Otter RISC-V Vector Extension Instruction Manual
Version 1.0

By Alexis Peralta

Table of Contents

The RISC-V OTTER Vector Extension Instruction Set

The RISC-V OTTER Vector Extension (RVV) is an addition to the RV32I instructions for the open source RISC-V architecture. This instruction set is based on a subset of the instructions presented in the version 0.7.1 draft of the "RISC-V 'V' Vector Extension" (https://github.com/riscv/riscv-v-spec/releases/tag/0.7.1).

Installing RISC-V Vector Extension binutils

To assemble the instructions, the GNU toolchain port for v0.7.x must be installed. The port is located at: https://github.com/riscv/riscv-gnu-toolchain/tree/rvv-0.7.x

Clone the toolchain using:

```
git clone --single-branch --branch rvv-0.7.x https://github.com/riscv/riscv-gnu-toolchain.git
cd riscv-gnu-toolchain
git submodule update --init –recursive
```

On Ubuntu, ensure the following libraries are installed to build the toolchain:

```
sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
```

To build the toolchain, run the following commands in Ubuntu:

```
./configure --prefix=/opt/riscv
make
export PATH=/opt/riscv/:$PATH
```

This could take a while. Instructions for building the toolchain on other operating systems can be found in the README file of the github repository located at: https://github.com/riscv/riscv-gnu-toolchain/tree/rvv-0.7.x.

There is currently no established gcc support for the RISC-V vector extension.

Assembling RISC-V Otter Vector Extension Assembly Instructions

After completing the installation and build of the RVV GNU toolchain, instructions can be assembled using the -march=rv32iv option:

```
riscv64-unknown-elf-gcc -c -o build/main.o src/main.s -O0 -march=rv32iv -mabi=ilp32
```

RISC-V OTTER Vector Extension Overview

Vector Terminology

Vector length (VL) refers to the number of elements which are stored in each vector. This value is stored in a CSR. An element is a specific value within a vector. Elements can have widths of 1 byte, 2 bytes, or 4 bytes. The term standard element width (SEW) refers to the current width of elements within a vector. SEW is also stored in a CSR. SEW occupies bits [4:2] of the vtype CSR. SEW is encoded according to the table below:

vtype[4:2]	SEW
000	8 bits
001	16 bits
010	32 bits

Vector Layout

Vectors are stored in the vector register file. Below is an example of a vector with a vector length of 8, and a SEW of 2 bytes (16 bits):

0 15	16 31	32 47	48 63	64 79	80 95	96 111	112 127
element 0	element 1	element 2	element 3	element 4	element 5	element 6	element 7

Figure 1. Vector Register Layout

Vector Masking

Some vector operations have a masked form. The vector mask is stored in vector register 0, v0. When masked operations are performed, the operation is only computed for the elements which the corresponding element in the vector mask has a value of 1, not 0. Elements for which the vector mask is 0 will retain the value of vs2 for ALU operations if vs2 is specifiable. For the masked vmv instruction, all elements for which the vector mask is 0 will be 0 in the destination vector. Vector masks can be written using vector load and store instructions. A vector mask for a vector with a length of 8 and SEW of 2 bytes could look like:

0 15	16 31	32 47	48 63	64 79	80 95	96 111	112 127
0	1	1	1	0	1	1	1

Figure 2. Vector Mask

For this mask, the operation would not be performed on elements 0 and 4.

The RISC-V OTTER Vector Extension Assembly Instruction Formats

The RVV extension has 11 different formats, shown in Table 1.

Instr Type	Instruction Format
VL	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 nf | mop | vm | lumop | rs1 | width | vd | VL
VLS	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 nf | mop | vm | rs2 | rs1 | width | vd | VLS
VLX	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 nf | mop | vm | vs2 | rs1 | width | vd | VLX
VS	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 nf | mop | vm | sumop | rs1 | width | vs3 | VS
VSS	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 nf | mop | vm | rs2 | rs1 | width | vs3 | VSS
VSX	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 nf | mop | vm | vs2 | rs1 | width | vs3 | VSX
VCI	31 | 30 20 | 19 15 | 14 12 | 11 7 | 6 0 0 | zimm[10:0] | rs1 | 111 | rd | VCI
VC	31 | 30 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 1 | 000000 | rs2 | rs1 | 111 | rd | VC
VV	31 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 funct6 | vm | vs2 | vs1 | 000 | vd | VV
VX	31 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 funct6 | vm | vs2 | rs1 | 100 | vd | VX
VI	31 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 funct6 | vm | vs2 | simm5 | 011 | vd | VI

Instruction Type: VL-type

Figure 3 shows the VL-type instruction format. Table 2 lists the instructions using the VL-type format.

VL	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 nf | mop | vm | lumop | rs1 | width | vd | VL

Figure 3. VL-type Instruction Format

	31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
vlb.v	X	100	vm	00000	rs1	000	vd	0000111
vlh.v	X	100	vm	00000	rs1	101	vd	0000111
vlw.v	X	100	vm	00000	rs1	110	vd	0000111
vlbu.v	X	000	vm	00000	rs1	000	vd	0000111
vlhu.v	X	000	vm	00000	rs1	101	vd	0000111
vlwu.v	X	000	vm	00000	rs1	110	vd	0000111

Table 2. Instructions using the VL-type

Instruction Type: VLS-type

VLS	31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
	nf	mop	vm	rs2	rs1	width	vd	VLS

Figure 4. VLS-type Instruction Format

	31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
vlsb.v	X	110	vm	rs2	rs1	000	vd	0000111
vlsh.v	X	110	vm	rs2	rs1	101	vd	0000111
vlsw.v	X	110	vm	rs2	rs1	110	vd	0000111
vlsbu.v	X	010	vm	rs2	rs1	000	vd	0000111
vlshu.v	X	010	vm	rs2	rs1	101	vd	0000111

	31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
vlswu.v	X	010	vm	rs2	rs1	110	vd	0000111

Table 3. Instructions using the VLS-type

Instruction Type: VLX-type

VLX	31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
	nf	mop	vm	vs2	rs1	width	vd	VLX

Figure 5. VLX-type Instruction Format

	31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
vlxb.v	X	111	vm	vs2	rs1	000	vd	0000111
vlxh.v	X	111	vm	vs2	rs1	101	vd	0000111
vlxw.v	X	111	vm	vs2	rs1	110	vd	0000111
vlxbu.v	X	011	vm	vs2	rs1	000	vd	0000111
vlxhu.v	X	011	vm	vs2	rs1	101	vd	0000111
vlxwu.v	X	011	vm	vs2	rs1	110	vd	0000111

Table 4. Instructions using the VLX-type

Instruction Type: VS-type

VS	31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
	nf	mop	vm	sumop	rs1	width	vs3	VS

Figure 6. VS-type Instruction Format

	31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
vsb.v	X	000	vm	00000	rs1	000	vs3	0100111

	31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
vsh.v	X	000	vm	00000	rs1	101	vs3	0100111
vsw.v	X	000	vm	00000	rs1	110	vs3	0100111

Table 5. Instructions using the VS-type

Instruction Type: VSS-type

VSS	31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
	nf	mop	vm	rs2	rs1	width	vs3	VSS

Figure 7. VSS-type Instruction Format

	31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
vssb.v	X	010	vm	rs2	rs1	000	vs3	0100111
vssh.v	X	010	vm	rs2	rs1	101	vs3	0100111
vssw.v	X	010	vm	rs2	rs1	110	vs3	0100111

Table 6. Instructions using the VSS-type

Instruction Type: VSX-type

VSX	31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
	nf	mop	vm	vs2	rs1	width	vs3	VSX

Figure 8. VSX-type Instruction Format

	31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
vsxb.v	X	011	vm	vs2	rs1	000	vs3	0100111
vsxh.v	X	011	vm	vs2	rs1	101	vs3	0100111
vsxw.v	X	011	vm	vs2	rs1	110	vs3	0100111

Table 7. Instructions using the VSX-type

Instruction Type: VCI-type

VCI	31	30 20	19 15	14 12	11 7	6 0
	0	zimm[10:0]	rs1	111	rd	VCI

Figure 9. VCI-type Instruction Format

vsetvli	31	30 20	19 15	14 12	11 7	6 0
	0	zimm[10:0]	rs1	111	rd	1010111

Table 8. Instructions using the VCI-type

Instruction Type: VC-type

VC	31	30 25	24 20	19 15	14 12	11 7	6 0
	1	000000	rs2	rs1	111	rd	VC

Figure 10. VC-type Instruction Format

vsetvl	31	30 25	24 20	19 15	14 12	11 7	6 0
	1	000000	rs2	rs1	111	rd	1010111

Table 9. Instructions using the VC-type

Instruction Type: VV-type

VV	31 26	25	24 20	19 15	14 12	11 7	6 0
	funct6	vm	vs2	vs1	000	vd	VV

Figure 11. VV-type Instruction Format

	31 26	25	24 20	19 15	14 12	11 7	6 0
vadd.vv	000000	vm	vs2	vs1	000	vd	1010111
vsub.vv	000010	vm	vs2	vs1	000	vd	1010111
vand.vv	001001	vm	vs2	vs1	000	vd	1010111
vor.vv	001010	vm	vs2	vs1	000	vd	1010111
vxor.vv	001011	vm	vs2	vs1	000	vd	1010111

