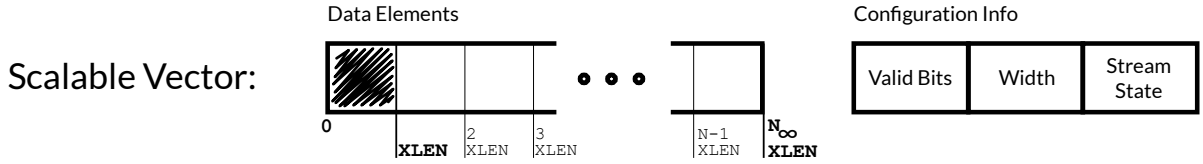
UVE Reference Card and Instruction Examples

Registers

– Vector/Stream Registers

UVE Vector Registers are scalable (undefined size at compile time), and range from `u0` to `u31`.



The minimum size of a implemented vector is XLEN[1]. Vector registers can be indefinitely large, as long as the size is multiple of XLEN. Each vector register is appended with additional information, such as an indicator of how many data elements are valid, the width of the data elements and the stream state.

—- **Valid Bits** Valid bits are useful to mark empty/inactive register elements during application execution.

—- **Width** The data elements can be of widths up to XLEN, starting at 8 bits (1B). Supported widths are:

| 8 bits - Byte | 16 bits - Half Word | 32 bits - Word | 64 bits - Double Word |

E.g., XLEN=32 ⇒ Supported widths:{8, 16, 32}

—- **Stream State** The stream state contains info on a current running and coupled stream. Information on when a dimension ends or the stream ends is contained here[2]. The Stream State does not provide information on whether the respective stream is or isn't running.

Notes: A streaming register is a common vector register that is associated with a stream through a special configuration instruction and until all the requested data is streamed. Otherwise, a register is a common vector register, and the Stream State does not contain valid information. While the stated widths may be available for unsigned and signed operations, depending on XLEN, 8-bits and 16-bits floating-point is unavailable in most architectures and ISAs.

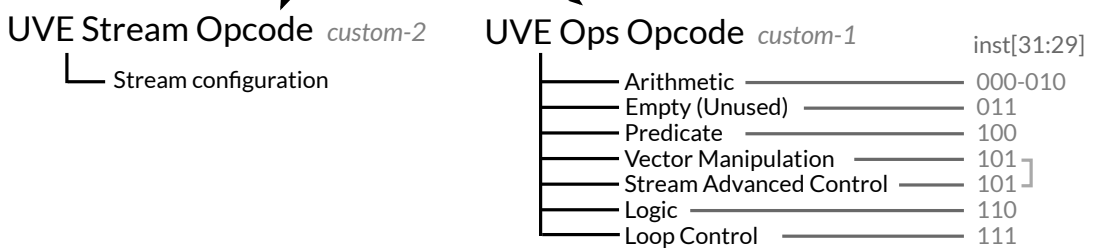
– Predicate Registers

Predicate registers control instruction execution through selective disabling/enabling of corresponding vector execution lanes (a predicate register is therefore similar to an array of booleans). UVE provides 16 predicate registers, of which `p0` to `p7` are Ready Predicates and `p8` to `p15` are standby predicates. Ready predicates (`p0` to `p15`) can be used by most data processing instructions, whereas Standby predicates can only be used as backup, and moved to Ready Predicates through a special predicate move instruction. There is no exchange instruction. The special register **p0** (true predicate) is always set to activate all lanes (hardwired to 1), and any write to it is disregarded.

Instructions

RISC-V base opcode map, inst[1:0] = 11

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



[1] XLEN is the maximum implemented register/data size of RISC-V. E.g., RV64 implements XLEN=64, RV32 implements XLEN=32. See also the RISC-V ISA unprivileged specification at https://riscv.org/technical/specifications/

[2] Check the Branches section for more detail

– Terminology

RTL description: RTL descriptions represent the associated operation, taking the generic form of:

$$Register = Operation(Register, Register, ...)?Predicate$$

Where elements take the form of:

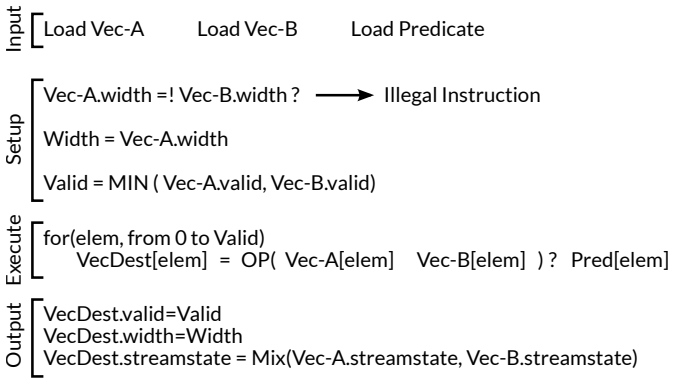
- Register:
  - vd#, vs# - Vector destination/source register in position #[3]
  - rd#, rs# - RISC-V General Purpose (scalar) destination/source register in position #
  - fd#, fs# - RISC-V Floating-Point destination/source register in position #
- Operation:
  - Generic Operation: Op(...). A pure representation, does not state the actual operation
  - Add: e.g., vd = vs1 + vs2
  - Shift Right Arithmetic: vd = vs1 » vs2
  - Generic Reduction Operation: RedOp(...). A pure representation of a reduction, all the elements will be reduced to one value
- ? Predicate: The presence of ?ps# or similar, indicates that ps# will predicate the operation

Instructions can also be FP (Floating-Point) or SG (signed), defaulting to USG (unsigned) when nothing specified. When marked (✓), there is a variant of the instruction for FP or SG. Note that some instructions have no unsigned form (e.g. ABS - absolute value), in this case footnote[4] is used.

– Arithmetic, Logic

Arithmetic and logic instructions have no information on the streaming process. These should be interpreted as common ALU instructions. Instructions do not specify the operation width (e.g. 8/16/32/64 bits), being encoded in the source vectors and passed to the destination vectors. If two vectors of distinct width are provided, the execution must raise an illegal instruction exception. Execution is conditioned based on predicates and valid index bits.

All instructions follow the general form:



—- Type:UVE-A (Arithmetic) RTL: vd = OP(vs1,vs2)?ps3 / vd = OP(vs1)?ps3

Instruction	Name	Encoding			RTL Description
		SubOpcode	FP	SG	
`add vd,vs1,vs2,ps3`	ADD	00000	✓	✓	`vd ← vs1 + vs2 ? ps3`
`sub vd,vs1,vs2,ps3`	SUB	00001	✓	✓	`vd ← vs1 - vs2 ? ps3`
`mul vd,vs1,vs2,ps3`	MUL	00010	✓	✓	`vd ← vs1 * vs2 ? ps3`
`div vd,vs1,vs2,ps3`	DIV	00011	✓	✓	`vd ← vs1 / vs2 ? ps3`
`abs vd,vs1, ps3`	Absolute Value	00100	✓	✓[4]	`vd ← ABS(vs1) ? ps3`
`mac vd,vs1,vs2,ps3`	MAC	00111	✓	✓	`vd ← ((vs1 * vs2) + vd) ? ps3`
`min vd,vs1,vs2,ps3`	minimum	01000	✓	✓	`vd ← MIN(vs1,vs2) ? ps3`
`max vd,vs1,vs2,ps3`	maximum	01000	✓	✓	`vd ← MAX(vs1,vs2) ? ps3`
`inc vd,vs1, ps3`	Increment by 1	01100	✓	✓	`vd ← vs1 + 1 ? ps3`
`dec vd,vs1, ps3`	Decrement by 1	01101	✓	✓	`vd ← vs1 - 1 ? ps3`

[3] Position # specifies the position of the register in the instruction encoding. Encodings are available in the instruction encoding types table.
[4] Unsigned version not available.

—- Type UVE-AR (Arithmetic Reduction) RTL: vd = RedOP(vs1)?ps3 / rd = RedOP(vs1)?ps3

Instruction		Name	Encoding SubOpcode	Encoding FP	Encoding SG	RTL Description
adde	vd,vs1,ps3	Sum Elements	00100	✓	✓	vd[0] ← SUM(vs1) ? ps3
sadde	rd,vs1,ps3	Sum Elements to scalar	00101	×	✓	rd ← SUM(vs1) ? ps3
fsadde	fd,vs1,ps3	Sum Elements to scalar	00101	✓	×4	fd ← SUM(vs1) ? ps3
adde.acc	vd,vs1,ps3	Sum Elements, accumulate	00100	✓	✓	vd[0] ← SUM(vs1) ? ps3
sadde.acc	rd,vs1,ps3	Sum Elements to scalar, accumulate	00101	×	✓	rd ← SUM(vs1) ? ps3
fsadde.acc	fd,vs1,ps3	Sum Elements to scalar, accumulate	00101	✓	×4	fd ← SUM(vs1) ? ps3
mins	vd,vs1,ps3	Minimum Elements	01010	✓	✓	vd[0] ← MIN(vs1) ? ps3
maxs	vd,vs1,ps3	Maximum Elements	01011	✓	✓	vd[0] ← MAX(vs1) ? ps3

