RFC ls008 SVP64 Management instructions

URLs:

- https://libre-soc.org/openpower/sv/
- https://libre-soc.org/openpower/sv/rfc/ls008/
- https://bugs.libre-soc.org/show_bug.cgi?id=1040
- https://git.openpower.foundation/isa/PowerISA/issues/123

Severity: Major

Status: New

Date: 24 Mar 2023

Target: v3.2B

Source: v3.0B

Books and Section affected:

```
Book I, new Scalar Chapter.  (Or, new Book on "Zero-Overhead Loop Subsystem")
Appendix E Power ISA sorted by opcode
Appendix F Power ISA sorted by version
Appendix G Power ISA sorted by Compliancy Subset
Appendix H Power ISA sorted by mnemonic
```

Summary

```
setvl    - Cray-style "Set Vector Length" instruction
svstep   - Vertical-First Mode explicit Step and Status
```

Submitter: Luke Leighton (Libre-SOC)

Requester: Libre-SOC

Impact on processor:

```
Addition of two new "Zero-Overhead-Loop-Control" DSP-style Vector-style
Management Instructions which can be implemented extremely efficiently
and effectively by inserting an additional phase between Decode and Issue.
More complex designs are NOT adversely impacted and in fact greatly benefit
```

Impact on software:

```
Requires support for new instructions in assembler, debuggers,
and related tools.
```

Keywords:

```
Cray Supercomputing, Vectorisation, Zero-Overhead-Loop-Control (ZOLC),
Scalable Vectors, Multi-Issue Out-of-Order, Sequential Programming Model,
Digital Signal Processing (DSP)
```

Motivation

Power ISA is synonymous with Supercomputing and the early Supercomputers (ETA-10, ILLIAC-IV, CDC200, Cray) had Vectorisation. It is therefore anomalous that Power ISA does not have Scalable Vectors. This presents the opportunity to modernise Power ISA keeping it at the top of Supercomputing.

Notes and Observations:

1. SVP64 is very much designed for ultra-light-weight Embedded use-cases all the way up to moving the bar of Supercomputing orders of magnitude above its present perception, whilst retaining at all times Sequential Programming Execution.
2. This proposal is the **base** for further Extensions. These include extending SVP64 onto the Scalar VSX instructions (with a **LONG TERM** view in 10+ years to deprecating the PackedSIMD aspects of VSX), to be discussed at a later time, the potential for extending VSX registers to 128 or beyond, and Arithmetic operations to a runtime-selectable choice of 128-bit, 256-bit, 512-bit or 1024-bit.
3. Massive reductions in instruction count of between 2x and 20x have been demonstrated with SVP64, which is far beyond anything ever achieved by any *general-purpose* ISA Extension added to any ISA in the history of Computing.

Changes

Add the following entries to:

- Section 1.3.2 Notation
- the Appendices of Book I
- Instructions of Book I as a new Section
- SVL-Form of Book I Section 1.6.1.6 and 1.6.2

Notation, Section 1.3.2

When destination register operands (`RT, RS`) are prefixed by a single underscore (`_RT, _RS`) the variable also contains the contents of the instruction field. This avoids confusion in pseudocode when a destination register is assigned (`RT <- x`) but earlier it was the operand bits that were checked (`if RT = 0`)

---

svstep: Vertical-First Stepping and status reporting

SVL-Form

- svstep RT,SVi,vf (Rc=0)
- svstep. RT,SVi,vf (Rc=1)

0-5	6-10	11.15	16..22	23-25	26-30	31	Form
PO	RT	/	SVi	/ / vf	XO	Rc	SVL-Form

Pseudo-code:

```
if SVi[3:4] = 0b11 then
    # store pack and unpack in SVSTATE
    SVSTATE[53] <- SVi[5]
    SVSTATE[54] <- SVi[6]
    RT <- [0]*62 || SVSTATE[53:54]
else
    # Vertical-First explicit stepping.
    step <- SVSTATE_NEXT(SVi, vf)
    RT <- [0]*57 || step
```

Special Registers Altered:

```
CR0                     (if Rc=1)
```

Description

svstep may be used to enquire about the REMAP Schedule and it may be used to alter Vectorisation State. When `vf=1` then stepping occurs. When `vf=0` the enquiry is performed without altering internal state. If `SVi=0, Rc=0, vf=0` the instruction is a `nop`.