	31 26	25	24 20	19 15	14 12	11 7	6 0
vmv.v.v	010111	vm	X	vs1	000	vd	1010111

Table 10. Instructions using the VV-type

Instruction Type: VX-type

VX	31 26	25	24 20	19 15	14 12	11 7	6 0
	funct6	vm	vs2	rs1	100	vd	VX

Figure 12. VX-type Instruction Format

	31 26	25	24 20	19 15	14 12	11 7	6 0
vadd.vx	000000	vm	vs2	rs1	100	vd	1010111
vsub.vx	000010	vm	vs2	rs1	100	vd	1010111
vand.vx	001001	vm	vs2	rs1	100	vd	1010111
vor.vx	001010	vm	vs2	rs1	100	vd	1010111
vxor.vx	001011	vm	vs2	rs1	100	vd	1010111
vmv.v.x	010111	vm	X	rs1	100	vd	1010111
vslideup.vx	001110	vm	vs2	rs1	100	vd	1010111
vslidedown.vx	001111	vm	vs2	rs1	100	vd	1010111

Table 11. Instructions using the VX-type

Instruction Type: VI-type

VI	31 26	25	24 20	19 15	14 12	11 7	6 0
	funct6	vm	vs2	simm5	011	vd	VI

Figure 13. VI-type Instruction Format

	31 26	25	24 20	19 15	14 12	11 7	6 0
vadd.vi	000000	vm	vs2	simm5	011	vd	1010111
vand.vi	001001	vm	vs2	simm5	011	vd	1010111
vor.vi	001010	vm	vs2	simm5	011	vd	1010111
vxor.vi	001011	vm	vs2	simm5	011	vd	1010111
vmv.v.i	010111	vm	X	simm5	011	vd	1010111
vslideup.vi	001110	vm	vs2	uimm5	100	vd	1010111
vslidedown.vi	001111	vm	vs2	uimm5	100	vd	1010111

Table 12. Instructions using the VI-type

RISC-V OTTER Vector Extension Assembly Instructions Brief Listing

Vector Configuration
vsetvl rd, rs1, rs2
vsetvli, rd, rs1, vtypei

Vector Load		
vlb.v vd, (rs1)	vlsb.v vd, (rs1), rs2	vlxb.v vd, (rs1), vs2
vlh.v vd, (rs1)	vlsh.v vd, (rs1), rs2	vlxh.v vd, (rs1), vs2
vlw.v vd, (rs1)	vlsw.v vd, (rs1), rs2	vlxw.v vd, (rs1), vs2
vlbu.v vd, (rs1)	vlsbu.v vd, (rs1), rs2	vlxbu.v vd, (rs1), vs2
vlhu.v vd, (rs1)	vlshu.v vd, (rs1), rs2	vlxhu.v vd, (rs1), vs2
vlwu.v vd, (rs1)	vlswu.v vd, (rs1), rs2	vlxwu.v vd, (rs1), vs2

Vector Store		
vsb.v vs3, (rs1)	vssb.v vs3, (rs1), rs2	vsxb.v vs3, (rs1), rs2
vsh.v vs3, (rs1)	vssh.v vs3, (rs1), rs2	vsxh.v vs3, (rs1), rs2
vsw.v vs3, (rs1)	vssw.v vs3, (rs1), rs2	vsxw.v vs3, (rs1), rs2

Vector Operations		
vadd.vv vd, vs2, vs1	vand.vi vd, vs2, imm	vxor.vi vd, vs2, imm
vadd.vx vd, vs2, rs1	vor.vv vd, vs2, vs1	vmv.v.v vd, vs2, vs1
vadd.vi vd, vs2, imm	vor.vx vd, vs2, rs1	vmv.v.x vd, vs2, rs1
vsub.vv vd, vs2, vs1	vor.vi vd, vs2, imm	vmv.v.i vd, vs2, imm
vsub.vx vd, vs2, rs1	vxor.vv vd, vs2, vs1	vslideup.vx vd, vs2, rs1
vand.vv vd, vs2, vs1	vxor.vx vd, vs2, rs1	vslideup.vi vd, vs2, uimm
vand.vx vd, vs2, rs1	vslidedown.vi vd, vs2, uimm	vslidedown.vx vd, vs2, rs1

Table 13. RVV Instruction format listing

RISC-V OTTER Vector Extension Assembly Instruction Overview

Table 14 lists RVV instructions including instruction format, description, and RTL description.

Instruction	Description	RTL
vsetvl rd, rs1, rs2	vector length and SEW configuration	`vl = rs1, vsew = rs2, rd = vl`
vsetvli, rd, rs1, vtypei	vector length and SEW configuration	`vl = rs1, vsew = imm, rd = vl`
vlb.v vd, (rs1)	unit-stride vector load bytes	`vd[i] = signext(M[X[rs1] + (i*unit-offset)])`
vlh.v vd, (rs1)	unit-stride vector load half-words	`vd[i] = signext(M[X[rs1] + (i*unit-offset)])`
vlw.v vd, (rs1)	unit-stride vector load words	`vd[i] = signext(M[X[rs1] + (i*unit-offset)])`
vlbu.v vd, (rs1)	unsigned unit-stride vector load bytes	`vd[i] = zeroext(M[X[rs1] + (i*unit-offset)])`
vlhu.v vd, (rs1)	unsigned unit-stride vector load half-words	`vd[i] = zeroext(M[X[rs1] + (i*unit-offset)])`
vlwu.v vd, (rs1)	unsigned unit-stride vector load words	`vd[i] = zeroext(M[X[rs1] + (i*unit-offset)])`

vlsb.v vd, (rs1), rs2	strided vector load bytes	`vd[i] = signext(M[X[rs1] + (i*X[rs2])])`
		`rs1 = base address`
		`rs2 = byte stride`

vlsh.v vd, (rs1), rs2	strided vector load half-words	`vd[i] = signext(M[X[rs1] + (i*X[rs2])])`
		`rs1 = base address`
		`rs2 = byte stride`

vlsw.v vd, (rs1), rs2	strided vector load words	`vd[i] = signext(M[X[rs1] + (i*X[rs2])])`
		`rs1 = base address`
		`rs2 = byte stride`

vlsbu.v vd, (rs1), rs2	unsigned strided vector load bytes	`RTL: vd[i] = zeroext(M[X[rs1] + (i*X[rs2])])`
		`rs1 = base address`
		`rs2 = byte stride`

vlshu.v vd, (rs1), rs2	unsigned strided vector load half-words	`vd[i] = zeroext(M[X[rs1] + (i*X[rs2])])`
		`rs1 = base address`
		`rs2 = byte stride`

vlswu.v vd, (rs1), rs2	unsigned strided vector load words	`vd[i] = zeroext(M[X[rs1] + (i*X[rs2])])`
		`rs1 = base address`
		`rs2 = byte stride`

vlxb.v vd, (rs1), vs2	indexed vector load bytes	`vd[i] = signext(M[X[rs1] + vs2[i]])`
		`rs1 = base address`
		`vs2 = byte indices`

vlxh.v vd, (rs1), vs2	indexed vector load half-words	`vd[i] = signext(M[X[rs1] + vs2[i]])`
		`rs1 = base address`
		`vs2 = byte indices`

vlxw.v vd, (rs1), vs2	indexed vector load words	`vd[i] = signext(M[X[rs1] + vs2[i]])`
		`rs1 = base address`
		`vs2 = byte indices`

vlxbu.v vd, (rs1), vs2	unsigned indexed vector load bytes	`vd[i] = zeroext(M[X[rs1] + vs2[i]])`
		`rs1 = base address`
		`vs2 = byte indices`

vlxhu.v vd, (rs1), vs2	unsigned indexed vector load half-words	`vd[i] = zeroext(M[X[rs1] + vs2[i]])`
		`rs1 = base address`
		`vs2 = byte indices`

vlxwu.v vd, (rs1), vs2	unsigned indexed vector load words	`vd[i] = zeroext(M[X[rs1] + vs2[i]])`
		`rs1 = base address`
		`vs2 = byte indices`

