Preamble

Last modified date: 2023-05-07

This document is an auto-generated version of the Draft SVP64 Specification available at

`https://libre-soc.org/openpower/sv`

for which the source code is available at

`https://git.libre-soc.org/?p=libreriscv.git;a=tree;f=openpower;hb=HEAD`

This PDF may be created with "make pdf" from the following file:

`https://git.libre-soc.org/?p=libreriscv.git;a=blob;f=openpower/Makefile;hb=HEAD`

by executing the following commands:

```
git clone https://git.libre-soc.org/git/libreriscv.git libresoc
cd libresoc/libresoc/openpower
make pdf
```

Simple-V Cray-style Vectors have been developed by the Libre-SOC Team, sponsored by the NLnet Foundation and NGI POINTER under EU Grants 871528 and 957073.

Simple-V is in DRAFT Status and will be submitted publicly (non-confidentially) through the OPF ISA WG "External Submissions" Process. Funding from NLnet, through their Privacy and Enhanced Trust Programme, requires full transparency.

As this document is under continuous rapid revision please check frequently at:

`https://ftp.libre-soc.org/simple_v_spec.pdf`

Contacts

For questions, comments, and clarification, please contact the following:

- Libre-SOC ISA Dev Mailing List - libre-soc-isa@lists.libre-soc.org
- Luke Kenneth Casson Leighton - Libre-SOC team lead and Red Semiconductor Ltd Director - lkcl@lkcl.net
- David Calderwood - Red Semiconductor Ltd Director - djac@calderwoodhan.com
- Toshaan Bharvani - OpenPOWER Foundation Technical Chair, VanTosh Director - toshaan@vantosh.com
- Konstantinos Margaritis - Engineer and Founder of VectorCamp, writing optimised assembler for a number of SIMD/Vector ISAs - konstantinos@vectorcamp.gr
- Dmitry Selyutin - Libre-SOC engineer, working on binutils SVP64 assembler - ghostmansd@gmail.com
- Jacob Lifshay - Libre-SOC engineer, CPU arch and verification - programmerjake@gmail.com
- Cesar Strauss - Libre-SOC engineer, CPU arch and verification - cestrauss@gmail.com
- Andrey Miroshnikov - Libre-SOC engineer, assisting with documentation - andrey@technepisteme.xyz

Executive Summary

Simple-V is a Scalable Vector ISA Extension **specifically tailored** for the uniquely powerful capabilities of the Power ISA. **SVP64** is the instruction set format. We invented Simple-V to be simple because we don't like complicated.

Simple-V does not modify harm or corrupt the existing Power ISA and does not interfere with an existing system. It needs only a small allocation of opcodes (five) to implement, whereas any other Vector implementation would require an intrusive fundamental overhaul of the Power ISA.

It is extremely important to think of Simple-V as a 2-Dimensional ISA: instructions vertical and registers horizontal otherwise it will be difficult to grasp and appreciate its RISC simplicity. Like all Cray-Style Scalable Vector ISAs, Simple-V binaries remain ubiquitous, the ISA uniform. The Compliancy Levels offer a means to scale up in complexity to meet the target application requirements.

- GPUs may implement massive-wide SIMD back-ends, focussing on number-crunching.
- Existing Multi-issue Superscalar implementations may insert Simple-V between decode and issue with minimal disruption.
- Single-issue in-order implementations are very straightforward.
- Inter-core communication (OpenCAPI, other) may still be utilised because SVP64 fundamentally remains and respects the Power ISA.

All implementations regardless of back-end capability may execute the exact same binaries *(this is known to be extremely important to the Power ISA ecosystem). If not done as carefully as SVP64, the addition of any other Scalable Vector Extension would require a significant number of opcodes, putting further pressure on Major Opcode space which was never designed with Scalable Vectors in mind. Contrast with RISC-V which was designed over a 7 year period with Cray-style Vectors right from the start.*

Even with this amount of time spent, SVP64 exceeds the capability of RVV. RISC-V could have been significantly enhanced if Simple V had been applied to it: this possibility was investigated very early but the decision was made to go with Power ISA instead.

Therefore it is crucial to note that Simple-V is **not RISC-V and is not RISC-V Vectors**. NEC SX Aurora, RVV, Simple-V and MRISC32 are all based on Cray-style Scalable Vectors of 50 years ago, hence the similarity, the provision of a `setvl` instruction, and why they are each called "Scalable" Vectors, because it is the `setvl` instruction that presents the **programmer** with explicit control over Vector length.

VSX and NEON are PackedSIMD, and AVX-512 and ARM SVE2 are Predicated SIMD ISAs. **None of them provide Scalability to the Programmer**. SVE2 is **Silicon** Scalable, not **Programmer** Scalable: the distinction is profoundly important (already causing problems). For Predicated SIMD, Programmers must emulate Cray-style scaling through explicit predicate masking, which increases instruction count in hot-loops.

description, URL
Unit tests and simulator for Power ISA v3.0 and SVP64
https://git.libre-soc.org/?p=openpower-isa.git;a=tree;f=src/openpower/decoder/isa;hb=HEAD
pypowersim tutorial
https://libre-soc.org/docs/pypowersim/
several thousand more ISA unit tests
https://git.libre-soc.org/?p=openpower-isa.git;a=tree;f=src/openpower/test;hb=HEAD
demo, showing 4.5x reduction in program size for MP3 decode, greatly simplifies assembler development
https://git.libre-soc.org/?p=openpower-isa.git;a=tree;f=media/audio/mp3;hb=HEAD
binutils support for DRAFT SVP64 (now upstream)
https://git.libre-soc.org/?p=binutils-gdb.git;a=shortlog;h=refs/heads/svp64-ng

ISA Comparison Table to DRAFT SVP64 - discussion and research at https://bugs.libre-soc.org/show_bug.cgi?id=893

ISA name	No opcodes	No intrinsics	Taxonomy / Class	Binary Compat	setvl scalable	Pred. Masks	Twin Pred	Vector regs	128-bit ops	Big int	LDST F/First	Data-dep F-first	Pred Result	HW Matrix	DCT FFT
SVP64	6 [1]	see [2]	Scalable [3]	yes	yes	yes	yes [4]	no [5]	see [6]	yes[7]	yes [8]	yes [9]	yes [10]	yes [11]	yes[12]
VSX	700+	700?[13]	PackedSIMD	yes	no	no	no	yes [14]	yes	no	no	no	no	yes [15]	no
NEON	~250[16]	7088 [17]	PackedSIMD	yes	no	no	no	yes	see [18]	no	no	no	no	no	no
SVE2	~1000[19]	6040 [20]	PredSIMD[21]	NO [22]	no [21]	yes	no	yes	see [18]	no	yes [8]	no	no	yes [23]	no
AVX512[24]	~1000s[25]	7256[26]	PredSIMD	yes	no	yes	no	yes	see[18]	no	no	no	no	yes[27]	no
RVV [28]	~190[29]	~25000[30]	Scalable[31]	NO [22]	yes	yes	no	yes	yes [32]	no	yes	no	no	no	no
AuroraSX[33]	~200[34]	unknown[35]	Scalable[36]	yes	yes	yes	no	yes	no	no	no	no	no	?	no
66000[37]	~200	unknown	AutoVec[37]	yes	see [37]	see[37]	no	see [37]	no	yes[38]	see [37]	no	no	no	no

[1] plus EXT001 24-bit prefixing using 25% of EXT001 space. See {SVP64 Chapter}

[2] If treated as a 1-Dimensional ISA, and designed badly, the 24-bit Prefix expands 200+ scalar instructions to well over a million intrinsics (N~=10^4 **times** M~=10^2). If treated as a 2-Dimensional ISA and designed well, there are far less. N prefix intrinsics **plus** M scalar instruction intrinsics, where N is likely to be of the order of 10^2 and M of the order of 10^2.

[3] A 2-Dimensional Scalable Vector ISA **specifically designed for the Power ISA** with both Horizontal-First and Vertical-First Modes. See {Vector ISA Comparison}

[4] on specific operations. See {SVP64 Augmentation Table} for full list. Key: 2P - Twin Predication, 1P - Single-Predicate

[5] SVP64 provides a Vector concept on top of the **Scalar** GPR, FPR and CR Fields, extended to 128 entries.

[6] SVP64 Vectorises Scalar ops. It is up to the **implementor** to choose (**optionally**) whether to apply SVP64 to e.g. VSX Quad-Precision (128-bit) instructions, to create 128-bit Vector ops.

[7] big-integer add is just `sv.adde`. For optimal performance Bigint Mul and divide first require addition of two scalar operations (in turn, naturally Vectorised by SVP64). See {Big Integer Analysis}

[8] LD/ST Fault-First: see {SVP64 Appendix} and ARM SVE Fault-First

[9] Data-dependent Fail-First: Based on LD/ST Fail-first, extended to data. Truncates VL based on failing Rc=1 test. Similar to Z80 CPIR. See {SVP64 Appendix}

[10] Predicate-result effectively turns any standard op into a type of "cmp". See {SVP64 Appendix}

[11] Any non-power-of-two Matrices up to 127 FMACs or other FMA-style op including Ternary Logical, full triple-loop Schedule. See {REMAP subsystem}

[12] DCT (Lee) and FFT Full Triple-loops supported, RADIX2-only. Normally only found in VLIW DSPs (TI MSP320, Qualcom Hexagon). See {REMAP subsystem}

[13] Altivec gcc intrinsics, contains links to additional VSX intrinsics for ISA 2.05/6/7, 3.0 and 3.1

[14] VSX's Vector Registers are mis-named: they are 100% PackedSIMD. AVX-512 is not a Vector ISA either. See Flynn's Taxonomy

[15] Power ISA v3.1 contains "Matrix Multiply Assist" (MMA) which due to PackedSIMD is restricted to RADIX2 and requires inline assembler loop-unrolling for non-power-of-two Matrix dimensions

[16] difficult to ascertain, see NEON/VFP. Critically depends on ARM Scalar instructions

[17] NEON 32-bit 2754 intrinsics, NEON 64-bit 4334 intrinsics.

[18] Although registers may be 128-bit in NEON, SVE2, and AVX, unlike VSX there are very few (or no) actual arithmetic 128-bit operations. Only RVV and SVP64 have the possibility of 128-bit ops

[19] difficult to exactly ascertain, see ARM Architecture Reference Manual Supplement, DDI 0584. Critically depends on ARM Scalar instructions.

[20] SVE: 4140 intrinsics, SVE2 1900 intrinsics

[21] ARM states that the Scalability is a Silicon-partner choice. Scalability in the ISA is **not available to the programmer**: there is no `setvl` instruction in SVE2, which is already causing assembler programmer difficulties. quote **"you may be stuck with only using the bottom 128 bits of the vector, or need to code specifically for each width"**

[22] "Silicon-Partner" Scaling achieved through allowing same instruction to act on different regfile size and bitwidth. This catastrophically results in binary non-interoperability.

[23] Scalable Matrix Optional Extension outer-product instructions SMOPA which are power-2 based on Silicon-partner SIMD width. Non-power-2 not supported but zero-input masking is.

[24] AVX512 Wikipedia, Lifecycle of an instruction set including full slides

[25] difficult to exactly ascertain, contains subsets. Critically depends on ISA support from earlier x86 ISA subsets (several more thousand instructions). See SIMD ISA listing

[26] Count includes SSE, SSE2, AVX, AVX2 and all AVX512 variants

[27] Advanced matrix Extensions supports BF16 and INT8 only. Separate regfile, power-of-two "tiles". Not general-purpose at all.

[28] RVV Spec

[29] RISC-V Vectors are not stand-alone, i.e. like SVE2 and AVX-512 are critically dependent on the Scalar ISA (an additional ~96 instructions for the Scalar RV64GC set, needed for Linux).

[30] RVV intrinsics listing page is 25,000 lines long.

[31] Like the original Cray RVV is a truly scalable Vector ISA (Cray setvl instruction). However, like SVE2, the Maximum Vector length is a Silicon-partner choice, which creates similar limitations that SVP64 does not have. The RISC-V Founders strongly discourage efforts by programmers to find out the Silicon's Maximum Vector Length, as an effort to steer programmers towards Silicon-independent assembler. **This requires all algorithms to contain a loop construct**. MAXVL in SVP64 is a Spec-hard-fixed quantity therefore loop constructs are not necessary 100% of the time.

[32] like SVP64 it is up to the hardware implementor (Silicon partner) to choose whether to support 128-bit elements.

[33] NEC SX Aurora is based on the original Cray Vectors

[34] Aurora ISA guide Appendix-3 11.1 p508

[35] Unknown. estimated to be of the order of length of RVV due to also being a Cray-style Scalable ISA, NEC maintains an LLVM hard fork

[36] Like the original Cray Vectors, the ISA Vector Length is independent of the underlying hardware, however Generation 1 has 256 elements per Vector register (3.2.4 p24, Aurora ISA guide)

[37] Mitch Alsup's MyISA 66000 is available on request. A powerful RISC ISA with a **Hardware-level auto-vectorisation** LOOP built-in as an extension named VVM. Classified as "Vertical-First".

[38] MyISA 66000 has a CARRY register up to 64-bit. Repeated application of FMA (esp. within Auto-Vectored LOOPS) automatically and inherently creates big-int operations with zero effort.

Part I

Scalable Vectors Primer

List of Acronyms

AVX-512 Intel Advanced Vector Extensions 512-bit

CPU Central Processing Unit

DCT Discrete Cosine Transform

DSP Digital Signal Processors

FFT Fast Fourier Transform

ISA Instruction Set Architecture

MMX Intel's first SIMD implementation

RVV RISC-V Vector extension

SIMD Single Instruction Multiple Data

SWAR SIMD Within A Register (see Flynn's Taxonomy)

SV (Scalable) Simple Vectorisation or Simple-V

VLIW Very Long Instruction Word

VSX 128-bit Packed SIMD Extension to the Power ISA

Summary

The proposed SV is a Scalable Vector Specification for a hardware for-loop **that ONLY uses scalar instructions**.

- The Power ISA v3.1 Spec is not altered. v3.1 Code-compatibility is guaranteed.
- Does not require sacrificing 32-bit Major Opcodes.
- Does not require adding duplicates of instructions (popcnt, popcntw, popcntd, vpopcntb, vpopcnth, vpopcntw, vpopcntd)
- Fully abstracted: does not create Micro-architectural dependencies (no fixed "Lane" size), one binary works across all existing *and future* implementations.
- Specifically designed to be easily implemented on top of an existing Micro-architecture (especially Superscalar Out-of-Order Multi-issue) without disruptive full architectural redesigns.
- Divided into Compliancy Levels to suit differing needs.
- At the highest Compliancy Level only requires five instructions (SVE2 requires appx 9,000. AVX-512 around 10,000. RVV around 300).
- Predication, often-requested, is added cleanly (without modifying the v3.1 Power ISA)
- In-registers arbitrary-sized Matrix Multiply is achieved in three instructions (without adding any v3.1 Power ISA instructions)
- Full DCT and FFT RADIX2 Triple-loops are achieved with dramatically reduced instruction count, and power consumption expected to greatly reduce. Normally found only in high-end VLIW DSP (TI MSP, Qualcomm Hexagon)
- Fail-First Load/Store allows Vectorised high performance strncpy to be implemented in around 14 instructions (hand-optimised VSX assembler is 240).
- Inner loop of MP3 implemented in under 100 instructions (gcc produces 450 for the same function on POWER9).

All areas investigated so far consistently showed reductions in executable size, which as outlined in [**?**] has an indirect reduction in power consumption due to less I-Cache/TLB pressure and also Issue remaining idle for long periods. Simple-V has been specifically and carefully crafted to respect the Power ISA's Supercomputing pedigree.

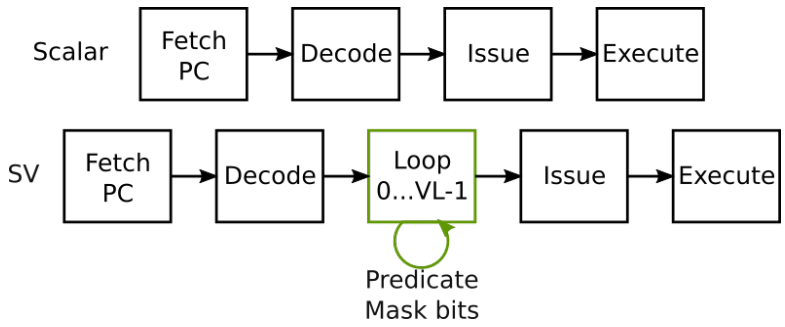

Figure 1: Showing how SV fits in between Decode and Issue

What is SIMD?

SIMD is a way of partitioning existing CPU registers of 64-bit length into smaller 8-, 16-, 32-bit pieces. [?][?] These partitions can then be operated on simultaneously, and the initial values and results being stored as entire 64-bit registers (SWAR). The SIMD instruction opcode includes the data width and the operation to perform.

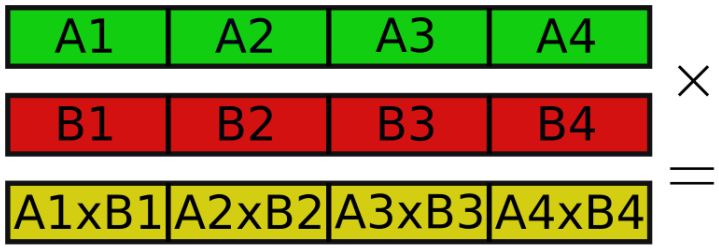

Figure 2: SIMD multiplication

This method can have a huge advantage for rapid processing of vector-type data (image/video, physics simulations, cryptography, etc.), [?], and thus on paper is very attractive compared to scalar-only instructions. *As long as the data width fits the workload, everything is fine.*

Shortfalls of SIMD

SIMD registers are of a fixed length and thus to achieve greater performance, CPU architects typically increase the width of registers (to 128-, 256-, 512-bit etc) for more partitions.

Additionally, binary compatibility is an important feature, and thus each doubling of SIMD registers also expands the instruction set. The number of instructions quickly balloons and this can be seen in for example IA-32 expanding from 80 to about 1400 instructions since the 1970s[?].

Five digit Opcode proliferation (10,000 instructions) is overwhelming. The following are just some of the reasons why SIMD is unsustainable as the number of instructions increase:

- Hardware design, ASIC routing etc.
- Compiler design
- Documentation of the ISA
- Manual coding and optimisation
- Time to support the platform
- Compilance Suite development and testing
- Protracted Variable-Length encoding (x86) severely compromises Multi-issue decoding

Scalable Vector Architectures

An older alternative exists to utilise data parallelism - vector architectures. Vector CPUs collect operands from the main memory, and store them in large, sequential vector registers.

A simple vector processor might operate on one element at a time, however as the element operations are usually independent, a processor could be made to compute all of the vector's elements simultaneously, taking advantage of multiple pipelines.

Typically, today's vector processors can execute two, four, or eight 64-bit elements per clock cycle. [?]. Vector ISAs are specifically designed to deal with (in hardware) fringe cases where an algorithm's element count is not

a multiple of the underlying hardware "Lane" width. The element data width is variable (8 to 64-bit just like in SIMD) but it is the *number* of elements being variable under control of a "setvl" instruction that specifically makes Vector ISAs "Scalable"

RVV supports a VL of up to 2^{16} or 65536 bits, which can fit 1024 64-bit words. [**?**]. The Cray-1 had 8 Vector Registers with up to 64 elements (64-bit each). An early Draft of RVV supported overlaying the Vector Registers onto the Floating Point registers, similar to MMX.

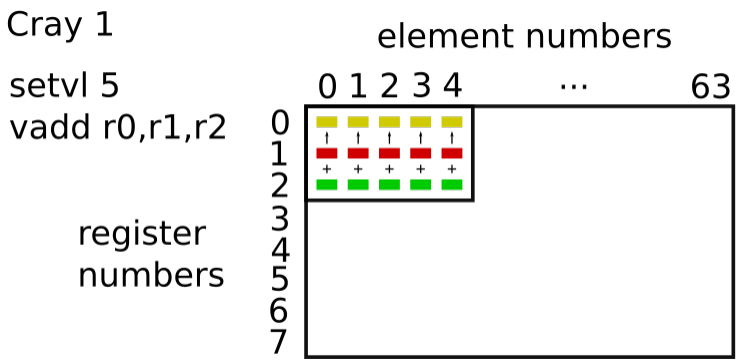

Figure 3: Cray Vector registers: 8 registers, 64 elements each

Simple-V's "Vector" Registers (a misnomer) are specifically designed to fit on top of the Scalar (GPR, FPR) register files, which are extended from the default of 32, to 128 entries in the high-end Compliancy Levels. This is a primary reason why Simple-V can be added on top of an existing Scalar ISA, and *in particular* why there is no need to add explicit Vector Registers or Vector instructions. The diagram below shows *conceptually* how a Vector's elements are sequentially and linearly mapped onto the *Scalar* register file:

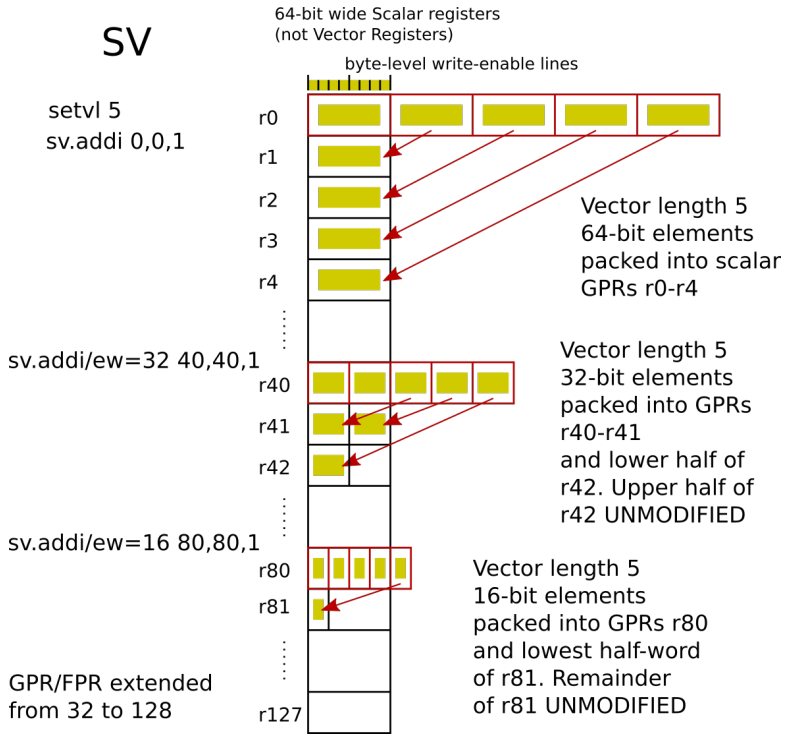

Figure 4: three instructions, same vector length, different element widths

Simple Vectorisation

SV is a Scalable Vector ISA designed for hybrid workloads (CPU, GPU, VPU, 3D). Includes features normally found only on Cray-style Supercomputers (Cray-1, NEC SX-Aurora) and GPUs. Keeps to a strict uniform RISC paradigm, leveraging a scalar ISA by using "Prefixing". **No dedicated vector opcodes exist in SV, at all**. SVP64 uses 25% of the Power ISA v3.1 64-bit Prefix space (EXT001) to create the SV Vectorisation Context for the 32-bit Scalar Suffix.

Main design principles

- Introduce by implementing on top of existing Power ISA
- Effectively a **hardware for-loop**, pauses main PC, issues multiple scalar operations
- Strictly preserves (leverages) underlying scalar execution dependencies as if the for-loop had been expanded into actual scalar instructions ("preserving Program Order")
- Augments existing instructions by adding "tags" - provides Vectorisation "context" rather than adding new opcodes.
- Does not modify or deviate from the underlying scalar Power ISA unless there's a significant performance boost or other advantage in the vector space
- Aimed at Supercomputing: avoids creating significant *sequential dependency hazards*, allowing **high performance multi-issue superscalar microarchitectures** to be leveraged.

Advantages include:

- Easy to create first (and sometimes only) implementation as a literal for-loop in hardware, simulators, and compilers.
- Obliterates SIMD opcode proliferation ($O(N^6)$) as well as dedicated Vectorisation ISAs. No more separate vector instructions.
- Reducing maintenance overhead (no separate Vector instructions). Adding any new Scalar instruction *automatically adds a Vectorised version of the same.*
- Easier for compilers, coders, documentation

Contents

Part II

Scalable Vectors for the Power ISA

Chapter 1

Fields and Forms

Power ISA Fields

These were originally taken from Power ISA v3.0B PDF, retain the Section Numbering from the original Power ISA v3.0B Specification PDF, and are in machine-readable format that may be parsed with the following program: power_fields.py

Some additions have been made for DRAFT Scalar instructions Forms: BM2-Form, TLI-Form and others. Other additions are for SVP64 such as SVM-Form, SVL-Form.

