OPF ISA WG External RFC LS001 v2 14Sep2022

- RFC Author: Luke Kenneth Casson Leighton.
- RFC Contributors/Ideas: Brad Frey, Paul Mackerras, Konstantinos Magritis, Cesar Strauss, Jacob Lifshay, Toshaan Bharvani, Dimitry Selyutin, Andrey Miroshnikov
- Funded by NLnet under the Privacy and Enhanced Trust Programme, EU Horizon2020 Grant 825310
- https://git.openpower.foundation/isa/PowerISA/issues/64 [[ls001/discussion]]

This proposal is to extend the Power ISA with an Abstract RISC-Paradigm Vectorisation Concept that may be orthogonally applied to **all and any** suitable Scalar instructions, present and future, in the Scalar Power ISA. The Vectorisation System is called "Simple-V" and the Prefix Format is called "SVP64". **Simple-V is not a Traditional Vector ISA and therefore does not add Vector opcodes or regfiles**. An ISA Concept similar to Simple-V was originally invented in 1994 by Peter Hsu (Architect of the MIPS R8000) but was dropped as MIPS did not have an Out-of-Order Microarchitecture at the time.

Simple-V is designed for Embedded Scenarios right the way through Audio/Visual DSPs to 3D GPUs and Supercomputing. As it does **not** add actual Vector Instructions, relying solely and exclusively on the **Scalar** ISA, it is **Scalar** instructions that need to be added to the **Scalar** Power ISA before Simple-V may orthogonally Vectorise them.

The goal of RED Semiconductor Ltd, an OpenPOWER Stakeholder, is to bring to market mass-volume general-purpose compute processors that are competitive in the 3D GPU Audio Visual DSP EDGE IoT desktop chromebook netbook smartphone laptop markets, performance-leveraged by Simple-V. To achieve this goal both Simple-V and accompanying Scalar** Power ISA instructions are needed. These include IEEE754 Transcendentals AV cryptographic Biginteger and bitmanipulation operations present in ARM Intel AMD and many other ISAs. Three additional FP-related sets are needed (missing from SFFS) - int_fp_mv fclass and fcvt and one set named crweird increase the capability of CR Fields.

*Thus as the primary motivation is to create a **Hybrid 3D CPU-GPU-VPU ISA** it becomes necesary to consider the Architectural Resource Allocation of not just Simple-V but the 80-100 Scalar instructions all at the same time.*

It is also critical to note that Simple-V **does not modify the Scalar Power ISA**, that **only** Scalar words may be Vectorised, and that Vectorised instructions are **not** permitted to be different from their Scalar words (`addi` must use the same Word encoding as `sv.addi`, and any new Prefixed instruction added **must** also be added as Scalar). The sole semi-exception is Vectorised Branch Conditional, in order to provide the usual Advanced Branching capability present in every Commercial 3D GPU ISA, but it is the *Vectorised* Branch-Conditional that is augmented, not Scalar Branch.

Basic principle

The inspiration for Simple-V came from the fact that on examination of every Vector ISA pseudocode encountered the Vector operations were expressed as a for-loop on a Scalar element operation, and then both a Scalar **and** a Vector instruction was added. With Zero-Overhead Looping *already* being common for over four decades it felt natural to separate the looping at both the ISA and the Hardware Level and thus provide only Scalar instructions (instantly halving the number of instructions), but rather than go the VLIW route (TI MSP Series) keep closely to existing Power ISA standard Scalar execution.

Thus the basic principle of Simple-V is to provide a Precise-Interruptible Zero-Overhead Loop system[1] with associated register "offsetting" which augments a Suffixed instruction as a "template", incrementing the register numbering progressively *and automatically* each time round the "loop". Thus it may be considered to be a form of "Sub-Program-Counter" and at its simplest level can replace a large sequence of regularly-increasing loop-unrolled instructions with just two: one to set the Vector length and one saying where to start from in the regfile.

On this sound and profoundly simple concept which leverages *Scalar* Micro-architectural capabilities much more comprehensive festures are easy to add, working up towards an ISA that easily matches the capability of powerful 3D GPU Vector Supercomputing ISAs, without ever adding even one single Vector opcode.

Extension Levels

Simple-V has been subdivided into levels akin to the Power ISA Compliancy Levels. For now let us call them "SV Extension Levels" to differentiate the two. The reason for the SV Extension Levels is the same as for the Power ISA Compliancy Levels (SFFS, SFS): to not overburden implementors with features that they do not need. *There is no dependence between the two types of Levels.* The resources below therefore are not all required for all SV Extension Levels but they are all required to be reserved.

Binary Interoperability

Power ISA has a reputation as being long-term stable. **Simple-V guarantees binary interoperability** by defining fixed register file bitwidths and size for a given set of instructions. The seduction of permitting different implementors to choose a register file bitwidth and size with the same instructions unfortunately has the catastrophic side-effect of introducing not only binary incompatibility but silent data corruption as well as no means to trap-and-emulate differing bitwidths.[2]

"Silicon-Partner" Scalability is identical to attempting to run 64-bit Power ISA binaries without setting - or having `MSR.SF` - on "Scaled" 32-bit hardware: **the same opcodes** were shared between 32 and 64 bit. `RESERVED` space is thus crucial to have, in

[1] first introduced in DSPs, Zero-Overhead Loops are astoundingly effective in reducing total number of instructions executed or needed. ZOLC reduces instructions by **25 to 80 percent**.

[2] imagine a hypothetical future VSX-256 using the exact same instructions as VSX. the binary incompatibility introducrd would catastrophically **and retroactively** damage existing IBM POWER8,9,10 hardware's reputation and that of Power ISA overall.

order to provide the **OPF ISA WG** - not implementors ("Silicon Partners") - with the option to properly review and decide any (if any) future expanded register file bitwidths and sizes[3], **under explicitly-distinguishable encodings** so as to guarantee long-term stability and binary interoperability.

Hardware Implementations

The fundamental principle of Simple-V is that it sits between Issue and Decode, pausing the Program-Counter to service a "Sub-PC" hardware for-loop. This is very similar to Zero-Overhead Loops in High-end DSPs (TI MSP Series).

Considerable effort has been expended to ensure that Simple-V is practical to implement on an extremely wide range of Industry-wide common **Scalar** micro-architectures. Finite State Machine (for ultra-low-resource and Mission-Critical), In-order single-issue, all the way through to Great-Big Out-of-Order Superscalar Multi-Issue. The SV Extension Levels specifically recognise these differing scenarios.

SIMD back-end ALUs particularly those with element-level predicate masks may be exploited to good effect with very little additional complexity to achieve high throughput, even on a single-issue in-order microarchitecture. As usually becomes quickly apparent with in-order, its limitations extend also to when Simple-V is deployed, which is why Multi-Issue Out-of-Order is the recommended (but not mandatory) Scalar Micro-architecture. Byte-level write-enable regfiles (like SRAMs) are strongly recommended, to avoid a Read-Modify-Write cycle.

The only major concern is in the upper SV Extension Levels: the Hazard Management for increased number of Scalar Registers to 128 (in current versions) but given that IBM POWER9/10 has VSX register numbering 64, and modern GPUs have 128, 256 and even 512 registers this was deemed acceptable. Strategies do exist in hardware for Hazard Management of such large numbers of registers, even for Multi-Issue microarchitectures.