The following Modes exist:

- `SVi=0`: appropriately step srcstep, dststep, subsrcstep and subdststep to the next element, taking pack and unpack into consideration.
- When `SVi` is 1-4 the REMAP Schedule for a given SVSHAPE may be returned in `RT`. SVi=1 selects SVSHAPE0 current state, through to SVi=4 selects SVSHAPE3.
- When `SVi` is 5, `SVSTATE.srcstep` is returned.
- When `SVi` is 6, `SVSTATE.dststep` is returned.
- When `SVi` is 7, `SVSTATE.ssubstep` is returned.
- When `SVi` is 8, `SVSTATE.dsubstep` is returned.
- When `SVi` is 0b1100 pack/unpack in SVSTATE is cleared
- When `SVi` is 0b1101 pack in SVSTATE is set, unpack is cleared
- When `SVi` is 0b1110 unpack in SVSTATE is set, pack is cleared
- When `SVi` is 0b1111 pack/unpack in SVSTATE are set

As this is a Single-Predicated (1P) instruction, predication may be applied to skip (or zero) elements.

- Vertical-First Mode will return the requested index (and move to the next state if `vf=1`)
- Horizontal-First Mode can be used to return all indices, i.e. walks through all possible states.

Vectorisation of svstep itself

As a 32-bit instruction, `svstep` may be itself be Vector-Prefixed, as `sv.svstep`. This will work perfectly well in Horizontal-First as it will in Vertical-First Mode.

Example: to obtain the full set of possible computed element indices use `sv.svstep *RT,SVi,1` which will store all computed element indices, starting from RT. If Rc=1 then a co-result Vector of CR Fields will also be returned, comprising the "loop end-points" of each of the inner loops when either Matrix Mode or DCT/FFT is set. In other words, for example, when the `xdim` inner loop reaches the end and on the next iteration it will begin again at zero, the CR Field `EQ` will be set. With a maximum of three loops within both Matrix and DCT/FFT Modes, the CR Field's EQ bit will be set at the end of the first inner loop, the LE bit for the second, the GT bit for the outermost loop and the SO bit set on the very last element, when all loops reach their maximum extent.

Programmer's note: VL in some situations, particularly larger Matrices (5x7x3 will set MAXVL=105), will cause `sv.svstep` to return a considerable number of values. Under such circumstances `sv.svstep/ew=8` is recommended.

Programmer's note: having conveniently obtained a pre-computed Schedule with `sv.svstep`, it may then be used as the input to Indexed REMAP Mode to achieve the exact same Schedule. It is evident however that before use some of the Indices may be arbitrarily altered as desired. `sv.svstep` helps the programmer avoid having to manually recreate Indices for certain types of common Loop patterns. In its simplest form, without REMAP (SVi=5 or SVi=6), is equivalent to the `iota` instruction found in other Vector ISAs

Vertical First Mode

Vertical First is effectively like an implicit single bit predicate applied to every SVP64 instruction. **ONLY** one element in each SVP64 Vector instruction is executed; srcstep and dststep do **not** increment automatically on completion of one instruction, and the Program Counter progresses **immediately** to the next instruction just as it would for any standard scalar v3.0B instruction.

A mode of srcstep (SVi=0) is called which can move srcstep and dststep on to the next element, still respecting predicate masks.

