/H/W/D} instruction multiplies each byte/half-word/word/double-word element in vector registers vk/xk with the corresponding byte/half-word/word/double-word element at corresponding positions in vector register vk/xk, then subtracts the lower half of the resulting product from the byte/half-word/word/double-word elements at corresponding positions in vector registers vd/xd.

VMSUB.B:

```
for i in range(16):
    product = VR[vj].B[i] * VR[vk].B[i]
    VR[vd].B[i] -= product[7:0]
```

VMSUB.H:

```
for i in range(8):
    product = VR[vj].H[i] * VR[vk].H[i]
    VR[vd].H[i] -= product[15:0]
```

VMSUB.W:

```
for i in range(4):
    product = VR[vj].W[i] * VR[vk].W[i]
    VR[vd].W[i] -= product[31:0]
```

VMSUB.D:

```
for i in range(2):
    product = VR[vj].D[i] * VR[vk].D[i]
    VR[vd].D[i] -= product[63:0]
```

XVMSUB.B:

```
for i in range(32):
    product = XR[xj].B[i] * XR[xk].B[i]
    XR[xd].B[i] -= product[7:0]
```

XVMSUB.H:

```
for i in range(16):
    product = XR[xj].H[i] * XR[xk].H[i]
    XR[xd].H[i] -= product[15:0]
```

XVMSUB.W:

```
for i in range(8):
    product = XR[xj].W[i] * XR[xk].W[i]
    XR[xd].W[i] -= product[31:0]
```

XVMSUB.D:

```
for i in range(4):
    product = XR[xj].D[i] * XR[xk].D[i]
    XR[xd].D[i] -= product[63:0]
```

1.2.1.16 [X]VMADDW{EV/OD}.{H.B/W.H/D.W/Q.D}[U], [X]VMADDW{EV/OD}.{H.BU.B/W.HU.H/D.WU.W/Q.DU.D}

Instruction format:				
	vmaddwev.h.b	vd, vj, vk	xvmaddwev.h.b	xd, xj, xk
	vmaddwev.w.h	vd, vj, vk	xvmaddwev.w.h	xd, xj, xk
	vmaddwev.d.w	vd, vj, vk	xvmaddwev.d.w	xd, xj, xk
	vmaddwev.q.d	vd, vj, vk	xvmaddwev.q.d	xd, xj, xk
	vmaddwod.h.b	vd, vj, vk	xvmaddwod.h.b	xd, xj, xk
	vmaddwod.w.h	vd, vj, vk	xvmaddwod.w.h	xd, xj, xk
	vmaddwod.d.w	vd, vj, vk	xvmaddwod.d.w	xd, xj, xk
	vmaddwod.q.d	vd, vj, vk	xvmaddwod.q.d	xd, xj, xk
	vmaddwev.h.bu	vd, vj, vk	xvmaddwev.h.bu	xd, xj, xk

vmaddwev.w.hu	vd, vj, vk	xvmaddwev.w.hu	xd, xj, xk
vmaddwev.d.wu	vd, vj, vk	xvmaddwev.d.wu	xd, xj, xk
vmaddwev.q.du	vd, vj, vk	xvmaddwev.q.du	xd, xj, xk
vmaddwod.h.bu	vd, vj, vk	xvmaddwod.h.bu	xd, xj, xk
vmaddwod.w.hu	vd, vj, vk	xvmaddwod.w.hu	xd, xj, xk
vmaddwod.d.wu	vd, vj, vk	xvmaddwod.d.wu	xd, xj, xk
vmaddwod.q.du	vd, vj, vk	xvmaddwod.q.du	xd, xj, xk
vmaddwev.h.bu.b	vd, vj, vk	xvmaddwev.h.bu.b	xd, xj, xk
vmaddwev.w.hu.h	vd, vj, vk	xvmaddwev.w.hu.h	xd, xj, xk
vmaddwev.d.wu.w	vd, vj, vk	xvmaddwev.d.wu.w	xd, xj, xk
vmaddwev.q.du.d	vd, vj, vk	xvmaddwev.q.du.d	xd, xj, xk
vmaddwod.h.bu.b	vd, vj, vk	xvmaddwod.h.bu.b	xd, xj, xk
vmaddwod.w.hu.h	vd, vj, vk	xvmaddwod.w.hu.h	xd, xj, xk
vmaddwod.d.wu.w	vd, vj, vk	xvmaddwod.d.wu.w	xd, xj, xk
vmaddwod.q.du.d	vd, vj, vk	xvmaddwod.q.du.d	xd, xj, xk

The [X]VMADDW{EV/OD}.{H.B/W.H/D.W/Q.D} instruction multiplies the signed integer bytes/half-word/word/double-word elements at even/odd positions in vector registers vj/xj with the corresponding byte/half-word/word/double-word elements at corresponding positions in vector register vk/xk. The double-width product is accumulated into the corresponding half-word/word/double-word/quad-word elements in vector register vd/xk.

VMADDWEV.H.B:

```
for i in range(8):
    product = signed(VR[vj].B[2i]) * signed(VR[vk].B[2i])
    VR[vd].H[i] += product[15:0]
```

VMADDWEV.W.H:

```
for i in range(4):
    product = signed(VR[vj].H[2i]) * signed(VR[vk].H[2i])
    VR[vd].W[i] += product[31:0]
```

VMADDWEV.D.W:

```
for i in range(2):
    product = signed(VR[vj].W[2i]) * signed(VR[vk].W[2i])
    VR[vd].D[i] += product[63:0]
```

VMADDWEV.Q.D:

```
for i in range(1):
    product = signed(VR[vj].D[2i]) * signed(VR[vk].D[2i])
    VR[vd].Q[i] += product[127:0]
```

VMADDWOD.H.B:

```
for i in range(8):
    product = signed(VR[vj].B[2i+1]) * signed(VR[vk].B[2i+1])
    VR[vd].H[i] += product[15:0]
```

VMADDWOD.W.H:

```
for i in range(4):
    product = signed(VR[vj].H[2i+1]) * signed(VR[vk].H[2i+1])
    VR[vd].W[i] += product[31:0]
```

VMADDWOD.D.W:

```
for i in range(2):
    product = signed(VR[vj].W[2i+1]) * signed(VR[vk].W[2i+1])
    VR[vd].D[i] += product[63:0]
```

VMADDWOD.Q.D:

```
for i in range(1):
    product = signed(VR[vj].D[2i+1]) * signed(VR[vk].D[2i+1])
    VR[vd].Q[i] += product[127:0]
```

XVMADDWEV.H.B:

```
for i in range(16):
    product = signed(XR[vj].B[2i]) * signed(XR[vk].B[2i])
    VR[vd].H[i] += product[15:0]
```

XVMADDWEV.W.H:

```
for i in range(8):
    product = signed(XR[vj].H[2i]) * signed(XR[vk].H[2i])
    VR[vd].W[i] += product[31:0]
```

XVMADDWEV.D.W:

```
for i in range(4):
    product = signed(XR[vj].W[2i]) * signed(XR[vk].W[2i])
    VR[vd].D[i] += product[63:0]
```

XVMADDWEV.Q.D:

```
for i in range(2):
    product = signed(XR[vj].D[2i]) * signed(XR[vk].D[2i])
    VR[vd].Q[i] += product[127:0]
```

XVMADDWOD.H.B:

```
for i in range(16):
    product = signed(XR[vj].B[2i+1]) * signed(XR[vk].B[2i+1])
    VR[vd].H[i] += product[15:0]
```

XVMADDWOD.W.H:

```
for i in range(8):
    product = signed(XR[vj].H[2i+1]) * signed(XR[vk].H[2i+1])
    VR[vd].W[i] += product[31:0]
```

XVMADDWOD.D.W:

```
for i in range(4):
    product = signed(XR[vj].W[2i+1]) * signed(XR[vk].W[2i+1])
    VR[vd].D[i] += product[63:0]
```

XVMADDWOD.Q.D:

```
for i in range(2):
    product = signed(XR[vj].D[2i+1]) * signed(XR[vk].D[2i+1])
    VR[vd].Q[i] += product[127:0]
```

The [X]VMADDW{EV/OD}.{H.B/W.H/D.W/Q.D}U instruction multiplies the unsigned integer byte/half-word/word/double-word elements at even/odd positions in vector register vj/xj with the corresponding unsigned integer byte/half-word/word/double-word elements at even/odd positions in vector register vk/xk. The double-width product is accumulated into the corresponding half-word/word/double-word/quad-word elements in vector register vd/xk.

VMADDWEV.H.BU:

```
for i in range(8):
    product = unsigned(VR[vj].B[2i]) * unsigned(VR[vk].B[2i])
    VR[vd].H[i] += product[15:0]
```

VMADDWEV.W.HU:

```
for i in range(4):
    product = unsigned(VR[vj].H[2i]) * unsigned(VR[vk].H[2i])
    VR[vd].W[i] += product[31:0]
```

VMADDWEV.D.WU:

```
for i in range(2):
    product = unsigned(VR[vj].W[2i]) * unsigned(VR[vk].W[2i])
    VR[vd].D[i] += product[63:0]
```

VMADDWEV.Q.DU:

```
for i in range(1):
    product = unsigned(VR[vj].D[2i]) * unsigned(VR[vk].D[2i])
    VR[vd].Q[i] += product[127:0]
```

VMADDWOD.H.BU:

```
for i in range(8):
    product = unsigned(VR[vj].B[2i+1]) * unsigned(VR[vk].B[2i+1])
    VR[vd].H[i] += product[15:0]
```

VMADDWOD.W.HU:

```
for i in range(4):
    product = unsigned(VR[vj].H[2i+1]) * unsigned(VR[vk].H[2i+1])
    VR[vd].W[i] += product[31:0]
```

VMADDWOD.D.WU:

```
for i in range(2):
    product = unsigned(VR[vj].W[2i+1]) * unsigned(VR[vk].W[2i+1])
    VR[vd].D[i] += product[63:0]
```

VMADDWOD.Q.DU:

```
for i in range(1):
    product = unsigned(VR[vj].D[2i+1]) * unsigned(VR[vk].D[2i+1])
    VR[vd].Q[i] += product[127:0]
```

XVMADDWEV.H.BU:

```
for i in range(16):
    product = unsigned(XR[vj].B[2i]) * unsigned(XR[vk].B[2i])
    VR[vd].H[i] += product[15:0]
```

XVMADDWEV.W.HU:

```
for i in range(8):
    product = unsigned(XR[vj].H[2i]) * unsigned(XR[vk].H[2i])
    VR[vd].W[i] += product[31:0]
```

XVMADDWEV.D.WU:

```
for i in range(4):
    product = unsigned(XR[vj].W[2i]) * unsigned(XR[vk].W[2i])
    VR[vd].D[i] += product[63:0]
```

XVMADDWEV.Q.DU:

```
for i in range(2):
    product = unsigned(XR[vj].D[2i]) * unsigned(XR[vk].D[2i])
    VR[vd].Q[i] += product[127:0]
```

XVMADDWOD.H.BU:

```
for i in range(16):
    product = unsigned(XR[vj].B[2i+1]) * unsigned(XR[vk].B[2i+1])
    VR[vd].H[i] += product[15:0]
```

XVMADDWOD.W.HU:

```
for i in range(8):
    product = unsigned(XR[vj].H[2i+1]) * unsigned(XR[vk].H[2i+1])
    VR[vd].W[i] += product[31:0]
```

XVMADDWOD.D.WU:

```
for i in range(4):
    product = unsigned(XR[vj].W[2i+1]) * unsigned(XR[vk].W[2i+1])
    VR[vd].D[i] += product[63:0]
```

XVMADDWOD.Q.DU:

```
for i in range(2):
    product = unsigned(XR[vj].D[2i+1]) * unsigned(XR[vk].D[2i+1])
    VR[vd].Q[i] += product[127:0]
```

The [X]VMADDW{EV/OD}.{H.BU.B/W.HU.H/D.WU.W/Q.DU.D} instructions multiply the unsigned integer byte/half-word/word/double-word elements at even/odd positions in vector register vj/xj with the corresponding signed integer byte/half-word/word/double-word elements at even/odd positions in vector register vk/xk. The double-width product is accumulated into the corresponding half-word/word/double-word/quad-word element positions in vector register vd/xk.

VMADDWEV.H.BU.B:

```
for i in range(8):
    product = unsigned(VR[vj].B[2i]) * signed(VR[vk].B[2i])
    VR[vd].H[i] += product[15:0]
```

VMADDWEV.W.HU.H:

```
for i in range(4):
    product = unsigned(VR[vj].H[2i]) * signed(VR[vk].H[2i])
    VR[vd].W[i] += product[31:0]
```

VMADDWEV.D.WU.W:

```
for i in range(2):
    product = unsigned(VR[vj].W[2i]) * signed(VR[vk].W[2i])
    VR[vd].D[i] += product[63:0]
```

VMADDWEV.Q.DU.D:

```
for i in range(1):
    product = unsigned(VR[vj].D[2i]) * signed(VR[vk].D[2i])
    VR[vd].Q[i] += product[127:0]
```

VMADDWOD.H.BU.B:

```
for i in range(8):
    product = unsigned(VR[vj].B[2i+1]) * signed(VR[vk].B[2i+1])
    VR[vd].H[i] += product[15:0]
```

VMADDWOD.W.HU.H:

```
for i in range(4):
    product = unsigned(VR[vj].H[2i+1]) * signed(VR[vk].H[2i+1])
    VR[vd].W[i] += product[31:0]
```

VMADDWOD.D.WU.W:

```
for i in range(2):
    product = unsigned(VR[vj].W[2i+1]) * signed(VR[vk].W[2i+1])
    VR[vd].D[i] += product[63:0]
```

VMADDWOD.Q.DU.D:

```
for i in range(1):
    product = unsigned(VR[vj].D[2i+1]) * signed(VR[vk].D[2i+1])
    VR[vd].Q[i] += product[127:0]
```

XVMADDWEV.H.BU.B:

```
for i in range(16) :
    product = unsigned(XR[vj].B[2i]) * signed(XR[vk].B[2i])
    VR[vd].H[i] += product[15:0]
```

XVMADDWEV.W.HU.H:

```
for i in range(8):
    product = unsigned(XR[vj].H[2i]) * signed(XR[vk].H[2i])
    VR[vd].W[i] += product[31:0]
```

XVMADDWEV.D.WU.W:

```
for i in range(4):
    product = unsigned(XR[vj].W[2i]) * signed(XR[vk].W[2i])
    VR[vd].D[i] += product[63:0]
```

XVMADDWEV.Q.DU.D:

```
for i in range(2):
    product = unsigned(XR[vj].D[2i]) * signed(XR[vk].D[2i])
    VR[vd].Q[i] += product[127:0]
```

XVMADDWOD.H.BU.B:

```
for i in range(16):
    product = unsigned(XR[vj].B[2i+1]) * signed(XR[vk].B[2i+1])
    VR[vd].H[i] += product[15:0]
```

XVMADDWOD.W.HU.H:

```
for i in range(8):
    product = unsigned(XR[vj].H[2i+1]) * signed(XR[vk].H[2i+1])
    VR[vd].W[i] += product[31:0]
```

XVMADDWOD.D.WU.W:

```
for i in range(4):
    product = unsigned(XR[vj].W[2i+1]) * signed(XR[vk].W[2i+1])
    VR[vd].D[i] += product[63:0]
```

XVMADDWOD.Q.DU.D:

```
for i in range(2):
    product = unsigned(XR[vj].D[2i+1]) * signed(XR[vk].D[2i+1])
    VR[vd].Q[i] += product[127:0]
```

1.2.1.17 [X]VDIV.{B/H/W/D}[U], [X]VMOD.{B/H/W/D}[U]

Instruction Format:				
	vdiv.b	vd, vj, vk	xvdiv.b	xd, xj, xk
	vdiv.h	vd, vj, vk	xvdiv.h	xd, xj, xk
	vdiv.w	vd, vj, vk	xvdiv.w	xd, xj, xk
	vdiv.d	vd, vj, vk	xvdiv.d	xd, xj, xk
	vdiv.bu	vd, vj, vk	xvdiv.bu	xd, xj, xk
	vdiv.hu	vd, vj, vk	xvdiv.hu	xd, xj, xk
	vdiv.wu	vd, vj, vk	xvdiv.wu	xd, xj, xk
	vdiv.du	vd, vj, vk	xvdiv.du	xd, xj, xk
	vmod.b	vd, vj, vk	xvmod.b	xd, xj, xk
	vmod.h	vd, vj, vk	xvmod.h	xd, xj, xk
	vmod.w	vd, vj, vk	xvmod.w	xd, xj, xk
	vmod.d	vd, vj, vk	xvmod.d	xd, xj, xk
	vmod.bu	vd, vj, vk	xvmod.bu	xd, xj, xk
	vmod.hu	vd, vj, vk	xvmod.hu	xd, xj, xk
	vmod.wu	vd, vj, vk	xvmod.wu	xd, xj, xk
	vmod.du	vd, vj, vk	xvmod.du	xd, xj, xk

The [X]VDIV.{B/H/W/D} instruction divides each signed integer byte/half-word/word/double-word element of vector register vj/xj by the signed integer byte/half-word/word/double-word element at the corresponding position in vector register vk/xk. The quotient is written into the byte/half-word/word/double-word elements at the corresponding position in vector register vd/xd.

VDIV.B:

```
for i in range(16) :
    VR[vd].B[i] = signed(VR[vj].B[i]) / signed(VR[vk].B[i])
```

VDIV.H:

```
for i in range(8):
    VR[vd].H[i] = signed(VR[vj].H[i]) / signed(VR[vk].H[i])
```

VDIV.W:

```
for i in range(4):
    VR[vd].W[i] = signed(VR[vj].W[i]) / signed(VR[vk].W[i])
```

VDIV.D:

```
for i in range(2):
    VR[vd].D[i] = signed(VR[vj].D[i]) / signed(VR[vk].D[i])
```

XVDIV.B:

```
for i in range(32) :
    XR[xd].B[i] = signed(XR[xj].B[i]) / signed(XR[xk].B[i])
```

XVDIV.H:

```
for i in range(16):
    XR[xd].H[i] = signed(XR[xj].H[i]) / signed(XR[xk].H[i])
```

XVDIV.W:

```
for i in range(8):
    XR[xd].W[i] = signed(XR[xj].W[i]) / signed(XR[xk].W[i])
```

XVDIV.D:

```
for i in range(4):
    XR[xd].D[i] = signed(XR[xj].D[i]) / signed(XR[xk].D[i])
```

The [X]VDIV.{B/H/W/D}U instruction divides each unsigned integer byte/half-word/word/double-word element of vector register vj/xj by the unsigned integer byte/half-word/word/double-word element at the corresponding position in vector register vk/xk. The quotient is written into the byte/half-word/word/double-word elements at the corresponding position in vector register vd/xd.

VDIV.BU:

```
for i in range(16) :
    VR[vd].B[i] = unsigned(VR[vj].B[i]) / unsigned(VR[vk].B[i])
```

VDIV.HU:

```
for i in range(8):
    VR[vd].H[i] = unsigned(VR[vj].H[i]) / unsigned(VR[vk].H[i])
```

VDIV.WU:

```
for i in range(4):
    VR[vd].W[i] = unsigned(VR[vj].W[i]) / unsigned(VR[vk].W[i])
```

VDIV.DU:

```
for i in range(2):
    VR[vd].D[i] = unsigned(VR[vj].D[i]) / unsigned(VR[vk].D[i])
```

XVDIV.BU:

```
for i in range(32) :
    XR[xj].B[i] = unsigned(XR[xk].B[i]) / unsigned(XR[xj].B[i])
```

XVDIV.HU:

```
for i in range(16):
    XR[xd].H[i] = unsigned(XR[xj].H[i]) / unsigned(XR[xk].H[i])
```

XVDIV.WU:

```
for i in range(8):
    XR[xd].W[i] = unsigned(XR[xj].W[i]) / unsigned(XR[xk].W[i])
```

XVDIV.DU:

```
for i in range(4):
    XR[xd].D[i] = unsigned(XR[xj].D[i]) / unsigned(XR[xk].D[i])
```

The [X]VMOD.{B/H/W/D} instruction divides each signed integer byte/half-word/word/double-word element of vector register vj/xj by the signed integer byte/half-word/word/double-word element at the corresponding position in vector register vk/xk. The remainder is written into the byte/half-word/word/double-word elements at the corresponding position in vector register vd/xd.

VMOD.B:

```
for i in range(16) :
    VR[vd].B[i] = signed(VR[vj].B[i]) % signed(VR[vk].B[i])
```

VMOD.H:

```
for i in range(8):
    VR[vd].H[i] = signed(VR[vj].H[i]) % signed(VR[vk].H[i])
```

VMOD.W:

```
for i in range(4):
    VR[vd].W[i] = signed(VR[vj].W[i]) % signed(VR[vk].W[i])
```

VMOD.D:

```
for i in range(2):
    VR[vd].D[i] = signed(VR[vj].D[i]) % signed(VR[vk].D[i])
```

XVMOD.B:

```
for i in range(32) :
    XR[xd].B[i] = signed(XR[xj].B[i]) % signed(XR[xk].B[i])
```

XVMOD.H:

```
for i in range(16):
    XR[xd].H[i] = signed(XR[xj].H[i]) % signed(XR[xk].H[i])
```

XVMOD.W:

```
for i in range(8):
    XR[xd].W[i] = signed(XR[xj].W[i]) % signed(XR[xk].W[i])
```

XVMOD.D:

```
for i in range(4):
    XR[xd].D[i] = signed(XR[xj].D[i]) % signed(XR[xk].D[i])
```

The [X]VMOD.{B/H/W/D}U instruction divides each unsigned integer byte/half-word/word/double-word element in vector register vj/xj by the unsigned integer byte/half-word/word/double-word element at the corresponding position in vector register vk/xk. The remainder is written into the byte/half-word/word/double-word elements at the corresponding position in vector register vd/xd.

VMOD.BU:

```
for i in range(16) :
    VR[vd].B[i] = unsigned(VR[vj].B[i]) % unsigned(VR[vk].B[i])
```

VMOD.HU:

```
for i in range(8):
    VR[vd].H[i] = unsigned(VR[vj].H[i]) % unsigned(VR[vk].H[i])
```

VMOD.WU:

```
for i in range(4):
    VR[vd].W[i] = unsigned(VR[vj].W[i]) % unsigned(VR[vk].W[i])
```

VMOD.DU:

```
for i in range(2):
    VR[vd].D[i] = unsigned(VR[vj].D[i]) % unsigned(VR[vk].D[i])
```

XVMOD.BU:

```
for i in range(32) :
    XR[xd].B[i] = unsigned(XR[xj].B[i]) % unsigned(XR[xk].B[i])
```

XVMOD.HU:

```
for i in range(16):
    XR[xd].H[i] = unsigned(XR[xj].H[i]) % unsigned(XR[xk].H[i])
```

XVMOD.WU:

```
for i in range(8):
    XR[xd].W[i] = unsigned(XR[xj].W[i]) % unsigned(XR[xk].W[i])
```

XVMOD.DU:

```
for i in range(4):
    XR[xd].D[i] = unsigned(XR[xj].D[i]) % unsigned(XR[xk].D[i])
```

1.2.1.18 [X]VSAT.{B/H/W/D}[U]

Instruction Format:				
	vsat.b	vd, vj, ui3	xvsat.b	xd, xj, ui3
	vsat.h	vd, vj, ui4	xvsat.h	xd, xj, ui4
	vsat.w	vd, vj, ui5	xvsat.w	xd, xj, ui5
	vsat.d	vd, vj, ui6	xvsat.d	xd, xj, ui6
	vsat.bu	vd, vj, ui3	xvsat.bu	xd, xj, ui3
	vsat.hu	vd, vj, ui4	xvsat.hu	xd, xj, ui4
	vsat.wu	vd, vj, ui5	xvsat.wu	xd, xj, ui5
	vsat.du	vd, vj, ui6	xvsat.du	xd, xj, ui6

The [X]VSAT.{B/H/W/D} instruction performs signed saturation arithmetic on the byte/half-word/word/double-word elements in the vector registers vj/xj to a specified number of significant bits. The number of significant bits for saturation arithmetic is determined by the immediate operand ui3/ui4/ui5/ui6. The result is written to the corresponding byte/half-word/word/double-word elements in vector register vd/xd.

VSAT.B:

```
for i in range(16) :
    VR[vd].B[i] = Sat_S(VR[vj].B[i], ui3)
```

VSAT.H:

```
for i in range(8):
    VR[vd].H[i] = Sat_S(VR[vj].H[i], ui4)
```

VSAT.W:

```
for i in range(16):
    VR[vd].W[i] = Sat_S(VR[vj].W[i], ui5)
```

VSAT.D:

```
for i in range(16):
    VR[vd].D[i] = Sat_S(VR[vj].D[i], ui6)
```

XVSAT.B:

```
for i in range(16):
    XR[x_j].B[i] = Sat_S(XR[xj].B[i], ui3)
```

XVSAT.H:

```
for i in range(8):
    XR[xd].H[i] = Sat_S(XR[xj].H[i], ui4)
```

XVSAT.W:

```
for i in range(16):
    XR[xd].W[i] = Sat_S(XR[xj].W[i], ui5)
```

XVSAT.D:

```
for i in range(16):
    XR[xd].D[i] = Sat_S(XR[xj].D[i], ui6)
```

The [X]VSAT.{B/H/W/D}U instruction performs unsigned saturation arithmetic on the byte/half-word/word/double-word elements in the vector registers vj/xj to a specified number of significant bits. The number of significant bits for saturation arithmetic is determined by the immediate operand ui3/ui4/ui5/ui6. The result is written to the corresponding byte/half-word/word/double-word elements in vector register vd/xd.

VSAT.BU:

```
for i in range(16) :
    VR[vd].B[i] = Sat_U(VR[vj].B[i], ui3)
```

VSAT.HU:

```
for i in range(8):
    VR[vd].H[i] = Sat_U(VR[vj].H[i], ui4)
```

VSAT.WU:

```
for i in range(16):
    VR[vd].W[i] = Sat_U(VR[vj].W[i], ui5)
```

VSAT.DU:

```
for i in range(16):
    VR[vd].D[i] = Sat_U(VR[vj].D[i], ui6)
```

XVSAT.BU:

```
for i in range(16):
    XR[xd].B[i] = Sat_U(XR[xj].B[i], ui3)
```

XVSAT.HU:

```
for i in range(8) :
    XR[xd].H[i] = Sat_U(XR[xj].H[i], ui4)
```

XVSAT.WU:

```
for i in range(16):
    XR[xd].W[i] = Sat_U(XR[xj].W[i], ui5)
```

XVSAT.DU:

```
for i in range(16):
    XR[xd].D[i] = Sat_U(XR[xj].D[i], ui6)
```

1.2.1.19 [X]VEXTH.H.B, [X]VEXTH.W.H, [X]VEXTH.D.W, [X]VEXTH.Q.D, [X]VEXTH.HU.BU, [X]VEXTH.WU.HU, [X]VEXTH.DU.WU, [X]VEXTH.QU.DU

Instruction Format:				
	vexth.h.b	vd, vj	xvexth.h.b	xd, xj
	vexth.w.h	vd, vj	xvexth.w.h	xd, xj
	vexth.d.w	vd, vj	xvexth.d.w	xd, xj
	vexth.q.d	vd, vj	xvexth.q.d	xd, xj
	vexth.hu.bu	vd, vj	xvexth.hu.bu	xd, xj
	vexth.wu.hu	vd, vj	xvexth.wu.hu	xd, xj
	vexth.du.wu	vd, vj	xvexth.du.wu	xd, xj
	vexth.qu.du	vd, vj	xvexth.qu.du	xd, xj

The VEXTH.H.B instruction sign-extends each byte element from the upper half of vector register vj into a half-word element and writes them into vector register vd. The VEXTH.W.H instruction sign-extends each half-word element from the upper half of vector register vj into a word element and writes them into vector register vd. The VEXTH.D.W instruction sign-extends each word element from the upper half of vector register vj into a double-word element and writes them into vector register vd. The VEXTH.Q.D instruction sign-extends each double-word element from the upper half of vector register vj into a quad-word element and writes them into vector register vd.

VEXTH.H.B:

```
for i in range(8):
    VR[vd].H[i] = SignExtend(VR[vj].B[i+8], 16)
```

VEXTH.W.H:

```
for i in range(4):
    VR[vd].W[i] = SignExtend(VR[vj].H[i+4], 32)
```

VEXTH.D.W:

```
for i in range(2):
    VR[vd].D[i] = SignExtend(VR[vj].W[i+2], 64)
```

VEXTH.Q.D:

```
VR[vd].Q[0] = SignExtend(VR[vj].D[1], 128)
```

The VEXTH.HU.BU instruction zero-extends each byte element from the upper half of vector register vj into a half-word element and writes them into vector register vd. The VEXTH.WU.HU instruction zero-extends each half-word element from the upper half of vector register vj into a word element and writes them into vector register vd. The VEXTH.DU.WU instruction zero-extends each word element from the upper half of vector register vj into a double-word element and writes them into vector register vd. The VEXTH.QU.DU instruction zero-extends each double-word element from the upper half of vector register vj into a quad-word element and writes them into vector register vd.

VEXTH.HU.BU:

```
for i in range(8):
    VR[vd].H[i] = ZeroExtend(VR[vj].B[i+8], 16)
```

VEXTH.WU.HU:

```
for i in range(4):
    VR[vd].W[i] = ZeroExtend(VR[vj].H[i+4], 32)
```

VEXTH.DU.WU:

```
for i in range(2):
    VR[vd].D[i] = ZeroExtend(VR[vj].W[i+2], 64)
```

VEXTH.QU.DU:

```
VR[vd].Q[0] = ZeroExtend(VR[vj].D[1], 128)
```

The XVEXTH.H.B instruction sign-extends each byte element from the upper part of each 128 bits of vector register xj into a half-word element and writes them into the corresponding 128 bits of vector register xd. The XVEXTH.W.H instruction sign-extends each half-word element from the upper part of each 128 bits of vector register xj into a word element and writes them into the corresponding 128 bits of vector register xd. The XVEXTH.D.W instruction sign-extends each word element from the upper part of each 128 bits of vector register xj into a double-word element and writes them into the corresponding 128 bits of vector register xd. The XVEXTH.Q.D instruction sign-extends each double-word element from the upper part of each 128 bits of vector register xj into a quad-word element and writes them into the corresponding 128 bits of vector register xd.

XVEXTH.H.B:

```
for i in range(8):
    XR[xd].H[i] = SignExtend(XR[xj].B[i+8], 16)
    XR[xd].H[i+8] = SignExtend(XR[xj].B[i+24], 16)
```

XVEXTH.W.H:

```
for i in range(4):
    XR[xd].W[i] = SignExtend(XR[xj].H[i+4], 32)
    XR[xd].W[i+4] = SignExtend(XR[xj].H[i+12], 32)
```

XVEXTH.D.W:

```
for i in range(2):
    XR[xd].D[i] = SignExtend(XR[xj].W[i+2], 64)
    XR[xd].D[i+2] = SignExtend(XR[xj].W[i+6], 64)
```

XVEXTH.Q.D:

```
XR[xd].Q[0] = SignExtend(XR[xj].D[1], 128)
XR[xd].Q[1] = SignExtend(XR[xj].D[3], 128)
```

The XVEXTH.HU.BU instruction zero-extends each byte element from the upper part of each 128 bits of vector register xj into a half-word element and writes them into the corresponding 128 bits of vector register xd. The XVEXTH.WU.HU instruction zero-extends each half-word element from the upper part of each 128 bits of vector register xj into a word element and writes them into the corresponding 128 bits of vector register xd. The XVEXTH.DU.WU instruction zero-extends each word element from the upper part of each 128 bits of vector register xj into a double-word element and writes them into the corresponding 128 bits of vector register xd. The XVEXTH.QU.DU instruction zero-extends each double-word element from the upper part of each 128 bits of vector register xj into a quad-word element and writes them into the corresponding 128 bits of vector register xd.

XVEXTH.HU.BU:

```
for i in range(8):
    XR[xd].H[i] = ZeroExtend(XR[xj].B[i+8], 16)
    XR[xd].H[i+8] = ZeroExtend(XR[xj].B[i+24], 16)
```

XVEXTH.WU.HU:

```
for i in range(4):
    XR[xd].W[i] = ZeroExtend(XR[xj].H[i+4], 32)
    XR[xd].W[i+4] = ZeroExtend(XR[xj].H[i+12], 32)
```

XVEXTH.DU.WU:

```
for i in range(2):
    XR[xd].D[i] = ZeroExtend(XR[xj].W[i+2], 64)
    XR[xd].D[i+2] = ZeroExtend(XR[xj].W[i+6], 64)
```

XVEXTH.QU.DU:

```
XR[xd].Q[0] = ZeroExtend(XR[xj].D[1], 128)
XR[xd].Q[1] = ZeroExtend(XR[xj].D[3], 128)
```

1.2.1.20 VEXT2XV.{H/W/D}.B, VEXT2XV.{HU/WU/DU}.BU, VEXT2XV.{W/D}.H, VEXT2XV.{WU/DU}.HU, VEXT2XV.D.W, VEXT2XV.DU.WU

Instruction Format:				
	vext2xv.h.b	xd, xj	vext2xv.hu.bu	xd, xj
	vext2xv.w.b	xd, xj	vext2xv.wu.bu	xd, xj
	vext2xv.d.b	xd, xj	vext2xv.du.bu	xd, xj
	vext2xv.w.h	xd, xj	vext2xv.wu.hu	xd, xj
	vext2xv.d.h	xd, xj	vext2xv.du.hu	xd, xj
	vext2xv.d.w	xd, xj	vext2xv.du.wu	xd, xj

The VEXT2XV.{H/W/D}.B instruction sign-extends each byte element in the lowest part of vector register xj to a half-word/word/double-word element and writes it into vector register xd.

VEXT2XV.H.B:

```
for i in range(16):
    XR[xd].H[i] = SignExtend(XR[xj].B[i], 16)
```

VEXT2XV.W.B:

```
for i in range(8):
    XR[xd].W[i] = SignExtend(XR[xj].B[i], 32)
```

VEXT2XV.D.B:

```
for i in range(4):
    XR[xd].D[i] = SignExtend(XR[xj].B[i], 64)
```

The VEXT2XV.{W/D}.H instruction sign-extends each half-word element in the lowest part of vector register xj to a word/double-word element and writes them into vector register xd.

VEXT2XV.W.H:

```
for i in range(8):
    XR[xd].W[i] = SignExtend(XR[xj].H[i], 32)
```

VEXT2XV.D.H:

```
for i in range(4):
    XR[xd].D[i] = SignExtend(XR[xj].H[i], 64)
```

The VEXT2XV.D.W instruction sign-extends each word element in the lowest part of vector register xj to a double-word element and writes it into vector register xd.

VEXT2XV.D.W:

```
for i in range(4):
    XR[xd].D[i] = SignExtend(XR[xj].W[i], 64)
```

The VEXT2XV.{HU/WU/DU}.BU instruction zero-extends each byte element in the lowest part of vector register xj to a half-word/word/double-word element and writes it into vector register xd.

VEXT2XV.HU.BU:

```
for i in range(16):
    XR[xd].H[i] = ZeroExtend(XR[xj].B[i], 16)
```

VEXT2XV.WU.BU:

```
for i in range(8):
    XR[xd].W[i] = ZeroExtend(XR[xj].B[i], 32)
```

VEXT2XV.DU.BU:

```
for i in range(4):
    XR[xd].D[i] = ZeroExtend(XR[xj].B[i], 64)
```

The VEXT2XV.{WU/DU}.HU instruction zero-extends each half-word element in the lowest part of vector register xj to a word/double-word element and writes it into vector register xd.

VEXT2XV.WU.HU:

```
for i in range(8):
    XR[xd].W[i] = ZeroExtend(XR[xj].H[i], 32)
```

VEXT2XV.DU.HU:

```
for i in range(4):
    XR[xd].D[i] = ZeroExtend(XR[xj].H[i], 64)
```

The VEXT2XV.DU.WU instruction zero-extends each word element in the lowest part of vector register xj to a double-word element and writes it into vector register xd.

VEXT2XV.DU.WU:

```
for i in range(4):
    XR[xd].D[i] = ZeroExtend(XR[xj].W[i], 64)
```

1.2.1.21 [X]VSIGNCOV.{B/H/W/D}

Instruction Format:	vsigncov.b	vd, vj, vk		xvsigncov.b	xd, xj, xk
	vsigncov.h	vd, vj, vk		xvsigncov.h	xd, xj, xk
	vsigncov.w	vd, vj, vk		xvsigncov.w	xd, xj, xk
	vsigncov.d	vd, vj, vk		xvsigncov.d	xd, xj, xk

The [X]VSIGNCOV.{B/H/W/D} instruction processes each byte/half-word/word/double-word element in vector register vj/xj base on the value (0, positive, or negative). The result is written to the corresponding byte/half-word/word/double-word elements in vector register vd/xd.

VSIGNCOV.B:

```
for i in range(16):
    VR[vd].B[i] = Sign_Convert(VR[vj].B[i], VR[vk].B[i])
```

VSIGNCOV.H:

```
for i in range(8):
    VR[vd].H[i] = Sign_Convert(VR[vj].H[i], VR[vk].H[i])
```

VSIGNCOV.W:

```
for i in range(4):
    VR[vd].W[i] = Sign_Convert(VR[vj].W[i], VR[vk].W[i])
```

VSIGNCOV.D:

```
for i in range(2):
    VR[vd].D[i] = Sign_Convert(VR[vj].D[i], VR[vk].D[i])
```

XVSIGNCOV.B:

```
for i in range(32):
    XR[xd].B[i] = Sign_Convert(XR[xj].B[i], XR[xk].B[i])
```

XVSIGNCOV.H:

```
for i in range(16):
    XR[xd].H[i] = Sign_Convert(XR[xj].H[i], XR[xk].H[i])
```

XVSIGNCOV.W:

```
for i in range(8):
    XR[xd].W[i] = Sign_Convert(XR[xj].W[i], XR[xk].W[i])
```

XVSIGNCOV.D:

```
for i in range(4):
    XR[xd].D[i] = Sign_Convert(XR[xj].D[i], XR[xk].D[i])
```

1.2.1.22 [X]VMSKLTZ.{B/H/W/D}

Instruction Format:	vmskltz.b	vd, vj	xvmskltz.b	xd, xj
	vmskltz.h	vd, vj	xvmskltz.h	xd, xj
	vmskltz.w	vd, vj	xvmskltz.w	xd, xj
	vmskltz.d	vd, vj	xvmskltz.d	xd, xj

The [X]VMSKLTZ.{B/H/W/D} instruction writes each signed integer byte/half-word/word/double-word elements in vector registers vj/xj that are less than 0 (i.e., the sign bits of each element) into the lowest bit of each 128 bits in vector registers vd/xd. The remaining bits in every 128 bits are filled with zeros.

VMSKLTZ.B:

```
VR[vd] = 0
for i in range(16):
    VR[vd].V[i] = VR[vj].B[i] < 0 ? 1'b1 : 1'b0
```

VMSKLTZ.H:

```
VR[vd] = 0
for i in range(8):
    VR[vd].V[i] = VR[vj].H[i] < 0 ? 1'b1 : 1'b0
```

VMSKLTZ.W:

```
VR[vd] = 0
for i in range(4):
    VR[vd].V[i] = VR[vj].W[i] < 0 ? 1'b1 : 1'b0
```

VMSKLTZ.D:

```
VR[vd] = 0
for i in range(2):
    VR[vd].V[i] = VR[vj].D[i] < 0 ? 1'b1 : 1'b0
```

XVMSKLTZ.B:

```
XR[xd] = 0
for i in range(16):
    XR[xd].V[i]     = XR[xj].B[i]    < 0 ? 1'b1 : 1'b0
    XR[xd].V[128+i] = XR[xj].B[i+16] < 0 ? 1'b1 : 1'b0
```

XVMSKLTZ.H:

```
XR[xd] = 0
for i in range(8):
    XR[xd].V[i]     = XR[xj].H[i]    < 0 ? 1'b1 : 1'b0
    XR[xd].V[128+i] = XR[xj].H[i+8]  < 0 ? 1'b1 : 1'b0
```

XVMSKLTZ.W:

```
XR[xd] = 0
for i in range(4):
    XR[xd].V[i]     = XR[xj].W[i]    < 0 ? 1'b1 : 1'b0
    XR[xd].V[128+i] = XR[xj].W[i+4]  < 0 ? 1'b1 : 1'b0
```

XVMSKLTZ.D:

```
XR[xd] = 0
for i in range(2):
    XR[xd].V[i]     = XR[xj].D[i]    < 0 ? 1'b1 : 1'b0
    XR[xd].V[128+i] = XR[xj].D[i+2]  < 0 ? 1'b1 : 1'b0
```

1.2.1.23 [X]VMSKGEZ.B, [X]VMSKNZ.B

Instruction Format:

vmskgez.b	vd, vj		xvmskgez.b	xd, xj
vmsknz.b	vd, vj		xvmsknz.b	xd, xj

The [X]VMSKGEZ.B instruction writes the result of whether each signed integer byte in vector register vj/xj is greater than or equal to 0 into the lowest bit of each 128 bits in vector register vd/xd. The remaining bits of every 128 bits are filled with zeros.

VMSKGEZ.B:

```
VR[vd] = 0
for i in range(16):
    VR[vd].V[i] = VR[vj].B[i] >= 0 ? 1'b1 : 1'b0
```

XVMSKGEZ.B:

```
XR[xd] = 0
for i in range(16):
    XR[xd].V[i]     = XR[xj].B[i]    >= 0 ? 1'b1 : 1'b0
    XR[xd].V[128+i] = XR[xj].B[i+16] >= 0 ? 1'b1 : 1'b0
```

The [X]VMSKNZ.B instruction writes the result of whether each signed integer byte in vector register vj/xj is not equal to 0 into the lowest bit of each 128 bits in vector register vd/xd. The remaining bits of each 128 bits are filled with zeros.

VMSKNZ.B:

```
VR[vd] = 0
for i in range(16):
    VR[vd].V[i] = VR[vj].B[i] != 0 ? 1'b1 : 1'b0
```

XVMSKNZ.B:

```
XR[xd] = 0
for i in range(16):
    XR[xd].V[i]     = XR[xj].B[i]    != 0 ? 1'b1 : 1'b0
    XR[xd].V[128+i] = XR[xj].B[i+16] != 0 ? 1'b1 : 1'b0
```

1.2.1.24 [X]VLDI

Instruction Format:

vldi	vd, i13	xvldi	xd, i13

The [X]VLDI instruction loads data into vector registers vd/xd based on the value of i13.

VLDI:

```
for i in range(2):
    VR[vd].D[i] = Set_Value(i13)
```

XVLDI:

```
for i in range(4):
    XR[xd].D[i] = Set_Value(i13)
```

1.2.2 Vector Bitwise Operation Instructions

1.2.2.1 [X]VAND.V, [X]VOR.V, [X]VXOR.V, [X]VNOR.V, [X]VANDN.V, [X]VORN.V

Instruction Format:	vand.v	vd, vj, vk	xvand.v	xd, xj, xk
	vor.v	vd, vj, vk	xvor.v	xd, xj, xk
	vxor.v	vd, vj, vk	xvxor.v	xd, xj, xk
	vnor.v	vd, vj, vk	xvnor.v	xd, xj, xk
	vandn.v	vd, vj, vk	xvandn.v	xd, xj, xk
	vorn.v	vd, vj, vk	xvorn.v	xd, xj, xk

The [X]VAND.V instruction performs a bitwise AND operation between the contents of vector register vj/xj and the contents of vector register vk/xk. The result is written into the vector register vd/xd.

VAND.V:

```
VR[vd] = VR[vj] & VR[vk]
```

XVAND.V:

```
XR[xd] = XR[xj] & XR[xk]
```

The [X]VOR.V instruction performs a bitwise OR operation between the contents of vector registers vj/xj and vector registers vk/xk. The result is written into the vector register vd/xd.

VOR.V:

```
VR[vd] = VR[vj] | VR[vk]
```

XVOR.V:

```
XR[xd] = XR[xj] | XR[xk]
```

The [X]VXOR.V instruction performs a bitwise XOR operation between the contents of vector registers vj/xj and vector registers vk/xk. The result is written into the vector register vd/xd.

VXOR.V:

```
VR[vd] = VR[vj] ^ VR[vk]
```

XVXOR.V:

```
XR[xd] = XR[xj] ^ XR[xk]
```

The [X]VNOR.V instruction performs a bitwise NOR operation between the contents of vector registers vj/xj and vk/xk. The result is written into the vector register vd/xd.

VNOR.V:

```
VR[vd] = ~ (VR[vj] | VR[vk])
```

XVNOR.V:

```
XR[xd] = ~ (XR[xj] | XR[xk])
```

The [X]VANDN.V instruction performs a bitwise AND-NOT operation between the contents of vector registers vj/xj and vk/xk. The result is written into the vector register vd/xd.

VANDN.V:

VR[vd] = ~ VR[vj] & VR[vk]

XVANDN.V:

XR[xd] = ~ XR[xj] & XR[xk]

The [X]VORN.V instruction performs a bitwise OR-NOT operation between the contents of vector registers vj/xj and vector registers vk/xk. The result is written into the vector register vd/xd.

VORN.V:

VR[vd] = VR[vj] | ~ VR[vk]

XVORN.V:

XR[xd] = XR[xj] | ~ XR[xk]

1.2.2.2 [X]VANDI.B, [X]VORI.B, [X]VXORI.B, [X]VNORI.B

Instruction Format:	vandi.b	vd, vj, ui8	xvandi.b	xd, xj, ui8
	vori.b	vd, vj, ui8	xvori.b	xd, xj, ui8
	vxori.b	vd, vj, ui8	xvxori.b	xd, xj, ui8
	vnori.b	vd, vj, ui8	xvnori.b	xd, xj, ui8

The [X]VANDI.B instruction performs a bitwise AND operation between each byte element in vector register vj/xj and immediate value ui8. The result is written into the corresponding byte elements in vector register vd/xd.

VANDI.B:

```
for i in range(16) :
    VR[vd].B[i] = VR[vj].B[i] & ui8
```

XVANDI.B:

```
for i in range(32):
    XR[xd].B[i] = XR[xj].B[i] & ui8
```

The [X]VORI.B instruction performs a bitwise OR operation between each byte element in vector register vj/xj and the immediate value ui8. The result is written into the corresponding byte element in vector register vd/xd.

VORI.B:

```
for i in range(16):
    VR[vd].B[i] = VR[vj].B[i] | ui8
```

XVORI.B:

```
for i in range(32):
    XR[xd].B[i] = XR[xj].B[i] | ui8
```

The [X]VXORI.B instruction performs a bitwise XOR operation between each byte element in vector register vj/xj and the immediate value ui8. The result is written into the byte element at the corresponding position in vector register vd/xd.

VXORI.B:

```
for i in range(16):
    VR[vd].B[i] = VR[vj].B[i] ^ ui8
```

XVXORI.B:

```
for i in range(32):
    XR[xd].B[i] = XR[xj].B[i] ^ ui8
```

The [X]VNORI.B instruction performs a bitwise OR-NOT operation between each byte element in vector register vj/xj and the immediate value ui8. The result is written into the byte element at the corresponding position in vector register vd/xd.

VNORI.B:

```
for i in range(16):
    VR[vd].B[i] = ~ (VR[vj].B[i] | ui8)
```

XVNORI.B:

```
for i in range(32):
    XR[xd].B[i] = ~ (XR[xj].B[i] | ui8)
```

1.2.2.3 [X]VSLL[I].{B/H/W/D}, [X]VSRL[I].{B/H/W/D}, [X]VSRA[I].{B/H/W/D}, [X]VROTR.[I]{B/H/W/D}

Instruction Format:				
	vsll.b	vd, vj, vk	xvsll.b	xd, xj, xk
	vsll.h	vd, vj, vk	xvsll.h	xd, xj, xk
	vsll.w	vd, vj, vk	xvsll.w	xd, xj, xk
	vsll.d	vd, vj, vk	xvsll.d	xd, xj, xk
	vslli.b	vd, vj, ui3	xvslli.b	xd, xj, ui3
	vslli.h	vd, vj, ui4	xvslli.h	xd, xj, ui4
	vslli.w	vd, vj, ui5	xvslli.w	xd, xj, ui5
	vslli.d	vd, vj, ui6	xvslli.d	xd, xj, ui6
	vsrl.b	vd, vj, vk	xvsrl.b	xd, xj, xk
	vsrl.h	vd, vj, vk	xvsrl.h	xd, xj, xk
	vsrl.w	vd, vj, vk	xvsrl.w	xd, xj, xk
	vsrl.d	vd, vj, vk	xvsrl.d	xd, xj, xk
	vsrli.b	vd, vj, ui3	xvsrli.b	xd, xj, ui3
	vsrli.h	vd, vj, ui4	xvsrli.h	xd, xj, ui4
	vsrli.w	vd, vj, ui5	xvsrli.w	xd, xj, ui5
	vsrli.d	vd, vj, ui6	xvsrli.d	xd, xj, ui6
	vsra.b	vd, vj, vk	xvsra.b	xd, xj, xk
	vsra.h	vd, vj, vk	xvsra.h	xd, xj, xk
	vsra.w	vd, vj, vk	xvsra.w	xd, xj, xk
	vsra.d	vd, vj, vk	xvsra.d	xd, xj, xk
	vsrai.b	vd, vj, ui3	xvsrai.b	xd, xj, ui3
	vsrai.h	vd, vj, ui4	xvsrai.h	xd, xj, ui4
	vsrai.w	vd, vj, ui5	xvsrai.w	xd, xj, ui5
	vsrai.d	vd, vj, ui6	xvsrai.d	xd, xj, ui6
	vrotr.b	vd, vj, vk	xvrotr.b	xd, xj, xk
	vrotr.h	vd, vj, vk	xvrotr.h	xd, xj, xk
	vrotr.w	vd,vj, vk	xvrotr.w	xd, xj, xk
	vrotr.d	vd, vj, vk	xvrotr.d	xd, xj, xk
	vrotri.b	vd, vj, ui3	xvrotri.b	xd, xj, ui3
	vrotri.h	vd, vj, ui4	xvrotri.h	xd, xj, ui4
	vrotri.w	vd, vj, ui5	xvrotri.w	xd, xj, ui5
	vrotri.d	vd, vj, ui6	xvrotri.d	xd, xj, ui6

The [X]VSLL.{B/H/W/D} instruction performs a logical left shift on each byte/half-word/word/double-word element in vector register vj/xj. The shift amount is derived from the values of the byte/half-word/word/double-word elements at the corresponding position in vector register vk/xk. The result is written into the byte/half-word/word/double-word elements at the corresponding position in vector register vd/xd.

VSLL.B:

```
for i in range(16):
    VR[vd].B[i] = SLL(VR[vj].B[i], VR[vk].B[i] % 8)
```

VSLL.H:

```
for i in range(8):
    VR[vd].H[i] = SLL(VR[vj].H[i], VR[vk].H[i] % 16)
```

VSLL.W:

```
for i in range(4):
    VR[vd].W[i] = SLL(VR[vj].W[i], VR[vk].W[i] % 32)
```

VSLL.D:

```
for i in range(2):
    VR[vd].D[i] = SLL(VR[vj].D[i], VR[vk].D[i] % 64)
```

XVSLL.B:

```
for i in range(32) :
    XR[xd].B[i] = SLL(XR[xj].B[i], XR[xk].B[i] % 8)
```

XVSLL.H:

```
for i in range(16) :
    XR[xd].H[i] = SLL(XR[xj].H[i], XR[xk].H[i] % 16)
```

XVSLL.W:

```
for i in range(8):
    XR[xd].W[i] = SLL(XR[xj].W[i], XR[xk].W[i] % 32)
```

XVSLL.D:

```
for i in range(4):
    XR[xd].D[i] = SLL(XR[xj].D[i], XR[xk].D[i] % 64)
```

The [X]VSLLI.{B/H/W/D} instruction performs a logical left shift on each byte/half-word/word/double-word element in vector register vj/xj. The shift amount is derived from the immediate value ui3/ui4/ui5/ui6 in the instruction code. The result is written into the corresponding byte/half-word/word/double-word elements in vector register vd/xd.

VSLLI.B:

```
for i in range(16) :
    VR[vd].B[i] = SLL(VR[vj].B[i], ui3)
```

VSLLI.H:

```
for i in range(8):
    VR[vd].H[i] = SLL(VR[vj].H[i], ui4)
```

VSLLI.W:

```
for i in range(4):
    VR[vd].W[i] = SLL(VR[vj].W[i], ui5)
```

VSLLI.D:

```
for i in range(2):
    VR[vd].D[i] = SLL(VR[vj].D[i], ui6)
```

XVSLLI.B:

```
for i in range(32) :
    XR[xd].B[i] = SLL(XR[xj].B[i], ui3)
```

XVSLLI.H:

```
for i in range(16) :
    XR[xd].H[i] = SLL(XR[xj].H[i], ui4)
```

XVSLLI.W:

```
for i in range(8):
    XR[xd].W[i] = SLL(XR[xj].W[i], ui5)
```

XVSLLI.D:

```
for i in range(4):
    XR[xd].D[i] = SLL(XR[xj].D[i], ui6)
```

The [X]VSRL.{B/H/W/D} instruction performs a logical right shift on each byte/half-word/word/double-word element in vector register vj/xj. The shift amount is derived from the values of the byte/half-word/word/double-word elements at the corresponding position in vector register vk/xk. The result is written into the byte/half-word/word/double-word elements at the corresponding position in vector register vd/xd.

VSRL.B:

```
for i in range(16):
    VR[vd].B[i] = SRL(VR[vj].B[i], VR[vk].B[i] % 8)
```

VSRL.H:

```
for i in range(8):
    VR[vd].H[i] = SRL(VR[vj].H[i], VR[vk].H[i] % 16)
```

VSRL.W:

```
for i in range(4):
    VR[vd].W[i] = SRL(VR[vj].W[i], VR[vk].W[i] % 32)
```

VSRL.D:

```
for i in range(2):
    VR[vd].D[i] = SRL(VR[vj].D[i], VR[vk].D[i] % 64)
```

XVSRL.B:

```
for i in range(32):
    XR[xd].B[i] = SRL(XR[xj].B[i], XR[xk].B[i] % 8)
```

XVSRL.H:

```
for i in range(16):
    XR[xd].H[i] = SRL(XR[xj].H[i], XR[xk].H[i] % 16)
```

XVSRL.W:

```
for i in range(8):
    XR[xd].W[i] = SRL(XR[xj].W[i], XR[xk].W[i] % 32)
```

XVSRL.D:

```
for i in range(4):
    XR[xd].D[i] = SRL(XR[xj].D[i], XR[xk].D[i] % 64)
```

The [X]VSRLI.{B/H/W/D} instruction performs a logical right shift on each byte/half-word/word/double-word element within vector register vj/xj. The shift amount is derived from the immediate value ui3/ui4/ui5/ui6 in the instruction code. The result is written into the corresponding byte/half-word/word/double-word elements in vector register vd/xd.

VSRLI.B:

```
for i in range(16) :
    VR[vd].B[i] = SRL(VR[vj].B[i], ui3)
```

VSRLI.H:

```
for i in range(8):
    VR[vd].H[i] = SRL(VR[vj].H[i], ui4)
```

VSRLI.W:

```
for i in range(4):
    VR[vd].W[i] = SRL(VR[vj].W[i], ui5)
```

VSRLI.D:

```
for i in range(2):
    VR[vd].D[i] = SRL(VR[vj].D[i], ui6)
```

XVSRLI.B:

```
for i in range(32) :
    XR[xd].B[i] = SRL(XR[xj].B[i], ui3)
```

XVSRLI.H:

```
for i in range(16):
    XR[xd].H[i] = SRL(XR[xj].H[i], ui4)
```

XVSRLI.W:

```
for i in range(8):
    XR[xd].W[i] = SRL(XR[xj].W[i], ui5)
```

XVSRLI.D:

```
for i in range(4):
    XR[xd].D[i] = SRL(XR[xj].D[i], ui6)
```

The [X]VSRA.{B/H/W/D} instruction performs arithmetic right shifts on each byte/half-word/word/double-word element in vector registers vj/xj. The shift amount is derived from the values of the byte/half-word/word/double-word elements at the corresponding position in vector registers vk/xk. The result is written into the byte/half-word/word/double-word elements at the corresponding position in vector registers vd/xd.