1.6.1 I-FORM

```
|0     |6         |30|31 |
| PO   |      LI  |AA|LK |
```

1.6.2 B-FORM

```
|0     |6   |11     |16     |30|31 |
| PO   |  BO|   BI  |   BD  |AA|LK |
```

1.6.2.1 BM-FORM

```
|0     |6    |10  |15  |22  |23    |31|
| PO   |  RS | me | sh | me |   XO |Rc|
```

1.6.2.2 BM2-FORM

```
|0     |6    |11    |16    |21   |26 |27    31|
| PO   |  RT |   RA |   RB |bm   |L  |   XO   |
```

1.6.2.2 CRB-FORM

```
|0     |6    |9   |11   |14   | 16  |19   |26   | 31|
| PO   | BF  | msk|BFA  | msk | BFB | //  | XO  | / |
| PO   | BF  | msk|BFA  | msk | BFB | TLI | XO  |TLI|
```

1.6.2.3 CW-FORM

```
|0     |6   |9 |11|12   |16   |19  |22   |26   |31|
| PO   | RA    |M |fmsk |BF   |XO  |fmap | XO     |
| PO   | BT    |M |fmsk |BF   |XO  |fmap | XO     |
| PO   | BF |  |M |fmsk |BF   |XO  |fmap | XO     |
```

1.6.2.3 CW2-FORM

```
|0     |6   |9 |11|12   |16   |19  |22   |26   |31|
| PO   | RT    |M |fmsk |BFA  |XO  |fmap | XO  |Rc|
```

1.6.3 SC-FORM

```
|0     |6    |11    |16    |20     |27  |30 |31  |
| PO   |  ///|   ///|   // |   LEV |  //|  1|  / |
```

1.6.4 D-FORM

```
|0     |6   |9  |10 |11   |16      |31 |
| PO   |    RT      |   RA|   D        |
| PO   |    RT      |   RA|   SI       |
| PO   |   RS       |   RA|   D        |
| PO   |   RS       |   RA|   UI       |
| PO   | BF | / | L |   RA|   SI       |
| PO   | BF | / | L |   RA|   UI       |
| PO   |    TO      |   RA|   SI       |
| PO   |   FRT      |   RA|   D        |
| PO   |   FRS      |   RA|   D        |
```

1.6.5 DS-FORM

```
|0     |6      |11    |16    |30  |31 |
| PO   |   RT  |   RA |   DS |  XO    |
| PO   |   RS  |   RA |   DS |  XO    |
| PO   |  RSp  |   RA |   DS |  XO    |
| PO   |  FRTp |   RA |   DS |  XO    |
| PO   |  FRSp |   RA |   DS |  XO    |
```

1.6.6 DQ-FORM

```
|0     |6     |11    |16    |28|29 |31 |
| PO   |  RTp |   RA |   DQ |   PT     |
| PO   |  S   |   RA |   DQ |SX| XO    |
| PO   |  T   |   RA |   DQ |TX| XO    |
```

V3.0B 1.6.6 DX-FORM

```
|0    |6   |11   |16   |26   |31
| PO  |  RT|   d1|   d0|   XO|d2
| PO  | FRS|   d1|   d0|   XO|d2
| PO  | FRS|     D     |   XO|D
```

1.6.7 X-FORM

```
|0     |6 |7|8|9  |10  |11|12|13  |15|16|17     |20|21    |31  |
| PO   |       RT      |    RA       |     ///     |   XO |  / |
| PO   |       RT      |    RA       |    RB       |   XO |  / |
| PO   |       RT      |    RA       |    RB       |   XO |EH  |
| PO   |       RT      |    RA       |    NB       |   XO |  / |
| PO   |       RT      | /|SR       |     ///     |   XO |  / |
| PO   |       RT      |     ///     |    RB       |   XO |  / |
| PO   |       RT      |     ///     |    RB       |   XO | 1  |
| PO   |       RT      |     ///     |     ///     |   XO |  / |
| PO   |       RS      |    RA       |    RB       |   XO |Rc  |
| PO   |       RT      |    RA       |    RB       |   XO |Rc  |
| PO   |       RS      |    RA       |    RB       |   XO | 1  |
| PO   |       RS      |    RA       |    RB       |   XO |  / |
| PO   |       RS      |    RA       |    NB       |   XO |  / |
| PO   |       RS      |    RA       |    SH       |   XO |Rc  |
| PO   |       RS      |    RA       |     ///     |   XO |Rc  |
| PO   |       RS      |    RA       |     ///     |   XO |  / |
| PO   |       RS      | /|SR       |     ///     |   XO |  / |
| PO   |       RS      |     ///     |    RB       |   XO |  / |
| PO   |       RS      |     ///     |     ///     |   XO |  / |
| PO   |       RS      |    ///   |L1|     ///     |   XO |  / |
| PO   |       TH      |    RA       |    RB       |   XO |  / |
| PO   |  BF  |/  | L  |    RA       |    RB       |   XO |  / |
| PO   |  BF  |//      |   FRA       |   FRB       |   XO |  / |
| PO   |  BF  |//      | BFA | //    |     ///     |   XO |  / |
| PO   |  BF  |//      |    ///   |W |    U     |/ |   XO |Rc  |
| PO   |  BF  |//      |     ///     |     ///     |   XO |  / |
| PO   |       TH      |    RA       |    RB       |   XO |  / |
| PO   | /| CT         |     ///     |     ///     |   XO |  / |
| PO   | /| CT         |    RA       |    RB       |   XO |  / |
| PO   |  /// | L2     |    RA       |    RB       |   XO |  / |
| PO   |  /// | L2     |     ///     |    RB       |   XO |  / |
| PO   |  /// | L2     |     ///     |     ///     |   XO |  / |
| PO   |  /// | L2     | /|     E    |     ///     |   XO |  / |
| PO   |     TO        |    RA       |    RB       |   XO |  / |
| PO   |   FRT         |    RA       |    RB       |   XO |  / |
```

```
| PO   |   FRT         |   FRA       |   FRB       |   XO |  / |
| PO   |  FRTp         |    RA       |    RB       |   XO |  / |
| PO   |   FRT         |     ///     |   FRB       |   XO |Rc  |
| PO   |   FRT         |     ///     |   FRBp      |   XO |Rc  |
| PO   |   FRT         |     ///     |     ///     |   XO |Rc  |
| PO   |  FRTp         |      ///    |   FRB       |   XO |Rc  |
| PO   |  FRTp         |      ///    |   FRBp      |   XO |Rc  |
| PO   |  FRTp         |   FRA       |   FRBp      |   XO |Rc  |
| PO   |  FRTp         |   FRAp      |   FRBp      |   XO |Rc  |
| PO   |  BF  |//      |   FRA       |   FRBp      |   XO |  / |
| PO   |  BF  |//      |   FRAp      |   FRBp      |   XO |  / |
| PO   |    FRT        |S |          | FRB         |   XO |Rc  |
| PO   |  FRTp         |S |          | FRBp        |   XO |Rc  |
| PO   |    FRS        |  RA         |  RB         |   XO |  / |
| PO   |  FRSp         |  RA         |  RB         |   XO |  / |
| PO   |     BT        |  ///        |  ///        |   XO |Rc  |
| PO   |     ///       |  RA         |  RB         |   XO |  / |
| PO   |     ///       |  ///        |  RB         |   XO |  / |
| PO   |     ///       |  ///        |  ///        |   XO |  / |
| PO   |     ///       |  ///        | E|///      |   XO |  / |
| PO   |  //|IH        |  ///        |  ///        |   XO |  / |
| PO   | A|//          |  ///        |  ///        |   XO | 1  |
| PO   | A|//     |R   |  ///        |  ///        |   XO | 1  |
| PO   |     ///       |  RA         |  RB         |   XO | 1  |
| PO   |  /// |WC      |  ///        |  ///        |   XO |  / |
| PO   |  /// |T       |  RA         |  RB         |   XO |  / |
| PO   |    VRT        |  RA         |  RB         |   XO |  / |
| PO   |   VRS         |  RA         |  RB         |   XO |  / |
| PO   |    MO         |  ///        |  ///        |   XO |  / |
| PO   |       RT      | /// |L3     |     ///     |   XO | /  |
| PO   |   FRT         |   FRA       |   FRB       |   XO | Rc |
| PO   |   FRT         |   FRA       |    RB       |   XO | Rc |
| PO   |   RT          |  ///        |   FRB       |   XO | Rc |
| PO   |   FRT         |  ///        |    RB       |   XO | Rc |
| PO   |   FRT         | IT  | ///   |    RB       |   XO | Rc |
```

1.6.7.1 DCT-FORM

```
|0     |6     |11      |16     |21      |26    |31  |
| PO   |  FRT |   FRA  |   FRB |   //   |   XO | Rc |
```

1.6.8 XL-FORM

```
|0     |6    |9  |11  |14   |16    |19|20|21    |31 |
| PO   |   BT    |     BA   |    BB      |   XO | / |
| PO   |   BO    |     BI   |  /// |BH   |   XO |LK |
| PO   |         |    ///             |S |   XO | / |
| PO   |  BF |// |BFA |//   |     ///    |   XO | / |
| PO   |              ///                |   XO | / |
| PO   |             OC                  |   XO | / |
```

1.6.9 XFX-FORM

```
|0     |6        |11|12             |20|21    |31 |
| PO   |  RT     |     spr             |   XO | / |
| PO   |  RT     |     tbr             |   XO | / |
| PO   |  RT     |0 |     ///          |   XO | / |
| PO   |  RT     |1 |  FXM          |/ |   XO | / |
| PO   |  RT     |     dcr             |   XO | / |
| PO   |  RT     |    pmrn             |   XO | / |
| PO   |  RT     |   BHRBE             |   XO | / |
| PO   |  DUI    |    DUIS             |   XO | / |
| PO   |  RS     |0 |  FXM          |/ |   XO | / |
| PO   |  RS     |1 |  FXM          |/ |   XO | / |
| PO   |  RS     |     spr             |   XO | / |
| PO   |  RS     |     dcr             |   XO | / |
| PO   |  RS     |    pmrn             |   XO | / |
```

1.6.10 XFL-FORM

```
|0     |6|7             |15|16        |21    |31 |
| PO   |L|    FLM       |W |FRB       |   XO |Rc |
```

1.6.11 XX1-FORM

```
|0     |6        |11       |16        |21    |31 |
| PO   |  T      |   RA    |   RB     |   XO |TX |
| PO   |  S      |   RA    |   RB     |   XO |SX |
```

1.6.12 XX2-FORM

```
|0     |6    |9  |11   |14  |16        |21 |30|31 |
| PO   |    T    |    ///   |    B     |XO |BX|TX |
| PO   |    T    | /// |UIM |    B     |XO |BX|TX |
| PO   |  BF | //|      /// |    B     |XO |BX| / |
```

1.6.13 XX3-FORM

```
|0     |6     |9    |11   |16   |21 |22  |24    |29|30|31 |
| PO   |     T       |   A |   B |         XO    |AX|BX|TX |
| PO   |     T       |   A |   B |Rc |       XO  |AX|BX|TX |
| PO   |  BF  |  // |   A |   B |         XO    |AX|BX|/  |
| PO   |     T       |   A |   B |XO |SHW |   XO |AX|BX|TX |
| PO   |     T       |   A |   B |XO |DM  |   XO |AX|BX|TX |
```

1.6.14 XX4-FORM

```
|0     |6   |11   |16   |21   |26  |28|29 |30|31 |
```

```
| PO   |  T |   A |   B |   C | XO |CX|AX |BX|TX |
```

1.6.15 XS-FORM

```
|0     |6    |11    |16    |21    |30|31 |
| PO   |  RS |   RA |   sh |   XO |sh|Rc |
```

1.6.15 XB-FORM

```
|0     |6    |11    |16    |22    |31 |
| PO   |  RT |   RA |  XBI |   XO |Rc |
```

1.6.16 XO-FORM

```
|0   |6   |11  |13   |16   |21  |22  |31  |
| PO | RT | RA       | RB  | OE | XO | Rc |
| PO | RT | RA       | RB  |  / | XO | Rc |
| PO | RT | RA       | RB  |  / | XO | /  |
| PO | RT | RA       | /// | OE | XO | Rc |
| PO | RT | IT | CVM | FRB | OE | XO | Rc |
```

1.6.17 A-FORM

```
|0     |6     |11     |16      |21      |26    |31 |
| PO   |  FRT |  FRA  |  FRB   |   FRC  |   XO |Rc |
| PO   |  FRT |  FRA  |  FRB   |   ///  |   XO |Rc |
| PO   |  FRT |  FRA  |  ///   |   FRC  |   XO |Rc |
| PO   |  FRT |  ///  |  FRB   |   ///  |   XO |Rc |
| PO   |   RT |  RA   |  RB    |   BC   |   XO |  /|
| PO   |   RT |  RA   |  RB    |   SH   |   XO |Rc |
```

1.6.18 M-FORM

```
|0     |6    |11    |16    |21    |26    |31|
| PO   |  RS |   RA |   RB |   MB |   ME |Rc|
| PO   |  RS |   RA |   SH |   MB |   ME |Rc|
```

1.6.19 MD-FORM

```
|0     |6    |11    |16    |21    |27|30|31|
| PO   |  RS |   RA |   sh |   mb |XO|sh|Rc|
| PO   |  RS |   RA |   sh |   me |XO|sh|Rc|
```

1.6.20 MDS-FORM

```
|0     |6    |11    |16    |21    |27    |31|
| PO   |  RS |   RA |   RB |   mb |   XO |Rc|
| PO   |  RS |   RA |   RB |   me |   XO |Rc|
```

1.6.21 VA-FORM

```
|0      |6     |11     |16     |21|22 |25|26   |31|
| PO    |  RT  |   RA  |   RB  |   RC    |   XO   |
| PO    |  VRT |   VRA |   VRB |  VRC    |   XO   |
| PO    |  VRT |   VRA |   VRB | /|SHB   |   XO   |
| PO    |  VRT |   VRA |   VRB | /|BFA|/ |   XO   |
```

1.6.21.1 VA2-FORM

```
|0      |6     |11     |16     |21  |24|26  |31  |
| PO    |  RT  |   RA  |   RB  | RC    | XO | Rc |
```

1.6.22 VC-FORM

```
|0      |6     |11     |16     |21|22   |31|
| PO    |  VRT |   VRA |   VRB |Rc|   XO   |
```

1.6.23 VX-FORM

```
|0      |6     |11           |16    |21   |31|
| PO    |  VRT |     VRA     | VRB  |   XO   |
| PO    |  VRT |      ///    | VRB  |   XO   |
| PO    |  VRT |     UIM     | VRB  |   XO   |
| PO    |  VRT | / UIM       | VRB  |   XO   |
| PO    |  VRT |  // UIM     | VRB  |   XO   |
| PO    |  VRT |   /// UIM   | VRB  |   XO   |
| PO    |  VRT |     SIM     |   ///|   XO   |
| PO    |  VRT |          ///|      |   XO   |
| PO    |      |///          | VRB  |   XO   |
```

1.6.24 EVX-FORM

```
|0      |6   |9 |11    |16    |21    |31|
| PO    |   RS  |   RA |   RB |   XO    |
| PO    |   RS  |   RA |   UI |   XO    |
| PO    |   RT  |   ///|   RB |   XO    |
| PO    |   RT  |   RA |   RB |   XO    |
| PO    |   RT  |   RA |   ///|   XO    |
| PO    |   RT  |   UI |   RB |   XO    |
| PO    |  BF|//|   RA |   RB |   XO    |
```

```
| PO    |   RT  |   RA |   UI |   XO    |
| PO    |   RT  |   SI |   ///|   XO    |
```

1.6.25 EVS-FORM

```
|0      |6   |11    |16    |21    |29 |31 |
| PO    |  RT|   RA |   RB |   XO |BFA    |
```

1.6.26 Z22-FORM

```
|0      |6  |9 |11     |16     |22    |31 |
| PO    | BF|//|   FRA |   DCM |   XO | / |
| PO    | BF|//|  FRAp |   DCM |   XO | / |
| PO    | BF|//|   FRA |   DGM |   XO | / |
| PO    | BF|//|  FRAp |   DGM |   XO | / |
| PO    |  FRT |   FRA |    SH |   XO |Rc |
| PO    |  FRTp|  FRAp |    SH |   XO |Rc |
```

1.6.27 Z23-FORM

```
|0     |6     |11    |15 |16     |21 |23    |31 |
| PO   |  FRT |    TE    |   FRB |RMC|   XO |Rc |
| PO   |  FRTp|    TE    |  FRBp |RMC|   XO |Rc |
| PO   |  FRT |   FRA    |   FRB |RMC|   XO |Rc |
| PO   |  RT  |   RA     |   RB  |sm |   XO |Rc |
| PO   |  RT  |   RA     |   RB  |CY |   XO |Rc |
| PO   |  FRTp|   FRA    |  FRBp |RMC|   XO |Rc |
| PO   |  FRTp|  FRAp    |  FRBp |RMC|   XO |Rc |
| PO   |  FRT |  /// | R | FRB   |RMC|   XO |Rc |
| PO   |  FRTp|  /// | R | FRBp  |RMC|   XO |Rc |
```

1.6.29 SVI-FORM

```
|0     |6    |11    |16   |21 |23  |24|25|26    31|
| PO   |  SVG|rmm   | SVd |ew |SVyx|mm|sk|   XO   |
```

1.6.30 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 |
```

1.6.31 SVC-FORM

```
|0     |6    |9    |11     |
| PO   | SCi | SCm | SCi   |
```

1.6.32 SVR-FORM

```
|0     |6    |9    |11    | 15     |
| PO   | SCi | SCm | SRb  | SRi    |
```

1.6.33 SVD-FORM

```
|0     |6    |11   |16   |21      |31 |
| PO   | RT  |   RA| RC  |  SVD       |
| PO   | RS  |   RA| RC  |  SVD       |
| PO   | FRT |   RA| RC  |  SVD       |
| PO   | FRS |   RA| RC  |  SVD       |
```

1.6.34 SVDS-FORM

```
|0     |6      |11    |16   |21    |30  |31 |
| PO   |   RT  |   RA | RC  | SVDS |  XO    |
| PO   |   RS  |   RA | RC  | SVDS |  XO    |
```

1.6.35 SVM-FORM

```
|0     |6        |11      |16    |21    |25 |26    |31  |
| PO   |  SVxd   |   SVyd | SVzd | SVrm |vf |   XO      |
```

1.6.35.1 SVM2-FORM

```
|0     |6     |10  |11      |16    |21 |24|25 |26    |31  |
| PO   | SVo  |SVyx|   rmm  | SVd  |XO |mm|sk |   XO      |
```

1.6.36 SVRM-FORM

```
|0     |6     |11  |13   |15   |17   |19   |21  |22   |26     |31 |
| PO   | SVme |mi0 | mi1 | mi2 | mo0 | mo1 |pst |///  | XO        |
```

1.6.37 TLI-FORM

```
|0   |6   |11   |16   |21   |29  |31 |
| PO | RT |  RA |  RB | TLI | XO |Rc |
| PO | RT |  RA |  RB | TLI | XO |L  |
```

1.6.38 MM-FORM

```
|0    |6    |11   |16   |21   |24 |25  |31  |
| PO  | FRT | FRA | FRB | FMM     | XO | Rc |
| PO  | RT  | RA  | RB  | MMM | / | XO | Rc |
```

1.6.28 Instruction Fields

```
A (6)
    Field used by the tbegin. instruction to specify an
    implementation-specific function.
    Field used by the tend. instruction to specify the
    completion of the outer transaction and all nested
    transactions.
    Formats: X
AA (30)
    Absolute Address.
    0    The immediate field represents an address
         relative to the current instruction address. For
         I-form branches the effective address of the
         branch target is the sum of the LI field
         sign-extended to 64 bits and the address of
         the branch instruction. For B-form branches
         the effective address of the branch target is
         the sum of the BD field sign-extended to 64
         bits and the address of the branch instruction.
    1    The immediate field represents an absolute
         address. For I-form branches the effective
         address of the branch target is the LI field
         sign-extended to 64 bits. For B-form branches
         the effective address of the branch target is
         the BD field sign-extended to 64 bits.
    Formats: B, I
AX,A (29,11:15)
    Fields that are concatenated to specify a VSR to
    be used as a source.
    Formats: XX3, XX4
BA (11:15)
    Field used to specify a bit in the CR to be used as
    a source.
    Formats: XL
BB (16:20)
     Field used to specify a bit in the CR to be used as
     a source.
     Formats: XL
BC (21:25)
     Field used to specify a bit in the CR to be used as
     a source.
     Formats: A
BD (16:29)
     Immediate field used to specify a 14-bit signed
     two's complement branch displacement which is
     concatenated on the right with 0b00 and
```

```
     sign-extended to 64 bits.
     Formats: B
BF (6:8)
     Field used to specify one of the CR fields or one of
     the FPSCR fields to be used as a target.
     Formats: D, X, XL, XX2, XX3, Z22
BFA (22:24)
     Field used to specify one of the CR fields
     to be used as a source.
     Formats: VA
BFA (29:31)
     Field used to specify one of the CR fields or one of
     the FPSCR fields to be used as a source.
     Formats: EVS
BFA (11:13)
     Field used to specify one of the CR fields or one of
     the FPSCR fields to be used as a source.
     Formats: X, XL
BH (19:20)
     Field used to specify a hint in the Branch Condi-
     tional to Link Register and Branch Conditional to
     Count Register instructions. The encoding is
     described in Section 2.4, 'Branch Instructions'.
     Formats: XL
BHRBE (11:20)
     Field used to identify the BHRB entry to be used
     as a source by the Move From Branch History
     Rolling Buffer instruction.
     Formats: XFX
BI (11:15)
     Field used to specify a bit in the CR to be tested by
     a Branch Conditional instruction.
     Formats: B, XL
bm (21:25)
     Field used to specify the Bit-mask Mode for bmask
     Formats: BM2
BO (6:10)
     Field used to specify options for the Branch Condi-
     tional instructions. The encoding is described in
     Section 2.4, 'Branch Instructions'.
     Formats: B, XL, X, XL
BT (6:10)
     Field used to specify a bit in the CR or in the
     FPSCR to be used as a target.
     Formats: XL
BX,B (30,16:20)
    Fields that are concatenated to specify a VSR to
    be used as a source.
    Formats: XX2, XX3, XX4
CT (7:10)
    Field used in X-form instructions to specify a cache
    target (see Section 4.3.2 of Book II).
    Formats: X
CVM (13:15)
    Field used to specify conversion mode for
```

```
        integer -> floating-point conversion.
        Formats: XO
    CX,C (28,21:25)
        Fields that are concatenated to specify a VSR to
        be used as a source.
        Formats: XX4
    CY (21:22)
        Immediate field used for addex instruction
        Formats: Z23
    D (16:31)
        Immediate field used to specify a 16-bit signed
        two's complement integer which is sign-extended
        to 64 bits.
        Formats: D
    d0,d1,d2 (16:25,11:15,31)
        Immediate fields that are concatenated to specify a
        16-bit signed two's complement integer which is
        sign-extended to 64 bits.
        Formats: DX
    dc,dm,dx (25,29,11:15)
        Immediate fields that are concatenated to specify
        Data Class Mask.
        Formats: XX2
    DCM (16:21)
        Immediate field used to specify Data Class Mask.
        Formats: Z22
    DCMX (9:15)
        Immediate field used to specify Data Class Mask.
        Formats: X, XX2
    DGM (16:21)
        Immediate field used as the Data Group Mask.
        Formats: Z22
    DM (22:23)
        Immediate field used by xxpermdi instruction as
        doubleword permute control.
        Formats: XX3
    DRM (18:20)
        Immediate operand field used to specify new deci-
        mal floating-point rounding mode.
        Formats: X
    DUI (6:10)
        Field used by the dnh instruction (see Book III-E).
        Formats: XFX
    DUIS (11:20)
        Field used by the dnh instruction (see Book III-E).
        Formats: XFX
    DQ (16:27)
        Immediate field used to specify a 12-bit signed
        two's complement integer which is concatenated
        on the right with 0b0000 and sign-extended to 64
        bits.
        Formats: DQ
    DS (16:29)
        Immediate field used to specify a 14-bit signed
        two's complement integer which is concatenated
```

```
    on the right with 0b00 and sign-extended to 64 bits.
    Formats: DS
EH (31)
    Field used to specify a hint in the Load and
    Reserve instructions. The meaning is described in
    Section 4.6.2, 'Load and Reserve and Store Con-
    ditional Instructions', in Book II.
    Formats: X
EO (11:12)
    Expanded opcode field
    Formats: X
EO (11:15)
    Expanded opcode field
    Formats: VX, X, XX2
EX (31)
    Field used to specify Inexact form of round to
    quad-precision integer.
    Formats: X
ew (21:22)
    Field used to specify the element width for SVI-Form
    Formats: SVI
FC (16:20)
    Field used to specify the function code in Load/
    Store Atomic instructions.
    Formats: X
FLM (7:14)
    Field mask used to identify the FPSCR fields that
    are to be updated by the mtfsf instruction.
    Formats: XFL
FMM (21:24)
    Field used to specify minimum/maximum mode for fminmax[s].
    Formats: MM
fmap (22:25)
    Field used to specify the CR Field set/clear map for CR Weird
    instructions.
    Formats: CW, CW2
fmsk (12:15)
    Field used to specify the CR Field mask for CR Weird instructions.
    Formats: CW, CW2
FRA (11:15)
    Field used to specify a FPR to be used as a
    source.
    Formats: A, MM, X, Z22, Z23, DCT
FRAp (11:15)
    Field used to specify an even/odd pair of FPRs to
    be concatenated and used as a source.
    Formats: X, Z22, Z23
FRB (16:20)
    Field used to specify an FPR to be used as a
    source.
    Formats: A, MM, X, XFL, XO, Z23, DCT
FRBp (16:20)
     Field used to specify an even/odd pair of FPRs to
     be concatenated and used as a source.
     Formats: X, Z23
```

```
FRC (21:25)
     Field used to specify an FPR to be used as a
     source.
     Formats: A
FRS (6:10)
     Field used to specify an FPR to be used as a
     source.
     Formats: D, X, DX
FRSp (6:10)
     Field used to specify an even/odd pair of FPRs to
     be concatenated and used as a source.
     Formats: DS, X
FRT (6:10)
     Field used to specify an FPR to be used as a tar-
     get.
     Formats: A, D, MM, X, Z22, Z23, DCT
FRTp (6:10)
     Field used to specify an even/odd pair of FPRs to
     be concatenated and used as a target.
     Formats: DS, X, Z22, Z23
FXM (12:19)
     Field mask used to identify the CR fields that are to
     be written by the mtcrf and mtocrf instructions, or
     read by the mfocrf instruction.
     Formats: XFX
IB (16:20)
     Immediate field used to specify a 5-bit signed inte-
     ger.
     Formats: MDS
IH (8:10)
     Field used to specify a hint in the SLB Invalidate
     All instruction. The meaning is described in
     Section 5.9.3.2, 'SLB Management Instructions',
     in Book III.
     Formats: X
IMM8 (13:20)
     Immediate field used to specify an 8-bit integer.
     Formats: X
IS (6:10)
     Immediate field used to specify a 5-bit signed inte-
     ger.
     Formats: MDS
IT (11:12)
    Field used to specify integer type for FPR <-> GPR conversions.
    Formats: X, XO
L (6)
     Field used to specify whether the mtfsf instruction
     updates the entire FPSCR.
     Formats: XFL
L2 (9:10)
     Field used by the Data Cache Block Flush instruc-
     tion (see Section 4.3.2 of Book II) and also by the
     Synchronize instruction (see Section 4.6.3 of Book
     II).
     Formats: X
```