Simple-V Architectural Resources

- No new Interrupt types are required. No modifications to existing Power ISA opcodes are required. No new Register Files are required (all because Simple-V is a category of Zero-Overhead Looping on Scalar instructions)
- GPR FPR and CR Field Register extend to 128. A future version may extend to 256 or beyond[4] or also extend VSX[5]
- 24-bits are needed within the main SVP64 Prefix (equivalent to a 2-bit XO)
- Another 24-bit (a second 2-bit XO) is needed for a planned future encoding, currently named "SVP64-Single"[6]
- A third 24-bits (third 2-bit XO) is strongly recommended to be `RESERVED` such that future unforeseen capability is needed (although this may be alternatively achieved with a mandatory PCR or MSR bit)
- To hold all Vector Context, four SPRs are needed. (Some 32/32-to-64 aliases are advantageous but not critical).
- Five 6-bit XO (A-Form) "Management" instructions are needed. These are Scalar 32-bit instructions and *may* be 64-bit-extended in future (safely within the SVP64 space: no need for an EXT001 encoding).

Summary of Simple-V Opcode space

- 75% of one Major Opcode (equivalent to the rest of EXT017)
- Five 6-bit XO 32-bit operations.

No further opcode space *for Simple-V* is envisaged to be required for at least the next decade (including if added on VSX)

Simple-V SPRs

- **SVSTATE** - Vectorisation State sufficient for Precise-Interrupt Context-switching and no adverse latency, it may be considered to be a "Sub-PC" and as such absolutely must be treated with the same respect and priority as MSR and PC.
- **SVSHAPE0-3** - these are 32-bit and may be grouped in pairs, they REMAP (shape) the Vectors[7]
- **SVLR** - again similar to LR for exactly the same purpose, SVSTATE is swapped with SVLR by SV-Branch-Conditional for exactly the same reason that NIA is swapped with LR

Vector Management Instructions

These fit into QTY 5of 6-bit XO 32-bit encoding (svshape and svshape2 share the same space):

- **setvl** - Cray-style Scalar Vector Length instruction
- **svstep** - used for Vertical-First Mode and for enquiring about internal state
- **svremap** - "tags" registers for activating REMAP
- **svshape** - convenience instruction for quickly setting up Matrix, DCT, FFT and Parallel Reduction REMAP
- **svshape2** - additional convenience instruction to set up "Offset" REMAP (fits within svshape's XO encoding)
- **svindex** - convenience instruction for setting up "Indexed" REMAP.

[3] an MSR bit or bits, conceptually equivalent to `MSR.SF` and added for the same reasons, would suffice perfectly.

[4] Prefix opcode space (or MSR bits) **must** be reserved in advance to do so, in order to avoid the catastrophic binary-incompatibility mistake made by RISC-V RVV and ARM SVE/2

[5] A future version or other Stakeholder *may* wish to drop Simple-V onto VSX: this would be a separate RFC

[6] SVP64-Single is remarkably similar to the "bit 1" of EXT001 being set to indicate that the 64-bits is to be allocated in full to a new encoding, but in fact SVP64-single still embeds v3.0 Scalar operations.

[7] although SVSHAPE0-3 should, realistically, be regarded as high a priority as SVSTATE, and given corresponding SVSRR and SVLR equivalents, it was felt that having to context-switch **five** SPRs on Interrupts and function calls was too much.

SVP64 24-bit Prefixes

The SVP64 24-bit Prefix (RM) options aim to reduce instruction count and assembler complexity. These Modes do not interact with SVSTATE per se. SVSTATE primarily controls the looping (quantity, order), RM influences the *elements* (the Suffix). There is however some close interaction when it comes to predication. REMAP is outlined separately.

- **element-width overrides**, which dynamically redefine each SFFS or SFS Scalar prefixed instruction to be 8-bit, 16-bit, 32-bit or 64-bit operands **without requiring new 8/16/32 instructions.**[8] This results in full BF16 and FP16 opcodes being added to the Power ISA **without adding BF16 or FP16 opcodes** including full conversion between all formats.
- **predication**. this is an absolutely essential feature for a 3D GPU VPU ISA. CR Fields are available as Predicate Masks hence the reason for their extension to 128. Twin-Predication is also provided: this may best be envisaged as back-to-back VGATHER-VSCATTER but is not restricted to LD/ST, its use saves on instruction count. Enabling one or other of the predicates provides all of the other types of operations found in Vector ISAs (VEXTRACT, VINSERT etc) again with no need to actually provide explicit such instructions.
- **Saturation**. applies to **all** LD/ST and Arithmetic and Logical operations (without adding explicit saturation ops)
- **Reduction and Prefix-Sum** (Fibonnacci Series) Modes, including a "Reverse Gear" (running loops backwards).
- **vec2/3/4 "Packing" and "Unpacking"** (similar to VSX `vpack` and `vpkss`) accessible in a way that is easier than REMAP, added for the same reasons that drove `vpack` and `vpkss` etc. to be added: pixel, audio, and 3D data manipulation. With Pack/Unpack being part of SVSTATE it can be applied *in-place* saving register file space (no copy/mv needed).
- **Load/Store "fault-first"** speculative behaviour, identical to SVE and RVV Fault-first: provides auto-truncation of a speculative sequential parallel LD/ST batch, helping solve the "SIMD Considered Harmful" stripmining problem from a Memory Access perspective.
- **Data-Dependent Fail-First**: a 100% Deterministic extension of the LDST ffirst concept: first `Rc=1 BO test` failure terminates looping and truncates VL to that exact point. Useful for implementing algorithms such as `strcpy` in around 14 high-performance Vector instructions, the option exists to include or exclude the failing element.
- **Predicate-result**: a strategic mode that effectively turns all and any operations into a type of `cmp`. An `Rc=1 BO test` is performed and if failing that element result is **not** written to the regfile. The `Rc=1` Vector of co-results **is** always written (subject to usual predication). Termed "predicate-result" because the combination of producing then testing a result is as if the test was in a follow-up predicated copy/mv operation, it reduces regfile pressure and instruction count. Also useful on saturated or other overflowing operations, the overflowing elements may be excluded from outputting to the regfile then post-analysed outside of critical hot-loops.

RM Modes

There are five primary categories of instructions in Power ISA, each of which needed slightly different Modes. For example, saturation and element-width overrides are meaningless to Condition Register Field operations, and Reduction is meaningless to LD/ST but Saturation saves register file ports in critical hot-loops. Thus the 24 bits may be suitably adapted to each category.

- Normal - arithmetic and logical including IEEE754 FP
- LD/ST immediate - includes element-strided and unit-strided
- LD/ST indexed
- CR Field ops
- Branch-Conditional - saves on instruction count in 3D parallel if/else

It does have to be pointed out that there is huge pressure on the Mode bits. There was therefore insufficient room, unlike the way that EXT001 was designed, to provide "identifying bits" *without first partially decoding the Suffix*.

Some considerable care has been taken to ensure that Decoding may be performed in a strict forward-pipelined fashion that, aside from changes in SVSTATE (necessarily cached and propagated alongside MSR and PC) and aside from the initial 32/64 length detection (also kept simple), a Multi-Issue Engine would have no difficulty (performance maximisable). With the initial partial RM Mode type-identification decode performed above the Vector operations may then easily be passed downstream in a fully forward-progressive piplined fashion to independent parallel units for further analysis.

Vectorised Branch-Conditional

As mentioned in the introduction this is the one sole instruction group that is different pseudocode from its scalar equivalent. However even there its various Mode bits and options can be set such that in the degenerate case the behaviour becomes identical to Scalar Branch-Conditional.

The two additional Modes within Vectorised Branch-Conditional, both of which may be combined, are `CTR-Mode` and `VLI-Test` (aka "Data Fail First"). CTR Mode extends the way that CTR may be decremented unconditionally within Scalar Branch-Conditional, and not only makes it conditional but also interacts with predication. VLI-Test provides the same option as Data-Dependent Fault-First to Deterministically truncate the Vector Length at the fail **or success** point.