In other words, where normal SVP64 Vectorisation acts "horizontally" by looping first through 0 to VL-1 and only then moving the PC to the next instruction, Vertical-First moves the PC onwards (vertically) through multiple instructions **with the same srcstep and dststep**, then an explict instruction used to advance srcstep/dststep. An outer loop is expected to be used (branch instruction) which completes a series of Vector operations.

Testing any end condition of any loop of any REMAP state allows branches to be used to create loops.

Programmer's note: when Predicate Non-Zeroing is used this indicates to the underlying hardware that any masked-out element must be skipped. This includes in Vertical-First Mode*, and programmers should be keenly aware that srcstep or dststep or both* may* jump by more than one as a result, because the actual request under these circumstances was to execute on the first available next *non-masked-out* element. It should be evident that it is the `sv.svstep` instruction that must be Predicated in order for the **entire** loop to use the Predicate correctly, and it is strongly recommended for all instructions within the same Vertical-First Loop to utilise the exact same Predicate Mask(s).*

Programmers should be aware that VL, srcstep and dststep and the SUBVL substeps are global in nature. Nested looping with different schedules is perfectly possible, as is calling of functions, however SVSTATE (and any associated SVSHAPEs if REMAP is being used) should obviously be stored on the stack in order to achieve this benefit not normally found in Vector ISAs.

Appendix

src_iterate

Note that `srcstep` and `ssubstep` are not the absolute final Element (and Sub-Element) offsets. `srcstep` still has to go through individual `REMAP` translation before becoming a per-operand (RA, RB, RC, RT, RS) Element-level Source offset.

Note also critically that `PACK` mode simply inverts the outer/order loops making SUBVL the outer loop and VL the inner.

```
    # source-stepping iterator
    subvl = SVSTATE.subvl
    vl = SVSTATE.vl
    pack = SVSTATE.pack
    unpack = SVSTATE.unpack
    ssubstep = SVSTATE.ssubstep
    end_ssub = ssubstep == subvl
    end_src = SVSTATE.srcstep == vl-1
    # first source step.
    srcstep = SVSTATE.srcstep
    # used below:
    #       sz      - from RM.MODE, source-zeroing
    #       srcmask - from RM.MODE, the source predicate
    if pack:
        # pack advances subvl in *outer* loop
        while True:
            assert srcstep <= vl-1
            end_src = srcstep == vl-1
            if end_src:
                if end_ssub:
                    loopend = True
                else:
                    SVSTATE.ssubstep += 1
                srcstep = 0  # reset
                break
            else:
                srcstep += 1  # advance srcstep
                if not sz:
                    break
                if ((1 << srcstep) & srcmask) != 0:
                    break
    else:
        # advance subvl in *inner* loop
        if end_ssub:
            while True:
                assert srcstep <= vl-1
                end_src = srcstep == vl-1
                if end_src:  # end-point
                    loopend = True
                    srcstep = 0
                    break
                else:
                    srcstep += 1
                if not sz:
                    break
                if ((1 << srcstep) & srcmask) != 0:
                    break
                else:
                    log("      sskip", bin(srcmask), bin(1 << srcstep))
            SVSTATE.ssubstep = 0b00  # reset
        else:
            # advance ssubstep
            SVSTATE.ssubstep += 1

    SVSTATE.srcstep = srcstep
```

---

dest_iterate

Note that `dststep` and `dsubstep` are not the absolute final Element (and Sub-Element) offsets. `dststep` still has to go through individual `REMAP` translation before becoming a per-operand (RT, RS/EA) destination Element-level offset, and `dsubstep` may also go through `(f)mv.swizzle` reordering.