vsb.v vs3, (rs1)	unit-stride vector store bytes	`M[X[rs1] + (i*unit-offset)] = vs3[i]` `rs1 = base address`
vsh.v vs3, (rs1)	unit-stride vector store half-words	`M[X[rs1] + (i*unit-offset)] = vs3[i]` `rs1 = base address`
vsw.v vs3, (rs1)	unit-stride vector store words	`M[X[rs1] + (i*unit-offset)] = vs3[i]` `rs1 = base address`

vssb.v vs3, (rs1), rs2	strided vector store bytes	`M[X[rs1] + (i*X[rs2])] = vs3[i]`
		`rs1 = base address`
		`rs2 = byte stride`

vssh.v vs3, (rs1), rs2	strided vector store half-words	`M[X[rs1] + (i*X[rs2])] = vs3[i]`
		`rs1 = base address`
		`rs2 = byte stride`

vssw.v vs3, (rs1), rs2	strided vector store words	`M[X[rs1] + (i*X[rs2])] = vs3[i]`
		`rs1 = base address`
		`rs2 = byte stride`

vsxb.v vs3, (rs1), rs2	indexed vector store bytes	`M[X[rs1] + vs2[i]] = vs3[i]`

		`rs1 = base address` `vs2 = byte indices`

vsxh.v vs3, (rs1), rs2	indexed vector store half-words	`M[X[rs1] + vs2[i]] = vs3[i]`
		`rs1 = base address`
		`vs2 = byte indices`

vsxw.v vs3, (rs1), rs2	indexed vector store words	`M[X[rs1] + vs2[i]] = vs3[i]`
		`rs1 = base address`
		`vs2 = byte indices`

vadd.vv vd, vs2, vs1	vector-vector element addition	`vd[i] = vs2[i] + vs1[i]`
vadd.vx vd, vs2, rs1	vector-register element addition	`vd[i] = vs2[i] + rs1`
vadd.vi vd, vs2, imm	vector-immediate element addition	`vd[i] = vs2[i] + imm`
vsub.vv vd, vs2, vs1	vector-vector element subtraction	`vd[i] = vs2[i] - vs1[i]`
vsub.vx vd, vs2, rs1	vector-register element subtraction	`vd[i] = vs2[i] - rs1`
vand.vv vd, vs2, vs1	vector-vector element bitwise AND	`vd[i] = vs2[i] & vs1[i]`
vand.vx vd, vs2, rs1	vector-register element bitwise AND	`vd[i] = vs2[i] & rs1`
vand.vi vd, vs2, imm	vector-immediate element bitwise AND	`vd[i] = vs2[i] & imm`
vor.vv vd, vs2, vs1	vector-vector element bitwise OR	`vd[i] = vs2[i] | vs1[i]`
vor.vx vd, vs2, rs1	vector-register element bitwise OR	`vd[i] = vs2[i] | rs1`
vor.vi vd, vs2, imm	vector-immediate element bitwise OR	`vd[i] = vs2[i] | imm`
vxor.vv vd, vs2, vs1	vector-vector element bitwise XOR	`vd[i] = vs2[i] ^ vs1[i]`
vxor.vx vd, vs2, rs1	vector-register element bitwise XOR	`vd[i] = vs2[i] ^ rs1`
vxor.vi vd, vs2, imm	vector-immediate element bitwise XOR	`vd[i] = vs2[i] ^ imm`
vmv.v.v vd, vs2, vs1	move vector elements	`vd[i] = vs1[i]`
vmv.v.x vd, vs2, rs1	move register value into vector elements	`vd[i] = rs1`
vmv.v.i vd, vs2, imm	move immediate value into vector elements	`vd[i] = imm`
vslideup.vx vd, vs2, rs1	barrel shift vector elements upwards by register value	`RTL: vd[i+ X[rs1]] = vs2[i],(i+ X[rs1]) < vl`

		`vd[i] = vs2[X[rs1] + i]`
vslideup.vi vd, vs2, uimm	barrel shift vector elements upwards by immediate value	`vd[i+ uimm] = vs2[i], (i+ uimm) < vl` `vd[i] = vs2[uimm + i]`
vslidedown.vx vd, vs2, rs1	barrel shift vector elements downwards by register value	`vd[i] = vs2[i + rs1], (i+ rs1) < vl` `vd[i + rs1] = vs2[i]`
vslidedown.vi vd, vs2, uimm	barrel shift vector elements downwards by immediate value	`vd[i] = vs2[I + uimm], (i+ uimm) < vl` `vd[i + uimm] = vs2[i]`

Table 14. RVV Instruction Overview

Detailed RISC-V OTTER Vector Extension Assembly Instruction Description

The following section lists each of the OTTER RVV instructions in a detailed format.

vsetvl	**vector length and SEW configuration**	
RTL: vl = rs1, vsew = rs2, rd = vl	**Forms:**	vsetvl rd, rs1, rs2

Description: The vsetvl instruction adjusts the current vector length and standard element width CSRs. The new vector length is provided in rs1. The new standard element width setting is provided in rs2. rd is overwritten by the new vector length. Source registers rs1 and rs2 are not affected.

Instruction Format: (VX-type)

31	30–25	24–20	19–15	14–12	11–7	6–0
1	000000	rs2	rs1	111	rd	1010111

Usage:

```
vsetvl x3, x1, x2      # x1 is written to vl CSR, x2 type is written to
                       # vsew CSR, x3 is overwritten by the new vl value
```

See also: vsetvli

vsetvli	**vector length and SEW configuration**	
RTL: vl = rs1, vsew = imm, rd = vl	**Forms:**	vsetvli, rd, rs1, vtypei

Description: The vsetvli instruction adjusts the current vector length and standard element width CSRs. The new vector length is provided in rs1. The new standard element width setting is provided by the type encoded in the immediate value. rd is overwritten by the new vector length. Source register rs1 is not affected.

Instruction Format: (VI-type)

31	30–20	19–15	14–12	11–7	6–0
0	zimm[10:0]	rs1	111	rd	1010111

Usage:

```
e8    # SEW=8b
e16   # SEW=16b
e32   # SEW=32b

vsetvli x3, x1, e8     # x1 is written to vl CSR, e8 type is written to
                       # vsew CSR, x3 is overwritten by the new vl value
```

See also: vsetvl

vlb.v	**unit-stride vector load bytes**		
RTL: vd[i] = signext(M[X[rs1] + (i*unit-offset)])		**Forms:**	vlb.v vd, (rs1)
Description: The vlb.v instruction loads a vector from memory represented as a contiguous sequence of bytes starting at the memory address stored in rs1. The vlb.v instruction forms the memory address of each vector element by using the starting address provided in rs1 and adding 8 for each successive vector element. The value in memory is sign-extended and copied into its corresponding vector element starting with v[0].			
Instruction Format: (VL-type)	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 X | 100 | vm | 00000 | rs1 | 000 | vd | 0000111		
Usage:	vlb.v v4, (x5) # bytes starting at M[x5] are copied into v4, # starting at v4[0], x5 is not affected		
See also: vlbu.v, vle.v			

vlh.v	**unit-stride vector load half-words**		
RTL: vd[i] = signext(M[X[rs1] + (i*unit-offset)])		**Forms:**	vlh.v vd, (rs1)
Description: The vlh.v instruction loads a vector from memory represented as a contiguous sequence of 2 byte half-words starting at the memory address stored in rs1. The vlh.v instruction forms the memory address of each vector element by using the starting address provided in rs1 and adding 16 for each successive vector element. The value in memory is sign-extended and copied into its corresponding vector element starting with v[0].			
Instruction Format: (VL-type)	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 X | 100 | vm | 00000 | rs1 | 101 | vd | 0000111		
Usage:	vlh.v v4, (x5) # 16-bit half-words starting at M[x5] are copied # into v4, starting at v4[0]; x5 is not affected		
See also: vlhu.v, vle.v			

vlw.v	**unit-stride vector load words**		
RTL: vd[i] = signext(M[X[rs1] + (i*unit-offset)])		**Forms:**	vlw.v vd, (rs1)
Description: The vlw.v instruction loads a vector from memory represented as a contiguous sequence of 4 bytes words starting at the memory address stored in rs1. The vlw.v instruction forms the memory address of each vector element by using the starting address provided in rs1 and adding 32 for each successive vector element. The value in memory is sign-extended and copied into its corresponding vector element starting with v[0].			
Instruction Format: (VL-type)	31 29 / 28 26 / 25 / 24 20 / 19 15 / 14 12 / 11 7 / 6 0 X / 100 / vm / 00000 / rs1 / 110 / vd / 0000111		
Usage:	vlw.v v4, (x5) # 32-bit words starting at M[x5] are copied # into v4, starting at v4[0]; x5 is not affected		
See also: vlwu.v, vle.v			

