Co-processor instruction set

The vector processor will be a co-processor to a RISC-V core.

Field-size	**7 bit**	**5 bit**	**5 bit**	**3 bit**	**5 bit**	**7 bit**	**Comment**
R-type	funct7	rs2	rs1	funct3	rd	opcode	Register-register ALU
I-type	imm		rs1	funct3	rd	opcode	Register-immediate ALU, load
S-type	imm	rs2	rs1	funct3	imm	opcode	Store
U-type	imm				rd	opcode	Loading immediates

- **opcode** - general class of operation e.g. OP-IMM for register-immediate ALU operations
- **rs1** - first source register
- **rs2** - second register
- **rd** - destination register
- **imm** - immediate value
- **funct3** - select type of operation e.g. sub, mul or add
- **funct7** - spill over details of operation

RISC-V ISAs have prefixes e.g. the standard integer instruction set has the ‘11’ prefix and thus has 30-bits of instruction space. Thus there are 3 other available 30 bit instruction spaces available although they can be occupied by the compressed-instruction extensions. As BRAM is relatively plentiful and can easily provide a 32-bit instruction read port, we are not going to implement compressed instructions and instead implement the vector processor as a full 30 bit extension with a prefix of ‘10’.

Version 1

VSOP

Perform arithmetic operations between vectors. As we experiment with different numerics, quite how these operations are performed will change e.g. saturated or stochastic rounded but the ISA will remain the same

Field-size	**7 bit**	**5 bit**	**5 bit**	**3 bit**	**5 bit**	**7 bit**
R-type	funct7	rs2	rs1	funct3	rd	0000010

- **rd** selects target vector register
- **rs1** and **rs2** select source vector registers
- **funct3** selects
  - VADD (000): `rd[i] = sat(rs1[i] + rs2[i])`
  - VSUB (010): `rd[i] = sat(rs1[i] - rs2[i])`
  - VMUL (100): `rd[i] = round(rs1[i] * rs2[i]) >> shift`
- **funct7**:

Field-size	**1 bit**	**2 bit**	**4 bit**
funct7	saturate	rounding	shift

  - **saturate** when performing an addition or subtraction (funct3 is 000 or 010), if this bit is set saturate the result between -32768 and 32767
  - **shift** when performing a multiplication (funct3 is VMUL), the result is 32-bit so we need to select which 16-bit component we actually want using a barrel shifter
  - **round** when performing a multiplication (func3 is 100), selects rounding mode:
    - 00 - standard round to zero `round(x) = x`
    - 01 - round to nearest `round(x) = x + (1 << (shift - 1))`
    - 10 - stochastic `round(x) = x + (RNG & ((1 << shift) - 1)`

VTST

Compare operands writes resultant 32-bit masks to scalar register file.

Field-size	**7 bit**	**5 bit**	**5 bit**	**3 bit**	**5 bit**	**7 bit**
R-type	0000000	rs2	rs1	funct3	rd	0001010

- **rd** selects target scalar register
- **rs1** and **rs2** select source vector registers
- **funct3** selects type of comparison
  - VTEQ (000): `rd = ((rs1[i] == rs2[i]) << i)`
  - VTNE (010): `rd = ((rs1[i] != rs2[i]) << i)`
  - VTLT (100): `rd = ((rs1[i] < rs2[i]) << i)`
  - VTGE (110): `rd = ((rs1[i] >= rs2[i]) << i)`

VSEL

Ternary operator. Used for implementing non-uniform control flow.

```
rd[i] = (rs1 & (1 << i)) ? rs2[i] : rd[i]
```

Field-size	**7 bit**	**5 bit**	**5 bit**	**3 bit**	**5 bit**	**7 bit**
R-type	0000000	rs2	rs1	000	rd	0001110

- **rd** selects target vector register
- **rs1** selects scalar register containing mask
- **rs2** selects vector register to move if mask is true

VLUI

This isn’t strictly necessary as you could just load immediates into scalar registers and move them into vector **but**, there may be more register pressure in the scalar register file (especially if we want to respect the RISC-V calling convention):

```
rd[i] = imm
```

Field-size	**7 bit**	**5 bit**	**5 bit**	**3 bit**	**5 bit**	**7 bit**
U-type	imm				rd	0000110

- **imm**:

Field-size	**4 bit**	**16 bit**
imm	zero	Immediate value

- **rd** selects vector register to load into

VFILL

Fills a vector register with the lower 16-bit of scalar register value. Loading directly from scalar memory into vector register would be possible but would involve integrating with XIF memory interface.

```
rd[i] = (rs1 & 0x80000000) >> 16 | (rs1 & 0x7FFF)
```

Field-size	**7 bit**	**5 bit**	**5 bit**	**3 bit**	**5 bit**	**7 bit**
I-type	000000000000		rs1	000	rd	0011010

- **rd** selects target vector register
- **rs1** selects scalar register containing value to fill. This value is treated as signed so sign bit needs to be combined with

VEXTRACT

Extract the value from one lane of vector register, sign extend to 32-bits and write to scalar register.

```
rd = (rs1[imm] << 16) >> 16
```

Field-size	**7 bit**	**5 bit**	**5 bit**	**3 bit**	**5 bit**	**7 bit**
I-type	0000000	imm	rs1	001	rd	0011010

- **rd** selects target scalar register

- **rs1** selects vector register value to extract. This value is treated as signed so sign bit needs to be combined with
- **imm** selects lane to extract

VSPC

Special functions used for random number generation etc. Random number generator is Xoroshiro32++. Because we assume vector lane values are signed, shifts down so you get random numbers in (0, 1) in s0.15 fixed point.

Field-size	**7 bit**	**5 bit**	**5 bit**	**3 bit**	**5 bit**	**7 bit**	**Comment**
I-type	000000000000		rs1	funct3	rd	0100010	Special

- **rd** selects vector target register
- **rs1** source vector register if appropriate
- **funct3** selects type of special function
  - VRNG (000): `rd = rng() >> 1`

VLOAD

Reads a vector register from vector memory. VLOADR0 and VLOADR1 are special versions used to load RNG seed into special seed registers.

Field-size	**7 bit**	**5 bit**	**5 bit**	**3 bit**	**5 bit**	**7 bit**	**Comment**
I-type	imm		rs1	000	rd	0010010	Vector load

- **imm** selects offset in bytes from base address for vector BRAM, this must be 64-byte vector-size aligned)
- **rd** selects vector register to load into
- **rs1** selects scalar register with base address in vector memory
- **funct3**
  - VLOADV (000): `rd[i] = rs1[(imm / 2) + i]`
  - VLOADR0 (001): rng[0] = `rs1[(imm / 2) + i]`
  - VLOADR1 (101): rng[1] = rs1[(imm / 2) + i]

VSTORE

Stores the contents of a vector register to vector memory:

```
rs1[(imm / 2) + i] = rs2[i]
```

Field-size	**7 bit**	**5 bit**	**5 bit**	**3 bit**	**5 bit**	**7 bit**	**Comment**
S-type	imm	rs2	rs1	funct3	imm	0010110	Store

- **imm** selects offset in bytes from base address (for vector BRAM, this must be 64-byte vector-size aligned)
- **rs1** selects scalar register with (vector-width-aligned) base address
- **rs2** selects vector register to store