Boolean Logic rules on sets (treating the Vector of CR Fields to be tested by `BO` as a set) dictate that the Branch should take place on either 'ALL' tests succeeding (or failing) or whether 'SOME' tests succeed (or fail). These options provide the ability to cover the majority of Parallel 3D GPU Conditions, saving up to **twelve** instructions especially given the close interaction with CTR in hot-loops.[9]

Also `SVLR` is introduced, which is a parallel twin of `LR`, and saving and restoring of LR and SVLR may be deferred until the final decision as to whether to branch. In this way `sv.bclrl` does not corrupt `LR`.

Vectorised Branch-Conditional due to its side-effects (e.g. reducing CTR or truncating VL) has practical uses even if the Branch is deliberately set to the next instruction (CIA+8). For example it may be used to reduce CTR by the number of bits set in a GPR, if that GPR is given as the predicate mask `sv.bc/pm=r3`.

[8] elwidth overrides does however mean that all SFS / SFFS pseudocode will need rewriting to be in terms of XLEN. This has the indirect side-effect of automatically making a 32-bit Scalar Power ISA Specification possible, as well as a future 128-bit one (Cross-reference: RISC-V RV32 and RV128

[9] adding a parity (XOR) option was too much. instead a parallel-reduction on `crxor` may be used in combination with a Scalar Branch.

LD/ST RM Modes

Traditional Vector ISAs have vastly more (and more complex) addressing modes than Scalar ISAs: unit strided, element strided, Indexed, Structure Packing. All of these had to be jammed in on top of existing Scalar instructions **without modifying or adding new Scalar instructions**. A small conceptual "cheat" was therefore needed. The Immediate (D) is in some Modes multiplied by the element index, which gives us element-strided. For unit-strided the width of the operation (`ld`, 8 byte) is multiplied by the element index and *substituted* for "D" when the immediate, D, is zero. Modifications to support this "cheat" on top of pre-existing Scalar HDL (and Simulators) have both turned out to be minimal.[10] Also added was the option to perform signed or unsigned Effective Address calculation, which comes into play only on LD/ST Indexed, when elwidth overrides are used. Another quirk: `RA` is never allowed to have its width altered: it remains 64-bit, as it is the Base Address.

One confusing thing is the unfortunate naming of LD/ST Indexed and REMAP Indexed: some care is taken in the spec to discern the two. LD/ST Indexed is Scalar `EA=RA+RB` (where **either** RA or RB may be marked as Vectorised), where obviously the order in which that Vector of RA (or RB) is read in the usual linear sequential fashion. REMAP Indexed affects the **order** in which the Vector of RA (or RB) is accessed, according to a schedule determined by *another* vector of offsets in the register file. Effectively this combines VSX `vperm` back-to-back with LD/ST operations *in the calculation of each Effective Address* in one instruction.

For DCT and FFT, normally it is very expensive to perform the "bit-inversion" needed for address calculation and/or reordering of elements. DCT in particular needs both bit-inversion *and Gray-Coding* offsets (a complexity that often "justifies" full assembler loop-unrolling). DCT/FFT REMAP **automatically** performs the required offset adjustment to get data loaded and stored in the required order. Matrix REMAP can likewise perform up to 3 Dimensions of reordering (on both Immediate and Indexed), and when combined with vec2/3/4 the reordering can even go as far as four dimensions (four nested fixed size loops).

Twin Predication is worth a special mention. Many Vector ISAs have special LD/ST `VCOMPRESS` and `VREDUCE` instructions, which sequentially skip elements based on predicate mask bits. They also add special `VINSERT` and `VEXTRACT` Register-based instructions to compensate for lack of single-element LD/ST (where in Simple-V you just use Scalar LD/ST). Also Broadcasting (`VSPLAT`) is either added to LDST or as Register-based.

All of the above modes are covered by Twin-Predication

In particular, a special predicate mode `1<<r3` uses the register `r3` *binary* value, converted to single-bit unary mask, effectively as a single (Scalar) Index *runtime*-dynamic offset into a Vector.[11] Combined with the (mis-named) "mapreduce" mode when used as a source predicate a `VSPLAT` (broadcast) is performed. When used as a destination predicate `1<<r3` provides `VINSERT` behaviour.

Also worth an explicit mention is that Twin Predication when using different source from destination predicate masks effectively combines back-to-back `VCOMPRESS` and `VEXPAND` (in a single instruction), and, further, that the benefits of Twin Predication are not limited to LD/ST, they may be applied to Arithmetic, Logical and CR Field operations as well.

Overall the LD/ST Modes available are astoundingly powerful, especially when combining arithmetic (lharx) with saturation, element-width overrides, Twin Predication, vec2/3/4 Structure Packing *and* REMAP, the combinations far exceed anything seen in any other Vector ISA in history, yet are really nothing more than concepts abstracted out in pure RISC form.[12]

CR Field RM Modes.

CR Field operations (`crand` etc.) are somewhat underappreciated in the Power ISA. The CR Fields however are perfect for providing up to four separate Vectors of Predicate Masks: `EQ LT GT SO` and thus some special attention was given to first making transfer between GPR and CR Fields much more powerful with the crweird operations, and secondly by adding powerful binary and ternary CR Field operations into the bitmanip extension.[13]

On these instructions RM Modes may still be applied (mapreduce and Data-Dependent Fail-first). The usefulness of being able to auto-truncate subsequent Vector Processing at the point at which a CR Field test fails, based on any arbitary logical operation involving `three` CR Field Vectors (`crternlogi`) should be clear, as should the benefits of being able to do mapreduce and REMAP Parallel Reduction on `crternlogi`: dramatic reduction in instruction count for Branch-based control flow when faced with complex analysis of multiple Vectors, including XOR-reduction (parity).

Overall the addition of the CR Operations and the CR RM Modes is about getting instruction count down and increasing the power and flexibility of CR Fields as pressed into service for the purpose of Predicate Masks.

SVP64Single 24-bits

The `SVP64-Single` 24-bit encoding focusses primarily on ensuring that all 128 Scalar registers are fully accessible, provides element-width overrides, one-bit predication and brings Saturation to all existing Scalar operations. BF16 and FP16 are thus provided in the Scalar Power ISA without one single explicit FP16 or BF16 32-bit opcode being added. The downside: such Scalar operations are all 64-bit encodings.

As SVP64Single is new and still under development, space for it may instead be `RESERVED`. It is however necessary in *some* form as there are limitations in SVP64 Register numbering, particularly for 4-operand instructions, that can only be easily overcome by SVP64Single.

[10] Setting this "multiplier" to 1 clearly leaves pre-existing Scalar behaviour completely intact as a degenerate case.

[11] Effectively: `GPR(RA+r3)`

[12] At least the CISC "auto-increment" modes are not present, from the CDC 6600 and Motorola 68000! although these would be fun to introduce they do unfortunately make for 3-in 3-out register profiles, all 64-bit, which explains why the 6600 and 68000 had separate special dedicated address regfiles.

[13] the alternative to powerful transfer instructions between GPR and CR Fields was to add the full duplicated suite of BMI and TBM operations present in GPR (popcnt, cntlz, set-before-first) as CR Field Operations. all of which was deemed inappropriate.

Vertical-First Mode

This is a Computer Science term that needed first to be invented. There exists only one other Vertical-First Vector ISA in the world: Mitch Alsup's VVM Extension for the 66000, details of which may be obtained publicly on `comp.arch` or directly from Mitch Alsup under NDA. Several people have independently derived Vertical-First: it simply did not have a Computer Science term associated with it.