vlbu.v	**unsigned unit-stride vector load bytes**		
RTL: vd[i] =zeroext(M[X[rs1] + (i*unit-offset)])		**Forms:**	vlbu.v vd, (rs1)
Description: The vlbu.v instruction loads a vector from memory represented as a contiguous sequence of bytes starting at the memory address stored in rs1. The vlbu.v instruction forms the memory address of each vector element by using the starting address provided in rs1 and adding 8 for each successive vector element. The value in memory is zero-extended and copied into its corresponding vector element starting with v[0].			
Instruction Format: (VL-type)	31 29 / 28 26 / 25 / 24 20 / 19 15 / 14 12 / 11 7 / 6 0 X / 000 / vm / 00000 / rs1 / 000 / vd / 0000111		
Usage:	vlbu.v v4, (x5) # bytes starting at M[x5] are copied into v4, # starting at v4[0], x5 is not affected		
See also: vlb.v, vle.v			

vlhu.v	**unsigned unit-stride vector load half-words**		
RTL: vd[i] =zeroext(M[X[rs1] + (i*unit-offset)])		**Forms:**	vlhu.v vd, (rs1)
Description: The vlhu.v instruction loads a vector from memory represented as a contiguous sequence of 2 byte half-words starting at the memory address stored in rs1. The vlhu.v instruction forms the memory address of each vector element by using the starting address provided in rs1 and adding 16 for each successive vector element. The value in memory is zero-extended and copied into its corresponding vector element starting with v[0].			
Instruction Format: (VL-type)	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 X | 000 | vm | 00000 | rs1 | 101 | vd | 0000111		
Usage:	vlhu.v v4, (x5) # 16-bit half-words starting at M[x5] are copied into # v4, starting at v4[0], x5 is not affected		
See also: vlh.v, vle.v			

vlwu.v	**unsigned unit-stride vector load words**		
RTL: vd[i] =zeroext(M[X[rs1] + (i*unit-offset)])		**Forms:**	vlwu.v vd, (rs1)
Description: The vlwu.v instruction loads a vector from memory represented as a contiguous sequence of 4 byte words starting at the memory address stored in rs1. The vlwu.v instruction forms the memory address of each vector element by using the starting address provided in rs1 and adding 32 for each successive vector element. The value in memory is zero-extended and copied into its corresponding vector element starting with v[0].			
Instruction Format: (VL-type)	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 X | 000 | vm | 00000 | rs1 | 110 | vd | 0000111		
Usage:	vlwu.v v4, (x5) # 32-bit words starting at M[x5] are copied into v4, # starting at v4[0], x5 is not affected		
See also: vlw.v, vle.v			

vlsb.v	**strided vector load bytes**		
RTL: vd[i] = signext(M[X[rs1] + (i*X[rs2])]) **rs1 = base address** **rs2 = byte stride**		**Forms:**	vlsb.v vd, (rs1), rs2
Description: The vlsb.v instruction loads a vector from memory from a contiguous set of byte-stride width values starting at the memory address stored in rs1. The byte stride is stored in rs2. The vlsb.v instruction forms the memory address of each vector element by using the starting address provided in rs1 and adding the byte stride provided in rs2 for each successive vector element. The value in memory is sign-extended and copied into its corresponding vector element starting with v[0].			
Instruction Format: (VLS-type)	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 X | 110 | vm | rs2 | rs1 | 000 | vd | 0000111		
Usage:	vlsb.v v4, (x5), x2 # 8-bit elements at x2 byte increments from # M[x5] are copied into v4; x5 and x2 not affected		
See also: vlb.v, vlxb.v			

vlsh.v	**strided vector load half words**		
RTL: vd[i] = signext(M[X[rs1] + (i*X[rs2])]) **rs1 = base address** **rs2 = byte stride**		**Forms:**	vlsh.v vd, (rs1), rs2
Description: The vlsh.v instruction loads a vector from memory from a contiguous set of byte-stride width values starting at the memory address stored in rs1. The byte stride is stored in rs2. The vlsh.v instruction forms the memory address of each vector element by using the starting address provided in rs1 and adding the byte stride provided in rs2 for each successive vector element. The value in memory is sign-extended and copied into its corresponding vector element starting with v[0].			
Instruction Format: (VLS-type)	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 X | 110 | vm | rs2 | rs1 | 101 | vd | 0000111		
Usage:	vlsh.v v4, (x5), x2 # 16-bit elements at x2 byte increments from # M[x5] are copied into v4; x5 and x2 not affected		
See also: vlh.v, vlxh.v			

`vlsw.v`	**strided vector load words**		
RTL: vd[i] = signext(M[X[rs1] + (i*X[rs2])]) **rs1 = base address** **rs2 = byte stride**		**Forms:**	vlsw.v vd, (rs1), rs2
Description: The vlsw.v instruction loads a vector from memory from a contiguous set of byte-stride width values starting at the memory address stored in rs1. The byte stride is stored in rs2. The vlsw.v instruction forms the memory address of each vector element by using the starting address provided in rs1 and adding the byte stride provided in rs2 for each successive vector element. The value in memory is sign-extended and copied into its corresponding vector element starting with v[0].			
Instruction Format: (VLS-type)	31 29 / 28 26 / 25 / 24 20 / 19 15 / 14 12 / 11 7 / 6 0 X / 110 / vm / rs2 / rs1 / 110 / vd / 0000111		
Usage:	vlsw.v v4, (x5), x2 # 32-bit elements at x2 byte increments from # M[x5] are copied into v4; x5 and x2 not affected		
See also: vlw.v, vlxw.v			

`vlsbu.v`	**unsigned strided vector load bytes**		
RTL: vd[i] = zeroext(M[X[rs1] + (i*X[rs2])]) **rs1 = base address** **rs2 = byte stride**		**Forms:**	vlsbu.v vd, (rs1), rs2
Description: The vlsbu.v instruction loads a vector from memory from a contiguous set of byte-stride width values starting at the memory address stored in rs1. The byte stride is stored in rs2. The vlsbu.v instruction forms the memory address of each vector element by using the starting address provided in rs1 and adding the byte stride provided in rs2 for each successive vector element. The value in memory is zero-extended and copied into its corresponding vector element starting with v[0].			
Instruction Format: (VLS-type)	31 29 / 28 26 / 25 / 24 20 / 19 15 / 14 12 / 11 7 / 6 0 X / 010 / vm / rs2 / rs1 / 000 / vd / 0000111		
Usage:	vlsbu.v v4, (x5), x2 # 8-bit elements at x2 byte increments from # M[x5] are copied into v4; x5 and x2 not affected		
See also: vlsb.v, vlxb.v			

vlshu.v	**unsigned strided vector load half-words**	
RTL: vd[i] = zeroext(M[X[rs1] + (i*X[rs2])]) **rs1 = base address** **rs2 = byte stride**	**Forms:**	vlshu.v vd, (rs1), rs2

Description: The vlshu.v instruction loads a vector from memory from a contiguous set of byte-stride width values starting at the memory address stored in rs1. The byte stride is stored in rs2. The vlshu.v instruction forms the memory address of each vector element by using the starting address provided in rs1 and adding the byte stride provided in rs2 for each successive vector element. The value in memory is zero-extended and copied into its corresponding vector element starting with v[0].

Instruction Format: (VLS-type)

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
X	010	vm	rs2	rs1	101	vd	0000111

Usage:

```
vlshu.v v4, (x5), x2    # 16-bit elements at x2 byte increments from
                        # M[x5] are copied into v4; x5 and x2 not affected
```

See also: vlsh.v, vlxh.v

vlswu.v	**unsigned strided vector load words**	
RTL: vd[i] = zeroext(M[X[rs1] + (i*X[rs2])]) **rs1 = base address** **rs2 = byte stride**	**Forms:**	vlswu.v vd, (rs1), rs2

Description: The vlswu.v instruction loads a vector from memory from a contiguous set of byte-stride width values starting at the memory address stored in rs1. The byte stride is stored in rs2. The vlswu.v instruction forms the memory address of each vector element by using the starting address provided in rs1 and adding the byte stride provided in rs2 for each successive vector element. The value in memory is zero-extended and copied into its corresponding vector element starting with v[0].