VSRA.B:

```
for i in range(16):
    VR[vd].B[i] = SRA(VR[vj].B[i], VR[vk].B[i] % 8)
```

VSRA.H:

```
for i in range(8):
    VR[vd].H[i] = SRA(VR[vj].H[i], VR[vk].H[i] % 16)
```

VSRA.W:

```
for i in range(4):
    VR[vd].W[i] = SRA(VR[vj].W[i], VR[vk].W[i] % 32)
```

VSRA.D:

```
for i in range(2):
    VR[vd].D[i] = SRA(VR[vj].D[i], VR[vk].D[i] % 64)
```

XVSRA.B:

```
for i in range(32):
    XR[xd].B[i] = SRA(XR[xj].B[i], XR[xk].B[i] % 8)
```

XVSRA.H:

```
for i in range(16):
    XR[xd].H[i] = SRA(XR[xj].H[i], XR[xk].H[i] % 16)
```

XVSRA.W:

```
for i in range(8):
    XR[xd].W[i] = SRA(XR[xj].W[i], XR[xk].W[i] % 32)
```

XVSRA.D:

```
for i in range(4):
    XR[xd].D[i] = SRA(XR[xj].D[i], XR[xk].D[i] % 64)
```

The [X]VSRAI.{B/H/W/D} instruction performs an arithmetic right shift on each byte/half-word/word/double-word element in vector register vj/xj. The shift amount is derived from the immediate value ui3/ui4/ui5/ui6 in the instruction code. The result is written into the corresponding byte/half-word/word/double-word elements in vector register vd/xd.

VSRAI.B:

```
for i in range(16):
    VR[vd].B[i] = SRA(VR[vj].B[i], ui3)
```

VSRAI.H:

```
for i in range(8):
    VR[vd].H[i] = SRA(VR[vj].H[i], ui4)
```

VSRAI.W:

```
for i in range(4):
    VR[vd].W[i] = SRA(VR[vj].W[i], ui5)
```

VSRAI.D:

```
for i in range(2):
    VR[vd].D[i] = SRA(VR[vj].D[i], ui6)
```

XVSRAI.B:

```
for i in range(32) :
    XR[xd].B[i] = SRA(XR[xj].B[i], ui3)
```

XVSRAI.H:

```
for i in range(16):
    XR[xd].H[i] = SRA(XR[xj].H[i], ui4)
```

XVSRAI.W:

```
for i in range(8):
    XR[xd].W[i] = SRA(XR[xj].W[i], ui5)
```

XVSRAI.D:

```
for i in range(4):
    XR[xd].D[i] = SRA(XR[xj].D[i], ui6)
```

The [X]VROTR.{B/H/W/D} instruction performs a cyclic right shift operation on each byte/half-word/word/double-word element in vector register vj/xj. The shift amount is derived from the values of the corresponding byte/half-word/word/double-word elements in vector register vk/xk. The result is written into the corresponding byte/half-word/word/double-word elements of vector register vd/xd.

VROTR.B:

```
for i in range(16) :
    VR[vd].B[i] = ROTR(VR[vj].B[i], VR[vk].B[i] % 8)
```

VROTR.H:

```
for i in range(8):
    VR[vd].H[i] = ROTR(VR[vj].H[i], VR[vk].H[i] % 16)
```

VROTR.W:

```
for i in range(4):
    VR[vd].W[i] = ROTR(VR[vj].W[i], VR[vk].W[i] % 32)
```

VROTR.D:

```
for i in range(2):
    VR[vd].D[i] = ROTR(VR[vj].D[i], VR[vk].D[i] % 64)
```

XVROTR.B:

```
for i in range(32) :
    XR[xd].B[i] = ROTR(XR[xj].B[i], XR[xk].B[i] % 8)
```

XVROTR.H:

```
for i in range(16):
    XR[xd].H[i] = ROTR(XR[xj].H[i], XR[xk].H[i] % 16)
```

XVROTR.W:

```
for i in range(8):
    XR[xd].W[i] = ROTR(XR[xj].W[i], XR[xk].W[i] % 32)
```

XVROTR.D:

```
for i in range(4):
    XR[xd].D[i] = ROTR(XR[xj].D[i], XR[xk].D[i] % 64)
```

The [X]VROTRI.{B/H/W/D} instruction performs a cyclic right shift on each byte/half-word/word/double-word element in vector register vj/xj. The shift amount is derived from the immediate value ui3/ui4/ui5/ui6 in the instruction code. The result is written into the corresponding byte/half-word/word/double-word elements in vector register vd/xd.

VROTRI.B:

```
for i in range(16) :
    VR[vd].B[i] = ROTR(VR[vj].B[i], ui3)
```

VROTRI.H:

```
for i in range(8):
    VR[vd].H[i] = ROTR(VR[vj].H[i], ui4)
```

VROTRI.W:

```
for i in range(4):
    VR[vd].W[i] = ROTR(VR[vj].W[i], ui5)
```

VROTRI.D:

```
for i in range(2):
    VR[vd].D[i] = ROTR(VR[vj].D[i], ui6)
```

XVROTRI.B:

```
for i in range(32):
    XR[xd].B[i] = ROTR(XR[xj].B[i], ui3)
```

XVROTRI.H:

```
for i in range(16):
    XR[xd].H[i] = ROTR(XR[xj].H[i], ui4)
```

XVROTRI.W:

```
for i in range(8):
    XR[xd].W[i] = ROTR(XR[xj].W[i], ui5)
```

XVROTRI.D:

```
for i in range(4):
    XR[xd].D[i] = ROTR(XR[xj].D[i], ui6)
```

1.2.2.4 [X]VSLLWIL.{H.B/W.H/D.W}, [X]VEXTL.Q.D, [X]VSLLWIL.{HU.BU/WU.HU/DU.WU}, [X]VEXTL.QU.DU

Instruction Format:	vsllwil.h.b	vd, vj, ui3	xvsllwil.h.b	xd, xj, ui3
	vsllwil.w.h	vd, vj, ui4	xvsllwil.w.h	xd, xj, ui4
	vsllwil.d.w	vd, vj, ui5	xvsllwil.d.w	xd, xj, ui5
	vextl.q.d	vd, vj	xvextl.q.d	xd, xj
	vsllwil.hu.bu	vd, vj, ui3	xvsllwil.hu.bu	xd, xj, ui3
	vsllwil.wu.hu	vd, vj, ui4	xvsllwil.wu.hu	xd, xj, ui4
	vsllwil.du.wu	vd, vj, ui5	xvsllwil.du.wu	xd, xj, ui5
	vextl.qu.du	vd, vj	xvextl.qu.du	xd, xj

The [X]VSLLWIL.{H.B/W.H/D.W} instruction performs a logical left shift on each byte/half-word/word element in the vector registers vj/xj, after sign-extending to double the width. The shift amount is derived from the immediate operand ui3/ui4/ui5. The result is sequentially written into the half-word/word/double-word elements within every 128 bits of the vector registers vd/xd.

VSLLWIL.H.B:

```
for i in range(8):
    VR[vd].H[i] = SLLW_S(VR[vj].B[i], ui3)
```

VSLLWIL.W.H:

```
for i in range(4):
    VR[vd].W[i] = SLLW_S(VR[vj].H[i], ui4)
```

VSLLWIL.D.W:

```
for i in range(2):
    VR[vd].D[i] = SLLW_S(VR[vj].W[i], ui5)
```

XVSLLWIL.H.B:

```
for i in range(8) :
    XR[xd].H[i] = SLLW_S(XR[xj].B[i], ui3)
    XR[xd].H[i+8] = SLLW_S(XR[xj].B[i+16], ui3)
```

XVSLLWIL.W.H:

```
for i in range(4):
    XR[xd].W[i] = SLLW_S(XR[xj].H[i], ui4)
    XR[xd].W[i+4] = SLLW_S(XR[xj].H[i+8], ui4)
```

XVSLLWIL.D.W:

```
for i in range(2) :
    XR[xd].D[i] = SLLW_S(XR[xj].W[i], ui5)
    XR[xd].D[i+2] = SLLW_S(XR[xj].W[i+4], ui5)
```

The [X]VEXTL.Q.D instruction sign-extends each double-word element in the lower part of each 128 bits of vector register vj/xj. The result is sequentially written into each quad-word elements of every 128 bits of vector register vd/xd.

VEXTL.Q.D:

```
for i in range(1) :
    VR[vd].Q[i] = SignExtend(VR[vj].D[i], 128)
```

XVEXTL.Q.D:

```
for i in range(1) :
    XR[xd].Q[i]= SignExtend(XR[vj].D[i], 128)
    XR[xd].Q[i+1] = SignExtend(XR[vj].D[i+2], 128)
```

The [X]VSLLWIL.{HU.BU/WU.HU/DU.WU} instruction performs a logical left shift operation on each byte/half-word/word element in the lower part of each 128 bits of vector register vj/xj, after zero-extending to double the width. The shift amount is derived from an immediate operand ui3/ui4/ui5. The result is sequentially written into the half-word/word/double-word elements in every 128 bits of vector registers vd/xd.

VSLLWIL.HU.BU:

```
for i in range(8):
    VR[vd].H[i] = SLLW_U(VR[vj].B[i], ui3)
```

VSLLWIL.WU.HU:

```
for i in range(4):
    VR[vd].W[i] = SLLW_U(VR[vj].H[i], ui4)
```

VSLLWIL.DU.WU:

```
for i in range(2):
    VR[vd].D[i] = SLLW_U(VR[vj].W[i], ui5)
```

XVSLLWIL.HU.BU:

```
for i in range(8) :
    XR[xd].H[i] = SLLW_U(XR[xj].B[i], ui3)
    XR[xd].H[i+8] = SLLW_U(XR[xj].B[i+16], ui3)
```

XVSLLWIL.WU.HU:

```
for i in range(4):
    XR[xd].W[i] = SLLW_UXR[xj].H[i], ui4)
    XR[xd].W[i+4] = SLLW_U(XR[xj].H[i+8], ui4)
```

XVSLLWIL.DU.WU:

```
for i in range(2):
    XR[xd].D[i] = SLLW_U(XR[xj].W[i], ui5)
    XR[xd].D[i+2] = SLLW_U(XR[xj].W[i+4], ui5)
```

The [X]VEXTL.QU.DU instruction zero-extends each double-word element in the lower part of each 128 bits of vector register vj/xj. The result is sequentially written into each quad-word element of each 128 bits of vector register vd/xd.

VEXTL.QU.DU:

```
for i in range(1) :
    VR[vd].Q[i] = ZeroExtend(VR[vj].D[i], 128)
```

XVEXTL.QU.DU:

```
for i in range(1) :
    XR[xd].Q[i]   = ZeroExtend(XR[vj].D[i], 128)
    XR[xd].Q[i+1] = ZeroExtend(XR[vj].D[i+2], 128)
```

1.2.2.5 [X]VSRLR[I].{B/H/W/D}, [X]VSRAR[I].{B/H/W/D}

Instruction Format:				
	vsrlr.b	vd, vj, vk	xvsrlr.b	xd, xj, xk
	vsrlr.h	vd, vj, vk	xvsrlr.h	xd, xj, xk
	vsrlr.w	vd, vj, vk	xvsrlr.w	xd, xj, xk
	vsrlr.d	vd, vj, vk	xvsrlr.d	xd, xj, xk
	vsrlri.b	vd, vj, ui3	xvsrlri.b	xd, xj, ui3
	vsrlri.h	vd, vj, ui4	xvsrlri.h	xd, xj, ui4
	vsrlri.w	vd, vj, ui5	xvsrlri.w	xd, xj, ui5
	vsrlri.d	vd, vj, ui6	xvsrlri.d	xd, xj, ui6
	vsrar.b	vd, vj, vk	xvsrar.b	xd, xj, xk
	vsrar.h	vd, vj, vk	xvsrar.h	xd, xj, xk
	vsrar.w	vd,vj, vk	xvsrar.w	xd, xj, xk
	vsrar.d	vd, vj, vk	xvsrar.d	xd, xj, xk
	vsrari.b	vd, vj, ui3	xvsrari.b	xd, xj, ui3
	vsrari.h	vd, vj, ui4	xvsrari.h	xd, xj, ui4
	vsrari.w	vd, vj, ui5	xvsrari.w	xd, xj, ui5
	vsrari.d	vd, vj, ui6	xvsrari.d	xd, xj, ui6

The [X]VSRLR.{B/H/W/D} instruction performs a logical right shift operation with rounding on each byte/half-word/word/double-word element in vector register vj/xj. The shift amount is derived from the values of the byte/half-word/word/double-word elements at the corresponding position in vector register vk/xk. The result is written to the byte/half-word/word/double-word elements at the corresponding position in vector register vd/xd.

VSRLR.B:

```
for i in range(16):
    VR[vd].B[i] = SRLR(VR[vj].B[i], VR[vk].B[i] % 8)
```

VSRLR.H:

```
for i in range(8):
    VR[vd].H[i] = SRLR(VR[vj].H[i], VR[vk].H[i] % 16)
```

VSRLR.W:

```
for i in range(4):
    VR[vd].W[i] = SRLR(VR[vj].W[i], VR[vk].W[i] % 32)
```

VSRLR.D:

```
for i in range(2):
    VR[vd].D[i] = SRLR(VR[vj].D[i], VR[vk].D[i] % 64)
```

XVSRLR.B:

```
for i in range(32) :
    XR[xd].B[i] = SRLR(XR[xj].B[i], XR[xk].B[i] % 8)
```

XVSRLR.H:

```
for i in range(16) :
    XR[xd].H[i] = SRLR(XR[xj].H[i], XR[xk].H[i] % 16)
```

XVSRLR.W:

```
for i in range(8):
    XR[xd].W[i] = SRLR(XR[xj].W[i], XR[xk].W[i] % 32)
```

XVSRLR.D:

```
for i in range(4):
    XR[xd].D[i] = SRLR(XR[xj].D[i], XR[xk].D[i] % 64)
```

The [X]VSRLRI.{B/H/W/D} instruction performs a logical right shift operation with rounding on each byte/half-word/word/double-word element in vector register vj/xj. The shift amount is derived from the immediate value ui3/ui4/ui5/ui6. The result is written to the byte/half-word/word/double-word elements at the corresponding position in vector register vd/xd.

VSRLRI.B:

```
for i in range(16) :
    VR[vd].B[i] = SRLR(VR[vj].B[i], ui3)
```

VSRLRI.H:

```
for i in range(8):
    VR[vd].H[i] = SRLR(VR[vj].H[i], ui4)
```

VSRLRI.W:

```
for i in range(4):
    VR[vd].W[i] = SRLR(VR[vj].W[i], ui5)
```

VSRLRI.D:

```
for i in range(2):
    VR[vd].D[i] = SRLR(VR[vj].D[i], ui6)
```

XVSRLRI.B:

```
for i in range(32) :
    XR[xd].B[i] = SRLR(XR[xj].B[i], ui3)
```

XVSRLRI.H:

```
for i in range(16) :
    XR[xd].H[i] = SRLR(XR[xj].H[i], ui4)
```

XVSRLRI.W:

```
for i in range(8) :
    XR[xd].W[i] = SRLR(XR[xj].W[i], ui5)
```

XVSRLRI.D:

```
for i in range(4):
    XR[xd].D[i] = SRLR(XR[xj].D[i], ui6)
```

The [X]VSRAR.{B/H/W/D} instruction performs an arithmetic right shift operation with rounding on each byte/half-word/word/double-word element in vector register vj/xj. The shift amount is derived from the values of the byte/half-word/word/double-word elements at the corresponding position in vector register vk/xk. The result is written to the byte/half-word/word/double-word elements at the corresponding position in vector register vd/xd.

VSRAR.B:

```
for i in range(16):
    VR[vd].B[i] = SRAR(VR[vj].B[i], VR[vk].B[i] % 8)
```

VSRAR.H:

```
for i in range(8):
    VR[vd].H[i] = SRAR(VR[vj].H[i], VR[vk].H[i] % 16)
```

VSRAR.W:

```
for i in range(4):
    VR[vd].W[i] = SRAR(VR[vj].W[i], VR[vk].W[i] % 32)
```

VSRAR.D:

```
for i in range(2):
    VR[vd].D[i] = SRAR(VR[vj].D[i], VR[vk].D[i] % 64)
```

XVSRAR.B:

```
for i in range(32) :
    XR[xd].B[i] = SRAR(XR[xj].B[i], XR[xk].B[i] % 8)
```

XVSRAR.H:

```
for i in range(16) :
    XR[xd].H[i] = SRAR(XR[xj].H[i], XR[xk].H[i] % 16)
```

XVSRAR.W:

```
for i in range(8):
    XR[xd].W[i] = SRAR(XR[xj].W[i], XR[xk].W[i] % 32)
```

XVSRAR.D:

```
for i in range(4):
    XR[xd].D[i] = SRAR(XR[xj].D[i], XR[xk].D[i] % 64)
```

The [X]VSRARI.{B/H/W/D} instruction performs an arithmetic right shift operation with rounding on the byte/half-word/word/double-word elements in vector register vj/xj. The shift amount is derived from the immediate value ui3/ui4/ui5/ui6 in the instruction. The result is written to the corresponding byte/half-word/word/double-word elements in vector register vd/xd.

VSRARI.B:

```
for i in range(16) :
    VR[vd].B[i] = SRAR(VR[vj].B[i], ui3)
```

VSRARI.H:

```
for i in range(8):
    VR[vd].H[i] = SRAR(VR[vj].H[i], ui4)
```

VSRARI.W:

```
for i in range(4):
    VR[vd].W[i] = SRAR(VR[vj].W[i], ui5)
```

VSRARI.D:

```
for i in range(2):
    VR[vd].D[i] = SRAR(VR[vj].D[i], ui6)
```

XVSRARI.B:

```
for i in range(32):
    XR[xd].B[i] = SRAR(XR[xj].B[i], ui3)
```

XVSRARI.H:

```
for i in range(16):
    XR[xd].H[i] = SRAR(XR[xj].H[i], ui4)
```

XVSRARI.W:

```
for i in range(8) :
    XR[xd].W[i] = SRAR(XR[xj].W[i], ui5)
```

XVSRARI.D:

```
for i in range(4):
    XR[xd].D[i] = SRAR(XR[xj].D[i], ui6)
```

1.2.2.6 [X]VSRLN.{B.H/H.W/W.D}, [X]VSRAN.{B.H/H.W/W.D}

Instruction Format:				
	vsrln.b.h	vd, vj, vk	xvsrln.b.h	xd, xj, xk
	vsrln.h.w	vd, vj, vk	xvsrln.h.w	xd, xj, xk
	vsrln.w.d	vd, vj, vk	xvsrln.w.d	xd, xj, xk
	vsran.b.h	vd, vj, vk	xvsran.b.h	xd, xj, xk
	vsran.h.w	vd, vj, vk	xvsran.h.w	xd, xj, xk
	vsran.w.d	vd, vj, vk	xvsran.w.d	xd, xj, xk

The [X]VSRLN.{B.H/H.W/W.D} instruction performs a logical right shift operation, truncating the width to half on half-word/word/double-word elements of each 128 bits in vector register vj/xj. The shift amount is derived from the corresponding half-word/word/double-word elements' value in vector register vk/xk. The lower part of the result is sequentially written into byte/half-word/word elements in the lower part of each 128 bits of vector registers vd/xd. The upper part of each 128 bits in vd/xd is set to 0.

VSRLN.B.H:

```
for i in range(8):
    VR[vd].B[i] = SRLN(VR[vj].H[i], VR[vk].H[i] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
```

VSRLN.H.W:

```
for i in range(4):
    VR[vd].H[i] = SRLN(VR[vj].W[i], VR[vk].W[i] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
```

VSRLN.W.D:

```
for i in range(2):
    VR[vd].W[i] = SRLN(VR[vj].D[i], VR[vk].D[i] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
```

XVSRLN.B.H:

```
for i in range(8):
    VR[vd].B[i] = SRLN(VR[vj].H[i], VR[vk].H[i] % 16)
    VR[vd].B[i+16] = SRLN(VR[vj].H[i+8], VR[vk].H[i+8] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
    VR[vd].B[i+16] = 0
```

XVSRLN.H.W:

```
for i in range(4):
    VR[vd].H[i] = SRLN(VR[vj].W[i], VR[vk].W[i] % 32)
    VR[vd].H[i+8] = SRLN(VR[vj].W[i+4], VR[vk].W[i+4] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
    VR[vd].H[i+8] = 0
```

XVSRLN.W.D:

```
for i in range(2):
    VR[vd].W[i] = SRLN(VR[vj].D[i], VR[vk].D[i] % 64)
    VR[vd].W[i+4] = SRLN(VR[vj].D[i+2], VR[vk].D[i+2] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
    VR[vd].W[i+4] = 0
```

The [X]VSRAN.{B.H/H.W/W.D} instruction performs an arithmetic right shift operation, truncating the width to half on half-word/word/double-word elements of each 128 bits in vector register vj/xj. The shift amount is derived from the corresponding half-word/word/double-word elements' value in vector register vk/xk. The lower part of the result is sequentially written into byte/half-word/word elements in the lower part of each 128 bits of vector registers vd/xd. The upper part of each 128 bits in vd/xd is set to 0.

VSRAN.B.H:

```
for i in range(8):
    VR[vd].B[i] = SRAN(VR[vj].H[i], VR[vk].H[i] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
```

VSRAN.H.W:

```
for i in range(4):
    VR[vd].H[i] = SRAN(VR[vj].W[i], VR[vk].W[i] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
```

VSRAN.W.D:

```
for i in range(2):
    VR[vd].W[i] = SRAN(VR[vj].D[i], VR[vk].D[i] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
```

XVSRAN.B.H:

```
for i in range(8):
    VR[vd].B[i] = SRAN(VR[vj].H[i], VR[vk].H[i] % 16)
    VR[vd].B[i+16] = SRAN(VR[vj].H[i+8], VR[vk].H[i+8] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
    VR[vd].B[i+16] = 0
```

XVSRAN.H.W:

```
for i in range(4):
    VR[vd].H[i] = SRAN(VR[vj].W[i], VR[vk].W[i] % 32)
    VR[vd].H[i+8] = SRAN(VR[vj].W[i+4], VR[vk].W[i+4] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
    VR[vd].H[i+8] = 0
```

XVSRAN.W.D:

```
for i in range(2):
    VR[vd].W[i] = SRAN(VR[vj].D[i], VR[vk].D[i] % 64)
    VR[vd].W[i+4] = SRAN(VR[vj].D[i+2], VR[vk].D[i+2] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
    VR[vd].W[i+4] = 0
```

1.2.2.7 [X]VSRLNI.{B.H/H.W/W.D/D.Q}, [X]VSRANI.{B.H/H.W/W.D/D.Q}

Instruction Format:				
	vsrlni.b.h	vd, vj, ui4	xvsrlni.b.h	xd, xj, ui4
	vsrlni.h.w	vd, vj, ui5	xvsrlni.h.w	xd, xj, ui5
	vsrlni.w.d	vd, vj, ui6	xvsrlni.w.d	xd, xj, ui6
	vsrlni.d.q	vd, vj, ui7	xvsrlni.d.q	xd, xj, ui7
	vsrani.b.h	vd, vj, ui4	xvsrani.b.h	xd, xj, ui4
	vsrani.h.w	vd, vj, ui5	xvsrani.h.w	xd, xj, ui5
	vsrani.w.d	vd, vj, ui6	xvsrani.w.d	xd, xj, ui6
	vsrani.d.q	vd, vj, ui7	xvsrani.d.q	xd, xj, ui7

The VSRLNI.{B/H/W/D} instruction concatenates vector register vd and vector register vj to form a 256-bit vector, and performs a logical right shift operation, truncating the width to half on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from the immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of vector register vd.

VSRLNI.B.H:

```
for i in range(8):
    VR[vd].B[i] = SRLN(VR[vj].H[i], ui4)
    VR[vd].B[i+8] = SRLN(VR[vd].H[i], ui4)
```

VSRLNI.H.W:

```
for i in range(4):
    VR[vd].H[i] = SRLN(VR[vj].W[i], ui5)
    VR[vd].H[i+4] = SRLN(VR[vd].W[i], ui5)
```

VSRLNI.W.D:

```
for i in range(2):
    VR[vd].W[i] = SRLN(VR[vj].D[i], ui6)
    VR[vd].W[i+2] = SRLN(VR[vd].D[i], ui6)
```

VSRLNI.D.Q:

```
for i in range(1):
    VR[vd].D[i] = SRLN(VR[vj].Q[i], ui7)
    VR[vd].D[i+1] = SRLN(VR[vd].Q[i], ui7)
```

The XVRSLNI.{B/H/W/D} instruction concatenates the lower half of vector register xd with the lower half of vector register xj to form a 256-bit vector, and performs a logical right shift operation, truncating the width to half on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from the immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of vector register xd. The upper half of vector register xd is concatenated to the upper half of vector register xj to form a 256-bit vector, and performs a logical right shift operation, truncating the width to half on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from the immediate value ui4/ui5/ui6/ui7. The upper half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of vector register xd.

XVSRLNI.B.H:

```
for i in range(8):
    XR[xd].B[i] = SRLN(XR[xj].H[i], ui4)
    XR[xd].B[i+8] = SRLN(XR[xd].H[i], ui4)
    XR[xd].B[i+16] = SRLN(XR[xj].H[i+8], ui4)
    XR[xd].B[i+24] = SRLN(XR[xd].H[i+8], ui4)
```

XVSRLNI.H.W:

```
for i in range(4) :
    XR[xd].H[i] = SRLN(XR[xj].W[i], ui5)
    XR[xd].H[i+4] = SRLN(XR[xd].W[i], ui5)
    XR[xd].H[i+8] = SRLN(XR[xj].W[i+4], ui5)
    XR[xd].H[i+12] = SRLN(XR[xd].W[i+4], ui5)
```

XVSRLNI.W.D:

```
for i in range(2) :
    XR[xd].W[i] = SRLN(XR[xj].D[i], ui6)
    XR[xd].W[i+2] = SRLN(XR[xd].D[i], ui6)
    XR[xd].W[i+4] = SRLN(XR[xj].D[i+2], ui6)
    XR[xd].W[i+6] = SRLN(XR[xd].D[i+2], ui6)
```

XVSRLNI.D.Q:

```
for i in range(1) :
    XR[xd].D[i] = SRLN(XR[xj].Q[i], ui7)
    XR[xd].D[i+1] = SRLN(XR[xd].Q[i], ui7)
    XR[xd].D[i+2] = SRLN(XR[xj].Q[i+1], ui7)
    XR[xd].D[i+3] = SRLN(XR[xd].Q[i+1], ui7)
```

The VSRANI.{B/H/W/D} instruction concatenates vector register vd and vector register vj to form a 256-bit vector, and performs an arithmetic right shift operation, truncating the width to half on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from the immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of vector register vd.

VSRANI.B.H:

```
for i in range(8):
    VR[vd].B[i] = SRAN(VR[vj].H[i], ui4)
    VR[vd].B[i+8] = SRAN(VR[vd].H[i], ui4)
```

VSRANI.H.W:

```
for i in range(4):
    VR[vd].H[i] = SRAN(VR[vj].W[i], ui5)
    VR[vd].H[i+4] = SRAN(VR[vd].W[i], ui5)
```

VSRANI.W.D:

```
for i in range(2):
    VR[vd].W[i] = SRAN(VR[vj].D[i], ui6)
    VR[vd].W[i+2] = SRAN(VR[vd].D[i], ui6)
```

VSRANI.D.Q:

```
for i in range(1):
    VR[vd].D[i] = SRAN(VR[vj].Q[i], ui7)
    VR[vd].D[i+1] = SRAN(VR[vd].Q[i], ui7)
```

The XVSRANI.{B/H/W/D} instruction concatenates the lower half of vector register xd with the lower half of vector register xj to form a 256-bit vector, and performs an arithmetic right shift operation, truncating the width to half on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from the immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of vector register xd. The upper half of vector register xd is concatenated to the upper half of vector register xj to form a 256-bit vector, and performs an arithmetic right shift operation, truncating the width to half on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from the immediate value ui4/ui5/ui6/ui7. The upper half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of vector register xd.

XVSRANI.B.H:

```
for i in range(8):
    XR[xd].B[i] = SRAN(XR[xj].H[i], ui4)
    XR[xd].B[i+8] = SRAN(XR[xd].H[i], ui4)
    XR[xd].B[i+16] = SRAN(XR[xj].H[i+8], ui4)
    XR[xd].B[i+24] = SRAN(XR[xd].H[i+8], ui4)
```

XVSRANI.H.W:

```
for i in range(4) :
    XR[xd].H[i] = SRAN(XR[xj].W[i], ui5)
    XR[xd].H[i+4] = SRAN(XR[xd].W[i], ui5)
    XR[xd].H[i+8] = SRAN(XR[xj].W[i+4], ui5)
    XR[xd].H[i+12] = SRAN(XR[xd].W[i+4], ui5)
```

XVSRANI.W.D:

```
for i in range(2) :
    XR[xd].W[i] = SRAN(XR[xj].D[i], ui6)
    XR[xd].W[i+2] = SRAN(XR[xd].D[i], ui6)
    XR[xd].W[i+4] = SRAN(XR[xj].D[i+2], ui6)
    XR[xd].W[i+6] = SRAN(XR[xd].D[i+2], ui6)
```

XVSRANI.D.Q:

```
for i in range(1) :
    XR[xd].D[i] = SRAN(XR[xj].Q[i], ui7)
    XR[xd].D[i+1] = SRAN(XR[xd].Q[i], ui7)
    XR[xd].D[i+2] = SRAN(XR[xj].Q[i+1], ui7)
    XR[xd].D[i+3] = SRAN(XR[xd].Q[i+1], ui7)
```

1.2.2.8 [X]VSRLRN.{B.H/H.W/W.D}, [X]VSRARN.{B.H/H.W/W.D}

Instruction Format:	vsrlrn.b.h	vd, vj, vk	xvsrlrn.b.h	xd, xj, xk
	vsrlrn.h.w	vd, vj, vk	xvsrlrn.h.w	xd, xj, xk
	vsrlrn.w.d	vd, vj, vk	xvsrlrn.w.d	xd, xj, xk
	vsrarn.b.h	vd, vj, vk	xvsrarn.b.h	xd, xj, xk
	vsrarn.h.w	vd, vj, vk	xvsrarn.h.w	xd, xj, xk
	vsrarn.w.d	vd, vj, vk	xvsrarn.w.d	xd, xj, xk

The [X]VSRLRN.{B.H/H.W/W.D} instruction performs a logical right shift with rounding on half-word/word/double-word elements of every 128 bits of vector register vj/xj. The shift amount is derived from the corresponding half-word/word/double-word elements in vector register vk/xk. The lower half of the result is sequentially written into the byte/half-word/word elements of the lower half of every 128 bits in vector registers vd/xd. The upper half of every 128 bits

in vd/xd is set to 0.

VSRLRN.B.H:

```
for i in range(8):
    VR[vd].B[i] = SRLRN(VR[vj].H[i], VR[vk].H[i] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
```

VSRLRN.H.W:

```
for i in range(4):
    VR[vd].H[i] = SRLRN(VR[vj].W[i], VR[vk].W[i] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
```

VSRLRN.W.D:

```
for i in range(2):
    VR[vd].W[i] = SRLRN(VR[vj].D[i], VR[vk].D[i] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
```

XVSRLRN.B.H:

```
for i in range(8):
    VR[vd].B[i] = SRLRN(VR[vj].H[i], VR[vk].H[i] % 16)
    VR[vd].B[i+16] = SRLRN(VR[vj].H[i+8], VR[vk].H[i+8] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
    VR[vd].B[i+16] = 0
```

XVSRLRN.H.W:

```
for i in range(4):
    VR[vd].H[i] = SRLRN(VR[vj].W[i], VR[vk].W[i] % 32)
    VR[vd].H[i+8] = SRLRN(VR[vj].W[i+4], VR[vk].W[i+4] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
    VR[vd].H[i+8] = 0
```

XVSRLRN.W.D:

```
for i in range(2):
    VR[vd].W[i] = SRLRN(VR[vj].D[i], VR[vk].D[i] % 64)
    VR[vd].W[i+4] = SRLRN(VR[vj].D[i+2], VR[vk].D[i+2] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
    VR[vd].W[i+4] = 0
```

The [X]VSRARN.{B.H/H.W/W.D} instruction performs an arithmetic right shift with rounding on half-word/word/double-word elements of every 128 bits of vector register vj/xj. The shift amount is derived from the corresponding half-word/word/double-word elements in vector register vk/xk. The lower half of the result is sequentially written into the byte/half-word/word elements of the lower half of every 128 bits in vector registers vd/xd. The upper half of every 128 bits in vd/xd is set to 0.

VSRARN.B.H:

```
for i in range(8):
    VR[vd].B[i] = SRARN(VR[vj].H[i], VR[vk].H[i] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
```

VSRARN.H.W:

```
for i in range(4):
    VR[vd].H[i] = SRARN(VR[vj].W[i], VR[vk].W[i] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
```

VSRARN.W.D:

```
for i in range(2):
    VR[vd].W[i] = SRARN(VR[vj].D[i], VR[vk].D[i] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
```

XVSRARN.B.H:

```
for i in range(8):
    VR[vd].B[i] = SRARN(VR[vj].H[i], VR[vk].H[i] % 16)
    VR[vd].B[i+16] = SRARN(VR[vj].H[i+8], VR[vk].H[i+8] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
    VR[vd].B[i+16] = 0
```

XVSRARN.H.W:

```
for i in range(4):
    VR[vd].H[i] = SRARN(VR[vj].W[i], VR[vk].W[i] % 32)
    VR[vd].H[i+8] = SRARN(VR[vj].W[i+4], VR[vk].W[i+4] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
    VR[vd].H[i+8] = 0
```

XVSRARN.W.D:

```
for i in range(2):
    VR[vd].W[i] = SRARN(VR[vj].D[i], VR[vk].D[i] % 64)
    VR[vd].W[i+4] = SRARN(VR[vj].D[i+2], VR[vk].D[i+2] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
    VR[vd].W[i+4] = 0
```

1.2.2.9 [X]VSRLRNI.{B.H/H.W/W.D/D.Q}, [X]VSRARNI.{B.H/H.W/W.D/D.Q}

Instruction Format:				
	vsrlrni.b.h	vd, vj, ui4	xvsrlrni.b.h	xd, xj, ui4
	vsrlrni.h.w	vd, vj, ui5	xvsrlrni.h.w	xd, xj, ui5
	vsrlrni.w.d	vd, vj, ui6	xvsrlrni.w.d	xd, xj, ui6
	vsrlrni.d.q	vd, vj, ui7	xvsrlrni.d.q	xd, xj, ui7
	vsrarni.b.h	vd, vj, ui4	xvsrarni.b.h	xd, xj, ui4
	vsrarni.h.w	vd, vj, ui5	xvsrarni.h.w	xd, xj, ui5
	vsrarni.w.d	vd, vj, ui6	xvsrarni.w.d	xd, xj, ui6
	vsrarni.d.q	vd, vj, ui7	xvsrarni.d.q	xd, xj, ui7

The VSRLRNI.{B/H/W/D} instruction concatenates vector register vd and vector register vj to form a 256-bit vector, and performs a logical right shift with rounding on each half-word/word/double-word/quad element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of vector register vd.

VSRLRNI.B.H:

```
for i in range(8):
    VR[vd].B[i] = SRLRN(VR[vj].H[i], ui4)
    VR[vd].B[i+8] = SRLRN(VR[vd].H[i], ui4)
```

VSRLRNI.H.W:

```
for i in range(4) :
    VR[vd].H[i] = SRLRN(VR[vj].W[i], ui5)
    VR[vd].H[i+4] = SRLRN(VR[vd].W[i], ui5)
```

VSRLRNI.W.D:

```
for i in range(2):
    VR[vd].W[i] = SRLRN(VR[vj].D[i], ui6)
    VR[vd].W[i+2] = SRLRN(VR[vd].D[i], ui6)
```

VSRLRNI.D.Q:

```
for i in range(1):
    VR[vd].D[i] = SRLRN(VR[vj].Q[i], ui7)
    VR[vd].D[i+1] = SRLRN(VR[vd].Q[i], ui7)
```

The XVSRLRNI.{B/H/W/D} instruction concatenates the lower half of vector register xd with the lower half of vector register xj to form a 256-bit vector, and performs a logical right shift with rounding on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of the lower half of vector register xd. The upper half of vector register xd is concatenated with the upper half of vector register xj to form a 256-bit vector, and performs a logical right shift operation with rounding on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The upper half of the result is sequentially written to the byte/half-word/word/double-word elements corresponding to the upper half of vector register xd.

XVSRLRNI.B.H:

```
for i in range(8):
    XR[xd].B[i] = SRLRN(XR[xj].H[i], ui4)
    XR[xd].B[i+8] = SRLRN(XR[xd].H[i], ui4)
    XR[xd].B[i+16] = SRLRN(XR[xj].H[i+8], ui4)
    XR[xd].B[i+24] = SRLRN(XR[xd].H[i+8], ui4)
```

XVSRLRNI.H.W:

```
for i in range(4) :
    XR[xd].H[i] = SRLRN(XR[xj].W[i], ui5)
    XR[xd].H[i+4] = SRLRN(XR[xd].W[i], ui5)
    XR[xd].H[i+8] = SRLRN(XR[xj].W[i+4], ui5)
    XR[xd].H[i+12] = SRLRN(XR[xd].W[i+4], ui5)
```

XVSRLRNI.W.D:

```
for i in range(2):
    XR[xd].W[i] = SRLRN(XR[xj].D[i], ui6)
    XR[xd].W[i+2] = SRLRN(XR[xd].D[i], ui6)
    XR[xd].W[i+4] = SRLRN(XR[xj].D[i+2], ui6)
    XR[xd].W[i+6] = SRLRN(XR[xd].D[i+2], ui6)
```

XVSRLRNI.D.Q:

```
for i in range(1) :
    XR[xd].D[i] = SRLRN(XR[xj].Q[i], ui7)
    XR[xd].D[i+1] = SRLRN(XR[xd].Q[i], ui7)
    XR[xd].D[i+2] = SRLRN(XR[xj].Q[i+1], ui7)
    XR[xd].D[i+3] = SRLRN(XR[xd].Q[i+1], ui7)
```

The VSRARNI.{B/H/W/D} instruction concatenates vector register vd and vector register vj to form a 256-bit vector, and performs an arithmetic right shift with rounding on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of vector register vd.

VSRARNI.B.H:

```
for i in range(8):
    VR[vd].B[i] = SRARN(VR[vj].H[i], ui4)
    VR[vd].B[i+8] = SRARN(VR[vd].H[i], ui4)
```

VSRARNI.H.W:

```
for i in range(4):
    VR[vd].H[i] = SRARN(VR[vj].W[i], ui5)
    VR[vd].H[i+4] = SRARN(VR[vd].W[i], ui5)
```

VSRARNI.W.D:

```
for i in range(2):
    VR[vd].W[i] = SRARN(VR[vj].D[i], ui6)
    VR[vd].W[i+2] = SRARN(VR[vd].D[i], ui6)
```

VSRARNI.D.Q:

```
for i in range(1):
    VR[vd].D[i] = SRARN(VR[vj].Q[i], ui7)
    VR[vd].D[i+1] = SRARN(VR[vd].Q[i], ui7)
```

The XVSRARNI.{B/H/W/D} instruction concatenates the lower half of vector register xd with the lower half of vector register xj to form a 256-bit vector, and performs an arithmetic right shift with rounding on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of the lower half of vector register xd. The upper half of vector register xd is concatenated with the upper half of vector register xj to form a 256-bit vector, and performs an arithmetic right shift operation with rounding on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The upper half of the result is sequentially written to the byte/half-word/word/double-word elements corresponding to the upper half of vector register xd.

XVSRARNI.B.H:

```
for i in range(8):
    XR[xd].B[i] = SRARN(XR[xj].H[i], ui4)
    XR[xd].B[i+8] = SRARN(XR[xd].H[i], ui4)
    XR[xd].B[i+16] = SRARN(XR[xj].H[i+8], ui4)
    XR[xd].B[i+24] = SRARN(XR[xd].H[i+8], ui4)
```

XVSRARNI.H.W:

```
for i in range(4) :
    XR[xd].H[i] = SRARN(XR[xj].W[i], ui5)
    XR[xd].H[i+4] = SRARN(XR[xd].W[i], ui5)
    XR[xd].H[i+8] = SRARN(XR[xj].W[i+4], ui5)
    XR[xd].H[i+12] = SRARN(XR[xd].W[i+4], ui5)
```

XVSRARNI.W.D:

```
for i in range(2) :
    XR[xd].W[i] = SRARN(XR[xj].D[i], ui6)
    XR[xd].W[i+2] = SRARN(XR[xd].D[i], ui6)
    XR[xd].W[i+4] = SRARN(XR[xj].D[i+2], ui6)
    XR[xd].W[i+6] = SRARN(XR[xd].D[i+2], ui6)
```

XVSRARNI.D.Q:

```
for i in range(1) :
    XR[xd].D[i] = SRARN(XR[xj].Q[i], ui7)
    XR[xd].D[i+1] = SRARN(XR[xd].Q[i], ui7)
    XR[xd].D[i+2] = SRARN(XR[xj].Q[i+1], ui7)
    XR[xd].D[i+3] = SRARN(XR[xd].Q[i+1], ui7)
```

1.2.2.10 [X]VSSRLN.{B.H/H.W/W.D}, [X]VSSRAN.{B.H/H.W/W.D}, [X]VSSRLN.{BU.H/HU.W/WU.D}, [X]VSSRAN.{BU.H/HU.W/WU.D}

Instruction Format:				
	vssrln.b.h	vd, vj, vk	xvssrln.b.h	xd, xj, xk
	vssrln.h.w	vd, vj, vk	xvssrln.h.w	xd, xj, xk
	vssrln.w.d	vd, vj, vk	xvssrln.w.d	xd, xj, xk
	vssran.b.h	vd, vj, vk	xvssran.b.h	xd, xj, xk
	vssran.h.w	vd, vj, vk	xvssran.h.w	xd, xj, xk
	vssran.w.d	vd, vj, vk	xvssran.w.d	xd, xj, xk
	vssrln.bu.h	vd, vj, vk	xvssrln.bu.h	xd, xj, xk
	vssrln.hu.w	vd, vj, vk	xvssrln.hu.w	xd, xj, xk

The [X]VSSRLN.{B.H/H.W/W.D} instruction performs a logical right shift with signed saturation arithmetic and half the width on each half-word/word/double-word element in every 128 bits of vector register vj/xj. The shift amount is derived from the corresponding half-word/word/double-word elements' value in vector register vk/xk. The lower half of the result is sequentially written into the byte/half-word/word elements of the lower half of each 128 bits in vector registers vd/xd. The upper half of each 128 bits in vd/xd is set to 0.

VSSRLN.B.H:

```
for i in range(8):
    VR[vd].B[i] = SSRLNS(VR[vj].H[i], VR[vk].H[i] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
```

VSSRLN.H.W:

```
for i in range(4):
    VR[vd].H[i] = SSRLNS(VR[vj].W[i], VR[vk].W[i] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
```

VSSRLN.W.D:

```
for i in range(2):
    VR[vd].W[i] = SSRLNS(VR[vj].D[i], VR[vk].D[i] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
```

XVSSRLN.B.H:

```
for i in range(8):
    VR[vd].B[i] = SSRLNS(VR[vj].H[i], VR[vk].H[i] % 16)
    VR[vd].B[i+16] = SSRLNS(VR[vj].H[i+8], VR[vk].H[i+8] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
    VR[vd].B[i+16] = 0
```

XVSSRLN.H.W:

```
for i in range(4):
    VR[vd].H[i] = SSRLNS(VR[vj].W[i], VR[vk].W[i] % 32)
    VR[vd].H[i+8] = SSRLNS(VR[vj].W[i+4], VR[vk].W[i+4] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
    VR[vd].H[i+8] = 0
```

XVSSRLN.W.D:

```
for i in range(2):
    VR[vd].W[i] = SSRLNS(VR[vj].D[i], VR[vk].D[i] % 64)
    VR[vd].W[i+4] = SSRLNS(VR[vj].D[i+2], VR[vk].D[i+2] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
    VR[vd].W[i+4] = 0
```

The [X]VSSRAN.{B.H/H.W/W.D} instruction performs an arithmetic right shift with signed saturation arithmetic and half the width on each half-word/word/double-word elements in every 128 bits of vector register vj/xj. The shift amount is derived from the corresponding half-word/word/double-word

elements' value in vector register vk/xk. The lower half of the result is sequentially written into the byte/half-word/word elements of the lower half of each 128 bits in vector registers vd/xd. The upper half of each 128 bits in vd/xd is set to 0.

VSSRAN.B.H:

```
for i in range(8):
    VR[vd].B[i] = SSRANS(VR[vj].H[i], VR[vk].H[i] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
```

VSSRAN.H.W:

```
for i in range(4):
    VR[vd].H[i] = SSRANS(VR[vj].W[i], VR[vk].W[i] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
```

VSSRAN.W.D:

```
for i in range(2):
    VR[vd].W[i] = SSRANS(VR[vj].D[i], VR[vk].D[i] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
```

XVSSRAN.B.H:

```
for i in range(8):
    VR[vd].B[i] = SSRANS(VR[vj].H[i], VR[vk].H[i] % 16)
    VR[vd].B[i+16] = SSRANS(VR[vj].H[i+8], VR[vk].H[i+8] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
    VR[vd].B[i+16] = 0
```

XVSSRAN.H.W:

```
for i in range(4):
    VR[vd].H[i] = SSRANS(VR[vj].W[i], VR[vk].W[i] % 32)
    VR[vd].H[i+8] = SSRANS(VR[vj].W[i+4], VR[vk].W[i+4] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
    VR[vd].H[i+8] = 0
```

XVSSRAN.W.D:

```
for i in range(2):
    VR[vd].W[i] = SSRANS(VR[vj].D[i], VR[vk].D[i] % 64)
    VR[vd].W[i+4] = SSRANS(VR[vj].D[i+2], VR[vk].D[i+2] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
    VR[vd].W[i+4] = 0
```

The [X]VSSRLN.{BU.H/HU.W/WU.D} instruction performs a logical right shift with unsigned saturation arithmetic and half the width on each half-word/word/double-word elements in every 128 bits of vector register vj/xj. The shift amount is derived from the corresponding half-word/word/double-word elements' value in vector register vk/xk. The lower half of the result is sequentially written into the byte/half-word/word elements of the lower half of each 128 bits in vector registers vd/xd. The upper half of each 128 bits in vd/xd is set to 0.

VSSRLN.BU.H:

```
for i in range(8):
    VR[vd].B[i] = SSRLNU(VR[vj].H[i], VR[vk].H[i] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
```

VSSRLN.HU.W:

```
for i in range(4):
        VR[vd].H[i] = SSRLNU(VR[vj].W[i], VR[vk].W[i] % 32)
for i in range(4, 8, 1):
        VR[vd].H[i] = 0
```

VSSRLN.WU.D:

```
for i in range(2):
        VR[vd].W[i] = SSRLNU(VR[vj].D[i], VR[vk].D[i] % 64)
for i in range(2, 4, 1):
        VR[vd].W[i] = 0
```

XVSSRLN.BU.H:

```
for i in range(8):
        VR[vd].B[i] = SSRLNU(VR[vj].H[i], VR[vk].H[i] % 16)
        VR[vd].B[i+16] = SSRLNU(VR[vj].H[i+8], VR[vk].H[i+8] % 16)
for i in range(8, 16, 1):
        VR[vd].B[i] = 0
        VR[vd].B[i+16] = 0
```

XVSSRLN.HU.W:

```
for i in range(4):
        VR[vd].H[i] = SSRLNU(VR[vj].W[i], VR[vk].W[i]%32)
        VR[vd].H[i+8] = SSRLNU(VR[vj].W[i+4], VR[vk].W[i+4]%32)
for i in range(4, 8, 1):
        VR[vd].H[i] = 0
        VR[vd].H[i+8] = 0
```

XVSSRLN.WU.D:

```
for i in range(2):
        VR[vd].W[i] = SSRLNU(VR[vj].D[i], VR[vk].D[i]%64)
        VR[vd].W[i+4] = SSRLNU(VR[vj].D[i+2], VR[vk].D[i+2]%64)
for i in range(2, 4, 1):
        VR[vd].W[i] = 0
        VR[vd].W[i+4] = 0
```

The [X]VSSRAN.{BU.H/HU.W/WU.D} instruction performs an arithmetic right shift with unsigned saturation arithmetic and half the width on each half-word/word/double-word elements in every 128 bits of vector register vj/xj. The shift amount is derived from the corresponding half-word/word/double-word elements' value in vector register vk/xk. The lower half of the result is sequentially written into the byte/half-word/word elements of the lower half of each 128 bits in vector registers vd/xd. The upper half of each 128 bits in vd/xd is set to 0.

VSSRAN.BU.H:

```
for i in range(8):
        VR[vd].B[i] = SSRANU(VR[vj].H[i], VR[vk].H[i] % 16)
for i in range(8, 16, 1):
        VR[vd].B[i] = 0
```

VSSRAN.HU.W:

```
for i in range(4):
        VR[vd].H[i] = SSRANU(VR[vj].W[i], VR[vk].W[i] % 32)
for i in range(4, 8, 1):
        VR[vd].H[i] = 0
```

VSSRAN.WU.D:

```
for i in range(2):
        VR[vd].W[i] = SSRANU(VR[vj].D[i], VR[vk].D[i] % 64)
for i in range(2, 4, 1):
        VR[vd].W[i] = 0
```

XVSSRAN.BU.H:

```
for i in range(8):
    VR[vd].B[i] = SSRANU(VR[vj].H[i], VR[vk].H[i] % 16)
    VR[vd].B[i+16] = SSRANU(VR[vj].H[i+8], VR[vk].H[i+8] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
    VR[vd].B[i+16] = 0
```

XVSSRAN.HU.W:

```
for i in range(4):
    VR[vd].H[i] = SSRANU(VR[vj].W[i], VR[vk].W[i] % 32)
    VR[vd].H[i+8] = SSRANU(VR[vj].W[i+4], VR[vk].W[i+4] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
    VR[vd].H[i+8] = 0
```

XVSSRAN.WU.D:

```
for i in range(2):
    VR[vd].W[i] = SSRANU(VR[vj].D[i], VR[vk].D[i] % 64)
    VR[vd].W[i+4] = SSRANU(VR[vj].D[i+2], VR[vk].D[i+2] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
    VR[vd].W[i+4] = 0
```

1.2.2.11 [X]VSSRLNI.{B.H/H.W/W.D/D.Q}, [X]VSSRANI.{B.H/H.W/W.D/D.Q}, [X]VSSRLNI.{BU.H/HU.W/WU.D/DU.Q}, [X]VSSRANI.{BU.H/HU.W/WU.D/DU.Q}

Instruction Format:

vssrlni.b.h	vd, vj, ui4		xvssrlni.b.h	xd, xj, ui4
vssrlni.h.w	vd, vj, ui5		xvssrlni.h.w	xd, xj, ui5
vssrlni.w.d	vd, vj, ui6		xvssrlni.w.d	xd, xj, ui6
vssrlni.d.q	vd, vj, ui7		xvssrlni.d.q	xd, xj, ui7
vssrani.b.h	vd, vj, ui4		xvssrani.b.h	xd, xj, ui4
vssrani.h.w	vd, vj, ui5		xvssrani.h.w	xd, xj, ui5
vssrani.w.d	vd, vj, ui6		xvssrani.w.d	xd, xj, ui6
vssrani.d.q	vd, vj, ui7		xvssrani.d.q	xd, xj, ui7
vssrlni.bu.h	vd, vj, ui4		xvssrlni.bu.h	xd, xj, ui4
vssrlni.hu.w	vd, vj, ui5		xvssrlni.hu.w	xd, xj, ui5
vssrlni.wu.d	vd, vj, ui6		xvssrlni.wu.d	xd, xj, ui6
vssrlni.du.q	vd, vj, ui7		xvssrlni.du.q	xd, xj, ui7
vssrani.bu.h	vd, vj, ui4		xvssrani.bu.h	xd, xj, ui4
vssrani.hu.w	vd, vj, ui5		xvssrani.hu.w	xd, xj, ui5
vssrani.wu.d	vd, vj, ui6		xvssrani.wu.d	xd, xj, ui6
vssrani.du.q	vd, vj, ui7		xvssrani.du.q	xd, xj, ui7

The VSSRLNI.{B.H/H.W/W.D/D.Q} instruction concatenates vector register vd and vector register vj to form a 256-bit vector, and performs a logical right shift with signed saturation arithmetic and half the width on each half-word/word/double-word/quad element within this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of vector register vd.

VSSRLNI.B.H:

```
for i in range(8):
    VR[vd].B[i] = SSRLNS(VR[vj].H[i], ui4)
    VR[vd].B[i+8] = SSRLNS(VR[vd].H[i], ui4)
```

VSSRLNI.H.W:

```
for i in range(4):
    VR[vd].H[i] = SSRLNS(VR[vj].W[i], ui5)
    VR[vd].H[i+4] = SSRLNS(VR[vd].W[i], ui5)
```

VSSRLNI.W.D:

```
for i in range(2):
    VR[vd].W[i] = SSRLNS(VR[vj].D[i], ui6)
    VR[vd].W[i+2] = SSRLNS(VR[vd].D[i], ui6)
```

VSSRLNI.D.Q:

```
for i in range(1) :
    VR[vd].D[i] = SSRLNS(VR[vj].Q[i], ui7)
    VR[vd].D[i+1] = SSRLNS(VR[vd].Q[i], ui7)
```

The XVSSRLNI.{B.H/H.W/W.D/D.Q} instruction concatenates the lower half of vector register xd with the lower half of vector register xj to form a 256-bit vector, and performs a logical right shift with signed saturation arithmetic and half the width on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of the lower half of vector register xd. The upper half of vector register xd is concatenated with the upper half of vector register xj to form a 256-bit vector, and performs a logical right shift with signed saturation arithmetic and half the width on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The upper half of the result is sequentially written to the byte/half-word/word/double-word elements corresponding to the upper half of vector register xd.

XVSSRLNI.B.H:

```
for i in range(8):
    XR[xd].B[i] = SSRLNS(XR[xj].H[i], ui4)
    XR[xd].B[i+8] = SSRLNS(XR[xd].H[i], ui4)
    XR[xd].B[i+16] = SSRLNS(XR[xj].H[i+8], ui4)
    XR[xd].B[i+24] = SSRLNS(XR[xd].H[i+8], ui4)
```

XVSSRLNI.H.W:

```
for i in range(4):
    XR[xd].H[i] = SSRLNS(XR[xj].W[i], ui5)
    XR[xd].H[i+4] = SSRLNS(XR[xd].W[i], ui5)
    XR[xd].H[i+8] = SSRLNS(XR[xj].W[i+4], ui5)
    XR[xd].H[i+12] = SSRLNS(XR[xd].W[i+4], ui5)
```

XVSSRLNI.W.D:

```
for i in range(2) :
    XR[xd].W[i] = SSRLNS(XR[xj].D[i], ui6)
    XR[xd].W[i+2] = SSRLNS(XR[xd].D[i], ui6)
    XR[xd].W[i+4] = SSRLNS(XR[xj].D[i+2], ui6)
    XR[xd].W[i+6] = SSRLNS(XR[xd].D[i+2], ui6)
```

XVSSRLNI.D.Q:

```
for i in range(1) :
    XR[xd].D[i] = SSRLNS(XR[xj].Q[i], ui7)
    XR[xd].D[i+1] = SSRLNS(XR[xd].Q[i], ui7)
    XR[xd].D[i+2] = SSRLNS(XR[xj].Q[i+1], ui7)
    XR[xd].D[i+3] = SSRLNS(XR[xd].Q[i+1], ui7)
```

The VSSRANI.{B.H/H.W/W.D/D.Q} instruction concatenates vector register vd and vector register vj to form a 256-bit vector, and performs an arithmetic right shift with signed saturation arithmetic and half the width on each half-word/word/double-word/quad element within this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of vector register vd.