If we envisage register and Memory layout to be Horizontal and instructions to be Vertical, and to then have some form of Loop System (wherther Zero-Overhead or just branch-conditional based) it is easier to then conceptualise VF vs HF Mode:

- Vertical-First progresses through *instructions* first before moving on to the next *register* (or Memory-address in the case of Mitch Alsup's VVM).
- Horizontal-First (also known as Cray-style Vectors) progresses through **registers** (or, register *elements* in traditional Cray-Vector ISAs) in full before moving on to the next *instruction*.

Mitch Alsup's VVM Extension is a form of hardware-level auto-vectorisation based around Zero-Overhead Loops. Using a Variable-Length Encoding all loop-invariant registers are "tagged" such that the Hazard Management Engine may perform optimally and do less work in automatically identifying parallelism opportunities. With it not being appropriate to use Variable-Length Encoding in the Power ISA a different much more explicit strategy was taken in Simple-V.

The biggest advantage inherent in Vertical-First is that it is very easy to introduce into compilers, because all looping, as far as programs is concerned, remains expressed as *Scalar assembler*.[14] Whilst Mitch Alsup's VVM biggest strength is its hardware-level auto-vectorisation but is limited in its ability to call functions, Simple-V's Vertical-First provides explicit control over the parallelism ("hphint")[15] and also allows for full state to be stored/restored (SVLR combined with LR), permitting full function calls to be made from inside Vertical-First Loops, and potentially allows arbitrarily-depth nested VF Loops.

Simple-V Vertical-First Looping requires an explicit instruction to move `SVSTATE` regfile offsets forward: `svstep`. An early version of Vectorised Branch-Conditional attempted to merge the functionality of `svstep` into `sv.bc`: it became CISC-like in its complexity and was quickly reverted.

Simple-V REMAP subsystem

REMAP is extremely advanced but brings features already present in other DSPs and Supercomputing ISAs. The usual sequential progression through elements is pushed through a hardware-defined *fully Deterministic* "remapping". Normally (without REMAP) algorithms are costly or convoluted to implement. They are typically implemented as hard-coded fully loop-unrolled assembler which is often auto-generated by specialist tools, or written entirely by hand. All REMAP Schedules *including Indexed* are 100% Deterministic from their point of declaration, making it possible to forward-plan Issue, Memory access and Register Hazard Management in Multi-Issue Micro-architectures.

If combined with Vertical-First then much more complex operations may exploit REMAP Schedules, such as Complex Number FFTs, by using Scalar intermediary temporary registers to compute results that have a Vector source or destination or both. Contrast this with a Standard Horizontal-First Vector ISA where the only way to perform Vectorised Complex Arithmetic would be to add Complex Vector Arithmetic operations, because due to the Horizontal (element-level) progression there is no way to utilise intermediary temporary (scalar) variables.[16]

- **DCT/FFT** REMAP brings more capability than TI's MSP-Series DSPs and Qualcom Hexagon DSPs, and is not restricted to Integer or FP. (Galois Field is possible, implementing NTT). Operates *in-place* significantly reducing register usage.
- **Matrix** REMAP brings more capability than any other Matrix Extension (AMD GPUs, Intel, ARM), not being restricted to Power-2 sizes. Also not limited to the type of operation, it may perform Warshall Transitive Closure, Integer Matrix, Bitmanipulation Matrix, Galois Field (carryless mul) Matrix, and with care potentially Graph Maximum Flow as well. Also suited to Convolutions, Matrix Transpose and rotate, *all* of which is in-place.
- **General-purpose Indexed** REMAP, this option is provided to implement an equivalent of VSX `vperm`, as a general-purpose catch-all means of covering algorithms outside of the other REMAP Engines.
- **Parallel Reduction** REMAP, performs an automatic map-reduce using *any suitable scalar operation*.

All REMAP Schedules are Precise-Interruptible. No latency penalty is caused by the fact that the Schedule is Parallel-Reduction, for example. The operations are Issued (Deterministically) as **Scalar** operations and thus any latency associated with **Scalar** operation Issue exactly as in a **Scalar** Micro-architecture will result. Contrast this with a Standard Vector ISA where frequently there is either considerable interrupt latency due to requiring a Parallel Reduction to complete in full, or partial results to be discarded and re-started should a high-priority Interrupt occur in the middle.

Note that predication is possible on REMAP but is hard to use effectively. It is often best to make copies of data (`VCOMPRESS`) then apply REMAP.

[14] Compiler auto-vectorisation for best exploitation of SIMD and Vector ISAs on Scalar programming languages (c, c++) is an Indusstry-wide known-hard decades-long problem. Cross-reference the number of hand-optimised assembler algorithms.

[15] intended for use when the compiler has determined the extent of Memory or register aliases in loops: `a[i] += a[i+4]` would necessitate a Vertical-First hphint of 4

[16] a case could be made for constructing Complex number arithmetic using multiple sequential Horizontal-First (Cray-style Vector) instructions. This may not be convenient in the least when REMAP is involved (such as Parallel Reduction of Complex Multiply).

Scalar Operations

The primary reason for mentioning the additional Scalar operations is because they are so numerous, with Power ISA not having advanced in the *general purpose* compute area in the past 12 years, that some considerable care is needed.

Summary: **Including Simple-V, to fit everything at least 75% of 3 separate Major Opcodes would be required**

Candidates (for all but the X-Form instructions) include:

- EXT006 (80% free)
- EXT017 (75% free but not recommended)
- EXT001 (50% free)
- EXT009 (100% free)
- EXT005 (100% free)
- brownfield space in EXT019 (25% but NOT recommended)

SVP64, SVP64-Single and SVP64-Reserved would require on their own each 25% of one Major Opcode for a total of 75% of one Major Opcode. The remaining **Scalar** opcodes, due to there being two separate sets of operations with 16-bit immediates, will require the other space totalling two 75% Majors.

Note critically that:

- Unlike EXT001, SVP64's 24-bits may **not** hold also any Scalar operations. There is no free available space: a 25th bit would be required. The entire 24-bits is **required** for the abstracted Hardware-Looping Concept **even when these 24-bits are zero**
- Any Scalar 64-bit instruction (regardless of how it is encoded) is unsafe to then Vectorise because this creates the situation of Prefixed-Prefixed, resulting in deep complexity in Hardware Decode at a critical juncture, as well as introducing 96-bit instructions.
- **All** of these Scalar instructions are candidates for Vectorisation. Thus none of them may be 64-bit-Scalar-only.

Minor Opcodes to fit candidates above

In order of size, for bitmanip and A/V DSP purposes:

- QTY 3of 2-bit XO: ternlogi, crternlogi, grevlogi
- QTY 7of 3-bit XO: xpermi, binlut, grevlog, swizzle-mv/fmv, bitmask, bmrevi
- QTY 8of 5/6-bit (A-Form): xpermi, bincrflut, bmask, fmvis, fishmv, bmrev, Galois Field
- QTY 30of 10-bit (X-Form): cldiv/mul, av-min/max/diff, absdac, xperm etc. (easily fit EXT019, EXT031).

Note: Some of the Galois Field operations will require QTY 1of Polynomial SPR (per userspace supervisor hypervisor).

EXT004

For biginteger math, two instructions in the same space as "madd" are to be proposed. They are both 3-in 2-out operations taking or producing a 64-bit "pair" (like RTp), and perform 128/64 mul and div/mod operations respectively. These are **not** the same as VSX operations which are 128/128, and they are **not** the same as existing Scalar mul/div/mod, all of which are 64/64 (or 64/32).