```
L (10)
     Field used to specify whether a fixed-point Com-
     pare instruction is to compare 64-bit numbers or
     32-bit numbers.
     Field used by the Compare Range Byte instruction
     to indicate whether to compare against 1 or 2
     ranges of bytes.
     Formats: D, X
L1 (15)
     Field used by the Move To Machine State Register
     instruction (see Book III).
     Field used by the SLB Move From Entry VSID and
     SLB Move From Entry ESID instructions for imple-
     mentation-specific purposes.
     Formats: X
L3 (14:15)
     Field used by the Deliver A Random Number
     instruction (see Section 3.3.9, 'Fixed-Point Arith-
     metic Instructions') to choose the random number
     format.
     Formats: X
L (26)
     Field used to specify whether mask-in occurs in bmask
     Formats: BM2
L (31)
     Field used to specify whether the grevlut instruction
     updates the whole GPR or the first half.
     Formats: TLI
LEV (20:26)
     Field used by the System Call instructions.
     Formats: SC
LI (6:29)
     Immediate field used to specify a 24-bit signed
     two's complement integer which is concatenated
     on the right with 0b00 and sign-extended to 64
     bits.
     Formats: I
LK (31)
     LINK bit.
     0     Do not set the Link Register.
     1     Set the Link Register. The address of the
           instruction following the Branch instruction is
           placed into the Link Register.
     Formats: B, I, XL
rmm (11:15)
    Field used to specify a REMAP shape for SVI-Form
    Formats: SVI
msk (9:10,14:15)
    Field used by crternlogi and crbinlut to select which bits
    of CR Field BF are to be modified. Requires BF to be Read-Modify-Write
    Formats: CRB
MB (21:25)
    Field used in M-form instructions to specify the first
    1-bit of a 64-bit mask, as described in
    Section 3.3.14, 'Fixed-Point Rotate and Shift
```

```
    Instructions' on page 101.
    Formats: M
mb (21:26)
    Field used in MD-form and MDS-form instructions
    to specify the first 1-bit of a 64-bit mask, as
    described in Section 3.3.14, 'Fixed-Point Rotate
    and Shift Instructions' on page 101.
    Formats: MD, MDS
me (21:26)
    Field used in MD-form and MDS-form instructions
    to specify the last 1-bit of a 64-bit mask, as
    described in Section 3.3.14, 'Fixed-Point Rotate
    and Shift Instructions' on page 101.
    Formats: MD, MDS
ME (26:30)
    Field used in M-form instructions to specify the last
    1-bit of a 64-bit mask, as described in
    Section 3.3.14, 'Fixed-Point Rotate and Shift
    Instructions' on page 101.
    Formats: M
mi0 (11:12)
    Field used in REMAP to select the SVSHAPE for 1st input register
    Formats: SVRM
mi1 (13:14)
    Field used in REMAP to select the SVSHAPE for 2nd input register
    Formats: SVRM
mi2 (15:16)
    Field used in REMAP to select the SVSHAPE for 3rd input register
    Formats: SVRM
mm (24)
    Field used to specify the meaning of the rmm field for SVI-Form
    and SVM2-Form
    Formats: SVI, SVM2
MMM (21:23)
    Field used to specify minimum/maximum mode for integer minmax.
    Formats: MM
mo0 (17:18)
    Field used in REMAP to select the SVSHAPE for 1st output register
    Formats: SVRM
mo1 (19:20)
    Field used in REMAP to select the SVSHAPE for 2nd output register
    Formats: SVRM
MO (6:10)
    Field used in X-form instructions to specify a sub-
    set of storage accesses.
    Formats: X
ms (23)
    Field used in Simple-V to specify whether MVL is to be set
    Formats: SVL
NB (16:20)
    Field used to specify the number of bytes to move
    in an immediate Move Assist instruction.
    Formats: X
OC (6:20)
    Field used by the Embedded Hypervisor Privilege
```

```
    instruction.
    Formats: XL
OE (21)
    Field used by XO-form instructions to enable set-
    ting OV and SO in the XER.
    Formats: XO
PO (0:5)
    Primary opcode field.
    Formats: all
PRS (14)
    Field used to specify whether to invalidate pro-
    cess- or partition-scoped entries for tlbie[l].
    Formats: X
PS (22)
    Field used to specify preferred sign for BCD opera-
    tions.
    Formats: VX
pst (21)
    Field used in REMAP to indicate "persistence" mode (REMAP
    continues to apply to multiple instructions)
    Formats: SVRM
PT (28:31)
    Immediate field used to specify a 4-bit unsigned
    value.
    Formats: DQ
R (10)
    Field used by the tbegin. instruction to specify the
    start of a ROT.
    Formats: X
R (15)
    Immediate field that specifies whether the RMC is
    specifying the primary or secondary encoding
    Field used to specify whether to invalidate Radix
    Tree or HPT entries for tlbie[l].
    Formats: X, Z23
RA (11:15)
    Field used to specify a GPR to be used as a
    source or as a target.
    Formats: A, BM2, D, DQ, DQE, DS, M, MD, MDS, MM, TX, VA, VA2, VX, X, XO, XS, SVL, XB, TLI, Z23
RB (16:20)
    Field used to specify a GPR to be used as a
    source.
    Formats: A, BM2, M, MDS, MM, VA, VA2, X, XO, TLI, Z23
Rc (21)
    RECORD bit.
    0    Do not alter the Condition Register.
    1    Set Condition Register Field 6 as described in
         Section 2.3.1,     'Condition  Register'     on
         page 30.
    Formats: VC, XX3
RC (21:25)
    Field used to specify a GPR to be used as a
    source.
    Formats: VA, VA2, SVD, SVDS
Rc (31)
```

```
    RECORD bit.
    0    Do not alter the Condition Register.
    1    Set Condition Register Field 0 or Field 1 as
         described in Section 2.3.1, 'Condition Regis-
         ter' on page 30.
    Formats: A, M, MD, MDS, MM, VA2, X, XFL, XO, XS, Z22, Z23, SVL, XB, TLI, DCT
RIC (12:13)
    Field used to specify what types of entries to inval-
    idate for tlbie[l].
    Formats: X
RM (19:20)
    Immediate operand field used to specify new
    binary floating-point rounding mode.
    Formats: X
RMC (21:22)
    Immediate field used for DFP rounding mode con-
    trol.
    Formats: Z23
rmm (11:15)
    REMAP Mode field for SVI-Form and SVM2-Form
    Formats: SVI, SVM2
RO (31)
    Round to Odd override
    Formats: X
RS (6:10)
    Field used to specify a GPR to be used as a
    source.
    Formats: D, DS, M, MD, MDS, X, XFX, XS
RSp (6:10)
    Field used to specify an even/odd pair of GPRs to
    be concatenated and used as a source.
    Formats: DS, X
RT (6:10)
    Field used to specify a GPR to be used as a target.
    Formats: A, BM2, D, DQE, DS, DX, MM, VA, VA2, VX, X, XFX, XO, XX2, SVL, XB, TLI, Z23
RTp (6:10)
    Field used to specify an even/odd pair of GPRs to
    be concatenated and used as a target.
    Formats: DQ, X
S (11)
    Immediate field that specifies signed versus
    unsigned conversion.
    Formats: X
S (20)
    Immediate field that specifies whether or not the
    rfebb     instruction   re-enables      event-based
    branches.
    Formats: XL
SCi (6:8)
    Index to SV Context Propagation SPR
    Formats: SVC, SVR
SCm (9:10)
    SV Context Propagation Mode
    Formats: SVC, SVR
SCi (11:31)
```

```
    SV Context Propagation immediate bitfield
    Formats: SVC
sm (21:22)
    Immediate field used for selecting operands (shift mode)
    Formats: Z23
SRb (11:14)
    SV REMAP byte-reversal field.
    Formats: SVC
SRi (15:31)
    SV REMAP immediate FIFO bitfield
    Formats: SVC
SH (16:20)
    Field used to specify a shift amount.
    Formats: M, X
SH (16:21)
    Field used to specify a shift amount.
    Formats: Z22
SH (21:25)
    Field used to specify a shift amount.
    Formats: A
sh (30,16:20)
    Fields that are concatenated to specify a shift
    amount.
    Formats: MD, XS
SHB (22:25)
    Field used to specify a shift amount in bytes.
    Formats: VA
SHW (22:23)
     Field used to specify a shift amount in words.
     Formats: XX3
SI (16:20)
     Immediate field used to specify a 5-bit signed inte-
     ger.
     Formats: X
SI (16:31)
     Immediate field used to specify a 16-bit signed
     integer.
     Formats: D
SIM (11:15)
     Immediate field used to specify a 5-bit signed inte-
     ger.
     Formats: VX
sk (25)
    Field used to specify dimensional skipping in svindex
    Formats: SVI, SVM2
SP (11:12)
     Immediate field that specifies signed versus
     unsigned conversion.
     Formats: X
spr (16:20,11:15)
     Field used to specify a Special Purpose Register
     for the mtspr and mfspr instructions.
     Formats: XFX
SPR (11:20)
     Field used to specify a Special Purpose Register
```

```
     for the mtspr and mfspr instructions.
     Formats: XFX
SR (12:15)
     Field used by the Segment Register Manipulation
     instructions (see Book III).
     Formats: X
SVd (16:20)
    Immediate field used to specify the size of the REMAP dimension
    in the svindex and svshape2 instructions
    Formats: SVI, SVM2
SVD (21:31)
    Immediate field used to specify an 11-bit signed
    two's complement integer which is sign-extended
    to 64 bits.
    Formats: SVD
SVDS (16:29)
    Immediate field used to specify a 9-bit signed
    two's complement integer which is concatenated
    on the right with 0b00 and sign-extended to 64 bits.
    Formats: SVDS
SVG (6:10)
    Field used to specify a GPR to be used as a
    source for indexing.
    Formats: SVI
SVi (16:22)
     Simple-V immediate field for setting VL or MVL
     Formats: SVL
SVme (6:10)
     Simple-V "REMAP" map-enable bits (0-4)
     Formats: SVRM
SVo (6:9)
    Field used by the svshape2 instruction as an offset
    Formats: SVM2
SVrm (21:24)
     Simple-V "REMAP" Mode
     Formats: SVM
SVxd (6:10)
     Simple-V "REMAP" x-dimension size
     Formats: SVM
SVyd (11:15)
     Simple-V "REMAP" y-dimension size
     Formats: SVM
SVzd (16:20)
     Simple-V "REMAP" z-dimension size
     Formats: SVM
SX,S (28,6:10)
     Fields SX and S are concatenated to specify a
     VSR to be used as a source.
     Formats: DQ
SX,S (31,6:10)
     Fields SX and S are concatenated to specify a
     VSR to be used as a source.
     Formats: X
T (9:10)
      Field used to specify the type of invalidation done
```

```
      by a TLB Invalidate Local instruction (see Book
      III-E).
     Formats: X
TBR (11:20)
     Field used by the Move From Time Base instruc-
     tion (see Section 6.1 of Book II).
     Formats: X
TE (11:15)
     Immediate field that specifies a DFP exponent.
     Formats: Z23
TH (6:10)
     Field used by the data stream variant of the dcbt
     and dcbtst instructions (see Section 4.3.2 of Book
     II).
     Formats: X
TLI (21:28)
     Field used by the ternlogi instruction as the
     look-up table.
     Formats: TLI
TLI (21:25,19:20,31)
    Field used by the crternlogi instruction as the
    look-up table.
    Formats: CRB
TO (6:10)
     Field used to specify the conditions on which to
     trap.     The    encoding     is    described   in
     Section 3.3.10.1,     'Character-Type     Compare
     Instructions' on page 87.
     Formats: D, X
TX,T (28,6:10)
     Fields that are concatenated to specify a VSR to
     be used as either a target.
     Formats: DQ
TX,T (31,6:10)
     Fields that are concatenated to specify a VSR to
     be used as either a target or a source.
     Formats: X, XX2, XX3, XX4
U (16:19)
     Immediate field used as the data to be placed into
     a field in the FPSCR.
     Formats: X
UI (16:20)
     Immediate field used to specify a 5-bit unsigned
     integer.
     Formats: TX
UI (16:31)
     Immediate field used to specify a 16-bit unsigned
     integer.
     Formats: D
UIM (11:15)
     Immediate field used to specify a 5-bit unsigned
     integer.
     Formats: VX, X
UIM (12:15)
     Immediate field used to specify a 4-bit unsigned
```

```
     integer.
     Formats: VX, XX2
UIM (13:15)
     Immediate field used to specify a 3-bit unsigned
     integer.
     Formats: VX
UIM (14:15)
     Immediate field used to specify a 2-bit unsigned
     integer.
     Formats: VX, XX2
VRA (11:15)
     Field used to specify a VR to be used as a source.
     Formats: VA, VC, VX
VRB (16:20)
    Field used to specify a VR to be used as a source.
    Formats: VA, VC, VX
VRC (21:25)
    Field used to specify a VR to be used as a source.
    Formats: VA
VRS (6:10)
    Field used to specify a VR to be used as a source.
    Formats: DS, X
VRT (6:10)
    Field used to specify a VR to be used as a target.
    Formats: DS, VA, VC, VX, X
vf (25)
    Field used in Simple-V to specify whether "Vertical" Mode is set
    Formats: SVL, SVM
vs (24)
    Field used in Simple-V to specify whether VL is to be set
    Formats: SVL
W (15)
    Field used by the mtfsfi and mtfsf instructions to
    specify the target word in the FPSCR.
    Formats: X, XFL
WC (9:10)
    Field used to specify the condition or conditions
    that cause instruction execution to resume after
    executing a wait instruction (see Section 4.6.4 of
    Book II).
    Formats: X
XBI (21:24)
    Field used to specify a bit in the XER.
    Formats: MDS, MDS, TX
XBI (16:21)
    Field used to specify a 6-bit unsigned immediate for bit manipulation
    instructions, such as grevi.
    Formats: XB
XO (21:23,26:31)
    Extended opcode field.
    Formats: SVM2
XO (21,23:31)
    Extended opcode field.
    Formats: VX
XO (21:24,26:28)
```

```
    Extended opcode field.
    Formats: XX2
XO (21:24:28)
    Extended opcode field.
    Formats: XX3
XO (21:28)
    Extended opcode field.
    Formats: XX3
XO (21:29)
    Extended opcode field.
    Formats: XS, XX2
XO (21:30)
    Extended opcode field.
    Formats: X, XFL, XFX, XL
XO (21:31)
    Extended opcode field.
    Formats: VX
XO (22:30)
    Extended opcode field.
    Formats: XO, XX3, Z22, XB
XO (22:31)
    Extended opcode field.
    Formats: VC
XO (23:30)
    Extended opcode field.
    Formats: X, Z23
XO (25:30)
    Extended opcode field.
    Formats: MM, TX
XO (26:27)
    Extended opcode field.
    Formats: XX4
XO (26:30)
    Extended opcode field.
    Formats: A, DX, VA2, SVL, CRB, DCT
XO (26:31)
    Extended opcode field.
    Formats: VA, SVM, SVRM, SVI
XO (27:29)
    Extended opcode field.
    Formats: MD
XO (27:30)
    Extended opcode field.
    Formats: MDS
XO (27:31)
    Extended opcode field.
    Formats: BM2
XO (29:31)
    Extended opcode field.
    Formats: DQ
XO (29:30)
    Extended opcode field.
    Formats: TLI
XO (30)
    Extended opcode field.
```

```
    Formats: SC
XO (30:31)
    Extended opcode field.
    Formats: DQE, DS, SC
SVyx (23)
    Field used to specify loop dimension order in svindex
    Formats: SVI
SVyx (10)
    Field used to specify loop dimension order in svshape2
    Formats: SVM2
```

Chapter 2

Scalable Vectors for the Power ISA

[[!tag standards]]

Obligatory Dilbert:

Links:

- https://bugs.libre-soc.org/show_bug.cgi?id=213
- https://youtu.be/ZQ5hw9AwO1U walkthrough video (19jun2022)
- https://ftp.libre-soc.org/simple_v_spec.pdf PDF version of this DRAFT specification

SV is in DRAFT STATUS. SV has not yet been submitted to the OpenPOWER Foundation ISA WG for review.

===

2.1 Scalable Vectors for the Power ISA

SV is designed as a strict RISC-paradigm Scalable Vector ISA for Hybrid 3D CPU GPU VPU workloads. As such it brings features normally only found in Cray Supercomputers (Cray-1, NEC SX-Aurora) and in GPUs, but keeps strictly to a *Simple* RISC principle of leveraging a *Scalar* ISA, exclusively using "Prefixing". **Not one single actual explicit Vector opcode exists in SV, at all**. It is suitable for low-power Embedded and DSP Workloads as much as it is for power-efficient Supercomputing.

Fundamental design principles:

- Taking the simplicity of the RISC paradigm and applying it strictly and uniformly to create a Scalable Vector ISA.
- Effectively a hardware for-loop, pausing PC, issuing multiple scalar operations
- Preserving the underlying scalar execution dependencies as if the for-loop had been expanded as actual scalar instructions (termed "preserving Program Order")
- Specifically designed to be Precise-Interruptible at all times (many Vector ISAs have operations which, due to higher internal accuracy or other complexity, must be effectively atomic only for the full Vector operation's duration, adversely affecting interrupt response latency, or be abandoned and started again)
- Augments ("tags") existing instructions, providing Vectorisation "context" rather than adding new instructions.
- Strictly does not interfere with or alter the non-Scalable Power ISA in any way
- In the Prefix space, does not modify or deviate from the underlying scalar Power ISA unless it provides significant performance or other advantage to do so in the Vector space (dropping the "sticky" characteristics of XER.SO and CR0.SO for example)
- Designed for Supercomputing: avoids creating significant sequential dependency hazards, allowing standard high performance superscalar multi-issue micro-architectures to be leveraged.

- Divided into Compliancy Levels to reduce cost of implementation for specific needs.

Advantages of these design principles:

- Simplicity of introduction and implementation on top of the existing Power ISA without disruption.
- It is therefore easy to create a first (and sometimes only) implementation as literally a for-loop in hardware, simulators, and compilers.
- Hardware Architects may understand and implement SV as being an extra pipeline stage, inserted between decode and issue, that is a simple for-loop issuing element-level sub-instructions.
- More complex HDL can be done by repeating existing scalar ALUs and pipelines as blocks and leveraging existing Multi-Issue Infrastructure
- As (mostly) a high-level "context" that does not (significantly) deviate from scalar Power ISA and, in its purest form being "a for loop around scalar instructions", it is minimally-disruptive and consequently stands a reasonable chance of broad community adoption and acceptance
- Completely wipes not just SIMD opcode proliferation off the map (SIMD is O(N^6) opcode proliferation) but off of Vectorisation ISAs as well. No more separate Vector instructions.

Comparative instruction count:

- ARM NEON SIMD: around 2,000 instructions, prerequisite: ARM Scalar.
- ARM SVE: around 4,000 instructions, prerequisite: NEON and ARM Scalar
- ARM SVE2: around 1,000 instructions, prerequisite: SVE, NEON, and ARM Scalar for a grand total of well over 7,000 instructions.
- Intel AVX-512: around 4,000 instructions, prerequisite AVX, AVX2, AVX-128 and AVX-256 which in turn critically rely on the rest of x86, for a grand total of well over 10,000 instructions.
- RISV-V RVV: 192 instructions, prerequisite 96 Scalar RV64GC instructions
- SVP64: **six** instructions, two of which are in the same space (svshape, svshape2), with 24-bit prefixing of prerequisite SFS (150) or SFFS (214) Compliancy Subsets. **There are no dedicated Vector instructions, only Scalar-prefixed**.

Comparative Basic Design Principle:

- ARM NEON and VSX: PackedSIMD. No instruction-overloaded meaning (every instruction is unique for a given register bitwidth, guaranteeing binary interoperability)
- Intel AVX-512 (and below): Hybrid Packed-Predicated SIMD with no instruction-overloading, guaranteeing binary interoperability but at the same time penalising the ISA with runaway opcode proliferation.
- ARM SVE/SVE2: Hybrid Packed-Predicated SIMD with instruction-overloading that destroys binary interoperability. This is hidden behind the misuse of the word "Scalable" and is **permitted under License** by "Silicon Partners".
- RISC-V RVV: Cray-style Scalable Vector but with instruction-overloading **permitted by the specification** that destroys binary interoperability.
- SVP64: Cray-style Scalable Vector with no instruction-overloaded meanings. The regfile numbers and bitwidths shall **not** change in a future revision (for the same instruction encoding): "Silicon Partner" Scaling is prohibited, in order to guarantee binary interoperability. Future revisions of SVP64 may extend VSX instructions to achieve larger regfiles, and non-interoperability on the same will likewise be prohibited.

SV comprises several {Compliancy Levels} suited to Embedded, Energy efficient High-Performance Compute, Distributed Computing and Advanced Computational Supercomputing. The Compliancy Levels are arranged such that even at the bare minimum Level, full Soft-Emulation of all optional and future features is possible.

2.2 Sub-pages

Pages being developed and examples

- {Exevutive Summary}
- {Overview Chapter} explaining the basics.
- {Compliancy Levels} for minimum subsets through to Advanced Supercomputing.
- [[sv/implementation]] implementation planning and coordination

- [[sv/po9_encoding]] a new DRAFT 64-bit space similar to EXT1xx, introducing new areas EXT232-63 and EXT300-363
- {SVP64 Chapter} contains the packet-format *only*, the {SVP64 Appendix} contains explanations and further details
- [[sv/svp64-single]] still under development
- {SVP64 Quirks} things in SVP64 that slightly break the rules or are not immediately apparent despite the RISC paradigm
- {SVP64 Augmentation Table} autogenerated table of SVP64 decoder augmentation
- {SPRs} SPRs
- [[sv/rfc]] RFCs to the OPF ISA WG

SVP64 "Modes":

- For condition register operations see {Condition Register Fields Mode} - SVP64 Condition Register ops: Guidelines on Vectorisation of any v3.0B base operations which return or modify a Condition Register bit or field.
- For LD/ST Modes, see {Load/Store Mode}.
- For Branch modes, see {Branch Mode} - SVP64 Conditional Branch behaviour: All/Some Vector CRs
- For arithmetic and logical, see {Arithmetic Mode}
- {Pack / Unpack} pack/unpack move to and from vec2/3/4, actually an RM.EXTRA Mode and a {REMAP subsystem} mode

Core SVP64 instructions:

- {setvl instruction} the Cray-style "Vector Length" instruction
- svremap, svindex and svshape: part of {REMAP subsystem} "Remapping" for Matrix Multiply, DCT/FFT and RGB-style "Structure Packing" as well as general-purpose Indexing. Also describes associated SPRs.
- {svstep instruction} Key stepping instruction, primarily for Vertical-First Mode and also providing traditional "Vector Iota" capability.

Please note: there are only six instructions in the whole of SV. Beyond this point are additional ***Scalar*** *instructions related to specific workloads that have nothing to do with the SV Specification*

2.3 Stability Guarantees in Simple-V

Providing long-term stability in an ISA is extremely challenging but critically important. It requires certain guarantees to be provided.

- Firstly: that instructions will never be ambiguously-defined.
- Secondly, that no instruction shall change meaning to produce different results on different hardware (present or future).
- Thirdly, that Scalar "defined words" (32 bit instruction encodings) if Vectorised will also always be implemented as identical Scalar instructions (the sole semi-exception being Vectorised Branch-Conditional)
- Fourthly, that implementors are not permitted to either add arbitrary features nor implement features in an incompatible way. *(Performance may differ, but differing results are not permitted).*
- Fifthly, that any part of Simple-V not implemented by a lower Compliancy Level is *required* to raise an illegal instruction trap (allowing soft-emulation), including if Simple-V is not implemented at all.
- Sixthly, that any `UNDEFINED` behaviour for practical implementation reasons is clearly documented for both programmers and hardware implementors.

In particular, given the strong recent emphasis and interest in "Scalable Vector" ISAs, it is most unfortunate that both ARM SVE and RISC-V RVV permit the exact same instruction to produce different results on different hardware depending on a "Silicon Partner" hardware choice. This choice catastrophically and irrevocably causes binary non-interoperability *despite being a "feature"*. Explained in https://m.youtube.com/watch?v=HNEm8zmkjBU it is the exact same binary-incompatibility issue faced by Power ISA on its 32- to 64-bit transition: 32-bit hardware was **unable** to trap-and-emulate 64-bit binaries because the opcodes were (are) the same.

It is therefore *guaranteed* that extensions to the register file width and quantity in Simple-V shall only be made in future by explicit means, ensuring binary compatibility.

2.4 Optional Scalar instructions

Additional Instructions for specific purposes (not SVP64)

All of these instructions below have nothing to do with SV. They are all entirely designed as Scalar instructions that, as Scalar instructions, stand on their own merit. Considerable lengths have been made to provide justifications for each of these *Scalar* instructions in a *Scalar* context, completely independently of SVP64.

Some of these Scalar instructions happen also designed to make Scalable Vector binaries more efficient, such as the crweird group. Others are to bring the Scalar Power ISA up-to-date within specific workloads, such as a JavaScript Rounding instruction (which saves 32 scalar instructions including seven branch instructions). None of them are strictly necessary but performance and power consumption may be (or, is already) compromised in certain workloads and use-cases without them.

Vector-related but still Scalar:

- {Swizzle Move} vec2/3/4 Swizzles (RGBA, XYZW) for 3D and CUDA. designed as a Scalar instruction.
- {SV Vector ops} scalar operations needed for supporting vectors
- {CR Weird ops} scalar instructions needed for effective predication

Stand-alone Scalar Instructions:

- {Bitmanip ops}
- [[sv/fcvt]] FP Conversion (due to OpenPOWER Scalar FP32)
- {FP Class ops} detect class of FP numbers
- {FP/Int Conversion ops} Move and convert GPR <-> FPR, needed for !VSX
- {Audio and Video Opcodes} scalar opcodes for Audio/Video
- [[prefix_codes]] Decode/encode prefix-codes, used by JPEG, DEFLATE, etc.
- TODO: OpenPOWER adaptation {Transcendentals}

Twin targetted instructions (two registers out, one implicit, just like Load-with-Update).

- {Fixed Point pseudocode}
- {Floating Point pseudocode}
- {Big Integer} Operations that help with big arithmetic

Explanation of the rules for twin register targets (implicit RS, FRS) explained in SVP64 {SVP64 Appendix}

2.5 Architectural Note

This section is primarily for the ISA Working Group and for IBM in their capacity and responsibility for allocating "Architectural Resources" (opcodes), but it is also useful for general understanding of Simple-V.

Simple-V is effectively a type of "Zero-Overhead Loop Control" to which an entire 24 bits are exclusively dedicated in a fully RISC-abstracted manner. Within those 24-bits there are no Scalar instructions, and no Vector instructions: there is *only* "Loop Control".

This is why there are no actual Vector operations in Simple-V: *all* suitable Scalar Operations are Vectorised or not at all. This has some extremely important implications when considering adding new instructions, and especially when allocating the Opcode Space for them. To protect SVP64 from damage, a "Hard Rule" has to be set:

```
Scalar Instructions must be simultaneously added in the corresponding
SVP64 opcode space with the exact same 32-bit "Defined Word" or they
must not be added at all.  Likewise, instructions planned for addition
```

```
in what is considered (wrongly) to be the exclusive "Vector" domain
must correspondingly be added in the Scalar space with the exact same
32-bit "Defined Word", or they must not be added at all.
```

Some explanation of the above is needed. Firstly, "Defined Word" is a term used in Section 1.6.3 of the Power ISA v3 1 Book I: it means, in short, "a 32 bit instruction", which can then be Prefixed by EXT001 to extend it to 64-bit (named EXT100-163). Prefixed-Prefixed (96-bit Variable-Length) encodings are prohibited in v3.1 and they are just as prohibited in Simple-V: it's too complex in hardware. This means that **only** 32-bit "Defined Words" may be Vectorised, and in particular it means that no 64-bit instruction (EXT100-163) may **ever** be Vectorised.

Secondly, the term "Vectoriseable" was used. This refers to "instructions which if SVP64-Prefixed are actually meaningful". `sc` is meaningless to Vectorise, for example, as is `sync` and `mtmsr` (there is only ever going to be one MSR).

The problem comes if the rationale is applied, "if unused, Unvectoriseable opcodes can therefore be allocated to alternative instructions mixing inside the SVP64 Opcode space", which unfortunately results in huge inadviseable complexity in HDL at the Decode Phase, attempting to discern between the two types. Worse than that, if the alternate 64-bit instruction is Vectoriseable but the 32-bit Scalar "Defined Word" is already allocated, how can there ever be a Scalar version of the alternate instruction? It would have to be added as a **completely different** 32-bit "Defined Word", and things go rapidly downhill in the Decoder as well as the ISA from there.

Therefore to avoid risk and long-term damage to the Power ISA:

- *even Unvectoriseable* "Defined Words" (`mtmsr`) must have the corresponding SVP64 Prefixed Space `RESERVED`, permanently requiring Illegal Instruction to be raised (the 64-bit encoding corresponding to an illegal `sv.mtmsr` if ever incorrectly attempted must be **defined** to raise an Exception)
- *Even instructions that may not be Scalar* (although for various practical reasons this is extremely rare if not impossible, if not just generally "strongly discouraged") which have no meaning or use as a 32-bit Scalar "Defined Word", **must** still have the Scalar "Defined Word" `RESERVED` in the scalar opcode space, as an Illegal Instruction.

A good example of the former is `mtmsr` because there is only one MSR register (`sv.mtmsr` is meaningless, as is `sv.sc`), and a good example of the latter is [[sv/mv.x]] which is so deeply problematic to add to any Scalar ISA that it was rejected outright and an alternative route taken (Indexed REMAP).

Another good example would be Cross Product which has no meaning at all in a Scalar ISA (Cross Product as a concept only applies to Mathematical Vectors). If any such Vector operation were ever added, it would be **critically** important to reserve the exact same *Scalar* opcode with the exact same "Defined Word" in the *Scalar* Power ISA opcode space, as an Illegal Instruction. There are good reasons why Cross Product has not been proposed, but it serves to illustrate the point as far as Architectural Resource Allocation is concerned.

Bottom line is that whilst this seems wasteful the alternatives are a destabilisation of the Power ISA and impractically-complex Hardware Decoders. With the Scalar Power ISA (v3.0, v3.1) already being comprehensive in the number of instructions, keeping further Decode complexity down is a high priority.

2.6 Other Scalable Vector ISAs

These Scalable Vector ISAs are listed to aid in understanding and context of what is involved.

- Original Cray ISA http://www.bitsavers.org/pdf/cray/CRAY_Y-MP/HR-04001-0C_Cray_Y-MP_Computer_Systems_Functional_Description_Jun90.pdf
- NEC SX Aurora (still in production, inspired by Cray) https://www.hpc.nec/documents/guide/pdfs/Aurora_ISA_guide.pdf
- RISC-V RVV (inspired by Cray) https://github.com/riscv/riscv-v-spec
- MRISC32 ISA Manual (under active development) https://github.com/mrisc32/mrisc32/tree/master/isa-manual
- Mitch Alsup's MyISA 66000 Vector Processor ISA Manual is available from Mitch on request.

A comprehensive list of 3D GPU, Packed SIMD, Predicated-SIMD and true Scalable Vector ISAs may be found at the {Vector ISA Comparison} page. Note: AVX-512 and SVE2 are *not Vector ISAs*, they are Predicated-SIMD. *Public discussions have taken place at Conferences attended by both Intel and ARM on adding a* `setvl` *instruction which would easily make both AVX-512 and SVE2 truly "Scalable".* {ISA Comparison Table} in tabular form.

2.7 Major opcodes summary

Simple-V itself only requires six instructions with 6-bit Minor XO (bits 26-31), and the SVP64 Prefix Encoding requires 25% space of the EXT001 Major Opcode. There are **no** Vector Instructions and consequently **no further opcode space is required**. Even though they are currently placed in the EXT022 Sandbox, the "Management" instructions (setvl, svstep, svremap, svshape, svindex) are designed to fit cleanly into EXT019 (exactly like `addpcis`) or other 5/6-bit Minor XO area (bits 25-31) that has space for Rc=1.

That said: for the target workloads for which Scalable Vectors are typically used, the Scalar ISA on which those workloads critically rely is somewhat anaemic. The Libre-SOC Team has therefore been addressing that by developing a number of Scalar instructions in specialist areas (Big Integer, Cryptography, 3D, Audio/Video, DSP) and it is these which require considerable Scalar opcode space.

Please be advised that even though SV is entirely DRAFT status, there is considerable concern that because there is not yet any two-way day-to-day communication established with the OPF ISA WG, we have no idea if any of these are conflicting with future plans by any OPF Members. **The External ISA WG RFC Process has now been ratified but Libre-SOC may not join the OPF as an entity because it does not exist except in name. Even if it existed it would be a conflict of interest to join the OPF, due to our funding remit from NLnet**. We therefore proceed on the basis of making public the intention to submit RFCs once the External ISA WG RFC Process is in place and, in a wholly unsatisfactory manner have to *hope and trust* that OPF ISA WG Members are reading this and take it into consideration.

Scalar Summary

As in above sections, it is emphasised strongly that Simple-V in no way critically depends on the 100 or so *Scalar* instructions also being developed by Libre-SOC.

None of these Draft opcodes are intended for private custom secret proprietary usage. They are all intended for entirely public, upstream, high-profile mass-volume day-to-day usage at the same level as add, popcnt and fld

- bitmanip requires two major opcodes (due to 16+ bit immediates) those are currently EXT022 and EXT05.
- brownfield encoding in one of those two major opcodes still requires multiple VA-Form operations (in greater numbers than EXT04 has spare)
- space in EXT019 next to addpcis and crops is recommended (or any other 5-6 bit Minor XO areas)
- many X-Form opcodes currently in EXT022 have no preference for a location at all, and may be moved to EXT059, EXT019, EXT031 or other much more suitable location.
- even if ratified and even if the majority (mostly X-Form) is moved to other locations, the large immediate sizes of the remaining bitmanip instructions means it would be highly likely these remaining instructions would need two major opcodes. Fortuitously the v3.1 Spec states that both EXT005 and EXT009 are available.

Additional observations

Note that there is no Sandbox allocation in the published ISA Spec for v3.1 EXT01 usage, and because SVP64 is already 64-bit Prefixed, Prefixed-Prefixed-instructions (SVP64 Prefixed v3.1 Prefixed) would become a whopping 96-bit long instruction. Avoiding this situation is a high priority which in turn by necessity puts pressure on the 32-bit Major Opcode space.

Note also that EXT022, the Official Architectural Sandbox area available for "Custom non-approved purposes" according to the Power ISA Spec, is under severe design pressure as it is insufficient to hold the full extent of the instruction additions required to create a Hybrid 3D CPU-VPU-GPU. Although the wording of the Power ISA Specification leaves open the *possibility* of not needing to propose ISA Extensions to the ISA WG, it is clear

that EXT022 is an inappropriate location for a large high-profile Extension intended for mass-volume product deployment. Every in-good-faith effort will therefore be made to work with the OPF ISA WG to submit SVP64 via the External RFC Process.

Whilst SVP64 is only 6 instructions the heavy focus on VSX for the past 12 years has left the SFFS Level anaemic and out-of-date compared to ARM and x86. This is very much a blessing, as the Scalar ISA has remained clean, making it highly suited to RISC-paradigm Scalable Vector Prefixing. Approximately 100 additional (optional) Scalar Instructions are up for proposal to bring SFFS up-to-date. None of them require or depend on PackedSIMD VSX (or VMX).

2.8 Other

Examples experiments future ideas discussion:

- Scalar register access above r31 and CR7.
- [[sv/propagation]] Context propagation including svp64, swizzle and remap
- [[sv/masked_vector_chaining]]
- [[sv/discussion]]
- [[sv/example_dep_matrices]]
- [[sv/major_opcode_allocation]]
- [[sv/byteswap]]
- [[sv/16_bit_compressed]] experimental
- [[sv/toc_data_pointer]] experimental
- [[sv/predication]] discussion on predication concepts
- [[sv/register_type_tags]]
- [[sv/mv.x]] deprecated in favour of Indexed REMAP

Additional links:

- https://www.sigarch.org/simd-instructions-considered-harmful/
- {Vector ISA Comparison} - a list of Packed SIMD, GPU, and other Scalable Vector ISAs
- {ISA Comparison Table} - a one-off (experimental) table comparing ISAs
- [[simple_v_extension]] old (deprecated) version
- [[openpower/sv/llvm]]

Chapter 3

Other Vector ISAs

[[!tag standards]]

3.1 Comparative analysis

These are all, deep breath, basically... required reading, *as well as and in addition* to a full and comprehensive deep technical understanding of the Power ISA, in order to understand the depth and background on SVP64 as a 3D GPU and VPU Extension.

I am keenly aware that each of them is 300 to 1,000 pages (just like the Power ISA itself).

This is just how it is.

Given the sheer overwhelming size and scope of SVP64 we have gone to **considerable lengths** to provide justification and rationalisation for adding the various sub-extensions to the Base Scalar Power ISA.

- Scalar bitmanipulation is justifiable for the exact same reasons the extensions are justifiable for other ISAs. The additional justification for their inclusion where some instructions are already (sort-of) present in VSX is that VSX is not mandatory, and the complexity of implementation of VSX is too high a price to pay at the Embedded SFFS Compliancy Level.
- Scalar FP-to-INT conversions, likewise. ARM has a javascript conversion instruction, Power ISA does not (and it costs a ridiculous 45 instructions to implement, including 6 branches!)
- Scalar Transcendentals (SIN, COS, ATAN2, LOG) are easily justifiable for High-Performance Compute workloads.

It also has to be pointed out that normally this work would be covered by multiple separate full-time Workgroups with multiple Members contributing their time and resources. In RISC-V there are over sixty Technical Working Groups https://riscv.org/community/directory-of-working-groups/

Overall the contributions that we are developing take the Power ISA out of the specialist highly-focussed market it is presently best known for, and expands it into areas with much wider general adoption and broader uses.

OpenCL specifications are linked here, these are relevant when we get to a 3D GPU / High Performance Compute ISA WG RFC: {Transcendentals}

(Failure to add Transcendentals to a 3D GPU is directly equivalent to *willfully* designing a product that is 100% destined for commercial rejection, due to the extremely high competitive performance/watt achieved by today's mass-volume GPUs.)

I mention these because they will be encountered in every single commercial GPU ISA, but they're not part of the "Base" (core design) of a Vector Processor. Transcendentals can be added as a sub-RFC.

3.2 SIMD ISAs commonly mistaken for Vector

There is considerable confusion surrounding Vector ISAs because of a mis-use of the word "Vector" in the marketing material of most well-known Packed SIMD ISAs of the past 3 decades. These Packed SIMD ISAs used features "inspired" from Scalable Vector ISAs.

- PackedSIMD VSX. VSX, which has the word "Vector" in its name, is "inspired" by Vector Processing but has no "Scaling" capability, and no Predicate masking. Both these factors put pressure on developers to use "inline assembler unrolling" and data repetition, which in turn is detrimental to both L1 Data and Instruction Caches. Adding Predicate Masks to the PackedSIMD VSX ISA would effectively double the number of PackedSIMD instructions (750 becomes 1,500) even if it were practical to do so (no available 32 bit encoding space).
- AVX / AVX2 / AVX128 / AVX256 / AVX512 again has the word "Vector" in its name but this in no way makes it a Vector ISA. None of the AVX-* family are "Scalable" however there is at least Predicate Masking in AVX-512.
- ARM NEON - accurately described as a Packed SIMD ISA in all literature.
- ARM SVE / SVE2 - **not a Scalable Vector ISA**, it is actually a hybrid PackedSIMD/PredicatedSIMD ISA: with 4-operand instructions being overwrite to fit into 32-bit there was no room for a predicate mask. The "Scaling" is, rather unfortunately, a parameter that is chosen by the *Hardware Architect*, rather than the programmer. The actual "Scalar" part as far as the programmer is concerned is supposed to be the Predicate Masks. However in practice, ARM NEON programmers have found it too hard to adapt and have instead attempted to fit the NEON SIMD paradigm on top of SVE. This has resulted in programmers writing **multiple variants** of near-identical hand-coded assembler in order to target different machines with different hardware widths, going directly against the advice given on ARM's developer documentation.

A good analogy explaining why "Silicon-Partner Scalability" is catastrophic is to note that the situation is near-identical to when IBM extended Power ISA from 32 to 64-bit. Existing 32-bit systems were **unable** to run or trap-and-emulate 64-bit instructions **because they were the exact same opcodes** and the "Silicon Scalability" of both RVV and ARM SVE/2 is the exact same mistake, but much worse. At least IBM provided an `MSR.SF` bit.

The saving grace of PackedSIMD VSX is that it did not fall to the seduction outlined in the "SIMD Considered Harmful" article https://www.sigarch.org/simd-instructions-considered-harmful/. It is clear that it is expected to deploy Multi-Issue to achieve high performance, which is a much cleaner approach that has not resulted in ISA poisoning such as that suffered by x86 (AVX).

3.3 Actual 3D GPU Architectures and ISAs (all SIMD)

All of these are not Scalable Vector ISAs, they are SIMD ISAs.

- Broadcom Videocore https://github.com/hermanhermitage/videocoreiv
- Etnaviv https://github.com/etnaviv/etna_viv/tree/master/doc
- Nyuzi http://www.cs.binghamton.edu/~millerti/nyuziraster.pdf
- MALI https://github.com/cwabbott0/mali-isa-docs
- AMD https://developer.amd.com/wp-content/resources/RDNA_Shader_ISA.pdf https://developer.amd.com/wp-content/resources/Vega_Shader_ISA_28July2017.pdf
- MIAOW which is *NOT* a 3D GPU, it is a processor which happens to implement a subset of the AMDGPU ISA (Southern Islands), aka a "GPGPU" https://miaowgpu.org/

3.4 Actual Scalar Vector Processor Architectures and ISAs

- NEC SX Aurora https://www.hpc.nec/documents/guide/pdfs/Aurora_ISA_guide.pdf
- Cray ISA http://www.bitsavers.org/pdf/cray/CRAY_Y-MP/HR-04001-0C_Cray_Y-MP_Computer_Systems_Functional_Description_Jun90.pdf

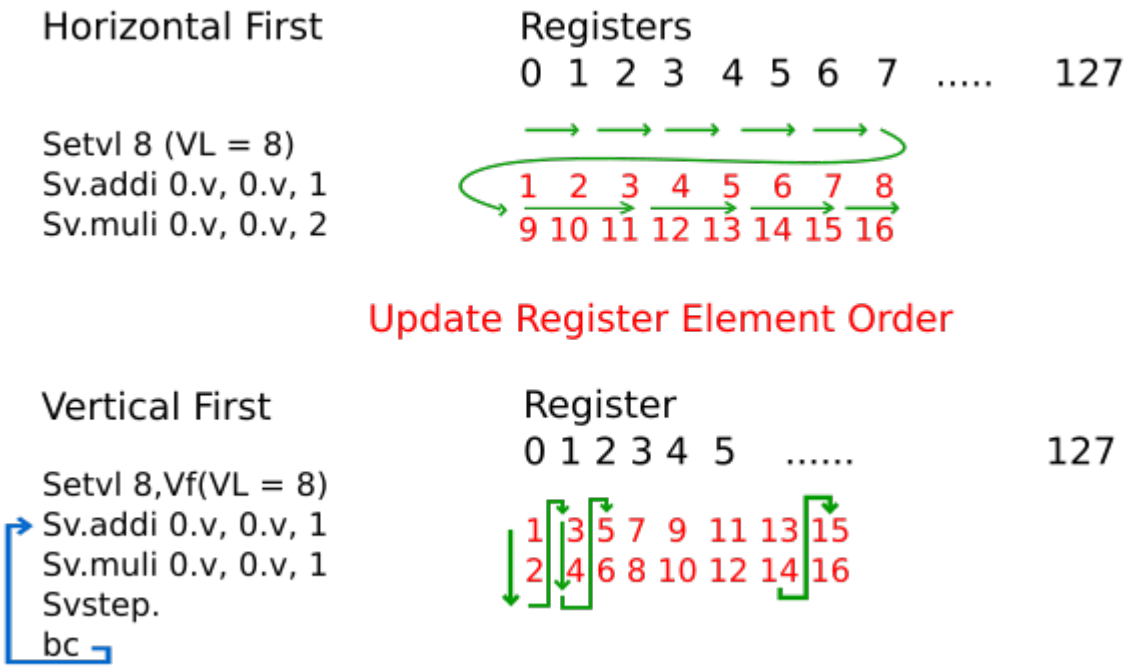

Figure 3.1: Horizontal vs Vertical

- RISC-V RVV https://github.com/riscv/riscv-v-spec
- MRISC32 ISA Manual (under active development) https://github.com/mrisc32/mrisc32/tree/master/isa-manual
- Mitch Alsup's MyISA 66000 Vector Processor ISA Manual is available from Mitch under NDA on direct contact with him. It is a different approach from the others, which may be termed "Cray-Style Horizontal-First" Vectorisation. 66000 is a *Vertical-First* Vector ISA with hardware-level auto-vectorisation.
- ETA-10 an extremely rare Scalable Vector Architecture from 1986, similar to the CDC Cyber 205. Only 25 machines were ever delivered. Page 3-220 of its ISA shows that it had Predicate Masks and Horizontal Reduction. Appendix H-1 shows it is likely a Memory-to-Memory Vector Architecture, and overcame the penalties normally associated with this by adding an explicit "Vector operand forwarding/chaining" instruction (Page 3-69). It is however clearly Scalable, up to Vector elements of 2^16.

The term Horizontal or Vertical alludes to the Matrix "Row-First" or "Column-First" technique, where:

- Horizontal-First processes all elements in a Vector before moving on to the next instruction
- Vertical-First processes *ONE* element per instruction, and requires loop constructs to explicitly step to the next element.

Vector-type Support by Architecture

Architecture	Horizontal	Vertical
MyISA 66000		X
Cray	X	
SX Aurora	X	
RVV	X	
SVP64	X	X

Chapter 4

Overview

4.1 SV Overview

SV is in DRAFT STATUS. SV has not yet been submitted to the OpenPOWER Foundation ISA WG for review.

This document provides an overview and introduction as to why SV (a [[!wikipedia Cray]]-style Vector augmentation to [[!wikipedia OpenPOWER]]) exists, and how it works.

Sponsored by NLnet under the Privacy and Enhanced Trust Programme

Links:

- This page: http://libre-soc.org/openpower/sv/overview
- FOSDEM2021 SimpleV for Power ISA
- FOSDEM2021 presentation https://www.youtube.com/watch?v=FS6tbfyb2VA
- [[discussion]] and bugreport feel free to add comments, questions.
- {Scalable Vectors for Power ISA}
- {SVP64 Chapter}
- x86 REP instruction: a useful way to quickly understand that the core of the SV concept is not new.
- Article about register tagging showing that tagging is not a new idea either. Register tags are also used in the Mill Architecture.

Table of contents:

[[!toc]]

4.2 Introduction: SIMD and Cray Vectors

SIMD, the primary method for easy parallelism of the past 30 years in Computer Architectures, is known to be harmful. SIMD provides a seductive simplicity that is easy to implement in hardware. With each doubling in width it promises increases in raw performance without the complexity of either multi-issue or out-of-order execution.

Unfortunately, even with predication added, SIMD only becomes more and more problematic with each power of two SIMD width increase introduced through an ISA revision. The opcode proliferation, at O(N^6), inexorably spirals out of control in the ISA, detrimentally impacting the hardware, the software, the compilers and the testing and compliance. Here are the typical dimensions that result in such massive proliferation, based on mass-volume DSPs and Micro-Processors:

- Operation (add, mul)

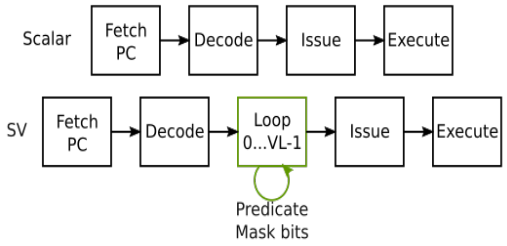

Figure 4.1: Single-Issue concept

- bitwidth (8, 16, 32, 64, 128)
- Conversion between bitwidths (FP16-FP32-64)
- Signed/unsigned
- HI/LO swizzle (Audio L/R channels)
- HI/LO selection on src 1
- selection on src 2
- selection on dest
- Example: AndesSTAR Audio DSP
- Saturation (Clamping at max range)

These typically are multiplied up to produce explicit opcodes numbering in the thousands on, for example the ARC Video/DSP cores.

Cray-style variable-length Vectors on the other hand result in stunningly elegant and small loops, exceptionally high data throughput per instruction (by one *or greater* orders of magnitude than SIMD), with no alarmingly high setup and cleanup code, where at the hardware level the microarchitecture may execute from one element right the way through to tens of thousands at a time, yet the executable remains exactly the same and the ISA remains clear, true to the RISC paradigm, and clean. Unlike in SIMD, powers of two limitations are not involved in the ISA or in the assembly code.

SimpleV takes the Cray style Vector principle and applies it in the abstract to a Scalar ISA in the same way that x86 used to do its "REP" instruction. In the process, "context" is applied, allowing amongst other things a register file size increase using "tagging" (similar to how x86 originally extended registers from 32 to 64 bit).

4.2.1 SV

The fundamentals are (just like x86 "REP"):

- The Program Counter (PC) gains a "Sub Counter" context (Sub-PC)
- Vectorisation pauses the PC and runs a Sub-PC loop from 0 to VL-1 (where VL is Vector Length)
- The [[Program Order]] of "Sub-PC" instructions must be preserved, just as is expected of instructions ordered by the PC.
- Some registers may be "tagged" as Vectors
- During the loop, "Vector"-tagged register are incremented by one with each iteration, executing the *same instruction* but with *different registers*
- Once the loop is completed *only then* is the Program Counter allowed to move to the next instruction.

Hardware (and simulator) implementors are free and clear to implement this as literally a for-loop, sitting in between instruction decode and issue. Higher performance systems may deploy SIMD backends, multi-issue and out-of-order execution, although it is strongly recommended to add predication capability directly into SIMD backend units.

A typical Cray-style Scalable Vector ISA (where a SIMD one has a fixed non-negotiable static parameter instead of a runtime-dynamic VL) performs its arithmetic as:

```
for i = 0 to VL-1:
```

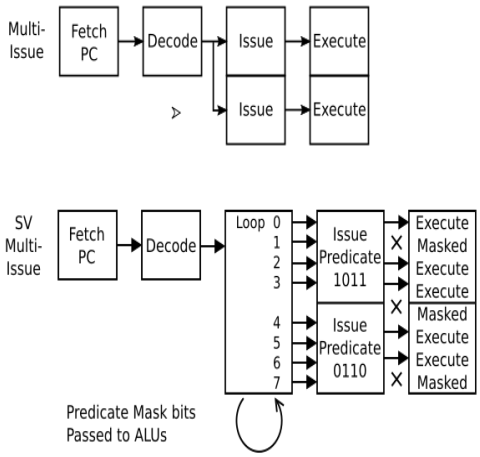

Figure 4.2: Multi-Issue with Predicated SIMD back-end ALUs

```
VPR(RT)[i] = VPR[RA][i] + VPR(RB)[i]
```

In Power ISA v3.0B pseudo-code form, an ADD operation in Simple-V, assuming both source and destination have been "tagged" as Vectors, is simply:

```
for i = 0 to VL-1:
     GPR(RT+i) = GPR(RA+i) + GPR(RB+i)
```

At its heart, SimpleV really is this simple. On top of this fundamental basis further refinements can be added which build up towards an extremely powerful Vector augmentation system, with very little in the way of additional opcodes required: simply external "context".

x86 was originally only 80 instructions: prior to AVX512 over 1,300 additional instructions have been added, almost all of them SIMD.

RISC-V RVV as of version 0.9 is over 188 instructions (more than the rest of RV64G combined: 80 for RV64G and 27 for C). Over 95% of that functionality is added to Power v3.0B, by SimpleV augmentation, with around 5 to 8 instructions.

Even in Power ISA v3.0B, the Scalar Integer ISA is around 150 instructions, with IEEE754 FP adding approximately 80 more. VSX, being based on SIMD design principles, adds somewhere in the region of 600 more. SimpleV again provides over 95% of VSX functionality, simply by augmenting the *Scalar* Power ISA, and in the process providing features such as predication, which VSX is entirely missing.

AVX512, SVE2, VSX, RVV, all of these systems have to provide different types of register files: Scalar and Vector is the minimum. AVX512 even provides a mini mask regfile, followed by explicit instructions that handle operations on each of them *and map between all of them*. SV simply not only uses the existing scalar regfiles (including CRs), but because operations exist within Power ISA to cover interactions between the scalar regfiles (`mfcr`, `fcvt`) there is very little that needs to be added.

In fairness to both VSX and RVV, there are things that are not provided by SimpleV:

- 128 bit or above arithmetic and other operations (VSX Rijndael and SHA primitives; VSX shuffle and bitpermute operations)
- register files above 128 entries
- Vector lengths over 64
- 32-bit instruction lengths. {SVP64 Chapter} had to be added as 64 bit.

These limitations, which stem inherently from the adaptation process of starting from a Scalar ISA, are not insurmountable. Over time, they may well be addressed in future revisions of SV.

The rest of this document builds on the above simple loop to add:

- Vector-Scalar, Scalar-Vector and Scalar-Scalar operation (of all register files: Integer, FP *and CRs*)
- Traditional Vector operations (VSPLAT, VINSERT, VCOMPRESS etc)
- Predication masks (essential for parallel if/else constructs)
- 8, 16 and 32 bit integer operations, and both FP16 and BF16.
- Compacted operations into registers (normally only provided by SIMD)
- Fail-on-first (introduced in ARM SVE2)
- A new concept: Data-dependent fail-first
- A completely new concept: "Twin Predication"
- vec2/3/4 "Subvectors" and Swizzling (standard fare for 3D)

All of this is *without modifying the Power v3.0B ISA*, except to add "wrapping context", similar to how v3.1B 64 Prefixes work.

4.3 Adding Scalar / Vector

The first augmentation to the simple loop is to add the option for all source and destinations to all be either scalar or vector. As a FSM this is where our "simple" loop gets its first complexity.