Note1: adde.acc and sadde.acc are versions of the adde and sadde instructions that accumulate the sum with the destination registers value. Both accumulation instructions have the acc flag (inst[20]) set.

Note2: sadde and sadde.acc require a RISC-V FP register when in FP mode.

—- Type:UVE-L (Logic) RTL: vd = OP(vs1,vs2)?ps3 / vd = OP(vs1)?ps3 / vd = OP(vs1,rs2)?ps3

Instruction		Name	Encoding SubOpcode	RTL Description
nand	vd,vs1,vs2,ps3	NAND	1100000	vd ← vs1 NAND vs2 ? ps3
and	vd,vs1,vs2,ps3	AND	1100001	vd ← vs1 AND vs2 ? ps3
nor	vd,vs1,vs2,ps3	NOR	1100010	vd ← vs1 NOR vs2 ? ps3
or	vd,vs1,vs2,ps3	OR	1100011	vd ← vs1 OR vs2 ? ps3
not	vd,vs1, ps3	NOT	1100100	vd ← NOT vs1 ? ps3
xor	vd,vs1,vs2,ps3	XOR	1100101	vd ← vs1 XOR vs2 ? ps3
sll	vd,vs1,vs2,ps3	Shift-Left Logical	1101001	vd ← vs1 SLL vs2 ? ps3
srl	vd,vs1,vs2,ps3	Shift-Right Logical	1101011	vd ← vs1 SRL vs2 ? ps3
sra	vd,vs1,vs2,ps3	Shift-Right Arithmetic	1101101	vd ← vs1 SRA vs2 ? ps3
ssll	vd,vs1,rs2,ps3	Shift-Left Logical	1101000	vd ← vs1 SLL rs2 ? ps3
ssrl	vd,vs1,rs2,ps3	Shift-Right Logical	1101010	vd ← vs1 SRL rs2 ? ps3
ssra	vd,vs1,rs2,ps3	Shift-Right Arithmetic	1101100	vd ← vs1 SRA rs2 ? ps3

– Vector Manipulation

Vector manipulation instructions do register-vector, vector-memory and vector-vector operations.

—- Loads and Stores - Type:UVE-M Load and store operations move data between memory and vectors, with a variant (.s/set new address) that increments the address register with the transaction size.

Instruction		Name	Encoding SubOpcode	RTL Description
ld.(width)[5]	vd,rs1,rs2	Vector load	1010000	vd ← for(i in 0 -> rs2 - 1) load(rs1+i*width);
ld.(width).s	vd,rs1,rs2	load and set new address	1010001	same as ld plus rs1 ← rs1 + (rs2 - 1)*width
st[6]	vd,rs1,rs2	Vector Store	1010010	for(i in 0 -> rs2 - 1) vd → store(rs1+i*width);
st.s	vd,rs1,rs2	Store and set new address	1010011	same as st plus rs1 ← rs1 + (rs2 - 1)*width

—- Register-Register Movement - Type:UVE-V Register to Register instructions and width conversion instructions.