VSSRANI.B.H:

```
for i in range(8):
    VR[vd].B[i] = SSRANS(VR[vj].H[i], ui4)
    VR[vd].B[i+8] = SSRANS(VR[vd].H[i], ui4)
```

VSSRANI.H.W:

```
for i in range(4):
    VR[vd].H[i] = SSRANS(VR[vj].W[i], ui5)
    VR[vd].H[i+4] = SSRANS(VR[vd].W[i], ui5)
```

VSSRANI.W.D:

```
for i in range(2):
    VR[vd].W[i] = SSRANS(VR[vj].D[i], ui6)
    VR[vd].W[i+2] = SSRANS(VR[vd].D[i], ui6)
```

VSSRANI.D.Q:

```
for i in range(1):
    VR[vd].D[i] = SSRANS(VR[vj].Q[i], ui7)
    VR[vd].D[i+1] = SSRANS(VR[vd].Q[i], ui7)
```

The XVSSRANI.{B.H/H.W/W.D/D.Q} instruction concatenates the lower half of vector register xd with the lower half of vector register xj to form a 256-bit vector, and performs an arithmetic right shift with signed saturation arithmetic and half the width on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of the lower half of vector register xd. The upper half of vector register xd is concatenated with the upper half of vector register xj to form a 256-bit vector, and performs an arithmetic right shift with signed saturation arithmetic and half the width on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The upper half of the result is sequentially written to the byte/half-word/word/double-word elements corresponding to the upper half of vector register xd.

XVSSRANI.B.H:

```
for i in range(8):
    XR[xd].B[i] = SSRANS(XR[xj].H[i], ui4)
    XR[xd].B[i+8] = SSRANS(XR[xd].H[i], ui4)
    XR[xd].B[i+16] = SSRANS(XR[xj].H[i+8], ui4)
    XR[xd].B[i+24] = SSRANS(XR[xd].H[i+8], ui4)
```

XVSSRANI.H.W:

```
for i in range(4) :
    XR[xd].H[i] = SSRANS(XR[xj].W[i], ui5)
    XR[xd].H[i+4] = SSRANS(XR[xd].W[i], ui5)
    XR[xd].H[i+8] = SSRANS(XR[xj].W[i+4], ui5)
    XR[xd].H[i+12] = SSRANS(XR[xd].W[i+4], ui5)
```

XVSSRANI.W.D:

```
for i in range(2):
    XR[xd].W[i] = SSRANS(XR[xj].D[i], ui6)
    XR[xd].W[i+2] = SSRANS(XR[xd].D[i], ui6)
    XR[xd].W[i+4] = SSRANS(XR[xj].D[i+2], ui6)
    XR[xd].W[i+6] = SSRANS(XR[xd].D[i+2], ui6)
```

XVSSRANI.D.Q:

```
for i in range(1) :
    XR[xd].D[i] = SSRANS(XR[xj].Q[i], ui7)
    XR[xd].D[i+1] = SSRANS(XR[xd].Q[i], ui7)
    XR[xd].D[i+2] = SSRANS(XR[xj].Q[i+1], ui7)
    XR[xd].D[i+3] = SSRANS(XR[xd].Q[i+1], ui7)
```

The VSSRLNI.{BU.H/HU.W/WU.D/DU.Q} instruction concatenates vector register vd and vector register vj to form a 256-bit vector, and performs a logical right shift with unsigned saturation arithmetic and half the width on each half-word/word/double-word/quad element within this vector. The shift amount

is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of vector register vd.

VSSRLNI.BU.H:

```
for i in range(8):
    VR[vd].B[i] = SSRLNU(VR[vj].H[i], ui4)
    VR[vd].B[i+8] = SSRLNU(VR[vd].H[i], ui4)
```

VSSRLNI.HU.W:

```
for i in range(4):
    VR[vd].H[i] = SSRLNU(VR[vj].W[i], ui5)
    VR[vd].H[i+4] = SSRLNU(VR[vd].W[i], ui5)
```

VSSRLNI.WU.D:

```
for i in range(2):
    VR[vd].W[i] = SSRLNU(VR[vj].D[i], ui6)
    VR[vd].W[i+2] = SSRLNU(VR[vd].D[i], ui6)
```

VSSRLNI.DU.Q:

```
for i in range(1):
    VR[vd].D[i] = SSRLNU(VR[vj].Q[i], ui7)
    VR[vd].D[i+1] = SSRLNU(VR[vd].Q[i], ui7)
```

The XVSSRLNI.{BU.H/HU.W/WU.D/DU.Q} instruction concatenates the lower half of vector register xd with the lower half of vector register xj to form a 256-bit vector, and performs a logical right shift with unsigned saturation arithmetic and half the width on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of the lower half of vector register xd. The upper half of vector register xd is concatenated with the upper half of vector register xj to form a 256-bit vector, and performs a logical right shift with unsigned saturation arithmetic and half the width on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The upper half of the result is sequentially written to the byte/half-word/word/double-word elements corresponding to the upper half of vector register xd.

XVSSRLNI.BU.H:

```
for i in range(8):
    XR[xd].B[i] = SSRLNU(XR[xj].H[i], ui4)
    XR[xd].B[i+8] = SSRLNU(XR[xd].H[i], ui4)
    XR[xd].B[i+16] = SSRLNU(XR[xj].H[i+8], ui4)
    XR[xd].B[i+24] = SSRLNU(XR[xd].H[i+8], ui4)
```

XVSSRLNI.HU.W:

```
for i in range(4) :
    XR[xd].H[i] = SSRLNU(XR[xj].W[i], ui5)
    XR[xd].H[i+4] = SSRLNU(XR[xd].W[i], ui5)
    XR[xd].H[i+8] = SSRLNU(XR[xj].W[i+4], ui5)
    XR[xd].H[i+12] = SSRLNU(XR[xd].W[i+4], ui5)
```

XVSSRLNI.WU.D:

```
for i in range(2) :
    XR[xd].W[i] = SSRLNU(XR[xj].D[i], ui6)
    XR[xd].W[i+2] = SSRLNU(XR[xd].D[i], ui6)
    XR[xd].W[i+4] = SSRLNU(XR[xj].D[i+2], ui6)
    XR[xd].W[i+6] = SSRLNU(XR[xd].D[i+2], ui6)
```

XVSSRLNI.DU.Q:

```
for i in range(1) :
    XR[xd].D[i] = SSRLNU(XR[xj].Q[i], ui7)
    XR[xd].D[i+1] = SSRLNU(XR[xd].Q[i], ui7)
    XR[xd].D[i+2] = SSRLNU(XR[xj].Q[i+1], ui7)
    XR[xd].D[i+3] = SSRLNU(XR[xd].Q[i+1], ui7)
```

The VSSRANI.{BU.H/HU.W/WU.D/DU.Q} instruction concatenates vector register vd and vector register vj to form a 256-bit vector, and performs an arithmetic right shift with unsigned saturation arithmetic and half the width on each half-word/word/double-word/quad element within this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of vector register vd.

VSSRANI.BU.H:

```
for i in range(8):
    VR[vd].B[i] = SSRANU(VR[vj].H[i], ui4)
    VR[vd].B[i+8] = SSRANU(VR[vd].H[i], ui4)
```

VSSRANI.HU.W:

```
for i in range(4):
    VR[vd].H[i] = SSRANU(VR[vj].W[i], ui5)
    VR[vd].H[i+4] = SSRANU(VR[vd].W[i], ui5)
```

VSSRANI.WU.D:

```
for i in range(2):
    VR[vd].W[i] = SSRANU(VR[vj].D[i], ui6)
    VR[vd].W[i+2] = SSRANU(VR[vd].D[i], ui6)
```

VSSRANI.DU.Q:

```
for i in range(1):
    VR[vd].D[i] = SSRANU(VR[vj].Q[i], ui7)
    VR[vd].D[i+1] = SSRANU(VR[vd].Q[i], ui7)
```

The XVSSRANI.{BU.H/HU.W/WU.D/DU.Q} instruction concatenates the lower half of vector register xd with the lower half of vector register xj to form a 256-bit vector, and performs an arithmetic right shift with unsigned saturation arithmetic and half the width on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of the lower half of vector register xd. The upper half of vector register xd is concatenated with the upper half of vector register xj to form a 256-bit vector, and performs an arithmetic right shift with unsigned saturation arithmetic and half the width on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The upper half of the result is sequentially written to the byte/half-word/word/double-word elements corresponding to the upper half of vector register xd.

XVSSRANI.BU.H:

```
for i in range(8):
    XR[xd].B[i] = SSRANU(XR[xj].H[i], ui4)
    XR[xd].B[i+8] = SSRANU(XR[xd].H[i], ui4)
    XR[xd].B[i+16] = SSRANU(XR[xj].H[i+8], ui4)
    XR[xd].B[i+24] = SSRANU(XR[xd].H[i+8], ui4)
```

XVSSRANI.HU.W:

```
for i in range(4) :
    XR[xd].H[i] = SSRANU(XR[xj].W[i], ui5)
    XR[xd].H[i+4] = SSRANU(XR[xd].W[i], ui5)
    XR[xd].H[i+8] = SSRANU(XR[xj].W[i+4], ui5)
    XR[xd].H[i+12] = SSRANU(XR[xd].W[i+4], ui5)
```

XVSSRANI.WU.D:

```
for i in range(2) :
    XR[xd].W[i] = SSRANU(XR[xj].D[i], ui6)
    XR[xd].W[i+2] = SSRANU(XR[xd].D[i], ui6)
    XR[xd].W[i+4] = SSRANU(XR[xj].D[i+2], ui6)
    XR[xd].W[i+6] = SSRANU(XR[xd].D[i+2], ui6)
```

XVSSRANI.DU.Q:

```
for i in range(1) :
    XR[xd].D[i] = SSRANU(XR[xj].Q[i], ui7)
    XR[xd].D[i+1] = SSRANU(XR[xd].Q[i], ui7)
    XR[xd].D[i+2] = SSRANU(XR[xj].Q[i+1], ui7)
    XR[xd].D[i+3] = SSRANU(XR[xd].Q[i+1], ui7)
```

1.2.2.12 [X]VSSRLRN.{B.H/H.W/W.D}, [X]VSSRARN.{B.H/H.W/W.D}, [X]VSSRLRN.{BU.H/HU.W/WU.D}, [X]VSSRARN.{BU.H/HU.W/WU.D}

Instruction Format:				
	vssrlrn.b.h	vd, vj, vk	xvssrlrn.b.h	xd, xj, xk
	vssrlrn.h.w	vd, vj, vk	xvssrlrn.h.w	xd, xj, xk
	vssrlrn.w.d	vd, vj, vk	xvssrlrn.w.d	xd, xj, xk
	vssrarn.b.h	vd, vj, vk	xvssrarn.b.h	xd, xj, xk
	vssrarn.h.w	vd, vj, vk	xvssrarn.h.w	xd, xj, xk
	vssrarn.w.d	vd, vj, vk	xvssrarn.w.d	xd, xj, xk
	vssrlrn.bu.h	vd, vj, vk	xvssrlrn.bu.h	xd, xj, xk
	vssrlrn.hu.w	vd, vj, vk	xvssrlrn.hu.w	xd, xj, xk
	vssrlrn.wu.d	vd, vj, vk	xvssrlrn.wu.d	xd, xj, xk
	vssrarn.bu.h	vd, vj, vk	xvssrarn.bu.h	xd, xj, xk
	vssrarn.hu.w	vd, vj, vk	xvssrarn.hu.w	xd, xj, xk
	vssrarn.wu.d	vd, vj, vk	xvssrarn.wu.d	xd, xj, xk

The [X]VSSRLRN.{B.H/H.W/W.D} instruction performs a logical right shift and signed saturation arithmetic with rounding to half the width on each half-word/word/double-word element in every 128 bits of vector registers vj/xj. The shift amount is derived from the corresponding half-word/word/doubleword elements at the corresponding position in vector register vk/xk. The lower half of the result is sequentially written into the byte/half-word/word elements in the lower half of every 128 bits in vector registers vd/xd, and the upper half of every 128 bits in vd/xd is set to 0.

VSSRLRN.B.H:

```
for i in range(8):
    VR[vd].B[i] = SSRLRNS(VR[vj].H[i], VR[vk].H[i] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
```

VSSRLRN.H.W:

```
for i in range(4):
    VR[vd].H[i] = SSRLRNS(VR[vj].W[i], VR[vk].W[i] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
```

VSSRLRN.W.D:

```
for i in range(2):
    VR[vd].W[i] = SSRLRNS(VR[vj].D[i], VR[vk].D[i] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
```

XVSSRLRN.B.H:

```
for i in range(8):
    VR[vd].B[i] = SSRLRNS(VR[vj].H[i], VR[vk].H[i] % 16)
    VR[vd].B[i+16] = SSRLRNS(VR[vj].H[i+8], VR[vk].H[i+8] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
    VR[vd].B[i+16] = 0
```

XVSSRLRN.H.W:

```
for i in range(4):
    VR[vd].H[i] = SSRLRNS(VR[vj].W[i], VR[vk].W[i] % 32)
    VR[vd].H[i+8] = SSRLRNS(VR[vj].W[i+4], VR[vk].W[i+4] % 32)
for i in range(4, 8, 1) :
    VR[vd].H[i] = 0
    VR[vd].H[i+8] = 0
```

XVSSRLRN.W.D:

```
for i in range(2):
    VR[vd].W[i] = SSRLRNS(VR[vj].D[i], VR[vk].D[i] % 64)
    VR[vd].W[i+4] = SSRLRNS(VR[vj].D[i+2], VR[vk].D[i+2] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
    VR[vd].W[i+4] = 0
```

The [X]VSSRARN.{B.H/H.W/W.D} instruction performs an arithmetic right shift and signed saturation arithmetic with rounding to half the width on each half-word/word/double-word element in every 128 bits of vector registers vj/xj. The shift amount is derived from the corresponding half-word/word/double-word elements at the corresponding position in vector register vk/xk. The lower half of the result is sequentially written into the byte/half-word/word elements in the lower half of every 128 bits in vector registers vd/xd, and the upper half of every 128 bits in vd/xd is set to 0.

VSSRARN.B.H:

```
for i in range(8):
    VR[vd].B[i] = SSRARNS(VR[vj].H[i], VR[vk].H[i] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
```

VSSRARN.H.W:

```
for i in range(4):
    VR[vd].H[i] = SSRARNS(VR[vj].W[i], VR[vk].W[i] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
```

VSSRARN.W.D:

```
for i in range(2):
    VR[vd].W[i] = SSRARNS(VR[vj].D[i], VR[vk].D[i] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
```

XVSSRARN.B.H:

```
for i in range(8):
    VR[vd].B[i] = SSRARNS(VR[vj].H[i], VR[vk].H[i] % 16)
    VR[vd].B[i+16] = SSRARNS(VR[vj].H[i+8], VR[vk].H[i+8] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
    VR[vd].B[i+16] = 0
```

XVSSRARN.H.W:

```
for i in range(4):
    VR[vd].H[i] = SSRARNS(VR[vj].W[i], VR[vk].W[i] % 32)
    VR[vd].H[i+8] = SSRARNS(VR[vj].W[i+4], VR[vk].W[i+4] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
    VR[vd].H[i+8] = 0
```

XVSSRARN.W.D:

```
for i in range(2):
    VR[vd].W[i] = SSRARNS(VR[vj].D[i], VR[vk].D[i] % 64)
    VR[vd].W[i+4] = SSRARNS(VR[vj].D[i+2], VR[vk].D[i+2] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
    VR[vd].W[i+4] = 0
```

The [X]VSSRLRN.{BU.H/HU.W/WU.D} instruction performs a logical right shift and unsigned saturation arithmetic with rounding to half the width on each half-word/word/double-word element in every 128 bits of vector registers vj/xj. The shift amount is derived from the corresponding half-word/word/double-word elements at the corresponding position in vector register vk/xk. The lower half of the result is sequentially written into the byte/half-word/word elements in the lower half of every 128 bits in vector registers vd/xd, and the upper half of every 128 bits in vd/xd is set to 0.

VSSRLRN.BU.H:

```
for i in range(8):
    VR[vd].B[i] = SSRLRNU(VR[vj].H[i], VR[vk].H[i] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
```

VSSRLRN.HU.W:

```
for i in range(4):
    VR[vd].H[i] = SSRLRNU(VR[vj].W[i], VR[vk].W[i] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
```

VSSRLRN.WU.D:

```
for i in range(2):
    VR[vd].W[i] = SSRLRNU(VR[vj].D[i], VR[vk].D[i] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
```

XVSSRLRN.BU.H:

```
for i in range(8):
    VR[vd].B[i] = SSRLRNU(VR[vj].H[i], VR[vk].H[i] % 16)
    VR[vd].B[i+16] = SSRLRNU(VR[vj].H[i+8], VR[vk].H[i+8] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
    VR[vd].B[i+16] = 0
```

XVSSRLRN.HU.W:

```
for i in range(4):
    VR[vd].H[i] = SSRLRNU(VR[vj].W[i], VR[vk].W[i] % 32)
    VR[vd].H[i+8] = SSRLRNU(VR[vj].W[i+4], VR[vk].W[i+4] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
    VR[vd].H[i+8] = 0
```

XVSSRLRN.WU.D:

```
for i in range(2):
    VR[vd].W[i] = SSRLRNU(VR[vj].D[i], VR[vk].D[i] % 64)
    VR[vd].W[i+4] = SSRLRNU(VR[vj].D[i+2], VR[vk].D[i+2] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
    VR[vd].W[i+4] = 0
```

The [X]VSSRARN.{BU.H/HU.W/WU.D} instruction performs an arithmetic right shift and unsigned saturation arithmetic with rounding to half the width on each half-word/word/double-word element in every 128 bits of vector registers vj/xj. The shift amount is derived from the corresponding half-word/word/double-word elements at the corresponding position in vector register vk/xk. The lower half of the result is sequentially written into the byte/half-

word/word elements in the lower half of every 128 bits in vector registers vd/xd, and the upper half of every 128 bits in vd/xd is set to 0.

VSSRARN.BU.H:

```
for i in range(8):
    VR[vd].B[i] = SSRARNU(VR[vj].H[i], VR[vk].H[i] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
```

VSSRARN.HU.W:

```
for i in range(4):
    VR[vd].H[i] = SSRARNU(VR[vj].W[i], VR[vk].W[i] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
```

VSSRARN.WU.D:

```
for i in range(2):
    VR[vd].W[i] = SSRARNU(VR[vj].D[i], VR[vk].D[i] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
```

XVSSRARN.BU.H:

```
for i in range(8):
    VR[vd].B[i] = SSRARNU(VR[vj].H[i], VR[vk].H[i] % 16)
    VR[vd].B[i+16] = SSRARNU(VR[vj].H[i+8], VR[vk].H[i+8] % 16)
for i in range(8, 16, 1):
    VR[vd].B[i] = 0
    VR[vd].B[i+16] = 0
```

XVSSRARN.HU.W:

```
for i in range(4):
    VR[vd].H[i] = SSRARNU(VR[vj].W[i], VR[vk].W[i] % 32)
    VR[vd].H[i+8] = SSRARNU(VR[vj].W[i+4], VR[vk].W[i+4] % 32)
for i in range(4, 8, 1):
    VR[vd].H[i] = 0
    VR[vd].H[i+8] = 0
```

XVSSRARN.WU.D:

```
for i in range(2):
    VR[vd].W[i] = SSRARNU(VR[vj].D[i], VR[vk].D[i] % 64)
    VR[vd].W[i+4] = SSRARNU(VR[vj].D[i+2], VR[vk].D[i+2] % 64)
for i in range(2, 4, 1):
    VR[vd].W[i] = 0
    VR[vd].W[i+4] = 0
```

1.2.2.13 [X]VSSRLRNI.{B.H/H.W/W.D/D.Q}, [X]VSSRARNI.{B.H/H.W/W.D/D.Q}, [X]VSSRLRNI.{BU.H/HU.W/WU.D/DU.Q}, [X]VSSRARNI.{BU.H/HU.W/WU.D/DU.Q}

Instruction Format:				
	vssrlrni.b.h	vd, vj, ui4	xvssrlrni.b.h	xd, xj, ui4
	vssrlrni.h.w	vd, vj, ui5	xvssrlrni.h.w	xd, xj, ui5
	vssrlrni.w.d	vd, vj, ui6	xvssrlrni.w.d	xd, xj, ui6
	vssrlrni.d.q	vd, vj, ui7	xvssrlrni.d.q	xd, xj, ui7
	vssrarni.b.h	vd, vj, ui4	xvssrarni.b.h	xd, xj, ui4
	vssrarni.h.w	vd, vj, ui5	xvssrarni.h.w	xd, xj, ui5
	vssrarni.w.d	vd, vj, ui6	xvssrarni.w.d	xd, xj, ui6
	vssrarni.d.q	vd, vj, ui7	xvssrarni.d.q	xd, xj, ui7
	vssrlrni.bu.h	vd, vj,ui4	xvssrlrni.bu.h	xd, xj, ui4
	vssrlrni.hu.w	vd, vj, ui5	xvssrlrni.hu.w	xd, xj, ui5
	vssrlrni.wu.d	vd, vj, ui6	xvssrlrni.wu.d	xd, xj, ui6
	vssrlrni.du.q	vd, vj, ui7	xvssrlrni.du.q	xd, xj, ui7

vssrarni.bu.h	vd, vj, ui4	xvssrarni.bu.h	xd, xj, ui4
vssrarni.hu.w	vd, vj, ui5	xvssrarni.hu.w	xd, xj, ui5
vssrarni.wu.d	vd, vj, ui6	xvssrarni.wu.d	xd, xj, ui6
vssrarni.du.q	vd, vj, ui7	xvssrarni.du.q	xd, xj, ui7

The VSSRLRNI.{B.H/H.W/W.D/D.Q} instruction concatenates vector register vd and vector register vj to form a 256-bit vector, and performs a logical right shift and signed saturation arithmetic with rounding to half the width on each half-word/word/double-word/quad element within this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of vector register vd.

VSSRLRNI.B.H:

```
for i in range(8):
    VR[vd].B[i] = SSRLRNS(VR[vj].H[i], ui4)
    VR[vd].B[i+8] = SSRLRNS(VR[vd].H[i], ui4)
```

VSSRLRNI.H.W:

```
for i in range(4):
    VR[vd].H[i] = SSRLRNS(VR[vj].W[i], ui5)
    VR[vd].H[i+4] = SSRLRNS(VR[vd].W[i], ui5)
```

VSSRLRNI.W.D:

```
for i in range(2) :
    VR[vd].W[i] = SSRLRNS(VR[vj].D[i], ui6)
    VR[vd].W[i+2] = SSRLRNS(VR[vd].D[i], ui6)
```

VSSRLRNI.D.Q:

```
for i in range(1):
    VR[vd].D[i] = SSRLRNS(VR[vj].Q[i], ui7)
    VR[vd].D[i+1] = SSRLRNS(VR[vd].Q[i], ui7)
```

The XVSSRLRNI.{B.H/H.W/W.D/D.Q} instruction concatenates the lower half of vector register xd with the lower half of vector register xj to form a 256-bit vector, and performs a logical right shift and signed saturation arithmetic with rounding to half the width on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of the lower half of vector register xd. The upper half of vector register xd is concatenated with the upper half of vector register xj to form a 256-bit vector, and performs a logical right shift and signed saturation arithmetic with rounding to half the width on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The upper half of the result is sequentially written to the byte/half-word/word/double-word elements corresponding to the upper half of vector register xd.

XVSSRLRNI.B.H:

```
for i in range(8):
    XR[xd].B[i] = SSRLRNS(XR[xj].H[i], ui4)
    XR[xd].B[i+8] = SSRLRNS(XR[xd].H[i], ui4)
    XR[xd].B[i+16] = SSRLRNS(XR[xj].H[i+8], ui4)
    XR[xd].B[i+24] = SSRLRNS(XR[xd].H[i+8], ui4)
```

XVSSRLRNI.H.W:

```
for i in range(4) :
    XR[xd].H[i] = SSRLRNS(XR[xj].W[i], ui5)
    XR[xd].H[i+4] = SSRLRNS(XR[xd].W[i], ui5)
    XR[xd].H[i+8] = SSRLRNS(XR[xj].W[i+4], ui5)
    XR[xd].H[i+12] = SSRLRNS(XR[xd].W[i+4], ui5)
```

XVSSRLRNI.W.D:

```
for i in range(2) :
    XR[xd].W[i] = SSRLRNS(XR[xj].D[i], ui6)
    XR[xd].W[i+2] = SSRLRNS(XR[xd].D[i], ui6)
    XR[xd].W[i+4] = SSRLRNS(XR[xj].D[i+2], ui6)
    XR[xd].W[i+6] = SSRLRNS(XR[xd].D[i+2], ui6)
```

XVSSRLRNI.D.Q:

```
for i in range(1) :
    XR[xd].D[i] = SSRLRNS(XR[xj].Q[i], ui7)
    XR[xd].D[i+1] = SSRLRNS(XR[xd].Q[i], ui7)
    XR[xd].D[i+2] = SSRLRNS(XR[xj].Q[i+1], ui7)
    XR[xd].D[i+3] = SSRLRNS(XR[xd].Q[i+1], ui7)
```

The VSSRARNI.{B.H/H.W/W.D/D.Q} instruction concatenates vector register vd and vector register vj to form a 256-bit vector, and performs an arithmetic right shift and signed saturation arithmetic with rounding to half the width on each half-word/word/double-word/quad element within this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of vector register vd.

VSSRARNI.B.H:

```
for i in range(8):
    VR[vd].B[i] = SSRARNS(VR[vj].H[i], ui4)
    VR[vd].B[i+8] = SSRARNS(VR[vd].H[i], ui4)
```

VSSRARNI.H.W:

```
for i in range(4):
    VR[vd].H[i] = SSRARNS(VR[vj].W[i], ui5)
    VR[vd].H[i+4] = SSRARNS(VR[vd].W[i], ui5)
```

VSSRARNI.W.D:

```
for i in range(2):
    VR[vd].W[i] = SSRARNS(VR[vj].D[i], ui6)
    VR[vd].W[i+2] = SSRARNS(VR[vd].D[i], ui6)
```

VSSRARNI.D.Q:

```
for i in range(1):
    VR[vd].D[i] = SSRARNS(VR[vj].Q[i], ui7)
    VR[vd].D[i+1] = SSRARNS(VR[vd].Q[i], ui7)
```

The XVSSRARNI.{B.H/H.W/W.D/D.Q} instruction concatenates the lower half of vector register xd with the lower half of vector register xj to form a 256-bit vector, and performs an arithmetic right shift and signed saturation arithmetic with rounding to half the width on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of the lower half of vector register xd. The upper half of vector register xd is concatenated with the upper half of vector register xj to form a 256-bit vector, and performs an arithmetic right shift and signed saturation arithmetic with rounding to half the width on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The upper half of the result is sequentially written to the byte/half-word/word/double-word elements corresponding to the upper half of vector register xd.

XVSSRARNI.B.H:

```
for i in range(8):
    XR[xd].B[i] = SSRARNS(XR[xj].H[i], ui4)
    XR[xd].B[i+8] = SSRARNS(XR[xd].H[i], ui4)
    XR[xd].B[i+16] = SSRARNS(XR[xj].H[i+8], ui4)
    XR[xd].B[i+24] = SSRARNS(XR[xd].H[i+8], ui4)
```

XVSSRARNI.H.W:

```
for i in range(4) :
    XR[xd].H[i] = SSRARNS(XR[xj].W[i], ui5)
    XR[xd].H[i+4] = SSRARNS(XR[xd].W[i], ui5)
    XR[xd].H[i+8] = SSRARNS(XR[xj].W[i+4], ui5)
    XR[xd].H[i+12] = SSRARNS(XR[xd].W[i+4], ui5)
```

XVSSRARNI.W.D:

```
for i in range(2) :
    XR[xd].W[i] = SSRARNS(XR[xj].D[i], ui6)
    XR[xd].W[i+2] = SSRARNS(XR[xd].D[i], ui6)
    XR[xd].W[i+4] = SSRARNS(XR[xj].D[i+2], ui6)
    XR[xd].W[i+6] = SSRARNS(XR[xd].D[i+2], ui6)
```

XVSSRARNI.D.Q:

```
for i in range(1) :
    XR[xd].D[i] = SSRARNS(XR[xj].Q[i], ui7)
    XR[xd].D[i+1] = SSRARNS(XR[xd].Q[i], ui7)
    XR[xd].D[i+2] = SSRARNS(XR[xj].Q[i+1], ui7)
    XR[xd].D[i+3] = SSRARNS(XR[xd].Q[i+1], ui7)
```

The VSSRLRNI.{BU.H/HU.W/WU.D/DU.Q} instruction concatenates vector register vd and vector register vj to form a 256-bit vector, and performs a logical right shift and unsigned saturation arithmetic with rounding to half the width on each half-word/word/double-word/quad element within this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of vector register vd.

VSSRLRNI.BU.H:

```
for i in range(8):
    VR[vd].B[i] = SSRLRNU(VR[vj].H[i], ui4)
    VR[vd].B[i+8] = SSRLRNU(VR[vd].H[i], ui4)
```

VSSRLRNI.HU.W:

```
for i in range(4) :
    VR[vd].H[i] = SSRLRNU(VR[vj].W[i], ui5)
    VR[vd].H[i+4] = SSRLRNU(VR[vd].W[i], ui5)
```

VSSRLRNI.WU.D:

```
for i in range(2):
    VR[vd].W[i] = SSRLRNU(VR[vj].D[i], ui6)
    VR[vd].W[i+2] = SSRLRNU(VR[vd].D[i], ui6)
```

VSSRLRNI.DU.Q:

```
for i in range(1):
    VR[vd].D[i] = SSRLRNU(VR[vj].Q[i], ui7)
    VR[vd].D[i+1] = SSRLRNU(VR[vd].Q[i], ui7)
```

The XVSSRLNI.{BU.H/HU.W/WU.D/DU.Q} instruction concatenates the lower half of vector register xd with the lower half of vector register xj to form a 256-bit vector, and performs a logical right shift and unsigned saturation arithmetic with rounding to half the width on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of the lower half of vector register xd. The upper half of vector register xd is concatenated with the upper half of vector register xj to form a 256-bit vector, and performs a logical right shift and unsigned saturation arithmetic with rounding to half the width on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The upper half of the result is sequentially written to the byte/half-word/word/double-word elements corresponding to the upper half of vector register xd.

XVSSRLRNI.BU.H:

```
for i in range(8):
    XR[xd].B[i] = SSRLRNU(XR[xj].H[i], ui4)
    XR[xd].B[i+8] = SSRLRNU(XR[xd].H[i], ui4)
    XR[xd].B[i+16] = SSRLRNU(XR[xj].H[i+8], ui4)
    XR[xd].B[i+24] = SSRLRNU(XR[xd].H[i+8], ui4)
```

XVSSRLRNI.HU.W:

```
for i in range(4) :
    XR[xd].H[i] = SSRLRNU(XR[xj].W[i], ui5)
    XR[xd].H[i+4] = SSRLRNU(XR[xd].W[i], ui5)
    XR[xd].H[i+8] = SSRLRNU(XR[xj].W[i+4], ui5)
    XR[xd].H[i+12] = SSRLRNU(XR[xd].W[i+4], ui5)
```

XVSSRLRNI.WU.D:

```
for i in range(2):
    XR[xd].W[i] = SSRLRNU(XR[xj].D[i], ui6)
    XR[xd].W[i+2] = SSRLRNU(XR[xd].D[i], ui6)
    XR[xd].W[i+4] = SSRLRNU(XR[xj].D[i+2], ui6)
    XR[xd].W[i+6] = SSRLRNU(XR[xd].D[i+2], ui6)
```

XVSSRLRNI.DU.Q:

```
for i in range(1) :
    XR[xd].D[i] = SSRLRNU(XR[xj].Q[i], ui7)
    XR[xd].D[i+1] = SSRLRNU(XR[xd].Q[i], ui7)
    XR[xd].D[i+2] = SSRLRNU(XR[xj].Q[i+1], ui7)
    XR[xd].D[i+3] = SSRLRNU(XR[xd].Q[i+1], ui7)
```

The VSSRARNI.{BU.H/HU.W/WU.D/DU.Q} instruction concatenates vector register vd and vector register vj to form a 256-bit vector, and performs an arithmetic right shift and unsigned saturation arithmetic with rounding to half the width on each half-word/word/double-word/quad element within this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of vector register vd.

VSSRARNI.BU.H:

```
for i in range(8):
    VR[vd].B[i] = SSRARNU(VR[vj].H[i], ui4)
    VR[vd].B[i+8] = SSRARNU(VR[vd].H[i], ui4)
```

VSSRARNI.HU.W:

```
for i in range(4):
    VR[vd].H[i] = SSRARNU(VR[vj].W[i], ui5)
    VR[vd].H[i+4] = SSRARNU(VR[vd].W[i], ui5)
```

VSSRARNI.WU.D:

```
for i in range(2):
    VR[vd].W[i] = SSRARNU(VR[vj].D[i], ui6)
    VR[vd].W[i+2] = SSRARNU(VR[vd].D[i], ui6)
```

VSSRARNI.DU.Q:

```
for i in range(1):
    VR[vd].D[i] = SSRARNU(VR[vj].Q[i], ui7)
    VR[vd].D[i+1] = SSRARNU(VR[vd].Q[i], ui7)
```

The XVSSRARNI.{BU.H/HU.W/WU.D/DU.Q} instruction concatenates the lower half of vector register xd with the lower half of vector register xj to form a 256-bit vector, and performs an arithmetic right shift and unsigned saturation arithmetic with rounding to half the width on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The lower half of the result is sequentially written into the corresponding byte/half-word/word/double-word elements of the lower half of vector register xd. The upper half of vector register xd is concatenated with the upper half of vector register xj to form a 256-bit vector, and performs an arithmetic right shift and unsigned saturation arithmetic with rounding to half the width on each half-word/word/double-word/quad-word element in this vector. The shift amount is derived from an immediate value ui4/ui5/ui6/ui7. The upper half of the result is sequentially written to the byte/half-word/word/double-word elements corresponding to the upper half of vector register xd.

XVSSRARNI.BU.H:

```
for i in range(8):
    XR[xd].B[i] = SSRARNU(XR[xj].H[i], ui4)
    XR[xd].B[i+8] = SSRARNU(XR[xd].H[i], ui4)
    XR[xd].B[i+16] = SSRARNU(XR[xj].H[i+8], ui4)
    XR[xd].B[i+24] = SSRARNU(XR[xd].H[i+8], ui4)
```

XVSSRARNI.HU.W:

```
for i in range(4) :
    XR[xd].H[i] = SSRARNU(XR[xj].W[i], ui5)
    XR[xd].H[i+4] = SSRARNU(XR[xd].W[i], ui5)
    XR[xd].H[i+8] = SSRARNU(XR[xj].W[i+4], ui5)
    XR[xd].H[i+12] = SSRARNU(XR[xd].W[i+4], ui5)
```

XVSSRARNI.WU.D:

```
for i in range(2) :
    XR[xd].W[i] = SSRARNU(XR[xj].D[i], ui6)
    XR[xd].W[i+2] = SSRARNU(XR[xd].D[i], ui6)
    XR[xd].W[i+4] = SSRARNU(XR[xj].D[i+2], ui6)
    XR[xd].W[i+6] = SSRARNU(XR[xd].D[i+2], ui6)
```

XVSSRARNI.DU.Q:

```
for i in range(1) :
    XR[xd].D[i] = SSRARNU(XR[xj].Q[i], ui7)
    XR[xd].D[i+1] = SSRARNU(XR[xd].Q[i], ui7)
    XR[xd].D[i+2] = SSRARNU(XR[xj].Q[i+1], ui7)
    XR[xd].D[i+3] = SSRARNU(XR[xd].Q[i+1], ui7)
```

1.2.2.14 [X]VCLO.{B/H/W/D}, [X]VCLZ.{B/H/W/D}

Instruction Format:	vclo.b	vd, vj	xvclo.b	xd, xj
	vclo.h	vd, vj	xvclo.h	xd, xj
	vclo.w	vd, vj	xvclo.w	xd, xj
	vclo.d	vd, vj	xvclo.d	xd, xj
	vclz.b	vd, vj	xvclz.b	xd, xj
	vclz.h	vd, vj	xvclz.h	xd, xj
	vclz.w	vd, vj	xvclz.w	xd, xj
	vclz.d	vd, vj	xvclz.d	xd, xj

The [X]VCLO.{B/H/W/D} instruction counts the number of consecutive 1s starting from the highest bit for each byte/half-word/word/double-word element in vector register vj/xj. The result is written to the corresponding byte/half-word/word/double-word elements in vector register vd/xd.

VCLO.B:

```
for i in range(16):
    VR[vd].B[i] = CLO(VR[vj].B[i])
```

VCLO.H:

```
for i in range(8):
    VR[vd].H[i] = CLO(VR[vj].H[i])
```

VCLO.W:

```
for i in range(4):
    VR[vd].W[i] = CLO(VR[vj].W[i])
```

VCLO.D:

```
for i in range(2):
    VR[vd].D[i] = CLO(VR[vj].D[i])
```

XVCLO.B:

```
for i in range(32) :
    XR[xd].B[i] = CLO(XR[xj].B[i])
```

XVCLO.H:

```
for i in range(16) :
    XR[xd].H[i] = CLO(XR[xj].H[i])
```

XVCLO.W:

```
for i in range(8):
    XR[xd].W[i] = CLO(XR[xj].W[i])
```

XVCLO.D:

```
for i in range(4):
    XR[xd].D[i] = CLO(XR[xj].D[i])
```

The [X]VCLZ.{B/H/W/D} counts the number of consecutive 0s starting from the highest bit in each byte/half-word/word/double-word element of vector register vj/xj. The result is written to the corresponding byte/half-word/word/double-word elements in vector register vd/xd.

VCLZ.B:

```
for i in range(16):
    VR[vd].B[i] = CLZ(VR[vj].B[i])
```

VCLZ.H:

```
for i in range(8):
    VR[vd].H[i] = CLZ(VR[vj].H[i])
```

VCLZ.W:

```
for i in range(4):
    VR[vd].W[i] = CLZ(VR[vj].W[i])
```

VCLZ.D:

```
for i in range(2):
    VR[vd].D[i] = CLZ(VR[vj].D[i])
```

XVCLZ.B:

```
for i in range(32) :
    XR[xd].B[i] = CLZ(XR[xj].B[i])
```

XVCLZ.H:

```
for i in range(16) :
    XR[xd].H[i] = CLZ(XR[xj].H[i])
```

XVCLZ.W:

```
for i in range(8):
    XR[xj].W[i] = CLZ(XR[i].W[xj])
```

XVCLZ.D:

```
for i in range(4):
    XR[xd].D[i] = CLZ(XR[xj].D[i])
```

1.2.2.15 [X]VPCNT.{B/H/W/D}

Instruction Format:	vpcnt.b	vd, vj		xvpcnt.b	xd, xj
	vpcnt.h	vd, vj		xvpcnt.h	xd, xj
	vpcnt.w	vd, vj		xvpcnt.w	xd, xj
	vpcnt.d	vd, vj		xvpcnt.d	xd, xj

The [X] The VPCNT.{B/H/W/D} instruction counts the number of 1s in each byte/half-word/word/double-word element of vector register vj/xj. The result is written to the corresponding byte/half-word/word/double-word elements in vector register vd/xd.

VPCNT.B:

```
for i in range(16) :
    VR[vd].B[i] = PCNT(VR[vj].B[i])
```

VPCNT.H:

```
for i in range(8):
    VR[vd].H[i] = PCNT(VR[vj].H[i])
```

VPCNT.W:

```
for i in range(4):
    VR[vd].W[i] = PCNT(VR[vj].W[i])
```

VPCNT.D:

```
for i in range(2):
    VR[vd].D[i] = PCNT(VR[vj].D[i])
```

XVPCNT.B:

```
for i in range(32):
    XR[xd].B[i] = PCNT(XR[xj].B[i])
```

XVPCNT.H:

```
for i in range(16):
    XR[xd].H[i] = PCNT(XR[xj].H[i])
```

XVPCNT.W:

```
for i in range(8) :
    XR[xd].W[i] = PCNT(XR[xj].W[i])
```

XVPCNT.D:

```
for i in range(4):
    XR[xd].D[i] = PCNT(XR[xj].D[i])
```

1.2.2.16 [X]VBITCLR[I].{B/H/W/D}, [X]VBITSET[I].{B/H/W/D}, [X]VBITREV[I].{B/H/W/D}

Instruction Format:				
	vbitclr.b	vd, vj, vk	xvbitclr.b	xd, xj, xk
	vbitclr.h	vd, vj, vk	xvbitclr.h	xd, xj, xk
	vbitclr.w	vd, vj, vk	xvbitclr.w	xd, xj, xk
	vbitclr.d	vd, vj, vk	xvbitclr.d	xd, xj, xk
	vbitclri.b	vd, vj, ui3	xvbitclri.b	xd, xj, ui3
	vbitclri.h	vd, vj, ui4	xvbitclri.h	xd, xj, ui4
	vbitclri.w	vd, vj, ui5	xvbitclri.w	xd, xj, ui5
	vbitclri.d	vd, vj, ui6	xvbitclri.d	xd, xj, ui6
	vbitset.b	vd, vj, vk	xvbitset.b	xd, xj, xk
	vbitset.h	vd, vj, vk	xvbitset.h	xd, xj, xk
	vbitset.w	vd, vj, vk	xvbitset.w	xd, xj, xk
	vbitset.d	vd, vj, vk	xvbitset.d	xd, xj, xk
	vbitseti.b	vd, vj, ui3	xvbitseti.b	xd, xj, ui3
	vbitseti.h	vd, vj, ui4	xvbitseti.h	xd, xj, ui4
	vbitseti.w	vd, vj, ui5	xvbitseti.w	xd, xj, ui5
	vbitseti.d	vd, vj, ui6	xvbitseti.d	xd, xj, ui6
	vbitrev.b	vd, vj, vk	xvbitrev.b	xd, xj, xk
	vbitrev.h	vd, vj, vk	xvbitrev.h	xd, xj, xk
	vbitrev.w	vd, vj, vk	xvbitrev.w	xd, xj, xk
	vbitrev.d	vd, vj, vk	xvbitrev.d	xd, xj, xk
	vbitrevi.b	vd, vj, ui3	xvbitrevi.b	xd, xj, ui3
	vbitrevi.h	vd, vj, ui4	xvbitrevi.h	xd, xj, ui4
	vbitrevi.w	vd, vj, ui5	xvbitrevi.w	xd, xj, ui5
	vbitrevi.d	vd, vj, ui6	xvbitrevi.d	xd, xj, ui6

The [X]VBITCLR.{B/H/W/D} instruction clears a specified bit to 0 in the byte/half-word/word/double-word elements of vector register vj/xj. The index of the bit to be cleared is derived from the byte/half-word/word/double-word elements at the corresponding position in vector register vk/xk.

VBITCLR.B:

```
for i in range(16):
    VR[vd].B[i] = Bit_Clear(VR[vj].B[i], VR[vk].B[i] % 8)
```

VBITCLR.H:

```
for i in range(8):
    VR[vd].H[i] = Bit_Clear(VR[vj].H[i], VR[vk].H[i] % 16)
```

VBITCLR.W:

```
for i in range(4):
    VR[vd].W[i] = Bit_Clear(VR[vj].W[i], VR[vk].W[i] % 32)
```

VBITCLR.D:

```
for i in range(2):
    VR[vd].D[i] = Bit_Clear(VR[vj].D[i], VR[vk].D[i] % 64)
```

XVBITCLR.B:

```
for i in range(32):
    XR[xd].B[i] = Bit_Clear(XR[xj].B[i], XR[xk].B[i] % 8)
```

XVBITCLR.H:

```
for i in range(16):
    XR[xd].H[i] = Bit_Clear(XR[xj].H[i], XR[xk].H[i] % 16)
```

XVBITCLR.W:

```
for i in range(8):
    XR[xd].W[i] = Bit_Clear(XR[xj].W[i], XR[xk].W[i] % 32)
```

XVBITCLR.D:

```
for i in range(4):
    XR[xd].D[i] = Bit_Clear(XR[xj].D[i], XR[xk].D[i] % 64)
```

The [X]VBITCLRI.{B/H/W/D} clears the specified bit to 0 in the byte/half-word/word/double-word elements of vector register vj/xj. The index of the bit to be cleared is derived from the immediate value ui3/ui4/ui5/ui6 in the instruction.

VBITCLRI.B:

```
for i in range(16):
    VR[vd].B[i] = Bit_Clear(VR[vj].B[i], ui3)
```

VBITCLRI.H:

```
for i in range(8):
    VR[vd].H[i] = Bit_Clear(VR[vj].H[i], ui4)
```

VBITCLRI.W:

```
for i in range(4):
    VR[vd].W[i] = Bit_Clear(VR[vj].W[i], ui5)
```

VBITCLRI.D:

```
for i in range(2):
    VR[vd].D[i] = Bit_Clear(VR[vj].D[i], ui6)
```

XVBITCLRI.B:

```
for i in range(32) :
    XR[xd].B[i] = Bit_Clear(XR[xj].B[i], ui3)
```

XVBITCLRI.H:

```
for i in range(16):
    XR[xd].H[i] = Bit_Clear(XR[xj].H[i], ui4)
```

XVBITCLRI.W:

```
for i in range(8):
    XR[xd].W[i] = Bit_Clear(XR[xj].W[i], ui5)
```

XVBITCLRI.D:

```
for i in range(4):
    XR[xd].D[i] = Bit_Clear(XR[xj].D[i], ui6)
```

The [X]VBITSET.{B/H/W/D} instruction sets a specified bit to 1 in the byte/half-word/word/double-word elements of vector register vj/xj. The index of the bit to be set is derived from the byte/half-word/word/double-word elements at the corresponding position in vector register vk/xk.

VBITSET.B:

```
for i in range(16):
    VR[vd].B[i] = Bit_Set(VR[vj].B[i], VR[vk].B[i] % 8)
```

VBITSET.H:

```
for i in range(8):
    VR[vd].H[i] = Bit_Set(VR[vj].H[i], VR[vk].H[i] % 16)
```

VBITSET.W:

```
for i in range(4):
    VR[vd].W[i] = Bit_Set(VR[vj].W[i], VR[vk].W[i] % 32)
```

VBITSET.D:

```
for i in range(2):
    VR[vd].D[i] = Bit_Set(VR[vj].D[i], VR[vk].D[i] % 64)
```

XVBITSET.B:

```
for i in range(32):
    XR[xd].B[i] = Bit_Set(XR[xj].B[i], XR[xk].B[i] % 8)
```

XVBITSET.H:

```
for i in range(16):
    XR[xd].H[i] = Bit_Set(XR[xj].H[i], XR[xk].H[i] % 16)
```

XVBITSET.W:

```
for i in range(8):
    XR[xd].W[i] = Bit_Set(XR[xj].W[i], XR[xk].W[i] % 32)
```

XVBITSET.D:

```
for i in range(4):
    XR[xd].D[i] = Bit_Set(XR[xj].D[i], XR[xk].D[i] % 64)
```

The [X]VBITSETI.{B/H/W/D} instruction sets a specified bit to 1 in the byte/half-word/word/double-word elements of vector register vj/xj. The index of the bit to be set is derived from the immediate value ui3/ui4/ui5/ui6 in the instruction.

VBITSETI.B:

```
for i in range(16):
    VR[vd].B[i] = Bit_Set(VR[vj].B[i], ui3)
```

VBITSETI.H:

```
for i in range(8):
    VR[vd].H[i] = Bit_Set(VR[vj].H[i], ui4)
```

VBITSETI.W:

```
for i in range(4):
    VR[vd].W[i] = Bit_Set(VR[vj].W[i], ui5)
```

VBITSETI.D:

```
for i in range(2):
    VR[vd].D[i] = Bit_Set(VR[vj].D[i], ui6)
```

XVBITSETI.B:

```
for i in range(32):
    XR[xd].B[i] = Bit_Set(XR[xj].B[i], ui3)
```

XVBITSETI.H:

```
for i in range(16):
    XR[xd].H[i] = Bit_Set(XR[xj].H[i], ui4)
```

XVBITSETI.W:

```
for i in range(8):
    XR[xd].W[i] = Bit_Set(XR[xj].W[i], ui5)
```

XVBITSETI.D:

```
for i in range(4):
    XR[xd].D[i] = Bit_Set(XR[xj].D[i], ui6)
```

The [X]VBITREV.{B/H/W/D} instruction inverts a specified bit in the byte/half-word/word/double-word elements of vector register vj/xj. The index of the bit to be inverted is derived from the corresponding byte/half-word/word/double-word elements at position vk/xk in the vector register.

VBITREV.B:

```
for i in range(16) :
    VR[vd].B[i] = Bit_Reverse(VR[vj].B[i], VR[vk].B[i] % 8)
```

VBITREV.H:

```
for i in range(8) :
    VR[vd].H[i] = Bit_Reverse(VR[vj].H[i], VR[vk].H[i] % 16)
```

VBITREV.W:

```
for i in range(4):
    VR[vd].W[i] = Bit_Reverse(VR[vj].W[i], VR[vk].W[i] % 32)
```

VBITREV.D:

```
for i in range(2):
    VR[vd].D[i] = Bit_Reverse(VR[vj].D[i], VR[vk].D[i] % 64)
```

XVBITREV.B:

```
for i in range(32) :
    XR[xd].B[i] = Bit_Reverse(XR[xj].B[i], XR[xk].B[i] % 8)
```

XVBITREV.H:

```
for i in range(16):
    XR[xd].H[i] = Bit_Reverse(XR[xj].H[i], XR[xk].H[i] % 16)
```

XVBITREV.W:

```
for i in range(8):
    XR[xd].W[i] = Bit_Reverse(XR[xj].W[i], XR[xk].W[i] % 32)
```

XVBITREV.D:

```
for i in range(4):
    XR[xd].D[i] = Bit_Reverse(XR[xj].D[i], XR[xk].D[i] % 64)
```

The [X]VBITREVI.{B/H/W/D} instruction inverts a specified bit in the byte/half-word/word/double-word elements of vector register vj/xj. The index of the bit to be inverted is derived from the immediate value ui3/ui4/ui5/ui6 in the instruction.

VBITREVI.B:

```
for i in range(16):
    VR[vd].B[i] = Bit_Reverse(VR[vj].B[i], ui3)
```

VBITREVI.H:

```
for i in range(8):
    VR[vd].H[i] = Bit_Reverse(VR[vj].H[i], ui4)
```

VBITREVI.W:

```
for i in range(4):
    VR[vd].W[i] = Bit_Reverse(VR[vj].W[i], ui5)
```

VBITREVI.D:

```
for i in range(2):
    VR[vd].D[i] = Bit_Reverse(VR[vj].D[i], ui6)
```

XVBITREVI.B:

```
for i in range(32) :
    XR[xd].B[i] = Bit_Reverse(XR[xj].B[i], ui3)
```

XVBITREVI.H:

```
for i in range(16):
    XR[xd].H[i] = Bit_Reverse(XR[xj].H[i], ui4)
```

XVBITREVI.W:

```
for i in range(8):
    XR[xd].W[i] = Bit_Reverse(XR[xj].W[i], ui5)
```

XVBITREVI.D:

```
for i in range(4):
    XR[xd].D[i] = Bit_Reverse(XR[xj].D[i], ui6)
```

1.2.3 Vector String Processing Instructions

1.2.3.1 [X]VFRSTP[I].{B/H}

Instruction Format:	vfrstp.b	vd, vj, vk	xvfrstp.b	xd, xj, xk
	vfrstp.h	vd, vj, vk	xvfrstp.h	xd, xj, xk
	vfrstpi.b	vd, vj, ui5	xvfrstpi.b	xd, xj, ui5
	vfrstpi.h	vd, vj, ui5	xvfrstpi.h	xd, xj, ui5

The [X]VFRSTP.{B/H} instruction iterates through each byte/half-word element in every 128 bits of vector register vj/xj from low to high. It writes the index value of the first negative element into a specified byte/half-word element in vector register vd/xd. If no negative value is found, the maximum valid index value plus one is written. Values in other positions of vector registers vd/xd remain unchanged. The position in vector registers vd/xd where the result is recorded is determined by the lowest byte/half-word element in every 128 bits of vector registers vk/xk.

VFRSTP.B:

```
idx = 0
for i in range(16):
    if VR[vj].B[i] < 0: break
    idx = idx + 1
m = VR[vk].B[0] % 16
VR[vd].B[m] = idx
```

VFRSTP.H:

```
idx = 0
for i in range(8):
    if VR[vj].H[i] < 0: break
    idx = idx + 1
m = VR[vk].H[0] % 8
VR[vd].H[m] = idx
```

XVFRSTP.B:

```
idx0 = 0
for i in range(16):
        if XR[xj].B[i] < 0: break
        idx0 = idx0 + 1
m0 = XR[xk].B[0] % 16
XR[xd].B[m0] = idx0 idx1 = 0
for i in range(16, 32, 1):
        if XR[xj].B[i] < 0: break
        idx1 = idx1 + 1
m1 = XR[xk].B[16] % 16
XR[xd].B[m1] = idx1
```

XVFRSTP.H:

```
idx0 = 0
for i in range(8):
        if XR[xj].H[i] < 0: break
        idx0 = idx0 + 1
m0 = XR[xk].H[0] % 8
XR[xd].H[m0] = idx0 idx1 = 0
for i in range(8, 16, 1):
        if XR[xj].H[i] < 0: break
        idx1 = idx1 + 1
m1 = XR[xk].H[8] % 8
XR[xd].H[m1] = idx1
```

The [X]VFRSTPI.{B/H} instruction iterates through each byte/half-word element in every 128 bits of vector register vj/xj from low to high. It writes the index value of the first negative element into a specified byte/half-word element in vector register vd/xd. If no negative value is found, the maximum valid index value plus one is written. Values in other positions of vector registers vd/xd remain unchanged. The position in vector registers vd/xd where the result is recorded is determined by the immediate value ui5 in the instruction code.

VFRSTPI.B:

```
idx = 0
for i in range(16):
        if VR[vj].B[i] < 0: break
        idx = idx + 1
m = ui5%16 VR[vd].B[m] = idx
```

VFRSTPI.H:

```
idx = 0
for i in range(8):
        if VR[vj].H[i] < 0: break
        idx = idx + 1
m = ui5 % 8 VR[vd].H[m] = idx
```

XVFRSTPI.B:

```
m = ui5 % 16 idx0 = 0
for i in range(16):
        if XR[xj].B[i] < 0: break
        idx0 = idx0 + 1
XR[xd].B[m] = idx0 idx1 = 0
for i in range(16, 32, 1):
        if XR[xj].B[i] < 0: break
        idx1 = idx1 + 1
XR[xd].B[m] = idx1
```

XVFRSTPI.H:

```
m = ui5 % 8 idx0 = 0
for i in range(8):
    if XR[xj].H[i] < 0: break
    idx0 = idx0 + 1
XR[xd].H[m] = idx0 idx1 = 0
for i in range(8, 16, 1):
    if XR[xj].H[i] < 0: break
    idx1 = idx1 + 1
XR[xd].H[m] = idx1
```

1.2.4 Vector Floating-Point Operations Instructions

1.2.4.1 [X]VF{ADD/SUB/MUL/DIV}.{S/D}

Instruction Format:	vfadd.s	vd, vj, vk	xvfadd.s	xd, xj, xk
	vfadd.d	vd, vj, vk	xvfadd.d	xd, xj, xk
	vfsub.s	vd, vj, vk	xvfsub.s	xd, xj, xk
	vfsub.d	vd, vj, vk	xvfsub.d	xd, xj, xk
	vfmul.s	vd, vj, vk	xvfmul.s	xd, xj, xk
	vfmul.d	vd, vj, vk	xvfmul.d	xd, xj, xk
	vfdiv.s	vd, vj, vk	xvfdiv.s	xd, xj, xk
	vfdiv.d	vd, vj, vk	xvfdiv.d	xd, xj, xk

The [X]VFADD.{S/D} instruction adds the corresponding single-precision/double-precision floating-point elements in vector register vk/xk to each single-precision/double-precision floating-point element in vector register vj/xj. The results are written to the corresponding single-precision/double-precision floating-point elements in vector register vd/xd. The floating-point addition operation follows the specification for the addition(x,y) operation in the IEEE 754-2008 standard.

