RVV-lite v0.5: A Modest Proposal for Reducing the RISC-V Vector Extension

Guy Lemieux*, Caroline White, Fredy Alves, Farid Chalibi

University of British Columbia, Vancouver, Canada

Abstract

The monolithic RISC-V Vector (RVV) extension is prohibitively large for small embedded systems. We propose RVV-lite v0.5, a breakdown of RVV into layers and options. Each subset is forward compatible and can run on full RVV systems unmodified. We hope to spark discussions towards a standard modular RVV specification.

Introduction

With over 400 instructions and 6 data types, the RISC-V Vector extension (RVV) is considerably larger than the entire non-privileged RISC-V scalar ISA at roughly 120 instructions. The cost of implementing even the minimal form of RVV (Zve32x which strips 64b integers and all floating-point) can be prohibitive.

To provide smaller implementation options, this paper subdivides RVV into layers, optional extensions, and extra 'leftover' instructions. Any program written using an RVV-lite subset is fully compatible with the full RVV specification. We use instruction layering; the inclusion of a layer implies inclusion of all lower layers. In many cases, upper layers can re-use circuitry in lower layers. Layers also simplify software tooling and configuration management of both hw and sw.

Results

RVV-lite v0.5 partitions Zve* into 7 layers, 5 options and 3 extension layers. Implementation results for Saturn-V, our reference implementation, are shown in Table 1. With a 64-bit datapath, the implementation operates above 150MHz and ranges in area up to 7,200 LUTs, 11 DSPs, and 37 BRAMs in an AMD UltraScale+ FPGA. To improve RVV-lite, we welcome discussion and collaboration, hopefully leading to an official RISC-V standard.

RVV-lite restricts the SEW/LMUL ratio to 8 to simplify register file design. Programmers benefit from the same VLMAX across all element sizes.

A minimal RVV-lite built using only A.1(a) has no computational instructions, but it could be a basis for adding only custom instructions. Adding ALU operations in A.1(b) requires a second read port and 16 more BRAM; the other 4 BRAM and 2 URAM are buffers in the (unoptimized) memory interface. The remaining instructions within A.1 don't require much area, except slide-by-1 instructions in A.1(g) which must support 8/16/32/64 bit slides.

Widening add/sub instructions (A.2) impose new width constraints on the register file ports. Reductions (A.3) add logic that spans lanes. In (A.4), the shift instructions also use the multiplier logic; widening/narrowing A.4(d) needs extra multiplexing logic. The slide-by-N (A.5) reuses slide-by-1 logic, but scales super-linearly beyond 64b. The 64b multiply/shift (A.5) re-uses the 32b logic, but it still needs more. Likewise, fixed-point instructions (A.7) and masks (B.1) are costly. Saturn-V does not yet implement other layers, but Vicuna `https://github.com/vproc/vicuna` shown in Table 2 is a nearly complete Zve32x.

Table 1: *Saturn-V implementation on AMD FPGA (64b width, VLEN=16,384, B=BRAM, U=URAM, D=DSP)*

Configuration	LUTs	B	U	D	Fmax
A.1(a) Load/Store/Cfg	1,220	20	2	0	177.2
A.1(b) Basic ALU Ops	2,584	36	2	0	181.7
A.1(c) Min/Max.	2,630	36	2	0	183.3
A.1(d) Basic Mask	3,155	37	2	0	178.6
A.1(e) Vector Move	3,237	37	2	0	175.4
A.1(f) Whole Reg	3,251	37	2	0	185.4
A.1(g) Slide-by-1	3,835	37	2	0	160.1
A.2 Widen Add/Sub	4,070	37	2	0	177.1
A.3 Reduction	4,523	37	2	0	162.1
A.4(a) 32b Mul/ShiftL	4,833	37	2	8	172.1
A.4(b) 16b MulH/ShiftR	4,857	37	2	8	183.8
A.4(c) 32b MulH/ShiftR	5,165	37	2	8	166.5
A.4(d) Wide Mul, Nar. Shift	5,647	37	2	8	167.3
A.5 Slide-by-N	5,932	37	2	8	187.3
A.6 64b Mul/Shift	6,273	37	2	11	161.0
A.7 FXP	6,523	37	2	11	169.4
B.1 Mask (including A.7)	7,164	37	2	11	172.0

Table 2: *Comparison (Zve32x only, VLEN=128)*

Configuration	Width	LUTs	B	D	Fmax
Saturn-V A.7+B.1	64	5,343	6	8	220.4
Vicuna (2022-07-22)	64	8,581	0	10	72.0

*Corresponding author: `lemieux@ece.ubc.ca`

RVV-lite ISA Summary

RVV-lite v0.5 subdivides the Zve* version of RVV into 7 integer layers, 5 optional extensions, and 3 extension layers.