EXT059 and EXT063

Additionally for High-Performance Compute and Competitive 3D GPU, IEEE754 FP Transcendentals are required, as are some DCT/FFT "Twin-Butterfly" operations. For each of EXT059 and EXT063:

- QTY 33of X-Form "1-argument" (fsin, fsins, fcos, fcoss)
- QTY 15of X-Form "2-argument" (pow, atan2, fhypot)
- QTY 5of A-Form "3-in 2-out" FP Butterfly operations for DCT/FFT
- QTY 8of X-Form "2-in 2-out" FP Butterfly operations (again for DCT/FFT)
- An additional 16 instructions for IEEE754-2019 (fminss/fmaxss, fminmag/fmaxmag) under evaluation as of 08Sep2022

Adding new opcodes.

With Simple-V being a type of Zero-Overhead Loop Engine on top of Scalar operations some clear guidelines are needed on how both existing "Defined Words" (Public v3.1 Section 1.6.3 term) and future Scalar operations are added within the 64-bit space. Examples of legal and illegal allocations are given later.

The primary point is that once an instruction is defined in Scalar 32-bit form its corresponding space **must** be reserved in the SVP64 area with the exact same 32-bit form, even if that instruction is "Unvectoriseable" (`sc`, `sync`, `rfid` and `mtspr` for example). Instructions may **not** be added in the Vector space without also being added in the Scalar space, and vice-versa, *even if Unvectoriseable*.

This is extremely important because the worst possible situation is if a conflicting Scalar instruction is added by another Stakeholder, which then turns out to be Vectoriseable: it would then have to be added to the Vector Space with a *completely different Defined Word* and things go rapidly downhill in the Decode Phase from there. Setting a simple inviolate rule helps avoid this scenario but does need to be borne in mind when discussing potential allocation schemes, as well as when new Vectoriseable Opcodes are proposed for addition by future RFCs: the opcodes **must** be uniformly added to Scalar **and** Vector spaces, or added in one and reserved in the other, or not added at all in either.[17]

[17] two efforts were made to mix non-uniform encodings into Simple-V space: one deliberate to see how it would go, and one accidental. They both went extremely badly, the deliberate one costing over two months to add then remove.

Potential Opcode allocation solution (superseded)

Note this scheme is superseded below but kept for completeness as it defines terms and context. There are unfortunately some inviolate requirements that directly place pressure on the EXT000-EXT063 (32-bit) opcode space to such a degree that it risks jeapordising the Power ISA. These requirements are:

- all of the scalar operations must be Vectoriseable
- all of the scalar operations intended for Vectorisation must be in a 32-bit encoding (not prefixed-prefixed to 96-bit)
- bringing Scalar Power ISA up-to-date from the past 12 years needs 75% of two Major opcodes all on its own

There exists a potential scheme which meets (exceeds) the above criteria, providing plenty of room for both Scalar (and Vectorised) operations, *and* provides SVP64-Single with room to grow. It is based loosely around Public v3.1 EXT001 Encoding.[18]

0-5	6	7	8-31	Description
PO	0	0	0000	new-suffix `RESERVED1`
PO	0	0	!zero	new-suffix, scalar (SVP64Single), or `RESERVED3`
PO	1	0	0000	new scalar-only word, or `RESERVED2`
PO	1	0	!zero	old-suffix, scalar (SVP64Single), or `RESERVED4`
PO	0	1	nnnn	new-suffix, vector (SVP64)
PO	1	1	nnnn	old-suffix, vector (SVP64)

- **PO** - Primary Opcode. Likely candidates: EXT005, EXT009
- **bit 6** - specifies whether the suffix is old (EXT000-EXT063) or new (EXTn00-EXTn63, n greater than 1)
- **bit 7** - defines whether the Suffix is Scalar-Prefixed or Vector-Prefixed (caveat: see bits 8-31)
- **old-suffix** - the EXT000 to EXT063 32-bit Major opcodes of Power ISA 3.0
- **new scalar-only** - a **new** Major Opcode area **exclusively** for Scalar-only instructions that shall **never** be Prefixed by SVP64 (RESERVED2 EXT300-EXT363)
- **new-suffix** - a **new** Major Opcode area (RESERVED1 EXT200-EXT263) that **may** be Prefixed by SVP64 and SVP64Single
- **0000** - all 24 bits bits 8-31 are zero (0x000000)
- **!zero** - bits 8-31 may be any value *other* than zero (0x000001-0xffffff)
- **nnnn** - bits 8-31 may be any value in the range 0x000000 to 0xffffff
- **SVP64Single** - a (TBD) *Scalar* Encoding that is near-identical to SVP64 except that it is equivalent to hard-coded VL=1 at all times. Predication is permitted, Element-width-overrides is permitted, Saturation is permitted. If not allocated within the scope of this RFC then these are requested to be `RESERVED` for a future Simple-V proposal.
- **SVP64** - a (well-defined, 2 years) DRAFT Proposal for a Vectorisation Augmentation of suffixes.

For the needs identified by Libre-SOC (75% of 2 POs), `RESERVED1` space *needs* allocation to new POs, `RESERVED2` does not.[19]

	Scalar (bit7=0,8-31=0000)	Scalar (bit7=0,8-31=!zero)	Vector (bit7=1)
new bit6=0	`RESERVED1`:{EXT200-263}	`RESERVED3`:SVP64-Single:{EXT200-263}	SVP64:{EXT200-263}
old bit6=1	`RESERVED2`:{EXT300-363}	`RESERVED4`:SVP64-Single:{EXT000-063}	SVP64:{EXT000-063}

- **`RESERVED2:{EXT300-363}`** (not strictly necessary to be added) is not and **cannot** ever be Vectorised or Augmented by Simple-V or any future Simple-V Scheme. it is a pure **Scalar-only** word-length PO Group. It may remain `RESERVED`.
- **`RESERVED1:{EXT200-263}`** is also a new set of 64 word-length Major Opcodes. These opcodes would be Simple-V-Augmentable unlike `EXT300-363` which may **never** be Simple-V-Augmented under any circumstances.
- **RESERVED3:`SVP64-Single:{EXT200-263}`** - Major opcodes 200-263 with Single-Augmentation, providing a one-bit predicate mask, element-width overrides on source and destination, and the option to extend the Scalar Register numbering (r0-32 extends to r0-127). **Placing of alternative instruction encodings other than those exactly defined in EXT200-263 is prohibited**.
- **RESERVED4:`SVP64-Single:{EXT000-063}`** - Major opcodes 000-063 with Single-Augmentation, just like SVP64-Single on EXT200-263, these are in effect Single-Augmented-Prefixed variants of the v3.0 32-bit Power ISA. Alternative instruction encodings other than the exact same 32-bit word from EXT000-EXT063 are likewise prohibited.
- **`SVP64:{EXT000-063}`** and **`SVP64:{EXT200-263}`** - Full Vectorisation of EXT000-063 and EXT200-263 respectively, these Prefixed instructions are likewise prohibited from being a different encoding from their 32-bit scalar versions.

Limitations of this scheme is that new 32-bit Scalar operations have to have a 32-bit "prefix pattern" in front of them. If commonly-used this could increase binary size. Thus the Encodings EXT300-363 and EXT200-263 should only be allocated for less-popular operations. However the scheme does have the strong advantage of *tripling* the available number of Major Opcodes in the Power ISA, caveat being that care on allocation is needed because EXT200-EXT263 may be SVP64-Augmented whilst EXT300-EXT363 may **not**. The issues of allocation for bitmanip etc. from Libre-SOC is therefore overwhelmingly made moot. The only downside is that there is no `SVP64-Reserved` which will have to be achieved with SPRs (PCR or MSR).