Instruction Format: (VLS-type)

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
X	010	vm	rs2	rs1	110	vd	0000111

Usage:

```
vlswu.v v4, (x5), x2    # 32-bit elements at x2 byte increments from
                        # M[x5] are copied into v4; x5 and x2 not affected
```

See also: vlsw.v, vlxw.v

vlxb.v	**indexed vector load bytes**		
RTL: vd[i] = signext(M[X[rs1] + vs2[i]]) **rs1 = base address** **vs2 = byte indices**		**Forms:**	vlxb.v vd, (rs1), vs2
Description: The vlxb.v instruction loads a vector from memory from a set of memory offsets specified in vs2 and the base address in rs1. The byte indices are stored in vs2. The vlxb.v instruction forms the memory address of each vector element by adding the byte index provided in vs2 for each element to the base address. The value in memory is sign-extended and copied into its corresponding vector element starting with v[0].			
Instruction Format: (VLX-type)	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 X | 111 | vm | vs2 | rs1 | 000 | vd | 0000111		
Usage:	vlxb.v v4, (x5), v3 # 8-bit elements at M[x5 + v3[i]] are copied into # v4 starting at v4[0], x5 and v3 not affected		
See also: vlb.v, vlsb.v			

vlxh.v	**indexed vector load half-words**		
RTL: vd[i] = signext(M[X[rs1] + vs2[i]]) **rs1 = base address** **vs2 = byte indices**		**Forms:**	vlxh.v vd, (rs1), vs2
Description: The vlxh.v instruction loads a vector from memory from a set of memory offsets specified in vs2 and the base address in rs1. The byte indices are stored in vs2. The vlxh.v instruction forms the memory address of each vector element by adding the byte index provided in vs2 for each element to the base address. The value in memory is sign-extended and copied into its corresponding vector element starting with v[0].			
Instruction Format: (VLX-type)	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 X | 111 | vm | vs2 | rs1 | 101 | vd | 0000111		
Usage:	vlxh.v v4, (x5), v3 # 16-bit elements at M[x5 + v3[i]] are copied into # v4 starting at v4[0], x5 and v3 not affected		
See also: vlh.v, vlsh.v			

vlxw.v	

RTL: vd[i] = signext(M[X[rs1] + vs2[i]]) **rs1 = base address** **vs2 = byte indices**	**Forms:**	vlxw.v vd, (rs1), vs2

Description: The vlxw.v instruction loads a vector from memory from a set of memory offsets specified in vs2 and the base address in rs1. The byte indices are stored in vs2. The vlxw.v instruction forms the memory address of each vector element by adding the byte index provided in vs2 for each element to the base address. The value in memory is sign-extended and copied into its corresponding vector element starting with v[0].

Instruction Format: (VLX-type)

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
X	111	vm	vs2	rs1	110	vd	0000111

Usage:

```
vlxw.v v4, (x5), v3     # 32-bit elements at M[x5 + v3[i]] are copied into
                        # v4 starting at v4[0], x5 and v3 not affected
```

See also: vlw.v, vlsw.v

vlxbu.v	**unsigned index vector load half-words**

RTL: vd[i] = zeroext(M[X[rs1] + vs2[i]]) **rs1 = base address** **vs2 = byte indices**	**Forms:**	vlxbu.v vd, (rs1), vs2

Description: The vlxbu.v instruction loads a vector from memory from a set of memory offsets specified in vs2 and the base address in rs1. The byte indices are stored in vs2. The vlxbu.v instruction forms the memory address of each vector element by adding the byte index provided in vs2 for each element to the base address. The value in memory is zero-extended and copied into its corresponding vector element starting with v[0].

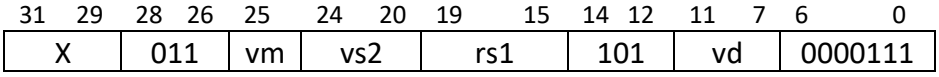
Instruction Format: (VLX-type)

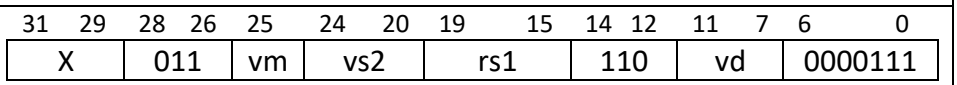
31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
X	011	vm	vs2	rs1	000	vd	0000111

Usage:

```
vlxbu.v v4, (x5), v3    # 8-bit elements at M[x5 + v3[i]] are copied into
                        # v4 starting at v4[0], x5 and v3 not affected
```

See also: vlxb.v, vlsb.v

vlxhu.v	**unsigned indexed vector load half words**	
RTL: vd[i] = zeroext(M[X[rs1] + vs2[i]]) **rs1 = base address** **vs2 = byte indices**	**Forms:**	vlxhu.v vd, (rs1), vs2
Description: The vlxhu.v instruction loads a vector from memory from a set of memory offsets specified in vs2 and the base address in rs1. The byte indices are stored in vs2. The vlxhu.v instruction forms the memory address of each vector element by adding the byte index provided in vs2 for each element to the base address. The value in memory is zero-extended and copied into its corresponding vector element starting with v[0].		
Instruction Format: (VLX-type)		
Usage:	vlxhu.v v4, (x5), v3 # 16-bit elements at M[x5 + v3[i]] are copied into # v4 starting at v4[0], x5 and v3 not affected	
See also: vlxh.v, vlsh.v		

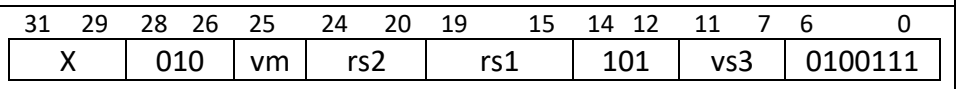
vlxwu.v	**unsigned indexed vector load words**	
RTL: vd[i] = zeroext(M[X[rs1] + vs2[i]]) **rs1 = base address** **vs2 = byte indices**	**Forms:**	vlxwu.v vd, (rs1), vs2
Description: The vlxwu.v instruction loads a vector from memory from a set of memory offsets specified in vs2 and the base address in rs1. The byte indices are stored in vs2. The vlxwu.v instruction forms the memory address of each vector element by adding the byte index provided in vs2 for each element to the base address. The value in memory is zero-extended and copied into its corresponding vector element starting with v[0].		
Instruction Format: (VLX-type)		
Usage:	vlxwu.v v4, (x5), v3 # 32-bit elements at M[x5 + v3[i]] are copied into # v4 starting at v4[0], x5 and v3 not affected	
See also: vlw.v, vlsw.v		

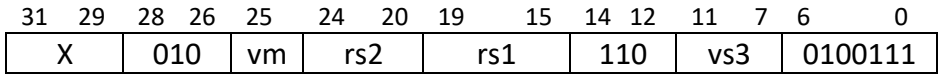
vsb.v	**unit-stride vector store bytes**		
RTL: M[X[rs1] + (i*unit-offset)] = vs3[i] **rs1 = base address**		**Forms:**	vsb.v vs3, (rs1)
Description: The vsb.v instruction stores the vector elements from vs3 as a contiguous sequence of bytes in memory starting at rs1. The memory address which each element is stored at is calculated as the base address added to the element's index number of bytes. The values in vs3 and rs1 are unaffected.			
Instruction Format: (VS-type)	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 X | 000 | vm | 00000 | rs1 | 000 | vs3 | 0100111		
Usage:	vsb.v v3, (x2) # v3[i] is stored at M[x2 + (i * 1 bytes)]		
See also: vssb.v, vsxb.v			

vsh.v	**unit-stride vector store half-words**		
RTL: M[X[rs1] + (i*unit-offset)] = vs3[i] **rs1 = base address**		**Forms:**	vsh.v vs3, (rs1)
Description: The vsh.v instruction stores the vector elements from vs3 as a contiguous sequence of half-words in memory starting at rs1. The memory address which each element is stored at is calculated as the base address added to the element's index number of 2-byte half-words. The values in vs3 and rs1 are unaffected.			
Instruction Format: (VS-type)	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 X | 000 | vm | 00000 | rs1 | 101 | vs3 | 0100111		
Usage:	vsh.v v3, (x2) # v3[i] is stored at M[x2 + (i * 2 bytes)]		
See also: vssh.v vsxh.v			

vsw.v	**unit-stride vector store words**		
RTL: M[X[rs1] + (i*unit-offset)] = vs3[i] **rs1 = base address**		**Forms:**	vsw.v vs3, (rs1)
Description: The vsw.v instruction stores the vector elements from vs3 as a contiguous sequence of words in memory starting at rs1. The memory address which each element is stored at is calculated as the base address added to the element's index number of 4-byte half-words. The values in vs3 and rs1 are unaffected.			
Instruction Format: (VS-type)	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 X | 000 | vm | 00000 | rs1 | 110 | vs3 | 0100111		
Usage:	vsw.v v3, (x2) # v3[i] is stored at M[x2 + (i * 4 bytes)]		
See also: vssw.v, vsxw.v			