Notation key:

- VXIF denotes up to 4 operand modes: V=vv, X=vx, I=vi, F=vf
- YY denotes up to 3 operand types: vs1/rs1/imm
- ZZ denotes up to 3 operand types: vs1/rs1/uimm
- rs1 is from X or F register set, i.e. X[rs1] or F[rs1]
- K denotes register group size: 1, 2, 4, 8
- EW/IW denotes element/index width: 8, 16, 32, 64
- KEW denotes reg group/EW sizes: 1re8, 2re16, 4re32, 8re64
- L denotes index of last element, i.e. L = VL-1
- S/U denotes signed/unsigned
- LOP (logic-op): and, or, xor
- FOP (float-op): add, sub, mul, div, min, max
- MOP (mask-op) and, nand, andnot, xor, or, nor, ornot, xnor
- ROP (reduction-op): sum, and, or, xor, maxu, max, minu, min
- MCMP (mask-compare): seq, sne, sle, sleu
- ADD/SUB/MUL/MIN/MAX: add/sub/mul/min/max, addu/subu/mulu/minu/maxu
- MULH: mulh, mulhu, mulhsu
- ADDSUBU/MINMAXU/EQ/GTE/LTE: addu/minu/eq/gt/lt, subu/maxu/ne/ge/le
- SLT/SGT: slt/sgt, sltu/sgtu

RVV-lite Integer Layers

A.1 Core (8/16/32/64b)

```
a) vsetvli      rd,rs1,vtypei       # rd=VL=f(AVL), AVL=rs1, new vtype
a) vsetivli     rd,uimm,vtypei      # rd=VL=f(AVL), AVL=uimm, new vtype
a) vsetvl       rd, rs1, rs2        # rd=VL=f(AVL), AVL=rs1, vtype=rs2
a) vleEW.v      vd,(rs1),vm         # vector load, EEW=EW
a) vseEW.v      vs3,(rs1),vm        # vector store, EEW=EW
b) vid.v        vd,vm               # vd[i] = i
b) vLOP.VXI     vd,vs2,YY,vm        # vd[i] = vs2[i] LOP YY
b) vADDSUB.VXI  vd,vs2,YY,vm        # vd[i] = vs2[i] ADDSUB YY
b) vrsub.XI     vd,vs2,YY,vm        # vd[i] = YY - vs2[i]
c) vMINMAX{U}.VX vd,vs2,YY,vm       # vd[i] = MINMAX{U}(vs2[i], YY)
d) vmMCMP.VXI   vd,vs2,YY,vm        # vd.m[i] = (vs2[i] MCMP YY)
d) vmSLT.VX     vd,vs2,YY,vm        # vd.m[i] = (vs2[i] < YY)
d) vmSGT.XI     vd,vs2,YY,vm        # vd.m[i] = (vs2[i] > YY)
d) vmMOP.mm     vd,vs2,vs1          # vd.m[i] = MOP(vs2.m[i],vs1.m[i])
e) vmv.v.VXI    vd,YY               # vd[i] = YY, XI modes: integer splat
e) vmv.x.s      rd,vs2              # x[rd] = vs2[0], scalar copy (vs1=0)
e) vmv.s.x      vd,rs1              # vd[0] = x[rs1], scalar copy (vs2=0)
f) vmvKr.v      vd,vs2              # whole-vec. reg. group copy EMUL=K
f) vlKEW.v      vd,(a0)             # whole reg EMUL=K, VLEN/EW elem, ign. VL
f) vsKr.v       vd,(a1)             # whole reg EMUL=K, VLEN bits, ignores VL
g) vslide1up.vx     vd,vs2,rs1,vm   # vd[i+1]=vs2[i], vd[0]=X[rs1]
g) vslide1down.vx   vd,vs2,rs1,vm   # vd[i]=vs2[i+1], vd[L]=X[rs1]
```

A.2 Widen Add/Sub (8/16/32b)

```
vwADDSUB{U}.VX   vd,vs2,YY,vm       # vd[i] = vs2[i] ADDSUB{U} YY
```

A.3 Reduction (8/16/32/64b)

```
vredROP.vs vd,vs2,vs1,vm # vd[0]=ROP(vs1[0],vs2[*])
```