Instruction		Name	Encoding SubOpcode	Encoding Options	RTL Description
mv	vd,rs1,ps4	Vector Move	101010000	1|-|-	vd ← vs1 ? ps4;
mvt	vd,rs1,ps4	Vector Move Transpose	101010001	1|-|-	vd ← T(vs1) ? ps4
mvvs	rd,vs1	Move Vector to Scalar	101010010	-|-|-	rd ← vs1[0]
mvsv.(width)	vd,rs1	Move Scalar to Vector	101010011	-|width	vd[0] ← (width)rs1
dp.(width)	vd,rs1,ps4	Duplicate Scalar to Vector	101011000	1|width	vd[N:0] ← rs1 ? ps4
Width Conversion instructions[7]					
conv.(width)	vd,vs1	Convert as unsigned	101010100	1|width	vd ← US(width)vs1
conv.fp.(width)	vd,vs1	Convert as floating-point	101010101	1|width	vd ← FP(width)vs1
conv.sg.(width)	vd,vs1	Convert as signed	101010110	1|width	vd ← SG(width)vs1
No Stream instructions[8]					
mv.u	vd,rs1,ps4	Vector Move No Stream	101010000	0|-|-	vd ← vs1 ? ps4;
mvt.u	vd,rs1,ps4	Vector Move Transpose No Stream	101010001	0|-|-	vd ← T(vs1) ? ps4
dp.u.(width)	vd,rs1,ps4	Duplicate Scalar to Vector No Stream	101011000	0|width	vd[N:0] ← rs1 ? ps4
conv.u.(width)	vd,vs1	Convert as unsigned	101010100	0|width	vd ← US(width)vs1
conv.u.fp.(width)	vd,vs1	Convert as floating-point	101010101	0|width	vd ← FP(width)vs1
conv.u.sg.(width)	vd,vs1	Convert as signed	101010110	0|width	vd ← SG(width)vs1

Example on move transpose

Example on width convert

– Loop Control Branching - Types:UVE-SB

UVE provides special instructions for loop control through stream-based branching. These instructions do not iterate the stream contents and use the register contents to evaluate the branch condition. In specific, these branches will look at the `Stream State` flags of the register. Note that this branch requires that any instruction iterates the stream before the evaluation of any branch condition.

[5] `width` is one of b - byte, h - half, w - word, d - double.

[6] Store instructions have no `width` specifier. Encoding defaults to b000.

[7] Convert vector data from one width to another (e.g. 8->32; 64->16)

[8] `No Stream` variants will not update the stream state when executing, i.e. will not iterate the stream contents.

Instruction		Name	Encoding		RTL Description
			sopc	Options	
`sb.nc`	`vs1,imm`	Branch if stream not complete	111	`1|111|0`	`if(vs1 not complete) branch to imm+PC`
`sb.c`	`vs1,imm`	Branch if stream complete	111	`0|111|0`	`if(vs1 complete) branch to imm+PC`
`sb.ndc.(dim)`[9]	`vs1,imm`	Branch if stream not complete	111	`1|dim|0`	`if(vs1.dim not complete) branch to imm+PC`
`sb.dc.(dim)`	`vs1,imm`	Branch if stream complete	111	`0|dim|0`	`if(vs1.dim complete) branch to imm+PC`

Stream-State Example

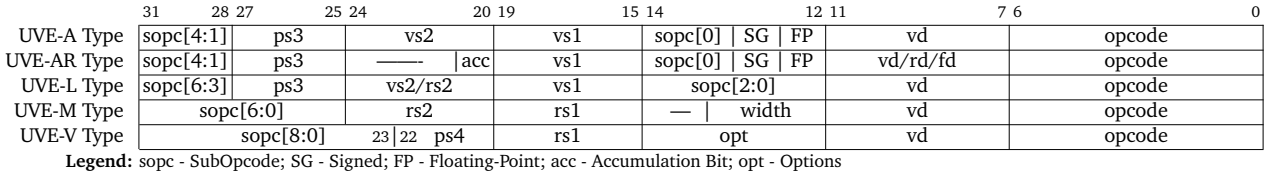
– Lane Control Predicates

To create and manipulate predicate registers, specific instructions are provided:

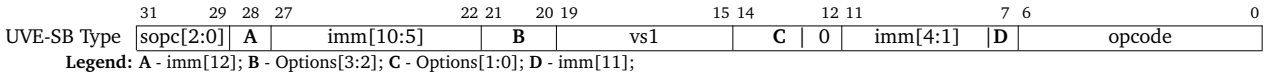
– Stream Control

Stream state can be controlled on-the-run to allow context swaps and early terminations.

– Instruction Formats



Legend: sopc - SubOpcode; SG - Signed; FP - Floating-Point; acc - Accumulation Bit; opt - Options



Legend: **A** - imm[12]; **B** - Options[3:2]; **C** - Options[1:0]; **D** - imm[11];

– Stream Configuration Instructions

- vector length configuration
- branches
- context saving
- Pattern examples (1D, 2D, ind, tri)
- Examples (stream, trisolv)
  - Automatic Consumption

[9] `dim` is a dimension to evaluate the condition, can be any from 1 (b000) to L (b111), where L is the last dimension (8). The sb.nc and sb.c branch instructions are special cases where `dim` is set to L.