Most importantly what this scheme does not do is provide large areas for other (non-Vectoriseable) RFCs.

[18] Recall that EXT100 to EXT163 is for Public v3.1 64-bit-augmented Operations prefixed by EXT001, for which, from Section 1.6.3, bit 6 is set to 1. This concept is where the above scheme originated. Section 1.6.3 uses the term "defined word" to refer to pre-existing EXT000-EXT063 32-bit instructions so prefixed to create the new numbering EXT100-EXT163, respectively

[19] reminder that this proposal only needs 75% of two POs for Scalar instructions. The rest of EXT200-263 is for general use.

Potential Opcode allocation solution (2)

One of the risks of the bit 6/7 scheme above is that there is no room to share PO9 (EXT009) with other potential uses. A workaround for that is as follows:

- EXT009, like EXT001 of Public v3.1, is **defined** as a 64-bit encoding. This makes Multi-Issue Length-identification trivial.
- bit 6 if 0b1 is 100% for Simple-V augmentation of (Public v3.1 1.6.3) "Defined Words" (aka EXT000-063), with the exception of 0x26000000 as a Prefix, which is a new RESERVED encoding.
- when bit 6 is 0b0 and bits 32-33 are 0b11 are **defined** as also allocated to Simple-V
- all other patterns are `RESERVED` for other non-Vectoriseable purposes (just over 37.5%).

0-5	6	7	8-31	32:33	Description
PO9?	0	0	!zero	00-10	RESERVED (other)
PO9?	0	1	xxxx	00-10	RESERVED (other)
PO9?	x	0	0000	xx	RESERVED (other)
PO9?	0	0	!zero	11	SVP64 (current and future)
PO9?	0	1	xxxx	11	SVP64 (current and future)
PO9?	1	0	!zero	xx	SVP64 (current and future)
PO9?	1	1	xxxx	xx	SVP64 (current and future)

This ensures that any potential for future conflict over uses of the EXT009 space, jeapordising Simple-V in the process, are avoided, yet leaves huge areas (just over 37.5% of the 64-bit space) for other (non-Vectoriseable) uses.

These areas thus need to be Allocated (SVP64 and Scalar EXT248-263):

0-5	6	7	8-31	32-3	Description
PO	0	0	!zero	0b11	SVP64Single:EXT248-263, or `RESERVED3`
PO	0	0	0000	0b11	Scalar EXT248-263
PO	0	1	nnnn	0b11	SVP64:EXT248-263
PO	1	0	!zero	nn	SVP64Single:EXT000-063 or `RESERVED4`
PO	1	1	nnnn	nn	SVP64:EXT000-063

and reserved areas, QTY 1of 32-bit, and QTY 3of 55-bit, are:

0-5	6	7	8-31	32-3	Description
PO9?	1	0	0000	xx	`RESERVED1` or EXT300-363 (32-bit)
PO9?	0	x	xxxx	0b00	`RESERVED2` or EXT200-216 (55-bit)
PO9?	0	x	xxxx	0b01	`RESERVED2` or EXT216-231 (55-bit)
PO9?	0	x	xxxx	0b10	`RESERVED2` or EXT232-247 (55-bit)

- SVP64Single (`RESERVED3/4`) is *planned* for a future RFC (but needs reserving as part of this RFC)
- `RESERVED1/2` is available for new general-purpose (non-Vectoriseable) 32-bit encodings (other RFCs)
- EXT248-263 is for "new" instructions which **must** be granted corresponding space in SVP64.
- Anything Vectorised-EXT000-063 is **automatically** being requested as 100% Reserved for every single "Defined Word" (Public v3.1 1.6.3 definition). Vectorised-EXT001 or EXT009 is defined as illegal.
- Any **future** instruction added to EXT000-063 likewise, must **automatically** be assigned corresponding reservations in the SVP64:EXT000-063 and SVP64Single:EXT000-063 area, regardless of whether the instruction is Vectoriseable or not.

Bit-allocation Summary:

- EXT3nn and other areas provide space for up to QTY 4of non-Vectoriseable EXTn00-EXTn47 ranges.
- QTY 3of 55-bit spaces also exist for future use (longer by 3 bits than opcodes allocated in EXT001)
- Simple-V EXT2nn is restricted to range EXT248-263
- non-Simple-V (non-Vectoriseable) EXT2nn (if ever requested in any future RFC) is restricted to range EXT200-247
- Simple-V EXT0nn takes up 50% of PO9 for this and future Simple-V RFCs

This however potentially puts SVP64 under pressure (in 5-10 years). Ideas being discussed already include adding LD/ST-with-Shift and variant Shift-Immediate operations that require large quantity of Primary Opcodes. To ensure that there is room in future, it may be better to allocate 25% to `RESERVED`:

0-5	6	7	8-31	32	Description
PO9?	1	0	0000	x	EXT300-363 or `RESERVED1` (32-bit)
PO9?	0	x	xxxx	0b0	EXT200-232 or `RESERVED2` (56-bit)
PO9?	0	x	xxxx	0b1	EXT232-263 and SVP64(/V/S)

The clear separation between Simple-V and non-Simple-V stops conflict in future RFCs, both of which get plenty of space. EXT000-063 pressure is reduced in both Vectoriseable and non-Vectoriseable, and the 100+ Vectoriseable Scalar operations identified by Libre-SOC may safely be proposed and each evaluated on their merits.

EXT000-EXT063

These are Scalar word-encodings. Often termed "v3.0 Scalar" in this document Power ISA v3.1 Section 1.6.3 Book I calls it a "defined word".

0-5	6-31
PO	EXT000-063 "Defined word"

SVP64Single:{EXT000-063} bit6=old bit7=scalar

This encoding, identical to SVP64Single:{EXT248-263}, introduces SVP64Single Augmentation of Scalar "defined words". All meanings must be identical to EXT000-063, and is is likewise prohibited to add an instruction in this area without also adding the exact same (non-Augmented) instruction in EXT000-063 with the exact same Scalar word. Bits 32-37 0b00000 to 0b11111 represent EXT000-063 respectively. Augmenting EXT001 or EXT009 is prohibited.

0-5	6	7	8-31	32-63
PO (9)?	1	0	!zero	SVP64Single:{EXT000-063}

SVP64:{EXT000-063} bit6=old bit7=vector

This encoding is identical to **SVP64:{EXT248-263}** except it is the Vectorisation of existing v3.0/3.1 Scalar-words, EXT000-063. All the same rules apply with the addition that Vectorisation of EXT001 or EXT009 is prohibited.

0-5	6	7	8-31	32-63
PO (9)?	1	1	nnnn	SVP64:{EXT000-063}

{EXT248-263} bit6=new bit7=scalar

This encoding represents the opportunity to introduce EXT248-263. It is a Scalar-word encoding, and does not require implementing SVP64 or SVP64-Single, but does require the Vector-space to be allocated. PO2 is in the range 0b11000 to 0b111111 to represent EXT248-263 respectively.

0-5	6	7	8-31	32-3	34-37	38-63
PO (9)?	0	0	0000	0b11	PO2[2:5]	{EXT248-263}

SVP64Single:{EXT248-263} bit6=new bit7=scalar

This encoding, which is effectively "implicit VL=1" and comprising (from bits 8-31 being non-zero) *at least some* form of Augmentation, it represents the opportunity to Augment EXT248-263 with the SVP64Single capabilities. Must be allocated under Scalar *and* SVP64 simultaneously.