vssb.v	**strided vector store bytes**		
RTL: M[X[rs1] + (i*X[rs2])] = vs3[i] **rs1 = base address** **rs2 = byte stride**		**Forms:**	vssb.v vs3, (rs1), rs2
Description: The vssb.v instruction stores the vector elements from vs3 as a contiguous sequence of byte-stride width elements in memory starting at rs1. The memory address which each element is stored at is calculated as the base address added to the element's index number of byte-stride width elements. The values in vs3 and rs1 are unaffected.			
Instruction Format: (VSS-type)	31 29 | 28 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 X | 010 | vm | rs2 | rs1 | 000 | vs3 | 0100111		
Usage:	vssb.v v3, (x4), x5 # v3[i] is stored at M[x4 + (i * x5)]		
See also: vsb.v, vsxb.v			

vssh.v	**strided vector store half-words**	
RTL: M[X[rs1] + (i*X[rs2])] = vs3[i] **rs1 = base address** **rs2 = byte stride**	**Forms:**	vssh.v vs3, (rs1), rs2
Description: The vssh.v instruction stores the vector elements from vs3 as a contiguous sequence of byte-stride width elements in memory starting at rs1. The memory address which each element is stored at is calculated as the base address added to the element's index number of byte-stride width elements. The values in vs3 and rs1 are unaffected.		
Instruction Format: (VSS-type)		
Usage:	vssh.v v3, (x4), x5 # v3[i] is stored at M[x4 + (i * x5)]	
See also: vsh.v, vsxh.v		

vssw.v	**strided vector store words**	
RTL: M[X[rs1] + (i*X[rs2])] = vs3[i] **rs1 = base address** **rs2 = byte stride**	**Forms:**	vssw.v vs3, (rs1), rs2
Description: The vssw.v instruction stores the vector elements from vs3 as a contiguous sequence of byte-stride width elements in memory starting at rs1. The memory address which each element is stored at is calculated as the base address added to the element's index number of byte-stride width elements. The values in vs3 and rs1 are unaffected.		
Instruction Format: (VSS-type)		
Usage:	vssw.v v3, (x4), x5 # v3[i] is stored at M[x4 + (i * x5)]	
See also: vsw.v, vsxw.v		

vsxb.v	**indexed vector store bytes**

RTL: M[X[rs1] + vs2[i]] = vs3[i] **rs1 = base address** **vs2 = byte indices**	**Forms:**	vsxb.v vs3, (rs1), vs2

Description: The vsxb.v instruction stores the vector elements from vs3 at the byte indices specified by vs2. The memory address which each element is stored at is calculated as the base address added to the byte offset at the corresponding element in vs2. The values in vs3 and rs1 are unaffected.

Instruction Format: (VSX-type)

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
X	011	vm	vs2	rs1	000	vs3	0100111

Usage: vsxb.v v3, (x6), v4 # v3[i] is stored at M[x6 + v4[i]]

See also: vsb.v, vssb.v

vsxh.v	**indexed vector store half-words**

RTL: M[X[rs1] + vs2[i]] = vs3[i] **rs1 = base address** **vs2 = byte indices**	**Forms:**	vsxh.v vs3, (rs1), vs2

Description: The vsxh.v instruction stores the vector elements from vs3 at the byte indices specified by vs2. The memory address which each element is stored at is calculated as the base address added to the byte offset at the corresponding element in vs2. The values in vs3 and rs1 are unaffected.

Instruction Format: (VSX-type)

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
X	011	vm	vs2	rs1	101	vs3	0100111

Usage: vsxh.v v3, (x6), v4 # v3[i] is stored at M[x6 + v4[i]]

See also: vsh.v, vssh.v

vsxw.v	**indexed vector store words**

RTL: M[X[rs1] + vs2[i]] = vs3[i] **rs1 = base address** **vs2 = byte indices**	**Forms:**	vsxw.v vs3, (rs1), vs2

Description: The vsxw.v instruction stores the vector elements from vs3 at the byte indices specified by vs2. The memory address which each element is stored at is calculated as the base address added to the byte offset at the corresponding element in vs2. The values in vs3 and rs1 are unaffected.

Instruction Format: (VSX-type)

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
X	011	vm	vs2	rs1	110	vs3	0100111

Usage:

```
vsxw.v v3, (x6), v4   # v3[i] is stored at M[x6 + v4[i]]
```

See also: vsw.v, vssw.v

vadd.vv	**Vector-vector element addition**

RTL: vd[i] = vs2[i] + vs1[i]	**Forms:**	vadd.vv vd, vs2, vs1 vadd.vv vd, vs2, vs1, vm

Description: The vadd.vv instruction performs addition on each element of vs2 and vs1 and stores the result in vd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same vector register as the destination. Both source operands are treated as signed values in 2's complement format. The add instruction ignores any arithmetic overflow resulting from the operation. The masked version of this instruction only copies the elements according to the vector mask stored in v0. Elements excluded by the mask will assume the value in vs2.

Instruction Format: (VV-type)

31 26	25	24 20	19 15	14 12	11 7	6 0
000000	vm	vs2	vs1	000	vd	1010111

Usage:

```
vadd.vv v3, v3, v2        # addition of v2 and v3
                          # result stored in v3, v2 not affected
vadd.vv v3, v3, v2, vm    # addition of v2 and v3 where vm[i] == 1
                          # result stored in v3, v2 not affected
```

vadd.vx	**vector-register element addition**	
RTL: vd[i] = vs2[i] + rs1	**Forms:**	vadd.vx vd, vs2, rs1 vadd.vx vd, vs2, rs1, vm

Description: The vadd.vx instruction performs addition on each element of vs2 and rs1 and stores the result in vd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same vector register as the destination. Both source operands are treated as signed values in 2's complement format. The add instruction ignores any arithmetic overflow resulting from the operation. The masked version of this instruction only copies the elements according to the vector mask stored in v0. Elements excluded by the mask will assume the value in vs2.

Instruction Format: (VX-type)

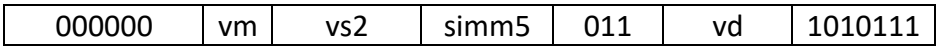
31 26	25	24 20	19 15	14 12	11 7	6 0
000000	vm	vs2	rs1	100	vd	1010111

Usage:

```
vadd.vx v3, v3, x2        # addition of v3 and x2
                          # result stored in v3, x2 not affected
vadd.vx v3, v3, x2, vm    # addition of v3 and x2 where vm[i] == 1
                          # result stored in v3, x2 not affected
```

See also: vadd.vv, vadd.vi

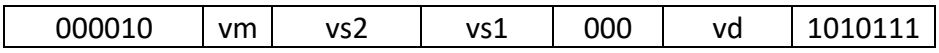
See also: vadd.vx, vadd.vi

vadd.vi	**vector-immediate element addition**		
RTL: vd[i] = vs2[i] + imm		**Forms:**	vadd.vi vd, vs2, imm vadd.vi vd, vs2, imm, vm
Description: The vadd.vi instruction performs addition on each element of vs2 and an immediate. value and stores the result in vd. The instruction overwrites value in the destination operand; source operand is not affected unless it specifies the same vector register as the destination. Both source operands are treated as signed values in 2's complement format. The add instruction ignores any arithmetic overflow resulting from the operation. The immediate value is truncated to the size of the current standard element width. The masked version of this instruction only copies the elements according to the vector mask stored in v0. Elements excluded by the mask will assume the value in vs2.			
Instruction Format: (VI-type)			
Usage:	vadd.vi v3, v3, 5 # addition of v3 and 5 # result stored in v3 vadd.vi v3, v3, 5, vm # addition of v3 and 5 where vm[i] == 1 # result stored in v3		
See also: vadd.vv, vadd.vx			

vsub.vv	**vector-vector element subtraction**	
RTL: vd[i] = vs2[i] - vs1[i]	**Forms:**	vsub.vv vd, vs2, vs1 vsub.vv vd, vs2, vs1, vm

Description: The vsub.vv instruction performs subtraction on each element of vs2 and vs1 and stores the result in vd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same vector register as the destination. Both source operands are treated as signed values in 2's complement format. The masked version of this instruction only copies the elements according to the vector mask stored in v0. Elements excluded by the mask will assume the value in vs2.