A.4 Mul/Shift (mostly 8/16/32b)

```
a) vmul.VX      vd,vs2,YY,vm    # vd[i] = LSB(vs2[i] * YY)  (8/16/32b)
a) vsll.VXI     vd,vs2,ZZ,vm    # vd[i] = vs2[i] << YY      (8/16/32b)
b) vsr{l/a}.VXI vd,vs2,ZZ,vm    # vd[i] = vs2[i] {>}>> YY   (8/16b)
b) vMULH.VX     vd,vs2,YY,vm    # vd[i] = MSB(vs2[i] * YY)  (8/16b)
c) vsr{l/a}.VXI vd,vs2,ZZ,vm    # vd[i] = vs2[i] {>}>> YY   (32b)
c) vMULH.VX     vd,vs2,YY,vm    # vd[i] = MSB(vs2[i] * YY)  (32b)
d) vwMUL.VX     vd,vs2,YY,vm    # vd[i] = vs2[i] * YY       (8/16/32b)
d) vwmulsu.VX   vd,vs2,YY,vm    # vd[i] = vs2[i] S*U YY     (8/16/32b)
d) vnsrl.wX     vd,vs2,x0,vm    # vd[i] = vs2[i]            (8/16/32b)
```

A.5 Slide-by-N (8/16/32/64b)

```
vslideup.XI   vd,vs2,ZZ,vm  # vd[i+ZZ] = vs2[i]
vslidedown.XI vd,vs2,ZZ,vm  # vd[i] = vs2[i+ZZ]
```

A.6 Multiply/Shift (64b)

```
vmul.VX  vd,vs2,YY,vm       # vd[i] = LSB(vs2[i] * YY)
vsll.VXI vd,vs2,ZZ,vm       # vd[i] = vs2[i] << YY
vsr{l/a}.VXI vd,vs2,ZZ,vm   # vd[i] = vs2[i] {>}>> YY
```

A.7 Fixed-point (8/16/32/64b)

```
vaADDSUB{U}.VX  vd, vs2, YY, vm # round_US(vs2[i] ADDSUB{U} YY, 1)
vsmul.VX        vd, vs2, YY, vm # vd[i]=clip(round_S(vs2[i]*YY,SEW-1)) (no 64b)
vssr{l/a}.VXI   vd, vs2, ZZ, vm # vd[i]=round_{U/S}(vs2[i],ZZ)
```

RVV-lite Options (not layered)

B.1 Mask (8/16/32/64b)

```
vlm.v       vd, (rs1)           # ld mask of ceil(vl/8) bytes
vsm.v       vs3,(rs1)           # st mask of ceil(vl/8) bytes
vadc.VXIm   vd,vs2,YY,v0        # vd[i]=vs2[i]+vs1[i]+v0.m[i]
vmadc.VXm   vd,vs2,YY,v0        # vd.m[i]=cout(vs2[i]+vs1[i]+v0.m[i])
vmadc.VXI   vd,vs2,YY           # vd.m[i]=cout(vs2[i]+vs1[i])
vsbc.VXm    vd,vs2,YY,v0        # vd[i]=vs2[i]-vs1[i]-v0.m[i]
vmsbc.VXm   vd,vs2,YY,v0        # vd.m[i]=brrw(vs2[i]-vs1[i]-v0.m[i])
vmsbc.VX    vd,vs2,YY           # vd.m[i]=brrw(vs2[i]-vs1[i])
vcpop.m     rd,vs2,vm           # x[rd] = sum(vs2.m[i]), count bits
vfirst.m    rd,vs2,vm           # x[rd] = idx_of_first_one(vs2.m)
vmerge.VXIm vd,vs2,YY,v0        # vd[i] = v0.m[i] ? YY : vs2[i]
```

B.2 Strided Memory (8/16/32/64b)

```
vlseEW.v  vd,  (rs1), rs2, vm   # strided ld, EEW=EW
vsseEW.v  vs3, (rs1), rs2, vm   # strided st, EEW=EW
```

B.3 Indexed Memory (8/16/32/64b)

```
vl{u/o}xeiIW.v  vd,  (rs1), vs2, vm # {un}ordered, indexed load
vs{u/o}xeiIW.v  vs3, (rs1), vs2, vm # {un}ordered, indexed store
```