Note also critically that `UNPACK` mode simply inverts the outer/order loops making SUBVL the outer loop and VL the inner.

```
# dest step iterator
vl = SVSTATE.vl
subvl = SVSTATE.subvl
unpack = SVSTATE.unpack
dsubstep = SVSTATE.dsubstep
end_dsub = dsubstep == subvl
dststep = SVSTATE.dststep
end_dst = dststep == vl-1
# used below:
#       dz      - from RM.MODE, destination-zeroing
#       dstmask - from RM.MODE, the destination predicate
if unpack:
    # unpack advances subvl in *outer* loop
    while True:
        assert dststep <= vl-1
        end_dst = dststep == vl-1
        if end_dst:
            if end_dsub:
                loopend = True
            else:
                SVSTATE.dsubstep += 1
            dststep = 0  # reset
            break
        else:
            dststep += 1  # advance dststep
            if not dz:
                break
            if ((1 << dststep) & dstmask) != 0:
                break
else:
    # advance subvl in *inner* loop
    if end_dsub:
        while True:
            assert dststep <= vl-1
            end_dst = dststep == vl-1
            if end_dst:  # end-point
                loopend = True
                dststep = 0
                break
            else:
                dststep += 1
            if not dz:
                break
            if ((1 << dststep) & dstmask) != 0:
                break
        SVSTATE.dsubstep = 0b00  # reset
    else:
        # advance ssubstep
        SVSTATE.dsubstep += 1

SVSTATE.dststep = dststep
```

SVSTATE_NEXT

```
    if SVi = 1 then return REMAP SVSHAPE0 current offset
    if SVi = 2 then return REMAP SVSHAPE1 current offset
    if SVi = 3 then return REMAP SVSHAPE2 current offset
    if SVi = 4 then return REMAP SVSHAPE3 current offset
    if SVi = 5 then return SVSTATE.srcstep  # VL source step
    if SVi = 6 then return SVSTATE.dststep  # VL dest step
    if SVi = 7 then return SVSTATE.ssubstep # SUBVL source step
    if SVi = 8 then return SVSTATE.dsubstep # SUBVL dest step

    # SVi=0, explicit iteration requezted
    src_iterate();
    dst_iterate();
    return 0
```

at_loopend

Both Vertical-First and Horizontal-First may use this algorithm to determine if the "end-of-looping" (end of Sub-Program-Counter) has been reached. Horizontal-First Mode will immediately move to the next instruction, where `svstep.` will set `CR0.EQ` to 1.

```
    # tells if this is the last possible element.
    subvl = SVSTATE.subvl
    vl = SVSTATE.vl
    end_ssub = SVSTATE.ssubstep == subvl
    end_dsub = SVSTATE.dsubstep == subvl
    if SVSTATE.srcstep == vl-1 and end_ssub:
        return True
    if SVSTATE.dststep == vl-1 and end_dsub:
        return True
    return False
```

[[!tag standards]]

---

setvl: Set Vector Length

See links:

- http://lists.libre-soc.org/pipermail/libre-soc-dev/2020-November/001366.html
- https://bugs.libre-soc.org/show_bug.cgi?id=535
- https://bugs.libre-soc.org/show_bug.cgi?id=587
- https://bugs.libre-soc.org/show_bug.cgi?id=568 TODO
- https://bugs.libre-soc.org/show_bug.cgi?id=927 bug - RT>=32
- https://bugs.libre-soc.org/show_bug.cgi?id=862 VF Predication
- https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc#vsetvlivsetvl-instructions
- {svstep instruction}
- pseudocode {Simple-V pseudocode}

Add the following section to the Simple-V Chapter

setvl

SVL-Form

0-5	6-10	11-15	16-22	23 24 25	26-30	31	FORM
PO	RT	RA	SVi	ms vs vf	XO	Rc	SVL-Form

- setvl RT,RA,SVi,vf,vs,ms (Rc=0)
- setvl. RT,RA,SVi,vf,vs,ms (Rc=1)