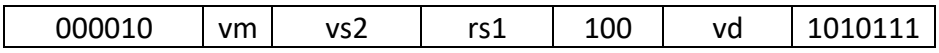
Instruction Format: (-type)



Usage:

```
vsub.vv v3, v3, v2        # subtraction of v2 from v3
                          # result stored in v3, v2 not affected
vsub.vv v3, v3, v2, vm    # subtraction of v2 from v3 where vm[i] == 1
                          # result stored in v3, v2 not affected
```

See also: vsub.vx

vsub.vx	**vector -register element subtraction**		
RTL: vd[i] = vs2[i] - rs1[i]		**Forms:**	vsub.vx vd, vs2, rs1 vsub.vx vd, vs2, rs1, vm
Description: The vsub.vx instruction performs subtraction on each element of vs2 and rs1 and stores the result in vd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same vector register as the destination. Both source operands are treated as signed values in 2's complement format. The masked version of this instruction only copies the elements according to the vector mask stored in v0. Elements excluded by the mask will assume the value in vs2.			
Instruction Format: (VX-type)			
Usage:	vsub.vx v3, v3, x2 # subtraction of x2 from v3 # result stored in v3, x2 not affected vsub.vx v3, v3, x2, vm # subtraction of x2 from v3 where vm[i] == 1 # result stored in v3, x2 not affected		
See also: vsub.vv			

vand.vv	**vector-vector element bitwise AND**

RTL: vd[i] = vs2[i] & vs1[i]	**Forms:**	vand.vv vd, vs2, vs1 vand.vv vd, vs2, vs1, vm

Description: The vand.vv instruction performs a bitwise AND on each element of vs2 and vs1 and stores the result in vd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same vector register as the destination. The masked version of this instruction only copies the elements according to the vector mask stored in v0. Elements excluded by the mask will assume the value in vs2.

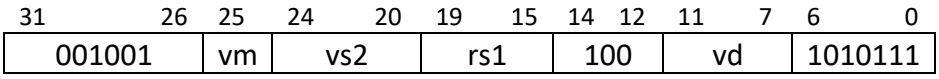
Instruction Format: (VV-type)

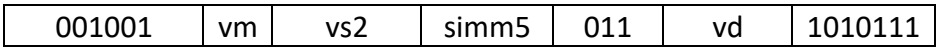
31 26	25	24 20	19 15	14 12	11 7	6 0
001001	vm	vs2	vs1	000	vd	1010111

Usage:

```
vand.vv v3, v3, v2          # bitwise AND of v2 and v3
                            # result stored in v3, v2 not affected
vand.vv v3, v3, v2, vm      # bitwise AND of v2 and v3 where vm[i] == 1
                            # result stored in v3, v2 not affected
```

See also: vand.vx, vand.vi

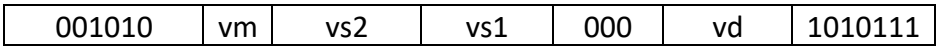
vand.vx	**vector-register element bitwise AND**		
RTL: vd[i] = vs2[i] & rs1[i]		**Forms:**	vand.vx vd, vs2, rs1 vand.vx vd, vs2, rs1, vm
Description: The vand.vx instruction performs a. bitwise AND on each element of vs2 and rs1 and stores the result in vd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same vector register as the destination. The masked version of this instruction only copies the elements according to the vector mask stored in v0. Elements excluded by the mask will assume the value in vs2.			
Instruction Format: (VX-type)			
Usage:	vand.vx v3, v3, x2 # bitwise AND of x2 and v3 # result stored in v3, x2 not affected vand.vx v3, v3, x2, vm # bitwise AND of x2 and v3 where vm[i] == 1 # result stored in v3, x2 not affected		
See also: vand.vv, vand.vi			

vand.vi	**vector-immediate bitwise AND**		
RTL: vd[i] = vs2[i] & imm		**Forms:**	vand.vi vd, vs2, imm vand.vi vd, vs2, imm, vm
Description: The vand.vi instruction performs a bitwise AND on each element of vs2 and the immediate value and stores the result in vd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same vector register as the destination. The masked version of this instruction only copies the elements according to the vector mask stored in v0. Elements excluded by the mask will assume the value in vs2.			
Instruction Format: (VI-type)			
Usage:	vand.vi v3, v3, 2 # bitwise AND of 2 and v3 # result stored in v3 vand.vi v3, v3, 2, vm # bitwise AND of 2 and v3 where vm[i] == 1 # result stored in v3		
See also: vand.vv, vand.vx			

vor.vv	**vector-vector element bitwise OR**	
RTL: vd[i] = vs2[i] | vs1[i]	**Forms:**	vor.vv vd, vs2, vs1 vor.vv vd, vs2, vs1, vm

Description: The vor.vv instruction performs a bitwise OR on each element of vs2 and vs1 and stores the result in vd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same vector register as the destination. The masked version of this instruction only copies the elements according to the vector mask stored in v0. Elements excluded by the mask will assume the value in vs2.

Instruction Format: (VV-type)



Usage:

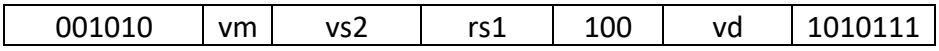
```
vor.vv v3, v3, v2        # bitwise OR of v2 and v3
                           # result stored in v3, v2 not affected
vor.vv v3, v3, v2, vm    # bitwise OR of v2 and v3 where vm[i] == 1
                           # result stored in v3, v2 not affected
```

See also: vor.vx, vor.vi

vor.vx	**vector-register element bitwise OR**	
RTL: vd[i] = vs2[i] | rs1[i]	**Forms:**	vor.vx vd, vs2, rs1 vor.vx vd, vs2, rs1, vm

Description: The vor.vx instruction performs a. bitwise OR on each element of vs2 and rs1 and stores the result in vd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same vector register as the destination. The masked version of this instruction only copies the elements according to the vector mask stored in v0. Elements excluded by the mask will assume the value in vs2.

Instruction Format: (VX-type)



Usage:

```
vor.vx v3, v3, x2        # bitwise OR of x2 and v3
                         # result stored in v3, x2 not affected
vor.vx v3, v3, x2, vm    # bitwise OR of x2 and v3 where vm[i] == 1
                         # result stored in v3, x2 not affected
```

See also: vor.vv, vor.vi

vor.vi	**vector-immediate element bitwise OR**	
RTL: vd[i] = vs2[i] | imm	**Forms:**	vor.vi vd, vs2, imm vor.vi vd, vs2, imm, vm

Description: The vor.vi instruction performs a bitwise OR on each element of vs2 and the immediate value and stores the result in vd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same vector register as the destination. The masked version of this instruction only copies the elements according to the vector mask stored in v0. Elements excluded by the mask will assume the value in vs2.

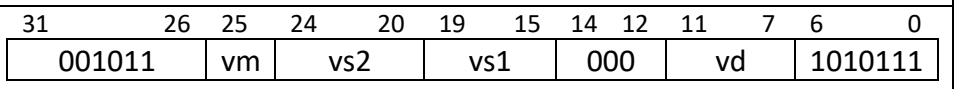
Instruction Format: (VI-type)

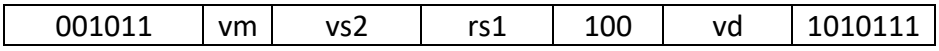
31 26	25	24 20	19 15	14 12	11 7	6 0
001010	vm	vs2	simm5	011	vd	1010111

Usage:

```
vor.vi v3, v3, 2          # bitwise OR of 2 and v3
                          # result stored in v3
vor.vi v3, v3, 2, vm      # bitwise OR of 2 and v3 where vm[i] == 1
                          # result stored in v3
```

See also: vor.vv, vor.vx

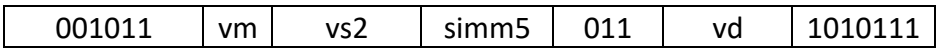
vxor.vv	**vector-vector element bitwise XOR**		
RTL: vd[i] = vs2[i] ^ vs1[i]		**Forms:**	vxor.vv vd, vs2, vs1 vxor.vv vd, vs2, vs1, vm
Description: The vxor.vv instruction performs a bitwise XOR on each element of vs2 and vs1 and stores the result in vd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same vector register as the destination. The masked version of this instruction only copies the elements according to the vector mask stored in v0. Elements excluded by the mask will assume the value in vs2.			
Instruction Format: (VV-type)			
Usage:	vxor.vv v3, v3, v2 # bitwise XOR of v2 and v3 # result stored in v3, v2 not affected vxor.vv v3, v3, v2, vm # bitwise XOR of v2 and v3 where vm[i] == 1 # result stored in v3, v2 not affected		
See also: vxor.vx, vxor.vi			

vxor.vx	**vector-register bitwise XOR**		
RTL: vd[i] = vs2[i] ^ rs1[i]		**Forms:**	vxor.vx vd, vs2, rs1 vxor.vx vd, vs2, rs1, vm
Description: The vxor.vx instruction performs a bitwise XOR on each element of vs2 and rs1 and stores the result in vd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same vector register as the destination. The masked version of this instruction only copies the elements according to the vector mask stored in v0. Elements excluded by the mask will assume the value in vs2.			
Instruction Format: (VX-type)			
Usage:	vxor.vx v3, v3, x2 # bitwise XOR of x2 and v3 # result stored in v3, x2 not affected vxor.vx v3, v3, x2, vm # bitwise XOR of x2 and v3 where vm[i] == 1 # result stored in v3, x2 not affected		
See also: vxor.vv, vxor.vi			

vxor.vi	**vector-immediate element bitwise XOR**	
RTL: vd[i] = vs2[i] ^ imm	**Forms:**	vxor.vi vd, vs2, imm vxor.vi vd, vs2, imm, vm

Description: The vxor.vi instruction performs a bitwise XOR on each element of vs2 and the immediate value and stores the result in vd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same vector register as the destination. The masked version of this instruction only copies the elements according to the vector mask stored in v0. Elements excluded by the mask will assume the value in vs2.