B.4 Float (binary32)

```
a) vmfEQ.VF      vd,vs2,YY,vm       # vd[i] = (vs2[i] EQ YY)
a) vfsgnj.VF     vd,vs2,YY,vm       # vd[i]={sgn(YY),abs(vs2[i])}
a) vfsgnjn.VF    vd,vs2,YY,vm       # vd[i]={~sgn(YY),abs(vs2[i])}
a) vfsgnjx.VF    vd,vs2,YY,vm       # vd[i]={sgn(YY*vs2[i]),abs(vs2[i])}
a) vfclass.v     vd,vs2,vm          # vd[i] = classify( vs2[i] )
a) vfcvt.{xu/x}.f.v     vd,vs2,vm  # float to {u}int
a) vfcvt.rtz.{xu/x}.f.v vd,vs2,vm  # float to {u}int (round to zero trunc.)
a) vfcvt.f.{xu/x}.v     vd,vs2,vm  # {u}int to float
a) vfmerge.vfm   vd,vs2,rs1,v0      # vd[i] = v0.m[i]?f[rs1]:vs2[i]
a) vfmv.v.f      vd,rs1             # vd[i] = f[rs1] (float splat)
a) vfmv.f.s      rd,vs2             # f[rd] = vs2[0] (rs1=0)
a) vfmv.s.f      vd,rs1             # vd[0] = f[rs1] (vs2=0)
a) vfslide1up.F   vd,vs2,YY,vm      # vd[i+1]=vs2[i],vd[0]=F[YY]
a) vfslide1down.F vd,vs2,YY,vm      # vd[i]=vs2[i+1],vd[L]=F[YY]
b) vfADDSUB.VF   vd,vs2,YY,vm       # vd[i] = ADDSUB( vs2[i], YY )
b) vfrsub.vf     vd,vs2,rs1,vm      # vd[i] = f[rs1] - vs2[i]
b) vfMINMAX.VF   vd,vs2,YY,vm       # vd[i] = MINMAX( vs2[i], YY )
b) vmfGTE.vf     vd,vs2,rs1,vm      # vd[i] = (vs2[i] GTE f[rs1])
b) vmfLTE.VF     vd,vs2,YY,vm       # vd[i] = (vs2[i] LTE YY)
b) vfredMINMAX.vs vd,vs2,vs1,vm     # vd[0] = fMINMAX(vs1[0],vs2[*])
b) vfred{u/o}sum.vs  vd,vs2,vs1,vm  # vd[0]={un}ord_fsum(vs1[0],vs2[*])
c) vfmul.VF      vd,vs2,YY,vm       # vd[i] = fmul( vs2[i], YY )
d) vfdiv.VF      vd,vs2,YY,vm       # vd[i] = fdiv( vs2[i], YY )
d) vfrdiv.vf     vd,vs2,rs1,vm      # vd[i] = f[rs1] / vs2[i]
e) vfsqrt.v      vd,vs2,vm          # vd[i] = sqrt( v2[i] )
```

B.5 Double (binary64, requires B.4)

```
vfwADDSUB.VF vd, vs2, YY, vm    # vs2[i] ADDSUB YY
vfwmul.VF    vd, vs2, YY, vm    # vs2[i] * YY
vfwADDSUB.wVF  vd, vs2, YY, vm  # vs2[i] ADDSUB YY
vfwcvt.{xu/x}.f.v  vd, vs2, vm      # SEW float to 2SEW {u}int
vfwcvt.rtz.{xu/x}.f.v vd, vs2, vm   # SEW float to 2SEW {u}int (round to zero trunc.)
vfwcvt.f.{xu/x/f}.v  vd, vs2, vm    # SEW {{u}int/float} to 2SEW float
vfncvt.{xu/x}.f.w vd, vs2, vm       # 2SEW float to SEW {u}int
vfncvt.rtz.{xu/x}.f.w vd, vs2, vm   # 2SEW float to SEW {u}int (round to zero trunc.)
vfncvt.f.{xu/x/f}.w vd, vs2, vm     # 2SEW {{u}int/float} to SEW float
vfncvt.rod.f.f.w vd, vs2, vm        # 2SEW float to SEW float (round to odd)
vfwred{u/o}sum.vs vd, vs2, vs1, vm  # vd[0] = {un}ord_fsum(vs1[0],vs2[*])
```

Extension Layers

C.1 Integer Extension (requires A.7)

This section, which has been omitted for brevity, contains all remaining integer instructions in the full Zve* ISA. There are approximately 100 instructions in this group.

C.2 Float Extension (requires B.4)

```
vf{n}macc.VF  vd, YY, vs2, vm   # vd[i] = {-/+}(YY * vs2[i]) {-/+} vd[i]
vf{n}msac.VF  vd, YY, vs2, vm   # vd[i] = {-/+}(YY * vs2[i]) {+/-} vd[i]
vf{n}madd.VF  vd, YY, vs2, vm   # vd[i] = {-/+}(YY * vd[i]) {-/+} vs2[i]
vf{n}msub.VF  vd, YY, vs2, vm   # vd[i] = {-/+}(YY * vd[i]) {+/-} vs2[i]
vfrsqrt7.v    vd, vs2, vm       # vd[i] = 1.0 / sqrt( v2[i] )
vfrec7.v      vd, vs2, vm       # vd[i] = 1.0 / v2[i]
```

C.3 Double Extension (requires B.5)

```
vfw{n}macc.VF  vd, YY, vs2, vm  # vd[i] = {-/+} (YY * vs2[i]) {-/+} vd[i]
vfw{n}msac.VF vd, YY, vs2, vm   # vd[i] = {-/+}(YY * vs2[i]) {+/-} vd[i]
```