Pseudo-code:

```
overflow <- 0b0    # sets CR.SO if set and if Rc=1
VLimm <- SVi + 1
# set or get MVL
if ms = 1 then MVL <- VLimm[0:6]
else           MVL <- SVSTATE[0:6]
# set or get VL
if vs = 0                then VL <- SVSTATE[7:13]
else if _RA != 0         then
    if (RA) >u 0b1111111 then
        VL <- 0b1111111
        overflow <- 0b1
    else                      VL <- (RA)[57:63]
else if _RT = 0          then VL <- VLimm[0:6]
else if CTR >u 0b1111111 then
    VL <- 0b1111111
    overflow <- 0b1
else                          VL <- CTR[57:63]
# limit VL to within MVL
if VL >u MVL then
    overflow <- 0b1
    VL <- MVL
SVSTATE[0:6] <- MVL
SVSTATE[7:13] <- VL
if _RT != 0 then
   GPR(_RT) <- [0]*57 || VL
# MAXVL is a static "state-reset" opportunity so VF is only set then.
if ms = 1 then
     SVSTATE[63] <- vf   # set Vertical-First mode
     SVSTATE[62] <- 0b0  # clear persist bit
```

Special Registers Altered:

```
CR0                     (if Rc=1)
SVSTATE
```

- `SVi` - bits 16-22 - an immediate operand for setting MVL and/or VL
- `ms` - bit 23 - allows for setting of MVL
- `vs` - bit 24 - allows for setting of VL
- `vf` - bit 25 - sets “Vertical First Mode”.

Note that in immediate setting mode VL and MVL start from **one** but that this is compensated for in the assembly notation. i.e. that an immediate value of 1 in assembler notation actually places the value 0b0000000 in the `SVi` field bits: on execution the `setvl` instruction adds one to the decoded `SVi` field bits, resulting in VL/MVL being set to 1. This allows VL to be set to values ranging from 1 to 128 with only 7 bits instead of 8. Setting VL/MVL to 0 would result in all Vector operations becoming `nop`. If this is truly desired (nop behaviour) then setting VL and MVL to zero is to be done via the [[SVSTATE SPR|sv/sprs]].

Note that setmvli is a pseudo-op, based on RA/RT=0, and setvli likewise

```
setvli   VL=8   : setvl  r0, r0, VL=8, vf=0, vs=1, ms=0
setvli.  VL=8   : setvl. r0, r0, VL=8, vf=0, vs=1, ms=0
setmvli  MVL=8  : setvl  r0, r0, MVL=8, vf=0, vs=0, ms=1
setmvli. MVL=8  : setvl. r0, r0, MVL=8, vf=0, vs=0, ms=1
```

Additional pseudo-op for obtaining VL without modifying it (or any state):

```
getvl  r5      : setvl  r5, r0, vf=0, vs=0, ms=0
getvl. r5      : setvl. r5, r0, vf=0, vs=0, ms=0
```

Note that whilst it is possible to set both MVL and VL from the same immediate, it is not possible to set them to different immediates in the same instruction. Doing so would require two instructions.

Use of setvl results in changes to the SVSTATE SPR. see {SPRs}

Selecting sources for VL

There is considerable opcode pressure, consequently to set MVL and VL from different sources is as follows:

condition	effect
`vs=1, RA=0, RT!=0`	VL,RT set to MIN(MVL, CTR)
`vs=1, RA=0, RT=0`	VL set to MIN(MVL, SVi+1)
`vs=1, RA!=0, RT=0`	VL set to MIN(MVL, RA)
`vs=1, RA!=0, RT!=0`	VL,RT set to MIN(MVL, RA)

The reasoning here is that the opportunity to set RT equal to the immediate `SVi+1` is sacrificed in favour of setting from CTR.

Unusual Rc=1 behaviour

Normally, the return result from an instruction is in `RT`. With it being possible for `RT=0` to mean that `CTR` mode is to be read, some different semantics are needed.

CR Field 0, when `Rc=1`, may be set even if `RT=0`. The reason is that overflow may occur: `VL`, if set either from an immediate or from `CTR`, may not exceed `MAXVL`, and if it is, `CR0.SO` must be set.