0-5	6	7	8-31	32-3	34-37	38-63
PO (9)?	0	0	!zero	0b11	PO2[2:5]	SVP64Single:{EXT248-263}

SVP64:{EXT248-263} bit6=new bit7=vector

This encoding, which permits VL to be dynamic (settable from GPR or CTR) is the Vectorisation of EXT248-263. Instructions may not be placed in this category without also being implemented as pure Scalar *and* SVP64Single. Unlike SVP64Single however, there is **no reserved encoding** (bits 8-24 zero). VL=1 may occur dynamically at runtime, even when bits 8-31 are zero.

0-5	6	7	8-31	32-3	34-37	38-63
PO (9)?	0	1	nnnn	0b11	PO2[2:5]	SVP64:{EXT248-263}

RESERVED2 / EXT300-363 bit6=old bit7=scalar

This is entirely at the discretion of the ISA WG. Libre-SOC is *not* proposing the addition of EXT300-363: it is merely a possibility for future. The reason the space is not needed is because this is within the realm of Scalar-extended (SVP64Single), and with the 24-bit prefix area being all-zero (bits 8-31) this is defined as "having no augmentation" (in the Simple-V Specification it is termed `Scalar Identity Behaviour`). This in turn makes this prefix a *degenerate duplicate* so may be allocated for other purposes.

0-5	6	7	8-31	32-63
PO (9)?	1	0	0000	EXT300-363 or `RESERVED1`

Example Legal Encodings and RESERVED spaces

This section illustrates what is legal encoding, what is not, and why the 4 spaces should be `RESERVED` even if not allocated as part of this RFC.

legal, scalar and vector

width	assembler	prefix?	suffix	description
32bit	fishmv	none	0x12345678	scalar EXT0nn
64bit	ss.fishmv	0x26!zero	0x12345678	scalar SVP64Single:EXT0nn
64bit	sv.fishmv	0x27nnnnnn	0x12345678	vector SVP64:EXT0nn

OR:

width	assembler	prefix?	suffix	description
64bit	fishmv	0x24000000	0x12345678	scalar EXT2nn
64bit	ss.fishmv	0x24!zero	0x12345678	scalar SVP64Single:EXT2nn
64bit	sv.fishmv	0x25nnnnnn	0x12345678	vector SVP64:EXT2nn

Here the encodings are the same, 0x12345678 means the same thing in all cases. Anything other than this risks either damage (truncation of capabilities of Simple-V) or far greater complexity in the Decode Phase.

This drives the compromise proposal (above) to reserve certain EXT2nn POs right across the board (in the Scalar Suffix side, irrespective of Prefix), some allocated to Simple-V, some not.

illegal due to missing

width	assembler	prefix?	suffix	description
32bit	fishmv	none	0x12345678	scalar EXT0nn
64bit	ss.fishmv	0x26!zero	0x12345678	scalar SVP64Single:EXT0nn
64bit	unallocated	0x27nnnnnn	0x12345678	vector SVP64:EXT0nn

This is illegal because the instruction is possible to Vectorise, therefore it should be **defined** as Vectoriseable.

illegal due to unvectoriseable

width	assembler	prefix?	suffix	description
32bit	mtmsr	none	0x12345678	scalar EXT0nn
64bit	ss.mtmsr	0x26!zero	0x12345678	scalar SVP64Single:EXT0nn
64bit	sv.mtmsr	0x27nnnnnn	0x12345678	vector SVP64:EXT0nn

This is illegal because the instruction `mtmsr` is not possible to Vectorise, at all. This does **not** convey an opportunity to allocate the space to an alternative instruction.

illegal unvectoriseable in EXT2nn

width	assembler	prefix?	suffix	description
64bit	mtmsr2	0x24000000	0x12345678	scalar EXT2nn
64bit	ss.mtmsr2	0x24!zero	0x12345678	scalar SVP64Single:EXT2nn
64bit	sv.mtmsr2	0x25nnnnnn	0x12345678	vector SVP64:EXT2nn

For a given hypothetical `mtmsr2` which is inherently Unvectoriseable whilst it may be put into the scalar EXT2nn space it may **not** be allocated in the Vector space. As with Unvectoriseable EXT0nn opcodes this does not convey the right to use the 0x24/0x26 space for alternative opcodes. This hypothetical Unvectoriseable operation would be better off being allocated as EXT001 Prefixed, EXT000-063, or hypothetically in EXT300-363.

ILLEGAL: dual allocation

width	assembler	prefix?	suffix	description
32bit	fredmv	none	0x12345678	scalar EXT0nn
64bit	ss.fredmv	0x26!zero	0x12345678	scalar SVP64Single:EXT0nn
64bit	sv.fishmv	0x27nnnnnn	0x12345678	vector SVP64:EXT0nn

the use of 0x12345678 for fredmv in scalar but fishmv in Vector is illegal. the suffix in both 64-bit locations must be allocated to a Vectoriseable EXT000-063 “Defined Word” (Public v3.1 Section 1.6.3 definition) or not at all.

illegal unallocated scalar EXT0nn or EXT2nn:

width	assembler	prefix?	suffix	description
32bit	unallocated	none	0x12345678	scalar EXT0nn
64bit	ss.fredmv	0x26!zero	0x12345678	scalar SVP64Single:EXT0nn
64bit	sv.fishmv	0x27nnnnnn	0x12345678	vector SVP64:EXT0nn

and:

width	assembler	prefix?	suffix	description
64bit	unallocated	0x24000000	0x12345678	scalar EXT2nn
64bit	ss.fishmv	0x24!zero	0x12345678	scalar SVP64Single:EXT2nn
64bit	sv.fishmv	0x25nnnnnn	0x12345678	vector SVP64:EXT2nn

Both of these Simple-V operations are illegally-allocated. The fact that there does not exist a scalar "Defined Word" (even for EXT200-263) - the unallocated block - means that the instruction may **not** be allocated in the Simple-V space.

illegal attempt to put Scalar EXT004 into Vector EXT2nn

width	assembler	prefix?	suffix	description
32bit	unallocated	none	0x10345678	scalar EXT0nn
64bit	ss.fishmv	0x24!zero	0x10345678	scalar SVP64Single:EXT2nn
64bit	sv.fishmv	0x25nnnnnn	0x10345678	vector SVP64:EXT2nn

This is an illegal attempt to place an EXT004 "Defined Word" (Public v3.1 Section 1.6.3) into the EXT2nn Vector space. This is not just illegal it is not even possible to achieve. If attempted, by dropping EXT004 into bits 32-37, the top two MSBs are actually *zero*, and the Vector EXT2nn space is only legal for Primary Opcodes in the range 248-263, where the top two MSBs are 0b11. Thus this faulty attempt actually falls unintentionally into `RESERVED` "Non-Vectoriseable" Encoding space.

illegal attempt to put Scalar EXT001 into Vector space

width	assembler	prefix?	suffix	description
64bit	EXT001	0x04nnnnnn	any	scalar EXT001
96bit	sv.EXT001	0x24!zero	EXT001	scalar SVP64Single:EXT001
96bit	sv.EXT001	0x25nnnnnn	EXT001	vector SVP64:EXT001

This becomes in effect an effort to define 96-bit instructions, which are illegal due to cost at the Decode Phase (Variable-Length Encoding). Likewise attempting to embed EXT009 (chained) is also illegal. The implications are clear unfortunately that all 64-bit EXT001 Scalar instructions are Unvectoriseable.