```
function op_add(RT, RA, RB) # add not VADD!
  int id=0, irs1=0, irs2=0;
  for i = 0 to VL-1:
    ireg[RT+id] <= ireg[RA+irs1] + ireg[RB+irs2];
    if (!RT.isvec) break;
    if (RT.isvec)  { id += 1; }
    if (RA.isvec)  { irs1 += 1; }
    if (RB.isvec)  { irs2 += 1; }
```

This could have been written out as eight separate cases: one each for when each of `RA`, `RB` or `RT` is scalar or vector. Those eight cases, when optimally combined, result in the pseudocode above.

With some walkthroughs it is clear that the loop exits immediately after the first scalar destination result is written, and that when the destination is a Vector the loop proceeds to fill up the register file, sequentially, starting at `RT` and ending at `RT+VL-1`. The two source registers will, independently, either remain pointing at `RB` or `RA` respectively, or, if marked as Vectors, will march incrementally in lockstep, producing element results along the way, as the destination also progresses through elements.

In this way all the eight permutations of Scalar and Vector behaviour are covered, although without predication the scalar-destination ones are reduced in usefulness. It does however clearly illustrate the principle.

Note in particular: there is no separate Scalar add instruction and separate Vector instruction and separate Scalar-Vector instruction, *and there is no separate Vector register file*: it's all the same instruction, on the standard register file, just with a loop. Scalar happens to set that loop size to one.

The important insight from the above is that, strictly speaking, Simple-V is not really a Vectorisation scheme at all: it is more of a hardware ISA "Compression scheme", allowing as it does for what would normally require multiple sequential instructions to be replaced with just one. This is where the rule that Program Order must be preserved in Sub-PC execution derives from. However in other ways, which will emerge below, the "tagging" concept presents an opportunity to include features definitely not common outside of Vector ISAs, and in that regard it's definitely a class of Vectorisation.

4.3.1 Register "tagging"

As an aside: in {SVP64 Chapter} the encoding which allows SV to both extend the range beyond r0-r31 and to determine whether it is a scalar or vector is encoded in two to three bits, depending on the instruction.

The reason for using so few bits is because there are up to *four* registers to mark in this way (`fma`, `isel`) which starts to be of concern when there are only 24 available bits to specify the entire SV Vectorisation Context. In fact, for a small subset of instructions it is just not possible to tag every single register. Under these rare circumstances a tag has to be shared between two registers.

Below is the pseudocode which expresses the relationship which is usually applied to *every* register:

```
if extra3_mode:
    spec = EXTRA3 # bit 2 s/v, 0-1 extends range
else:
    spec = EXTRA2 << 1 # same as EXTRA3, shifted
if spec[2]: # vector
     RA.isvec = True
     return (RA << 2) | spec[0:1]
else:         # scalar
     RA.isvec = False
     return (spec[0:1] << 5) | RA
```

Here we can see that the scalar registers are extended in the top bits, whilst vectors are shifted up by 2 bits, and then extended in the LSBs. Condition Registers have a slightly different scheme, along the same principle, which takes into account the fact that each CR may be bit-level addressed by Condition Register operations.

Readers familiar with the Power ISA will know of Rc=1 operations that create an associated post-result "test", placing this test into an implicit Condition Register. The original researchers who created the POWER ISA chose CR0 for Integer, and CR1 for Floating Point. These *also become Vectorised* - implicitly - if the associated destination register is also Vectorised. This allows for some very interesting savings on instruction count due to the very same CR Vectors being predication masks.

4.4 Adding single predication

The next step is to add a single predicate mask. This is where it gets interesting. Predicate masks are a bitvector, each bit specifying, in order, whether the element operation is to be skipped ("masked out") or allowed. If there is no predicate, it is set to all 1s, which is effectively the same as "no predicate".

```
function op_add(RT, RA, RB) # add not VADD!
  int id=0, irs1=0, irs2=0;
  predval = get_pred_val(FALSE, RT); # dest mask
  for i = 0 to VL-1:
    if (predval & 1<<i) # predication bit test
       ireg[RT+id] <= ireg[RA+irs1] + ireg[RB+irs2];
       if (!RT.isvec) break;
    if (RT.isvec)  { id += 1; }
    if (RA.isvec)  { irs1 += 1; }
    if (RB.isvec)  { irs2 += 1; }
```

The key modification is to skip the creation and storage of the result if the relevant predicate mask bit is clear, but *not the progression through the registers.*

A particularly interesting case is if the destination is scalar, and the first few bits of the predicate are zero. The loop proceeds to increment the Vector *source* registers until the first nonzero predicate bit is found, whereupon a single *Scalar* result is computed, and *then* the loop exits. This in effect uses the predicate to perform *Vector source indexing.* This case was not possible without the predicate mask. Also, interestingly, the predicate mode `1<<r3` is specifically provided as a way to select one single entry from a Vector.

If all three registers are marked as Vector then the "traditional" predicated Vector behaviour is provided. Yet, just as before, all other options are still provided, right the way back to the pure-scalar case, as if this were a straight Power ISA v3.0B non-augmented instruction.

Single Predication therefore provides several modes traditionally seen in Vector ISAs:

- VINSERT: the predicate may be set as a single bit, the sources are scalar and the destination a vector.
- VSPLAT (result broadcasting) is provided by making the sources scalar and the destination a vector, and having no predicate set or having multiple bits set.
- VSELECT is provided by setting up (at least one of) the sources as a vector, using a single bit in the predicate, and the destination as a scalar.

All of this capability and coverage without even adding one single actual Vector opcode, let alone 180, 600 or 1,300!

4.5 Predicate "zeroing" mode

Sometimes with predication it is ok to leave the masked-out element alone (not modify the result) however sometimes it is better to zero the masked-out elements. Zeroing can be combined with bit-wise ORing to build up vectors from multiple predicate patterns: the same combining with nonzeroing involves more mv operations and predicate mask operations. Our pseudocode therefore ends up as follows, to take the enhancement into account:

```
function op_add(RT, RA, RB) # add not VADD!
  int id=0, irs1=0, irs2=0;
  predval = get_pred_val(FALSE, RT); # dest pred
  for i = 0 to VL-1:
    if (predval & 1<<i) # predication bit test
       ireg[RT+id] <= ireg[RA+irs1] + ireg[RB+irs2];
       if (!RT.isvec) break;
    else if zeroing:   # predicate failed
       ireg[RT+id] = 0 # set element  to zero
    if (RT.isvec)  { id += 1; }
    if (RA.isvec)  { irs1 += 1; }
    if (RB.isvec)  { irs2 += 1; }
```

Many Vector systems either have zeroing or they have nonzeroing, they do not have both. This is because they usually have separate Vector register files. However SV sits on top of standard register files and consequently there are advantages to both, so both are provided.

4.6 Element Width overrides

All good Vector ISAs have the usual bitwidths for operations: 8/16/32/64 bit integer operations, and IEEE754 FP32 and 64. Often also included is FP16 and more recently BF16. The *really* good Vector ISAs have variable-width vectors right down to bitlevel, and as high as 1024 bit arithmetic per element, as well as IEEE754 FP128.

SV has an "override" system that *changes* the bitwidth of operations that were intended by the original scalar ISA designers to have (for example) 64 bit operations (only). The override widths are 8, 16 and 32 for integer, and FP16 and FP32 for IEEE754 (with BF16 to be added in the future).

This presents a particularly intriguing conundrum given that the Power Scalar ISA was never designed with for example 8 bit operations in mind, let alone Vectors of 8 bit.

The solution comes in terms of rethinking the definition of a Register File. The typical regfile may be considered to be a multi-ported SRAM block, 64 bits wide and usually 32 entries deep, to give 32 64 bit registers. In c this would be:

```
typedef uint64_t reg_t;
reg_t int_regfile[32]; // standard scalar 32x 64bit
```

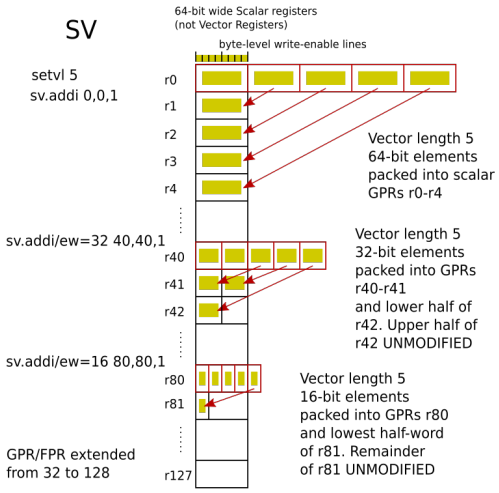

Figure 4.3: image

Conceptually, to get our variable element width vectors, we may think of the regfile as instead being the following c-based data structure, where all types uint16_t etc. are in little-endian order:

```
#pragma(packed)
typedef union {
    uint8_t  actual_bytes[8];
    uint8_t  b[0]; // array of type uint8_t
    uint16_t s[0]; // array of LE ordered uint16_t
    uint32_t i[0];
    uint64_t l[0]; // default Power ISA uses this
} reg_t;

reg_t int_regfile[128]; // SV extends to 128 regs
```

This means that Vector elements start from locations specified by 64 bit "register" but that from that location onwards the elements *overlap subsequent registers*.

Here is another way to view the same concept, bearing in mind that it is assumed a LE memory order:

```
uint8_t reg_sram[8*128];
uint8_t *actual_bytes = &reg_sram[RA*8];
if elwidth == 8:
    uint8_t *b = (uint8_t*)actual_bytes;
    b[idx] = result;
if elwidth == 16:
    uint16_t *s = (uint16_t*)actual_bytes;
    s[idx] = result;
if elwidth == 32:
    uint32_t *i = (uint32_t*)actual_bytes;
    i[idx] = result;
if elwidth == default:
    uint64_t *l = (uint64_t*)actual_bytes;
    l[idx] = result;
```

Starting with all zeros, setting `actual_bytes[3]` in any given `reg_t` to 0x01 would mean that:

- b[0..2] = 0x00 and b[3] = 0x01
- s[0] = 0x0000 and s[1] = 0x0001

- i[0] = 0x00010000
- l[0] = 0x0000000000010000

In tabular form, starting an elwidth=8 loop from r0 and extending for 16 elements would begin at r0 and extend over the entirety of r1:

```
    | byte0 | byte1 | byte2 | byte3 | byte4 | byte5 | byte6 | byte7 |
    | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- |
r0  | b[0]  | b[1]  | b[2]  | b[3]  | b[4]  | b[5]  | b[6]  | b[7]  |
r1  | b[8]  | b[9]  | b[10] | b[11] | b[12] | b[13] | b[14] | b[15] |
```

Starting an elwidth=16 loop from r0 and extending for 7 elements would begin at r0 and extend partly over r1. Note that b0 indicates the low byte (lowest 8 bits) of each 16-bit word, and b1 represents the top byte:

```
    | byte0 | byte1 | byte2 | byte3 | byte4 | byte5 | byte6 | byte7 |
    | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- |
r0  | s[0].b0  b1   | s[1].b0  b1   | s[2].b0  b1   |  s[3].b0  b1  |
r1  | s[4].b0  b1   | s[5].b0  b1   | s[6].b0  b1   |  unmodified   |
```

Likewise for elwidth=32, and a loop extending for 3 elements. b0 through b3 represent the bytes (numbered lowest for LSB and highest for MSB) within each element word:

```
    | byte0 | byte1 | byte2 | byte3 | byte4 | byte5 | byte6 | byte7 |
    | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- |
r0  | w[0].b0  b1      b2      b3   | w[1].b0  b1      b2      b3   |
r1  | w[2].b0  b1      b2      b3   | unmodified    unmodified      |
```

64-bit (default) elements access the full registers. In each case the register number (`RT`, `RA`) indicates the *starting* point for the storage and retrieval of the elements.

Our simple loop, instead of accessing the array of regfile entries with a computed index `iregs[RT+i]`, would access the appropriate element of the appropriate width, such as `iregs[RT].s[i]` in order to access 16 bit elements starting from RT. Thus we have a series of overlapping conceptual arrays that each start at what is traditionally thought of as "a register". It then helps if we have a couple of routines:

```
get_polymorphed_reg(reg, bitwidth, offset):
    reg_t res = 0;
    if (!reg.isvec): # scalar
        offset = 0
    if bitwidth == 8:
        reg.b = int_regfile[reg].b[offset]
    elif bitwidth == 16:
        reg.s = int_regfile[reg].s[offset]
    elif bitwidth == 32:
        reg.i = int_regfile[reg].i[offset]
    elif bitwidth == default: # 64
        reg.l = int_regfile[reg].l[offset]
    return res

set_polymorphed_reg(reg, bitwidth, offset, val):
    if (!reg.isvec): # scalar
        offset = 0
    if bitwidth == 8:
        int_regfile[reg].b[offset] = val
    elif bitwidth == 16:
        int_regfile[reg].s[offset] = val
    elif bitwidth == 32:
        int_regfile[reg].i[offset] = val
    elif bitwidth == default: # 64
        int_regfile[reg].l[offset] = val
```

These basically provide a convenient parameterised way to access the register file, at an arbitrary vector element offset and an arbitrary element width. Our first simple loop thus becomes:

```
for i = 0 to VL-1:
   src1 = get_polymorphed_reg(RA, srcwid, i)
   src2 = get_polymorphed_reg(RB, srcwid, i)
   result = src1 + src2 # actual add here
   set_polymorphed_reg(RT, destwid, i, result)
```

With this loop, if elwidth=16 and VL=3 the first 48 bits of the target register will contain three 16 bit addition results, and the upper 16 bits will be *unaltered*.

Note that things such as zero/sign-extension (and predication) have been left out to illustrate the elwidth concept. Also note that it turns out to be important to perform the operation internally at effectively an *infinite* bitwidth such that any truncation, rounding errors or other artefacts may all be ironed out. This turns out to be important when applying Saturation for Audio DSP workloads, particularly for multiply and IEEE754 FP rounding. By "infinite" this is conceptual only: in reality, the application of the different truncations and width-extensions set a fixed deterministic practical limit on the internal precision needed, on a per-operation basis.

Other than that, element width overrides, which can be applied to *either* source or destination or both, are pretty straightforward, conceptually. The details, for hardware engineers, involve byte-level write-enable lines, which is exactly what is used on SRAMs anyway. Compiler writers have to alter Register Allocation Tables to byte-level granularity.

One critical thing to note: upper parts of the underlying 64 bit register are *not zero'd out* by a write involving a non-aligned Vector Length. An 8 bit operation with VL=7 will *not* overwrite the 8th byte of the destination. The only situation where a full overwrite occurs is on "default" behaviour. This is extremely important to consider the register file as a byte-level store, not a 64-bit-level store.

4.6.1 Why a LE regfile?

The concept of having a regfile where the byte ordering of the underlying SRAM seems utter nonsense. Surely, a hardware implementation gets to choose the order, right? It's memory only where LE/BE matters, right? The bytes come in, all registers are 64 bit and it's just wiring, right?

Ordinarily this would be 100% correct, in both a scalar ISA and in a Cray style Vector one. The assumption in that last question was, however, "all registers are 64 bit". SV allows SIMD-style packing of vectors into the 64 bit registers, where one instruction and the next may interpret that very same register as containing elements of completely different widths.

Consequently it becomes critically important to decide a byte-order. That decision was - arbitrarily - LE mode. Actually it wasn't arbitrary at all: it was such hell to implement BE supported interpretations of CRs and LD/ST in LibreSOC, based on a terse spec that provides insufficient clarity and assumes significant working knowledge of the Power ISA, with arbitrary insertions of 7-index here and 3-bitindex there, the decision to pick LE was extremely easy.

Without such a decision, if two words are packed as elements into a 64 bit register, what does this mean? Should they be inverted so that the lower indexed element goes into the HI or the LO word? should the 8 bytes of each register be inverted? Should the bytes in each element be inverted? Should the element indexing loop order be broken onto discontiguous chunks such as 32107654 rather than 01234567, and if so at what granularity of discontinuity? These are all equally valid and legitimate interpretations of what constitutes "BE" and they all cause merry mayhem.

The decision was therefore made: the c typedef union is the canonical definition, and its members are defined as being in LE order. From there, implementations may choose whatever internal HDL wire order they like as long as the results produced conform to the elwidth pseudocode.

Note: it turns out that both x86 SIMD and NEON SIMD follow this convention, namely that both are implicitly LE, even though their ISA Manuals may not explicitly spell this out

- https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Application-Level-Memory-Model/Endian-support/Endianness-in-Advanced-SIMD?lang=en
- https://stackoverflow.com/questions/24045102/how-does-endianness-work-with-simd-registers
- https://llvm.org/docs/BigEndianNEON.html

4.6.2 Source and Destination overrides

A minor fly in the ointment: what happens if the source and destination are over-ridden to different widths? For example, FP16 arithmetic is not accurate enough and may introduce rounding errors when up-converted to FP32 output. The rule is therefore set:

```
The operation MUST take place effectively at infinite precision:
actual precision determined by the operation and the operand widths
```

In pseudocode this is:

```
for i = 0 to VL-1:
   src1 = get_polymorphed_reg(RA, srcwid, i)
   src2 = get_polymorphed_reg(RB, srcwid, i)
   opwidth = max(srcwid, destwid) # usually
   result = op_add(src1, src2, opwidth) # at max width
   set_polymorphed_reg(rd, destwid, i, result)
```

In reality the source and destination widths determine the actual required precision in a given ALU. The reason for setting "effectively" infinite precision is illustrated for example by Saturated-multiply, where if the internal precision was insufficient it would not be possible to correctly determine the maximum clip range had been exceeded.

Thus it will turn out that under some conditions the combination of the extension of the source registers followed by truncation of the result gets rid of bits that didn't matter, and the operation might as well have taken place at the narrower width and could save resources that way. Examples include Logical OR where the source extension would place zeros in the upper bits, the result will be truncated and throw those zeros away.

Counterexamples include the previously mentioned FP16 arithmetic, where for operations such as division of large numbers by very small ones it should be clear that internal accuracy will play a major role in influencing the result. Hence the rule that the calculation takes place at the maximum bitwidth, and truncation follows afterwards.

4.6.3 Signed arithmetic

What happens when the operation involves signed arithmetic? Here the implementor has to use common sense, and make sure behaviour is accurately documented. If the result of the unmodified operation is sign-extended because one of the inputs is signed, then the input source operands must be first read at their overridden bitwidth and *then* sign-extended:

```
for i = 0 to VL-1:
 src1 = get_polymorphed_reg(RA, srcwid, i)
 src2 = get_polymorphed_reg(RB, srcwid, i)
 opwidth = max(srcwid, destwid)
 # srces known to be less than result width
 src1 = sign_extend(src1, srcwid, opwidth)
 src2 = sign_extend(src2, srcwid, opwidth)
 result = op_signed(src1, src2, opwidth) # at max width
 set_polymorphed_reg(rd, destwid, i, result)
```

The key here is that the cues are taken from the underlying operation.

4.6.4 Saturation

Audio DSPs need to be able to clip sound when the "volume" is adjusted, but if it is too loud and the signal wraps, distortion occurs. The solution is to clip (saturate) the audio and allow this to be detected. In practical terms this is a post-result analysis however it needs to take place at the largest bitwidth i.e. before a result is element width truncated. Only then can the arithmetic saturation condition be detected:

```
for i = 0 to VL-1:
   src1 = get_polymorphed_reg(RA, srcwid, i)
   src2 = get_polymorphed_reg(RB, srcwid, i)
   opwidth = max(srcwid, destwid)
   # unsigned add
   result = op_add(src1, src2, opwidth) # at max width
   # now saturate (unsigned)
   sat = min(result, (1<<destwid)-1)
   set_polymorphed_reg(rd, destwid, i, sat)
   # set sat overflow
   if Rc=1:
      CR[i].ov = (sat != result)
```

So the actual computation took place at the larger width, but was post-analysed as an unsigned operation. If however "signed" saturation is requested then the actual arithmetic operation has to be carefully analysed to see what that actually means.

In terms of FP arithmetic, which by definition has a sign bit (so always takes place as a signed operation anyway), the request to saturate to signed min/max is pretty clear. However for integer arithmetic such as shift (plain shift, not arithmetic shift), or logical operations such as XOR, which were never designed to have the assumption that its inputs be considered as signed numbers, common sense has to kick in, and follow what CR0 does.

CR0 for Logical operations still applies: the test is still applied to produce CR.eq, CR.lt and CR.gt analysis. Following this lead we may do the same thing: although the input operations for and OR or XOR can in no way be thought of as "signed" we may at least consider the result to be signed, and thus apply min/max range detection -128 to +127 when truncating down to 8 bit for example.

```
for i = 0 to VL-1:
   src1 = get_polymorphed_reg(RA, srcwid, i)
   src2 = get_polymorphed_reg(RB, srcwid, i)
   opwidth = max(srcwid, destwid)
   # logical op, signed has no meaning
   result = op_xor(src1, src2, opwidth)
   # now saturate (signed)
   sat = min(result, (1<<destwid-1)-1)
   sat = max(result, -(1<<destwid-1))
   set_polymorphed_reg(rd, destwid, i, sat)
```

Overall here the rule is: apply common sense then document the behaviour really clearly, for each and every operation.

4.7 Quick recap so far

The above functionality pretty much covers around 85% of Vector ISA needs. Predication is provided so that parallel if/then/else constructs can be performed: critical given that sequential if/then statements and branches simply do not translate successfully to Vector workloads. VSPLAT capability is provided which is approximately 20% of all GPU workload operations. Also covered, with elwidth overriding, is the smaller arithmetic operations

that caused ISAs developed from the late 80s onwards to get themselves into a tiz when adding "Multimedia" acceleration aka "SIMD" instructions.

Experienced Vector ISA readers will however have noted that VCOMPRESS and VEXPAND are missing, as is Vector "reduce" (mapreduce) capability and VGATHER and VSCATTER. Compress and Expand are covered by Twin Predication, and yet to also be covered is fail-on-first, CR-based result predication, and Subvectors and Swizzle.

4.7.1 SUBVL

Adding in support for SUBVL is a matter of adding in an extra inner for-loop, where register src and dest are still incremented inside the inner part. Predication is still taken from the VL index, however it is applied to the whole subvector:

```
function op_add(RT, RA, RB) # add not VADD!
  int id=0, irs1=0, irs2=0;
  predval = get_pred_val(FALSE, rd);
  for i = 0 to VL-1:
    if (predval & 1<<i) # predication uses intregs
      for (s = 0; s < SUBVL; s++)
        sd = id*SUBVL + s
        srs1 = irs1*SUBVL + s
        srs2 = irs2*SUBVL + s
        ireg[RT+sd] <= ireg[RA+srs1] + ireg[RB+srs2];
      if (!RT.isvec) break;
    if (RT.isvec)  { id += 1; }
    if (RA.isvec)  { irs1 += 1; }
    if (RB.isvec)  { irs2 += 1; }
```

The primary reason for this is because Shader Compilers treat vec2/3/4 as "single units". Recognising this in hardware is just sensible.

4.8 Swizzle

Swizzle is particularly important for 3D work. It allows in-place reordering of XYZW, ARGB etc. and access of sub-portions of the same in arbitrary order *without* requiring timeconsuming scalar mv instructions (scalar due to the convoluted offsets).

Swizzling does not just do permutations: it allows arbitrary selection and multiple copying of vec2/3/4 elements, such as XXXZ as the source operand, which will take 3 copies of the vec4 first element (vec4[0]), placing them at positions vec4[0], vec4[1] and vec4[2], whilst the "Z" element (vec4[2]) was copied into vec4[3].

With somewhere between 10% and 30% of operations in 3D Shaders involving swizzle this is a huge saving and reduces pressure on register files due to having to use significant numbers of mv operations to get vector elements to "line up".

In SV given the percentage of operations that also involve initialisation to 0.0 or 1.0 into subvector elements the decision was made to include those:

```
swizzle = get_swizzle_immed() # 12 bits
for (s = 0; s < SUBVL; s++)
    remap = (swizzle >> 3*s) & 0b111
    if remap == 0b000: continue            # skip
    if remap == 0b001: break               # end marker
    if remap == 0b010: ireg[rd+s] <= 0.0   # constant 0
    elif remap == 0b011: ireg[rd+s] <= 1.0 # constant 1
```

```
else:                                   # XYZW
   sm = id*SUBVL + (remap-4)
   ireg[rd+s] <= ireg[RA+sm]
```

Note that a value of 0b000 will leave the target subvector element untouched. This is equivalent to a predicate mask which is built-in, in immediate form, into the {Swizzle Move} operation. mv.swizzle is rare in that it is one of the few instructions needed to be added that are never going to be part of a Scalar ISA. Even in High Performance Compute workloads it is unusual: it is only because SV is targetted at 3D and Video that it is being considered.

Some 3D GPU ISAs also allow for two-operand subvector swizzles. These are sufficiently unusual, and the immediate opcode space required so large (12 bits per vec4 source), that the tradeoff balance was decided in SV to only add mv.swizzle.

4.9 Twin Predication

Twin Predication is cool. Essentially it is a back-to-back VCOMPRESS-VEXPAND (a multiple sequentially ordered VINSERT). The compress part is covered by the source predicate and the expand part by the destination predicate. Of course, if either of those is all 1s then the operation degenerates *to* VCOMPRESS or VEXPAND, respectively.