In reality it is `VL` being set. Therefore, rather than `CR0` testing `RT` when `Rc=1`, CR0.EQ is set if `VL=0`, CR0.GE is set if `VL` is non-zero.

SUBVL

Sub-vector elements are not be considered "Vertical". The vec2/3/4 is to be considered as if the "single element". Caveats exist for {Swizzle Move} and {Pack / Unpack} when Pack/Unpack is enabled, due to the order in which VL and SUBVL loops are applied being swapped (outer-inner becomes inner-outer)

Examples

Core concept loop

```
loop:
    setvl a3, a0, MVL=8    #  update a3 with vl
                           # (# of elements this iteration)
                           # set MVL to 8
    # do vector operations at up to 8 length (MVL=8)
    # ...
    sub a0, a0, a3   # Decrement count by vl
    bnez a0, loop    # Any more?
```

Loop using Rc=1

```
    my_fn:
      li r3, 1000
      b test
    loop:
      sub r3, r3, r4
      ...
    test:
      setvli. r4, r3, MVL=64
      bne cr0, loop
    end:
      blr
```

Load/Store-Multi (selective)

Up to 64 FPRs will be loaded, here. `r3` is set one per bit for each FP register required to be loaded. The block of memory from which the registers are loaded is contiguous (no gaps): any FP register which has a corresponding zero bit in `r3` is *unaltered*. In essence this is a selective LD-multi with "Scatter" capability.

```
setvli r0, MVL=64, VL=64
sv.fld/dm=r3 *r0, 0(r30) # selective load 64 FP registers
```

Up to 64 FPRs will be saved, here. Again, `r3`

```
setvli r0, MVL=64, VL=64
sv.stfd/sm=r3 *fp0, 0(r30) # selective store 64 FP registers
```

[[!tag standards]]

SVSTATE SPR

The format of the SVSTATE SPR is as follows:

Field	Name	Description
0:6	maxvl	Max Vector Length
7:13	vl	Vector Length
14:20	srcstep	for srcstep = 0..VL-1
21:27	dststep	for dststep = 0..VL-1
28:29	dsubstep	for substep = 0..SUBVL-1
30:31	ssubstep	for substep = 0..SUBVL-1
32:33	mi0	REMAP RA/FRA/BFA SVSHAPE0-3
34:35	mi1	REMAP RB/FRB/BFB SVSHAPE0-3
36:37	mi2	REMAP RC/FRT SVSHAPE0-3
38:39	mo0	REMAP RT/FRT/BF SVSHAPE0-3
40:41	mo1	REMAP EA/RS/FRS SVSHAPE0-3
42:46	SVme	REMAP enable (RA-RT)
47:52	rsvd	reserved
53	pack	PACK (srcstrp reorder)
54	unpack	UNPACK (dststep order)
55:61	hphint	Horizontal Hint
62	RMpst	REMAP persistence
63	vfirst	Vertical First mode

Notes:

- The entries are truncated to be within range. Attempts to set VL to greater than MAXVL will truncate VL.
- Setting srcstep, dststep to 64 or greater, or VL or MVL to greater than 64 is reserved and will cause an illegal instruction trap.

SVSTATE Fields

SVSTATE is a standard SPR that (if REMAP is not activated) contains sufficient self-contaned information for a full context save/restore. SVSTATE contains (and permits setting of):