VFADD.S:

```
for i in range(4):
    VR[vd].S[i] = FP32_addition(VR[vj].S[i], VR[vk].S[i])
```

VFADD.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_addition(VR[vj].D[i], VR[vk].D[i])
```

XVFADD.S:

```
for i in range(8):
    XR[xd].S[i] = FP32_addition(XR[xj].S[i], XR[xk].S[i])
```

XVFADD.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_addition(XR[xj].D[i], XR[xk].D[i])
```

The [X]VFSUB.{S/D} instruction subtracts the corresponding single-precision/double-precision floating-point element in vector register vk/xk from each single-precision/double-precision floating-point element in vector register vj/xj, writing the result to the corresponding single-precision/double-precision floating-point element in vector register vd/xd. The floating-point subtraction operation follows the specifications for the subtraction(x,y) operation in the IEEE 754-2008 standard.

VFSUB.S:

```
for i in range(4):
    VR[vd].S[i] = FP32_subtraction(VR[vj].S[i], VR[vk].S[i])
```

VFSUB.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_subtraction(VR[vj].D[i], VR[vk].D[i])
```

XVFSUB.S:

```
for i in range(8):
    XR[xd].S[i] = FP32_subtraction(XR[xj].S[i], XR[xk].S[i])
```

XVFSUB.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_subtraction(XR[xj].D[i], XR[xk].D[i])
```

The [X]VFMUL.{S/D} instruction multiplies each single-precision/double-precision floating-point element in vector register vj/xj by the corresponding single-precision/double-precision floating-point element in vector register vk/xk. The results are written to the corresponding single-precision/double-precision floating-point elements in vector register vd/xd. The floating-point multiplication operation follows the specification for the multiplication(x,y) operation in the IEEE 754-2008 standard.

VFMUL.S:

```
for i in range(4):
    VR[vd].S[i] = FP32_multiplication(VR[vj].S[i], VR[vk].S[i])
```

VFMUL.D:

```
for i in range(2) :
    VR[vd].D[i] = FP64_multiplication(VR[vj].D[i], VR[vk].D[i])
```

XVFMUL.S:

```
for i in range(8):
    XR[xd].S[i] = FP32_multiplication(XR[xj].S[i], XR[xk].S[i])
```

XVFMUL.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_multiplication(XR[xj].D[i], XR[xk].D[i])
```

The [X]VFDIV.{S/D} instruction divides each single-precision/double-precision floating-point element in vector register vj/xj by the corresponding single-precision/double-precision floating-point element in vector register vk/xk. The result is written to the corresponding single-precision/double-precision floating-point element in vector register vd/xd. The floating-point division operation follows the division(x,y) specification defined in the IEEE 754-2008 standard.

VFDIV.S:

```
for i in range(4):
    VR[vd].S[i] = FP32_division(VR[vj].S[i], VR[vk].S[i])
```

VFDIV.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_division(VR[vj].D[i], VR[vk].D[i])
```

XVFDIV.S:

```
for i in range(8):
    XR[xd].S[i] = FP32_division(XR[xj].S[i], XR[xk].S[i])
```

XVFDIV.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_division(XR[xj].D[i], XR[xk].D[i])
```

1.2.4.2 [X]VF{MADD/MSUB/NMADD/NMSUB}.{S/D}

Instruction Format:				
	vfmadd.s	vd, vj, vk, va	xvfmadd.s	xd, xj, xk, xa
	vfmadd.d	vd, vj, vk, va	xvfmadd.d	xd, xj, xk, xa
	vfmsub.s	vd, vj, vk, va	xvfmsub.s	xd, xj, xk, xa
	vfmsub.d	vd, vj, vk, va	xvfmsub.d	xd, xj, xk, xa
	vfnmadd.s	vd, vj, vk, va	xvfnmadd.s	xd, xj, xk, xa
	vfnmadd.d	vd, vj, vk, va	xvfnmadd.d	xd, xj, xk, xa
	vfnmsub.s	vd, vj, vk, va	xvfnmsub.s	xd, xj, xk, xa
	vfnmsub.d	vd, vj, vk, va	xvfnmsub.d	xd, xj, xk, xa

The [X]VFMADD.{S/D} instruction multiplies each single-precision/double-precision floating-point element in vector register vj/xj by the corresponding single-precision/double-precision floating-point element in vector register vk/xk, and adds the result to the corresponding single-precision/double-precision floating-point element in vector register va/xa. The result is written into the corresponding single-precision/double-precision floating-point element in vector register vd/xd. The floating-point fused multiply-add operation follows the specification for the fusedMultiplyAdd(x,y,z) operation in the IEEE 754-2008 standard.

VFMADD.S:

```
for i in range(4):
    VR[vd].S[i] = FP32_fusedMultiplyAdd(VR[vj].S[i], VR[vk].S[i], VR[va].S[i])
```

VFMADD.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_fusedMultiplyAdd(VR[vj].D[i], VR[vk].D[i], VR[va].D[i])
```

XVFMADD.S:

```
for i in range(8):
    XR[xd].S[i] = FP32_fusedMultiplyAdd(XR[xj].S[i], XR[xk].S[i], XR[xa].S[i])
```

XVFMADD.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_fusedMultiplyAdd(XR[xj].D[i], XR[xk].D[i], XR[xa].D[i])
```

The [X]VFMSUB.{S/D} instruction multiplies each single-precision/double-precision floating-point element in vector register vj/xj by the corresponding single-precision/double-precision floating-point element in vector register vk/xk, and subtracts the corresponding single-precision/double-precision floating-point element in vector register va/xa from the result. The result is written into the corresponding single-precision/double-precision floating-point element in vector register vd/xd. The floating-point fused multiply-add operation follows the specification for the fusedMultiplyAdd(x,y,z) operation in the IEEE 754-2008 standard.

VFMSUB.S:

```
for i in range(4):
    VR[vd].S[i] = FP32_fusedMultiplyAdd(VR[vj].S[i], VR[vk].S[i], -VR[va].S[i])
```

VFMSUB.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_fusedMultiplyAdd(VR[vj].D[i], VR[vk].D[i], -VR[va].D[i])
```

XVFMSUB.S:

```
for i in range(8):
    XR[xd].S[i] = FP32_fusedMultiplyAdd(XR[xj].S[i], XR[xk].S[i], -XR[xa].S[i])
```

XVFMSUB.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_fusedMultiplyAdd(XR[xj].D[i], XR[xk].D[i], -XR[xa].D[i])
```

The [X]VFNMADD.{S/D} instruction multiplies each single-precision/double-precision floating-point element in vector register vj/xj by the corresponding single-precision/double-precision floating-point element in vector register vk/xk. Then it adds the result to the corresponding single-precision/double-precision floating-point element in vector register va/xa, and negates the result. The final result is written into the corresponding single-precision/double-precision floating-point element in vector register vd/xd. The floating-point fused multiply-add operation follows the specification for the fusedMultiplyAdd(x,y,z) operation in the IEEE 754-2008 standard.

VFNMADD.S:

```
for i in range(4):
    VR[vd].S[i] = -FP32_fusedMultiplyAdd(VR[vj].S[i], VR[vk].S[i], VR[va].S[i])
```

VFNMADD.D:

```
for i in range(2):
    VR[vd].D[i] = -FP64_fusedMultiplyAdd(VR[vj].D[i], VR[vk].D[i], VR[va].D[i])
```

XVFNMADD.S:

```
for i in range(8):
    XR[xd].S[i] = -FP32_fusedMultiplyAdd(XR[xj].S[i], XR[xk].S[i], XR[xa].S[i])
```

XVFNMADD.D:

```
for i in range(4):
    XR[xd].D[i] = -FP64_fusedMultiplyAdd(XR[xj].D[i], XR[xk].D[i], XR[xa].D[i])
```

The [X]VFNMSUB.{S/D} instruction multiplies each single-precision/double-precision floating-point element in vector register vj/xj by the corresponding single-precision/double-precision floating-point element in vector register vk/xk. Then it subtracts the corresponding single-precision/double-precision floating-point element in vector register va/xa from the result, and negates the result. The final result is written into the corresponding single-precision/double-precision floating-point element in vector register vd/xd. The floating-point fused multiply-add operation follows the specification for the fusedMultiplyAdd(x,y,z) operation in the IEEE 754-2008 standard.

VFNMSUB.S:

```
for i in range(4):
    VR[vd].S[i] = -FP32_fusedMultiplyAdd(VR[vj].S[i], VR[vk].S[i], -VR[va].S[i])
```

VFNMSUB.D:

```
for i in range(2):
    VR[vd].D[i] = -FP64_fusedMultiplyAdd(VR[vj].D[i], VR[vk].D[i], -VR[va].D[i])
```

XVFNMSUB.S:

```
for i in range(8):
    XR[xd].S[i] = -FP32_fusedMultiplyAdd(XR[xj].S[i], XR[xk].S[i], -XR[xa].S[i])
```

XVFNMSUB.D:

```
for i in range(4):
    XR[xd].D[i] = -FP64_fusedMultiplyAdd(XR[xj].D[i], XR[xk].D[i], -XR[xa].D[i])
```

1.2.4.3 [X]VF{MAX/MIN}.{S/D}

Instruction Format:				
vfmax.s	vd, vj, vk	xvfmax.d	vd, vj, vk
vfmax.d	xd, xj, xk	xvfmax.s	xd, xj, xk
vfmin.s	vd, vj, vk	xvfmin.s	xd, xj, xk
vfmin.d	vd, vj, vk	xvfmin.d	xd, xj, xk

The [X]VFMAX.{S/D} instruction compares the single-precision/double-precision floating-point elements in vector register vj/xj with the corresponding single-precision/double-precision floating-point elements in vector register vk/xk. It writes the larger value to the corresponding single-precision/double-precision floating-point element in vector register vd/xd. This comparison follows the maxNum(x,y) operation specification in the IEEE 754-2008 standard.

VFMAX.S:

```
for i in range(4):
    VR[vd].S[i] = FP32_maxNum(VR[vj].S[i], VR[vk].S[i])
```

VFMAX.D:

```
for i in range(2) :
    VR[vd].D[i] = FP64_maxNum(VR[vj].D[i], VR[vk].D[i])
```

XVFMAX.S:

```
for i in range(8) :
    XR[xd].S[i] = FP32_maxNum(XR[xj].S[i], XR[xk].S[i])
```

XVFMAX.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_maxNum(XR[xj].D[i], XR[xk].D[i])
```

The [X]VFMIN.{S/D} instruction compares the single-precision/double-precision floating-point elements in vector register vj/xj with the corresponding single-precision/double-precision floating-point elements in vector register vk/xk. It writes the smaller value to the corresponding single-precision/double-precision floating-point element in vector register vd/xd. This comparison follows the minNum(x,y) operation specification in the IEEE 754-2008 standard.

VFMIN.S:

```
for i in range(4):
    VR[vd].S[i] = FP32_minNum(VR[vj].S[i], VR[vk].S[i])
```

VFMIN.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_minNum(VR[vj].D[i], VR[vk].D[i])
```

XVFMIN.S:

```
for i in range(8):
    XR[xd].S[i] = FP32_minNum(XR[xj].S[i], XR[xk].S[i])
```

XVFMIN.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_minNum(XR[xj].D[i], XR[xk].D[i])
```

1.2.4.4 [X]VF{MAXA/MINA}.{S/D}

Instruction Format:				
	vfmaxa.s	vd, vj, vk	xfmaxa.s	xd, xj, xk
	vfmaxa.d	vd, vj, vk	xfmaxa.d	xd, xj, xk
	vfmina.s	vd, vj, vk	xvfmina.s	xd, xj, xk
	vfmina.d	vd, vj, vk	xvfmina.d	xd, xj, xk

The [X]VFMAXA.{S/D} instruction performs an absolute value comparison between the single-precision/double-precision floating-point elements in vector register vj/xj and the corresponding single-precision/double-precision floating-point elements in vector register vk/xk. It writes the element with a larger absolute value to the corresponding single-precision/double-precision floating-point element in vector register vd/xd. This comparison follows the maxNumMag(x,y) operation specification in the IEEE 754-2008 standard.

VFMAXA.S:

```
for i in range(4):
    VR[vd].S[i] = FP32_maxNumMag(VR[vj].S[i], VR[vk].S[i])
```

VFMAXA.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_maxNumMag(VR[vj].D[i], VR[vk].D[i])
```

XVFMAXA.S:

```
for i in range(8):
    XR[xd].S[i] = FP32_maxNumMag(XR[xj].S[i], XR[xk].S[i])
```

XVFMAXA.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_maxNumMag(XR[xj].D[i], XR[xk].D[i])
```

The [X]VFMINA.{S/D} instruction performs an absolute value comparison between the single-precision/double-precision floating-point elements in vector register vj/xj and the corresponding single-precision/double-precision floating-point elements in vector register vk/xk. It writes the element with a smaller absolute value to the corresponding single-precision/double-precision floating-point element in vector register vd/xd. This comparison follows the minNumMag(x,y) operation specification in the IEEE 754-2008 standard.

VFMINA.S:

```
for i in range(4):
    VR[vd].S[i] = FP32_minNumMag(VR[vj].S[i], VR[vk].S[i])
```

VFMINA.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_minNumMag(VR[vj].D[i], VR[vk].D[i])
```

XVFMINA.S:

```
for i in range(8):
    XR[xd].S[i] = FP32_minNumMag(XR[xj].S[i], XR[xk].S[i])
```

XVFMINA.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_minNumMag(XR[xj].D[i], XR[xk].D[i])
```

1.2.4.5 [X]VFLOGB.{S/D}

Instruction Format:

vflogb.s	vd, vj		xvflogb.s	xd, xj
vflogb.d	vd, vj		xvflogb.d	xd, xj

The [X]VFLOGB.{S/D} instruction computes the base-2 logarithm of the single-precision/double-precision floating-point element in vector register vj/xj. The result is written into the corresponding single-precision/double-precision floating-point element in vector register vd/xd. This floating-point operation follows the logB(x) specification in the IEEE 754-2008 standard.

VFLOGB.S:

```
for i in range(4):
    VR[vd].S[i] = FP32_logB(VR[vj].S[i])
```

VFLOGB.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_logB(VR[vj].D[i])
```

XVFLOGB.S:

```
for i in range(8) :
    XR[xd].S[i] = FP32_logB(XR[xj].S[i])
```

XVFLOGB.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_logB(XR[xj].D[i])
```

1.2.4.6 [X]VFCLASS.{S/D}

Instruction Format:

vfclass.s	vd, vj		xvfclass.s	xd, xj
vfclass.d	vd, vj		xvfclass.d	xd, xj

The [X]VFCLASS.{S/D} instruction performs type checks on each single-precision/double-precision floating-point element in vector registers vj/xj. When the data satisfies the condition corresponding to a particular bit, the corresponding bit in the result information vector is set to 1. The results are written into the corresponding positions of vector registers vd/xd, padded with zeros to the right.

The element verification process follows the class(x) operation specifications in the IEEE 754-2008 standard. The result comprises 10 bits of information, with each bit's meaning detailed in the table below:

Bit0	Bit1	Bit2	Bit3	Bit4	Bit5	Bit6	Bit7	Bit8	Bit9
SNaN	QNaN	negative value				positive value			
		∞	normal	subnormal	0	∞	normal	subnormal	0

VFCLASS.S:

```
for i in range(4):
    VR[vd].S[i] = FP32_class(VR[vj].S[i])
```

VFCLASS.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_class(VR[vj].D[i])
```

XVFCLASS.S:

```
for i in range(8):
    XR[xd].S[i] = FP32_class(XR[xj].S[i])
```

XVFCLASS.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_class(XR[xj].D[i])
```

1.2.4.7 [X]VF{SQRT/RECIP/RSQRT}.{S/D}

Instruction Format:				
	vfsqrt.s	vd, vj	xvfsqrt.s	xd, xj
	vfsqrt.d	vd, vj	xvfsqrt.d	xd, xj
	vfrecip.s	vd, vj	xvfrecip.s	xd, xj
	vfrecip.d	vd, vj	xvfrecip.d	xd, xj
	vfrsqrt.s	vd, vj	xvfrsqrt.s	xd, xj
	vfrsqrt.d	vd, vj	xvfrsqrt.d	xd, xj

The [X]VFSQRT.{S/D} instruction performs a square root operation on each single-precision/double-precision floating-point element in vector registers vj/xj. The results are written to the corresponding single-precision/double-precision floating-point elements in vector registers vd/xd. The floating-point square root operation follows the specifications for the squareRoot(x) operation in the IEEE 754-2008 standard.

VFSQRT.S:

```
for i in range(4) :
    VR[vd].S[i] = FP32_squareRoot(VR[vj].S[i])
```

VFSQRT.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_squareRoot(VR[vj].D[i])
```

XVFSQRT.S:

```
for i in range(8):
    XR[xd].S[i] = FP32_squareRoot(XR[xj].S[i])
```

XVFSQRT.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_squareRoot(XR[xj].D[i])
```

The [X]VFRECIP.{S/D} instruction computes the reciprocal of each single-precision/double-precision floating-point element in vector register vj/xj. The results are written to the corresponding single-precision/double-precision floating-point elements in vector registers vd/xd. This operation is equivalent to performing division(1.0,x) according to the IEEE 754-2008 standard specification.

VFRECIP.S:

```
for i in range(4) :
    VR[vd].S[i] = FP32_division(1.0, VR[vj].S[i])
```

VFRECIP.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_division(1.0, VR[vj].D[i])
```

XVFRECIP.S:

```
for i in range(8) :
    XR[xd].S[i] = FP32_division(1.0, XR[xj].S[i])
```

XVFRECIP.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_division(1.0, XR[xj].D[i])
```

The [X]VFRSQRT.{S/D} instruction performs square root followed by reciprocal operations on each single-precision/double-precision floating-point element in vector registers vj/xj. The results are written to the corresponding single-precision/double-precision floating-point elements in vector registers vd/xd. This operation is equivalent to performing division(1.0, squareRoot(x)) according to the IEEE 754-2008 standard specification.

VFRSQRT.S:

```
for i in range(4):
    VR[vd].S[i] = FP32_division(1.0, FP_squareRoot(VR[vj].S[i]))
```

VFRSQRT.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_division(1.0, FP_squareRoot(VR[vj].D[i]))
```

XVFRSQRT.S:

```
for i in range(8) :
    XR[xd].S[i] = FP32_division(1.0, FP_squareRoot(XR[xj].S[i]))
```

XVFRSQRT.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_division(1.0, FP_squareRoot(XR[xj].D[i]))
```

1.2.4.8 [X]VF{RECIPE/RSQRTE}.{S/D}

Instruction Format:					
	vfrecipe.s	vd, vj		xvfrecipe.s	xd, xj
	vfrecipe.d	vd, vj		xvfrecipe.d	xd, xj
	vfrsqrte.s	vd, vj		xvfrsqrte.s	xd, xj
	vfrsqrte.d	vd, vj		xvfrsqrte.d	xd, xj

LoongArch V1.1 Instructions

The [X]VFRECIPE.{S/D} instruction calculates the single-precision/double-precision approximation obtained by dividing the corresponding element in vector register vj/xj by 1.0, writing the result to floating-point register fd. The relative error of the approximation is less than 2^{-14}.

When the input value is 2^N, the output value is 2^{-N}. The results for inputs of QNaN, SNaN, ±∞, ±0, the conditions that generate floating-point exceptions, and the default results when a floating-point exception is generated but not triggered are identical to those of the [X]VFRECIPE.{S/D} instruction.

VFRECIPE.S:

```
for i in range(4):
    VR[vd].S[i] = FP32_reciprocal_estimate(VR[vj].S[i])
```

VFRECIPE.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_reciprocal_estimate(VR[vj].D[i])
```

XVFRECIPE.S:

```
for i in range(8) :
    XR[xd].S[i] = FP32_reciprocal_estimate(XR[xj].S[i])
```

XVFRECIPE.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_reciprocal_estimate(XR[xj].D[i])
```

The [X]VFRSQRTE.{S/D} instruction takes each single-precision/double-precision floating-point element in vector register vj/xj, computes the square root of that element, divides the single-precision/double-precision floating-point result by 1.0, and writes the result to floating-point register fd. The relative error of the resulting approximation is less than 2^{-14}.

When the input value is 2^{2N}, the output value is 2^{-N}. The results for inputs of QNaN, SNaN, ±∞, ±0, the conditions that generate floating-point exceptions, and the default results when a floating-point exception is generated but not triggered are identical to those of the [X]VFRSQRT.{S/D} instruction.

VFRSQRTE.S:

```
for i in range(4) :
    VR[vd].S[i] = FP32_reciprocal_squareroot_estimate(VR[vj].S[i])
```

VFRSQRTE.D:

```
for i in range(2) :
    VR[vd].D[i] = FP64_reciprocal_squareroot_estimate(VR[vj].D[i])
```

XVFRSQRTE.S:

```
for i in range(8):
    XR[xd].S[i] = FP32_reciprocal_squareroot_estimate(XR[xj].S[i])
```

XVFRSQRTE.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_reciprocal_squareroot_estimate(XR[xj].D[i])
```

1.2.5 Vector Floating-Point Conversion Instructions

1.2.5.1 [X]VFCVT{L/H}.{S.H/D.S}, [X]VFCVT.{H.S/S.D}

Instruction Format:				
	vfcvtl.s.h	vd, vj	xvfcvtl.s.h	xd, xj
	vfcvth.s.h	vd, vj	xvfcvth.s.h	xd, xj
	vfcvtl.d.s	vd, vj	xvfcvtl.d.s	xd, xj
	vfcvth.d.s	vd, vj	xvfcvth.d.s	xd, xj
	vfcvt.h.s	vd, vj, vk	xvfcvt.h.s	xd, xj, xk
	vfcvt.s.d	vd, vj, vk	xvfcvt.s.d	xd, xj, xk

The [X]VFCVTL.{S.H/D.S} instruction converts each half-precision/single-precision floating-point element in the lower half of each 128 bits in vector registers vj/xj to single-precision/double-precision floating-point values. The results are sequentially written to vector registers vd/xd. The rounding mode for the results is determined by the RM field in the FCSR. The floating-point precision conversion operation follows the specifications of the convertFormat(x) operation in the IEEE 754-2008 standard.

VFCVTL.S.H:

```
for i in range(4):
    VR[vd].S[i] = FP32_convertFormat(VR[vj].H[i], FP16)
```

VFCVTL.D.S:

```
for i in range(2):
    VR[vd].D[i] = FP64_convertFormat(VR[vj].S[i], FP32)
```

XVFCVTL.S.H:

```
for i in range(4):
    XR[xd].S[i+0] = FP32_convertFormat(XR[xj].H[i], FP16)
    XR[xd].S[i+4] = FP32_convertFormat(XR[xj].H[i+8], FP16)
```

XVFCVTL.D.S:

```
for i in range(2):
    XR[xd].D[i+0] = FP64_convertFormat(XR[xj].S[i], FP32)
    XR[xd].D[i+2] = FP64_convertFormat(XR[xj].S[i+4], FP32)
```

The [X]VFCVTH.{S.H/D.S} instruction converts each half-precision/single-precision floating-point element in the upper half of each 128 bits in vector register vj/xj to single-precision/double-precision floating-point values. The results are sequentially written to the vector register vd/xd. The rounding mode for the results is determined by the RM field in the FCSR. The floating-point precision conversion operation follows the specifications for the convertFormat(x) operation in the IEEE 754-2008 standard.

VFCVTH.S.H:

```
for i in range(4):
    VR[vd].S[i] = FP32_convertFormat(VR[vj].H[i+4], FP16)
```

VFCVTH.D.S:

```
for i in range(2):
VR[vd].D[i] = FP64_convertFormat(VR[vj].S[i+2], FP32)
```

XVFCVTH.S.H:

```
for i in range(4):
    XR[xd].S[i+0] = FP32_convertFormat(XR[xj].H[i+4], FP16)
    XR[xd].S[i+4] = FP32_convertFormat(XR[xj].H[i+12], FP16)
```

XVFCVTH.D.S:

```
for i in range(2):
    XR[xd].D[i+0] = FP64_convertFormat(XR[xj].S[i+2], FP32)
    XR[xd].D[i+2] = FP64_convertFormat(XR[xj].S[i+6], FP32)
```

The [X]VFCVT.{H.S/S.D} instruction converts each single-precision/double-precision floating-point element in every 128 bits of vector register vj/xj into a half-precision/single-precision floating-point number, writing the results sequentially into the upper half of each 128 bits in vector registers vd/xd. It converts each single-precision/double-precision floating-point element in every 128 bits of vector registers vk/xk to half-precision/single-precision floating-point numbers,

writing the results sequentially into the lower half of each 128 bits in vector registers vd/xd. The rounding mode for the result is determined by the RM field in the FCSR. The floating-point precision conversion operation follows the specifications for the convertFormat(x) operation in the IEEE 754-2008 standard.

VFCVT.H.S:

```
for i in range(4) :
    VR[vd].H[i+4] = FP16_convertFormat(VR[vj].S[i], FP32)
    VR[vd].H[i+0] = FP16_convertFormat(VR[vk].S[i], FP32)
```

VFCVT.S.D:

```
for i in range(2) :
    VR[vd].S[i+2] = FP32_convertFormat(VR[vj].D[i], FP64)
    VR[vd].S[i+0] = FP32_convertFormat(VR[vk].D[i], FP64)
```

XVFCVT.H.S:

```
for i in range(4):
    XR[xd].H[i+4] = FP16_convertFormat(XR[xj].S[i], FP32)
    XR[xd].H[i+0] = FP16_convertFormat(XR[xk].S[i], FP32)
    XR[xd].H[i+12] = FP16_convertFormat(XR[xj].S[i+4], FP32)
    XR[xd].H[i+8] = FP16_convertFormat(XR[xk].S[i+4], FP32)
```

XVFCVT.S.D:

```
for i in range(2) :
    XR[xd].S[i+2] = FP32_convertFormat(XR[xj].D[i], FP64)
    XR[xd].S[i+0] = FP32_convertFormat(XR[xk].D[i], FP64)
    XR[xd].S[i+6] = FP32_convertFormat(XR[xj].D[i+2], FP64)
    XR[xd].S[i+4] = FP32_convertFormat(XR[xk].D[i+2], FP64)
```

1.2.5.2 [X]VFRINT[{RNE/RZ/RP/RM}].{S/D}

Instruction Format:				
	vfrintrne.s	vd, vj	xvfrintrne.s	xd, xj
	vfrintrne.d	vd, vj	xvfrintrne.d	xd, xj
	vfrintrz.s	vd, vj	xvfrintrz.s	xd, xj
	vfrintrz.d	vd, vj	xvfrintrz.d	xd, xj
	vfrintrp.s	vd, vj	xvfrintrp.s	xd, xj
	vfrintrp.d	vd, vj	xvfrintrp.d	xd, xj
	vfrintrm.s	vd, vj	xvfrintrm.s	xd, xj
	vfrintrm.d	vd, vj	xvfrintrm.d	xd, xj
	vfrint.s	vd, vj	xvfrint.s	xd, xj
	vfrint.d	vd, vj	xvfrint.d	xd, xj

The [X]VFRINTRNE.{S/D} instruction converts each single-precision/double-precision floating-point element in vector register vj/xj to a single-precision/double-precision floating-point number with integer value. The result is written to the corresponding single-precision/double-precision floating-point element in vector register vd/xd. The result uses the "rounding to nearest (even-middle rounding)" method.

VFRINTRNE.S:

```
for i in range(4):
    VR[vd].S[i] = FP32_roundToInteger(VR[vj].S[i], FCSR.Enables.I, 0)
```

VFRINTRNE.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_roundToInteger(VR[vj].D[i], FCSR.Enables.I, 0)
```

XVFRINTRNE.S:

```
for i in range(8):
    XR[xd].S[i] = FP32_roundToInteger(XR[xj].S[i], FCSR.Enables.I, 0)
```

XVFRINTRNE.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_roundToInteger(XR[xj].D[i], FCSR.Enables.I, 0)
```

The [X]VFRINTRZ.{S/D} instruction converts each single-precision/double-precision floating-point element in vector register vj/xj into a single-precision/double-precision floating-point number with integer values, writing the results to the corresponding single-precision/double-precision floating-point elements in vector registers vd/xd. The results are rounded toward zero.

VFRINTRZ.S:

```
for i in range(4):
    VR[vd].S[i] = FP32_roundToInteger(VR[vj].S[i], FCSR.Enables.I, 1)
```

VFRINTRZ.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_roundToInteger(VR[vj].D[i], FCSR.Enables.I, 1)
```

XVFRINTRZ.S:

```
for i in range(8) :
    XR[xd].S[i] = FP32_roundToInteger(XR[xj].S[i], FCSR.Enables.I, 1)
```

XVFRINTRZ.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_roundToInteger(XR[xj].D[i], FCSR.Enables.I, 1)
```

The [X]VFRINTRP.{S/D} instruction converts each single-precision/double-precision floating-point element in vector register vj/xj into a single-precision/double-precision floating-point number with integer values, writing the results to the corresponding single-precision/double-precision floating-point elements in vector registers vd/xd. The results are rounded toward positive infinity.

VFRINTRP.S:

```
for i in range(4) :
    VR[vd].S[i] = FP32_roundToInteger(VR[vj].S[i], FCSR.Enables.I, 2)
```

VFRINTRP.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_roundToInteger(VR[vj].D[i], FCSR.Enables.I, 2)
```

XVFRINTRP.S:

```
for i in range(8):
    XR[xd].S[i] = FP32_roundToInteger(XR[xj].S[i], FCSR.Enables.I, 2)
```

XVFRINTRP.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_roundToInteger(XR[xj].D[i], FCSR.Enables.I, 2)
```

The [X]VFRINTRM.{S/D} instruction converts each single-precision/double-precision floating-point element in vector register vj/xj to a single-precision/double-precision floating-point number with an integer value. The result is written to the corresponding single-precision/double-precision floating-point element in vector register vd/xd. The result is rounded toward negative infinity.

VFRINTRM.S:

```
for i in range(4) :
    VR[vd].S[i] = FP32_roundToInteger(VR[vj].S[i], FCSR.Enables.I, 3)
```

VFRINTRM.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_roundToInteger(VR[vj].D[i], FCSR.Enables.I, 3)
```

XVFRINTRM.S:

```
for i in range(8):
    XR[xd].S[i] = FP32_roundToInteger(XR[xj].S[i], FCSR.Enables.I, 3)
```

XVFRINTRM.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_roundToInteger(XR[xj].D[i], FCSR.Enables.I, 3)
```

The [X]VFRINT.{S/D} instruction converts each single-precision/double-precision floating-point element in vector register vj/xj to a single-

precision/double-precision floating-point number with an integer value. The result is written to the corresponding single-precision/double-precision floating-point element in vector register vd/xd. The rounding mode for the result is determined by the RM field in FCSR.

VFRINT.S:

```
for i in range(4) :
    VR[vd].S[i] = FP32_roundToInteger(VR[vj].S[i], FCSR.Enables.I, FCSR.RM)
```

VFRINT.D:

```
for i in range(2):
    VR[vd].D[i] = FP64_roundToInteger(VR[vj].D[i], FCSR.Enables.I, FCSR.RM)
```

XVFRINT.S:

```
for i in range(8):
    XR[xd].S[i] = FP32_roundToInteger(XR[xj].S[i], FCSR.Enables.I, FCSR.RM)
```

XVFRINT.D:

```
for i in range(4):
    XR[xd].D[i] = FP64_roundToInteger(XR[xj].D[i], FCSR.Enables.I, FCSR.RM)
```

1.2.5.3 [X]VFTINT[{RNE/RZ/RP/RM}].{W.S/L.D}, [X]VFTINT[RZ].{WU.S/LU.D}, [X]VFTINT[{RNE/RZ/RP/RM}].W.D, [X]VFTINT[{RNE/RZ/RP/RM}]{L/H}.L.S

Instruction Format:				
	vftintrne.w.s	vd, vj	xvftintrne.w.s	xd, xj
	vftintrne.l.d	vd, vj	xvftintrne.l.d	xd, xj
	vftintrz.w.s	vd, vj	xvftintrz.w.s	xd, xj
	vftintrz.l.d	vd, vj	xvftintrz.l.d	xd, xj
	vftintrp.w.s	vd, vj	xvftintrp.w.s	xd, xj
	vftintrp.l.d	vd, vj	xvftintrp.l.d	xd, xj
	vftintrm.w.s	vd, vj	xvftintrm.w.s	xd, xj
	vftintrm.l.d	vd, vj	xvftintrm.l.d	xd, xj
	vftint.w.s	vd, vj	xvftint.w.s	xd, xj
	vftint.l.d	vd, vj	xvftint.l.d	xd, xj
	vftintrz.wu.s	vd, vj	xvftintrz.wu.s	xd, xj
	vftintrz.lu.d	vd, vj	xvftintrz.lu.d	xd, xj
	vftint.wu.s	vd, vj	xvftint.wu.s	xd, xj
	vftint.lu.d	vd, vj	xvftint.lu.d	xd, xj
	vftintrne.w.d	vd, vj, vk	xvftintrne.w.d	xd, xj, xk
	vftintrz.w.d	vd, vj, vk	xvftintrz.w.d	xd, xj, xk
	vftintrp.w.d	vd, vj, vk	xvftintrp.w.d	xd, xj, xk
	vftintrm.w.d	vd, vj, vk	xvftintrm.w.d	xd, xj, xk
	vftint.w.d	vd, vj, vk	xvftint.w.d	xd, xj, xk
	vftintrnel.l.s	vd, vj	xvftintrnel.l.s	xd, xj
	vftintrneh.l.s	vd, vj	xvftintrneh.l.s	xd, xj
	vftintrzl.l.s	vd, vj	xvftintrzl.l.s	xd, xj
	vftintrzh.l.s	vd, vj	xvftintrzh.l.s	xd, xj
	vftintrpl.l.s	vd, vj	xvftintrpl.l.s	xd, xj
	vftintrph.l.s	vd, vj	xvftintrph.l.s	xd, xj
	vftintrml.l.s	vd, vj	xvftintrml.l.s	xd, xj
	vftintrmh.l. s	vd, vj	xvftintrmh.l.s	xd, xj
	vftintl.l.s	vd, vj	xvftintl.l.s	xd, xj
	vftinth.l.s	vd, vj	xvftinth.l.s	xd, xj

The [X]VFTINTRNE.{W.S/ L.D} instruction converts each single-precision/double-precision floating-point element in vector registers vj/xj into signed integer/double-words. The results are written to the corresponding positions in vector registers vd/xd. The results are rounded using the "round-to-nearest (even-if-odd)" method.

VFTINTRNE.W.S:

```
for i in range(4):
    VR[vd].W[i] = Sint32_convertToInteger(VR[vj].S[i], FCSR.Enables.I, 0)
```

VFTINTRNE.L.D:

```
for i in range(2):
    VR[vd].L[i] = Sint64_convertToInteger(VR[vj].D[i], FCSR.Enables.I, 0)
```

XVFTINTRNE.W.S:

```
for i in range(8):
    XR[xd].W[i] = Sint32_convertToInteger(XR[xj].S[i], FCSR.Enables.I, 0)
```

XVFTINTRNE.L.D:

```
for i in range(4):
    XR[xd].L[i] = Sint64_convertToInteger(XR[xj].D[i], FCSR.Enables.I, 0)
```

The [X]VFTINTRZ.{W.S/L.D} instruction converts each single-precision/double-precision floating-point element in vector register vj/xj to a signed integer/double-word. The results are written to the corresponding elements in vector register vd/xd. The results are rounded toward zero.

VFTINTRZ.W.S:

```
for i in range(4):
    VR[vd].W[i] = Sint32_convertToInteger(VR[vj].S[i], FCSR.Enables.I, 1)
```

VFTINTRZ.L.D:

```
for i in range(2):
    VR[vd].L[i] = Sint64_convertToInteger(VR[vj].D[i], FCSR.Enables.I, 1)
```

XVFTINTRZ.W.S:

```
for i in range(8):
    XR[xd].W[i] = Sint32_convertToInteger(XR[xj].S[i], FCSR.Enables.I, 1)
```

XVFTINTRZ.L.D:

```
for i in range(4):
    XR[xd].L[i] = Sint64_convertToInteger(XR[xj].D[i], FCSR.Enables.I, 1)
```

The [X]VFTINTRP.{W.S/L.D} instruction converts each single-precision/double-precision floating-point element in vector register vj/xj to a signed integer/double-word, writing the result to the corresponding element positions in vector register vd/xd. The results are rounded toward positive infinity.

VFTINTRP.W.S:

```
for i in range(4):
    VR[vd].W[i] = Sint32_convertToInteger(VR[vj].S[i], FCSR.Enables.I, 2)
```

VFTINTRP.L.D:

```
for i in range(2):
    VR[vd].L[i] = Sint64_convertToInteger(VR[vj].D[i], FCSR.Enables.I, 2)
```

XVFTINTRP.W.S:

```
for i in range(8):
    XR[xd].W[i] = Sint32_convertToInteger(XR[xj].S[i], FCSR.Enables.I, 2)
```

XVFTINTRP.L.D:

```
for i in range(4):
    XR[xd].L[i] = Sint64_convertToInteger(XR[xj].D[i], FCSR.Enables.I, 2)
```

The [X]VFTINTRM.{W.S/L.D} instruction converts each single-precision/double-precision floating-point element in vector register vj/xj to a signed integer/double-word. The results are written to the corresponding elements in vector register vd/xd. The results are rounded toward negative infinity.

VFTINTRM.W.S:

```
for i in range(4):
    VR[vd].W[i] = Sint32_convertToInteger(VR[vj].S[i], FCSR.Enables.I, 3)
```

VFTINTRM.L.D:

```
for i in range(2):
    VR[vd].L[i] = Sint64_convertToInteger(VR[vj].D[i], FCSR.Enables.I, 3)
```

XVFTINTRM.W.S:

```
for i in range(8):
    XR[xd].W[i] = Sint32_convertToInteger(XR[xj].S[i], FCSR.Enables.I, 3)
```

XVFTINTRM.L.D:

```
for i in range(4):
    XR[xd].L[i] = Sint64_convertToInteger(XR[xj].D[i], FCSR.Enables.I, 3)
```

The [X]VFTINT.{W.S/L.D} instruction converts each single-precision/double-precision floating-point element in vector register vj/xj to a signed integer/double-word, writing the result to the corresponding element in vector register vd/xd. The rounding mode for the result is determined by the RM field in FCSR.

VFTINT.W.S:

```
for i in range(4) :
    VR[vd].W[i] = Sint32_convertToInteger(VR[vj].S[i], FCSR.Enables.I, FCSR.RM)
```

VFTINT.L.D:

```
for i in range(2):
    VR[vd].L[i] = Sint64_convertToInteger(VR[vj].D[i], FCSR.Enables.I, FCSR.RM)
```

XVFTINT.W.S:

```
for i in range(8):
    XR[xd].W[i] = Sint32_convertToInteger(XR[xj].S[i], FCSR.Enables.I, FCSR.RM)
```

XVFTINT.L.D:

```
for i in range(4):
    XR[xd].L[i] = Sint64_convertToInteger(XR[xj].D[i], FCSR.Enables.I, FCSR.RM)
```

The [X]VFTINTRZ.{WU.S/LU.D} instruction converts each single-precision/double-precision floating-point element in vector register vj/xj to an unsigned integer/double-word. The results are written to the corresponding elements in vector register vd/xd. The results are rounded toward zero.

VFTINTRZ.WU.S:

```
for i in range(4) :
    VR[vd].W[i] = Uint32_convertToInteger(VR[vj].S[i], FCSR.Enables.I, 1)
```

VFTINTRZ.LU.D:

```
for i in range(2):
    VR[vd].L[i] = Uint64_convertToInteger(VR[vj].D[i], FCSR.Enables.I, 1)
```

XVFTINTRZ.WU.S:

```
for i in range(8):
    XR[xd].W[i] = Uint32_convertToInteger(XR[xj].S[i], FCSR.Enables.I, 1)
```

XVFTINTRZ.LU.D:

```
for i in range(4):
    XR[xd].L[i] = Uint64_convertToInteger(XR[xj].D[i], FCSR.Enables.I, 1)
```

The [X]VFTINT.{WU.S/LU.D} instruction converts each single-precision/double-precision floating-point element in vector register vj/xj to an unsigned integer/double-word. The results are written to the corresponding elements in vector register vd/xd. The rounding mode for the results is determined by the RM field in FCSR.

VFTINT.WU.S:

```
for i in range(4):
    VR[vj].W[i] = Uint32_convertToInteger(VR[vj].S[i], FCSR.Enables.I, FCSR.RM)
```

VFTINT.LU.D:

```
for i in range(2) :
    VR[vd].L[i] = Uint64_convertToInteger(VR[vj].D[i], FCSR.Enables.I, FCSR.RM)
```

XVFTINT.WU.S:

```
for i in range(8):
    XR[xd].W[i] = Uint32_convertToInteger(XR[xj].S[i], FCSR.Enables.I, FCSR.RM)
```

XVFTINT.LU.D:

```
for i in range(4) :
    XR[xd].L[i] = Uint64_convertToInteger(XR[xj].D[i], FCSR.Enables.I, FCSR.RM)
```

The [X]VFTINTRNE.W.D instruction converts each double-precision floating-point element in every 128 bits of vector registers vj/xj into a signed integer. The results are sequentially written into the upper half of each 128-bit in vector register vd/xd. It converts each double-precision floating-point element in vector register vk/xk to a signed integer, writing the results sequentially to the lower half of each 128 bits in vector register vd/xd. The results are rounded using "round-to-nearest (mid-rounding)" precision.

VFTINTRNE.W.D:

```
for i in range(2):
    VR[vd].W[i+2] = Sint32_convertToInteger(VR[vj].D[i], FCSR.Enables.I, 0)
    VR[vd].W[i+0] = Sint32_convertToInteger(VR[vk].D[i], FCSR.Enables.I, 0)
```

XVFTINTRNE.W.D:

```
for i in range(2):
    XR[xd].W[i+2] = Sint32_convertToInteger(XR[xj].D[i], FCSR.Enables.I, 0)
    XR[xd].W[i+0] = Sint32_convertToInteger(XR[xk].D[i], FCSR.Enables.I, 0)
    XR[xd].W[i+6] = Sint32_convertToInteger(XR[xj].D[i+2], FCSR.Enables.I,0)
    XR[xd].W[i+4] = Sint32_convertToInteger(XR[xk].D[i+2], FCSR.Enables.I,0)
```

The [X]VFTINTRZ.W.D instruction converts each double-precision floating-point element in every 128 bits of vector registers vj/xj into a signed integer. The results are sequentially written into the upper half of each 128-bit in vector register vd/xd. It converts each double-precision floating-point element in vector register vk/xk to a signed integer, writing the results sequentially to the lower half of each 128 bits in vector register vd/xd. The results are rounded toward zero.

VFTINTRZ.W.D:

```
for i in range(2):
    VR[vd].W[i+2] = Sint32_convertToInteger(VR[vj].D[i], FCSR.Enables.I, 1)
    VR[vd].W[i+0] = Sint32_convertToInteger(VR[vk].D[i], FCSR.Enables.I, 1)
```

XVFTINTRZ.W.D:

```
for i in range(2):
    XR[xd].W[i+2] = Sint32_convertToInteger(XR[xj].D[i], FCSR.Enables.I, 1)
    XR[xd].W[i+0] = Sint32_convertToInteger(XR[xk].D[i], FCSR.Enables.I, 1)
    XR[xd].W[i+6] = Sint32_convertToInteger(XR[xj].D[i+2], FCSR.Enables.I,1)
    XR[xd].W[i+4] = Sint32_convertToInteger(XR[xk].D[i+2], FCSR.Enables.I,1)
```

The [X]VFTINTRP.W.D instruction converts each double-precision floating-point element in every 128 bits of vector registers vj/xj into a signed integer. The results are sequentially written into the upper half of each 128-bit in vector register vd/xd. It converts each double-precision floating-point element in vector register vk/xk to a signed integer, writing the results sequentially to the lower half of each 128 bits in vector register vd/xd. The results are rounded toward positive infinity.

VFTINTRP.W.D:

```
for i in range(2):
    VR[vd].W[i+2] = Sint32_convertToInteger(VR[vj].D[i], FCSR.Enables.I, 2)
    VR[vd].W[i+0] = Sint32_convertToInteger(VR[vk].D[i], FCSR.Enables.I, 2)
```

XVFTINTRP.W.D:

```
for i in range(2):
    XR[xd].W[i+2] = Sint32_convertToInteger(XR[xj].D[i], FCSR.Enables.I, 2)
    XR[xd].W[i+0] = Sint32_convertToInteger(XR[xk].D[i], FCSR.Enables.I, 2)
    XR[xd].W[i+6] = Sint32_convertToInteger(XR[xj].D[i+2], FCSR.Enables.I,2)
    XR[xd].W[i+4] = Sint32_convertToInteger(XR[xk].D[i+2], FCSR.Enables.I,2)
```

The [X]VFTINTRM.W.D instruction converts each double-precision floating-point element in every 128 bits of vector registers vj/xj into a signed integer. The results are sequentially written into the upper half of each 128-bit in vector register vd/xd. It converts each double-precision floating-point element in vector register vk/xk to a signed integer, writing the results sequentially to the lower half of each 128 bits in vector register vd/xd. The results are rounded toward negative infinity.

VFTINTRM.W.D:

```
for i in range(2):
    VR[vd].W[i+2] = Sint32_convertToInteger(VR[vj].D[i], FCSR.Enables.I, 3)
    VR[vd].W[i+0] = Sint32_convertToInteger(VR[vk].D[i], FCSR.Enables.I, 3)
```

XVFTINTRM.W.D:

```
for i in range(3):
    XR[xd].W[i+2] = Sint32_convertToInteger(XR[xj].D[i], FCSR.Enables.I, 3)
    XR[xd].W[i+0] = Sint32_convertToInteger(XR[xk].D[i], FCSR.Enables.I, 3)
    XR[xd].W[i+6] = Sint32_convertToInteger(XR[xj].D[i+2], FCSR.Enables.I,3)
    XR[xd].W[i+4] = Sint32_convertToInteger(XR[xk].D[i+2], FCSR.Enables.I,3)
```

The [X]VFTINT.W.D instruction converts each double-precision floating-point element in every 128 bits of vector registers vj/xj into a signed integer. The results are sequentially written into the upper half of each 128-bit in vector register vd/xd. It converts each double-precision floating-point element in vector register vk/xk to a signed integer, writing the results sequentially to the lower half of each 128 bits in vector register vd/xd. The rounding mode for the results is determined by the RM field in FCSR.

VFTINT.W.D:

```
for i in range(2):
    VR[vd].W[i+2] = Sint32_convertToInteger(VR[vj].D[i], FCSR.Enables.I, FCSR.RM)
    VR[vd].W[i+0] = Sint32_convertToInteger(VR[vk].D[i], FCSR.Enables.I, FCSR.RM)
```

XVFTINT.W.D:

```
for i in range(2) :
    XR[xd].W[i+2] = Sint32_convertToInteger(XR[xj].D[i], FCSR.Enables.I, FCSR.RM)
    XR[xd].W[i+0] = Sint32_convertToInteger(XR[xk].D[i], FCSR.Enables.I, FCSR.RM)
    XR[xd].W[i+6] = Sint32_convertToInteger(XR[xj].D[i+2], FCSR.Enables.I,FCSR.RM)
    XR[xd].W[i+4] = Sint32_convertToInteger(XR[xk].D[i+2], FCSR.Enables.I, FCSR.RM)
```

The [X]VFTINTRNEL.L.S instruction converts each single-precision floating-point element in the lower half of each 128 bits within vector registers vj/xj into a signed integer double-word. The results are sequentially written to vector registers vd/xd. The results are rounded using the "round-to-nearest (even-if-odd)" method.

VFTINTRNEL.L.S:

```
for i in range(2):
    VR[vd].D[i] = Sint64_convertToInteger(VR[vj].S[i], FCSR.Enables.I, 0)
```

XVFTINTRNEL.L.S:

```
for i in range(2):
    XR[xd].D[i+0] = Sint64_convertToInteger(XR[xj].S[i], FCSR.Enables.I, 0)
    XR[xd].D[i+2] = Sint64_convertToInteger(XR[xj].S[i+4], FCSR.Enables.I,0)
```

The [X]VFTINTRNEH.L.S instruction converts each single-precision floating-point element in the upper half of every 128 bits in vector registers vj/xj into a signed integer doubleword. The results are sequentially written to vector registers vd/xd. The results are rounded using the "round-to-nearest (even-if-odd)" method.

VFTINTRNEH.L.S:

```
for i in range(2):
    VR[vd].D[i] = Sint64_convertToInteger(VR[vj].S[i+2], FCSR.Enables.I, 0)
```

XVFTINTRNEH.L.S:

```
for i in range(2):
    XR[xd].D[i+0] = Sint64_convertToInteger(XR[xj].S[i+2], FCSR.Enables.I,0)
    XR[xd].D[i+2] = Sint64_convertToInteger(XR[xj].S[i+6], FCSR.Enables.I,0)
```

The [X]VFTINTRZL.L.S instruction converts each single-precision floating-point element in the lower half of every 128 bits in vector registers vj/xj into a signed integer double-word. The results are sequentially written to vector registers vd/xd. The results are rounded toward zero.

VFTINTRZL.L.S:

```
for i in range(2) :
    VR[vd].D[i] = Sint64_convertToInteger(VR[vj].S[i], FCSR.Enables.I, 1)
```

XVFTINTRZL.L.S:

```
for i in range(2):
    XR[xd].D[i+0] = Sint64_convertToInteger(XR[xj].S[i], FCSR.Enables.I, 1)
    XR[xd].D[i+2] = Sint64_convertToInteger(XR[xj].S[i+4], FCSR.Enables.I,1)
```

The [X]VFTINTRZH.L.S instruction converts each single-precision floating-point element in the upper half of every 128 bits in vector registers vj/xj into a signed integer double-word. The results are sequentially written to vector registers vd/xd. The results are rounded toward zero.

VFTINTRZH.L.S:

```
for i in range(2):
    VR[vd].D[i] = Sint64_convertToInteger(VR[vj].S[i+2], FCSR.Enables.I, 1)
```

XVFTINTRZH.L.S:

```
for i in range(2):
    XR[xd].D[i+0] = Sint64_convertToInteger(XR[xj].S[i+2], FCSR.Enables.I,1)
    XR[xd].D[i+2] = Sint64_convertToInteger(XR[xj].S[i+6], FCSR.Enables.I,1)
```

The [X]VFTINTRPL.L.S instruction converts each single-precision floating-point element in the lower half of every 128 bits in vector registers vj/xj into a signed integer double-word. The results are sequentially written to vector registers vd/xd. The results are rounded toward positive infinity.

VFTINTRPL.L.S:

```
for i in range(2) :
    VR[vd].D[i] = Sint64_convertToInteger(VR[vj].S[i], FCSR.Enables.I, 2)
```

XVFTINTRPL.L.S:

```
for i in range(2):
    XR[xd].D[i+0] = Sint64_convertToInteger(XR[xj].S[i], FCSR.Enables.I, 2)
    XR[xd].D[i+2] = Sint64_convertToInteger(XR[xj].S[i+4], FCSR.Enables.I,2)
```

The [X]VFTINTRPH.L.S instruction converts each single-precision floating-point element in the upper half of every 128 bits in vector registers vj/xj into a signed integer double-word. The results are sequentially written to vector registers vd/xd. The results are rounded toward positive infinity.

VFTINTRPH.L.S:

```
for i in range(2) :
    VR[vd].D[i] = Sint64_convertToInteger(VR[vj].S[i+2], FCSR.Enables.I, 2)
```

XVFTINTRPH.L.S:

```
for i in range(2):
    XR[xd].D[i+0] = Sint64_convertToInteger(XR[xj].S[i+2], FCSR.Enables.I,2)
    XR[xd].D[i+2] = Sint64_convertToInteger(XR[xj].S[i+6], FCSR.Enables.I, 2)
```

The [X]VFTINTRML.L.S instruction converts each single-precision floating-point element in the lower half of every 128 bits in vector registers vj/xj into a signed integer double-word. The results are sequentially written to vector registers vd/xd. The results are rounded toward negative infinity.

VFTINTRML.L.S:

```
for i in range(2) :
    VR[vd].D[i] = Sint64_convertToInteger(VR[vj].S[i], FCSR.Enables.I, 3)
```

XVFTINTRML.L.S:

```
for i in range(2):
    XR[xd].D[i+0] = Sint64_convertToInteger(XR[xj].S[i], FCSR.Enables.I, 3)
    XR[xd].D[i+2] = Sint64_convertToInteger(XR[xj].S[i+4], FCSR.Enables.I, 3)
```

The [X]VFTINTRMH.L.S instruction converts each single-precision floating-point element in the upper half of every 128 bits in vector registers vj/xj into a signed integer double-word. The results are sequentially written to vector registers vd/xd. The results are rounded toward negative infinity.

VFTINTRMH.L.S:

```
for i in range(2) :
    VR[vd].D[i] = Sint64_convertToInteger(VR[vj].S[i+2], FCSR.Enables.I, 3)
```

XVFTINTRMH.L.S:

```
for i in range(2):
    XR[xd].D[i+0] = Sint64_convertToInteger(XR[xj].S[i+2], FCSR.Enables.I,3)
    XR[xd].D[i+2] = Sint64_convertToInteger(XR[xj].S[i+6], FCSR.Enables.I, 3)
```

The [X]VFTINTL.L.S instruction converts each single-precision floating-point element in the lower half of every 128 bits in vector register vj/xj into a signed integer double-word. The results are sequentially written to vector registers vd/xd. The rounding mode for the results is determined by the RM field in FCSR.

VFTINTL.L.S:

```
for i in range(2) :
    VR[vd].D[i] = Sint64_convertToInteger(VR[vj].S[i], FCSR.Enables.I, FCSR. RM)
```

XVFTINTL.L.S:

```
for i in range(2):
    XR[xd].D[i+0] = Sint64_convertToInteger(XR[xj].S[i], FCSR.Enables.I, FCSR.RM)
    XR[xd].D[i+2] = Sint64_convertToInteger(XR[xj].S[i+4], FCSR.Enables.I, FCSR.RM)
```

The [X]VFTINTH.L.S instruction converts each single-precision floating-point element in the upper half of every 128 bits in the vector register vj/xj into a signed integer double-word. The results are sequentially written to vector register vd/xd. The rounding mode for the results is determined by the RM field in FCSR.

VFTINTH.L.S:

```
for i in range(2) :
    VR[vd].D[i] = Sint64_convertToInteger(VR[vj].S[i+2], FCSR.Enables.I, FCSR.RM)
```

XVFTINTH.L.S:

```
for i in range(2):
    XR[xd].D[i+0] = Sint64_convertToInteger(XR[xj].S[i+2], FCSR.Enables.I, FCSR.RM)
    XR[xd].D[i+2] = Sint64_convertToInteger(XR[xj].S[i+6], FCSR.Enables.I, FCSR.RM)
```

1.2.5.4 [X]VFFINT.{S.W/D.L}[U], [X]VFFINT.S.L, [X]VFFINT{L/H}.D.W

Instruction Format:				
	vffint.s.w	vd, vj	xvffint.s.w	xd, xj
	vffint.d.l	vd, vj	xvffint.d.l	xd, xj
	vffint.s.wu	vd, vj	xvffint.s.wu	xd, xj
	vffint.d.lu	vd, vj	xvffint.d.lu	xd, xj
	vffintl.d.w	vd, vj	xvffintl.d.w	xd, xj
	vffinth.d.w	vd, vj	xvffinth.d.w	xd, xj
	vffint.s.l	vd, vj	xvffint.s.l	xd, xj

The [X]VFFINT.{S.W/D.L} instruction converts the signed integer/double-word elements in vector registers vj/xj to single-precision/double-precision floating-point numbers, writing the results to the corresponding elements in vector registers vd/xd. The rounding mode for the results is determined by the RM

field in the FCSR.

VFFINT.S.W:

```
for i in range(4) :
    VR[vd].S[i] = FP32_convertFromInt(VR[vj].W[i], SINT32)
```

VFFINT.D.L:

```
for i in range(2):
    VR[vd].D[i] = FP64_convertFromInt(VR[vj].L[i], SINT64)
```

XVFFINT.S.W:

```
for i in range(8):
    XR[xd].S[i] = FP32_convertFromInt(XR[xj].W[i], SINT32)
```

XVFFINT.D.L:

```
for i in range(4):
    XR[xd].D[i] = FP64_convertFromInt(XR[xj].L[i], SINT64)
```

The [X]VFFINT.{S.W/D.L}U instruction converts each unsigned integer/double-word element in vector register vj/xj to a single-precision/double-precision floating-point number, writing the result to the corresponding element in vector register vd/xd. The rounding mode for the result is determined by the RM field in the FCSR.

VFFINT.S.WU:

```
for i in range(4):
    VR[vd].S[i] = FP32_convertFromInt(VR[vj].W[i], UINT32)
```

VFFINT.D.LU:

```
for i in range(2):
    VR[vd].D[i] = FP64_convertFromInt(VR[vj].L[i], UINT64)
```

XVFFINT.S.WU:

```
for i in range(8):
    XR[xd].S[i] = FP32_convertFromInt(XR[xj].W[i], UINT32)
```

XVFFINT.D.LU:

```
for i in range(4):
    XR[xd].D[i] = FP64_convertFromInt(XR[xj].L[i], UINT64)
```

The [X]VFFINT.S.L instruction converts each signed integer double-word element in every 128 bits of vector registers vj/xj into a single-precision floating-point number. The results are sequentially written to the upper half of each 128 bits in vector registers vd/xd. It converts each signed integer double-word element in vector register vk/xk to a single-precision floating-point number, writing the results sequentially to the upper half of each 128 bits in vector register vd/xd. The rounding mode for the results is determined by the RM field in FCSR.

VFFINT.S.L:

```
for i in range(2):
    VR[vd].S[i+2] = FP32_convertFromInt(VR[vj].L[i], UINT64)
    VR[vd].S[i+0] = FP32_convertFromInt(VR[vk].L[i], UINT64)
```

XVFFINT.S.L:

```
for i in range(2):
    XR[xd].S[i+2] = FP32_convertFromInt(XR[xj].L[i], UINT64)
    XR[xd].S[i+0] = FP32_convertFromInt(XR[xk].L[i], UINT64)
    XR[xd].S[i+6] = FP32_convertFromInt(XR[xj].L[i+2], UINT64)
    XR[xd].S[i+4] = FP32_convertFromInt(XR[xk].L[i+2], UINT64)
```

The [X]VFFINTL.D.W instruction converts each signed integer element in each lower 128 bits of vector register vj/xj into a double-precision floating-point number, writing the results sequentially into vector register vd/xd. The rounding mode for the results is determined by the RM field in the FCSR.

VFFINTL.D.W:

```
for i in range(2) :
    VR[vd].D[i] = FP64_convertFromInt(VR[vj].W[i], SINT32)
```

XVFFINTL.D.W:

```
for i in range(2):
    XR[xd].D[i+0] = FP64_convertFromInt(XR[xj].W[i], SINT32)
    XR[xd].D[i+2] = FP64_convertFromInt(XR[xj].W[i+4], SINT32)
```

The [X]VFFINTH.D.W instruction converts each signed integer element in each upper 128 bits of vector register vj/xj into a double-precision floating-point number, writing the results sequentially into vector register vd/xd. The rounding mode for the results is determined by the RM field in the FCSR.

VFFINTH.D.W:

```
for i in range(2) :
    VR[vd].D[i] = FP64_convertFromInt(VR[vj].W[i+2], SINT32)
```

XVFFINTH.D.W:

```
for i in range(2):
    XR[xd].D[i+0] = FP64_convertFromInt(XR[xj].W[i+2], SINT32)
    XR[xd].D[i+2] = FP64_convertFromInt(XR[xj].W[i+6], SINT32)
```

1.2.6 Vector Integer Comparison, Floating-Point Comparison, and Selection Instructions

1.2.6.1 [X]VSEQ[I].{B/H/W/D}

Instruction Format:	vseq.b	vd, vj, vk	xvseq.b	xd, xj, xk
	vseq.h	vd, vj, vk	xvseq.h	xd, xj, xk
	vseq.w	vd, vj, vk	xvseq.w	xd, xj, xk
	vseq.d	vd, vj, vk	xvseq.d	xd, xj, xk
	vseqi.b	vd, vj, si5	xvseqi.b	xd, xj, si5
	vseqi.h	vd, vj, si5	xvseqi.h	xd, xj, si5
	vseqi.w	vd, vj, si5	xvseqi.w	xd, xj, si5
	vseqi.d	vd, vj, si5	xvseqi.d	xd, xj, si5

The [X]VSEQ.{B/H/W/D} instruction compares the integer byte/half-word/word/double-word elements in vector register vj/xj with the corresponding integer byte/half-word/word/double-word elements in vector register vk/xk. If two elements are equal, the corresponding integer byte/half-word/word/double-word elements in vector register vd/xd are set to all 1, or they are set to all 0.

VSEQ.B:

```
for i in range(16):
    VR[vd].B[i] = (VR[vj].B[i] == VR[vk].B[i]) ? {8{1'b1}} : 0
```

VSEQ.H:

```
for i in range(8):
    VR[vd].H[i] = (VR[vj].H[i] == VR[vk].H[i]) ? {16{1'b1}} : 0
```

VSEQ.W:

```
for i in range(4):
    VR[vd].W[i] = (VR[vj].W[i] == VR[vk].W[i]) ? {32{1'b1}} : 0
```

VSEQ.D:

```
for i in range(2):
    VR[vd].D[i] = (VR[vj].D[i] == VR[vk].D[i]) ? {64{1'b1}} : 0
```

XVSEQ.B:

```
for i in range(32):
    XR[xd].B[i] = (XR[xj].B[i] == XR[xk].B[i]) ? {8{1'b1}} : 0
```

XVSEQ.H:

```
for i in range(16):
    XR[xd].H[i] = (XR[xj].H[i] == XR[xk].H[i]) ? {16{1'b1}} : 0
```

XVSEQ.W:

```
for i in range(8):
    XR[xd].W[i] = (XR[xj].W[i] == XR[xk].W[i]) ? {32{1'b1}} : 0
```

XVSEQ.D:

```
for i in range(4):
    XR[xd].D[i] = (XR[xj].D[i] == XR[xk].D[i]) ? {64{1'b1}} : 0
```

The [X]VSEQI.{B/H/W/D} instruction compares the integer byte/half-word/word/double-word elements in vector register vj/xj with immediate value si5, after sign-extending to integer byte/half-word/word/double-word. If they are equal, the corresponding integer byte/half-word/word/double-word elements in vector register vd/xd are set to all 1, or they are set to all 0.

VSEQI.B:

```
for i in range(16):
    VR[vd].B[i] = (VR[vj].B[i] == SignExtend(si5, 8)) ? {8{1'b1}} : 0
```

VSEQI.H:

```
for i in range(8):
    VR[vd].B[i] = (VR[vj].B[i] == SignExtend(si5, 16)) ? {16{1'b1}} : 0
```

VSEQI.W:

```
for i in range(4):
    VR[vd].B[i] = (VR[vj].B[i] == SignExtend(si5, 32)) ? {32{1'b1}} : 0
```

VSEQI.D:

```
for i in range(2):
    VR[vd].B[i] = (VR[vj].B[i] == SignExtend(si5, 64)) ? {64{1'b1}} : 0
```

XVSEQI.B:

```
for i in range(32):
    XR[xd].B[i] = (XR[xj].B[i] == SignExtend(si5, 8)) ? {8{1'b1}} : 0
```

XVSEQI.H:

```
for i in range(16):
    XR[xd].B[i] = (XR[xj].B[i] == SignExtend(si5, 16)) ? {16{1'b1}} : 0
```

XVSEQI.W:

```
for i in range(8):
    XR[xd].B[i] = (XR[xj].B[i] == SignExtend(si5, 32)) ? {32{1'b1}} : 0
```

XVSEQI.D:

```
for i in range(4):
    XR[xd].B[i] = (XR[xj].B[i] == SignExtend(si5, 64)) ? {64{1'b1}} : 0
```

1.2.6.2 [X]VSLE[I].{B/H/W/D}[U]

Instruction Format:

vsle.b	vd, vj, vk	xvsle.b	xd, xj, xk
vsle.h	vd, vj, vk	xvsle.h	xd, xj, xk
vsle.w	vd, vj, vk	xvsle.w	xd, xj, xk
vsle.d	vd, vj, vk	xvsle.d	xd, xj, xk
vslei.b	vd, vj, si5	xvslei.b	xd, xj, si5
vslei.h	vd, vj, si5	xvslei.h	xd, xj, si5
vslei.w	vd, vj, si5	xvslei.w	xd, xj, si5
vslei.d	vd, vj, si5	xvslei.d	xd, xj, si5
vsle.bu	vd, vj, vk	xvsle.bu	xd ,xj, xk
vsle.hu	vd, vj, vk	xvsle.hu	xd, xj, xk
vsle.wu	vd, vj, vk	xvsle.wu	xd, xj, xk
vsle.du	vd, vj, vk	xvsle.du	xd, xj, xk
vslei.bu	vd, vj, ui5	xvslei.bu	xd, xj, ui5
vslei.hu	vd, vj, ui5	xvslei.hu	xd, xj, ui5

vslei.wu	vd, vj, ui5	xvslei.wu	xd, xj, ui5
vslei.du	vd, vj, ui5	xvslei.du	xd, xj, ui5

The [X]VSLE.{B/H/W/D} instruction compares the integer byte/half-word/word/double-word elements in vector register vj/xj with the corresponding integer byte/half-word/word/double-word elements in vector register vk/xk, treating all elements as signed numbers. If the former is less than or equal to the latter, the corresponding integer byte/half-word/word/double-word elements in vector register vd/xd are set to all 1, or they are set to all 0.

VSLE.B:

```
for i in range(16):
    VR[vd].B[i] = (signed(VR[vj].B[i]) <= signed(VR[vk].B[i])) ? {8{1'b1}} : 0
```

VSLE.H:

```
for i in range(8):
    VR[vd].H[i] = (signed(VR[vj].H[i]) <= signed(VR[vk].H[i])) ? {16{1'b1}} : 0
```

VSLE.W:

```
for i in range(4) :
    VR[vd].W[i] = (signed(VR[vj].W[i]) <= signed(VR[vk].W[i])) ? {32{1'b1}} :0
```

VSLE.D:

```
for i in range(2):
    VR[vd].D[i] = (signed(VR[vj].D[i]) <= signed(VR[vk].D[i])) ? {64{1'b1}} : 0
```

XVSLE.B:

```
for i in range(32):
    XR[xd].B[i] = (signed(XR[xj].B[i]) <= signed(XR[xk].B[i])) ? {8{1'b1}} : 0
```

XVSLE.H:

```
for i in range(16):
    XR[xd].H[i] = (signed(XR[xj].H[i]) <= signed(XR[xk].H[i])) ? {16{1'b1}} : 0
```

XVSLE.W:

```
for i in range(8):
    XR[xd].W[i] = (signed(XR[xj].W[i]) <= signed(XR[xk].W[i])) ? {32{1'b1}} : 0
```

XVSLE.D:

```
for i in range(4):
    XR[xd].D[i] = (signed(XR[xj].D[i]) <= signed(XR[xk].D[i])) ? {64{1'b1}} : 0
```

The [X]VSLEI.{B/H/W/D} instruction compares the integer byte/half-word/word/double-word elements in vector register vj/xj with immediate number si5, after sign-extending to the corresponding integer byte/half-word/word/double-word. If the former is less than or equal to the latter, the corresponding integer byte/half-word/word/double-word elements in vector register vd/xd are set to all 1, or they are set to all 0.

VSLEI.B:

```
for i in range(16):
    VR[vd].B[i] = (signed(VR[vj].B[i]) <= SignExtend(si5, 8)) ? {8{1'b1}} : 0
```

VSLEI.H:

```
for i in range(8):
    VR[vd].H[i] = (signed(VR[vj].H[i]) <= SignExtend(si5, 16)) ? {16{1'b1}} : 0
```

VSLEI.W:

```
for i in range(4):
    VR[vd].W[i] = (signed(VR[vj].W[i]) <= SignExtend(si5, 32)) ? {32{1'b1}} : 0
```

VSLEI.D:

```
for i in range(2):
    VR[vd].D[i] = (signed(VR[vj].D[i]) <= SignExtend(si5, 64)) ? {64{1'b1}} : 0
```

XVSLEI.B:

```
for i in range(32):
    XR[xd].B[i] = (signed(XR[xj].B[i]) <= SignExtend(si5, 8)) ? {8{1'b1}} : 0
```

XVSLEI.H:

```
for i in range(16):
    XR[xd].H[i] = (signed(XR[xj].H[i]) <= SignExtend(si5, 16)) ? {16{1'b1}} : 0
```

XVSLEI.W:

```
for i in range(8):
    XR[xd].W[i] = (signed(XR[xj].W[i]) <= SignExtend(si5, 32)) ? {32{1'b1}} : 0
```

XVSLEI.D:

```
for i in range(4):
    XR[xd].D[i] = (signed(XR[xj].D[i]) <= SignExtend(si5, 64)) ? {64{1'b1}} : 0
```

The [X]VSLE.{B/H/W/D}U instruction compares the integer byte/half-word/word/double-word elements in vector register vj/xj with the corresponding integer byte/half-word/word/double-word elements in vector register vk/xk, treating all elements as unsigned numbers. If the former is less than or equal to the latter, the corresponding integer byte/half-word/word/double-word elements in vector register vd/xd are set to all 1, or they are set to all 0.

VSLE.BU:

```
for i in range(16):
    VR[vd].B[i] = (unsigned(VR[vj].B[i]) <= unsigned(VR[vk].B[i])) ? {8{1'b1}} : 0
```

VSLE.HU:

```
for i in range(8) :
    VR[vd].H[i] = (unsigned(VR[vj].H[i]) <= unsigned(VR[vk].H[i])) ? {16{1'b 1} : 0
```

VSLE.WU:

```
for i in range(4):
    VR[vd].W[i] = (unsigned(VR[vj].W[i]) <= unsigned(VR[vk].W[i])) ? {32{1'b 1}} : 0
```

VSLE.DU:

```
for i in range(2) :
    VR[vd].D[i] = (unsigned(VR[vj].D[i]) <= unsigned(VR[vk].D[i])) ? {64{1'b 1}} : 0
```

XVSLE.BU:

```
for i in range(32):
    XR[xd].B[i] = (unsigned(XR[xj].B[i]) <= unsigned(XR[xk].B[i])) ? {8{1'b1}} : 0
```

XVSLE.HU:

```
for i in range(16):
    XR[xd].H[i] = (unsigned(XR[xj].H[i]) <= unsigned(XR[xk].H[i])) ? {16{1'b 1} : 0
```

XVSLE.WU:

```
for i in range(8):
    XR[xd].W[i] = (unsigned(XR[xj].W[i]) <= unsigned(XR[xk].W[i])) ? {32{1'b 1}} : 0
```

XVSLE.DU:

```
for i in range(4) :
    XR[xd].D[i] = (unsigned(XR[xj].D[i]) <= unsigned(XR[xk].D[i])) ? {64{1'b 1'b}} : 0
```

The [X]VSLEI.{B/H/W/D}U instruction compares the integer byte/half-word/word/double-word elements in vector register vj/xj with immediate number ui5, after zero-extending to the corresponding integer byte/half-word/word/double-word. If the former is less than or equal to the latter, the corresponding integer byte/half-word/word/double-word elements in vector register vd/xd are set to all 1, or they are set to all 0.

VSLEI.BU:

```
for i in range(16):
    VR[vd].B[i] = (unsigned(VR[vj].B[i]) <= ZeroExtend(ui5, 8)) ? {8{1'b1}} : 0
```

VSLEI.HU:

```
for i in range(8):
    VR[vd].H[i] = (unsigned(VR[vj].H[i]) <= ZeroExtend(ui5, 16)) ? {16{1'b1}}: 0
```

VSLEI.WU:

```
for i in range(4):
    VR[vd].W[i] = (unsigned(VR[vj].W[i]) <= ZeroExtend(ui5, 32)) ? {32{1'b1}} : 0
```

VSLEI.DU:

```
for i in range(2):
    VR[vd].D[i] = (unsigned(VR[vj].D[i]) <= ZeroExtend(ui5, 64)) ? {64{1'b1}}: 0
```

XVSLEI.BU:

```
for i in range(32):
    XR[xd].B[i] = (unsigned(XR[xj].B[i]) <= ZeroExtend(ui5, 8)) ? {8{1'b1}} : 0
```

XVSLEI.HU:

```
for i in range(16):
    XR[xd].H[i] = (unsigned(XR[xj].H[i]) <= ZeroExtend(ui5, 16)) ? {16{1'b1}}: 0
```

XVSLEI.WU:

```
for i in range(8):
    XR[xd].W[i] = (unsigned(XR[xj].W[i]) <= ZeroExtend(ui5, 32)) ? {32{1'b1}} : 0
```

XVSLEI.DU:

```
for i in range(4):
    XR[xd].D[i] = (unsigned(XR[xj].D[i]) <= ZeroExtend(ui5, 64)) ? {64{1'b1}}: 0
```

1.2.6.3 [X]VSLT[I].{B/H/W/D}[U]

Instruction Format:

vslt.b	vd, vj, vk	xvslt.b	xd, xj, xk
vslt.h	vd, vj, vk	xvslt.h	xd, xj, xk
vslt.w	vd, vj, vk	xvslt.w	xd, xj, xk
vslt.d	vd, vj, vk	xvslt.d	xd, xj, xk
vslti.b	vd, vj, si5	xvslti.b	xd, xj, si5
vslti.h	vd, vj, si5	xvslti.h	xd, xj, si5
vslti.w	vd, vj, si5	xvslti.w	xd, xj, si5
vslti.d	vd, vj, si5	xvslti.d	xd, xj, si5
vslt.bu	vd, vj, vk	xvslt.bu	xd, xj, xk
vslt.hu	vd, vj, vk	xvslt.hu	xd, xj, xk
vslt.wu	vd, vj, vk	xvslt.wu	xd, xj, xk
vslt.du	vd, vj, vk	xvslt.du	xd, xj, xk
vslti.bu	vd, vj, ui5	xvslti.bu	xd, xj, ui5
vslti.hu	vd, vj, ui5	xvslti.hu	xd, xj, ui5
vslti.wu	vd, vj, ui5	xvslti.wu	xd, xj, ui5
vslti.du	vd, vj, ui5	xvslti.du	xd, xj, ui5

The [X]VSLT.{B/H/W/D} instruction compares the integer byte/half-word/word/double-word elements in vector register vj/xj with the corresponding integer byte/half-word/word/double-word elements in vector register vk/xk, treating all elements as signed numbers. If the former is less than the latter, the corresponding integer byte/half-word/word/double-word elements in vector register vd/xd are set to all 1, or they are set to all 0.

VSLT.B:

```
for i in range(16):
    VR[vd].B[i] = (signed(VR[vj].B[i]) < signed(VR[vk].B[i])) ? {8{1'b1}} : 0
```

VSLT.H:

```
for i in range(8):
    VR[vd].H[i] = (signed(VR[vj].H[i]) < signed(VR[vk].H[i])) ? {16{1'b1}} : 0
```

VSLT.W:

```
for i in range(4):
    VR[vd].W[i] = (signed(VR[vj].W[i]) < signed(VR[vk].W[i])) ? {32{1'b1}} : 0
```

VSLT.D:

```
for i in range(2):
    VR[vd].D[i] = (signed(VR[vj].D[i]) < signed(VR[vk].D[i])) ? {64{1'b1}} : 0
```

XVSLT.B:

```
for i in range(32) :
    XR[xd].B[i] = (signed(XR[xj].B[i]) < signed(XR[xk].B[i])) ? {8{1'b1}} : 0
```

XVSLT.H:

```
for i in range(16):
    XR[xd].H[i] = (signed(XR[xj].H[i]) < signed(XR[xk].H[i])) ? {16{1'b1}} : 0
```

XVSLT.W:

```
for i in range(8):
    XR[xd].W[i] = (signed(XR[xj].W[i]) < signed(XR[xk].W[i])) ? {32{1'b1}} : 0
```

XVSLT.D:

```
for i in range(4) :
    XR[xd].D[i] = (signed(XR[xj].D[i]) < signed(XR[xk].D[i])) ? {64{1'b1}} : 0
```

The [X]VSLTI.{B/H/W/D} instruction compares the integer byte/half-word/word/double-word elements in vector register vj/xj with immediate value si5, after sign-extending to integer byte/half-word/word/double-word. If the former is less than the latter, the corresponding integer byte/half-word/word/double-word elements in vector register vd/xd are set to all 1, or they are set to all 0.

VSLTI.B:

```
for i in range(16) :
    VR[vd].B[i] = (signed(VR[vj].B[i]) < SignExtend(si5, 8)) ? {8{1'b1}} : 0
```

VSLTI.H:

```
for i in range(8):
    VR[vd].H[i] = (signed(VR[vj].H[i]) < SignExtend(si5, 16)) ? {16{1'b1}} : 0
```

VSLTI.W:

```
for i in range(4):
    VR[vd].W[i] = (signed(VR[vj].W[i]) < SignExtend(si5, 32)) ? {32{1'b1}}  0
```

VSLTI.D:

```
for i in range(2) :
    VR[vd].D[i] = (signed(VR[vj].D[i]) < SignExtend(si5, 64)) ? {64{1'b1}} : 0
```

XVSLTI.B:

```
for i in range(32):
    XR[xd].B[i] = (signed(XR[xj].B[i]) < SignExtend(si5, 8)) ? {8{1'b1}} : 0
```

XVSLTI.H:

```
for i in range(16):
    XR[xd].H[i] = (signed(XR[xj].H[i]) < SignExtend(si5, 16)) ? {16{1'b1}} : 0
```

XVSLTI.W:

```
for i in range(8):
    XR[xd].W[i] = (signed(XR[xj].W[i]) < SignExtend(si5, 32)) ? {32{1'b1}} : 0
```

XVSLTI.D:

```
for i in range(4):
    XR[xd].D[i] = (signed(XR[xj].D[i]) < SignExtend(si5, 64)) ? {64{1'b1}} : 0
```

The [X]VSLT.{B/H/W/D}U instruction compares the integer byte/half-word/word/double-word elements in vector register vj/xj with the corresponding integer byte/half-word/word/double-word elements in vector register vk/xk, treating all elements as unsigned numbers. If the former is less than the latter, the corresponding integer byte/half-word/word/double-word elements in vector register vd/xd are set to all 1, or they are set to all 0.

VSLT.BU:

```
for i in range(16):
    VR[vd].B[i] = (unsigned(VR[vj].B[i]) < unsigned(VR[vk].B[i])) ? {8{1'b1}} : 0
```

VSLT.HU:

```
for i in range(8) :
    VR[vd].H[i] = (unsigned(VR[vj].H[i]) < unsigned(VR[vk].H[i])) ? {16{1'b1}}: 0
```

VSLT.WU:

```
for i in range(4):
    VR[vd].B[i] = (unsigned(VR[vj].W[i]) < unsigned(VR[vk].W[i])) ? {32{1'b1}} : 0
```

VSLT.DU:

```
for i in range(2) :
    VR[vd].H[i] = (unsigned(VR[vj].D[i]) < unsigned(VR[vk].D[i])) ? {64{1'b1}}: 0
```

XVSLT.BU:

```
for i in range(32):
    XR[xd].B[i] = (unsigned(XR[xj].B[i]) < unsigned(XR[xk].B[i])) ? {8{1'b1}} : 0
```

XVSLT.HU:

```
for i in range(16) :
    XR[xd].H[i] = (unsigned(XR[xj].H[i]) < unsigned(XR[xk].H[i])) ? {16{1'b1}}: 0
```

XVSLT.WU:

```
for i in range(8):
    XR[xd].W[i] = (unsigned(XR[xj].W[i]) < unsigned(XR[xk].W[i])) ? {32{1'b1}} : 0
```

XVSLT.DU:

```
for i in range(4) :
    XR[xd].D[i] = (unsigned(XR[xj].D[i]) < unsigned(XR[xk].D[i])) ? {64{1'b1}}: 0
```

The [X]VSLTI.{B/H/W/D}U instruction compares the integer byte/half-word/word/double-word elements in vector register vj/xj with immediate value ui5, after zero-extending to integer byte/half-word/word/double-word. If the former is less than the latter, the corresponding integer byte/half-word/word/double-word elements in vector register vd/xd are set to all 1, or they are set to all 0.

VSLTI.BU:

```
for i in range(16) :
    VR[vd].B[i] = (unsigned(VR[vj].B[i]) < ZeroExtend(ui5, 8)) ? {8{1'b1}} : 0
```

VSLTI.HU:

```
for i in range(8) :
    VR[vd].H[i] = (unsigned(VR[vj].H[i]) < ZeroExtend(ui5, 16)) ? {16{1'b1}} : 0
```

VSLTI.WU:

```
for i in range(4) :
    VR[vd].W[i] = (unsigned(VR[vj].W[i]) < ZeroExtend(ui5, 32)) ? {32{1'b1}} : 0
```

VSLTI.DU:

```
for i in range(2):
    VR[vd].D[i] = (unsigned(VR[vj].D[i]) < ZeroExtend(ui5, 64)) ? {64{1'b1}} : 0
```

XVSLTI.BU:

```
for i in range(32):
    XR[xd].B[i] = (unsigned(XR[xj].B[i]) < ZeroExtend(ui5, 8)) ? {8{1'b1}} : 0
```

XVSLTI.HU:

```
for i in range(16):
    XR[xd].H[i] = (unsigned(XR[xj].H[i]) < ZeroExtend(ui5, 16)) ? {16{1'b1}} : 0
```

XVSLTI.WU:

```
for i in range(8):
    XR[xd].W[i] = (unsigned(XR[xj].W[i]) < ZeroExtend(ui5, 32)) ? {32{1'b1}} :0
```

XVSLTI.DU:

```
for i in range(4):
    XR[xd].D[i] = (unsigned(XR[xj].D[i]) < ZeroExtend(ui5, 64)) ? {64{1'b1}} : 0
```

1.2.6.4 [X]VFCMP.cond.{S/D}

Instruction Format:	vfcmp.cond.s	vd, vj, vk	xvfcmp.cond.s	xd, xj, xk
	vfcmp.cond.d	vd, vj, vk	xvfcmp.cond.d	xd, xj, xk

The [X]VFCMP.cond.The {S/D} instruction performs floating-point comparisons between each single-precision/double-precision floating-point element in vector register vj/xj and the corresponding single-precision/double-precision floating-point element at the same position in vector register vk/xk. Based on whether the comparison result is true or false, it sets the corresponding single-precision/double-precision floating-point elements in vector register vd/xd to all 1 or all 0.

This instruction supports 22 types of floating-point comparison conditions (cond). These comparison conditions and their corresponding judgment criteria are shown in the table below.

Mnemonic	cond	Meaning	True Condition	QNaN Whether Throw an exception	Corresponding IEEE 754-2008 Function
CAF	0x0	No	None	No	
CUN	0x8	Cannot be compared	UN		compareQuietUnordered
CEQ	0x4	equal	EQ		compareQuietEqual
CUEQ	0xC	Equal or cannot be compared	UN EQ		
CLT	0x2	Less than	LT		compareQuietLess
CULT	0xE	Less Than or Not Comparable	UN LT		compareQuietLessUnordered
CLE	0x6	Less Than or Equal To	LT EQ		compareQuietLessEqual
CULE	0xE	Less than or equal to or cannot be compared	UN LT EQ		compareQuietNotGreater
CNE	0x10	Not equal	GT LT		
COR	0x14	Ordered	GT LT EQ		
CUNE	0x18	Cannot be compared or unequal	UN GT LT		compareSignalingNotEqual
SAF	0x1	No	None	Yes	
SUN	0x9	Not greater than, less than, or equal to	UN		
SEQ	0x5	equal	EQ		compareSignalingEqual
SUEQ	0xD	Not greater than or less than	UN EQ		
SLT	0x3	Less than	LT		compareSignalingLess
SULT	0xB	Not greater than or equal to	UN LT		compareSignalingLessUnordered
SLE	0x7	Less than or equal to	LT EQ		compareSignalingLessEqual
SULE	0xF	Not Greater Than	UN LT EQ		compareSignalingNotGreater
SNE	0x11	Not Greater Than	GT LT		
SOR	0x15	Ordered	GT LT EQ		
SUNE	0x19	Cannot be compared or unequal	UN GT LT		

Note: UN indicates cannot be compared, EQ indicates equal, LT indicates less than. When at least one of two operands is NaN, the two numbers cannot be compared.

VFCMP.cond.S:

```
for i in range(4) :
    VR[vd].S[i] = Floating_Point_Compare(VR[vj].S[i], VR[vk].S[i], Single, cond) ? {32{1'b1}} : 0
```

VFCMP.cond.D:

```
for i in range(2) :
    VR[vd].D[i] = Floating_Point_Compare(VR[vj].D[i], VR[vk].D[i], Double, cond) ? {64{1'b1}} : 0
```

XVFCMP.cond.S:

```
for i in range(8) :
    XR[xd].S[i] = Floating_Point_Compare(XR[xj].S[i], XR[xk].S[i], Single, cond) ? {32{1'b1}} : 0
```

XVFCMP.cond.D:

```
for i in range(4) :
    XR[xd].D[i] = Floating_Point_Compare(XR[xj].D[i], XR[xk].D[i], Double, cond) ? {64{1'b1}} : 0
```

1.2.6.5 [X]VBITSEL.V

Instruction Format: vbitsel.v vd, vj, vk, va xvbitsel.v xd, xj, xk, xa

The [X]VBITSEL.V instruction performs bitwise selection on the contents of vector registers vj/xj and vk/xk based on the contents of vector registers va/xa, writing the result to vector registers vd/xd.

VBITSEL.V:

```
for i in range(128):
    VR[vd].V[i] = VR[va].V[i] ? VR[vk].V[i] : VR[vj].V[i]
```

XVBITSEL.V:

```
for i in range(256):
    XR[xd].V[i] = XR[xa].V[i] ? XR[xk].V[i] : XR[xj].V[i]
```

1.2.6.6 [X]VBITSELI.B

Instruction Format: vbitseli.b vd, vj, ui8 xvbitseli.b xd, xj, ui8

The [X]VBITSELI.B instruction performs bitwise selection on the byte elements at corresponding positions in vector register vj/xj or on the immediate value ui8, based on the contents of each byte element in vector register vd/xd. The result is written to the corresponding byte elements in vector register vd/xd.

VBITSELI.B:

```
for i in range(16) :
    VR[vd].B[i] = (~VR[vd].B[i] & VR[vj].B[i]) | (VR[vd].B[i] & ui8)
```

XVBITSELI.B:

```
for i in range(32):
    XR[xd].B[i] = (~XR[xd].B[i] & XR[xj].B[i]) | (XR[xd].B[i] & ui8)
```

1.2.6.7 [X]VSET{EQZ/NEZ}.V, [X]VSETANYEQZ.{B/H/W/D}, [X]VSETALLNEZ.{B/H/W/D}

Instruction Format:

vseteqz.v	cd, vj	xvseteqz.v	cd, xj
vsetnez.v	cd, vj	xvsetnez.v	cd, xj
vsetanyeqz.b	cd, vj	xvsetanyeqz.b	cd, xj
vsetanyeqz.h	cd, vj	xvsetanyeqz.h	cd, xj
vsetanyeqz.w	cd, vj	xvsetanyeqz.w	cd, xj
vsetanyeqz.d	cd, vj	xvsetanyeqz.d	cd, xj
vsetallnez.b	cd, vj	xvsetallnez.b	cd, xj
vsetallnez.h	cd, vj	xvsetallnez.h	cd, xj
vsetallnez.w	cd, vj	xvsetallnez.w	cd, xj
vsetallnez.d	cd, vj	xvsetallnez.d	cd, xj

The [X]VSETEQZ.V instruction checks whether the value in vector register vj/xj is 0, and writes the result to condition register cd.

VSETEQZ.V:

```
CFR[cd] = (VR[vj] == 0)
```

XVSETEQZ.V:

```
CFR[cd] = (XR[xj] == 0)
```

The [X]VSETNEZ.V checks whether the value in the vector register vj/xj is non-zero. The result is written to the condition register cd.

VSETNEZ.V:

```
CFR[cd] = (VR[vj] != 0)
```

XVSETNEZ.V:

```
CFR[cd] = (XR[xj] != 0)
```

The [X]VSETANYEQZ.{B/H/W/D} instruction checks whether any byte/half-word/word/double-word elements in vector register vj/xj are zero. If true, the condition register cd is set to 1, or it is set to 0.

VSETANYEQZ.B:

```
bool_res = 0
for i in range(16):
        bool_res |= (VR[vj].B[i] == 0)
CFR[cd] = bool_res
```

VSETANYEQZ.H:

```
bool_res = 0
for i in range(8):
        bool_res |= (VR[vj].H[i] == 0)
CFR[cd] = bool_res
```

VSETANYEQZ.W:

```
bool_res = 0
for i in range(4):
        bool_res |= (VR[vj].W[i] == 0)
CFR[cd] = bool_res
```

VSETANYEQZ.D:

```
bool_res = 0
for i in range(2):
        bool_res |= (VR[vj].D[i] == 0)
CFR[cd] = bool_res
```

XVSETANYEQZ.B:

```
bool_res = 0
for i in range(32):
        bool_res |= (XR[xj].B[i] == 0)
CFR[cd] = bool_res
```

XVSETANYEQZ.H:

```
bool_res = 0
for i in range(16):
        bool_res |= (XR[xj].H[i] == 0)
CFR[cd] = bool_res
```

XVSETANYEQZ.W:

```
bool_res = 0
for i in range(8):
        bool_res |= (XR[xj].W[i] == 0)
CFR[cd] = bool_res
```

XVSETANYEQZ.D:

```
bool_res = 0
for i in range(4):
        bool_res |= (XR[xj].D[i] == 0)
CFR[cd] = bool_res
```

The [X]VSETALLNEZ.{B/H/W/D} instruction checks whether all bytes/half-words/words/double-words in vector register vj/xj are non-zero. If true, the condition register cd is set to 1, or it is set to 0.

VSETALLNEZ.B:

```
bool_res = 1
for i in range(16):
        bool_res &= (VR[vj].B[i] != 0)
CFR[cd] = bool_res
```

VSETALLNEZ.H:

```
bool_res = 1
for i in range(8):
        bool_res &= (VR[vj].H[i] != 0)
CFR[cd] = bool_res
```

VSETALLNEZ.W:

```
bool_res = 1
for i in range(4):
        bool_res &= (VR[vj].W[i] != 0)
CFR[cd] = bool_res
```

VSETALLNEZ.D:

```
bool_res = 1
for i in range(2):
        bool_res &= (VR[vj].D[i] != 0)
CFR[cd] = bool_res
```

XVSETALLNEZ.B:

```
bool_res = 1
for i in range(32):
        bool_res &= (XR[xj].B[i] != 0)
CFR[cd] = bool_res
```

XVSETALLNEZ.H:

```
bool_res = 1
for i in range(16):
        bool_res &= (XR[xj].H[i] != 0)
CFR[cd] = bool_res
```

XVSETALLNEZ.W:

```
bool_res = 1
for i in range(8):
        bool_res &= (XR[xj].W[i] != 0)
CFR[cd] = bool_res
```

XVSETALLNEZ.D:

```
bool_res = 1
for i in range(4):
        bool_res &= (XR[xj].D[i] != 0)
CFR[cd] = bool_res
```

1.2.7 Vector Shuffle and Mix Instructions

1.2.7.1 VINSGR2VR.{B/H/W/D}, XVINSGR2VR.{W/D}, VPICKVE2GR.{B/H/W/D}[U], XVPICKVE2GR.{W/D}[U]

Instruction Format:	vinsgr2vr.b	vd, rj, ui4		
	vinsgr2vr.h	vd, rj, ui3		
	vinsgr2vr.w	vd, rj, ui2	xvinsgr2vr.w	xd, rj, ui3
	vinsgr2vr.d	vd, rj, ui1	xvinsgr2vr.d	xd, rj, ui2
	vpickve2gr.b	rd, vj, ui4		
	vpickve2gr.h	rd, vj, ui3		
	vpickve2gr.w	rd, vj, ui2	xvpickve2gr.w	rd, xj, ui3
	vpickve2gr.d	rd, vj, ui1	xvpickve2gr.d	rd, xj, ui2
	vpickve2gr.bu	rd, vj, ui4		
	vpickve2gr.hu	rd, vj, ui3		
	vpickve2gr.wu	rd, vj, ui2	xvpickve2gr.wu	rd, xj, ui3
	vpickve2gr.du	rd, vj, ui1	xvpickve2gr.du	rd, xj, ui2

The VINSGR2VR.{B/H/W/D} instruction writes the lowest byte/half-word/word/double-word element of general-purpose register rj into vector register vd indexed by the immediate ui4/ui3/ui2/ui1. The remaining elements of the vector register vd remain unchanged.

VINSGR2VR.B:

VR[vd].B[ui4] = GR[rj][7:0]

VINSGR2VR.H:

VR[vd].H[ui3] = GR[rj][15:0]

VINSGR2VR.W:

VR[vd].W[ui2] = GR[rj][31:0]

VINSGR2VR.D:

VR[vd].D[ui1] = GR[rj][63:0]

The XVINSGR2VR.{W/D} instruction writes the lowest word/double-word element of general-purpose register rj into vector register xd indexed by the immediate values ui3/ui2. The remaining elements of the vector register xd remain unchanged.

XVINSGR2VR.W:

XR[xd].W[ui3] = GR[rj][31:0]

XVINSGR2VR.D:

XR[xd].D[ui2] = GR[rj][63:0]

The VPICKVE2GR.{B/H/W/D} instruction writes the signed integer byte/half-word/word/double-word elements indexed by the immediate value ui4/ui3/ui2/ui1 in vector register vj, after sign-extending, into general-purpose register rd.

VPICKVE2GR.B:

GR[rd] = SignExtend(VR[vj].B[ui4])

VPICKVE2GR.B:

GR[rd] = SignExtend(VR[vj].H[ui3])

VPICKVE2GR.W:

GR[rd] = SignExtend(VR[vj].W[ui2])

VPICKVE2GR.D:

GR[rd] = SignExtend(VR[vj].D[ui1])

The VPICKVE2GR.{B/H/W/D}U instruction writes the unsigned integer byte/half-word/word/double-word elements indexed by the immediate value ui4/ui3/ui2/ui1 in vector register vj, after zero-extending, to general-purpose register rd.

VPICKVE2GR.BU:

GR[rd] = ZeroExtend(VR[vj].B[ui4])

VPICKVE2GR.HU:

GR[rd] = ZeroExtend(VR[vj].H[ui3])

VPICKVE2GR.WU:

GR[rd] = ZeroExtend(VR[vj].W[ui2])

VPICKVE2GR.DU:

GR[rd] = ZeroExtend(VR[vj].D[ui1])

The XVPICKVE2GR.{W/D} instruction writes the signed integer or double-word elements in vector register xj, indexed by the immediate operand ui3/ui2, to general-purpose register rd after signed extension.

XVPICKVE2GR.W:

GR[rd] = SignExtend(XR[xj].W[ui3])

XVPICKVE2GR.D:

GR[rd] = SignExtend(XR[xj].D[ui2])

The XVPICKVE2GR.{W/D}U The instruction writes the unsigned integer/double-word elements in vector register xj, indexed by immediate ui3/ui2, after zero-extending, into general-purpose register rd.

XVPICKVE2GR.WU:

GR[rd] = ZeroExtend(XR[xj].W[ui3])

XVPICKVE2GR.DU:

GR[rd] = ZeroExtend(XR[xj].D[ui2])

1.2.7.2 [X]VREPLGR2VR.{B/H/W/D}

Instruction Format:	vreplgr2vr.b	vd, rj	xvreplgr2vr.b	xd, rj
	vreplgr2vr.h	vd, rj	xvreplgr2vr.h	xd, rj
	vreplgr2vr.w	vd, rj	xvreplgr2vr.w	xd, rj
	vreplgr2vr.d	vd, rj	xvreplgr2vr.d	xd, rj

The [X]VREPLGR2VR.{B/H/W/D} instruction copies the contents of general-purpose register rj [7:0]/[15:0]/[31:0]/[63:0] bits to the corresponding bytes/half-words/words/double-word elements in vector register vd/xd.

VREPLGR2VR.B:

```
for i in range(16) :
    VR[vd].B[i] = GR[rj][7:0]
```

VREPLGR2VR.H:

```
for i in range(8) :
    VR[vd].H[i] = GR[rj][15:0]
```

VREPLGR2VR.W:

```
for i in range(4):
    VR[vd].W[i] = GR[rj][31:0]
```

VREPLGR2VR.D:

```
for i in range(2):
    VR[vd].D[i] = GR[rj][63:0]
```

XVREPLGR2VR.B:

```
for i in range(32):
    XR[xd].B[i] = GR[rj][7:0]
```

XVREPLGR2VR.H:

```
for i in range(16):
    XR[xd].H[i] = GR[rj][15:0]
```

XVREPLGR2VR.W:

```
for i in range(8):
    XR[xd].W[i] = GR[rj][31:0]
```

XVREPLGR2VR.D:

```
for i in range(4):
    XR[xd].D[i] = GR[rj][63:0]
```

1.2.7.3 [X]VREPLVE.{B/H/W/D}, VREPLVEI.{B/H/W/D}, XVREPL128VEI.{B/H/W/D}

Instruction Format:				
	vreplve.b	vd, vj, rk	xvreplve.b	xd, xj, rk
	vreplve.h	vd, vj, rk	xvreplve.h	xd, xj, rk
	vreplve.w	vd, vj, rk	xvreplve.w	xd, xj, rk
	vreplve.d	vd, vj, rk	xvreplve.d	xd, xj, rk
	vreplvei.b	vd, vj, ui4	xvrepl128vei.b	xd, xj, ui4
	vreplvei.h	vd, vj, ui3	xvrepl128vei.h	xd, xj, ui3
	vreplvei.w	vd, vj, ui2	xvrepl128vei.w	xd, xj, ui2
	vreplvei.d	vd, vj, ui1	xvrepl128vei.d	xd, xj, ui1

The [X]VREPLVE.{B/H/W/D} instruction copies a specified byte/half-word/word/double-word element from every 128 bits in vector register vj/xj to the corresponding byte/half-word/word/double-word elements in the specified 128 bits in vector register vd/xd. The index value for the specified element is derived from the general-purpose register rk.

VREPLVE.B:

```
t = GR[rk] % 16
for i in range(16) :
    VR[vd].B[i] = VR[vj].B[t]
```

VREPLVE.H:

```
t = GR[rk] % 8
for i in range(8):
    VR[vd].H[i] = VR[vj].H[t]
```

VREPLVE.W:

```
t = GR[rk] % 4
for i in range(4) :
    VR[vd].W[i] = VR[vj].W[t]
```

VREPLVE.D:

```
t = GR[rk] % 2
for i in range(2) :
    VR[vd].D[i] = VR[vj].D[t]
```

XVREPLVE.B:

```
t = GR[rk] % 16
for i in range(16) :
    XR[xd].B[i] = XR[xj].B[t]
    XR[xd].B[i+16] = XR[xj].B[t+16]
```

XVREPLVE.H:

```
t = GR[rk] % 8
for i in range(8):
    XR[xd].H[i] = XR[xj].H[t]
    XR[xd].H[i+8] = XR[xj].H[t+8]
```

XVREPLVE.W:

```
t = GR[rk] % 4
for i in range(4):
    XR[xd].W[i] = XR[xj].W[t]
    XR[xd].W[i+4] = XR[xj].W[t+4]
```

XVREPLVE.D:

```
t = GR[rk] % 2
for i in range(2):
    XR[xd].D[i] = XR[xj].D[t]
    XR[xd].D[i+2] = XR[xj].D[t+2]
```

The VREPLVEI.{B/H/W/D} instruction copies a specified byte/half-word/word/double-word element from vector register vj to the corresponding byte/half-word/word/double-word elements in vector register vd/xd. The index value for the specified element is derived from the immediate value ui4/ui3/ui2/ui1.

VREPLVEI.B:

```
t = ui4
for i in range(16) :
    VR[vd].B[i] = VR[vj].B[t]
```

VREPLVEI.H:

```
t = ui3
for i in range(8) :
    VR[vd].H[i] = VR[vj].H[t]
```

VREPLVEI.W:

```
t = ui2
for i in range(4):
    VR[vd].W[i] = VR[vj].W[t]
```

VREPLVEI.D:

```
t = ui1
for i in range(2) :
    VR[vd].D[i] = VR[vj].D[t]
```

The XVREPL128VEI.{B/H/W/D} instruction copies the specified byte/half-word/word/double-word elements in every 128 bits of vector register xj to the corresponding byte/half-word/word/double-word elements in 128 bits of vector register xd. The index value for the specified element is derived from an immediate value ui4/ui3/ui2/ui1.

XVREPL128VEI.B:

```
t = ui4
for i in range(16) :
    XR[xd].B[i] = XR[xj].B[t]
    XR[xd].B[i+16] = XR[xj].B[t+16]
```

XVREPL128VEI.H:

```
t = ui3
for i in range(8):
    XR[xd].H[i] = XR[xj].H[t]
    XR[xd].H[i+8] = XR[xj].H[t+8]
```

XVREPL128VEI.W:

```
t = ui2
for i in range(4):
    XR[xd].W[i] = XR[xj].W[t]
    XR[xd].W[i+4] = XR[xj].W[t+4]
```

XVREPL128VEI.D:

```
t = ui1
for i in range(2):
    XR[xd].D[i] = XR[xj].D[t]
    XR[xd].D[i+2] = XR[xj].D[t+2]
```

1.2.7.4 XVREPLVE0.{B/H/W/D/Q}

Instruction Format:	xvreplve0.b	xd, xj
	xvreplve0.h	xd, xj
	xvreplve0.w	xd, xj
	xvreplve0.d	xd, xj
	xvreplve0.q	xd, xj

The XVREPLVE0.{B/H/W/D/Q} instruction copies the lowest byte/half-word/word/double-word/quad-word element from vector register xj into the corresponding byte/half-word/word/double-word/quad-word element of vector register xd.

XVREPLVE0.B:

```
for i in range(16) :
    XR[vd].B[i] = XR[vj].B[0]
```

XVREPLVE0.H:

```
for i in range(8) :
    XR[vd].H[i] = XR[vj].H[0]
```

XVREPLVE0.W:

```
for i in range(4):
    XR[vd].W[i] = XR[vj].W[0]
```

XVREPLVE0.D:

```
for i in range(2):
    XR[vd].D[i] = XR[vj].D[0]
```

XVREPLVE0.Q:

```
for i in range(1):
    XR[vd].Q[i] = XR[vj].Q[0]
```

1.2.7.5 XVINSVE0.{W/D}

Instruction Format:	xvinsve0.w	xd, xj, ui3
	xvinsve0.d	xd, xj, ui2

The XVINSVE0.{W/D} instruction writes the 0th word/double-word element from vector register xj into the index of ui3/ui2 word/double-word element of vector register xd. The remaining contents of the vector register xd are unchanged.

XVINSVE0.W:

```
XR[xd].W[ui3] = XR[xj].W[0]
```

XVINSVE0.D:

```
XR[xd].D[ui2] = XR[xj].D[0]
```

1.2.7.6 XVPICKVE.{W/D}

Instruction Format:	xvpickve.w	xd, xj, ui3
	xvpickve.d	xd, xj, ui2

The XVPICKVE.{W/D} instruction writes the index of ui3/ui2 word/double-word element from vector register xj into the 0th word/double-word element of vector register xd. The remaining contents of the vector register xd are set to 0.

XVPICKVE.W:

```
XR[xd].W[0] = XR[xj].W[ui3]
XR[xd][255:32] = 0
```

XVPICKVE.D:

```
XR[xd].D[0] = XR[xj].D[ui2]
XR[xd][255:64] = 0
```

1.2.7.7 [X]VBSLL.V, [X]VBSRL.V

Instruction Format:	vbsll.v	vd, vj, ui5	xvbsll.v	xd, xj, ui5
	vbsrl.v	vd, vj, ui5	xvbsrl.v	xd, xj, ui5

The [X]VBSLL.V instruction shifts the data in vector register vj/xj left by ui5 bytes per 128 bits, padding the low bits with zeros, and writes the result to the corresponding 128 bits in vector register vd/xd.

VBSLL.V:

```
idx = ui5[3:0] * 8
VR[vd] = {VR[vj][127-idx:0], {idx{8'b0}}}
```

XVBSLL.V:

```
idx = ui5[3:0] * 8
XR[xd][127:0]   = {XR[xj][127-idx:0], {idx{8'b0}}}
XR[xd][255:128] = {XR[xj][255-idx:128], {idx{8'b0}}}
```

The [X]VBSRL.V instruction shifts the data in vector register vj/xj right by ui5 bytes per 128 bits, padding the high bits with zeros. The result is written to the corresponding 128 bits in vector register vd/xd.

VBSRL.V:

```
idx = ui5[3:0] * 8
VR[vd] = {{idx{8'b0}}, VR[vj][127:idx]}
```

XVBSRL.V:

```
idx = ui5[3:0] * 8
XR[xd][127:0]   = {{idx{8'b0}}, XR[xj][127:idx]}
XR[xd][255:128] = {{idx{8'b0}}, XR[xj][255:idx]}
```

1.2.7.8 [X]VPACK{EV/OD}.{B/H/W/D}

Instruction Format:	vpackev.b	vd, vj, vk	xvpackev.b	xd, xj, xk
	vpackev.h	vd, vj, vk	xvpackev.h	xd, xj, xk
	vpackev.w	vd, vj, vk	xvpackev.w	xd, xj, xk
	vpackev.d	vd, vj, vk	xvpackev.d	xd, xj, xk
	vpackod.b	vd, vj, vk	xvpackod.b	xd, xj, xk
	vpackod.h	vd, vj, vk	xvpackod.h	xd, xj, xk
	vpackod.w	vd, vj, vk	xvpackod.w	xd, xj, xk
	vpackod.d	vd, vj, vk	xvpackod.d	xd, xj, xk

The [X]VPACKEV.{B/H/W/D} instruction packs the byte/half-word/word/double-word elements at each even position in vector register vj/xj with the corresponding byte/half-word/word/double-word elements at the even positions in vector register vk/xk. These pairs are then written sequentially into the vector register vd/xd.

VPACKEV.B:

```
for i in range(8) :
        VR[vd].B[2i+1] = VR[vj].B[2i]
        VR[vd].B[2i+0] = VR[vk].B[2i]
```

VPACKEV.H:

```
for i in range(4):
    VR[vd].H[2i+1] = VR[vj].H[2i]
    VR[vd].H[2i+0] = VR[vk].H[2i]
```

VPACKEV.W:

```
for i in range(2) :
    VR[vd].W[2i+1] = VR[vj].W[2i]
    VR[vd].W[2i+0] = VR[vk].W[2i]
```

VPACKEV.D:

```
for i in range(1):
    VR[vd].D[2i+1] = VR[vj].D[2i]
    VR[vd].D[2i+0] = VR[vk].D[2i]
```

XVPACKEV.B:

```
for i in range(16) :
    XR[xd].B[2i+1] = XR[xj].B[2i]
    XR[xd].B[2i+0] = XR[xk].B[2i]
```

XVPACKEV.H:

```
for i in range(8):
    XR[xd].H[2i+1] = XR[xj].H[2i]
    XR[xd].H[2i+0] = XR[xk].H[2i]
```

XVPACKEV.W:

```
for i in range(4):
    XR[xd].W[2i+1] = XR[xj].W[2i]
    XR[xd].W[2i+0] = XR[xk].W[2i]
```

XVPACKEV.D:

```
for i in range(2) :
    XR[xd].D[2i+1] = XR[xj].D[2i]
    XR[xd].D[2i+0] = XR[xk].D[2i]
```

The [X]VPACKOD.{B/H/W/D} instruction packs the byte/half-word/word/double-word elements at each odd position in vector register vj/xj with the corresponding byte/half-word/word/double-word elements at the odd positions in vector register vk/xk. These pairs are then written sequentially into the vector register vd/xd.

VPACKOD.B:

```
for i in range(8):
    VR[vd].B[2i+1] = VR[vj].B[2i+1]
    VR[vd].B[2i+0] = VR[vk].B[2i+1]
```

VPACKOD.H:

```
for i in range(4):
    VR[vd].H[2i+1] = VR[vj].H[2i+1]
    VR[vd].H[2i+0] = VR[vk].H[2i+1]
```

VPACKOD.W:

```
for i in range(2):
    VR[vd].W[2i+1] = VR[vj].W[2i+1]
    VR[vd].W[2i+0] = VR[vk].W[2i+1]
```

VPACKOD.D:

```
for i in range(1):
    VR[vd].D[2i+1] = VR[vj].D[2i+1]
    VR[vd].D[2i+0] = VR[vk].D[2i+1]
```

XVPACKOD.B:

```
for i in range(16) :
    XR[xd].B[2i+1] = XR[xj].B[2i+1]
    XR[xd].B[2i+0] = XR[xk].B[2i+1]
```

XVPACKOD.H:

```
for i in range(8) :
    XR[xd].H[2i+1] = XR[xj].H[2i+1]
    XR[xd].H[2i+0] = XR[xk].H[2i+1]
```

XVPACKOD.W:

```
for i in range(4):
    XR[xd].W[2i+1] = XR[xj].W[2i+1]
    XR[xd].W[2i+0] = XR[xk].W[2i+1]
```

XVPACKOD.D:

```
for i in range(2):
    XR[xd].D[2i+1] = XR[xj].D[2i+1]
    XR[xd].D[2i+0] = XR[xk].D[2i+1]
```

1.2.7.9 [X]VPICK{EV/OD}.{B/H/W/D}

Instruction Format:				
	vpickev.b	vd, vj, vk	xvpickev.b	xd, xj, xk
	vpickev.h	vd, vj, vk	xvpickev.h	xd, xj, xk
	vpickev.w	vd, vj, vk	xvpickev.w	xd, xj, xk
	vpickev.d	vd, vj, vk	xvpickev.d	xd, xj, xk
	vpickod.b	vd, vj, vk	xvpickod.b	xd, xj, xk
	vpickod.h	vd, vj, vk	xvpickod.h	xd, xj, xk
	vpickod.w	vd, vj, vk	xvpickod.w	xd, xj, xk
	vpickod.d	vd, vj, vk	xvpickod.d	xd, xj, xk

The [X]VPICKEV.{B/H/W/D} instruction writes the byte/half-word/word/double-word elements at even positions in every 128 bits of vector registers vj/xj into the upper half of the corresponding 128 bits in vector registers vd/xd. It writes the byte/half-word/word/double-word elements at even positions in every 128 bits of vector registers vk/xk into the lower half of the corresponding 128 bits in vector registers vd/xd.

VPICKEV.B:

```
for i in range(8):
    VR[vd].B[i+8] = VR[vj].B[2i]
    VR[vd].B[i+0] = VR[vk].B[2i]
```

VPICKEV.H:

```
for i in range(4):
    VR[vd].H[i+4] = VR[vj].H[2i]
    VR[vd].H[i+0] = VR[vk].H[2i]
```

VPICKEV.W:

```
for i in range(2):
    VR[vd].W[i+2] = VR[vj].W[2i]
    VR[vd].W[i+0] = VR[vk].W[2i]
```

VPICKEV.D:

```
for i in range(1):
    VR[vd].D[i+1] = VR[vj].D[2i]
    VR[vd].D[i+0] = VR[vk].D[2i]
```

XVPICKEV.B:

```
for i in range(8):
    VR[vd].B[i+8] = VR[vj].B[2i]
    VR[vd].B[i+0] = VR[vk].B[2i]
    VR[vd].B[i+24] = VR[vj].B[2i+16]
    VR[vd].B[i+16] = VR[vk].B[2i+16]
```

XVPICKEV.H:

```
for i in range(4) :
    VR[vd].H[i+4] = VR[vj].H[2i]
    VR[vd].H[i+0] = VR[vk].H[2i]
    VR[vd].H[i+12] = VR[vj].H[2i+8]
    VR[vd].H[i+8] = VR[vk].H[2i+8]
```

XVPICKEV.W:

```
for i in range(2) :
    VR[vd].W[i+2] = VR[vj].W[2i]
    VR[vd].W[i+0] = VR[vk].W[2i]
    VR[vd].W[i+6] = VR[vj].W[2i+4]
    VR[vd].W[i+4] = VR[vk].W[2i+4]
```

XVPICKEV.D:

```
for i in range(1):
    VR[vd].D[i+1] = VR[vj].D[2i]
    VR[vd].D[i+0] = VR[vk].D[2i]
    VR[vd].D[i+3] = VR[vj].D[2i+2]
    VR[vd].D[i+2] = VR[vk].D[2i+2]
```

The [X]VPICKOD.{B/H/W/D} instruction writes the byte/half-word/word/double-word elements at odd positions in every 128 bits of vector register vj/xj into the upper half of the corresponding 128 bits in vector register vd/xd. It writes the byte/half-word/word/double-word elements at odd positions in every 128 bits of vector register vk/xk into the lower half of the corresponding 128 bits in vector register vd/xd.

VPICKOD.B:

```
for i in range(8):
    VR[vd].B[i+8] = VR[vj].B[2i+1]
    VR[vd].B[i+0] = VR[vk].B[2i+1]
```

VPICKOD.H:

```
for i in range(4):
    VR[vd].H[i+4] = VR[vj].H[2i+1]
    VR[vd].H[i+0] = VR[vk].H[2i+1]
```

VPICKOD.W:

```
for i in range(2):
    VR[vd].W[i+2] = VR[vj].W[2i+1]
    VR[vd].W[i+0] = VR[vk].W[2i+1]
```

VPICKOD.D:

```
for i in range(1):
    VR[vd].D[i+1] = VR[vj].D[2i+1]
    VR[vd].D[i+0] = VR[vk].D[2i+1]
```

XVPICKOD.B:

```
for i in range(8):
    VR[vd].B[i+8] = VR[vj].B[2i+1]
    VR[vd].B[i+0] = VR[vk].B[2i+1]
    VR[vd].B[i+24] = VR[vj].B[2i+17]
    VR[vd].B[i+16] = VR[vk].B[2i+17]
```

XVPICKOD.H:

```
for i in range(4):
    VR[vd].H[i+4] = VR[vj].H[2i+1]
    VR[vd].H[i+0] = VR[vk].H[2i+1]
    VR[vd].H[i+12] = VR[vj].H[2i+9]
    VR[vd].H[i+8] = VR[vk].H[2i+9]
```

XVPICKOD.W:

```
for i in range(2) :
    VR[vd].W[i+2] = VR[vj].W[2i+1]
    VR[vd].W[i+0] = VR[vk].W[2i+1]
    VR[vd].W[i+6] = VR[vj].W[2i+5]
    VR[vd].W[i+4] = VR[vk].W[2i+5]
```

XVPICKOD.D:

```
for i in range(1):
    VR[vd].D[i+1] = VR[vj].D[2i+1]
    VR[vd].D[i+0] = VR[vk].D[2i+1]
    VR[vd].D[i+3] = VR[vj].D[2i+3]
    VR[vd].D[i+2] = VR[vk].D[2i+3]
```

1.2.7.10 [X]VILV{L/H}.{B/H/W/D}

Instruction Format:				
	vilvl.b	vd, vj, vk	xvilvl.b	xd, xj, xk
	vilvl.h	vd, vj, vk	xvilvl.h	xd, xj, xk
	vilvl.w	vd, vj, vk	xvilvl.w	xd, xj, xk
	vilvl.d	vd, vj, vk	xvilvl.d	xd, xj, xk
	vilvh.b	vd, vj, vk	xvilvh.b	xd, xj, xk
	vilvh.h	vd, vj, vk	xvilvh.h	xd, xj, xk
	vilvh.w	vd, vj, vk	xvilvh.w	xd, xj, xk
	vilvh.d	vd, vj, vk	xvilvh.d	xd, xj, xk

The [X]VILVL.{B/H/W/D} writes each byte/half-word/word/double-word elements from the lower half of every 128 bits in vector register vk/xk and the corresponding byte/half-word/word/double-word elements at the corresponding position in vector register vj/xj into the corresponding 128 bits of vector register vd/xd in alternating sequence.

VILVL.B:

```
for i in range(8) :
    VR[vd].B[2i+1] = VR[vj].B[i]
    VR[vd].B[2i+0] = VR[vk].B[i]
```

VILVL.H:

```
for i in range(4):
    VR[vd].H[2i+1] = VR[vj].H[i]
    VR[vd].H[2i+0] = VR[vk].H[i]
```

VILVL.W:

```
for i in range(2) :
    VR[vd].W[2i+1] = VR[vj].W[i]
    VR[vd].W[2i+0] = VR[vk].W[i]
```

VILVL.D:

```
for i in range(1):
    VR[vd].D[2i+1] = VR[vj].D[i]
    VR[vd].D[2i+0] = VR[vk].D[i]
```

XVILVL.B:

```
for i in range(8) :
    VR[vd].B[2i+1] = VR[vj].B[i]
    VR[vd].B[2i+0] = VR[vk].B[i]
    VR[vd].B[2i+17] = VR[vj].B[i+16]
    VR[vd].B[2i+16] = VR[vk].B[i+16]
```

XVILVL.H:

```
for i in range(4) :
    VR[vd].H[2i+1] = VR[vj].H[i]
    VR[vd].H[2i+0] = VR[vk].H[i]
    VR[vd].H[2i+9] = VR[vj].H[i+8]
    VR[vd].H[2i+8] = VR[vk].H[i+8]
```

XVILVL.W:

```
for i in range(2):
    VR[vd].W[2i+1] = VR[vj].W[i]
    VR[vd].W[2i+0] = VR[vk].W[i]
    VR[vd].W[2i+5] = VR[vj].W[i+4]
    VR[vd].W[2i+4] = VR[vk].W[i+4]
```

XVILVL.D:

```
for i in range(1):
    VR[vd].D[2i+1] = VR[vj].D[i]
    VR[vd].D[2i+0] = VR[vk].D[i]
    VR[vd].D[2i+3] = VR[vj].D[i+2]
    VR[vd].D[2i+2] = VR[vk].D[i+2]
```

The [X]VILVH.{B/H/W/D} writes each byte/half-word/word/double-word elements from the upper half of every 128 bits in vector register vk/xk and the corresponding byte/half-word/word/double-word elements at the corresponding position in vector register vj/xj into the corresponding 128 bits of vector register vd/xd in alternating sequence.

VILVH.B:

```
for i in range(8) :
    VR[vd].B[2i+1] = VR[vj].B[i+8]
    VR[vd].B[2i+0] = VR[vk].B[i+8]
```

VILVH.H:

```
for i in range(4):
    VR[vd].H[2i+1] = VR[vj].H[i+4]
    VR[vd].H[2i+0] = VR[vk].H[i+4]
```

VILVH.W:

```
for i in range(2) :
    VR[vd].W[2i+1] = VR[vj].W[i+2]
    VR[vd].W[2i+0] = VR[vk].W[i+2]
```

VILVH.D:

```
for i in range(1):
    VR[vd].D[2i+1] = VR[vj].D[i+1]
    VR[vd].D[2i+0] = VR[vk].D[i+1]
```

XVILVH.B:

```
for i in range(8) :
    VR[vd].B[2i+1] = VR[vj].B[i+8]
    VR[vd].B[2i+0] = VR[vk].B[i+8]
    VR[vd].B[2i+17] = VR[vj].B[i+24]
    VR[vd].B[2i+16] = VR[vk].B[i+24]
```

XVILVH.H:

```
for i in range(4) :
    VR[vd].H[2i+1] = VR[vj].H[i+4]
    VR[vd].H[2i+0] = VR[vk].H[i+4]
    VR[vd].H[2i+9] = VR[vj].H[i+12]
    VR[vd].H[2i+8] = VR[vk].H[i+12]
```

XVILVH.W:

```
for i in range(2) :
    VR[vd].W[2i+1] = VR[vj].W[i+2]
    VR[vd].W[2i+0] = VR[vk].W[i+2]
    VR[vd].W[2i+5] = VR[vj].W[i+6]
    VR[vd].W[2i+4] = VR[vk].W[i+6]
```

XVILVH.D:

```
for i in range(1) :
    VR[vd].D[2i+1] = VR[vj].D[i+1]
    VR[vd].D[2i+0] = VR[vk].D[i+1]
    VR[vd].D[2i+3] = VR[vj].D[i+3]
    VR[vd].D[2i+2] = VR[vk].D[i+3]
```

1.2.7.11 [X]VSHUF.B

Instruction Format: vshuf.b vd, vj, vk, va xvshuf.b xd, xj, xk, xa

The [X]VSHUF.B instruction concatenates every 128 bits from vector register vj/xj with the corresponding 128 bits from vector register vk/xk to form a 256-bit intermediate vector. It uses the byte element index values stored in the corresponding 128 bits of vector register va/xa to select the corresponding byte elements from the intermediate vector, and writes them to the byte element positions in vector registers vd/xd that correspond to the index values. The result is undefined when the highest 3 bits of the byte storing the index value are non-zero.

VSHUF.B:

```
vec0 = {VR[vj], VR[vk]}
for i in range(16) :
    t0 = VR[va].B[i][4:0]
    VR[vd].B[i] = vec0.B[t0]
```

XVSHUF.B:

```
vec0 = {XR[xj][127:0], XR[xk][127:0]}
vec1 = {XR[xj][255:128], XR[xk][255:128]}
for i in range(16):
    t0 = XR[xa].B[i][4:0]
    XR[xd].B[i] = vec0.B[t0]
    t1 = XR[xa].B[i+16][4:0]
    XR[xd].B[i+16] = vec1.B[t1]
```

1.2.7.12 [X]VSHUF.{H/W/D}

Instruction Format:	vshuf.h	vd, vj, vk	xvshuf.h	xd, xj, xk
	vshuf.w	vd, vj, vk	xvshuf.w	xd, xj, xk
	vshuf.d	vd, vj, vk	xvshuf.d	xd, xj, xk

The [X]VSHUF.{H/W/D} instruction concatenates every 128 bits from vector register vj/xj with the corresponding 128 bits from vector register vk/xk to form a 256-bit intermediate vector. It uses the half-word/word/double-word element index values stored in the corresponding 128 bits of vector register vd/xd to select the corresponding half-word/word/double-word elements from the intermediate vector, and writes them into the half-word/word/double-word element positions in vector registers vd/xd that correspond to the index values.

VSHUF.H:

```
vec0 = {VR[vj], VR[vk]}
for i in range(8) :
    t0 = VR[vd].H[i][3:0]
    VR[vd].H[i] = vec0.H[t0]
```

VSHUF.W:

```
vec0 = {VR[vj], VR[vk]}
for i in range(4):
    t0 = VR[vd].W[i][2:0]
    VR[vd].W[i] = vec0.W[t0]
```

VSHUF.D:

```
vec0 = {VR[vj], VR[vk]}
for i in range(2) :
    t0 = VR[vd].D[i][1:0]
    VR[vd].D[i] = vec0.D[t0]
```

XVSHUF.H:

```
vec0 = {XR[xj][127:0], XR[xk][127:0]}
vec1 = {XR[xj][255:128], XR[xk][255:128]}
for i in range(8):
    t0 = XR[xd].H[i][3:0]
    XR[xd].H[i] = vec0.H[t0]
    t1 = XR[xd].H[i+8][3:0]
    XR[xd].H[i+16] = vec1.H[t1]
```

XVSHUF.W:

```
vec0 = {XR[xj][127:0], XR[xk][127:0]}
vec1 = {XR[xj][255:128], XR[xk][255:128]}
for i in range(4):
    t0 = XR[xd].W[i][2:0]
    XR[xd].W[i] = vec0.W[t0]
    t1 = XR[xd].W[i+4][2:0]
    XR[xd].W[i+16] = vec1.W[t1]
```

XVSHUF.D:

```
vec0 = {XR[xj][127:0], XR[xk][127:0]}
vec1 = {XR[xj][255:128], XR[xk][255:128]}
for i in range(2):
    t0 = XR[xd].D[i][1:0]
    XR[xd].D[i] = vec0.D[t0]
    t1 = XR[xd].D[i+2][1:0]
    XR[xd].D[i+16] = vec1.D[t1]
```

1.2.7.13 XVPERM.W

Instruction Format: xvperm.w xd, xj, xk

The XVPERM.W instruction uses the value of the lowest 3 bits of each word element in vector register xk as an index value. It selects one of the eight word elements from vector register xj and writes it into the corresponding position in vector register xd.

XVPERM.W:

```
for i in range(8) :
    t = XR[xk].W[i] % 8
    XR[xd].W[i] = XR[xj].W[t]
```

1.2.7.14 [X]VSHUF4I.{B/H/W/D}

Instruction Format:				
	vshuf4i.b	vd, vj, ui8	xvshuf4i.b	xd, xj, ui8
	vshuf4i.h	vd, vj, ui8	xvshuf4i.h	xd, xj, ui8
	vshuf4i.w	vd, vj, ui8	xvshuf4i.w	xd, xj, ui8
	vshuf4i.d	vd, vj, ui8	xvshuf4i.d	xd, xj, ui8

The [X]VSHUF4I.{B/H/W} instruction shuffles each group of four consecutive bytes/half-word/word/double-word elements in vector register vj/xj according to the immediate value. The results are written into the corresponding group in the vector register vd/xd.

VSHUF4I.B:

```
for i in range(4):
    vec0 = {VR[vj].B[4i+3], VR[vj].B[4i+2], VR[vj].B[4i+1], VR[vj].B[4i]}
    VR[vd].B[4i+0] = vec0.B[ui8[1:0]]
    VR[vd].B[4i+1] = vec0.B[ui8[3:2]]
    VR[vd].B[4i+2] = vec0.B[ui8[5:4]]
    VR[vd].B[4i+3] = vec0.B[ui8[7:6]]
```

VSHUF4I.H:

```
for i in range(2) :
    vec0 = {VR[vj].H[4i+3], VR[vj].H[4i+2], VR[vj].H[4i+1], VR[vj].H[4i]}
    VR[vd].H[4i+0] = vec0.H[ui8[1:0]]
    VR[vd].H[4i+1] = vec0.H[ui8[3:2]]
    VR[vd].H[4i+2] = vec0.H[ui8[5:4]]
    VR[vd].H[4i+3] = vec0.H[ui8[7:6]]
```

VSHUF4I.W:

```
for i in range(1):
    vec0 = {VR[vj].W[4i+3], VR[vj].W[4i+2], VR[vj].W[4i+1], VR[vj].W[4i]}
    VR[vd].W[4i+0] = vec0.W[ui8[1:0]]
    VR[vd].W[4i+1] = vec0.W[ui8[3:2]]
    VR[vd].W[4i+2] = vec0.W[ui8[5:4]]
    VR[vd].W[4i+3] = vec0.W[ui8[7:6]]
```

XVSHUF4I.B:

```
for i in range(4):
    vec0 = {XR[xj].B[4i+3], XR[xj].B[4i+2], XR[xj].B[4i+1], XR[xj].B[4i]}
    XR[xd].B[4i+0] = vec0.B[ui8[1:0]]
    XR[xd].B[4i+1] = vec0.B[ui8[3:2]]
    XR[xd].B[4i+2] = vec0.B[ui8[5:4]]
    XR[xd].B[4i+3] = vec0.B[ui8[7:6]]
    vec1 = {XR[xj].B[4i+19], XR[xj].B[4i+18], XR[xj].B[4i+17], XR[xj].B[4i+16]}
    XR[xd].B[4i+16] = vec1.B[ui8[1:0]]
    XR[xd].B[4i+17] = vec1.B[ui8[3:2]]
    XR[xd].B[4i+18] = vec1.B[ui8[5:4]]
    XR[xd].B[4i+19] = vec1.B[ui8[7:6]]
```

XVSHUF4I.H:

```
for i in range(2) :
    vec0 = {XR[xj].H[4i+3], XR[xj].H[4i+2], XR[xj].H[4i+1], XR[xj].H[4i]}
    XR[xd].H[4i+0] = vec0.H[ui8[1:0]]
    XR[xd].H[4i+1] = vec0.H[ui8[3:2]]
    XR[xd].H[4i+2] = vec0.H[ui8[5:4]]
    XR[xd].H[4i+3] = vec0.H[ui8[7:6]]
    vec1 = {XR[xj].H[4i+11], XR[xj].H[4i+10], XR[xj].H[4i+9], XR[xj].H[4i+8]}
    XR[xd].H[4i+8] = vec1.H[ui8[1:0]]
    XR[xd].H[4i+9] = vec1.H[ui8[3:2]]
    XR[xd].H[4i+10] = vec1.H[ui8[5:4]]
    XR[xd].H[4i+11] = vec1.H[ui8[7:6]]
```

XVSHUF4I.W:

```
for i in range(1):
    vec0 = {XR[xj].W[4i+3], XR[xj].W[4i+2], XR[xj].W[4i+1], XR[xj].W[4i]}
    XR[xd].W[4i+0] = vec0.W[ui8[1:0]]
    XR[xd].W[4i+1] = vec0.W[ui8[3:2]]
    XR[xd].W[4i+2] = vec0.W[ui8[5:4]]
    XR[xd].W[4i+3] = vec0.W[ui8[7:6]]
    vec1 = {XR[xj].W[4i+7], XR[xj].W[4i+6], XR[xj].W[4i+5], XR[xj].W[4i+4]}
    XR[xd].W[4i+4] = vec1.W[ui8[1:0]]
    XR[xd].W[4i+5] = vec1.W[ui8[3:2]]
    XR[xd].W[4i+6] = vec1.W[ui8[5:4]]
    XR[xd].W[4i+7] = vec1.W[ui8[7:6]]
```

The [X]VSHUF4I.D instruction selects two of four double-word elements from every 128 bits of vector registers vj/xj and vd/xd based on the immediate value. The results are written into the corresponding 128 bits of vector registers vd/xd.

VSHUF4I.D:

```
vec0 = {VR[vj], VR[vd]}
VR[vd].D[0] = vec0.D[ui8[1:0]]
VR[vd].D[1] = vec0.D[ui8[3:2]]
```

XVSHUF4I.D:

```
vec0 = {XR[xj][127:0], XR[xd][127:0]}
XR[xd].D[0] = vec0.D[ui8[1:0]]
XR[xd].D[1] = vec0.D[ui8[3:2]]
vec1 = {XR[xj][255:128], XR[xd][255:128]}
XR[xd].D[2] = vec1.D[ui8[1:0]]
XR[xd].D[3] = vec1.D[ui8[3:2]]
```

1.2.7.15 VPREMI.W, XVPREMI.W, XVPERMI.D, XVPERMI.Q

Instruction Format:

```
vpermi.w    vd, vj, ui8
xvpermi.w   xd, xj, ui8
xvpermi.d   xd, xj, ui8
xvpermi.q   xd, xj, ui8
```

The [X]VPERMI.W instruction uses the values of ui8[1:0] and ui8[3:2] as index values to select one element each from the four word elements in every 128 bits of vector registers vj/xj, writing them respectively into the 0th and 1st word elements corresponding to the 128-bit positions in vector registers vd/xd. It also uses the values of ui8[5:4] and ui8[7:6] as index values to select one element each from the four word elements in every 128 bits of vector registers vd/xd, writing them respectively into the 0th and 1st word elements corresponding to the 128-bit positions in vector registers vd/xd.

VPERMI.W:

```
VR[vd].W[0] = VR[vj].W[ui8[1:0]]
VR[vd].W[1] = VR[vj].W[ui8[3:2]]
VR[vd].W[2] = VR[vd].W[ui8[5:4]]
VR[vd].W[3] = VR[vd].W[ui8[7:6]]
```

XVPERMI.W:

```
XR[xd].W[0] = XR[xj].W[ui8[1:0]]
XR[xd].W[1] = XR[xj].W[ui8[3:2]]
XR[xd].W[2] = XR[xd].W[ui8[5:4]]
XR[xd].W[3] = XR[xd].W[ui8[7:6]]
XR[xd].W[4] = XR[xj].W[ui8[1:0]+4]
XR[xd].W[5] = XR[xj].W[ui8[3:2]+4]
XR[xd].W[6] = XR[xd].W[ui8[5:4]+4]
XR[xd].W[7] = XR[xd].W[ui8[7:6]+4]
```

The XVPERMI.D instruction uses the values ui8[1:0], ui8[3:2], ui8[5:4], and ui8[7:6] as indices to select one of four double-word elements from vector

register xj. The results are sequentially written into the corresponding four double-word elements of vector register xd.

XVPERMI.D:

XR[xd].D[0] = XR[xj].D[ui8[1:0]]
XR[xd].D[1] = XR[xj].D[ui8[3:2]]
XR[xd].D[2] = XR[xj].D[ui8[5:4]]
XR[xd].D[3] = XR[xj].D[ui8[7:6]]

The XVPERMI.Q instruction selects one value each from the four quad-word elements of vector registers xj and xd based on the values of ui8[1:0] and ui8[5:4], writing them into two quad-word elements of vector register xd.

XVPERMI.Q:

vec = {XR[xd], XR[xj]}
XR[xd].Q[0] = vec.Q[ui8[1:0]]
XR[xd].Q[1] = vec.Q[ui8[5:4]]

1.2.7.16 [X]VEXTRINS.{B/H/W/D}

Instruction Format:	vextrins.b	vd, vj, ui8	xvextrins.b	xd, xj, ui8
	vextrins.h	vd, vj, ui8	xvextrins.h	xd, xj, ui8
	vextrins.w	vd, vj, ui8	xvextrins.w	xd, xj, ui8
	vextrins.d	vd, vj, ui8	xvextrins.d	xd, xj, ui8

The [X]VEXTRINS.{B/H/W/D} instruction loads a byte/half-word/word/double-word element from every 128 bits of vector registers vj/xj, indexed by ui8[3:0]/ui8[2:0]/ui8[1:0]/ui8[0], into the corresponding byte/half-word/word/double-word position in every 128 bits of vector registers vd/xd, indexed by ui8[7:4]/ui8[6:4]/ui8[5:4]/ui8[4]. The remaining contents of vector registers vd/xd remain unchanged.

VEXTRINS.B:

VR[vd].B[ui8[7:4]] = VR[vj].B[ui8[3:0]]

VEXTRINS.H:

VR[vd].H[ui8[6:4]] = VR[vj].H[ui8[2:0]]

VEXTRINS.W:

VR[vd].W[ui8[5:4]] = VR[vj].W[ui8[1:0]]

VEXTRINS.D:

VR[vd].D[ui8[4]] = VR[vj].D[ui8[0]]

XVEXTRINS.B:

XR[xd].B[ui8[7:4]] = XR[xj].B[ui8[3:0]]
XR[xd].B[ui8[7:4]+16] = XR[xj].B[ui8[3:0]+16]

XVEXTRINS.H:

XR[xd].H[ui8[6:4]] = XR[xj].H[ui8[2:0]]
XR[xd].H[ui8[6:4]+8] = XR[xj].H[ui8[2:0]+8]

XVEXTRINS.W:

XR[xd].W[ui8[5:4]] = XR[xj].W[ui8[1:0]]
XR[xd].W[ui8[5:4]+4] = XR[xj].W[ui8[1:0]+4]

XVEXTRINS.D:

XR[xd].D[ui8[4]] = XR[xj].D[ui8[0]]
XR[xd].D[ui8[4]+2] = XR[xj].D[ui8[0]+2]

1.2.8 Vector Memory Access Instructions

1.2.8.1 [X]VLD[X], [X]VST[X]

Instruction Format:	vld	vd, rj, si12	xvld	xd, rj, si12
	vst	vd, rj, si12	xvst	xd, rj, si12
	vldx	vd, rj, rk	xvldx	xd, rj, rk

vstx	vd, rj, rk	xvstx	xd, rj, rk

The VLD and VLDX instructions retrieve 16 bytes of data from memory and write it to the vector register vd.

The VST and VSTX instructions write 16 bytes of data from vector register vd to memory.

The XVLD and XVLDX instructions retrieve 32 bytes of data from memory and write it to the vector register xd.

The XVST and XVSTX instructions write the 32 bytes of data from vector register xd to memory.

The memory access address calculation for VLD, VST, XVLD, and XVST is calculated by adding the value in general-purpose register rj to the sign-extended 12-bit immediate value si12.

The memory access address for VLDX, VSTX, XVLDX, and XVSTX is calculated by adding the value in general-purpose register rj to the value in the general-purpose register rk.

The above memory access instructions do not trigger address misalignment exceptions.

1.2.8.2 [X]VLDREPL.{B/H/W/D}, [X]VSTELM.{B/H/W/D}

Instruction Format:				
	vldrepl.b	vd, rj, si12	xvldrepl.b	xd, rj, si12
	vldrepl.h	vd,rj, si11	xvldrepl.h	xd, rj, si11
	vldrepl.w	vd, rj, si10	xvldrepl.w	xd, rj, si10
	vldrepl.d	vd, rj, si9	xvldrepl.d	xd, rj, si9
	vstelm.b	vd, rj, si8, idx	xvstelm.b	xd, rj, si8, idx
	vstelm.h	vd, rj, si8, idx	xvstelm.h	xd, rj, si8, idx
	vstelm.w	vd, rj, si8, idx	xvstelm.w	xd, rj, si8, idx
	vstelm.d	vd, rj, si8, idx	xvstelm.d	xd, rj, si8, idx

The [X]VLDREPL.{B/H/W/D} instruction retrieves a byte/half-word/word/double-word from memory and broadcasts it to each byte/half-word/word/double-word element of vector register vd/xd.

The [X]VSTELM.{B/H/W/D} instruction writes the byte/half-word/word/double-word elements specified by the immediate value idx in the vector register vd/xd to memory.

The memory access address for the above instructions is calculated by adding the value in the general-purpose register rj to the offset value obtained by multiplying the immediate value by the element width. That is, the virtual address of [X]VLDREPL.B is GR[rj] + SignExtend(si12). The virtual address of [X]VLDREPL.H is GR[rj] + SignExtend(si11 << 1). The virtual address of [X]VLDREPL.W is GR[rj] + SignExtend(si10 << 2). The virtual address of [X]VLDREPL.D is GR[rj] + SignExtend(si9 << 3). The virtual address of [X]VSTELM.B is GR[rj]+SignExtend(si8). The virtual address of [X]VSTELM.H is GR[rj]+SignExtend(si8<<1). The virtual address of [X]VSTELM.W is GR[rj]+SignExtend(si8<<2). The virtual address of [X]VSTELM.D is GR[rj]+SignExtend(si8<<3).

The above memory access instructions will not trigger an address misalignment exception.

Note that the immediate address offset values in the assembly representation of these instructions are in bytes. For [X]VLDREPL.H, the offset value in the assembly representation corresponds to si11<<1 in the instruction code. For [X]VLDREPL.W, the offset in the assembly representation corresponds to si10<<2 in the instruction code. For [X]VLDREPL.D, it corresponds to si9<<3. For [X]VSTELM.H, it corresponds to si8<<1;For [X]VSTELM.W, the offset value in the assembly representation corresponds to si8<<2 in the instruction code. For [X]VSTELM.D, the offset value in the assembly representation corresponds to si8<<3 in the instruction code.

2 Appendix A: Pseudocode Description of Function Definitions

2.1 Operator Definitions in Pseudocode

This section lists the meanings of statement keywords and various operators used in the pseudocode, along with their precedence relationships.

Additionally, common conventions for representing numerical values in different bases within the pseudocode are defined as follows:

- No prefix or the prefix "'d" or "##'d" denotes decimal numbers, where the "##'d" prefix indicates the decimal number's width is ## bits;
- The prefix "'b", or "##'b" prefixes denote binary numbers, where "##'b" indicates a binary number with a width of ## bits;
- The prefix "'h" or "##'h" denotes a hexadecimal number, where the "##'h" prefix indicates the hexadecimal number has a width of ## bits. In hexadecimal values, A through F are represented by uppercase letters.

Table 2-1 Explanation of Statement Keywords

Operator	Meaning
Return type Function name(variable, …): Function body return Return value	Function definition

if Condition1:
Execute statement 1
elif Condition 2:
Execute statement 2
else:
Execute statement 3
	Conditional statement

case Variable of: Value 1: Execute Statement 1 Value 2: Execute Statement 2 default: Default execution statement	case conditional statement
Condition ? TRUE Execution Statement : FALSE Execution Statement	Conditional judgment statement
for Loop variable in Sequence: Execution statement	for loop statement
range(N)	Integer sequence from 0 to N-1 with step size 1
range(Start value, End value, Step value)	A sequence from start value (inclusive) to end value (exclusive) with specified step size
break	Terminates the current loop
signed(…)	Signed integer
unsigned(…)	Unsigned integer
fp16(…)	Half-precision floating-point number
fp32(…)	Single-precision floating-point number
fp64(…)	Double-precision floating-point number
boolean	Boolean type
bit	Bit type
integer	Integer type
bits(N)	N-bit type
ZeroExtend(Variable, N)	Variable zero-extended to N bits
SignExtend(Variable, N)	SignExtend(variable, N)
isSNaN(Variable)	Returns TRUE if variable is a signaling NaN, FALSE otherwise
isQNaN(Variable)	TRUE if variable is a quiet NaN, FALSE otherwise

Operator	Meaning
SignalException(Exception)	Trigger an exception
#	Single-line comment
=	Assignment

Table 2-2 Explanation of Bit String Operator

Operator	Meaning
[M:N]	Nth to Mth bits of the bit string
{N{M}}	String M repeated N times and concatenated
{N, M, …}	Concatenate bit strings N, M, … in sequence

Table 2-3 Explanation of Arithmetic Operators

Operator	Meaning
+	Addition
-	Subtraction
*	Multiply
/	Divide
%	Modulo
**	Exponentiation

Table 2-4: Explanation of Comparison Operator

Operator	Meaning
==	Equal
!=	Not equal to
>	greater than
<	Less than
≥	Greater than or equal to
≤	Less than or equal to

Table 2-5: Explanation of Bitwise Operator

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise complement
<<	Logical left shift
>>	Logical right shift
>>>	Arithmetic Right Shift

Table 2-6: Explanation of Logical Operator

Operator	Meaning
and	Logical AND
or	Logical OR
not	logical negation

The operator precedence in pseudocode, listed from highest to lowest, is shown in Table 2-7:

Table 2-7: Operator Precedence

Operator	Meaning
**	Exponentiation
~	Bitwise complement
*, /, %	Multiplication, Division, Modulo
+, -	Addition, Subtraction
<<, >>, >>>	Logical left shift, logical right shift, arithmetic right shift
&	Bitwise AND
^, |	Bitwise XOR, Bitwise OR
>, <, >=, <=	Greater than, Less than, Greater than or equal to, Less than or equal to
==, !=	equals, does not equal
not	logical NOT
and, or	logical AND, logical OR

2.2 Pseudocode Description of Function

The pseudocode definitions used in this manual's instruction descriptions are as follows.

2.2.1 Saturation Arithmetic of Signed Integer Addition

```
{bits(N) } SSat_Add_S(bits(N) x, bits(N) y):
   sum = signed(x) + signed(y)
   if sum > 2**(N-1)-1 :
         return {1'b0, {N-1{1'b1}}}
   elif sum < -2**(N-1) :
         return {1'b1, {N-1{1'b0}}}
   else :
         return sum[N-1:0]
```

2.2.2 Saturation Arithmetic of Unsigned Integer Addition

```
{bits(N) } USat_Add_U(bits(N) x, bits(N) y):
   sum = unsigned(x) + unsigned(y)
   if sum > 2**N-1 :
           return {N{1'b1}}
   else :
           return sum[N-1:0]
```

2.2.3 Saturation Arithmetic of Signed Integer Subtraction

```
{bits(N) } SSat_Sub_S(bits(N) x, bits(N) y):
   diff = signed(x) - signed(y)
   if diff > 2**(N-1)-1 :
           return {1'b0, {N-1{1'b1}}}
   elif diff < -2**(N-1) :
           return {1'b1, {N-1{1'b0}}}
   else :
           return diff[N-1:0]
```

2.2.4 Saturation Arithmetic of Unsigned Integer Subtraction

```
{bits(N) } USat_Sub_U(bits(N) x, bits(N) y):
   diff = unsigned(x) - unsigned(y)
   if diff < 0 :
           return 0
   else :
           return diff[N-1:0]
```

2.2.5 Signed Integer Maximum Operation

```
{bits(N) } Max_S(bits(N) x, bits(N) y):
   if signed(x) > signed(y) :
           return x
   else :
           return y
```

2.2.6 Signed Integer Minimum Operation

```
{bits(N) } Min_S(bits(N) x, bits(N) y):
   if signed(x) < signed(y) :
           return x
   else :
           return y
```

2.2.7 Unsigned Integer Maximum Operation

```
{bits(N) } Max_U(bits(N) x, bits(N) y):
   if unsigned(x) > unsigned(y) :
           return x
   else :
           return y
```

2.2.8 Unsigned integer Minimum Operation

```
{bits(N) } Min_U(bits(N) x, bits(N) y):
   if unsigned(x) < unsigned(y) :
           return x
   else :
           return y
```

2.2.9 Absolute Operation of Signed Integer

```
{bits(N) } Abs(bits(N) x):
   if signed(x) < 0 :
           return -x
   else :
           return x
```

2.2.10 Absolute Operation of Difference between Signed Integers

```
{bits(N) } Abs_Diff_S(bits(N) x, bits(N) y):
   svalx = SignExtend(x, N+1)
   svaly = SignExtend(y, N+1)
   diff[N:0] = svalx - svaly
   result[N:0] = Abs(diff[N:0])
   return result[N-1:0]
```

2.2.11 Absolute Operation of Difference between Unsigned Integers

```
{bits(N) } Abs_Diff_U(bits(N) x, bits(N) y):
   uvalx = ZeroExtend(x, N+1)
   uvaly = ZeroExtend(y, N+1)
   diff[N:0] = uvalx - uvaly
   result[N:0] = Abs(diff[N:0])
   return result[N-1:0]
```

2.2.12 Logical Left Shift Operation

```
bits(N) SLL(bits(N) x, integer sa):
   if sa==0 :
           result = x
   else :
           result = {x[N-sa-1:0], {sa{1'b0}}}
   return result
```

2.2.13 Logical Right Shift Operation

```
bits(N) SRL(bits(N) x, integer sa):
   if sa==0 :
           result = x
   else :
           result = {{sa{1'b0}}, x[N-1:sa]}
   return result
```

2.2.14 Arithmetic Right Shift Operation

```
bits(N) SRA(bits(N) x, integer sa):
   if sa==0 :
           result = x
   else :
           result = {{sa{x[N-1]}}, x[N-1:sa]}
   return result
```

2.2.15 Circular Right Shift Operation

```
bits(N) ROTR(bits(N) x, integer sa):
   if sa==0 :
           result = x
   else :
           result = {x[sa-1:0], x[N-1:sa]}
   return result
```

2.2.16 Circular Left Shift Operation

```
bits(N) ROTL(bits(N) x, integer sa):
   if sa == 0:
           result = x
   else :
           result = {x[N-sa-1:sa], x[N-1:N-sa]}
   return result
```

2.2.17 Logical Left Shift with Zero Extension to Double Width

```
bits(2N) SLLW_U(bits(N) x, integer sa):
   dx = ZeroExtend(x, 2N)
   if sa == 0:
           return dx
   else :
           return {dx[2N-sa-1:0], {sa{1'b0}}}
```

2.2.18 Logical Left Shift with Sign Extension to Double Width

```
bits(2N) SLLW_S(bits(N) x, integer sa):
    dx = SignExtend(x, 2N)
    if sa == 0:
            return dx
    else :
            return {dx[2N-sa-1:0], {sa{1'b0}}}
```

2.2.19 Logical Right Shift with Rounding

```
bits(N) SRLR(bits(N) x, integer sa):
   if sa==0 :
           result = x
   else :
           result = {{sa{1'b0}}, x[N-1:sa]} + x[sa-1]
   return result
```

2.2.20 Arithmetic Right Shift with Rounding

```
bits(N) SRAR(bits(N) x, integer sa):
   if sa==0 :
           result = x
   else :
           result = {{sa{x[N-1]}}, x[N-1:sa]} + x[sa-1]
   return result
```

2.2.21 Logical Right Shift with Half Width

```
bits(N) SRLN(bits(2N) x, integer sa):
    if sa==0 :
            shft_res = x
    else :
            shft_res = {{sa{1'b0}}, x[2N-1:sa]}
    return shft_res[N-1:0]
```

2.2.22 Arithmetic Right Shift with Half Width

```
bits(N) SRAN(bits(2N) x, integer sa):
   if sa==0 :
           shft_res = x
   else :
           shft_res = {{sa{x[2N-1]}}, x[2N-1:sa]}
   return shft_res[N-1:0]
```

2.2.23 Logical Right Shift with Rounding and Half Width

```
bits(N) SRLRN(bits(2N) x, integer sa):
   if sa==0 :
           shft_res = x
   else :
           shft_res = {{sa{1'b0}}, x[2N-1:sa]} + x[sa-1]
   return shft_res[N-1:0]
```

2.2.24 Arithmetic Right Shift with Rounding and Half Width

```
bits(N) SRARN(bits(2N) x, integer sa):
   if sa==0 :
           shft_res = x
   else :
           shft_res = {{sa{x[2N-1]}}, x[2N-1:sa]} + x[sa-1]
   return shft_res[N-1:0]
```

2.2.25 Logical Right Shift and Unsigned Saturation Arithmetic with Half Width

```
bits(N) SSRLNU(bits(2N) x, integer sa):
   if sa==0 :
           shft_res = x
   else :
           shft_res = {{sa{1'b0}}, x[2N-1:sa]}
   if shft_res > 2**N-1 :
           return {N{1'b1}}
   else :
           return shft_res[N-1:0]
```

2.2.26 Logical Right Shift and Signed Saturation Arithmetic with Half Width

```
bits(N) SSRLNS(bits(2N) x, integer sa):
   if sa==0 :
           shft_res = x
   else :
           shft_res = {{sa{1'b0}}, x[2N-1:sa]}
   if unsigned(shft_res) > 2**(N-1)-1 :
           return {1'b0, {N-1{1'b1}}}
   else :
           return shft_res[N-1:0]
```

2.2.27 Arithmetic Right Shift and Unsigned Saturation Arithmetic with Half Width

```
bits(N) SSRANU(bits(2N) x, integer sa):
   if sa==0 :
           shft_res = x
   elif signed(x) < 0 :
           shft_res = 0
   else :
           shft_res = {{sa{x[2N-1]}}, x[2N-1:sa]}
   if shft_res > 2**N-1 :
           return {N{1'b1}}
   else :
           return shft_res[N-1:0]
```

2.2.28 Arithmetic Right Shift and Signed Saturation Arithmetic with Half Width

```
bits(N) SSRANS(bits(2N) x, integer sa):
   if sa==0 :
           shft_res = x
   else :
           shft_res = {{sa{x[2N-1]}}, x[2N-1:sa]}
   if shft_res > 2**(N-1)-1 :
           return {1'b0, {N-1{1'b1}}}
   elif shft_res < -2**(N-1) :
           return {1'b1, {N-1{1'b0}}}
   else :
           return shft_res[N-1:0]
```

2.2.29 Logical Right Shift and Unsigned Saturation Arithmetic with Rounding and Half Width

```
bits(N) SSRLRNU(bits(2N) x, integer sa):
   if sa==0 :
           shft_res = x
   else :
           shft_res = {{sa{1'b0}}, x[2N-1:sa]} + x[sa-1]
   if shft_res > 2**N-1 :
           return {N{1'b1}}
   else :
           return shft_res[N-1:0]
```

2.2.30 Logical Right Shift and Signed Saturation Arithmetic with Rounding and Half Width

```
bits(N) SSRLRNS(bits(2N) x, integer sa):
   if sa==0 :
           shft_res = x
   else :
           shft_res = {{sa{1'b0}}, x[2N-1:sa]} + x[sa-1]
   if unsigned(shft_res) > 2**(N-1)-1 :
           return {1'b0, {N-1{1'b1}}}
   else :
           return shft_res[N-1:0]
```

2.2.31 Arithmetic Right Shift and Unsigned Saturation Arithmetic with Rounding and Half Width

```
bits(N) SSRARNU(bits(2N) x, integer sa):
   if sa==0 :
           shft_res = x
   elseif signed(x) < 0 :
           shft_res = 0
   else:
           shft_res = {{sa{x[2N-1]}}, x[2N-1:sa]} + x[sa-1]
   if shft_res > 2**(N-1):
           return {N{1'b1}}
   else:
           return shft_res[N-1:0]
```

2.2.32 Arithmetic Right Shift and Signed Saturation Arithmetic with Rounding and Half Width

```
bits(N) SSRARNS(bits(2N) x, integer sa):
   if sa==0 :
           shft_res = x
   else :
           shft_res = {{sa{x[2N-1]}}, x[2N-1:sa]} + x[sa-1]
   if shft_res > 2**(N-1)-1 :
           return {1'b0, {N-1{1'b1}}}
   elif shft_res < -2**(N-1) :
           return {1'b1, {N-1{1'b0}}}
   else:
           return shft_res[N-1:0]
```

2.2.33 Signed Saturation Arithmetic up to Valid Bits

```
{bits(N)} Sat_S(bits(N) x, bits(M) y):
   if x > 2**y-1 :
           return {{(N-y){1'b0}}, {y{1'b1}}}
   elif x < -2**y :
           return {{(N-y){1'b1}}, {y{1'b0}}}
   elif x < -2**y :
           return x[N-1:0]
```

2.2.34 Unsigned Saturation Arithmetic up to Valid Bits

```
{bits(N) } Sat_U(bits(N) x, bits(M) y):
   if x > 2**(y+1)-1 :
           return {{(N-y-1){1'b0}}, {(y+1){1'b1}}}
   else :
           return x[N-1:0]
```

2.2.35 Operation of Counting the Number of Consecutive 1 Starting from Highest Bit

```
{bits(N) } CLO(bits(N) x):
   cnt = 0
   for i in range(N):
           if x[N-1-i] == 1'b0:
                   return cnt
           else:
                   cnt = cnt + 1
```

2.2.36 Operation of Counting the Number of Consecutive 0 Starting from Highest Bit

```
{bits(N) } CLZ(bits(N) x):
   cnt = 0
   for i in range(N):
           if x[N-1-i] == 1'b1:
                   return cnt
           else:
                   cnt = cnt + 1
```

2.2.37 Operation of Counting the number of 1

```
{bits(N)} PCNT(bits(N) x):
   cnt = 0
   for i in range(N):
           cnt += x[i]
   return cnt
```

2.2.38 Operation of Setting a Specific Bit to 0 in an Element

```
{bits(N) } Bit_Clear(bits(N) x, bit(M) y):
   for i in range(N) :
           res_bits[i] = (i == y) ? 1'b0 : x[i]
   return res_bits
```

2.2.39 Operation of Setting a Specific Bit to 1 in an Element

```
{bits(N) } Bit_Set(bits(N) x, bit(M) y):
   for i in range(N) :
           res_bits[i] = (i==y) ? 1'b1 : x[i]
   return res_bits
```

2.2.40 Operation of Invert a Specified Bit in an Element

```
{bits(N) } Bit_Reverse(bits(N) x, bit(M) y):
   for i in range(N) :
           res_bits[i] = (i==y) ? ~x[i] : x[i]
   return res_bits
```

2.2.41 Operation of Converting Value Based on Sign Bit

```
{bits(N)} Sign_Convert(bits(N) x, bits(N) y):
   return x==0 ? 0 : x<0 ? -y : y
```

2.2.42 Operation of Single-precision Floating-point Number Rounding

```
{bits(32) } FP32_roundToInteger(bits(N) x, bits(1) I_en, bits(2) rm):
   if (I_en) :
           return FP32_roundToIntegralExact(x)
   elif (rm=0) :
           return FP32_roundToIntegerTiesToEven(x)
   elif (rm=1) :
           return FP32_roundToIntegerTowardZero(x)
   elif (rm=2) :
           return FP32_roundToIntegerTowardPositive(x)
   elif (rm=3) :
           return FP32_roundToIntegerTowardNegative(x)
```

2.2.43 Operation of Double-precision Floating-point Number Rounding

```
{bits(64) } FP64_roundToInteger(bits(N) x, bits(1) I_en, bits(2) rm):
   if (I_en) :
           return FP64_roundToIntegralExact(x)
   elif (rm=0) :
           return FP64_roundToIntegerTiesToEven(x)
   elif (rm=1) :
           return FP64_roundToIntegerTowardZero(x)
   elif (rm=2) :
           return FP64_roundToIntegerTowardPositive(x)
   elif (rm=3) :
           return FP64_roundToIntegerTowardNegative(x)
```

2.2.44 Operation of Converting Floating-point Number to Signed Integer Word

```
{bits(32) } Sint32_convertToInteger(bits(N) x, bits(1) I_en, bits(2) rm):
   case {I_en, rm} of:
           {1'b1, 2'd0}: return Sint32_convertToIntegerExactTiesToEven(x)
           {1'b1, 2'd1}: return Sint32_convertToIntegerExactTowardZero(x)
           {1'b1, 2'd2}: return Sint32_convertToIntegerExactTowardPositive(x)
           {1'b1, 2'd3}: return Sint32_convertToIntegerExactTowardNegative(x)
           {1'b0, 2'd0}: return Sint32_convertToIntegerTiesToEven(x)
           {1'b0, 2'd1}: return Sint32_convertToIntegerTowardZero(x)
           {1'b0, 2'd2}: return Sint32_convertToIntegerTowardPositive(x)
           {1'b0, 2'd3}: return Sint32_convertToIntegerTowardNegative(x)
```

2.2.45 Operation of Converting Floating-point Number to Signed Integer Double-word

```
{bits(64) } Sint64_convertToInteger(bits(N) x, bits(1) I_en, bits(2) rm):
   case {I_en, rm} of:
           {1'b1, 2'd0}: return Sint64_convertToIntegerExactTiesToEven(x)
           {1'b1, 2'd1}: return Sint64_convertToIntegerExactTowardZero(x)
           {1'b1, 2'd2}: return Sint64_convertToIntegerExactTowardPositive(x)
           {1'b1, 2'd3}: return Sint64_convertToIntegerExactTowardNegative(x)
           {1'b0, 2'd0}: return Sint64_convertToIntegerTiesToEven(x)
           {1'b0, 2'd1}: return Sint64_convertToIntegerTowardZero(x)
           {1'b0, 2'd2}: return Sint64_convertToIntegerTowardPositive(x)
           {1'b0, 2'd3}: return Sint64_convertToIntegerTowardNegative(x)
```

2.2.46 Operation of Converting Floating-point Number to Unsigned Integer Word

```
{bits(32) } Uint32_convertToInteger(bits(N) x, bits(1) I_en, bits(2) rm):
   case {I_en, rm} of:
           {1'b1, 2'd0}: return Uint32_convertToIntegerExactTiesToEven(x)
           {1'b1, 2'd1}: return Uint32_convertToIntegerExactTowardZero(x)
           {1'b1, 2'd2}: return Uint32_convertToIntegerExactTowardPositive(x)
           {1'b1, 2'd3}: return Uint32_convertToIntegerExactTowardNegative(x)
           {1'b0, 2'd0}: return Uint32_convertToIntegerTiesToEven(x)
           {1'b0, 2'd1}: return Uint32_convertToIntegerTowardZero(x)
           {1'b0, 2'd2}: return Uint32_convertToIntegerTowardPositive(x)
           {1'b0, 2'd3}: return Uint32_convertToIntegerTowardNegative(x)
```

2.2.47 Operation of Converting Floating-point Number to Unsigned Integer Double-word

```
{bits(64) } Uint64_convertToInteger(bits(N) x, bits(1) I_en, bits(2) rm):
   case {I_en, rm} of:
           {1'b1, 2'd0}: return Uint64_convertToIntegerExactTiesToEven(x)
           {1'b1, 2'd1}: return Uint64_convertToIntegerExactTowardZero(x)
           {1'b1, 2'd2}: return Uint64_convertToIntegerExactTowardPositive(x)
           {1'b1, 2'd3}: return Uint64_convertToIntegerExactTowardNegative(x)
           {1'b0, 2'd0}: return Uint64_convertToIntegerTiesToEven(x)
           {1'b0, 2'd1}: return Uint64_convertToIntegerTowardZero(x)
           {1'b0, 2'd2}: return Uint64_convertToIntegerTowardPositive(x)
           {1'b0, 2'd3}: return Uint64_convertToIntegerTowardNegative(x)
```

2.2.48 Operation of Generating Value Based on Immediate