Instruction Format: (VI-type)



Usage:

```
vxor.vi v3, v3, 2          # bitwise XOR of 2 and v3
                           # result stored in v3
vxor.vi v3, v3, 2, vm      # bitwise XOR of 2 and v3 where vm[i] == 1
                           # result stored in v3
```

See also: vxor.vv, vxor.vx

vmv.v.v	**move vector elements**

RTL: vd[i] = vs1[i]	**Forms:**	vmv.v.v vd, vs1 vmv.v.v vd, vs1, vm

Description: The vmv.v.v instruction moves each element of vs1 into vd. The instruction overwrites value in the destination operand; The source operand is not affected unless it specifies the same vector register as the destination. The masked version of this instruction only copies the elements according to the vector mask stored in v0. The masked version zeroes out elements excluded by the vector mask.

Instruction Format: (VV-type)

31 26	25	24 20	19 15	14 12	11 7	6 0
010111	vm	X	vs1	000	vd	1010111

Usage:

```
vmv.v.v v3, v2      # moves the elements of v2 to v3
                    # result stored in v3, v2 not affected
vmv.v.v v3, v2, vm  # moves the elements of v2 to v3 where vm[i] == 1
                    # result stored in v3, v2 not affected
```

See also: vmv.v.x, vmv.v.v.i

vmv.v.x	**move register contents into vector elements**

RTL: vd[i] = rs1[i]	**Forms:**	vmv.v.x vd, rs1 vmv.v.x vd, rs1, vm

Description: The vmv.v.x instruction moves the value of rs1 into each element of vd. The instruction overwrites value in the destination operand; The source operand is not affected unless it specifies the same vector register as the destination. The masked version of this instruction only copies the elements according to the vector mask stored in v0. The masked version zeroes out elements excluded by the vector mask.

Instruction Format: (VX-type)

31 26	25	24 20	19 15	14 12	11 7	6 0
010111	vm	X	rs1	100	vd	1010111

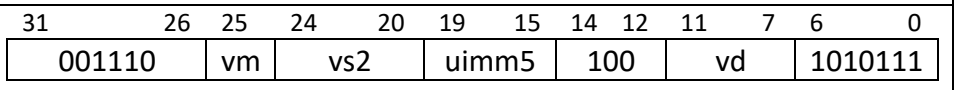
Usage:

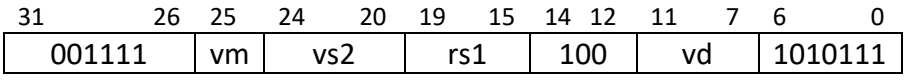
```
vmv.v.x v3, x2      # moves the contents of x2 to v3
                    # result stored in v3, x2 not affected
vmv.v.x v3, x2, vm  # moves the contents of x2 to v3 where vm[i] == 1
                    # result stored in v3, x2 not affected
```

See also: vmv.v.v, vmv.v.v.i

vmv.v.i	**move immediate value into vector elements**		
RTL: vd[i] = imm		**Forms:**	vmv.v.i vd, imm vmv.v.ivd, imm, vm
Description: The vmv.v.x instruction moves the immediate value into each element of vd. The instruction overwrites value in the destination operand. The masked version of this instruction only copies the elements according to the vector mask stored in v0. The masked version zeroes out elements excluded by the vector mask.			
Instruction Format: (VI-type)	31 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 010111 | vm | X | simm5 | 011 | vd | 1010111		
Usage:	vmv.v.i v3, 2 # moves 2 into each element of v3 # result stored in v3 vmv.v.i v3, 2, vm # moves 2 to into each element of v3 where # vm[i] == 1, result stored in v3		
See also: vmv.v.v, vmv.v.v.x			

vslideup.vx	**upwards barrel shift of vector elements**		
RTL: vd[i+ rs1] = vs2[i], (i+ rs1) < vl **vd[i] = vs2[rs1 + i]**		**Forms:**	vslideup.vx vd, vs2, rs1
Description: The vslideup.vx instruction performs a barrel shift upwards of the elements of vs2 by the number of elements given in rs1. The result is stored in vd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same vector register as the destination.			
Instruction Format: (VX-type)	31 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 001110 | vm | vs2 | rs1 | 100 | vd | 1010111		
Usage:	vslideup.vx v3, v4, x2 # barrel shift up elements of v4 by x2 # result stored in v3, v4 not affected		
See also: vslideup.vi			

vslideup.vi	**upwards barrel shift of vector elements**		
RTL: vd[i+ uimm] = vs2[i], (i+ uimm) < vl **vd[i] = vs2[uimm + i]**		**Forms:**	vslideup.vi vd, vs2, uimm[4:0]
Description: The vslideup.vi instruction performs a barrel shift upwards of the elements of vs2 by the number of elements given in the unsigned immediate value. The result is stored in vd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same vector register as the destination.			
Instruction Format: (VI-type)			
Usage:	vslideup.vi v3, v4, 2 # barrel shift up elements of v4 by 2 # result stored in v3, v4 not affected		
See also: vslideup.vx			

vslidedown.vx	**downwards barrel shift of vector elements**		
RTL: vd[i] = vs2[i + rs1], (i+ rs1) < vl **vd[i + rs1] = vs2[i]**		**Forms:**	vslidedown.vx vd, vs2, rs1
Description: The vslidedown.vx instruction performs a barrel shift downwards of the elements of vs2 by the number of elements given in rs1. The result is stored in vd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same vector register as the destination.			
Instruction Format: (VX-type)			
Usage:	vslidedown.vx v3, v4, x2 # barrel shift down elements of v4 by x2 # result stored in v3, v4 not affected		
See also: vslidedown.vi			

vslidedown.vi	**downwards barrel shift of vector elements**		
RTL: vd[i] = vs2[I + uimm], (i+ uimm) < vl **vd[i + uimm] = vs2[i]**		**Forms:**	vslidedown.vi vd, vs2, uimm[4:0]
Description: The vslidedown.vi instruction performs a barrel shift down of the elements of vs2 by the number of elements given in the unsigned immediate value. The result is stored in vd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same vector register as the destination.			
Instruction Format: (VI-type)	31 26 / 25 / 24 20 / 19 15 / 14 12 / 11 7 / 6 0 001111 / vm / vs2 / uimm5 / 100 / vd / 1010111		
Usage:	vslidedown.vi v3, v4, 2 # barrel shift down elements of v4 by 2 # result stored in v3, v4 not affected		
See also: vslidedown.vx			

RISC-V OTTER Vector Extension Assembly Examples

Vector Implementation of memcpy

```
.data
source: .byte 1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,
              2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32
dest: .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
.text

.global main
.type main, @function
main:
      la x5, source
      la x6, dest
      li x4, 32
      li x3, 16
      vsetvli x1, x3, e8
loop:
      vlb.v v1, (x5)
      add x5, x5, x1
      sub x4, x4, x1
      vsb.v v1, (x6)
      add x6, x6, x1
      bnez x4, loop
      j main
```

Vector-Vector Add

```
.data
vector1: .byte 1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31
vector2: .byte 2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32
vector3: .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
.text

.global main
.type main, @function
main:
      la x5, vector1
      la x6, vector2
      la x7, vector3
      li x4, 16
      vsetvli x0,x4,e8
      vlb.v v1,(x5)
      vlb.v v2,(x6)
      vadd.vv v3, v1, v2
      vsb.v v3, (x7)
      j main
```