- MVL (the Maximum Vector Length) - declares (statically) how much of a regfile is to be reserved for Vector elements
- VL - Vector Length
- dststep - the destination element offset of the current parallel instruction being executed
- srcstep - for twin-predication, the source element offset as well.
- ssubstep - the source subvector element offset of the current parallel instruction being executed
- dsubstep - the destination subvector element offset of the current parallel instruction being executed
- vfirst - Vertical First mode. srcstep, dststep and substep **do not advance** unless explicitly requested to do so with pseudo-op svstep (a mode of setvl)
- RMpst - REMAP persistence. REMAP will apply only to the following instruction unless this bit is set, in which case REMAP "persists". Reset (cleared) on use of the `setvl` instruction if used to alter VL or MVL.
- Pack - if set then srcstep/substep VL/SUBVL loop-ordering is inverted.
- UnPack - if set then dststep/substep VL/SUBVL loop-ordering is inverted.
- hphint - Horizontal Parallelism Hint. Indicates that no Hazards exist between groups of elements in sequential multiples of this number (before REMAP). By definition: elements for which `FLOOR(srcstep/hphint)` is equal *before REMAP* are in the same parallelism "group". In Vertical First Mode hardware **MUST ONLY** process elements in the same group, and must stop Horizontal Issue at the last element of a given group. Set to zero to indicate "no hint".
- SVme - REMAP enable bits, indicating which register is to be REMAPed: RA, RB, RC, RT and EA are the canonical (typical) register names associated with each bit, with RA being the LSB and EA being the MSB. See table below for ordering. When `SVme` is zero (0b00000) REMAP is **fully disabled and inactive** regardless of the contents of `SVSTATE`, `mi0-mi2/mo0-mo1`, or the four `SVSHAPEn` SPRs
- mi0-mi2/mo0-mo1 - when the corresponding SVme bit is enabled, these indicate the SVSHAPE (0-3) that the corresponding register (RA etc) should use, as long as the register's corresponding SVme bit is set

Programmer's Note: the fact that REMAP is entirely dormant when `SVme` is zero allows establishment of REMAP context well in advance, followed by utilising `svremap` at a precise (or the very last) moment. Some implementations may exploit this to cache (or take some time to prepare caches) in the background whilst other (unrelated) instructions are being executed. This is particularly important to bear in mind when using `svindex` which will require hardware to perform (and cache) additional GPR reads.

Programmer's Note: when REMAP is activated it becomes necessary on any context-switch (Interrupt or Function call) to detect (or know in advance) that REMAP is enabled and to additionally save/restore the four SVSHAPE SPRs, SVHAPE0-3. Given that this is expected to be a rare occurrence it was deemed unreasonable to burden every context-switch or function call with mandatory save/restore of SVSHAPEs, and consequently it is a *callee* (and Trap Handler) responsibility. Callees (and Trap Handlers) **MUST** avoid using all and any SVP64 instructions during the period where state could be adversely affected. SVP64 purely relies on Scalar instructions, so Scalar instructions (except the SVP64 Management ones and mtspr and mfspr) are 100% guaranteed to have zero impact on SVP64 state.

Max Vector Length (maxvl)

MAXVECTORLENGTH is the same concept as MVL in RISC-V RVV, except that it is variable length and may be dynamically set (normally from an immediate field only). MVL is limited to 7 bits (in the first version of SVP64) and consequently the

maximum number of elements is limited to between 0 and 127.

Programmer's Note: Except by directly using `mtspr` on SVSTATE, which may result in performance penalties on some hardware implementations, SVSTATE's `maxvl` field may only be set **statically** as an immediate, by the `setvl` instruction. It may **NOT** be set dynamically from a register. Compiler writers and assembly programmers are expected to perform static register file analysis, subdivision, and allocation and only utilise `setvl`. Direct writing to SVSTATE in order to "bypass" this Note could, in less-advanced implementations, potentially cause stalling, particularly if SVP64 instructions are issued directly after the `mtspr` to SVSTATE.

Vector Length (vl)

The actual Vector length, the number of elements in a "Vector", `SVSTATE.vl` may be set entirely dynamically at runtime from a number of sources. `setvl` is the primary instruction for setting Vector Length. `setvl` is conceptually similar but different from the Cray, SX Aurora, and RISC-V RVV equivalent. Similar to RVV, VL is set to be within the range 0 <= VL <= MVL. Unlike RVV, VL is set **exactly** according to the following:

```
VL = (RT|0) = MIN(vlen, MVL)
```

where 0 <= MVL <= 127 and vlen may come from an immediate, `RA`, or from the `CTR` SPR, depending on options selected with the `setvl` instruction.