```
{bits(64) } Set_Value(bits(13) x):
   if x[12]==1'b0 :
           case x[11:10] of :
                   2'b00 : data = {8{x[7:0]}}
                   2'b01 : data = {4{SignExtend(x[9:0], 16)}}
                   2'b10 : data = {2{SignExtend(x[9:0], 32)}}
                   2'b11 : data = SignExtend(x[9:0], 64)
   else :
           case x[11:8] of :
                   4'b0000 : data = {2{24'b0, x[7:0]}}
                   4'b0001 : data = {2{16'b0, x[7:0], 8'b0}}
                   4'b0010 : data = {2{8'b0, x[7:0], 16'b0}}
                   4'b0011 : data = {2{x[7:0], 24'b0}}
                   4'b0100 : data = {4{8'b0, x[7:0]}}
                   4'b0101 : data = {4{x[7:0], 8'b0}}
                   4'b0110 : data = {2{16'b0, x[7:0], 8'hFF}}
                   4'b0111 : data = {2{8'b0, x[7:0], 16'hFFFF}}
                   4'b1000 : data = {8{x[7:0]}}
                   4'b1001 : data = {{8{x[7]}}, {8{x[6]}}, {8{x[5]}}, {8{x[4]}},
                                     {8{x[3]}}, {8{x[2]}}, {8{x[1]}}, {8{x[0]}}}
                   4'b1010 : data = {2{x[7], ~x[6], {5{x[6]}}, x[5:0], 19'b0}}
                   4'b1011 : data = {32'b0, x[7], ~x[6], {5{x[6]}}, x[5:0], 19'b0}
                   4'b1100 : data = {x[7], ~x[6], {8{x[6]}}, x[5:0], 48'b0}
                   default : Raise_Exception(INE)
   return data
```

3 Appendix B: Instruction Code List

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VFMADD.S	**vd, vj, vk, va**	0	0	0	0	1	0	0	1	0	0	0	1	va					vk					vj					vd				
VFMADD.D	**vd, vj, vk, va**	0	0	0	0	1	0	0	1	0	0	0	1	0	va				vk					vj					vd				
VFMSUB.S	**vd, vj, vk, va**	0	0	0	0	1	0	0	1	0	1	0	1	va					vk					vj					vd				
VFMSUB.D	**vd, vj, vk, va**	0	0	0	0	1	0	0	1	0	1	1	0	va					vk					vj					vd				
VFNMADD.S	**vd, vj, vk, va**	0	0	0	0	1	0	0	1	1	0	0	1	va					vk					vj					vd				
VFNMADD.D	**vd, vj, vk, va**	0	0	0	0	1	0	0	1	1	0	1	0	va					vk					vj					vd				
VFNMSUB.S	**vd, vj, vk, va**	0	0	0	0	1	0	0	1	1	1	0	1	va					vk					vj					vd				
VFNMSUB.D	**vd, vj, vk, va**	0	0	0	0	1	0	0	1	1	1	1	0	va					vk					vj					vd				
XVFMADD.S	**xd, xj, xk, xa**	0	0	0	0	1	0	1	0	0	0	0	1	xa					xk					xj					xd				
XVFMADD.D	**xd, xj, xk, xa**	0	0	0	0	1	0	1	0	0	0	1	0	xa					xk					xj					xd				
XVFMSUB.S	**xd, xj, xk, xa**	0	0	0	0	1	0	1	0	0	1	0	1	xa					xk					xj					xd				
XVFMSUB.D	**xd, xj, xk, xa**	0	0	0	0	1	0	1	0	0	1	1	0	xa					xk					xj					xd				
XVFNMADD.S	**xd, xj, xk, xa**	0	0	0	0	1	0	1	0	1	0	0	1	xa					xk					xj					xd				
XVFNMADD.D	**xd, xj, xk, xa**	0	0	0	0	1	0	1	0	1	0	1	0	xa					xk					xj					xd				
XVFNMSUB.S	**xd, xj, xk, xa**	0	0	0	0	1	0	1	0	1	1	0	1	xa					xk					xj					xd				
XVFNMSUB.D	**xd, xj, xk, xa**	0	0	0	0	1	0	1	0	1	1	1	0	xa					xk					xj					xd				
VFCMP.cond.S	**vd, vj, vk**	0	0	0	0	1	1	0	0	0	1	0	1	cond					vk					vj					vd				
VFCMP.cond.D	**vd, vj, vk**	0	0	0	0	1	1	0	0	0	1	1	0	cond					vk					vj					vd				
XVFCMP.cond.S	**xd, xj, xk**	0	0	0	0	1	1	0	0	1	0	0	1	cond					xk					xj					xd				
XVFCMP.cond.D	**xd, xj, xk**	0	0	0	0	1	1	0	0	1	0	1	0	cond					xk					xj					xd				
VBITSEL.V	**vd, vj, vk, va**	0	0	0	0	1	1	0	1	0	0	0	1	va					vk					vj					vd				
XVBITSEL.V	**xd, xj, xk, xa**	0	0	0	0	1	1	0	1	0	0	1	0	xa					xk					xj					xd				
VSHUF.B	**vd, vj, vk, va**	0	0	0	0	1	1	0	1	0	1	0	1	va					vk					vj					vd				
XVSHUF.B	**xd, xj, xk, xa**	0	0	0	0	1	1	0	1	0	1	1	0	xa					xk					xj					xd				
VLD	**vd, rj, si12**	0	0	1	0	1	1	0	0	0	0	si12												rj					vd				
VST	**vd, rj, si12**	0	0	1	0	1	1	0	0	0	1	si12												rj					vd				
XVLD	**xd, rj, si12**	0	0	1	0	1	1	0	0	1	0	si12												rj					xd				
XVST	**xd, rj, si12**	0	0	1	0	1	1	0	0	1	1	si12												rj					xd				
VLDREPL.D	**vd, rj, si9**	0	0	1	1	0	0	0	0	0	0	0	1	0	si9									rj					vd				
VLDREPL.W	**vd, rj, si10**	0	0	1	1	0	0	0	0	0	0	1	0	si10										rj					vd				
VLDREPL.H	**vd, rj, si11**	0	0	1	1	0	0	0	0	0	1	0	si11											rj					vd				
VLDREPL.B	**vd, rj, si12**	0	0	1	1	0	0	0	0	1	0	si12												rj					vd				
VSTELM.D	**vd, rj, si8, idx**	0	0	1	1	0	0	0	1	0	0	0	1	0	idx	si8								rj					vd				
VSTELM.W	**vd, rj, si8, idx**	0	0	1	1	0	0	0	1	0	0	0	1	0	idx	si8								rj					vd				
VSTELM.H	**vd, rj, si8, idx**	0	0	1	1	0	0	0	1	0	1	0	idx	si8										rj					vd				
VSTELM.B	**vd, rj, si8, idx**	0	0	1	1	0	0	0	1	1	0	idx	si8											rj					vd				
XVLDREPL.D	**xd, rj, si9**	0	0	1	1	0	0	1	0	0	0	0	1	0	si9									rj					xd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
XVLDREPL.W	**xd, rj, si10**	0	0	1	1	0	0	1	0	0	0	1	0	si10										rj					xd				
XVLDREPL.H	**xd, rj, si11**	0	0	1	1	0	0	1	0	0	1	0	si11											rj					xd				
XVLDREPL.B	**xd, rj, si12**	0	0	1	1	0	0	1	0	1	0	si12												rj					xd				
XVSTELM.D	**xd, rj, si8, idx**	0	0	1	1	0	0	1	1	0	0	0	1	idx		si8								rj					xd				
XVSTELM.W	**xd, rj, si8, idx**	0	0	1	1	0	0	1	1	0	0	1	idx		si8									rj					xd				
XVSTELM.H	**xd, rj, si8, idx**	0	0	1	1	0	0	1	1	0	1	idx		si8										rj					xd				
XVSTELM.B	**xd, rj, si8, idx**	0	0	1	1	0	0	1	1	1	idx		si8											rj					xd				
VLDX	**vd, rj, rk**	0	0	1	1	1	0	0	0	0	1	0	0	0	0	0	0	0	rk					rj					vd				
VSTX	**vd, rj, rk**	0	0	1	1	1	0	0	0	0	1	0	0	0	1	0	0	0	rk					rj					vd				
XVLDX	**xd, rj, rk**	0	0	1	1	1	0	0	0	0	1	0	0	1	0	0	0	0	rk					rj					xd				
XVSTX	**xd, rj, rk**	0	0	1	1	1	0	0	0	0	1	0	0	1	1	0	0	0	rk					rj					xd				
VSEQ.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	vk					vj					vd				
VSEQ.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	vk					vj					vd				
VSEQ.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	vk					vj					vd				
VSEQ.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	vk					vj					vd				
VSLE.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	vk					vj					vd				
VSLE.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	vk					vj					vd				
VSLE.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	vk					vj					vd				
VSLE.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	vk					vj					vd				
VSLE.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	vk					vj					vd				
VSLE.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	vk					vj					vd				
VSLE.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	vk					vj					vd				
VSLE.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	vk					vj					vd				
VSLT.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	vk					vj					vd				
VSLT.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	vk					vj					vd				
VSLT.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	0	vk					vj					vd				
VSLT.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	vk					vj					vd				
VSLT.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	vk					vj					vd				
VSLT.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	1	vk					vj					vd				
VSLT.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	vk					vj					vd				
VSLT.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	1	vk					vj					vd				
VADD.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	1	0	vk					vj					vd				
VADD.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	1	1	vk					vj					vd				
VADD.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	1	0	vk					vj					vd				
VADD.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	1	1	vk					vj					vd				
VSUB.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	vk					vj					vd				
VSUB.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	1	1	0	1	vk					vj					vd				
VSUB.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	0	vk					vj					vd				
VSUB.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	1	vk					vj					vd				
VADDWEV.H.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	0	vk					vj					vd				
VADDWEV.W.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	vk					vj					vd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VADDWEV.D.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	1	1	1	1	1	0	vk					vj					vd				
VADDWEV.Q.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1	vk					vj					vd				
VSUBWEV.H.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	vk					vj					vd				
VSUBWEV.W.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	vk					vj					vd				
VSUBWEV.D.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1	0	vk					vj					vd				
VSUBWEV.Q.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1	1	vk					vj					vd				
VADDWOD.H.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	0	0	0	1	0	0	vk					vj					vd				
VADDWOD.W.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	0	0	0	1	0	1	vk					vj					vd				
VADDWOD.D.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	0	0	0	1	1	0	vk					vj					vd				
VADDWOD.Q.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	0	0	0	1	1	1	vk					vj					vd				
VSUBWOD.H.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	0	0	1	0	0	0	vk					vj					vd				
VSUBWOD.W.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	0	0	1	0	0	1	vk					vj					vd				
VSUBWOD.D.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	0	vk					vj					vd				
VSUBWOD.Q.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	1	vk					vj					vd				
VADDWEV.H.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	0	1	1	1	0	0	vk					vj					vd				
VADDWEV.W.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	0	1	1	1	0	1	vk					vj					vd				
VADDWEV.D.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	0	1	1	1	1	0	vk					vj					vd				
VADDWEV.Q.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	0	1	1	1	1	1	vk					vj					vd				
VSUBWEV.H.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	vk					vj					vd				
VSUBWEV.W.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	1	0	0	0	0	1	vk					vj					vd				
VSUBWEV.D.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	1	0	0	0	1	0	vk					vj					vd				
VSUBWEV.Q.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	1	0	0	0	1	1	vk					vj					vd				
VADDWOD.H.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	1	0	0	1	0	0	vk					vj					vd				
VADDWOD.W.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	1	0	0	1	0	1	vk					vj					vd				
VADDWOD.D.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	1	0	0	1	1	0	vk					vj					vd				
VADDWOD.Q.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	1	0	0	1	1	1	vk					vj					vd				
VSUBWOD.H.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	1	0	1	0	0	0	vk					vj					vd				
VSUBWOD.W.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	1	0	1	0	0	1	vk					vj					vd				
VSUBWOD.D.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	1	0	1	0	1	0	vk					vj					vd				
VSUBWOD.Q.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	1	0	1	0	1	1	vk					vj					vd				
VADDWEV.H.BU.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	1	1	1	1	0	0	vk					vj					vd				
VADDWEV.W.HU.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	1	1	1	1	0	1	vk					vj					vd				
VADDWEV.D.WU.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	1	1	1	1	1	0	vk					vj					vd				
VADDWEV.Q.DU.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1	vk					vj					vd				
VADDWOD.H.BU.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	vk					vj					vd				
VADDWOD.W.HU.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1	vk					vj					vd				
VADDWOD.D.WU.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	0	0	0	1	0	vk					vj					vd				
VADDWOD.Q.DU.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	0	0	0	1	1	vk					vj					vd				
VSADD.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	0	0	1	1	0	vk					vj					vd				
VSADD.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	0	0	1	1	1	vk					vj					vd				
VSADD.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	0	0	1	1	0	vk					vj					vd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VSADD.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	0	1	1	1	1	vk					vj					vd				
VSSUB.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	1	0	0	0	0	vk					vj					vd				
VSSUB.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	1	0	0	0	1	vk					vj					vd				
VSSUB.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	1	0	0	1	0	vk					vj					vd				
VSSUB.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	1	0	0	1	1	vk					vj					vd				
VSADD.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	1	0	1	0	0	vk					vj					vd				
VSADD.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	1	0	1	0	1	vk					vj					vd				
VSADD.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	1	0	1	1	0	vk					vj					vd				
VSADD.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	1	0	1	1	1	vk					vj					vd				
VSSUB.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	1	1	0	0	0	vk					vj					vd				
VSSUB.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	1	1	0	0	1	vk					vj					vd				
VSSUB.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	1	1	0	1	0	vk					vj					vd				
VSSUB.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	0	1	1	0	1	1	vk					vj					vd				
VHADDW.H.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	0	1	0	0	0	vk					vj					vd				
VHADDW.W.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	0	1	0	0	1	vk					vj					vd				
VHADDW.D.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	0	1	0	1	0	vk					vj					vd				
VHADDW.Q.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	0	1	0	1	1	vk					vj					vd				
VHSUBW.H.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	0	1	1	0	0	vk					vj					vd				
VHSUBW.W.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	0	1	1	0	1	vk					vj					vd				
VHSUBW.D.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	0	1	1	1	0	vk					vj					vd				
VHSUBW.Q.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	0	1	1	1	1	vk					vj					vd				
VHADDW.HU.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	1	0	0	0	0	vk					vj					vd				
VHADDW.WU.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	1	0	0	0	1	vk					vj					vd				
VHADDW.DU.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	1	0	0	1	0	vk					vj					vd				
VHADDW.QU.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	1	0	0	1	1	vk					vj					vd				
VHSUBW.HU.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	1	0	1	0	0	vk					vj					vd				
VHSUBW.WU.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	1	0	1	0	1	vk					vj					vd				
VHSUBW.DU.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	1	0	1	1	0	vk					vj					vd				
VHSUBW.QU.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	1	0	1	1	1	vk					vj					vd				
VADDA.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	1	1	0	0	0	vk					vj					vd				
VADDA.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	1	1	0	0	1	vk					vj					vd				
VADDA.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	1	1	0	1	0	vk					vj					vd				
VADDA.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	0	1	1	1	0	1	1	vk					vj					vd				
VABSD.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0	vk					vj					vd				
VABSD.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0	0	1	vk					vj					vd				
VABSD.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0	1	0	vk					vj					vd				
VABSD.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1	vk					vj					vd				
VABSD.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	0	0	1	0	0	vk					vj					vd				
VABSD.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	0	0	1	0	1	vk					vj					vd				
VABSD.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	0	0	1	1	0	vk					vj					vd				
VABSD.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	0	0	1	1	1	vk					vj					vd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VAVG.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	0	1	0	0	0	vk					vj					vd				
VAVG.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	0	1	0	0	1	vk					vj					vd				
VAVG.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	0	1	0	1	0	vk					vj					vd				
VAVG.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	0	1	0	1	1	vk					vj					vd				
VAVG.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	0	1	1	0	0	vk					vj					vd				
VAVG.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	0	1	1	0	1	vk					vj					vd				
VAVG.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	0	1	1	1	0	vk					vj					vd				
VAVG.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	0	1	1	1	1	vk					vj					vd				
VAVGR.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	1	0	0	0	0	vk					vj					vd				
VAVGR.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	1	0	0	0	1	vk					vj					vd				
VAVGR.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	1	0	0	1	0	vk					vj					vd				
VAVGR.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	1	0	0	1	1	vk					vj					vd				
VAVGR.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	1	0	1	0	0	vk					vj					vd				
VAVGR.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	1	0	1	0	1	vk					vj					vd				
VAVGR.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	1	0	1	1	0	vk					vj					vd				
VAVGR.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	0	1	0	1	1	1	vk					vj					vd				
VMAX.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	vk					vj					vd				
VMAX.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	1	0	0	0	0	1	vk					vj					vd				
VMAX.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	1	0	0	0	1	0	vk					vj					vd				
VMAX.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	1	0	0	0	1	1	vk					vj					vd				
VMIN.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	1	0	0	1	0	0	vk					vj					vd				
VMIN.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	1	0	0	1	0	1	vk					vj					vd				
VMIN.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	1	0	0	1	1	0	vk					vj					vd				
VMIN.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	1	0	0	1	1	1	vk					vj					vd				
VMAX.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	1	0	1	0	0	0	vk					vj					vd				
VMAX.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	1	0	1	0	0	1	vk					vj					vd				
VMAX.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	1	0	1	0	1	0	vk					vj					vd				
VMAX.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	1	0	1	0	1	1	vk					vj					vd				
VMIN.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	1	0	1	1	0	0	vk					vj					vd				
VMIN.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	1	0	1	1	0	1	vk					vj					vd				
VMIN.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	1	0	1	1	1	0	vk					vj					vd				
VMIN.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	0	1	1	1	0	1	1	1	1	vk					vj					vd				
VMUL.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	0	0	1	0	0	0	vk					vj					vd				
VMUL.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	0	0	1	0	0	1	vk					vj					vd				
VMUL.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	0	0	1	0	1	0	vk					vj					vd				
VMUL.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	0	0	1	0	1	1	vk					vj					vd				
VMUH.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	0	0	1	1	0	0	vk					vj					vd				
VMUH.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	0	0	1	1	0	1	vk					vj					vd				
VMUH.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	0	0	1	1	1	0	vk					vj					vd				
VMUH.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	0	0	1	1	1	1	vk					vj					vd				
VMUH.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	0	1	0	0	0	0	vk					vj					vd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VMUH.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	0	1	0	0	0	1	vk					vj					vd				
VMUH.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	0	1	0	0	1	0	vk					vj					vd				
VMUH.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	0	1	0	0	1	1	vk					vj					vd				
VMULWEV.H.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	1	0	0	0	0	0	vk					vj					vd				
VMULWEV.W.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	1	0	0	0	0	1	vk					vj					vd				
VMULWEV.D.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	1	0	0	0	1	0	vk					vj					vd				
VMULWEV.Q.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	1	0	0	0	1	1	vk					vj					vd				
VMULWOD.H.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	1	0	0	1	0	0	vk					vj					vd				
VMULWOD.W.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	1	0	0	1	0	1	vk					vj					vd				
VMULWOD.D.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	1	0	0	1	1	0	vk					vj					vd				
VMULWOD.Q.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	1	0	0	1	1	1	vk					vj					vd				
VMULWEV.H.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	1	1	0	0	0	0	vk					vj					vd				
VMULWEV.W.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	1	1	0	0	0	1	vk					vj					vd				
VMULWEV.D.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	1	1	0	0	1	0	vk					vj					vd				
VMULWEV.Q.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	1	1	0	0	1	1	vk					vj					vd				
VMULWOD.H.BU	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	1	1	0	1	0	0	vk					vj					vd				
VMULWOD.W.HU	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	1	1	0	1	0	1	vk					vj					vd				
VMULWOD.D.WU	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	1	1	0	1	1	0	vk					vj					vd				
VMULWOD.Q.DU	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	0	1	1	0	1	1	1	vk					vj					vd				
VMULWEV.H.BU.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	0	0	0	0	0	vk					vj					vd				
VMULWEV.W.HU.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	0	0	0	0	1	vk					vj					vd				
VMULWEV.D.WU.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	0	0	0	1	0	vk					vj					vd				
VMULWEV.Q.DU.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	0	0	0	1	1	vk					vj					vd				
VMULWOD.H.BU.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	0	0	1	0	0	vk					vj					vd				
VMULWOD.W.HU.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	0	0	1	0	1	vk					vj					vd				
VMULWOD.D.WU.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	0	0	1	1	0	vk					vj					vd				
VMULWOD.Q.DU.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	0	0	1	1	1	vk					vj					vd				
VMADD.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	1	0	0	0	0	vk					vj					vd				
VMADD.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	1	0	0	0	1	vk					vj					vd				
VMADD.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	1	0	0	1	0	vk					vj					vd				
VMADD.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	1	0	0	1	1	vk					vj					vd				
VMSUB.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	1	0	1	0	0	vk					vj					vd				
VMSUB.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	1	0	1	0	1	vk					vj					vd				
VMSUB.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	1	0	1	1	0	vk					vj					vd				
VMSUB.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	1	0	1	1	1	vk					vj					vd				
VMADDWEV.H.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	1	1	0	0	0	vk					vj					vd				
VMADDWEV.W.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	1	1	0	0	1	vk					vj					vd				
VMADDWEV.D.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	1	1	0	1	0	vk					vj					vd				
VMADDWEV.Q.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	1	1	0	1	1	vk					vj					vd				
VMADDWOD.H.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	1	1	1	0	0	vk					vj					vd				
VMADDWOD.W.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	1	1	1	0	1	vk					vj					vd				

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VMADDWOD.D.W **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	1	1	1	1	0	vk					vj					vd				
VMADDWOD.Q.D **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	0	1	1	1	1	1	vk					vj					vd				
VMADDWEV.H.BU **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	1	0	1	0	0	0	vk					vj					vd				
VMADDWEV.W.HU **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	1	0	1	0	0	1	vk					vj					vd				
VMADDWEV.D.WU **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	1	0	1	0	1	0	vk					vj					vd				
VMADDWEV.Q.DU **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	1	0	1	0	1	1	vk					vj					vd				
VMADDWOD.H.BU **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	1	0	1	1	0	0	vk					vj					vd				
VMADDWOD.W.HU **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	1	0	1	1	0	1	vk					vj					vd				
VMADDWOD.D.WU **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	1	0	1	1	1	0	vk					vj					vd				
VMADDWOD.Q.DU **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	1	0	1	1	1	1	vk					vj					vd				
VMADDWEV.H.BU.B **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	1	1	1	0	0	0	vk					vj					vd				
VMADDWEV.W.HU.H **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	1	1	1	0	0	1	vk					vj					vd				
VMADDWEV.D.WU.W **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	1	1	1	0	1	0	vk					vj					vd				
VMADDWEV.Q.DU.D **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	1	1	1	0	1	1	vk					vj					vd				
VMADDWOD.H.BU.B **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	1	1	1	1	0	0	vk					vj					vd				
VMADDWOD.W.HU.H **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	1	1	1	1	0	1	vk					vj					vd				
VMADDWOD.D.WU.W **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	1	1	1	1	1	0	vk					vj					vd				
VMADDWOD.Q.DU.D **vd, vj, vk**	0	1	1	1	0	0	0	0	1	0	1	1	1	1	1	1	1	vk					vj					vd				
VDIV.B **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	0	0	0	0	0	vk					vj					vd				
VDIV.H **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	0	0	0	0	1	vk					vj					vd				
VDIV.W **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	0	0	0	1	0	vk					vj					vd				
VDIV.D **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	0	0	0	1	1	vk					vj					vd				
VMOD.B **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	0	0	1	0	0	vk					vj					vd				
VMOD.H **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	0	0	1	0	1	vk					vj					vd				
VMOD.W **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	0	0	1	1	0	vk					vj					vd				
VMOD.D **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	0	0	1	1	1	vk					vj					vd				
VDIV.BU **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	0	1	0	0	0	vk					vj					vd				
VDIV.HU **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	0	1	0	0	1	vk					vj					vd				
VDIV.WU **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	0	1	0	1	0	vk					vj					vd				
VDIV.DU **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	0	1	0	1	1	vk					vj					vd				
VMOD.BU **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	0	1	1	0	0	vk					vj					vd				
VMOD.HU **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	0	1	1	0	1	vk					vj					vd				
VMOD.WU **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	0	1	1	1	0	vk					vj					vd				
VMOD.DU **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	0	1	1	1	1	vk					vj					vd				
VSLL.B **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	1	0	0	0	0	vk					vj					vd				
VSLL.H **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	1	0	0	0	1	vk					vj					vd				
VSLL.W **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	1	0	0	1	0	vk					vj					vd				
VSLL.D **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	1	0	0	1	1	vk					vj					vd				
VSRL.B **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	1	0	1	0	0	vk					vj					vd				
VSRL.H **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	1	0	1	0	1	vk					vj					vd				
VSRL.W **vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	1	0	1	1	0	vk					vj					vd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VSRL.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	1	0	1	1	1	vk					vj					vd				
VSRA.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	1	1	0	0	0	vk					vj					vd				
VSRA.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	1	1	0	0	1	vk					vj					vd				
VSRA.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	1	1	0	1	0	vk					vj					vd				
VSRA.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	1	1	0	1	1	vk					vj					vd				
VROTR.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	1	1	1	0	0	vk					vj					vd				
VROTR.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	1	1	1	0	1	vk					vj					vd				
VROTR.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	1	1	1	1	0	vk					vj					vd				
VROTR.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	0	1	1	1	1	1	vk					vj					vd				
VSRLR.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	0	vk					vj					vd				
VSRLR.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	vk					vj					vd				
VSRLR.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	0	0	0	1	0	vk					vj					vd				
VSRLR.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	0	0	0	1	1	vk					vj					vd				
VSRAR.B	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	0	0	1	0	0	vk					vj					vd				
VSRAR.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	0	0	1	0	1	vk					vj					vd				
VSRAR.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	0	0	1	1	0	vk					vj					vd				
VSRAR.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	0	0	1	1	1	vk					vj					vd				
VSRLN.B.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	0	1	0	0	1	vk					vj					vd				
VSRLN.H.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	0	1	0	1	0	vk					vj					vd				
VSRLN.W.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	0	1	0	1	1	vk					vj					vd				
VSRAN.B.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	0	1	1	0	1	vk					vj					vd				
VSRAN.H.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	0	1	1	1	0	vk					vj					vd				
VSRAN.W.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	0	1	1	1	1	vk					vj					vd				
VSRLRN.B.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	1	0	0	0	1	vk					vj					vd				
VSRLRN.H.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	1	0	0	1	0	vk					vj					vd				
VSRLRN.W.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	1	0	0	1	1	vk					vj					vd				
VSRARN.B.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	1	0	1	0	1	vk					vj					vd				
VSRARN.H.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	1	0	1	1	0	vk					vj					vd				
VSRARN.W.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	1	0	1	1	1	vk					vj					vd				
VSSRLN.B.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	1	1	0	0	1	vk					vj					vd				
VSSRLN.H.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	1	1	0	1	0	vk					vj					vd				
VSSRLN.W.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	1	1	0	1	1	vk					vj					vd				
VSSRAN.B.H	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	1	1	1	0	1	vk					vj					vd				
VSSRAN.H.W	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	1	1	1	1	0	vk					vj					vd				
VSSRAN.W.D	**vd, vj, vk**	0	1	1	1	0	0	0	0	1	1	1	1	1	1	1	1	1	vk					vj					vd				
VSSRLRN.B.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	1	vk					vj					vd				
VSSRLRN.H.W	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	vk					vj					vd				
VSSRLRN.W.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	1	1	vk					vj					vd				
VSSRARN.B.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	1	1	vk					vj					vd				
VSSRARN.H.W	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	vk					vj					vd				
VSSRARN.W.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	1	1	vk					vj					vd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VSSRLN.BU.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	0	1	0	0	1	vk					vj						vd			
VSSRLN.HU.W	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	0	1	0	1	0	vk					vj						vd			
VSSRLN.WU.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	0	1	0	1	1	vk					vj						vd			
VSSRAN.BU.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	0	1	1	0	1	vk					vj						vd			
VSSRAN.HU.W	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	0	1	1	1	0	vk					vj						vd			
VSSRAN.WU.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	0	1	1	1	1	vk					vj						vd			
VSSRLRN.BU.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	1	0	0	0	1	vk					vj						vd			
VSSRLRN.HU.W	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	1	0	0	1	0	vk					vj						vd			
VSSRLRN.WU.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	1	0	0	1	1	vk					vj						vd			
VSSRARN.BU.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	1	0	1	0	1	vk					vj						vd			
VSSRARN.HU.W	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	1	0	1	1	0	vk					vj						vd			
VSSRARN.WU.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	1	0	1	1	1	vk					vj						vd			
VBITCLR.B	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	1	1	0	0	0	vk					vj						vd			
VBITCLR.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	1	1	0	0	1	vk					vj						vd			
VBITCLR.W	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	1	1	0	1	0	vk					vj						vd			
VBITCLR.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	1	1	0	1	1	vk					vj						vd			
VBITSET.B	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	1	1	1	0	0	vk					vj						vd			
VBITSET.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	1	1	1	0	1	vk					vj						vd			
VBITSET.W	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	1	1	1	1	0	vk					vj						vd			
VBITSET.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	0	1	1	1	1	1	vk					vj						vd			
VBITREV.B	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	0	0	0	0	0	vk					vj						vd			
VBITREV.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	0	0	0	0	1	vk					vj						vd			
VBITREV.W	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	0	0	0	1	0	vk					vj						vd			
VBITREV.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	0	0	0	1	1	vk					vj						vd			
VPACKEV.B	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	0	1	1	0	0	vk					vj						vd			
VPACKEV.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	0	1	1	0	1	vk					vj						vd			
VPACKEV.W	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	0	1	1	1	0	vk					vj						vd			
VPACKEV.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	0	1	1	1	1	vk					vj						vd			
VPACKOD.B	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	1	0	0	0	0	vk					vj						vd			
VPACKOD.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	1	0	0	0	1	vk					vj						vd			
VPACKOD.W	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	1	0	0	1	0	vk					vj						vd			
VPACKOD.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	1	0	0	1	1	vk					vj						vd			
VILVL.B	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	1	0	1	0	0	vk					vj						vd			
VILVL.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	1	0	1	0	1	vk					vj						vd			
VILVL.W	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	1	0	1	1	0	vk					vj						vd			
VILVL.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	1	0	1	1	1	vk					vj						vd			
VILVH.B	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	1	1	0	0	0	vk					vj						vd			
VILVH.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	1	1	0	0	1	vk					vj						vd			
VILVH.W	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	1	1	0	1	0	vk					vj						vd			
VILVH.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	1	1	0	1	1	vk					vj						vd			
VPICKEV.B	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	1	1	1	0	0	vk					vj						vd			

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VPICKEV.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	1	1	1	0	1	vk					vj					vd				
VPICKEV.W	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	1	1	1	1	0	vk					vj					vd				
VPICKEV.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	0	1	1	1	1	1	1	vk					vj					vd				
VPICKOD.B	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	0	0	0	0	0	vk					vj					vd				
VPICKOD.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	0	0	0	0	1	vk					vj					vd				
VPICKOD.W	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	0	0	0	1	0	vk					vj					vd				
VPICKOD.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	0	0	0	1	1	vk					vj					vd				
VREPLVE.B	**vd, vj, rk**	0	1	1	1	0	0	0	1	0	0	1	0	0	0	1	0	0	rk					vj					vd				
VREPLVE.H	**vd, vj, rk**	0	1	1	1	0	0	0	1	0	0	1	0	0	0	1	0	1	rk					vj					vd				
VREPLVE.W	**vd, vj, rk**	0	1	1	1	0	0	0	1	0	0	1	0	0	0	1	1	0	rk					vj					vd				
VREPLVE.D	**vd, vj, rk**	0	1	1	1	0	0	0	1	0	0	1	0	0	0	1	1	1	rk					vj					vd				
VAND.V	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	0	1	1	0	0	vk					vj					vd				
VOR.V	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	0	1	1	0	1	vk					vj					vd				
VXOR.V	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	0	1	1	1	0	vk					vj					vd				
VNOR.V	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	0	1	1	1	1	vk					vj					vd				
VANDN.V	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	1	0	0	0	0	vk					vj					vd				
VORN.V	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	1	0	0	0	1	vk					vj					vd				
VFRSTP.B	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	1	0	1	1	0	vk					vj					vd				
VFRSTP.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	1	0	1	1	1	vk					vj					vd				
VADD.Q	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	1	1	0	1	0	vk					vj					vd				
VSUB.Q	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	1	1	0	1	1	vk					vj					vd				
VSIGNCOV.B	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	1	1	1	0	0	vk					vj					vd				
VSIGNCOV.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	1	1	1	0	1	vk					vj					vd				
VSIGNCOV.W	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	1	1	1	1	0	vk					vj					vd				
VSIGNCOV.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	0	1	1	1	1	1	vk					vj					vd				
VFADD.S	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	1	0	0	0	0	1	vk					vj					vd				
VFADD.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	1	0	0	0	1	0	vk					vj					vd				
VFSUB.S	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	1	0	0	1	0	1	vk					vj					vd				
VFSUB.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	1	0	0	1	1	0	vk					vj					vd				
VFMUL.S	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	1	1	0	0	0	1	vk					vj					vd				
VFMUL.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	1	1	0	0	1	0	vk					vj					vd				
VFDIV.S	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	1	1	0	1	0	1	vk					vj					vd				
VFDIV.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	1	1	0	1	1	0	vk					vj					vd				
VFMAX.S	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	1	1	1	0	0	1	vk					vj					vd				
VFMAX.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	1	1	1	0	1	0	vk					vj					vd				
VFMIN.S	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	1	1	1	1	0	1	vk					vj					vd				
VFMIN.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	0	1	1	1	1	1	1	0	vk					vj					vd				
VFMAXA.S	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	1	0	0	0	0	0	0	1	vk					vj					vd				
VFMAXA.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	1	0	0	0	0	0	1	0	vk					vj					vd				
VFMINA.S	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	1	0	0	0	0	1	0	1	vk					vj					vd				
VFMINA.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	1	0	0	0	0	1	1	0	vk					vj					vd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VFCVT.H.S	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	1	0	0	0	1	1	0	0	vk					vj					vd				
VFCVT.S.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	1	0	0	0	1	1	0	1	vk					vj					vd				
VFFINT.S.L	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	1	0	0	1	0	0	0	0	vk					vj					vd				
VFTINT.W.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	1	0	0	1	0	0	1	1	vk					vj					vd				
VFTINTRM.W.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	1	0	0	1	0	1	0	0	vk					vj					vd				
VFTINTRP.W.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	1	0	0	1	0	1	0	1	vk					vj					vd				
VFTINTRZ.W.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	1	0	0	1	0	1	1	0	vk					vj					vd				
VFTINTRNE.W.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	1	0	0	1	0	1	1	1	vk					vj					vd				
VSHUF.H	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	1	1	1	1	0	1	0	1	vk					vj					vd				
VSHUF.W	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	1	1	1	1	0	1	1	0	vk					vj					vd				
VSHUF.D	**vd, vj, vk**	0	1	1	1	0	0	0	1	0	1	1	1	1	0	1	1	1	vk					vj					vd				
VSEQI.B	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	si5					vj					vd				
VSEQI.H	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	0	0	0	0	0	1	si5					vj					vd				
VSEQI.W	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	0	0	0	0	1	0	si5					vj					vd				
VSEQI.D	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	0	0	0	0	1	1	si5					vj					vd				
VSLEI.B	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	0	0	0	1	0	0	si5					vj					vd				
VSLEI.H	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	0	0	0	1	0	1	si5					vj					vd				
VSLEI.W	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	0	0	0	1	1	0	si5					vj					vd				
VSLEI.D	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	0	0	0	1	1	1	si5					vj					vd				
VSLEI.BU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	0	1	0	0	0	ui5					vj					vd				
VSLEI.HU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	0	1	0	0	1	ui5					vj					vd				
VSLEI.WU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	0	1	0	1	0	ui5					vj					vd				
VSLEI.DU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	0	1	0	1	1	ui5					vj					vd				
VSLTI.B	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	0	0	1	1	0	0	si5					vj					vd				
VSLTI.H	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	0	0	1	1	0	1	si5					vj					vd				
VSLTI.W	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	0	0	1	1	1	0	si5					vj					vd				
VSLTI.D	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	0	0	1	1	1	1	si5					vj					vd				
VSLTI.BU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	1	0	0	0	0	ui5					vj					vd				
VSLTI.HU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	1	0	0	0	1	ui5					vj					vd				
VSLTI.WU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	1	0	0	1	0	ui5					vj					vd				
VSLTI.DU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	1	0	0	1	1	ui5					vj					vd				
VADDI.BU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	1	0	1	0	0	ui5					vj					vd				
VADDI.HU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	1	0	1	0	1	ui5					vj					vd				
VADDI.WU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	1	0	1	1	0	ui5					vj					vd				
VADDI.DU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	1	0	1	1	1	ui5					vj					vd				
VSUBI.BU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	1	1	0	0	0	ui5					vj					vd				
VSUBI.HU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	1	1	0	0	1	ui5					vj					vd				
VSUBI.WU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	1	1	0	1	0	ui5					vj					vd				
VSUBI.DU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	1	1	0	1	1	ui5					vj					vd				
VBSLL.V	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	1	1	1	0	0	ui5					vj					vd				
VBSRL.V	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	0	1	1	1	0	1	ui5					vj					vd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VMAXI.B	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	1	0	0	0	0	0	si5					vj					vd				
VMAXI.H	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	1	0	0	0	0	1	si5					vj					vd				
VMAXI.W	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	1	0	0	0	1	0	si5					vj					vd				
VMAXI.D	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	1	0	0	0	1	1	si5					vj					vd				
VMINI.B	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	1	0	0	1	0	0	si5					vj					vd				
VMINI.H	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	1	0	0	1	0	1	si5					vj					vd				
VMINI.W	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	1	0	0	1	1	0	si5					vj					vd				
VMINI.D	**vd, vj, si5**	0	1	1	1	0	0	1	0	1	0	0	1	0	0	1	1	1	si5					vj					vd				
VMAXI.BU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	1	0	1	0	0	0	ui5					vj					vd				
VMAXI.HU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	1	0	1	0	0	1	ui5					vj					vd				
VMAXI.WU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	1	0	1	0	1	0	ui5					vj					vd				
VMAXI.DU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	1	0	1	0	1	1	ui5					vj					vd				
VMINI.BU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	1	0	1	1	0	0	ui5					vj					vd				
VMINI.HU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	1	0	1	1	0	1	ui5					vj					vd				
VMINI.WU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	1	0	1	1	1	0	ui5					vj					vd				
VMINI.DU	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	1	0	1	1	1	1	ui5					vj					vd				
VFRSTPI.B	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	1	1	0	1	0	0	ui5					vj					vd				
VFRSTPI.H	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	0	1	1	0	1	0	1	ui5					vj					vd				
VCLO.B	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	0	0	0	vj					vd				
VCLO.H	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	0	0	1	vj					vd				
VCLO.W	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	0	1	0	vj					vd				
VCLO.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	0	1	1	vj					vd				
VCLZ.B	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	0	0	vj					vd				
VCLZ.H	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	0	1	vj					vd				
VCLZ.W	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	1	0	vj					vd				
VCLZ.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	1	1	vj					vd				
VPCNT.B	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	0	0	vj					vd				
VPCNT.H	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	0	1	vj					vd				
VPCNT.W	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	0	1	vj					vd				
VPCNT.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	0	1	vj					vd				
VNEG.B	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	1	0	vj					vd				
VNEG.H	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	1	1	vj					vd				
VNEG.W	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	1	0	vj					vd				
VNEG.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	1	1	vj					vd				
VMSKLTZ.B	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	0	0	vj					vd				
VMSKLTZ.H	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	0	1	vj					vd				
VMSKLTZ.W	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	0	0	vj					vd				
VMSKLTZ.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	0	1	vj					vd				
VMSKGEZ.B	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	0	0	vj					vd				
VMSKNZ.B	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	0	0	1	1	0	vj					vd				
VSETEQZ.V	**cd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	0	1	0	0	1	1	0	vj				0	0	cd		

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VSETNEZ.V	**cd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	0	0	1	1	1	vj					0	0	cd		
VSETANYEQZ.B	**cd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	0	1	0	0	0	vj					0	0	cd		
VSETANYEQZ.H	**cd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	0	1	0	0	1	vj					0	0	cd		
VSETANYEQZ.W	**cd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	0	1	0	1	0	vj					0	0	cd		
VSETANYEQZ.D	**cd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	0	1	0	1	1	vj					0	0	cd		
VSETALLNEZ.B	**cd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	0	1	1	0	0	vj					0	0	cd		
VSETALLNEZ.H	**cd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	0	1	1	0	1	vj					0	0	cd		
VSETALLNEZ.W	**cd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1	0	vj					0	0	cd		
VSETALLNEZ.D	**cd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1	1	vj					0	0	cd		
VFLOGB.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	1	0	0	0	1	vj					vd				
VFLOGB.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	1	0	0	1	0	vj					vd				
VFCLASS.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	1	0	1	0	1	vj					vd				
VFCLASS.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	1	0	1	1	0	vj					vd				
VFSQRT.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	vj					vd				
VFSQRT.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	vj					vd				
VFRECIP.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	1	1	1	0	1	vj					vd				
VFRECIP.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	1	1	1	1	0	vj					vd				
VFRSQRT.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	0	0	0	0	0	1	vj					vd				
VFRSQRT.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	0	0	0	0	1	0	vj					vd				
VFRECIPE.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	0	0	0	1	0	1	vj					vd				
VFRECIPE.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	0	0	0	1	1	0	vj					vd				
VFRSQRTE.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	0	0	1	0	0	1	vj					vd				
VFRSQRTE.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	0	0	1	0	1	0	vj					vd				
VFRINT.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	0	0	1	1	0	1	vj					vd				
VFRINT.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	0	0	1	1	1	0	vj					vd				
VFRINTRM.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	0	1	0	0	0	1	vj					vd				
VFRINTRM.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	0	1	0	0	1	0	vj					vd				
VFRINTRP.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	0	1	0	1	0	1	vj					vd				
VFRINTRP.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	0	1	0	1	1	0	vj					vd				
VFRINTRZ.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	0	1	1	0	0	1	vj					vd				
VFRINTRZ.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	0	1	1	0	1	0	vj					vd				
VFRINTRNE.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	0	1	1	1	0	1	vj					vd				
VFRINTRNE.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	0	1	1	1	1	0	vj					vd				
VFCVTL.S.H	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	1	1	1	0	1	0	vj					vd				
VFCVTH.S.H	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	1	1	1	0	1	1	vj					vd				
VFCVTL.D.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	1	1	1	1	0	0	vj					vd				
VFCVTH.D.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	1	1	1	1	0	1	vj					vd				
VFFINT.S.W	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	0	0	0	0	0	vj					vd				
VFFINT.S.WU	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	0	0	0	0	1	vj					vd				
VFFINT.D.L	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	0	0	0	1	0	vj					vd				
VFFINT.D.LU	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	0	0	0	1	1	vj					vd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VFFINTL.D.W	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	0	0	1	0	0	vj					vd				
VFFINTH.D.W	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	0	0	1	0	1	vj					vd				
VFTINT.W.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	0	1	1	0	0	vj					vd				
VFTINT.L.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	0	1	1	0	1	vj					vd				
VFTINTRM.W.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	0	1	1	1	0	vj					vd				
VFTINTRM.L.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	0	1	1	1	1	vj					vd				
VFTINTRP.W.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	1	0	0	0	0	vj					vd				
VFTINTRP.L.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	1	0	0	0	1	vj					vd				
VFTINTRZ.W.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	1	0	0	1	0	vj					vd				
VFTINTRZ.L.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	1	0	0	1	1	vj					vd				
VFTINTRNE.W.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	1	0	1	0	0	vj					vd				
VFTINTRNE.L.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	1	0	1	0	1	vj					vd				
VFTINT.WU.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	1	0	1	1	0	vj					vd				
VFTINT.LU.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	1	0	1	1	1	vj					vd				
VFTINTRZ.WU.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	1	1	1	0	0	vj					vd				
VFTINTRZ.LU.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	0	1	1	1	0	1	vj					vd				
VFTINTL.L.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	0	0	0	0	0	vj					vd				
VFTINTH.L.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	0	0	0	0	1	vj					vd				
VFTINTRML.L.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	0	0	0	1	0	vj					vd				
VFTINTRMH.L.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	0	0	0	1	1	vj					vd				
VFTINTRPL.L.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	0	0	1	0	0	vj					vd				
VFTINTRPH.L.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	0	0	1	0	1	vj					vd				
VFTINTRZL.L.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	0	0	1	1	0	vj					vd				
VFTINTRZH.L.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	0	0	1	1	1	vj					vd				
VFTINTRNEL.L.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	0	1	0	0	0	vj					vd				
VFTINTRNEH.L.S	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	0	1	0	0	1	vj					vd				
VEXTH.H.B	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	1	1	0	0	0	vj					vd				
VEXTH.W.H	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	1	1	0	0	1	vj					vd				
VEXTH.D.W	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	1	1	0	1	0	vj					vd				
VEXTH.Q.D	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	1	1	0	1	1	vj					vd				
VEXTH.HU.BU	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	1	1	1	0	0	vj					vd				
VEXTH.WU.HU	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	1	1	1	0	1	vj					vd				
VEXTH.DU.WU	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	1	1	1	1	0	vj					vd				
VEXTH.QU.DU	**vd, vj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	1	1	1	1	1	vj					vd				
VREPLGR2VR.B	**vd, rj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	rj					vd				
VREPLGR2VR.H	**vd, rj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	1	0	0	0	0	0	1	rj					vd				
VREPLGR2VR.W	**vd, rj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	1	0	0	0	0	1	0	rj					vd				
VREPLGR2VR.D	**vd, rj**	0	1	1	1	0	0	1	0	1	0	0	1	1	1	1	1	0	0	0	0	1	1	rj					vd				
VROTRI.B	**vd, vj, ui3**	0	1	1	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	ui3			vj					vd				
VROTRI.H	**vd, vj, ui4**	0	1	1	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	1	ui4			vj					vd				
VROTRI.W	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	1	0	0	0	0	0	0	1	ui5				vj					vd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VROTRI.D	**vd, vj, ui6**	0	1	1	1	0	0	1	0	1	0	1	0	0	0	0	1	ui6						vj					vd				
VSRLRI.B	**vd, vj, ui3**	0	1	1	1	0	0	1	0	1	0	1	0	0	1	0	0	0	0	1	ui3			vj					vd				
VSRLRI.H	**vd, vj, ui4**	0	1	1	1	0	0	1	0	1	0	1	0	0	1	0	0	0	1	ui4				vj					vd				
VSRLRI.W	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	1	0	0	1	0	0	1	ui5					vj					vd				
VSRLRI.D	**vd, vj, ui6**	0	1	1	1	0	0	1	0	1	0	1	0	0	1	0	1	ui6						vj					vd				
VSRARI.B	**vd, vj, ui3**	0	1	1	1	0	0	1	0	1	0	1	0	1	0	0	0	0	0	1	ui3			vj					vd				
VSRARI.H	**vd, vj, ui4**	0	1	1	1	0	0	1	0	1	0	1	0	1	0	0	0	0	1	ui4				vj					vd				
VSRARI.W	**vd, vj, ui5**	0	1	1	1	0	0	1	0	1	0	1	0	1	0	0	0	1	ui5					vj					vd				
VSRARI.D	**vd, vj, ui6**	0	1	1	1	0	0	1	0	1	0	1	0	1	0	0	1	ui6						vj					vd				
VINSGR2VR.B	**vd, rj, ui4**	0	1	1	1	0	0	1	0	1	1	1	0	1	0	1	1	1	0	ui4				rj					vd				
VINSGR2VR.H	**vd, rj, ui3**	0	1	1	1	0	0	1	0	1	1	1	0	1	0	1	1	1	1	0	ui3			rj					vd				
VINSGR2VR.W	**vd, rj, ui2**	0	1	1	1	0	0	1	0	1	1	1	0	1	0	1	1	1	1	1	0	ui2		rj					vd				
VINSGR2VR.D	**vd, rj, ui1**	0	1	1	1	0	0	1	0	1	1	1	0	1	0	1	1	1	1	1	1	0	ui1	rj					vd				
VPICKVE2GR.B	**rd, vj, ui4**	0	1	1	1	0	0	1	0	1	1	1	0	1	1	1	1	1	0	ui4				vj					rd				
VPICKVE2GR.H	**rd, vj, ui3**	0	1	1	1	0	0	1	0	1	1	1	0	1	1	1	1	1	1	0	ui3			vj					rd				
VPICKVE2GR.W	**rd, vj, ui2**	0	1	1	1	0	0	1	0	1	1	1	0	1	1	1	1	1	1	1	0	ui2		vj					rd				
VPICKVE2GR.D	**rd, vj, ui1**	0	1	1	1	0	0	1	0	1	1	1	0	1	1	1	1	1	1	1	1	0	ui1	vj					rd				
VPICKVE2GR.BU	**rd, vj, ui4**	0	1	1	1	0	0	1	0	1	1	1	1	0	0	1	1	1	0	ui4				vj					rd				
VPICKVE2GR.HU	**rd, vj, ui3**	0	1	1	1	0	0	1	0	1	1	1	1	0	0	1	1	1	1	0	ui3			vj					rd				
VPICKVE2GR.WU	**rd, vj, ui2**	0	1	1	1	0	0	1	0	1	1	1	1	0	0	1	1	1	1	1	0	ui2		vj					rd				
VPICKVE2GR.DU	**rd, vj, ui1**	0	1	1	1	0	0	1	0	1	1	1	1	0	0	1	1	1	1	1	1	0	ui1	vj					rd				
VREPLVEI.B	**vd, vj, ui4**	0	1	1	1	0	0	1	0	1	1	1	1	0	1	1	1	1	0	ui4				vj					vd				
VREPLVEI.H	**vd, vj, ui3**	0	1	1	1	0	0	1	0	1	1	1	1	0	1	1	1	1	1	0	ui3			vj					vd				
VREPLVEI.W	**vd, vj, ui2**	0	1	1	1	0	0	1	0	1	1	1	1	0	1	1	1	1	1	1	0	ui2		vj					vd				
VREPLVEI.D	**vd, vj, ui1**	0	1	1	1	0	0	1	0	1	1	1	1	0	1	1	1	1	1	1	1	0	ui1	vj					vd				
VSLLWIL.H.B	**vd, vj, ui3**	0	1	1	1	0	0	1	1	0	0	0	0	1	0	0	0	0	0	1	ui3			vj					vd				
VSLLWIL.W.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	0	0	0	1	0	0	0	0	1	ui4				vj					vd				
VSLLWIL.D.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	0	0	0	1	0	0	0	1	ui5					vj					vd				
VEXTL.Q.D	**vd, vj**	0	1	1	1	0	0	1	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	vj					vd				
VSLLWIL.HU.BU	**vd, vj, ui3**	0	1	1	1	0	0	1	1	0	0	0	0	1	1	0	0	0	0	1	ui3			vj					vd				
VSLLWIL.WU.HU	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	0	0	0	1	1	0	0	0	1	ui4				vj					vd				
VSLLWIL.DU.WU	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	0	0	0	1	1	0	0	1	ui5					vj					vd				
VEXTL.QU.DU	**vd, vj**	0	1	1	1	0	0	1	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	vj					vd				
VBITCLRI.B	**vd, vj, ui3**	0	1	1	1	0	0	1	1	0	0	0	1	0	0	0	0	0	0	1	ui3			vj					vd				
VBITCLRI.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	0	0	1	0	0	0	0	0	1	ui4				vj					vd				
VBITCLRI.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	0	0	1	0	0	0	0	1	ui5					vj					vd				
VBITCLRI.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	0	0	1	0	0	0	1	ui6						vj					vd				
VBITSETI.B	**vd, vj, ui3**	0	1	1	1	0	0	1	1	0	0	0	1	0	1	0	0	0	0	1	ui3			vj					vd				
VBITSETI.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	0	0	1	0	1	0	0	0	1	ui4				vj					vd				
VBITSETI.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	0	0	1	0	1	0	0	1	ui5					vj					vd				
VBITSETI.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	0	0	1	0	1	0	1	ui6						vj					vd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VBITREVI.B	**vd, vj, ui3**	0	1	1	1	0	0	1	1	0	0	0	1	1	0	0	0	0	0	1	ui3			vj					vd				
VBITREVI.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	0	0	1	1	0	0	0	0	1	ui4				vj					vd				
VBITREVI.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	0	0	1	1	0	0	0	1	ui5					vj					vd				
VBITREVI.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	0	0	1	1	0	0	1	ui6						vj					vd				
VSAT.B	**vd, vj, ui3**	0	1	1	1	0	0	1	1	0	0	1	0	0	1	0	0	0	0	1	ui3			vj					vd				
VSAT.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	0	1	0	0	1	0	0	0	1	ui4				vj					vd				
VSAT.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	0	1	0	0	1	0	0	1	ui5					vj					vd				
VSAT.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	0	1	0	0	1	0	1	ui6						vj					vd				
VSAT.BU	**vd, vj, ui3**	0	1	1	1	0	0	1	1	0	0	1	0	1	0	0	0	0	0	1	ui3			vj					vd				
VSAT.HU	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	0	1	0	1	0	0	0	0	1	ui4				vj					vd				
VSAT.WU	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	0	1	0	1	0	0	0	1	ui5					vj					vd				
VSAT.DU	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	0	1	0	1	0	0	1	ui6						vj					vd				
VSLLI.B	**vd, vj, ui3**	0	1	1	1	0	0	1	1	0	0	1	0	1	1	0	0	0	0	1	ui3			vj					vd				
VSLLI.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	0	1	0	1	1	0	0	0	1	ui4				vj					vd				
VSLLI.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	0	1	0	1	1	0	0	1	ui5					vj					vd				
VSLLI.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	0	1	0	1	1	0	1	ui6						vj					vd				
VSRLI.B	**vd, vj, ui3**	0	1	1	1	0	0	1	1	0	0	1	1	0	0	0	0	0	0	1	ui3			vj					vd				
VSRLI.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	0	1	1	0	0	0	0	0	1	ui4				vj					vd				
VSRLI.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	0	1	1	0	0	0	0	1	ui5					vj					vd				
VSRLI.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	0	1	1	0	0	0	1	ui6						vj					vd				
VSRAI.B	**vd, vj, ui3**	0	1	1	1	0	0	1	1	0	0	1	1	0	1	0	0	0	0	1	ui3			vj					vd				
VSRAI.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	0	1	1	0	1	0	0	0	1	ui4				vj					vd				
VSRAI.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	0	1	1	0	1	0	0	1	ui5					vj					vd				
VSRAI.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	0	1	1	0	1	0	1	ui6						vj					vd				
VSRLNI.B.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	1	0	0	0	0	0	0	0	1	ui4				vj					vd				
VSRLNI.H.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	1	0	0	0	0	0	0	1	ui5					vj					vd				
VSRLNI.W.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	1	0	0	0	0	0	1	ui6						vj					vd				
VSRLNI.D.Q	**vd, vj, ui7**	0	1	1	1	0	0	1	1	0	1	0	0	0	0	1	ui7							vj					vd				
VSRLRNI.B.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	1	0	0	0	1	0	0	0	1	ui4				vj					vd				
VSRLRNI.H.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	1	0	0	0	1	0	0	1	ui5					vj					vd				
VSRLRNI.W.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	1	0	0	0	1	0	1	ui6						vj					vd				
VSRLRNI.D.Q	**vd, vj, ui7**	0	1	1	1	0	0	1	1	0	1	0	0	0	1	1	ui7							vj					vd				
VSSRLNI.B.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	1	0	0	1	0	0	0	0	1	ui4				vj					vd				
VSSRLNI.H.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	1	0	0	1	0	0	0	1	ui5					vj					vd				
VSSRLNI.W.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	1	0	0	1	0	0	1	ui6						vj					vd				
VSSRLNI.D.Q	**vd, vj, ui7**	0	1	1	1	0	0	1	1	0	1	0	0	1	0	1	ui7							vj					vd				
VSSRLNI.BU.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	1	0	0	1	1	0	0	0	1	ui4				vj					vd				
VSSRLNI.HU.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1	ui5					vj					vd				
VSSRLNI.WU.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	1	0	0	1	1	0	1	ui6						vj					vd				
VSSRLNI.DU.Q	**vd, vj, ui7**	0	1	1	1	0	0	1	1	0	1	0	0	1	1	1	ui7							vj					vd				
VSSRLRNI.B.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	1	0	1	0	0	0	0	0	1	ui4				vj					vd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VSSRLRNI.H.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	1	0	1	0	0	0	0	1	ui5					vj						vd			
VSSRLRNI.W.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	1	0	1	0	0	0	1	ui6						vj						vd			
VSSRLRNI.D.Q	**vd, vj, ui7**	0	1	1	1	0	0	1	1	0	1	0	1	0	0	1	ui7							vj						vd			
VSSRLRNI.BU.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	1	0	1	0	1	0	0	0	1	ui4				vj						vd			
VSSRLRNI.HU.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	1	0	1	0	1	0	0	1	ui5					vj						vd			
VSSRLRNI.WU.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	1	0	1	0	1	0	1	ui6						vj						vd			
VSSRLRNI.DU.Q	**vd, vj, ui7**	0	1	1	1	0	0	1	1	0	1	0	1	0	1	1	ui7							vj						vd			
VSRANI.B.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	1	0	1	1	0	0	0	0	1	ui4				vj						vd			
VSRANI.H.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	1	0	1	1	0	0	0	1	ui5					vj						vd			
VSRANI.W.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	1	0	1	1	0	0	1	ui6						vj						vd			
VSRANI.D.Q	**vd, vj, ui7**	0	1	1	1	0	0	1	1	0	1	0	1	1	0	1	ui7							vj						vd			
VSRARNI.B.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	1	0	1	1	1	0	0	0	1	ui4				vj						vd			
VSRARNI.H.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	1	0	1	1	1	0	0	1	ui5					vj						vd			
VSRARNI.W.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	1	0	1	1	1	0	1	ui6						vj						vd			
VSRARNI.D.Q	**vd, vj, ui7**	0	1	1	1	0	0	1	1	0	1	0	1	1	1	1	ui7							vj						vd			
VSSRANI.B.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	1	1	0	0	0	0	0	0	1	ui4				vj						vd			
VSSRANI.H.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	1	1	0	0	0	0	0	1	ui5					vj						vd			
VSSRANI.W.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	1	1	0	0	0	0	1	ui6						vj						vd			
VSSRANI.D.Q	**vd, vj, ui7**	0	1	1	1	0	0	1	1	0	1	1	0	0	0	1	ui7							vj						vd			
VSSRANI.BU.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	1	1	0	0	1	0	0	0	1	ui4				vj						vd			
VSSRANI.HU.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	1	1	0	0	1	0	0	1	ui5					vj						vd			
VSSRANI.WU.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	1	1	0	0	1	0	1	ui6						vj						vd			
VSSRANI.DU.Q	**vd, vj, ui7**	0	1	1	1	0	0	1	1	0	1	1	0	0	1	1	ui7							vj						vd			
VSSRARNI.B.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	1	1	0	1	0	0	0	0	1	ui4				vj						vd			
VSSRARNI.H.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	1	1	0	1	0	0	0	1	ui5					vj						vd			
VSSRARNI.W.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	1	1	0	1	0	0	1	ui6						vj						vd			
VSSRARNI.D.Q	**vd, vj, ui7**	0	1	1	1	0	0	1	1	0	1	1	0	1	0	1	ui7							vj						vd			
VSSRARNI.BU.H	**vd, vj, ui4**	0	1	1	1	0	0	1	1	0	1	1	0	1	1	0	0	0	1	ui4				vj						vd			
VSSRARNI.HU.W	**vd, vj, ui5**	0	1	1	1	0	0	1	1	0	1	1	0	1	1	0	0	1	ui5					vj						vd			
VSSRARNI.WU.D	**vd, vj, ui6**	0	1	1	1	0	0	1	1	0	1	1	0	1	1	0	1	ui6						vj						vd			