Use cases

In the following examples the programs are fully executable under the Libre-SOC Simple-V-augmented Power ISA Simulator. Reproducible (scripted) Installation instructions: https://libre-soc.org/HDL_workflow/devscripts/

LD/ST-Multi

Context-switching saving and restoring of registers on the stack often requires explicit loop-unrolling to achieve effectively. In SVP64 it is possible to use a Predicate Mask to "compact" or "expand" a swathe of desired registers, dynamically. Known as "VCOMPRESS" and "VEXPAND", runtime-configurable LD/ST-Multi is achievable with 2 instructions.

```
# load 64 registers off the stack, in-order, skipping unneeded ones
# by using CR0-CR63's "EQ" bits to select only those needed.
setvli 64
sv.ld/sm=EQ *rt,0(ra)
```

Twin-Predication, re-entrant

This example demonstrates two key concepts: firstly Twin-Predication (separate source predicate mask from destination predicate mask) and that sufficient state is stored within the Vector Context SPR, SVSTATE, for full re-entrancy on a Context Switch or function call *even if in the middle of executing a loop*. Also demonstrates that it is permissible for a programmer to write **directly** to the SVSTATE SPR, and still expect Deterministic Behaviour. It's not exactly recommended (performance may be impacted by direct SVSTATE access), but it is not prohibited either.

```
292     # checks that we are able to resume in the middle of a VL loop,
293     # after an interrupt, or after the user has updated src/dst step
294     # let's assume the user has prepared src/dst step before running this
295     # vector instruction
296     # test_intpred_reentrant
297         #   reg num        0 1 2 3 4 5 6 7 8 9 10 11 12
298         #   srcstep=1                          v
299         #   src r3=0b0101                   Y  N  Y  N
300         #                                   :     |
301         #                                + - - +     |
302         #                                :   +-------+
303         #                                :   |
304         #   dest ~r3=0b1010           N Y N Y
305         #   dststep=2                    ^
306
307         sv.extsb/sm=r3/dm=~r3 *5, *9
```

https://git.libre-soc.org/?p=openpower-isa.git;a=blob;f=src/openpower/decoder/isa/test_caller_svp64_predication.py;hb=HEAD

Matrix Multiply

Matrix Multiply of any size (non-power-2) up to a total of 127 operations is achievable with only three instructions. Normally in any other SIMD ISA at least one source requires Transposition and often massive rolling repetition of data is required. These 3 instructions may be used as the "inner triple-loop kernel" of the usual 6-loop Massive Matrix Multiply.

```
28     # test_sv_remap1   5x4 by 4x3 matrix multiply
29                       svshape 5, 4, 3, 0, 0
30                       svremap 31, 1, 2, 3, 0, 0, 0
31                       sv.fmadds *0, *8, *16, *0
```

https://git.libre-soc.org/?p=openpower-isa.git;a=blob;f=src/openpower/decoder/isa/test_caller_svp64_matrix.py;hb=HEAD

Parallel Reduction

Parallel (Horizontal) Reduction is often deeply problematic in SIMD and Vector ISAs. Parallel Reduction is Fully Deterministic in Simple-V and thus may even usefully be deployed on non-associative and non-commutative operations.

```
75     # test_sv_remap2
76                       svshape 7, 0, 0, 7, 0
77                       svremap 31, 1, 0, 0, 0, 0, 0 # different order
78                       sv.subf *0, *8, *16
```

https://git.libre-soc.org/?p=openpower-isa.git;a=blob;f=src/openpower/decoder/isa/test_caller_svp64_parallel_reduce.py;hb=HEAD

DCT

DCT has dozens of uses in Audio-Visual processing and CODECs. A full 8-wide in-place triple-loop Inverse DCT may be achieved in 8 instructions. Expanding this to 16-wide is a matter of setting `svshape 16` **and the same instructions used**. Lee Composition may be deployed to construct non-power-two DCTs. The cosine table may be computed (once) with 18 Vector instructions (one of them `fcos`)

```
1014     # test_sv_remap_fpmadds_ldbrev_idct_8_mode_4
1015                       # LOAD bit-reversed with half-swap
1016                       svshape 8, 1, 1, 14, 0
1017                       svremap 1, 0, 0, 0, 0, 0, 0
1018                       sv.lfs/els *0, 4(1)
1019                       # Outer butterfly, iterative sum
1020                       svremap 31, 0, 1, 2, 1, 0, 1
1021                       svshape 8, 1, 1, 11, 0
1022                       sv.fadds *0, *0, *0
1023                       # Inner butterfly, twin +/- MUL-ADD-SUB
1024                       svshape 8, 1, 1, 10, 0
1025                       sv.ffmadds *0, *0, *0, *8
```

https://git.libre-soc.org/?p=openpower-isa.git;a=blob;f=src/openpower/decoder/isa/test_caller_svp64_dct.py;hb=HEAD

3D GPU style "Branch Conditional"

(*Note: Specification is ready, Simulator still under development of full specification capabilities*) This example demonstrates a 2-long Vector Branch-Conditional only succeeding if *all* elements in the Vector are successful. This avoids the need for additional instructions that would need to perform a Parallel Reduction of a Vector of Condition Register tests down to a single value, on which a Scalar Branch-Conditional could then be performed. Full Rationale at https://libre-soc.org/openpower/sv/branches/

```
80   # test_sv_branch_cond_all
81       for i in [7, 8, 9]:
83               addi 1, 0, i+1        # set r1 to i
84               addi 2, 0, i          # set r2 to i
85               cmpi cr0, 1, 1, 8     # compare r1 with 8 and store to cr0
86               cmpi cr1, 1, 2, 8     # compare r2 with 8 and store to cr1
87               sv.bc/all 12, *1, 0xc # bgt 0xc - branch if BOTH
88                                     # r1 AND r2 greater 8 to the nop below
89               addi 3, 0, 0x1234,    # if tests fail this shouldn't execute
90               or 0, 0, 0            # branch target
```

https://git.libre-soc.org/?p=openpower-isa.git;a=blob;f=src/openpower/decoder/isa/test_caller_svp64_bc.py;hb=HEAD

Big-Integer Math

Remarkably, `sv.adde` is inherently a big-integer Vector Add, using `CA` chaining between **Scalar** operations. Using Vector LD/ST and recalling that the first and last `CA` may be chained in and out of an entire **Vector**, unlimited-length arithmetic is possible.

```
26     # test_sv_bigint_add
32
33         r3/r2: 0x0000_0000_0000_0001 0xffff_ffff_ffff_ffff +
34         r5/r4: 0x8000_0000_0000_0000 0x0000_0000_0000_0001 =
35         r1/r0: 0x8000_0000_0000_0002 0x0000_0000_0000_0000
36
37                           sv.adde *0, *2, *4
```

A 128/64-bit shift may be used as a Vector shift by a Scalar amount, by merging two 64-bit consecutive registers in succession.

```
62     # test_sv_bigint_scalar_shiftright(self):
64
65     r3                    r2                    r1                        r4
66     0x0000_0000_0000_0002 0x8000_8000_8000_8001 0xffff_ffff_ffff_ffff >> 4
67     0x0000_0000_0000_0002 0x2800_0800_0800_0800 0x1fff_ffff_ffff_ffff
68
69                           sv.dsrd *0,*1,4,1
```

Additional 128/64 Mul and Div/Mod instructions may similarly be exploited to perform roll-over in arbitrary-length arithmetic: effectively they use one of the two 64-bit output registers as a form of "64-bit Carry In-Out".

All of these big-integer instructions are Scalar instructions standing on their own merit and may be utilised even in a Scalar environment to improve performance. When used with Simple-V they may also be used to improve performance and also greatly simplify unlimited-length biginteger algorithms.

https://git.libre-soc.org/?p=openpower-isa.git;a=blob;f=src/openpower/decoder/isa/test_caller_svp64_bigint.py;hb=HEAD

[[!tag opf_rfc]]