Programmer's Note: conceptual understanding of Cray-style Vectors is far beyond the scope of the Power ISA Technical Reference. Guidance on the 50-year-old Cray Vector paradigm is best sought elsewhere: good studies include Academic Courses given on the 1970s Cray Supercomputers over at least the past three decades.

SUBVL - Sub Vector Length

This is a "group by quantity" that effectively asks each iteration of the hardware loop to load SUBVL elements of width elwidth at a time. Effectively, SUBVL is like a SIMD multiplier: instead of just 1 operation issued, SUBVL operations are issued.

The main effect of SUBVL is that predication bits are applied per **group**, rather than by individual element. Legal values are 0 to 3, representing 1 operation (1 element) thru 4 operations (4 elements) respectively. Elements are best though of in the context of 3D, Audio and Video: two Left and Right Channel "elements" or four ARGB "elements", or three XYZ coordinate "elements".

`subvl` is again primarily set by the `setvl` instruction. Not to be confused with `hphint`.

Directly related to `subvl` is the `pack` and `unpack` Mode bits of `SVSTATE`. See `svstep` instruction for how to set Pack and Unpack Modes.

Horizontal Parallelism

A problem exists for hardware where it may not be able to detect that a programmer (or compiler) knows of opportunities for parallelism and lack of overlap between loops.

For hphint, the number chosen must be consistently executed **every time**. Hardware is not permitted to execute five computations for one instruction then three on the next. hphint is a hint from the compiler to hardware that exactly this many elements may be safely executed in parallel, without hazards (including Memory accesses). Interestingly, when hphint is set equal to VL, it is in effect as if Vertical First mode were not set, because the hardware is given the option to run through all elements in an instruction. This is exactly what Horizontal-First is: a for-loop from 0 to VL-1 except that the hardware may *choose* the number of elements.

Note to programmers: changing VL during the middle of such modes should be done only with due care and respect for the fact that SVSTATE has exactly the same peer-level status as a Program Counter.

SVL-Form

Add the following to Book I, 1.6.1, SVL-Form

```
|0     |6    |11    |16   |23 |24 |25 |26    |31 |
| PO   |  RT |   RA | SVi |ms |vs |vf |   XO |Rc |
| PO   |  RT | /    | SVi |/  |/  |vf |   XO |Rc |
```

- Add `SVL` to `RA (11:15)` Field in Book I, 1.6.2
- Add `SVL` to `RT (6:10)` Field in Book I, 1.6.2
- Add `SVL` to `Rc (31)` Field in Book I, 1.6.2
- Add `SVL` to `XO (26:31)` Field in Book I, 1.6.2

Add the following to Book I, 1.6.2

```
ms (23)
    Field used in Simple-V to specify whether MVL (maxvl in the SVSTATE SPR)
    is to be set
    Formats: SVL
vf (25)
    Field used in Simple-V to specify whether "Vertical" Mode is set
    (vfirst in the SVSTATE SPR)
    Formats: SVL
vs (24)
    Field used in Simple-V to specify whether VL (vl in the SVSTATE SPR) is to be set
    Formats: SVL
SVi (16:22)
     Simple-V immediate field used by setvl for setting VL or MVL
     (vl, maxvl in the SVSTATE SPR)
     and used as a "Mode of Operation" selector in svstep
     Formats: SVL
```

Appendices

```
Appendix E Power ISA sorted by opcode
Appendix F Power ISA sorted by version
Appendix G Power ISA sorted by Compliancy Subset
Appendix H Power ISA sorted by mnemonic
```

Form	Book	Page	Version	mnemonic	Description
SVL	I	#	3.0B	svstep	Vertical-First Stepping and status reporting
SVL	I	#	3.0B	setvl	Cray-like establishment of Looping (Vector) context

[[!tag opf_rfc]]