VSSRARNI.DU.Q	**vd, vj, ui7**	0	1	1	1	0	0	1	1	0	1	1	0	1	1	1	ui7							vj						vd			
VEXTRINS.D	**vd, vj, ui8**	0	1	1	1	0	0	1	1	1	0	0	0	0	0	ui8								vj						vd			
VEXTRINS.W	**vd, vj, ui8**	0	1	1	1	0	0	1	1	1	0	0	0	0	1	ui8								vj						vd			
VEXTRINS.H	**vd, vj, ui8**	0	1	1	1	0	0	1	1	1	0	0	0	1	0	ui8								vj						vd			
VEXTRINS.B	**vd, vj, ui8**	0	1	1	1	0	0	1	1	1	0	0	0	1	1	ui8								vj						vd			
VSHUF4I.B	**vd, vj, ui8**	0	1	1	1	0	0	1	1	1	0	0	1	0	0	ui8								vj						vd			
VSHUF4I.H	**vd, vj, ui8**	0	1	1	1	0	0	1	1	1	0	0	1	0	1	ui8								vj						vd			
VSHUF4I.W	**vd, vj, ui8**	0	1	1	1	0	0	1	1	1	0	0	1	1	0	ui8								vj						vd			
VSHUF4I.D	**vd, vj, ui8**	0	1	1	1	0	0	1	1	1	0	0	1	1	1	ui8								vj						vd			
VBITSELI.B	**vd, vj, ui8**	0	1	1	1	0	0	1	1	1	1	0	0	0	1	ui8								vj						vd			
VANDI.B	**vd, vj, ui8**	0	1	1	1	0	0	1	1	1	1	0	1	0	0	ui8								vj						vd			

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VORI.B	**vd, vj, ui8**	0	1	1	1	0	0	1	1	1	1	0	1	0	1	ui8								vj					vd				
VXORI.B	**vd, vj, ui8**	0	1	1	1	0	0	1	1	1	1	0	1	1	0	ui8								vj					vd				
VNORI.B	**vd, vj, ui8**	0	1	1	1	0	0	1	1	1	1	0	1	1	1	ui8								vj					vd				
VLDI	**vd, i13**	0	1	1	1	0	0	1	1	1	1	1	0	0	0	i13													vd				
VPERMI.W	**vd, vj, ui8**	0	1	1	1	0	0	1	1	1	1	1	0	0	1	ui8								vj					vd				
XVSEQ.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	xk						xj					xd				
XVSEQ.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	1	xk						xj					xd				
XVSEQ.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	0	0	0	1	0	xk						xj					xd				
XVSEQ.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	0	0	0	1	1	xk						xj					xd				
XVSLE.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	0	0	1	0	0	xk						xj					xd				
XVSLE.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	0	0	1	0	1	xk						xj					xd				
XVSLE.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	0	0	1	1	0	xk						xj					xd				
XVSLE.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	0	0	1	1	1	xk						xj					xd				
XVSLE.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	0	1	0	0	0	xk						xj					xd				
XVSLE.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	0	1	0	0	1	xk						xj					xd				
XVSLE.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	0	1	0	1	0	xk						xj					xd				
XVSLE.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	0	1	0	1	1	xk						xj					xd				
XVSLT.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	0	1	1	0	0	xk						xj					xd				
XVSLT.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	0	1	1	0	1	xk						xj					xd				
XVSLT.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	0	1	1	1	0	xk						xj					xd				
XVSLT.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	0	1	1	1	1	xk						xj					xd				
XVSLT.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	1	0	0	0	0	xk						xj					xd				
XVSLT.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	1	0	0	0	1	xk						xj					xd				
XVSLT.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	1	0	0	1	0	xk						xj					xd				
XVSLT.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	1	0	0	1	1	xk						xj					xd				
XVADD.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	1	0	1	0	0	xk						xj					xd				
XVADD.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	1	0	1	0	1	xk						xj					xd				
XVADD.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	1	0	1	1	0	xk						xj					xd				
XVADD.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	1	0	1	1	1	xk						xj					xd				
XVSUB.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	1	1	0	0	0	xk						xj					xd				
XVSUB.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	1	1	0	0	1	xk						xj					xd				
XVSUB.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	1	1	0	1	0	xk						xj					xd				
XVSUB.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	1	1	0	1	1	xk						xj					xd				
XVADDWEV.H.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	1	1	1	1	0	xk						xj					xd				
XVADDWEV.W.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	1	1	1	1	1	xk						xj					xd				
XVADDWEV.D.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	1	1	1	1	0	xk						xj					xd				
XVADDWEV.Q.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	0	1	1	1	1	1	xk						xj					xd				
XVSUBWEV.H.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	0	0	0	0	0	xk						xj					xd				
XVSUBWEV.W.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	0	0	0	0	1	xk						xj					xd				
XVSUBWEV.D.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	0	0	0	0	0	xk						xj					xd				
XVSUBWEV.Q.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	0	0	0	0	1	xk						xj					xd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
XVADDWOD.H.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	0	0	0	1	0	0	xk					xj					xd				
XVADDWOD.W.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	0	0	0	1	0	1	xk					xj					xd				
XVADDWOD.D.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	0	0	0	1	1	0	xk					xj					xd				
XVADDWOD.Q.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	0	0	0	1	1	1	xk					xj					xd				
XVSUBWOD.H.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	0	0	1	0	0	0	xk					xj					xd				
XVSUBWOD.W.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	0	0	1	0	0	1	xk					xj					xd				
XVSUBWOD.D.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	0	0	1	0	1	0	xk					xj					xd				
XVSUBWOD.Q.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	0	0	1	0	1	1	xk					xj					xd				
XVADDWEV.H.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	0	1	1	1	0	0	xk					xj					xd				
XVADDWEV.W.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	0	1	1	1	0	1	xk					xj					xd				
XVADDWEV.D.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	0	1	1	1	1	0	xk					xj					xd				
XVADDWEV.Q.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	0	1	1	1	1	1	xk					xj					xd				
XVSUBWEV.H.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	1	0	0	0	0	0	xk					xj					xd				
XVSUBWEV.W.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	1	0	0	0	0	1	xk					xj					xd				
XVSUBWEV.D.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	1	0	0	0	1	0	xk					xj					xd				
XVSUBWEV.Q.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	1	0	0	0	1	1	xk					xj					xd				
XVADDWOD.H.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	1	0	0	1	0	0	xk					xj					xd				
XVADDWOD.W.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	1	0	0	1	0	1	xk					xj					xd				
XVADDWOD.D.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	1	0	0	1	1	0	xk					xj					xd				
XVADDWOD.Q.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	1	0	0	1	1	1	xk					xj					xd				
XVSUBWOD.H.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	1	0	1	0	0	0	xk					xj					xd				
XVSUBWOD.W.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	1	0	1	0	0	1	xk					xj					xd				
XVSUBWOD.D.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	1	0	1	0	1	0	xk					xj					xd				
XVSUBWOD.Q.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	1	0	1	0	1	1	xk					xj					xd				
XVADDWEV.H.BU.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	1	1	1	1	0	0	xk					xj					xd				
XVADDWEV.W.HU.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	1	1	1	1	0	1	xk					xj					xd				
XVADDWEV.D.WU.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	1	1	1	1	1	0	xk					xj					xd				
XVADDWEV.Q.DU.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	0	1	1	1	1	1	1	1	xk					xj					xd				
XVADDWOD.H.BU.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	0	0	0	0	0	xk					xj					xd				
XVADDWOD.W.HU.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	0	0	0	0	1	xk					xj					xd				
XVADDWOD.D.WU.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	0	0	0	1	0	xk					xj					xd				
XVADDWOD.Q.DU.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	0	0	0	1	1	xk					xj					xd				
XVSADD.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	0	1	1	0	0	xk					xj					xd				
XVSADD.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	0	1	1	0	1	xk					xj					xd				
XVSADD.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	0	1	1	1	0	xk					xj					xd				
XVSADD.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	0	1	1	1	1	xk					xj					xd				
XVSSUB.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	1	0	0	0	0	xk					xj					xd				
XVSSUB.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	1	0	0	0	1	xk					xj					xd				
XVSSUB.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	1	0	0	1	0	xk					xj					xd				
XVSSUB.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	1	0	0	1	1	xk					xj					xd				
XVSADD.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	1	0	1	0	0	xk					xj					xd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
XVSADD.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	1	0	1	0	1	xk					xj					xd				
XVSADD.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	1	0	1	1	0	xk					xj					xd				
XVSADD.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	1	0	1	1	1	xk					xj					xd				
XVSSUB.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	1	1	0	0	0	xk					xj					xd				
XVSSUB.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	1	1	0	0	1	xk					xj					xd				
XVSSUB.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	1	1	0	1	0	xk					xj					xd				
XVSSUB.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	0	1	1	0	1	1	xk					xj					xd				
XVHADDW.H.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	0	1	0	0	0	xk					xj					xd				
XVHADDW.W.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	0	1	0	0	1	xk					xj					xd				
XVHADDW.D.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	0	1	0	1	0	xk					xj					xd				
XVHADDW.Q.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	0	1	0	1	1	xk					xj					xd				
XVHSUBW.H.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	xk					xj					xd				
XVHSUBW.W.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	0	1	1	0	1	xk					xj					xd				
XVHSUBW.D.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	0	1	1	1	0	xk					xj					xd				
XVHSUBW.Q.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	0	1	1	1	1	xk					xj					xd				
XVHADDW.HU.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	1	0	0	0	0	xk					xj					xd				
XVHADDW.WU.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	1	0	0	0	1	xk					xj					xd				
XVHADDW.DU.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	1	0	0	1	0	xk					xj					xd				
XVHADDW.QU.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	1	0	0	1	1	xk					xj					xd				
XVHSUBW.HU.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	1	0	1	0	0	xk					xj					xd				
XVHSUBW.WU.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	1	0	1	0	1	xk					xj					xd				
XVHSUBW.DU.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	1	0	1	1	0	xk					xj					xd				
XVHSUBW.QU.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	1	0	1	1	1	xk					xj					xd				
XVADDA.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	1	1	0	0	0	xk					xj					xd				
XVADDA.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	1	1	0	0	1	xk					xj					xd				
XVADDA.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	1	1	0	1	0	xk					xj					xd				
XVADDA.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	0	1	1	1	0	1	1	xk					xj					xd				
XVABSD.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	0	0	0	0	0	xk					xj					xd				
XVABSD.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	0	0	0	0	1	xk					xj					xd				
XVABSD.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	0	0	0	1	0	xk					xj					xd				
XVABSD.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	0	0	0	1	1	xk					xj					xd				
XVABSD.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	0	0	1	0	0	xk					xj					xd				
XVABSD.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	0	0	1	0	1	xk					xj					xd				
XVABSD.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	0	0	1	1	0	xk					xj					xd				
XVABSD.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	0	0	1	1	1	xk					xj					xd				
XVAVG.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	0	1	0	0	0	xk					xj					xd				
XVAVG.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	0	1	0	0	1	xk					xj					xd				
XVAVG.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	0	1	0	1	0	xk					xj					xd				
XVAVG.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	0	1	0	1	1	xk					xj					xd				
XVAVG.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	0	1	1	0	0	xk					xj					xd				
XVAVG.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	0	1	1	0	1	xk					xj					xd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
XVAVG.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	0	1	1	1	0	xk					xj					xd				
XVAVG.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	0	1	1	1	1	xk					xj					xd				
XVAVGR.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	1	0	0	0	0	xk					xj					xd				
XVAVGR.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	1	0	0	0	1	xk					xj					xd				
XVAVGR.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	1	0	0	1	0	xk					xj					xd				
XVAVGR.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	1	0	0	1	1	xk					xj					xd				
XVAVGR.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	1	0	1	0	0	xk					xj					xd				
XVAVGR.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	1	0	1	0	1	xk					xj					xd				
XVAVGR.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	1	0	1	1	0	xk					xj					xd				
XVAVGR.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	0	1	0	1	1	1	xk					xj					xd				
XVMAX.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	1	0	0	0	0	0	xk					xj					xd				
XVMAX.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	1	0	0	0	0	1	xk					xj					xd				
XVMAX.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	1	0	0	0	1	0	xk					xj					xd				
XVMAX.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	1	0	0	0	1	1	xk					xj					xd				
XVMIN.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	1	0	0	1	0	0	xk					xj					xd				
XVMIN.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	1	0	0	1	0	1	xk					xj					xd				
XVMIN.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	1	0	0	1	1	0	xk					xj					xd				
XVMIN.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	1	0	0	1	1	1	xk					xj					xd				
XVMAX.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	1	0	1	0	0	0	xk					xj					xd				
XVMAX.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	1	0	1	0	0	1	xk					xj					xd				
XVMAX.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	1	0	1	0	1	0	xk					xj					xd				
XVMAX.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	1	0	1	0	1	1	xk					xj					xd				
XVMIN.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	1	0	1	1	0	0	xk					xj					xd				
XVMIN.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	1	0	1	1	0	1	xk					xj					xd				
XVMIN.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	1	0	1	1	1	0	xk					xj					xd				
XVMIN.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	0	1	1	1	0	1	1	1	1	xk					xj					xd				
XVMUL.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	0	0	1	0	0	0	xk					xj					xd				
XVMUL.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	0	0	1	0	0	1	xk					xj					xd				
XVMUL.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	0	0	1	0	1	0	xk					xj					xd				
XVMUL.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	0	0	1	0	1	1	xk					xj					xd				
XVMUH.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	0	0	1	1	0	0	xk					xj					xd				
XVMUH.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	0	0	1	1	0	1	xk					xj					xd				
XVMUH.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	0	0	1	1	1	0	xk					xj					xd				
XVMUH.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	0	0	1	1	1	1	xk					xj					xd				
XVMUH.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	0	1	0	0	0	0	xk					xj					xd				
XVMUH.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	0	1	0	0	0	1	xk					xj					xd				
XVMUH.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	0	1	0	0	1	0	xk					xj					xd				
XVMUH.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	0	1	0	0	1	1	xk					xj					xd				
XVMULWEV.H.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	1	0	0	0	0	0	xk					xj					xd				
XVMULWEV.W.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	1	0	0	0	0	1	xk					xj					xd				
XVMULWEV.D.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	1	0	0	0	1	0	xk					xj					xd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
XVMULWEV.Q.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	1	0	0	0	1	1	xk					xj					xd				
XVMULWOD.H.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	1	0	0	1	0	0	xk					xj					xd				
XVMULWOD.W.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	1	0	0	1	0	1	xk					xj					xd				
XVMULWOD.D.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	1	0	0	1	1	0	xk					xj					xd				
XVMULWOD.Q.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	1	0	0	1	1	1	xk					xj					xd				
XVMULWEV.H.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	1	1	0	0	0	0	xk					xj					xd				
XVMULWEV.W.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	1	1	0	0	0	1	xk					xj					xd				
XVMULWEV.D.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	1	1	0	0	1	0	xk					xj					xd				
XVMULWEV.Q.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	1	1	0	0	1	1	xk					xj					xd				
XVMULWOD.H.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	1	1	0	1	0	0	xk					xj					xd				
XVMULWOD.W.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	1	1	0	1	0	1	xk					xj					xd				
XVMULWOD.D.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	1	1	0	1	1	0	xk					xj					xd				
XVMULWOD.Q.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	0	1	1	0	1	1	1	xk					xj					xd				
XVMULWEV.H.BU.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	0	0	0	0	0	xk					xj					xd				
XVMULWEV.W.HU.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	0	0	0	0	1	xk					xj					xd				
XVMULWEV.D.WU.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	0	0	0	1	0	xk					xj					xd				
XVMULWEV.Q.DU.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	0	0	0	1	1	xk					xj					xd				
XVMULWOD.H.BU.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	0	0	1	0	0	xk					xj					xd				
XVMULWOD.W.HU.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	0	0	1	0	1	xk					xj					xd				
XVMULWOD.D.WU.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	0	0	1	1	0	xk					xj					xd				
XVMULWOD.Q.DU.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	0	0	1	1	1	xk					xj					xd				
XVMADD.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	1	0	0	0	0	xk					xj					xd				
XVMADD.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	1	0	0	0	1	xk					xj					xd				
XVMADD.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	1	0	0	1	0	xk					xj					xd				
XVMADD.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	1	0	0	1	1	xk					xj					xd				
XVMSUB.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	xk					xj					xd				
XVMSUB.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	1	0	1	0	1	xk					xj					xd				
XVMSUB.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	1	0	1	1	0	xk					xj					xd				
XVMSUB.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	1	0	1	1	1	xk					xj					xd				
XVMADDWEV.H.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	1	1	0	0	0	xk					xj					xd				
XVMADDWEV.W.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	1	1	0	0	1	xk					xj					xd				
XVMADDWEV.D.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	1	1	0	1	0	xk					xj					xd				
XVMADDWEV.Q.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	1	1	0	1	1	xk					xj					xd				
XVMADDWOD.H.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	1	1	1	0	0	xk					xj					xd				
XVMADDWOD.W.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	1	1	1	0	1	xk					xj					xd				
XVMADDWOD.D.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	1	1	1	1	0	xk					xj					xd				
XVMADDWOD.Q.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	0	1	1	1	1	1	xk					xj					xd				
XVMADDWEV.H.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	1	0	1	0	0	0	xk					xj					xd				
XVMADDWEV.W.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	1	0	1	0	0	1	xk					xj					xd				
XVMADDWEV.D.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	1	0	1	0	1	0	xk					xj					xd				
XVMADDWEV.Q.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	1	0	1	0	1	1	xk					xj					xd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
XVMADDWOD.H.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	1	0	1	1	0	0	xk					xj					xd				
XVMADDWOD.W.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	1	0	1	1	0	1	xk					xj					xd				
XVMADDWOD.D.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	1	0	1	1	1	0	xk					xj					xd				
XVMADDWOD.Q.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	1	0	1	1	1	1	xk					xj					xd				
XVMADDWEV.H.BU.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	1	1	1	0	0	0	xk					xj					xd				
XVMADDWEV.W.HU.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	1	1	1	0	0	1	xk					xj					xd				
XVMADDWEV.D.WU.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	1	1	1	0	1	0	xk					xj					xd				
XVMADDWEV.Q.DU.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	1	1	1	0	1	1	xk					xj					xd				
XVMADDWOD.H.BU.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	1	1	1	1	0	0	xk					xj					xd				
XVMADDWOD.W.HU.H	**xk, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	1	1	1	1	0	1	xk					xj					xd				
XVMADDWOD.D.WU.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	1	1	1	1	1	0	xk					xj					xd				
XVMADDWOD.Q.DU.D	**xk, xj, xk**	0	1	1	1	0	1	0	0	1	0	1	1	1	1	1	1	1	xk					xj					xd				
XVDIV.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	0	0	0	0	0	xk					xj					xd				
XVDIV.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	0	0	0	0	1	xk					xj					xd				
XVDIV.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	0	0	0	1	0	xk					xj					xd				
XVDIV.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	0	0	0	1	1	xk					xj					xd				
XVMOD.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	0	0	1	0	0	xk					xj					xd				
XVMOD.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	0	0	1	0	1	xk					xj					xd				
XVMOD.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	0	0	1	1	0	xk					xj					xd				
XVMOD.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	0	0	1	1	1	xk					xj					xd				
XVDIV.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	0	1	0	0	0	xk					xj					xd				
XVDIV.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	0	1	0	0	1	xk					xj					xd				
XVDIV.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	0	xk					xj					xd				
XVDIV.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	xk					xj					xd				
XVMOD.BU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	0	1	1	0	0	xk					xj					xd				
XVMOD.HU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	0	1	1	0	1	xk					xj					xd				
XVMOD.WU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	0	1	1	1	0	xk					xj					xd				
XVMOD.DU	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	0	1	1	1	1	xk					xj					xd				
XVSLL.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	1	0	0	0	0	xk					xj					xd				
XVSLL.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	1	0	0	0	1	xk					xj					xd				
XVSLL.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	1	0	0	1	0	xk					xj					xd				
XVSLL.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	1	0	0	1	1	xk					xj					xd				
XVSRL.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	1	0	1	0	0	xk					xj					xd				
XVSRL.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	1	0	1	0	1	xk					xj					xd				
XVSRL.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	1	0	1	1	0	xk					xj					xd				
XVSRL.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	1	0	1	1	1	xk					xj					xd				
XVSRA.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	1	1	0	0	0	xk					xj					xd				
XVSRA.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	1	1	0	0	1	xk					xj					xd				
XVSRA.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	1	1	0	1	0	xk					xj					xd				
XVSRA.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	1	1	0	1	1	xk					xj					xd				
XVROTR.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	1	1	1	0	0	xk					xj					xd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
XVROTR.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	1	1	1	0	1	xk					xj					xd				
XVROTR.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	1	1	1	1	0	xk					xj					xd				
XVROTR.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	0	1	1	1	1	1	xk					xj					xd				
XVSRLR.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	0	0	0	0	0	xk					xj					xd				
XVSRLR.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	0	0	0	0	1	xk					xj					xd				
XVSRLR.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	0	0	0	1	0	xk					xj					xd				
XVSRLR.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	0	0	0	1	1	xk					xj					xd				
XVSRAR.B	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	0	0	1	0	0	xk					xj					xd				
XVSRAR.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	0	0	1	0	1	xk					xj					xd				
XVSRAR.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	0	0	1	1	0	xk					xj					xd				
XVSRAR.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	0	0	1	1	1	xk					xj					xd				
XVSRLN.B.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	0	1	0	0	1	xk					xj					xd				
XVSRLN.H.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	0	1	0	1	0	xk					xj					xd				
XVSRLN.W.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	0	1	0	1	1	xk					xj					xd				
XVSRAN.B.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	0	1	1	0	1	xk					xj					xd				
XVSRAN.H.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	0	1	1	1	0	xk					xj					xd				
XVSRAN.W.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	0	1	1	1	1	xk					xj					xd				
XVSRLRN.B.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	1	0	0	0	1	xk					xj					xd				
XVSRLRN.H.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	1	0	0	1	0	xk					xj					xd				
XVSRLRN.W.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	1	0	0	1	1	xk					xj					xd				
XVSRARN.B.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	1	0	1	0	1	xk					xj					xd				
XVSRARN.H.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	1	0	1	1	0	xk					xj					xd				
XVSRARN.W.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	1	0	1	1	1	xk					xj					xd				
XVSSRLN.B.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	1	1	0	0	1	xk					xj					xd				
XVSSRLN.H.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	1	1	0	1	0	xk					xj					xd				
XVSSRLN.W.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	1	1	0	1	1	xk					xj					xd				
XVSSRAN.B.H	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	1	1	1	0	1	xk					xj					xd				
XVSSRAN.H.W	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	0	xk					xj					xd				
XVSSRAN.W.D	**xd, xj, xk**	0	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	xk					xj					xd				
XVSSRLRN.B.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	1	xk					xj					xd				
XVSSRLRN.H.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	0	0	0	1	0	xk					xj					xd				
XVSSRLRN.W.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	0	0	0	1	1	xk					xj					xd				
XVSSRARN.B.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	0	0	1	0	1	xk					xj					xd				
XVSSRARN.H.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	0	0	1	1	0	xk					xj					xd				
XVSSRARN.W.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	0	0	1	1	1	xk					xj					xd				
XVSSRLN.BU.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	0	1	0	0	1	xk					xj					xd				
XVSSRLN.HU.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	0	1	0	1	0	xk					xj					xd				
XVSSRLN.WU.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	0	1	0	1	1	xk					xj					xd				
XVSSRAN.BU.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	0	1	1	0	1	xk					xj					xd				
XVSSRAN.HU.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	0	1	1	1	0	xk					xj					xd				
XVSSRAN.WU.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	0	1	1	1	1	xk					xj					xd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
XVSSRLRN.BU.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	1	0	0	0	1	xk					xj					xd				
XVSSRLRN.HU.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	1	0	0	1	0	xk					xj					xd				
XVSSRLRN.WU.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	1	0	0	1	1	xk					xj					xd				
XVSSRARN.BU.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	1	0	1	0	1	xk					xj					xd				
XVSSRARN.HU.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	1	0	1	1	0	xk					xj					xd				
XVSSRARN.WU.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	1	0	1	1	1	xk					xj					xd				
XVBITCLR.B	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	1	1	0	0	0	xk					xj					xd				
XVBITCLR.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	1	1	0	0	1	xk					xj					xd				
XVBITCLR.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	1	1	0	1	0	xk					xj					xd				
XVBITCLR.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	1	1	0	1	1	xk					xj					xd				
XVBITSET.B	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	1	1	1	0	0	xk					xj					xd				
XVBITSET.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	1	1	1	0	1	xk					xj					xd				
XVBITSET.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	1	1	1	1	0	xk					xj					xd				
XVBITSET.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	0	1	1	1	1	1	xk					xj					xd				
XVBITREV.B	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	0	0	0	0	0	xk					xj					xd				
XVBITREV.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	0	0	0	0	1	xk					xj					xd				
XVBITREV.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	0	0	0	1	0	xk					xj					xd				
XVBITREV.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	0	0	0	1	1	xk					xj					xd				
XVPACKEV.B	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	0	1	1	0	0	xk					xj					xd				
XVPACKEV.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	0	1	1	0	1	xk					xj					xd				
XVPACKEV.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	0	1	1	1	0	xk					xj					xd				
XVPACKEV.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	0	1	1	1	1	xk					xj					xd				
XVPACKOD.B	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	1	0	0	0	0	xk					xj					xd				
XVPACKOD.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	1	0	0	0	1	xk					xj					xd				
XVPACKOD.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	1	0	0	1	0	xk					xj					xd				
XVPACKOD.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	1	0	0	1	1	xk					xj					xd				
XVILVL.B	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	1	0	1	0	0	xk					xj					xd				
XVILVL.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	1	0	1	0	1	xk					xj					xd				
XVILVL.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	1	0	1	1	0	xk					xj					xd				
XVILVL.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	1	0	1	1	1	xk					xj					xd				
XVILVH.B	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	1	1	0	0	0	xk					xj					xd				
XVILVH.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	1	1	0	0	1	xk					xj					xd				
XVILVH.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	1	1	0	1	0	xk					xj					xd				
XVILVH.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	1	1	0	1	1	xk					xj					xd				
XVPICKEV.B	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	1	1	1	0	0	xk					xj					xd				
XVPICKEV.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	1	1	1	0	1	xk					xj					xd				
XVPICKEV.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	1	1	1	1	0	xk					xj					xd				
XVPICKEV.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	0	1	1	1	1	1	1	xk					xj					xd				
XVPICKOD.B	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	0	0	0	0	0	xk					xj					xd				
XVPICKOD.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	0	0	0	0	1	xk					xj					xd				
XVPICKOD.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	0	0	0	1	0	xk					xj					xd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
XVPICKOD.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	0	0	0	1	1	xk					xj						xd			
XVREPLVE.B	**xd, xj, rk**	0	1	1	1	0	1	0	1	0	0	1	0	0	0	1	0	0	rk					xj						xd			
XVREPLVE.H	**xd, xj, rk**	0	1	1	1	0	1	0	1	0	0	1	0	0	0	1	0	1	rk					xj						xd			
XVREPLVE.W	**xd, xj, rk**	0	1	1	1	0	1	0	1	0	0	1	0	0	0	1	1	0	rk					xj						xd			
XVREPLVE.D	**xd, xj, rk**	0	1	1	1	0	1	0	1	0	0	1	0	0	0	1	1	1	rk					xj						xd			
XVAND.V	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	0	1	1	0	0	xk					xj						xd			
XVOR.V	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	0	1	1	0	1	xk					xj						xd			
XVXOR.V	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	0	1	1	1	0	xk					xj						xd			
XVNOR.V	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	0	1	1	1	1	xk					xj						xd			
XVANDN.V	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	1	0	0	0	0	xk					xj						xd			
XVORN.V	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	1	0	0	0	1	xk					xj						xd			
XVFRSTP.B	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	1	0	1	1	0	xk					xj						xd			
XVFRSTP.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	1	0	1	1	1	xk					xj						xd			
XVADD.Q	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	1	1	0	1	0	xk					xj						xd			
XVSUB.Q	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	1	1	0	1	1	xk					xj						xd			
XVSIGNCOV.B	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	xk					xj						xd			
XVSIGNCOV.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	1	1	1	0	1	xk					xj						xd			
XVSIGNCOV.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	0	xk					xj						xd			
XVSIGNCOV.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	1	xk					xj						xd			
XVFADD.S	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	1	0	0	0	0	1	xk					xj						xd			
XVFADD.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	1	0	0	0	1	0	xk					xj						xd			
XVFSUB.S	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	1	0	0	1	0	1	xk					xj						xd			
XVFSUB.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	1	0	0	1	1	0	xk					xj						xd			
XVFMUL.S	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	1	1	0	0	0	1	xk					xj						xd			
XVFMUL.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	1	1	0	0	1	0	xk					xj						xd			
XVFDIV.S	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	1	1	0	1	0	1	xk					xj						xd			
XVFDIV.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	1	1	0	1	1	0	xk					xj						xd			
XVFMAX.S	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	1	1	1	0	0	1	xk					xj						xd			
XVFMAX.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	1	1	1	0	1	0	xk					xj						xd			
XVFMIN.S	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	1	1	1	1	0	1	xk					xj						xd			
XVFMIN.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	0	1	1	1	1	1	1	0	xk					xj						xd			
XVFMAXA.S	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	xk					xj						xd			
XVFMAXA.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	1	0	0	0	0	0	1	0	xk					xj						xd			
XVFMINA.S	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	1	0	0	0	0	1	0	1	xk					xj						xd			
XVFMINA.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	1	0	0	0	0	1	1	0	xk					xj						xd			
XVFCVT.H.S	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	1	0	0	0	1	1	0	0	xk					xj						xd			
XVFCVT.S.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	1	0	0	0	1	1	0	1	xk					xj						xd			
XVFFINT.S.L	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	1	0	0	1	0	0	0	0	xk					xj						xd			
XVFTINT.W.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	1	0	0	1	0	0	1	1	xk					xj						xd			
XVFTINTRM.W.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	1	0	0	1	0	1	0	0	xk					xj						xd			
XVFTINTRP.W.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	1	0	0	1	0	1	0	1	xk					xj						xd			

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
XVFTINTRZ.W.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	1	0	0	1	0	1	1	0	xk					xj					xd				
XVFTINTRNE.W.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	1	0	0	1	0	1	1	1	xk					xj					xd				
XVSHUF.H	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	1	1	1	1	0	1	0	1	xk					xj					xd				
XVSHUF.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	1	1	1	1	0	1	1	0	xk					xj					xd				
XVSHUF.D	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	1	1	1	1	0	1	1	1	xk					xj					xd				
XVPERM.W	**xd, xj, xk**	0	1	1	1	0	1	0	1	0	1	1	1	1	1	0	1	0	xk					xj					xd				
XVSEQI.B	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	0	0	0	0	0	0	si5					xj					xd				
XVSEQI.H	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	0	0	0	0	0	1	si5					xj					xd				
XVSEQI.W	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	0	0	0	0	1	0	si5					xj					xd				
XVSEQI.D	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	0	0	0	0	1	1	si5					xj					xd				
XVSLEI.B	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	0	0	0	1	0	0	si5					xj					xd				
XVSLEI.H	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	0	0	0	1	0	1	si5					xj					xd				
XVSLEI.W	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	0	0	0	1	1	0	si5					xj					xd				
XVSLEI.D	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	0	0	0	1	1	1	si5					xj					xd				
XVSLEI.BU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	0	1	0	0	0	ui5					xj					xd				
XVSLEI.HU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	0	1	0	0	1	ui5					xj					xd				
XVSLEI.WU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	0	1	0	1	0	ui5					xj					xd				
XVSLEI.DU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	0	1	0	1	1	ui5					xj					xd				
XVSLTI.B	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	0	0	1	1	0	0	si5					xj					xd				
XVSLTI.H	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	0	0	1	1	0	1	si5					xj					xd				
XVSLTI.W	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	0	0	1	1	1	0	si5					xj					xd				
XVSLTI.D	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	0	0	1	1	1	1	si5					xj					xd				
XVSLTI.BU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	1	0	0	0	0	ui5					xj					xd				
XVSLTI.HU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	1	0	0	0	1	ui5					xj					xd				
XVSLTI.WU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	1	0	0	1	0	ui5					xj					xd				
XVSLTI.DU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	1	0	0	1	1	ui5					xj					xd				
XVADDI.BU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	1	0	1	0	0	ui5					xj					xd				
XVADDI.HU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	1	0	1	0	1	ui5					xj					xd				
XVADDI.WU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	1	0	1	1	0	ui5					xj					xd				
XVADDI.DU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	1	0	1	1	1	ui5					xj					xd				
XVSUBI.BU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	1	1	0	0	0	ui5					xj					xd				
XVSUBI.HU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	1	1	0	0	1	ui5					xj					xd				
XVSUBI.WU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	1	1	0	1	0	ui5					xj					xd				
XVSUBI.DU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	1	1	0	1	1	ui5					xj					xd				
XVBSLL.V	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	1	1	1	0	0	ui5					xj					xd				
XVBSRL.V	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	0	1	1	1	0	1	ui5					xj					xd				
XVMAXI.B	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	1	0	0	0	0	0	si5					xj					xd				
XVMAXI.H	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	1	0	0	0	0	1	si5					xj					xd				
XVMAXI.W	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	1	0	0	0	1	0	si5					xj					xd				
XVMAXI.D	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	1	0	0	0	1	1	si5					xj					xd				
XVMINI.B	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	1	0	0	1	0	0	si5					xj					xd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
XVMINI.H	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	1	0	0	1	0	1	si5					xj					xd				
XVMINI.W	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	1	0	0	1	1	0	si5					xj					xd				
XVMINI.D	**xd, xj, si5**	0	1	1	1	0	1	1	0	1	0	0	1	0	0	1	1	1	si5					xj					xd				
XVMAXI.BU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	1	0	1	0	0	0	ui5					xj					xd				
XVMAXI.HU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	1	0	1	0	0	1	ui5					xj					xd				
XVMAXI.WU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	1	0	1	0	1	0	ui5					xj					xd				
XVMAXI.DU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	1	0	1	0	1	1	ui5					xj					xd				
XVMINI.BU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	1	0	1	1	0	0	ui5					xj					xd				
XVMINI.HU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	1	0	1	1	0	1	ui5					xj					xd				
XVMINI.WU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	1	0	1	1	1	0	ui5					xj					xd				
XVMINI.DU	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	1	0	1	1	1	1	ui5					xj					xd				
XVFRSTPI.B	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	1	1	0	1	0	0	ui5					xj					xd				
XVFRSTPI.H	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	1	1	0	1	0	1	ui5					xj					xd				
XVCLO.B	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	0	0	0	0	xj					xd				
XVCLO.H	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	0	0	0	1	xj					xd				
XVCLO.W	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	0	0	1	0	xj					xd				
XVCLO.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	0	0	1	1	xj					xd				
XVCLZ.B	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	0	1	0	0	xj					xd				
XVCLZ.H	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	0	1	0	1	xj					xd				
XVCLZ.W	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	0	1	1	0	xj					xd				
XVCLZ.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	0	1	1	1	xj					xd				
XVPCNT.B	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	0	1	0	0	xj					xd				
XVPCNT.H	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	0	1	0	1	xj					xd				
XVPCNT.W	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	0	1	0	1	xj					xd				
XVPCNT.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	0	1	0	1	xj					xd				
XVNEG.B	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	0	1	1	0	xj					xd				
XVNEG.H	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	0	1	1	0	xj					xd				
XVNEG.W	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	0	1	1	1	xj					xd				
XVNEG.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	0	1	1	1	xj					xd				
XVMSKLTZ.B	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	1	0	0	0	xj					xd				
XVMSKLTZ.H	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	1	0	0	0	xj					xd				
XVMSKLTZ.W	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	1	0	0	1	xj					xd				
XVMSKLTZ.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	1	0	0	1	xj					xd				
XVMSKGEZ.B	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	1	0	1	0	xj					xd				
XVMSKNZ.B	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	0	1	1	0	0	xj					xd				
XVSETEQZ.V	**cd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	0	0	1	1	0	xj					0	0	cd		
XVSETNEZ.V	**cd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	0	0	1	1	1	xj					0	0	cd		
XVSETANYEQZ.B	**cd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	0	1	0	0	0	xj					0	0	cd		
XVSETANYEQZ.H	**cd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	0	1	0	0	1	xj					0	0	cd		
XVSETANYEQZ.W	**cd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	0	1	0	1	0	xj					0	0	cd		
XVSETANYEQZ.D	**cd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	0	1	0	1	1	xj					0	0	cd		

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
XVSETALLNEZ.B	**cd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	0	0	xj					0	0	cd		
XVSETALLNEZ.H	**cd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	0	1	xj					0	0	cd		
XVSETALLNEZ.W	**cd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1	0	xj					0	0	cd		
XVSETALLNEZ.D	**cd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1	1	xj					0	0	cd		
XVFLOGB.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	1	0	0	0	1	xj					xd				
XVFLOGB.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	1	0	0	1	0	xj					xd				
XVFCLASS.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	1	0	1	0	1	xj					xd				
XVFCLASS.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	1	0	1	1	0	xj					xd				
XVFSQRT.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	xj					xd				
XVFSQRT.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	xj					xd				
XVFRECIP.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	1	1	1	0	1	xj					xd				
XVFRECIP.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	0	1	1	1	1	1	0	xj					xd				
XVFRSQRT.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	0	0	0	0	0	1	xj					xd				
XVFRSQRT.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	0	0	0	0	1	0	xj					xd				
XVFRECIPE.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	0	0	0	1	0	1	xj					xd				
XVFRECIPE.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	0	0	0	1	1	0	xj					xd				
XVFRSQRTE.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	0	0	1	0	0	1	xj					xd				
XVFRSQRTE.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	0	0	1	0	1	0	xj					xd				
XVFRINT.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	0	0	1	1	0	1	xj					xd				
XVFRINT.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	0	0	1	1	1	0	xj					xd				
XVFRINTRM.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	0	1	0	0	0	1	xj					xd				
XVFRINTRM.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	0	1	0	0	1	0	xj					xd				
XVFRINTRP.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	0	1	0	1	0	1	xj					xd				
XVFRINTRP.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	0	1	0	1	1	0	xj					xd				
XVFRINTRZ.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	0	1	1	0	0	1	xj					xd				
XVFRINTRZ.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	0	1	1	0	1	0	xj					xd				
XVFRINTRNE.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	0	1	1	1	0	1	xj					xd				
XVFRINTRNE.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	0	1	1	1	1	0	xj					xd				
XVFCVTL.S.H	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	1	1	1	0	1	0	xj					xd				
XVFCVTH.S.H	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	1	1	1	0	1	1	xj					xd				
XVFCVTL.D.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	1	1	1	1	0	0	xj					xd				
XVFCVTH.D.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	0	1	1	1	1	1	0	1	xj					xd				
XVFFINT.S.W	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	0	0	0	0	0	xj					xd				
XVFFINT.S.WU	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	0	0	0	0	1	xj					xd				
XVFFINT.D.L	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	0	0	0	1	0	xj					xd				
XVFFINT.D.LU	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	0	0	0	1	1	xj					xd				
XVFFINTL.D.W	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	0	0	1	0	0	xj					xd				
XVFFINTH.D.W	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	0	0	1	0	1	xj					xd				
XVFTINT.W.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	0	1	1	0	0	xj					xd				
XVFTINT.L.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	0	1	1	0	1	xj					xd				
XVFTINTRM.W.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	0	1	1	1	0	xj					xd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
XVFTINTRM.L.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	0	1	1	1	1	xj					xd				
XVFTINTRP.W.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	1	0	0	0	0	xj					xd				
XVFTINTRP.L.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	1	0	0	0	1	xj					xd				
XVFTINTRZ.W.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	1	0	0	1	0	xj					xd				
XVFTINTRZ.L.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	1	0	0	1	1	xj					xd				
XVFTINTRNE.W.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	1	0	1	0	0	xj					xd				
XVFTINTRNE.L.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	1	0	1	0	1	xj					xd				
XVFTINT.WU.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	1	0	1	1	0	xj					xd				
XVFTINT.LU.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	1	0	1	1	1	xj					xd				
XVFTINTRZ.WU.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	1	1	1	0	0	xj					xd				
XVFTINTRZ.LU.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	0	1	1	1	0	1	xj					xd				
XVFTINTL.L.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	0	0	0	0	0	xj					xd				
XVFTINTH.L.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	0	0	0	0	1	xj					xd				
XVFTINTRML.L.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	0	0	0	1	0	xj					xd				
XVFTINTRMH.L.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	0	0	0	1	1	xj					xd				
XVFTINTRPL.L.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	0	0	1	0	0	xj					xd				
XVFTINTRPH.L.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	0	0	1	0	1	xj					xd				
XVFTINTRZL.L.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	0	0	1	1	0	xj					xd				
XVFTINTRZH.L.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	0	0	1	1	1	xj					xd				
XVFTINTRNEL.L.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	0	1	0	0	0	xj					xd				
XVFTINTRNEH.L.S	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	0	1	0	0	1	xj					xd				
XVEXTH.H.B	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	1	1	0	0	0	xj					xd				
XVEXTH.W.H	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	1	1	0	0	1	xj					xd				
XVEXTH.D.W	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	1	1	0	1	0	xj					xd				
XVEXTH.Q.D	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	1	1	0	1	1	xj					xd				
XVEXTH.HU.BU	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	1	1	1	0	0	xj					xd				
XVEXTH.WU.HU	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	1	1	1	0	1	xj					xd				
XVEXTH.DU.WU	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	1	1	1	1	0	xj					xd				
XVEXTH.QU.DU	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	0	1	1	1	1	1	1	xj					xd				
XVREPLGR2VR.B	**xd, rj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	0	0	0	rj					xd				
XVREPLGR2VR.H	**xd, rj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	0	0	1	rj					xd				
XVREPLGR2VR.W	**xd, rj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	0	1	0	rj					xd				
XVREPLGR2VR.D	**xd, rj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	0	1	1	rj					xd				
VEXT2XV.H.B	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	1	0	0	xj					xd				
VEXT2XV.W.B	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	1	0	1	xj					xd				
VEXT2XV.D.B	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	1	1	0	xj					xd				
VEXT2XV.W.H	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	0	1	1	1	xj					xd				
VEXT2XV.D.H	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	1	0	0	0	xj					xd				
VEXT2XV.D.W	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	1	0	0	1	xj					xd				
VEXT2XV.HU.BU	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	1	0	1	0	xj					xd				
VEXT2XV.WU.BU	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	1	0	1	1	xj					xd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
VEXT2XV.DU.BU	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	1	1	0	0	xj					xd				
VEXT2XV.WU.HU	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	1	1	0	1	xj					xd				
VEXT2XV.DU.HU	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	1	1	1	0	xj					xd				
VEXT2XV.DU.WU	**xd, xj**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	1	1	1	1	xj					xd				
XVHSELI.D	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	1	ui5					xj					xd				
XVROTRI.B	**xd, xj, ui3**	0	1	1	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	1	ui3		xj					xd				
XVROTRI.H	**xd, xj, ui4**	0	1	1	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	1	ui4			xj					xd				
XVROTRI.W	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	1	0	0	0	0	0	0	1	ui5				xj					xd				
XVROTRI.D	**xd, xj, ui6**	0	1	1	1	0	1	1	0	1	0	1	0	0	0	0	0	1	ui6					xj					xd				
XVSRLRI.B	**xd, xj, ui3**	0	1	1	1	0	1	1	0	1	0	1	0	0	1	0	0	0	0	0	1	ui3		xj					xd				
XVSRLRI.H	**xd, xj, ui4**	0	1	1	1	0	1	1	0	1	0	1	0	0	1	0	0	0	0	1	ui4			xj					xd				
XVSRLRI.W	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	1	0	0	1	0	0	0	1	ui5				xj					xd				
XVSRLRI.D	**xd, xj, ui6**	0	1	1	1	0	1	1	0	1	0	1	0	0	1	0	0	1	ui6					xj					xd				
XVSRARI.B	**xd, xj, ui3**	0	1	1	1	0	1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	ui3		xj					xd				
XVSRARI.H	**xd, xj, ui4**	0	1	1	1	0	1	1	0	1	0	1	0	1	0	0	0	0	0	1	ui4			xj					xd				
XVSRARI.W	**xd, xj, ui5**	0	1	1	1	0	1	1	0	1	0	1	0	1	0	0	0	1	ui5					xj					xd				
XVSRARI.D	**xd, xj, ui6**	0	1	1	1	0	1	1	0	1	0	1	0	1	0	0	1	ui6						xj					xd				
XVINSGR2VR.W	**xd, rj, ui3**	0	1	1	1	0	1	1	0	1	1	1	0	1	0	1	1	1	1	0	ui3			rj					xd				
XVINSGR2VR.D	**xd, rj, ui2**	0	1	1	1	0	1	1	0	1	1	1	0	1	0	1	1	1	1	1	0	ui2		rj					xd				
XVPICKVE2GR.W	**rd, xj, ui3**	0	1	1	1	0	1	1	0	1	1	1	0	1	1	1	1	1	1	0	ui3			xj					rd				
XVPICKVE2GR.D	**rd, xj, ui2**	0	1	1	1	0	1	1	0	1	1	1	0	1	1	1	1	1	1	1	0	ui2		xj					rd				
XVPICKVE2GR.WU	**rd, xj, ui3**	0	1	1	1	0	1	1	0	1	1	1	1	0	0	1	1	1	1	0	ui3			xj					rd				
XVPICKVE2GR.DU	**rd, xj, ui2**	0	1	1	1	0	1	1	0	1	1	1	1	0	0	1	1	1	1	1	0	ui2		xj					rd				
XVREPL128VEI.B	**xd, xj, ui4**	0	1	1	1	0	1	1	0	1	1	1	1	0	1	1	1	1	0	ui4				xj					xd				
XVREPL128VEI.H	**xd, xj, ui3**	0	1	1	1	0	1	1	0	1	1	1	1	0	1	1	1	1	1	0	ui3			xj					xd				
XVREPL128VEI.W	**xd, xj, ui2**	0	1	1	1	0	1	1	0	1	1	1	1	0	1	1	1	1	1	1	0	ui2		xj					xd				
XVREPL128VEI.D	**xd, xj, ui1**	0	1	1	1	0	1	1	0	1	1	1	1	0	1	1	1	1	1	1	1	0	ui1	xj					xd				
XVINSVE0.W	**xd, xj, ui3**	0	1	1	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	0	ui3			xj					xd				
XVINSVE0.D	**xd, xj, ui2**	0	1	1	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	ui2		xj					xd				
XVPICKVE.W	**xd, xj, ui3**	0	1	1	1	0	1	1	1	0	0	0	0	0	0	1	1	1	1	0	ui3			xj					xd				
XVPICKVE.D	**xd, xj, ui2**	0	1	1	1	0	1	1	1	0	0	0	0	0	0	1	1	1	1	1	0	ui2		xj					xd				
XVREPLVE0.B	**xd, xj**	0	1	1	1	0	1	1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0	xj					xd				
XVREPLVE0.H	**xd, xj**	0	1	1	1	0	1	1	1	0	0	0	0	0	1	1	1	1	0	0	0	0	0	xj					xd				
XVREPLVE0.W	**xd, xj**	0	1	1	1	0	1	1	1	0	0	0	0	0	1	1	1	1	1	0	0	0	0	xj					xd				
XVREPLVE0.D	**xd, xj**	0	1	1	1	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	0	0	0	xj					xd				
XVREPLVE0.Q	**xd, xj**	0	1	1	1	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	0	0	xj					xd				
XVSLLWIL.H.B	**xd, xj, ui3**	0	1	1	1	0	1	1	1	0	0	0	0	1	0	0	0	0	0	0	1	ui3		xj					xd				
XVSLLWIL.W.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	0	0	0	1	0	0	0	0	0	1	ui4			xj					xd				
XVSLLWIL.D.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	0	0	0	1	0	0	0	1	ui5					xj					xd				
XVEXTL.Q.D	**xd, xj**	0	1	1	1	0	1	1	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	xj					xd				
XVSLLWIL.HU.BU	**xd, xj, ui3**	0	1	1	1	0	1	1	1	0	0	0	0	1	1	0	0	0	0	0	1	ui3		xj					xd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
XVSLLWIL.WU.HU	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	0	0	0	1	1	0	0	0	1	ui4				xj					xd				
XVSLLWIL.DU.WU	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	0	0	0	1	1	0	0	1	ui5					xj					xd				
XVEXTL.QU.DU	**xd, xj**	0	1	1	1	0	1	1	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	xj					xd				
XVBITCLRI.B	**xd, xj, ui3**	0	1	1	1	0	1	1	1	0	0	0	1	0	0	0	0	0	0	1	ui3			xj					xd				
XVBITCLRI.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	0	0	1	0	0	0	0	0	1	ui4				xj					xd				
XVBITCLRI.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	0	0	1	0	0	0	0	1	ui5					xj					xd				
XVBITCLRI.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	0	0	1	0	0	0	1	ui6						xj					xd				
XVBITSETI.B	**xd, xj, ui3**	0	1	1	1	0	1	1	1	0	0	0	1	0	1	0	0	0	0	1	ui3			xj					xd				
XVBITSETI.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	0	0	1	0	1	0	0	0	1	ui4				xj					xd				
XVBITSETI.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	0	0	1	0	1	0	0	1	ui5					xj					xd				
XVBITSETI.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	0	0	1	0	1	0	1	ui6						xj					xd				
XVBITREVI.B	**xd, xj, ui3**	0	1	1	1	0	1	1	1	0	0	0	1	1	0	0	0	0	0	1	ui3			xj					xd				
XVBITREVI.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	0	0	1	1	0	0	0	0	1	ui4				xj					xd				
XVBITREVI.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	0	0	1	1	0	0	0	1	ui5					xj					xd				
XVBITREVI.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	0	0	1	1	0	0	1	ui6						xj					xd				
XVSAT.B	**xd, xj, ui3**	0	1	1	1	0	1	1	1	0	0	1	0	0	1	0	0	0	0	1	ui3			xj					xd				
XVSAT.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	0	1	0	0	1	0	0	0	1	ui4				xj					xd				
XVSAT.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	0	1	0	0	1	0	0	1	ui5					xj					xd				
XVSAT.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	0	1	0	0	1	0	1	ui6						xj					xd				
XVSAT.BU	**xd, xj, ui3**	0	1	1	1	0	1	1	1	0	0	1	0	1	0	0	0	0	0	1	ui3			xj					xd				
XVSAT.HU	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	0	1	0	1	0	0	0	0	1	ui4				xj					xd				
XVSAT.WU	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	0	1	0	1	0	0	0	1	ui5					xj					XD				
XVSAT.DU	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	0	1	0	1	0	0	1	ui6						xj					xd				
XVSLLI.B	**xd, xj, ui3**	0	1	1	1	0	1	1	1	0	0	1	0	1	1	0	0	0	0	1	ui3			xj					xd				
XVSLLI.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	0	1	0	1	1	0	0	0	1	ui4				xj					xd				
XVSLLI.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	0	1	0	1	1	0	0	1	ui5					xj					xd				
XVSLLI.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	0	1	0	1	1	0	1	ui6						xj					xd				
XVSRLI.B	**xd, xj, ui3**	0	1	1	1	0	1	1	1	0	0	1	1	0	0	0	0	0	0	1	ui3			xj					xd				
XVSRLI.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	0	1	1	0	0	0	0	0	1	ui4				xj					xd				
XVSRLI.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	0	1	1	0	0	0	0	1	ui5					xj					xd				
XVSRLI.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	0	1	1	0	0	0	1	ui6						xj					xd				
XVSRAI.B	**xd, xj, ui3**	0	1	1	1	0	1	1	1	0	0	1	1	0	1	0	0	0	0	1	ui3			xj					xd				
XVSRAI.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	0	1	1	0	1	0	0	0	1	ui4				xj					xd				
XVSRAI.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	0	1	1	0	1	0	0	1	ui5					xj					xd				
XVSRAI.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	0	1	1	0	1	0	1	ui6						xj					xd				
XVSRLNI.B.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	1	0	0	0	0	0	0	0	1	ui4				xj					xd				
XVSRLNI.H.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	1	0	0	0	0	0	0	1	ui5					xj					xd				
XVSRLNI.W.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	1	0	0	0	0	0	1	ui6						xj					xd				
XVSRLNI.D.Q	**xd, xj, ui7**	0	1	1	1	0	1	1	1	0	1	0	0	0	0	1	ui7							xj					xd				
XVSRLRNI.B.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	1	0	0	0	1	0	0	0	1	ui4				xj					xd				
XVSRLRNI.H.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	1	0	0	0	1	0	0	1	ui5					xj					xd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
XVSRLRNI.W.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	1	0	0	0	1	0	1	ui6						xj					xd				
XVSRLRNI.D.Q	**xd, xj, ui7**	0	1	1	1	0	1	1	1	0	1	0	0	0	1	1	ui7							xj					xd				
XVSSRLNI.B.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	1	0	0	1	0	0	0	0	1	ui4				xj					xd				
XVSSRLNI.H.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	1	0	0	1	0	0	0	1	ui5					xj					xd				
XVSSRLNI.W.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	1	0	0	1	0	0	1	ui6						xj					xd				
XVSSRLNI.D.Q	**xd, xj, ui7**	0	1	1	1	0	1	1	1	0	1	0	0	1	0	1	ui7							xj					xd				
XVSSRLNI.BU.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	1	0	0	1	1	0	0	0	1	ui4				xj					xd				
XVSSRLNI.HU.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	1	0	0	1	1	0	0	1	ui5					xj					xd				
XVSSRLNI.WU.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	1	0	0	1	1	0	1	ui6						xj					xd				
XVSSRLNI.DU.Q	**xd, xj, ui7**	0	1	1	1	0	1	1	1	0	1	0	0	1	1	1	ui7							xj					xd				
XVSSRLRNI.B.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	1	0	1	0	0	0	0	0	1	ui4				xj					xd				
XVSSRLRNI.H.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	1	0	1	0	0	0	0	1	ui5					xj					xd				
XVSSRLRNI.W.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	1	0	1	0	0	0	1	ui6						xj					xd				
XVSSRLRNI.D.Q	**xd, xj, ui7**	0	1	1	1	0	1	1	1	0	1	0	1	0	0	1	ui7							xj					xd				
XVSSRLRNI.BU.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	1	0	1	0	1	0	0	0	1	ui4				xj					xd				
XVSSRLRNI.HU.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	1	0	1	0	1	0	0	1	ui5					xj					xd				
XVSSRLRNI.WU.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	1	0	1	0	1	0	1	ui6						xj					xd				
XVSSRLRNI.DU.Q	**xd, xj, ui7**	0	1	1	1	0	1	1	1	0	1	0	1	0	1	1	ui7							xj					xd				
XVSRANI.B.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	1	0	1	1	0	0	0	0	1	ui4				xj					xd				
XVSRANI.H.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	1	0	1	1	0	0	0	1	ui5					xj					xd				
XVSRANI.W.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	1	0	1	1	0	0	1	ui6						xj					xd				
XVSRANI.D.Q	**xd, xj, ui7**	0	1	1	1	0	1	1	1	0	1	0	1	1	0	1	ui7							xj					xd				
XVSRARNI.B.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	1	0	1	1	1	0	0	0	1	ui4				xj					xd				
XVSRARNI.H.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	1	0	1	1	1	0	0	1	ui5					xj					xd				
XVSRARNI.W.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	1	0	1	1	1	0	1	ui6						xj					xd				
XVSRARNI.D.Q	**xd, xj, ui7**	0	1	1	1	0	1	1	1	0	1	0	1	1	1	1	ui7							xj					xd				
XVSSRANI.B.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	1	1	0	0	0	0	0	0	1	ui4				xj					xd				
XVSSRANI.H.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	1	1	0	0	0	0	0	1	ui5					xj					xd				
XVSSRANI.W.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	1	1	0	0	0	0	1	ui6						xj					xd				
XVSSRANI.D.Q	**xd, xj, ui7**	0	1	1	1	0	1	1	1	0	1	1	0	0	0	1	ui7							xj					xd				
XVSSRANI.BU.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	1	1	0	0	1	0	0	0	1	ui4				xj					xd				
XVSSRANI.HU.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	1	1	0	0	1	0	0	1	ui5					xj					xd				
XVSSRANI.WU.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	1	1	0	0	1	0	1	ui6						xj					xd				
XVSSRANI.DU.Q	**xd, xj, ui7**	0	1	1	1	0	1	1	1	0	1	1	0	0	1	1	ui7							xj					xd				
XVSSRARNI.B.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	1	1	0	1	0	0	0	0	1	ui4				xj					xd				
XVSSRARNI.H.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	1	1	0	1	0	0	0	1	ui5					xj					xd				
XVSSRARNI.W.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	1	1	0	1	0	0	1	ui6						xj					xd				
XVSSRARNI.D.Q	**xd, xj, ui7**	0	1	1	1	0	1	1	1	0	1	1	0	1	0	1	ui7							xj					xd				
XVSSRARNI.BU.H	**xd, xj, ui4**	0	1	1	1	0	1	1	1	0	1	1	0	1	1	0	0	0	1	ui4				xj					xd				
XVSSRARNI.HU.W	**xd, xj, ui5**	0	1	1	1	0	1	1	1	0	1	1	0	1	1	0	0	1	ui5					xj					xd				
XVSSRARNI.WU.D	**xd, xj, ui6**	0	1	1	1	0	1	1	1	0	1	1	0	1	1	0	1	ui6						xj					xd				

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
XVSSRARNI.DU.Q	**xd, xj, ui7**	0	1	1	1	0	1	1	1	0	1	1	0	1	1	1	ui7							xj					xd				
XVEXTRINS.D	**xd, xj, ui8**	0	1	1	1	0	1	1	1	1	0	0	0	0	0	ui8								xj					xd				
XVEXTRINS.W	**xd, xj, ui8**	0	1	1	1	0	1	1	1	1	0	0	0	0	1	ui8								xj					xd				
XVEXTRINS.H	**xd, xj, ui8**	0	1	1	1	0	1	1	1	1	0	0	0	1	0	ui8								xj					xd				
XVEXTRINS.B	**xd, xj, ui8**	0	1	1	1	0	1	1	1	1	0	0	0	1	1	ui8								xj					xd				
XVSHUF4I.B	**xd, xj, ui8**	0	1	1	1	0	1	1	1	1	0	0	1	0	0	ui8								xj					xd				
XVSHUF4I.H	**xd, xj, ui8**	0	1	1	1	0	1	1	1	1	0	0	1	0	1	ui8								xj					xd				
XVSHUF4I.W	**xd, xj, ui8**	0	1	1	1	0	1	1	1	1	0	0	1	1	0	ui8								xj					xd				
XVSHUF4I.D	**xd, xj, ui8**	0	1	1	1	0	1	1	1	1	0	0	1	1	1	ui8								xj					xd				
XVBITSELI.B	**xd, xj, ui8**	0	1	1	1	0	1	1	1	1	1	0	0	0	1	ui8								xj					xd				
XVANDI.B	**xd, xj, ui8**	0	1	1	1	0	1	1	1	1	1	0	1	0	0	ui8								xj					xd				
XVORI.B	**xd, xj, ui8**	0	1	1	1	0	1	1	1	1	1	0	1	0	1	ui8								xj					xd				
XVXORI.B	**xd, xj, ui8**	0	1	1	1	0	1	1	1	1	1	0	1	1	0	ui8								xj					xd				
XVNORI.B	**xd, xj, ui8**	0	1	1	1	0	1	1	1	1	1	0	1	1	1	ui8								xj					xd				
XVLDI	**xd, i13**	0	1	1	1	0	1	1	1	1	1	1	0	0	0	i13													xd				
XVPERMI.W	**xd, xj, ui8**	0	1	1	1	0	1	1	1	1	1	1	0	0	1	ui8								xj					xd				
XVPERMI.D	**xd, xj, ui8**	0	1	1	1	0	1	1	1	1	1	1	0	1	0	ui8								xj					xd				
XVPERMI.Q	**xd, xj, ui8**	0	1	1	1	0	1	1	1	1	1	1	0	1	1	ui8								xj					xd				