```
function op(RT, RS):
  ps = get_pred_val(FALSE, RS); # predication on src
  pd = get_pred_val(FALSE, RT); # ... AND on dest
  for (int i = 0, int j = 0; i < VL && j < VL;):
    if (RS.isvec) while (!(ps & 1<<i)) i++;
    if (RT.isvec) while (!(pd & 1<<j)) j++;
    reg[RT+j] = SCALAR_OPERATION_ON(reg[RS+i])
    if (RS.isvec) i++;
    if (RT.isvec) j++; else break
```

Here's the interesting part: given the fact that SV is a "context" extension, the above pattern can be applied to a lot more than just MV, which is normally only what VCOMPRESS and VEXPAND do in traditional Vector ISAs: move registers. Twin Predication can be applied to `extsw` or `fcvt`, LD/ST operations and even `rlwinmi` and other operations taking a single source and immediate(s) such as `addi`. All of these are termed single-source, single-destination.

LDST Address-generation, or AGEN, is a special case of single source, because elwidth overriding does not make sense to apply to the computation of the 64 bit address itself, but it *does* make sense to apply elwidth overrides to the data being accessed *at* that memory address.

It also turns out that by using a single bit set in the source or destination, *all* the sequential ordered standard patterns of Vector ISAs are provided: VSPLAT, VSELECT, VINSERT, VCOMPRESS, VEXPAND.

The only one missing from the list here, because it is non-sequential, is VGATHER (and VSCATTER): moving registers by specifying a vector of register indices (`regs[rd] = regs[regs[rs]]` in a loop). This one is tricky because it typically does not exist in standard scalar ISAs. If it did it would be called [[sv/mv.x]]. Once Vectorised, it's a VGATHER/VSCATTER.

4.10 Exception-based Fail-on-first

One of the major issues with Vectorised LD/ST operations is when a batch of LDs cross a page-fault boundary. With considerable resources being taken up with in-flight data, a large Vector LD being cancelled or unable to roll back is either a detriment to performance or can cause data corruption.

What if, then, rather than cancel an entire Vector LD because the last operation would cause a page fault, instead truncate the Vector to the last successful element?

This is called "fail-on-first". Here is strncpy, illustrated from RVV:

```
strncpy:
    c.mv a3, a0               # Copy dst
loop:
    setvli x0, a2, vint8    # Vectors of bytes.
    vlbff.v v1, (a1)        # Get src bytes
    vseq.vi v0, v1, 0       # Flag zero bytes
    vmfirst a4, v0          # Zero found?
    vmsif.v v0, v0          # Set mask up to and including zero byte.
    vsb.v v1, (a3), v0.t    # Write out bytes
    c.bgez a4, exit           # Done
    csrr t1, vl             # Get number of bytes fetched
    c.add a1, a1, t1          # Bump src pointer
    c.sub a2, a2, t1          # Decrement count.
    c.add a3, a3, t1          # Bump dst pointer
    c.bnez a2, loop           # Anymore?
exit:
    c.ret
```

Vector Length VL is truncated inherently at the first page faulting byte-level LD. Otherwise, with more powerful hardware the number of elements LOADed from memory could be dozens to hundreds or greater (memory bandwidth permitting).

With VL truncated the analysis looking for the zero byte and the subsequent STORE (a straight ST, not a ffirst ST) can proceed, safe in the knowledge that every byte loaded in the Vector is valid. Implementors are even permitted to "adapt" VL, truncating it early so that, for example, subsequent iterations of loops will have LD/STs on aligned boundaries.

SIMD strncpy hand-written assembly routines are, to be blunt about it, a total nightmare. 240 instructions is not uncommon, and the worst thing about them is that they are unable to cope with detection of a page fault condition.

Note: see https://bugs.libre-soc.org/show_bug.cgi?id=561

4.11 Data-dependent fail-first

Data-dependent fail-first *stops* at the first failure:

```
if Rc=0: BO = inv<<2 | 0b00 # test CR.eq bit z/nz
for i in range(VL):
    # predication test, skip all masked out elements.
    if predicate_masked_out(i): continue # skip
    result = op(iregs[RA+i], iregs[RB+i])
    CRnew = analyse(result) # calculates eq/lt/gt
    # now test CR, similar to branch
    if CRnew[BO[0:1]] != BO[2]:
        VL = i+VLi # truncate: only successes allowed
        break
    # test passed: store result (and CR?)
    if not RC1: iregs[RT+i] = result
    if RC1 or Rc=1: crregs[offs+i] = CRnew
```

This is particularly useful, again, for FP operations that might overflow, where it is desirable to end the loop early, but also desirable to complete at least those operations that were okay (passed the test) without also

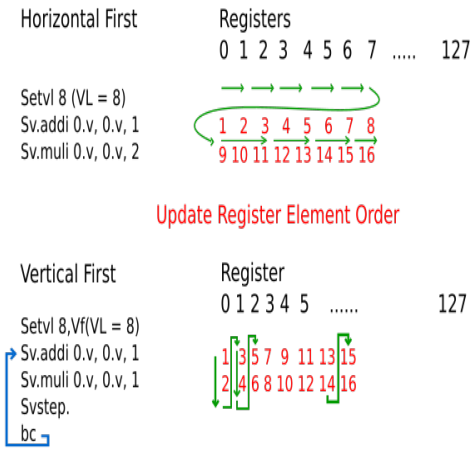

Figure 4.4: image

having to slow down execution by adding extra instructions that tested for the possibility of that failure, in advance of doing the actual calculation.

The only minor downside here though is the change to VL, which in some implementations may cause pipeline stalls.

4.12 Vertical-First Mode

This is a relatively new addition to SVP64 under development as of July 2021. Where Horizontal-First is the standard Cray-style for-loop, Vertical-First typically executes just the **one** scalar element in each Vectorised operation. That element is selected by srcstep and dststep *neither of which are changed as a side-effect of execution*. Illustrating this in pseodocode, with a branch/loop. To create loops, a new instruction `svstep` must be called, explicitly, with Rc=1:

```
loop:
  sv.addi r0.v, r8.v, 5 # GPR(0+dststep) = GPR(8+srcstep) + 5
  sv.addi r0.v, r8, 5   # GPR(0+dststep) = GPR(8        ) + 5
  sv.addi r0, r8.v, 5   # GPR(0        ) = GPR(8+srcstep) + 5
  svstep.               # srcstep++, dststep++, CR0.eq = srcstep==VL
  beq loop
```

Three examples are illustrated of different types of Scalar-Vector operations. Note that in its simplest form **only one** element is executed per instruction **not** multiple elements per instruction. (The more advanced version of Vertical-First mode may execute multiple elements per instruction, however the number executed **must** remain a fixed quantity.)

Now that such explicit loops can increment inexorably towards VL, of course we now need a way to test if srcstep or dststep have reached VL. This is achieved in one of two ways: {svstep instruction} has an Rc=1 mode where CR0 will be updated if VL is reached. A standard v3.0B Branch Conditional may rely on that. Alternatively, the number of elements may be transferred into CTR, as is standard practice in Power ISA. Here, SVP64 {Branch Mode} have a mode which allows CTR to be decremented by the number of vertical elements executed.

4.13 Instruction format

Whilst this overview shows the internals, it does not go into detail on the actual instruction format itself. There are a couple of reasons for this: firstly, it's under development, and secondly, it needs to be proposed to the OpenPOWER Foundation ISA WG for consideration and review.

That said: draft pages for {setvl instruction} and {SVP64 Chapter} are written up. The `setvl` instruction is pretty much as would be expected from a Cray style VL instruction: the only differences being that, firstly, the MAXVL (Maximum Vector Length) has to be specified, because that determines - precisely - how many of the *scalar* registers are to be used for a given Vector. Secondly: within the limit of MAXVL, VL is required to be set to the requested value. By contrast, RVV systems permit the hardware to set arbitrary values of VL.

The other key question is of course: what's the actual instruction format, and what's in it? Bearing in mind that this requires OPF review, the current draft is at the {SVP64 Chapter} page, and includes space for all the different modes, the predicates, element width overrides, SUBVL and the register extensions, in 24 bits. This just about fits into a Power v3.1B 64 bit Prefix by borrowing some of the Reserved Encoding space. The v3.1B suffix - containing as it does a 32 bi Power instruction - aligns perfectly with SV.

Further reading is at the main {Scalable Vectors for Power ISA} page.

4.14 Conclusion

Starting from a scalar ISA - Power v3.0B - it was shown above that, with conceptual sub-loops, a Scalar ISA can be turned into a Vector one, by embedding Scalar instructions - unmodified - into a Vector "context" using "Prefixing". With careful thought, this technique reaches 90% par with good Vector ISAs, increasing to 95% with the addition of a mere handful of additional context-vectoriseable scalar instructions ([[sv/mv.x]] amongst them).

What is particularly cool about the SV concept is that custom extensions and research need not be concerned about inventing new Vector instructions and how to get them to interact with the Scalar ISA: they are effectively one and the same. Any new instruction added at the Scalar level is inherently and automatically Vectorised, following some simple rules.

Chapter 5

Compliancy Levels

5.1 Simple-V Compliancy Levels

The purpose of the Compliancy Levels is to provide a documented stable base for implementors to achieve software interoperability without requiring a high and unnecessary hardware cost unrelated to their needs. The bare minimum requirement, particularly suited for Ultra-embedded, requires just one instruction, reservation of SPRs, and the rest may entirely be Soft-emulated by raising Illegal Instruction traps. At the other end of the spectrum is the full REMAP Structure Packing suitable for traditional Vector Processing workloads and High-performance energy-efficient DSP workloads.

To achieve full soft-emulated interoperability, all implementations **must**, at the bare minimum, raise Illegal Instruction traps for all SPRs including all reserved SPRs, all SVP64-related Context instructions (REMAP), as well as for the entire SVP64 Prefix space.

Even if the Power ISA Scalar Specification states that a given Scalar instruction need not or must not raise an illegal instruction on UNDEFINED behaviour, unimiplemented parts of SVP64 MUST* raise an illegal instruction trap when (and only when) that same Scalar instruction is Prefixed*. It is absolutely critical to note that when not Prefixed, under no circumstances shall the Scalar instruction deviate from the Scalar Power ISA Specification.

Summary of Compliancy Levels, each Level includes all lower levels:

- **Zero-Level**: Simple-V is not implemented (at all) in hardware. This Level is required to be listed because all capabilities of Simple-V must be Soft-emulatable.
- **Ultra-embedded**: `setvl` instruction and context-switching of SVSTATE to/from SVSRR1. Register Files as Standard Power ISA. `scalar identity` implemented.
- **Embedded**: `svstep` instruction, and support for Hardware for-looping in both Horizontal-First and Vertical-First Mode as well as Predication (Single and Twin) for the GPRs r3, r10 and r30. CR-Field-based Predicates do not need to be added.
- **Embedded DSP/AV**: 128 registers, element-width overrides, and Saturation and Mapreduce/Iteration Modes.
- **High-end DSP/AV**: Same as Embedded-DSP/AV except also including Indexed and Offset REMAP capability.
- **3D/Advanced/Supercomputing**: all SV Branch instructions; crweird and vector-assist instructions (`set-before-first` etc); Swizzle Move instructions; Matrix, DCT/FFT and Indexing REMAP capability; Fail-First and Predicate-Result Modes.

These requirements within each Level constitute the minimum mandatory capabilities. It is also permitted that any Level include any part of a higher Compliancy Level. For example: an Embedded Level is permitted to have 128 GPRs, FPRs and CR Fields, but the Compliance Tests for Embedded will only test for 32. DSP/VPU Level is permitted to implement the DCT REMAP capability, but will not be permitted to declare meeting the

3D/Advanced Level unless implementing *all* REMAP Capabilities.

Power ISA Compliancy Levels

The SV Compliancy Levels have nothing to do with the Power ISA Compliancy Levels (SFS, SFFS, Linux, AIX). They are separate and independent. It is perfectly fine to implement Ultra-Embedded on AIX, and perfectly fine to implement 3D/Advanced on SFS. **Compliance with SV Levels does not convey or remove the obligation of Compliance with SFS/SFFS/Linux/AIX Levels and vice-versa**.

5.1.1 Zero-Level

This level exists to indicate the critical importance of all and any features attempted to be executed on hardware that has no support at all for Simple-V being **required** to raise Illegal Exceptions. **This includes existing Power ISA Implementations:** IBM POWER being the most notable.

With parts of the Power ISA being "silent executed" (hints for example), it is absolutely critical to have all capabilities of Simple-V sit within full Illegal Instruction space of existing and future Hardware.

5.1.2 Ultra-Embedded Level

This level exists as an entry-level into SVP64, most suited to resource constrained soft cores, or Hardware implementations where unit cost is a much higher priority than execution speed.

This level sets the bare minimum requirements, where everything with the exception of `scalar identity` and the `setvl` instruction may be software-emulated through JIT Translation or Illegal Instruction traps. SVSTATE, as effectively a Sub-Program-Counter, joins MSR and PC (CIA, NIA) as direct peers and must be switched on any context-switch (Trap or Exception)

- PC is saved/restored to/from SRR0
- MSR is saved/restored to/from SRR1
- SVSTATE **must** also be saved/restored to/from SVSRR1

Any implementation that implements Hypervisor Mode must also correspondingly follow the Power ISA Spec guidelines for HSRR0 and HSRR1, and must save/restore SVSTATE to/from HSVSRR1 in all circumstances involving save/restore to/from HSRR0 and HSRR1.

Illegal Instruction Trap **must** be raised on:

- Any SV instructions not implemented
- any unimplemented SV Context SPRs read or written
- all unimplemented uses of the SVP64 Prefix
- non-scalar-identity SVP64 instructions

Implementors are free and clear to implement any other features of SVP64 however only by meeting all of the mandatory requirements above will Compliance with the Ultra-Embedded Level be achieved.

Note that `scalar identity` is defined as being when the execution of an SVP64 Prefixed instruction is identical in every respect to Scalar non-prefixed, i.e. as if the Prefix had not been present. Additionally all SV SPRs must be zero and the 24-bit `RM` field must be zero.

5.1.3 Embedded Level

This level is more suitable for Hardware implementations where performance and power saving begins to matter. A second instruction, `svstep`, used by Vertical-First Mode, is required, as is hardware-level looping in Horizontal-First Mode. Illegal Instruction trap may not be used to emulate `svstep`.

At the bare minimum, Twin and Single Predication must be supported for at least the GPRs r3, r10 and r30. CR Field Predication may also be supported in hardware but only by also increasing the number of CR Fields to the required total 128.

Another important aspect is that when Rc=1 is set, CR Field Vector co-results are produced. Should these exceed CR7 (CR8-CR127) and the number of CR Fields has not been increased to 128 then an Illegal Instruction Trap must be raised. In practical terms, to avoid this occurrence in Embedded software, MAXVL should not exceed 8 for Arithmetic or Logical operations with Rc=1.

Zeroing on source and destination for Predicates must also be supported (sz, dz) however all other Modes (Saturation, Fail-First, Predicate-Result, Iteration/Reduction) are entirely optional. Implementation of Element-Width Overrides is also optional.

One of the important side-benefits of this SV Compliancy Level is that it brings Hardware-level support for Scalar Predication (VL=MAXVL=1) to the entire Scalar Power ISA, completely without modifying the Scalar Power ISA. The cost in software is that Predicated instructions are Prefixed to 64-bit.

5.1.4 DSP / Audio / Video Level

This level is best suited to high-performance power-efficient but specialist Compute workloads. 128 GPRs, FPRs and CR Fields are all required, as is element-width overrides to allow data processing down to the 8-bit level. SUBVL support (Sub-Vector vec2/3/4) is also required, as is Pack/Unpack EXTRA format (helps with Pixel and Audio Stream Structured data)

All SVP64 Modes must be implemented in hardware: Saturation in particular is a necessity for Audio DSP work. Reduction as well to assist with Audio/Video.

It is not mandatory for this Level to have DCT/FFT REMAP Capability in hardware but due to the high prevalence of DCT and FFT in Audio, Video and DSP workloads it is strongly recommended. Matrix (Dimensional) REMAP and Swizzle may also be useful to help with 24-bit (3 byte) Structured Audio Streams and are also recommended but not mandatory.

5.1.5 High-end DSP

In this Compliancy Level the benefits of the Offset and Index REMAP subsystem becomes worth its hardware cost. In lower-performing DSP and A/V workloads it is not.

5.1.6 3D / Advanced / Supercomputing

This Compliancy Level is for highest performance and energy efficiency. All aspects of SVP64 must be entirely implemented, in full, in Hardware. How that is achieved is entirely at the discretion of the implementor: there are no hard requirements of any kind on the level of performance, just as there are none in the Vulkan(TM) Specification.

Throughout the SV Specification however there are hints to Micro-Architects: byte-level write-enable lines on Register Files is strongly recommended, for example, in order to avoid unnecessary Read-Modify-Write cycles and additional Register Hazard Dependencies on fine-grained (8/16/32-bit) operations. Just as with SRAMs multiple write-enable lines may be raised to update higher-width elements.

5.1.7 Examples

Assuming that hardware implements scalar operations only, and implements predication but not elwidth overrides:

```
setvli r0, 4            # sets VL equal to 4
sv.addi r5, r0, 1       # raises an 0x700 trap
setvli r0, 1            # sets VL equal to 1
sv.addi r5, r0, 1       # gets executed by hardware
sv.addi/ew=8 r5, r0, 1  # raises an 0x700 trap
sv.ori/sm=EQ r5, r0, 1  # executed by hardware
```

The first `sv.addi` raises an illegal instruction trap because VL has been set to 4, and this is not supported. Likewise elwidth overrides if requested always raise illegal instruction traps.

Such an implementation would qualify for the "Ultra-Embedded" SV Level. It would not qualify for the "Embedded" level because when VL=4 an Illegal Exception is raised, and the Embedded Level requires full VL Loop support in hardware.

[[!tag standards]]

Chapter 6

SVP64

6.1 SVP64 Zero-Overhead Loop Prefix Subsystem

- **DRAFT STATUS v0.1 18sep2021** Release notes https://bugs.libre-soc.org/show_bug.cgi?id=699

This document describes {Scalable Vectors for Power ISA} augmentation of the Power v3.0B ISA. It is in Draft Status and will be submitted to the [[!wikipedia OpenPOWER_Foundation]] ISA WG via the External RFC Process.

Credits and acknowledgements:

- Luke Leighton
- Jacob Lifshay
- Hendrik Boom
- Richard Wilbur
- Alexandre Oliva
- Cesar Strauss
- NLnet Foundation, for funding
- OpenPOWER Foundation
- Paul Mackerras
- Toshaan Bharvani
- IBM for the Power ISA itself

Links:

- http://lists.libre-soc.org/pipermail/libre-soc-dev/2020-December/001498.html>
- [[svp64/discussion]]
- {SVP64 Appendix}
- http://lists.libre-soc.org/pipermail/libre-soc-dev/2020-December/001650.html
- https://bugs.libre-soc.org/show_bug.cgi?id=550
- https://bugs.libre-soc.org/show_bug.cgi?id=573 TODO elwidth “infinite” discussion
- https://bugs.libre-soc.org/show_bug.cgi?id=574 Saturating description.
- https://bugs.libre-soc.org/show_bug.cgi?id=905 TODO [[sv/svp64-single]]
- https://bugs.libre-soc.org/show_bug.cgi?id=1045 External RFC ls010
- {Branch Mode} chapter
- {Load/Store Mode} chapter

Table of contents

[[!toc]]

6.1.1 Introduction

Simple-V is a type of Vectorisation best described as a "Prefix Loop Subsystem" similar to the 5 decades-old Zilog Z80 `LDIR` instruction and to the 8086 `REP` Prefix instruction. More advanced features are similar to the Z80 `CPIR` instruction. If naively viewed one-dimensionally as an actual Vector ISA it introduces over 1.5 million 64-bit True-Scalable Vector instructions on the SFFS Subset and closer to 10 million 64-bit True-Scalable Vector instructions if introduced on VSX. SVP64, the instruction format used by Simple-V, is therefore best viewed as an orthogonal RISC-paradigm "Prefixing" subsystem instead.

Except where explicitly stated all bit numbers remain as in the rest of the Power ISA: in MSB0 form (the bits are numbered from 0 at the MSB on the left and counting up as you move rightwards to the LSB end). All bit ranges are inclusive (so `4:6` means bits 4, 5, and 6, in MSB0 order). **All register numbering and element numbering however is LSB0 ordering** which is a different convention from that used elsewhere in the Power ISA.

The SVP64 prefix always comes before the suffix in PC order and must be considered an independent "Defined word" that augments the behaviour of the following instruction, but does **not** change the actual Decoding of that following instruction. **All prefixed 32-bit instructions (Defined Words) retain their non-prefixed encoding and definition**.

Two apparent exceptions to the above hard rule exist: SV Branch-Conditional operations and LD/ST-update "Post-Increment" Mode. Post-Increment was considered sufficiently high priority (significantly reducing hot-loop instruction count) that one bit in the Prefix is reserved for it (*Note the intention to release that bit and move Post-Increment instructions to EXT2xx, as part of {RFC ls011}*). Vectorised Branch-Conditional operations "embed" the original Scalar Branch-Conditional behaviour into a much more advanced variant that is highly suited to High-Performance Computation (HPC), Supercomputing, and parallel GPU Workloads.

Architectural Resource Allocation note: it is prohibited to accept RFCs which fundamentally violate this hard requirement. Under no circumstances must the Suffix space have an alternate instruction encoding allocated within SVP64 that is entirely different from the non-prefixed Defined Word. Hardware Implementors critically rely on this inviolate guarantee to implement High-Performance Multi-Issue micro-architectures that can sustain 100% throughput

Subset implementations in hardware are permitted, as long as certain rules are followed, allowing for full soft-emulation including future revisions. Compliancy Subsets exist to ensure minimum levels of binary interoperability expectations within certain environments. Details in the {SVP64 Appendix}.

6.1.2 SVP64 encoding features

A number of features need to be compacted into a very small space of only 24 bits:

- Independent per-register Scalar/Vector tagging and range extension on every register
- Element width overrides on both source and destination
- Predication on both source and destination
- Two different sources of predication: INT and CR Fields
- SV Modes including saturation (for Audio, Video and DSP), mapreduce, and fail-first mode.

Different classes of operations require different formats. The earlier sections cover the common formats and the four separate modes follow: CR operations (crops), Arithmetic/Logical (termed "normal"), Load/Store and Branch-Conditional.

6.1.3 Definition of Reserved in this spec.

For the new fields added in SVP64, instructions that have any of their fields set to a reserved value must cause an illegal instruction trap, to allow emulation of future instruction sets, or for subsets of SVP64 to be implemented in hardware and the rest emulated. This includes SVP64 SPRs: reading or writing values which are

not supported in hardware must also raise illegal instruction traps in order to allow emulation. Unless otherwise stated, reserved values are always all zeros.

This is unlike OpenPower ISA v3.1, which in many instances does not require a trap if reserved fields are nonzero. Where the standard Power ISA definition is intended the red keyword `RESERVED` is used.

6.1.4 Definition of "UnVectoriseable"

Any operation that inherently makes no sense if repeated is termed "UnVectoriseable" or "UnVectorised". Examples include `sc` or `sync` which have no registers. `mtmsr` is also classed as UnVectoriseable because there is only one `MSR`.

UnVectorised instructions are required to be detected as such if Prefixed (either SVP64 or SVP64Single) and an Illegal Instruction Trap raised.

Architectural Note: Given that a "pre-classification" Decode Phase is required (identifying whether the Suffix - Defined Word - is Arithmetic/Logical, CR-op, Load/Store or Branch-Conditional), adding "UnVectorised" to this phase is not unreasonable.

6.1.5 Definition of Strict Program Order

Strict Program Order is defined as giving the appearance, as far as programs are concerned, that instructions were executed strictly in the sequence that they occurred. A "Precise" out-of-order Micro-architecture goes to considerable lengths to ensure that this is the case.

Many Vector ISAs allow interrupts to occur in the middle of processing of large Vector operations, only under the condition that partial results are cleanly discarded, and continuation on return from the Trap Handler will restart the entire operation. The reason is that saving of full Architectural State is not practical.

Simple-V operates on an entirely different paradigm from traditional Vector ISAs: as a Sub-Program Counter where "Elements" are synonymous with Scalar instructions. With this in mind it is critical for implementations to observe Strict Element-Level Program Order at all times (often simply referred to as just "Strict Program Order" throughout this Chapter). *Any* element is Interruptible and Simple-V has been carefully designed to guarantee that Architectural State may be fully preserved and restored regardless of that same State, but it is not necessarily guaranteed that the amount of time needed to recover will be low latency (particularly if REMAP is active).

Interrupts still only save `MSR` and `PC` in `SRR0` and `SRR1` but the full SVP64 Architectural State may be saved and restored through manual copying of `SVSTATE` (and the four REMAP SPRs if in use at the time) Whilst this initially sounds unsafe in reality all that Trap Handlers (and function call stack save/restore) need do is avoid use of SVP64 Prefixed instructions to perform the necessary save/restore of Simple-V Architectural State. This capability also allows nested function calls to be made from inside Vertical-First Vector loops, which is very rare for Vector ISAs.

Strict Program Order is also preserved by the Parallel Reduction REMAP Schedule, but only at the cost of requiring the destination Vector to be used (Deterministically) to store partial progress of the Parallel Reduction.

The only major caveat for REMAP is that after an explicit change to Architectural State caused by writing to the Simple-V SPRs, some implementations may find it easier to take longer to calculate where in a given Schedule the re-mapping Indices were. Obvious examples include Interrupts occuring in the middle of a non-RADIX2 Matrix Multiply Schedule (5x3 by 3x3 for example), which will force implementations to perform divide and modulo calculations.

An additional caveat involves Condition Register Fields when also used as Predicate Masks. An operation that overwrites the same CR Fields that are simultaneously being used as a Predicate Mask is `UNDEFINED` behaviour if the overwritten CR field element was needed by a subsequent Element for its Predicate Mask bit. This allows implementations to relax some of the otherwise-draconian Register Hazards that would otherwise occur, and to consider internal cacheing of the CR-based Predicate bits, but some implementations *may not necessarily*

perform pre-reading and consequently the risk of overwrite is the responsibility of the Programmer. Special care is particularly needed here when using REMAP.

6.1.6 Register files, elements, and Element-width Overrides

The relationship between register files, elements, and element-width overrides is expressed as follows:

- register files are considered to be *byte-level* contiguous SRAMs, accessed exclusively in Little-Endian Byte-Order at all times
- elements are sequential contiguous unbounded arrays starting at the "address" of any given 64-bit GPR or FPR, numbered from 0 as the first, "spilling" into numerically-sequentially-increasing GPRs
- element-width overrides set the width of the *elements* in the sequentially-numbered contiguous array.

The relationship is best defined in Canonical form, below, in ANSI c as a union data structure. A key difference is that VSR elements are bounded fixed at 128-bit, where SVP64 elements are conceptually unbounded and only limited by the Maximum Vector Length.

Future specification note: SVP64 may be defined on top of VSRs in future. At which point VSX also gains conceptually unbounded VSR register elements

In the Upper Compliancy Levels of SVP64 the size of the GPR and FPR Register files are expanded from 32 to 128 entries, and the number of CR Fields expanded from CR0-CR7 to CR0-CR127. (Note: A future version of SVP64 is anticipated to extend the VSR register file).

Memory access remains exactly the same: the effects of `MSR.LE` remain exactly the same, affecting as they already do and remain **only** on the Load and Store memory-register operation byte-order, and having nothing to do with the ordering of the contents of register files or register-register operations.

The only major impact on Arithmetic and Logical operations is that all Scalar operations are defined, where practical and workable, to have three new widths: elwidth=32, elwidth=16, elwidth=8. The default of elwidth=64 is the pre-existing (Scalar) behaviour which remains 100% unchanged. Thus, `addi` is now joined by a 32-bit, 16-bit, and 8-bit variant of `addi`, but the sole exclusive difference is the width. *In no way* is the actual `addi` instruction fundamentally altered. FP Operations elwidth overrides are also defined, as explained in the {SVP64 Appendix}.

To be absolutely clear:

```
There are no conceptual arithmetic ordering or other changes over the
Scalar Power ISA definitions to registers or register files or to
arithmetic or Logical Operations beyond element-width subdivision
```

Element offset numbering is naturally **LSB0-sequentially-incrementing from zero, not MSB0-incrementing** including when element-width overrides are used, at which point the elements progress through each register sequentially from the LSB end (confusingly numbered the highest in MSB0 ordering) and progress incrementally to the MSB end (confusingly numbered the lowest in MSB0 ordering).

When exclusively using MSB0-numbering, SVP64 becomes unnecessarily complex to both express and subsequently understand: the required conditional subtractions from 63, 31, 15 and 7 needed to express the fact that elements are LSB0-sequential unfortunately become a hostile minefield, obscuring both intent and meaning. Therefore for the purposes of this section the more natural **LSB0 numbering is assumed** and it is left to the reader to translate to MSB0 numbering.

The Canonical specification for how element-sequential numbering and element-width overrides is defined is expressed in the following c structure, assuming a Little-Endian system, and naturally using LSB0 numbering everywhere because the ANSI c specification is inherently LSB0. Note the deliberate similarity to how VSX register elements are defined, from Figure 97, Book I, Section 6.3, Page 258:

```
#pragma pack
typedef union {
    uint8_t actual_bytes[8];
```

```
        // all of these are very deliberately unbounded arrays
        // that intentionally "wrap" into subsequent actual_bytes...
        uint8_t  bytes[]; // elwidth 8
        uint16_t hwords[]; // elwidth 16
        uint32_t words[]; // elwidth 32
        uint64_t dwords[]; // elwidth 64

    } el_reg_t;

    // ... here, as packed statically-defined GPRs.
    elreg_t int_regfile[128];

    // use element 0 as the destination
    void get_register_element(el_reg_t* el, int gpr, int element, int width) {
        switch (width) {
            case 64: el->dwords[0] = int_regfile[gpr].dwords[element];
            case 32: el->words[0] = int_regfile[gpr].words[element];
            case 16: el->hwords[0] = int_regfile[gpr].hwords[element];
            case 8 : el->bytes[0] = int_regfile[gpr].bytes[element];
        }
    }

    // use element 0 as the source
    void set_register_element(el_reg_t* el, int gpr, int element, int width) {
        switch (width) {
            case 64: int_regfile[gpr].dwords[element] = el->dwords[0];
            case 32: int_regfile[gpr].words[element] = el->words[0];
            case 16: int_regfile[gpr].hwords[element] = el->hwords[0];
            case 8 : int_regfile[gpr].bytes[element] = el->bytes[0];
        }
    }
```

Example Vector-looped add operation implementation when elwidths are 64-bit:

```
    # vector-add RT, RA,RB using the "uint64_t" union member, "dwords"
    for i in range(VL):
        int_regfile[RT].dword[i] = int_regfile[RA].dword[i] + int_regfile[RB].dword[i]
```

However if elwidth overrides are set to 16 for both source and destination:

```
    # vector-add RT, RA, RB using the "uint64_t" union member "hwords"
    for i in range(VL):
        int_regfile[RT].hwords[i] = int_regfile[RA].hwords[i] + int_regfile[RB].hwords[i]
```

The most fundamental aspect here to understand is that the wrapping into subsequent Scalar GPRs that occurs on larger-numbered elements including and especially on smaller element widths is **deliberate and intentional**. From this Canonical definition it should be clear that sequential elements begin at the LSB end of any given underlying Scalar GPR, progress to the MSB end, and then to the LSB end of the *next numerically-larger Scalar GPR*. In the example above if VL=5 and RT=1 then the contents of GPR(1) and GPR(2) will be as follows. For clarity in the table below:

- Both MSB0-ordered bitnumbering *and* LSB-ordered bitnumbering are shown
- The GPR-numbering is considered LSB0-ordered
- The Element-numbering (result0-result4) is LSB0-ordered
- Each of the results (result0-result4) are 16-bit
- "same" indicates "no change as a result of the Vectorised add"

```
    | MSB0:  | 0:15    | 16:31   | 32:47   | 48:63   |
    | LSB0:  | 63:48   | 47:32   | 31:16   | 15:0    |
```

```
|--------|---------|---------|---------|---------|
| GPR(0) | same    | same    | same    | same    |
| GPR(1) | result3 | result2 | result1 | result0 |
| GPR(2) | same    | same    | same    | result4 |
| GPR(3) | same    | same    | same    | same    |
| ...    | ...     | ...     | ...     | ...     |
| ...    | ...     | ...     | ...     | ...     |
```

Note that the upper 48 bits of GPR(2) would **not** be modified due to the example having VL=5. Thus on "wrapping" - sequential progression from GPR(1) into GPR(2) - the 5th result modifies **only** the bottom 16 LSBs of GPR(1).

Hardware Architectural note: to avoid a Read-Modify-Write at the register file it is strongly recommended to implement byte-level write-enable lines exactly as has been implemented in DRAM ICs for many decades. Additionally the predicate mask bit is advised to be associated with the element operation and alongside the result ultimately passed to the register file. When element-width is set to 64-bit the relevant predicate mask bit may be repeated eight times and pull all eight write-port byte-level lines HIGH. Clearly when element-width is set to 8-bit the relevant predicate mask bit corresponds directly with one single byte-level write-enable line. It is up to the Hardware Architect to then amortise (merge) elements together into both PredicatedSIMD Pipelines as well as simultaneous non-overlapping Register File writes, to achieve High Performance designs. Overall it helps to think of the register files as being much more akin to a byte-level-addressable SRAM.

If the 16-bit operation were to be followed up with a 32-bit Vectorised Operation, the exact same contents would be viewed as follows:

```
| MSB0:  | 0:31                | 32:63               |
| LSB0:  | 63:32               | 31:0                |
|--------|---------------------|---------------------|
| GPR(0) | same                | same                |
| GPR(1) | (result3 || result2) | (result1 || result0) |
| GPR(2) | same                | (same    || result4) |
| GPR(3) | same                | same                |
| ...    | ...                 | ...                 |
| ...    | ...                 | ...                 |
```

In other words, this perspective really is no different from the situation where the actual Register File is treated as an Industry-standard byte-level-addressable Little-Endian-addressed SRAM. Note that this perspective does **not** involve `MSR.LE` in any way shape or form because `MSR.LE` is directly in control of the Memory-to-Register byte-ordering. This section is exclusively about how to correctly perceive Simple-V-Augmented **Register** Files.

Comparative equivalent using VSR registers

For a comparative data point the VSR Registers may be expressed in the same fashion. The c code below is directly an expression of Figure 97 in Power ISA Public v3.1 Book I Section 6.3 page 258, *after compensating for MSB0 numbering in both bits and elements, adapting in full to LSB0 numbering, and obeying LE ordering.*

Crucial to understanding why the subtraction from 1,3,7,15 is present is because the Power ISA numbers VSX Registers elements also in MSB0 order. SVP64 very specifically numbers elements in **LSB0** order with the first element (numbered zero) being at the bitwise-numbered **LSB** end of the register, where VSX does the reverse: places the numerically-*highest* (last-numbered) element at the LSB end of the register.

```
#pragma pack
typedef union {
    // these do NOT match their Power ISA VSX numbering directly, they are all reversed
    // bytes[15] is actually VSR.byte[0] for example.  if this convention is not
    // followed then everything ends up in the wrong place
    uint8_t  bytes[16]; // elwidth 8, QTY 16 FIXED total
    uint16_t hwords[8]; // elwidth 16, QTY 8 FIXED total
    uint32_t words[4]; // elwidth 32, QTY 8 FIXED total
```

```
    uint64_t dwords[2]; // elwidth 64, QTY 2 FIXED total
    uint8_t actual_bytes[16]; // totals 128-bit
} el_reg_t;

elreg_t VSR_regfile[64];

static void check_num_elements(int elt, int width) {
    switch (width) {
        case 64: assert elt < 2;
        case 32: assert elt < 4;
        case 16: assert elt < 8;
        case 8 : assert elt < 16;
    }
}
void get_VSR_element(el_reg_t* el, int gpr, int elt, int width) {
    check_num_elements(elt, width);
    switch (width) {
        case 64: el->dwords[0] = VSR_regfile[gpr].dwords[1-elt];
        case 32: el->words[0] = VSR_regfile[gpr].words[3-elt];
        case 16: el->hwords[0] = VSR_regfile[gpr].hwords[7-elt];
        case 8 : el->bytes[0] = VSR_regfile[gpr].bytes[15-elt];
    }
}
void set_VSR_element(el_reg_t* el, int gpr, int elt, int width) {
    check_num_elements(elt, width);
    switch (width) {
        case 64: VSR_regfile[gpr].dwords[1-elt] = el->dwords[0];
        case 32: VSR_regfile[gpr].words[3-elt] = el->words[0];
        case 16: VSR_regfile[gpr].hwords[7-elt] = el->hwords[0];
        case 8 : VSR_regfile[gpr].bytes[15-elt] = el->bytes[0];
    }
}
```

For VSR Registers one key difference is that the overlay of different element widths is clearly a *bounded static quantity*, whereas for Simple-V the elements are unrestrained and permitted to flow into *successive underlying Scalar registers*. This difference is absolutely critical to a full understanding of the entire Simple-V paradigm and why element-ordering, bit-numbering *and register numbering* are all so strictly defined.

Implementations are not permitted to violate the Canonical definition. Software will be critically relying on the wrapped (overflow) behaviour inherently implied by the unbounded variable-length c arrays.

Illustrating the exact same loop with the exact same effect as achieved by Simple-V we are first forced to create wrapper functions, to cater for the fact that VSR register elements are static bounded:

```
int calc_VSR_reg_offs(int elt, int width) {
    switch (width) {
        case 64: return floor(elt / 2);
        case 32: return floor(elt / 4);
        case 16: return floor(elt / 8);
        case 8 : return floor(elt / 16);
    }
}
int calc_VSR_elt_offs(int elt, int width) {
    switch (width) {
        case 64: return (elt % 2);
        case 32: return (elt % 4);
        case 16: return (elt % 8);
```

```
            case 8 : return (elt % 16);
        }
    }
    void _set_VSR_element(el_reg_t* el, int gpr, int elt, int width) {
        int new_elt = calc_VSR_elt_offs(elt, width);
        int new_reg = calc_VSR_reg_offs(elt, width);
        set_VSR_element(el, gpr+new_reg, new_elt, width);
    }
```

And finally use these functions:

```
    # VSX-add RT, RA, RB using the "uint64_t" union member "hwords"
    for i in range(VL):
         el_reg_t result, ra, rb;
        _get_VSR_element(&ra, RA, i, 16);
        _get_VSR_element(&rb, RB, i, 16);
         result.hwords[0] = ra.hwords[0] + rb.hwords[0]; // use array 0 elements
        _set_VSR_element(&result, RT, i, 16);
```

6.1.7 Scalar Identity Behaviour

SVP64 is designed so that when the prefix is all zeros, and VL=1, no effect or influence occurs (no augmentation) such that all standard Power ISA v3.0/v3.1 instructions covered by the prefix are "unaltered". This is termed `scalar identity behaviour` (based on the mathematical definition for "identity", as in, "identity matrix" or better "identity transformation").

Note that this is completely different from when VL=0. VL=0 turns all operations under its influence into `nops` (regardless of the prefix) whereas when VL=1 and the SV prefix is all zeros, the operation simply acts as if SV had not been applied at all to the instruction (an "identity transformation").

The fact that `VL` is dynamic and can be set to any value at runtime based on program conditions and behaviour means very specifically that `scalar identity behaviour` is **not** a redundant encoding. If the only means by which VL could be set was by way of static-compiled immediates then this assertion would be false. VL should not be confused with MAXVL when understanding this key aspect of SimpleV.

6.1.8 Register Naming and size

As indicated above SV Registers are simply the GPR, FPR and CR register files extended linearly to larger sizes; SV Vectorisation iterates sequentially through these registers (LSB0 sequential ordering from 0 to VL-1).

Where the integer regfile in standard scalar Power ISA v3.0B/v3.1B is r0 to r31, SV extends this as r0 to r127. Likewise FP registers are extended to 128 (fp0 to fp127), and CR Fields are extended to 128 entries, CR0 thru CR127.

The names of the registers therefore reflects a simple linear extension of the Power ISA v3.0B / v3.1B register naming, and in hardware this would be reflected by a linear increase in the size of the underlying SRAM used for the regfiles.

Note: when an EXTRA field (defined below) is zero, SV is deliberately designed so that the register fields are identical to as if SV was not in effect i.e. under these circumstances (EXTRA=0) the register field names RA, RB etc. are interpreted and treated as v3.0B / v3.1B scalar registers. This is part of `scalar identity behaviour` described above.

Condition Register(s)

The Scalar Power ISA Condition Register is a 64 bit register where the top 32 MSBs (numbered 0:31 in MSB0 numbering) are not used. This convention is *preserved* in SVP64 and an additional 15 Condition Registers provided in order to store the new CR Fields, CR8-CR15, CR16-CR23 etc. sequentially. The top 32 MSBs

in each new SVP64 Condition Register are *also* not used: only the bottom 32 bits (numbered 32:63 in MSB0 numbering).

Programmer's note: using `sv.mfcr` *without element-width overrides to take into account the fact that the top 32 MSBs are zero and thus effectively doubling the number of GPR registers required to hold all 128 CR Fields would seem the only option because a source elwidth override to 32-bit would take only the bottom 16 LSBs of the Condition Register and set the top 16 LSBs to zeros. However in this case it is possible to use destination element-width overrides (for* `sv.mfcr`*. source overrides would be used on the GPR of* `sv.mtocrf`*), whereupon truncation of the 64-bit Condition Register(s) occurs, throwing away the zeros and storing the remaining (valid, desired) 32-bit values sequentially into (LSB0-convention) lower-numbered and upper-numbered halves of GPRs respectively. The programmer is expected to be aware however that the full width of the entire 64-bit Condition Register is considered to be "an element". This is* ***not*** *like any other Condition-Register instructions because all other CR instructions, on closer investigation, will be observed to all be CR-bit or CR-Field related. Thus a* `VL` *of 16 must be used*

Condition Register Fields as Predicate Masks

Condition Register Fields perform an additional duty in Simple-V: they are used for Predicate Masks. ARM's Scalar Instruction Set calls single-bit predication "Conditional Execution", and utilises Condition Codes for exactly this purpose to solve the problem caused by Branch Speculation. In a Vector ISA context the concept of Predication is naturally extended from single-bit to multi-bit, and the (well-known) benefits become all the more critical given that parallel branches in Vector ISAs are impossible (even a Vector ISA can only have Scalar branches).

However the Scalar Power ISA does not have Conditional Execution (for which, if it had ever been considered, Condition Register bits would be a perfect natural fit). Thus, when adding Predication using CR Fields via Simple-V it becomes a somewhat disruptive addition to the Power ISA.

To ameliorate this situation, particularly for pre-existing Hardware designs implementing up to Scalar Power ISA v3.1, some rules are set that allow those pre-existing designs not to require heavy modification to their existing Scalar pipelines. These rules effectively allow Hardware Architects to add the additional CR Fields CR8 to CR127 as if they were an **entirely separate register file**.

- any instruction involving more than 1 source 1 destination where one of the operands is a Condition Register is prohibited from using registers from both the CR0-7 group and the CR8-127 group at the same time.
- any instruction involving 1 source 1 destination where either the source or the destination is a Condition Register is prohibited from setting CR0-7 as a Vector.
- prohibitions are required to be enforced by raising Illegal Instruction Traps

Examples of permitted instructions:

```
sv.crand *cr8.eq, *cr16.le, *cr40.so # all CR8-CR127
sv.mfcr cr5, *cr40                   # only one source (CR40) copied to CR5
sv.mfcr *cr16, cr40                  # Vector-Splat CR40 onto CR16,17,18...
sv.mfcr *cr16, cr3                  # Vector-Splat CR3 onto CR16,17,18...
```

Examples of prohibited instructions:

```
sv.mfcr *cr0, cr40        # Vector-Splat onto CR0,1,2
sv.crand cr7, cr9, cr10   # crosses over between CR0-7 and CR8-127
```

6.1.9 Future expansion.

With the way that EXTRA fields are defined and applied to register fields, future versions of SV may involve 256 or greater registers in some way as long as the reputation of Power ISA for full backwards binary interoperability is preserved. Backwards binary compatibility may be achieved with a PCR bit (Program Compatibility Register) or an MSR bit analogous to SF. Further discussion is out of scope for this version of SVP64.

Additionally, a future variant of SVP64 will be applied to the Scalar (Quad-precision and 128-bit) VSX instructions. Element-width overrides are an opportunity to expand a future version of the Power ISA to 256-bit, 512-bit and 1024-bit operations, as well as doubling or quadrupling the number of VSX registers to 128 or 256. Again further discussion is out of scope for this version of SVP64.

6.1.10 SVP64 Remapped Encoding (RM[0:23])

In the SVP64 Vector Prefix spaces, the 24 bits 8-31 are termed RM. Bits 32-37 are the Primary Opcode of the Suffix "Defined Word". 38-63 are the remainder of the Defined Word. Note that the new EXT232-263 SVP64 area it is obviously mandatory that bit 32 is required to be set to 1.

0-5	6	7	8-31	32-37	38-64	Description
PO	0	1	RM[0:23]	1nnnnn	xxxxxxxx	SVP64:EXT232-263
PO	1	1	RM[0:23]	nnnnnn	xxxxxxxx	SVP64:EXT000-063

It is important to note that unlike EXT1xx 64-bit prefixed instructions there is insufficient space in RM to provide identification of any SVP64 Fields without first partially decoding the 32-bit suffix. Similar to the "Forms" (X-Form, D-Form) the RM format is individually associated with every instruction. However this still does not adversely affect Multi-Issue Decoding because the identification of the *length* of anything in the 64-bit space has been kept brutally simple (EXT009), and further decoding of any number of 64-bit Encodings in parallel at that point is fully independent.

Extreme caution and care must be taken when extending SVP64 in future, to not create unnecessary relationships between prefix and suffix that could complicate decoding, adding latency.

6.1.11 Common RM fields

The following fields are common to all Remapped Encodings:

Field Name	Field bits	Description
MASKMODE	0	Execution (predication) Mask Kind
MASK	1:3	Execution Mask
SUBVL	8:9	Sub-vector length

The following fields are optional or encoded differently depending on context after decoding of the Scalar suffix:

Field Name	Field bits	Description
ELWIDTH	4:5	Element Width
ELWIDTH_SRC	6:7	Element Width for Source
EXTRA	10:18	Register Extra encoding
MODE	19:23	changes Vector behaviour

- MODE changes the behaviour of the SV operation (result saturation, mapreduce)
- SUBVL groups elements together into vec2, vec3, vec4 for use in 3D and Audio/Video DSP work
- ELWIDTH and ELWIDTH_SRC overrides the instruction's destination and source operand width
- MASK (and MASK_SRC) and MASKMODE provide predication (two types of sources: scalar INT and Vector CR).
- Bits 10 to 18 (EXTRA) are further decoded depending on the RM category for the instruction, which is determined only by decoding the Scalar 32 bit suffix.

Similar to Power ISA X-Form etc. EXTRA bits are given designations, such as RM-1P-3S1D which indicates for this example that the operation is to be single-predicated and that there are 3 source operand EXTRA tags and one destination operand tag.

Note that if ELWIDTH != ELWIDTH_SRC this may result in reduced performance or increased latency in some implementations due to lane-crossing.

6.1.12 Mode

Mode is an augmentation of SV behaviour. Different types of instructions have different needs, similar to Power ISA v3.1 64 bit prefix 8LS and MTRR formats apply to different instruction types. Modes include Reduction, Iteration, arithmetic saturation, and Fail-First. More specific details in each section and in the {SVP64 Appendix}

- For condition register operations see {Condition Register Fields Mode}
- For LD/ST Modes, see {Load/Store Mode}.
- For Branch modes, see {Branch Mode}
- For arithmetic and logical, see {Arithmetic Mode}

6.1.13 ELWIDTH Encoding

Default behaviour is set to 0b00 so that zeros follow the convention of `scalar identity behaviour`. In this case it means that elwidth overrides are not applicable. Thus if a 32 bit instruction operates on 32 bit, `elwidth=0b00` specifies that this behaviour is unmodified. Likewise when a processor is switched from 64 bit to 32 bit mode, `elwidth=0b00` states that, again, the behaviour is not to be modified.

Only when elwidth is nonzero is the element width overridden to the explicitly required value.

6.1.13.1 Elwidth for Integers:

Value	Mnemonic	Description
00	DEFAULT	default behaviour for operation
01	`ELWIDTH=w`	Word: 32-bit integer
10	`ELWIDTH=h`	Halfword: 16-bit integer
11	`ELWIDTH=b`	Byte: 8-bit integer

This encoding is chosen such that the byte width may be computed as `8<<(3-ew)`

6.1.13.2 Elwidth for FP Registers:

Value	Mnemonic	Description
00	DEFAULT	default behaviour for FP operation
01	`ELWIDTH=f32`	32-bit IEEE 754 Single floating-point
10	`ELWIDTH=f16`	16-bit IEEE 754 Half floating-point
11	`ELWIDTH=bf16`	Reserved for `bf16`

Note: `bf16` is reserved for a future implementation of SV

Note that any IEEE754 FP operation in Power ISA ending in "s" (`fadds`) shall perform its operation at **half** the ELWIDTH then padded back out to ELWIDTH. `sv.fadds/ew=f32` shall perform an IEEE754 FP16 operation that is then "padded" to fill out to an IEEE754 FP32. When ELWIDTH=DEFAULT clearly the behaviour of `sv.fadds` is performed at 32-bit accuracy then padded back out to fit in IEEE754 FP64, exactly as for Scalar v3.0B "single" FP. Any FP operation ending in "s" where ELWIDTH=f16 or ELWIDTH=bf16 is reserved and must raise an illegal instruction (IEEE754 FP8 or BF8 are not defined).

6.1.13.3 Elwidth for CRs (no meaning)

Element-width overrides for CR Fields has no meaning. The bits are therefore used for other purposes, or when Rc=1, the Elwidth applies to the result being tested (a GPR or FPR), but not to the Vector of CR Fields.

6.1.14 SUBVL Encoding

The default for SUBVL is 1 and its encoding is 0b00 to indicate that SUBVL is effectively disabled (a SUBVL for-loop of only one element). this lines up in combination with all other "default is all zeros" behaviour.

Value	Mnemonic	Subvec	Description
00	`SUBVL=1`	single	Sub-vector length of 1
01	`SUBVL=2`	vec2	Sub-vector length of 2
10	`SUBVL=3`	vec3	Sub-vector length of 3
11	`SUBVL=4`	vec4	Sub-vector length of 4

The SUBVL encoding value may be thought of as an inclusive range of a sub-vector. SUBVL=2 represents a vec2, its encoding is 0b01, therefore this may be considered to be elements 0b00 to 0b01 inclusive.

Effectively, SUBVL is like a SIMD multiplier: instead of just 1 element operation issued, SUBVL element operations are issued (as an inner loop). The key difference between VL looping and SUBVL looping is that predication bits are applied per **group**, rather than by individual element.

Directly related to `subvl` is the `pack` and `unpack` Mode bits of `SVSTATE`.

6.1.15 MASK/MASK_SRC & MASKMODE Encoding

One bit (`MASKMODE`) indicates the mode: CR or Int predication. The two types may not be mixed.

Special note: to disable predication this field must be set to zero in combination with Integer Predication also being set to 0b000. this has the effect of enabling "all 1s" in the predicate mask, which is equivalent to "not having any predication at all".

`MASKMODE` may be set to one of 2 values:

Value	Description
0	MASK/MASK_SRC are encoded using Integer Predication
1	MASK/MASK_SRC are encoded using CR-based Predication

Integer Twin predication has a second set of 3 bits that uses the same encoding thus allowing either the same register (r3, r10 or r31) to be used for both src and dest, or different regs (one for src, one for dest).

Likewise CR based twin predication has a second set of 3 bits, allowing a different test to be applied.

Note that it cannot necessarily be assumed that Predicate Masks (whether INT or CR) are read in full *before* the operations proceed. In practice (for CR Fields) this creates an unnecessary block on parallelism, prohibiting "Vector Chaining". Therefore, it is up to the programmer to ensure that the CR field Elements used as Predicate Masks are not overwritten by any parallel Vector Loop. Doing so results in **UNDEFINED** behaviour, according to the definition outlined in the Power ISA v3.0B Specification.

Hardware Implementations are therefore free and clear to delay reading of individual CR fields until the actual predicated element operation needs to take place, safe in the knowledge that no programmer will have issued a Vector Instruction where previous elements could have overwritten (destroyed) not-yet-executed CR-Predicated element operations. This particularly is an issue when using REMAP, as the order in which CR-Field-based

Predicate Mask bits could be read on a per-element execution basis could well conflict with the order in which prior elements wrote to the very same CR Field.

Additionally Programmers should avoid using r3 r10 or r30 as destination registers when these are also used as a Predicate Mask. Doing so is again UNDEFINED behaviour.

6.1.15.1 Integer Predication (MASKMODE=0)

When the predicate mode bit is zero the 3 bits are interpreted as below. Twin predication has an identical 3 bit field similarly encoded.

`MASK` and `MASK_SRC` may be set to one of 8 values, to provide the following meaning:

Value	Mnemonic	Element `i` enabled if:
000	ALWAYS	predicate effectively all 1s
001	1 << R3	`i == R3`
010	R3	`R3 & (1 << i)` is non-zero
011	~R3	`R3 & (1 << i)` is zero
100	R10	`R10 & (1 << i)` is non-zero
101	~R10	`R10 & (1 << i)` is zero
110	R30	`R30 & (1 << i)` is non-zero
111	~R30	`R30 & (1 << i)` is zero

r10 and r30 are at the high end of temporary and unused registers, so as not to interfere with register allocation from ABIs.

6.1.15.2 CR-based Predication (MASKMODE=1)

When the predicate mode bit is one the 3 bits are interpreted as below. Twin predication has an identical 3 bit field similarly encoded.

`MASK` and `MASK_SRC` may be set to one of 8 values, to provide the following meaning:

Value	Mnemonic	Element `i` is enabled if
000	lt	`CR[offs+i].LT` is set
001	nl/ge	`CR[offs+i].LT` is clear
010	gt	`CR[offs+i].GT` is set
011	ng/le	`CR[offs+i].GT` is clear
100	eq	`CR[offs+i].EQ` is set
101	ne	`CR[offs+i].EQ` is clear
110	so/un	`CR[offs+i].FU` is set
111	ns/nu	`CR[offs+i].FU` is clear

`offs` is defined as CR32 (4x8) so as to mesh cleanly with Vectorised Rc=1 operations (see below). Rc=1 operations start from CR8 (TBD).

The CR Predicates chosen must start on a boundary that Vectorised CR operations can access cleanly, in full. With EXTRA2 restricting starting points to multiples of 8 (CR0, CR8, CR16...) both Vectorised Rc=1 and CR Predicate Masks have to be adapted to fit on these boundaries as well.

6.1.16 Extra Remapped Encoding

Shows all instruction-specific fields in the Remapped Encoding `RM[10:18]` for all instruction variants. Note that due to the very tight space, the encoding mode is *not* included in the prefix itself. The mode is "applied", similar to Power ISA "Forms" (X-Form, D-Form) on a per-instruction basis, and, like "Forms" are given a designation (below) of the form `RM-nP-nSnD`. The full list of which instructions use which remaps is here {SVP64 Augmentation Table}.

Please note the following:

```
Machine-readable CSV files have been autogenerated which will make the
task of creating SV-aware ISA decoders, documentation, assembler tools
compiler tools Simulators documentation all aspects of SVP64 easier
and less prone to mistakes.  Please avoid manual re-creation of
information from the written specification wording in this chapter,
and use the CSV files or use the Canonical tool which creates the CSV
files, named sv_analysis.py. The information contained within
sv_analysis.py is considered to be part of this Specification, even
encoded as it is in python3.
```

The mappings are part of the SVP64 Specification in exactly the same way as X-Form, D-Form. New Scalar instructions added to the Power ISA will need a corresponding SVP64 Mapping, which can be derived by-rote from examining the Register "Profile" of the instruction.

There are two categories: Single and Twin Predication. Due to space considerations further subdivision of Single Predication is based on whether the number of src operands is 2 or 3. With only 9 bits available some compromises have to be made.

- `RM-1P-3S1D` Single Predication dest/src1/2/3, applies to 4-operand instructions (fmadd, isel, madd).
- `RM-1P-2S1D` Single Predication dest/src1/2 applies to 3-operand instructions (src1 src2 dest)
- `RM-2P-1S1D` Twin Predication (src=1, dest=1)
- `RM-2P-2S1D` Twin Predication (src=2, dest=1) primarily for LDST (Indexed)
- `RM-2P-1S2D` Twin Predication (src=1, dest=2) primarily for LDST Update

6.1.16.1 RM-1P-3S1D

Field Name	Field bits	Description
Rdest_EXTRA2	`10:11`	extends Rdest (R*_EXTRA2 Encoding)
Rsrc1_EXTRA2	`12:13`	extends Rsrc1 (R*_EXTRA2 Encoding)
Rsrc2_EXTRA2	`14:15`	extends Rsrc2 (R*_EXTRA2 Encoding)
Rsrc3_EXTRA2	`16:17`	extends Rsrc3 (R*_EXTRA2 Encoding)
EXTRA2_MODE	`18`	used by `divmod2du` and `maddedu` for RS

These are for 3 operand in and either 1 or 2 out instructions. 3-in 1-out includes `madd RT,RA,RB,RC`. (DRAFT) instructions such as `maddedu` have an implicit second destination, RS, the selection of which is determined by bit 18.

6.1.16.2 RM-1P-2S1D

Field Name	Field bits	Description
Rdest_EXTRA3	`10:12`	extends Rdest
Rsrc1_EXTRA3	`13:15`	extends Rsrc1
Rsrc2_EXTRA3	`16:18`	extends Rsrc3

These are for 2 operand 1 dest instructions, such as `add RT, RA, RB`. However also included are unusual instructions with an implicit dest that is identical to its src reg, such as `rlwinmi`.

Normally, with instructions such as `rlwinmi`, the scalar v3.0B ISA would not have sufficient bit fields to allow an alternative destination. With SV however this becomes possible. Therefore, the fact that the dest is implicitly also a src should not mislead: due to the *prefix* they are different SV regs.

- `rlwimi RA, RS, ...`
- Rsrc1_EXTRA3 applies to RS as the first src
- Rsrc2_EXTRA3 applies to RA as the secomd src
- Rdest_EXTRA3 applies to RA to create an **independent** dest.

With the addition of the EXTRA bits, the three registers each may be *independently* made vector or scalar, and be independently augmented to 7 bits in length.

6.1.16.3 RM-2P-1S1D/2S

Field Name	Field bits	Description
Rdest_EXTRA3	`10:12`	extends Rdest
Rsrc1_EXTRA3	`13:15`	extends Rsrc1
MASK_SRC	`16:18`	Execution Mask for Source

`RM-2P-2S` is for `stw` etc. and is Rsrc1 Rsrc2.

Field Name	Field bits	Description
Rsrc1_EXTRA3	`10:12`	extends Rsrc1
Rsrc2_EXTRA3	`13:15`	extends Rsrc2
MASK_SRC	`16:18`	Execution Mask for Source

6.1.16.4 RM-1P-2S1D

single-predicate, three registers (2 read, 1 write)

Field Name	Field bits	Description
Rdest_EXTRA3	`10:12`	extends Rdest
Rsrc1_EXTRA3	`13:15`	extends Rsrc1
Rsrc2_EXTRA3	`16:18`	extends Rsrc2

6.1.16.5 RM-2P-2S1D/1S2D/3S

The primary purpose for this encoding is for Twin Predication on LOAD and STORE operations. see {Load/Store Mode} for detailed anslysis.

RM-2P-2S1D:

Field Name	Field bits	Description
Rdest_EXTRA2	`10:11`	extends Rdest (R*_EXTRA2 Encoding)
Rsrc1_EXTRA2	`12:13`	extends Rsrc1 (R*_EXTRA2 Encoding)
Rsrc2_EXTRA2	`14:15`	extends Rsrc2 (R*_EXTRA2 Encoding)
MASK_SRC	`16:18`	Execution Mask for Source

RM-2P-1S2D:

For RM-2P-1S2D dest2 is in bits 14:15

Field Name	Field bits	Description
Rdest_EXTRA2	10:11	extends Rdest (R*_EXTRA2 Encoding)
Rsrc1_EXTRA2	12:13	extends Rsrc1 (R*_EXTRA2 Encoding)
Rdest2_EXTRA2	14:15	extends Rdest22 (R*_EXTRA2 Encoding)
MASK_SRC	16:18	Execution Mask for Source

RM-2P-3S:

Also that for RM-2P-3S (to cover `stdx` etc.) the names are switched to 3 src: Rsrc1_EXTRA2, Rsrc2_EXTRA2, Rsrc3_EXTRA2.

Field Name	Field bits	Description
Rsrc1_EXTRA2	10:11	extends Rsrc1 (R*_EXTRA2 Encoding)
Rsrc2_EXTRA2	12:13	extends Rsrc2 (R*_EXTRA2 Encoding)
Rsrc3_EXTRA2	14:15	extends Rsrc3 (R*_EXTRA2 Encoding)
MASK_SRC	16:18	Execution Mask for Source

Note also that LD with update indexed, which takes 2 src and creates 2 dest registers (e.g. `lhaux RT,RA,RB`), does not have room for 4 registers and also Twin Predication. Therefore these are treated as RM-2P-2S1D and the src spec for RA is also used for the same RA as a dest.

Note that if ELWIDTH != ELWIDTH_SRC this may result in reduced performance or increased latency in some implementations due to lane-crossing.

6.1.17 R*_EXTRA2/3

EXTRA is the means by which two things are achieved:

1. Registers are marked as either Vector *or Scalar*
2. Register field numbers (limited typically to 5 bit) are extended in range, both for Scalar and Vector.

The register files are therefore extended:

- INT (GPR) is extended from r0-31 to r0-127
- FP (FPR) is extended from fp0-32 to fp0-fp127
- CR Fields are extended from CR0-7 to CR0-127

However due to pressure in `RM.EXTRA` not all these registers are accessible by all instructions, particularly those with a large number of operands (`madd`, `isel`).

In the following tables register numbers are constructed from the standard v3.0B / v3.1B 32 bit register field (RA, FRA) and the EXTRA2 or EXTRA3 field from the SV Prefix, determined by the specific RM-xx-yyyy designation for a given instruction. The prefixing is arranged so that interoperability between prefixing and nonprefixing of scalar registers is direct and convenient (when the EXTRA field is all zeros).

A pseudocode algorithm explains the relationship, for INT/FP (see {SVP64 Appendix} for CRs)

```
if extra3_mode:
    spec = EXTRA3
else:
    spec = EXTRA2 << 1 # same as EXTRA3, shifted
if spec[0]: # vector
```

```
        return (RA << 2) | spec[1:2]
    else:         # scalar
        return (spec[1:2] << 5) | RA
```

Future versions may extend to 256 by shifting Vector numbering up. Scalar will not be altered.

Note that in some cases the range of starting points for Vectors is limited.

6.1.17.1 INT/FP EXTRA3

If EXTRA3 is zero, maps to "scalar identity" (scalar Power ISA field naming).

Fields are as follows:

- Value: R_EXTRA3
- Mode: register is tagged as scalar or vector
- Range/Inc: the range of registers accessible from this EXTRA encoding, and the "increment" (accessibility). "/4" means that this EXTRA encoding may only give access (starting point) every 4th register.
- MSB..LSB: the bit field showing how the register opcode field combines with EXTRA to give (extend) the register number (GPR)

Encoding shown in LSB0: MSB down to LSB (MSB 6..0 LSB)

Value	Mode	Range/Inc	6..0
000	Scalar	r0-r31/1	0b00 RA
001	Scalar	r32-r63/1	0b01 RA
010	Scalar	r64-r95/1	0b10 RA
011	Scalar	r96-r127/1	0b11 RA
100	Vector	r0-r124/4	RA 0b00
101	Vector	r1-r125/4	RA 0b01
110	Vector	r2-r126/4	RA 0b10
111	Vector	r3-r127/4	RA 0b11

6.1.17.2 INT/FP EXTRA2

If EXTRA2 is zero will map to "scalar identity behaviour" i.e Scalar Power ISA register naming:

Encoding shown in LSB0: MSB down to LSB (MSB 6..0 LSB)

Value	Mode	Range/inc	6..0
00	Scalar	r0-r31/1	0b00 RA
01	Scalar	r32-r63/1	0b01 RA
10	Vector	r0-r124/4	RA 0b00
11	Vector	r2-r126/4	RA 0b10

Note that unlike in EXTRA3, in EXTRA2:

- the GPR Vectors may only start from `r0, r2, r4, r6, r8` and likewise FPR Vectors.
- the GPR Scalars may only go from `r0, r1, r2.. r63` and likewise FPR Scalars.

as there is insufficient bits to cover the full range.

6.1.17.3 CR Field EXTRA3

CR Field encoding is essentially the same but made more complex due to CRs being bit-based, because the application of SVP64 element-numbering applies to the CR *Field* numbering not the CR register *bit* numbering. Note that Vectors may only start from `CR0, CR4, CR8, CR12, CR16, CR20...` and Scalars may only go from `CR0, CR1, ... CR31`

Encoding shown in LSB0: MSB down to LSB (MSB 8..5 4..2 1..0 LSB), BA ranges are in MSB0.

For a 5-bit operand (BA, BB, BT):

Value	Mode	Range/Inc	8..5	4..2	1..0
000	Scalar	`CR0-CR7`/1	0b0000	BA[0:2]	BA[3:4]
001	Scalar	`CR8-CR15`/1	0b0001	BA[0:2]	BA[3:4]
010	Scalar	`CR16-CR23`/1	0b0010	BA[0:2]	BA[3:4]
011	Scalar	`CR24-CR31`/1	0b0011	BA[0:2]	BA[3:4]
100	Vector	`CR0-CR112`/16	BA[0:2] 0	0b000	BA[3:4]
101	Vector	`CR4-CR116`/16	BA[0:2] 0	0b100	BA[3:4]
110	Vector	`CR8-CR120`/16	BA[0:2] 1	0b000	BA[3:4]
111	Vector	`CR12-CR124`/16	BA[0:2] 1	0b100	BA[3:4]

For a 3-bit operand (e.g. BFA):

Value	Mode	Range/Inc	6..3	2..0
000	Scalar	`CR0-CR7`/1	0b0000	BFA
001	Scalar	`CR8-CR15`/1	0b0001	BFA
010	Scalar	`CR16-CR23`/1	0b0010	BFA
011	Scalar	`CR24-CR31`/1	0b0011	BFA
100	Vector	`CR0-CR112`/16	BFA 0	0b000
101	Vector	`CR4-CR116`/16	BFA 0	0b100
110	Vector	`CR8-CR120`/16	BFA 1	0b000
111	Vector	`CR12-CR124`/16	BFA 1	0b100

6.1.17.4 CR EXTRA2

CR encoding is essentially the same but made more complex due to CRs being bit-based, because the application of SVP64 element-numbering applies to the CR *Field* numbering not the CR register *bit* numbering. Note that Vectors may only start from CR0, CR8, CR16, CR24, CR32...

Encoding shown in LSB0: MSB down to LSB (MSB 8..5 4..2 1..0 LSB), BA ranges are in MSB0.

For a 5-bit operand (BA, BB, BC):

Value	Mode	Range/Inc	8..5	4..2	1..0
00	Scalar	`CR0-CR7`/1	0b0000	BA[0:2]	BA[3:4]
01	Scalar	`CR8-CR15`/1	0b0001	BA[0:2]	BA[3:4]
10	Vector	`CR0-CR112`/16	BA[0:2] 0	0b000	BA[3:4]
11	Vector	`CR8-CR120`/16	BA[0:2] 1	0b000	BA[3:4]

For a 3-bit operand (e.g. BFA):

Value	Mode	Range/Inc	6..3	2..0
00	Scalar	CR0-CR7/1	0b0000	BFA
01	Scalar	CR8-CR15/1	0b0001	BFA
10	Vector	CR0-CR112/16	BFA 0	0b000
11	Vector	CR8-CR120/16	BFA 1	0b000

6.1.18 Appendix

Now at its own page: {SVP64 Appendix}

[[!tag standards]]

Chapter 7

SPRs

7.1 SPRs

The full list of SPRs for Simple-V is:

SPR	Width	Description
SVSTATE	64-bit	Zero-Overhead Loop Architectural State
SVLR	64-bit	SVSTATE equivalent of LR-to-PC
SVSHAPE0	32-bit	REMAP Shape 0
SVSHAPE1	32-bit	REMAP Shape 1
SVSHAPE2	32-bit	REMAP Shape 2
SVSHAPE3	32-bit	REMAP Shape 3

Future versions of Simple-V will have at least 7 more SVSTATE SPRs, in a small "stack", as part of a full Zero-Overhead Loop Control subsystem.

7.1.1 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 (srcstep reorder)
54	unpack	UNPACK (dststep order)

Field	Name	Description
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 svstep
- 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/ssubstep VL/SUBVL loop-ordering is inverted.
- UnPack - if set then dststep/dsubstep 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(step/hphint)` is equal *before REMAP* are in the same parallelism "group", for both `srcstep` and `dststep`. In Vertical First Mode hardware **MUST** respect Strict Program Order but is permitted to merge multiple scalar loops into parallel batches, if Reservation Station resources are sufficient. 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 - 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 explicitly 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.

SVme REMAP area

Each bit of `SVSTATE.SVme` indicates whether the SVSHAPE (0-3) is active and to which register the REMAP applies. The application goes by *assembler operand names* on a per-mnemonic basis. Some instructions may have `RT` as a source and as a destination: REMAP applies **separately** to each use in this case. Also for Load/Store with Update the Effective Address (stored in EA) also may be separately REMAPed from RA as a source operand.

bit	applies	register applied
46	mi0	source RA / FRA / BA / BFA / RT / FRT
45	mi1	source RB / FRB / BB
44	mi2	source RC / FRC / BC
43	mo0	result RT / FRT / BT / BF
42	mo1	result Effective Address (RA) / FRS / RS

Max Vector Length (maxvl)

MAXVECTORLENGTH is a static (immediate-operand only) compile-time declaration of the maximum number of elements in a Vector. 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.

MAXVL is normally (in other True-Scalable Vector ISAs) an Architecturally-defined quantity related indirectly to the total available number of bits in the Vector Register File. Cray Vectors had a Hardware-Architectural set limit of MAXVL=64. RISC-V RVV has MAXVL defined in terms of a Silicon-Partner-selectable fixed number of bits. MAXVL in Simple-V is set in terms of the number of *elements* and may change at runtime.

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.

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, despite these being easy for a compiler to statically detect and potentially express. `hphint` is such an expression, declaring that elements within a batch are independent of each other (no Register *or Memory* Hazards).

Elements are considered to be in the same source batch if they have the same value of `FLOOR(srcstep/hphint)`. Likewise in the same destination batch for the same value `FLOOR(dststep/hphint)`. Four key observations here:

1. predication is **not** involved here. the number of actual elements involved is considered *before* predicate masks are applied.
2. twin predication can result in srcstep and dststep being in different batches
3. batch evaluation is done *before* REMAP, making Hazard elimination easier for Multi-Issue systems.
4. `hphint` is *not* limited to power-of-two. Hardware implementors may choose a lower parallelism hint up to `hphint` and may find power-of-two more convenient.

Regarding (4): if a smaller hint is chosen by hardware, actual parallelism (Dependency Hazard relaxation) must **never** exceed `hphint` and must still respect the batch boundaries, even if this results in just one element being considered Hazard-independent. Even under these circumstances Multi-Issue Register-renaming is possible, to introduce parallelism by a different route.

Hardware Architect note: each element within the same group may be treated as 100% independent from any other element within that group, and therefore neither Register Hazards nor Memory Hazards inter-element exist, but crucially inter-group definitely remains. This makes implementation far easier on resources because the Hazard Dependencies are effectively at a much coarser granularity than a single register. With element-width overrides extending down to the byte level reducing Dependency Hazard hardware complexity becomes even more important.

`hphint` may legitimately be set greater than `MAXVL`. This indicates to Multi-Issue hardware that even though MAXVL is relatively small the batches are *still independent* and therefore if Multi-Issue hardware chooses to allocate several batches up to `MAXVL` in size they are still independent, even if Register-renaming is deployed. This helps greatly simplify Multi-Issue systems by significantly reducing Hazards.

Considerable care must be taken when setting `hphint`. Matrix Outer Product could produce corrupted results if `hphint` is set to greater than the innermost loop depth. Parallel Reduction, DCT and FFT REMAP all are similarly critically affected by `hphint` in ways that if used correctly greatly increases ease of parallelism but if done incorrectly will also result in data corruption. Reduction/Iteration also requires care to correctly declare in `hphint` how many elements are independent. In the case of most Reduction use-cases the answer is almost certainly "none".

`hphint` must never be set on Atomic Memory operations, Cache-Inhibited Memory operations, or Load-Reservation Store-Conditional. Also if Load-with-Update Data-Dependent Fail-First is ever used for linked-list pointer-chasing, `hphint` should again definitely be disabled. Failure to do so results in `UNDEFINED` behaviour.

`hphint` may only be ignored by Hardware Implementors as long as full element-level Register and Memory Hazards are implemented *in full* (including right down to individual bytes of each register for when elwidth=8/16/32). In other words if `hphint` is to be ignored then implementations must consider the situation as if `hphint=0`.

Horizontal Parallelism in Vertical-First Mode

Setting `hphint` with Vertical-First is perfectly legitimate. Under these circumstances single-element strict Program Execution Order must be preserved at all times, but should there be a small enough program loop, than Out-of-Order Hardware may take the opportunity to *merge* consecutive element-based instructions into the *same Reservation Stations*, for multiple operations to be passed to massive-wide back-end SIMD ALUs or Vector-Chaining ALUs. **Only** elements within the same `hphint` group (across multiple such looped instructions) may be treated as mergeable in this fashion.

Note that if the loop of Vertical-First instructions cannot fit entirely into Reservation Stations then Hardware clearly cannot exploit the above optimisation opportunity, but at least there is no harm done: the loop is still correctly executed as Scalar instructions. Programmers do need to be aware though that short loops on some Hardware Implementations can be made considerably faster than on other Implementations.

7.1.2 SVLR

SV Link Register, exactly analogous to LR (Link Register) may be used for temporary storage of SVSTATE, and, in particular, Vectorised Branch-Conditional instructions may interchange SVLR and SVSTATE whenever LR and NIA are.

Note that there is no equivalent Link variant of SVREMAP or SVSHAPE0-3 (it would be too costly), so SVLR has limited applicability: REMAP SPRs must be saved and restored explicitly.

[[!tag standards]]

Chapter 8

Arithmetic Mode

8.1 Normal SVP64 Modes, for Arithmetic and Logical Operations

- https://bugs.libre-soc.org/show_bug.cgi?id=574
- https://bugs.libre-soc.org/show_bug.cgi?id=558#c47
- https://bugs.libre-soc.org/show_bug.cgi?id=936 write on failfirst
- {SVP64 Chapter}

Normal SVP64 Mode covers Arithmetic and Logical operations to provide suitable additional behaviour. The Mode field is bits 19-23 of the {SVP64 Chapter} RM Field.

Table of contents:

[[!toc]]

8.1.1 Mode

Mode is an augmentation of SV behaviour, providing additional functionality. Some of these alterations are element-based (saturation), others are Vector-based (mapreduce, fail-on-first).

{Load/Store Mode}, {Condition Register Fields Mode} and {Branch Mode} are covered separately: the following Modes apply to Arithmetic and Logical SVP64 operations:

- **simple** mode is straight vectorisation. No augmentations: the vector comprises an array of independently created results.
- **ffirst** or data-dependent fail-on-first: see separate section. The vector may be truncated depending on certain criteria. *VL is altered as a result.*
- **sat mode** or saturation: clamps each element result to a min/max rather than overflows / wraps. Allows signed and unsigned clamping for both INT and FP.
- **reduce mode**. If used correctly, a mapreduce (or a prefix sum) is performed. See {SVP64 Appendix}. Note that there are comprehensive caveats when using this mode, and it should not be confused with the Parallel Reduction {REMAP subsystem}. Also care is needed with `hphint`.

Note that ffirst and reduce modes are not anticipated to be high-performance in some implementations. ffirst due to interactions with VL, and reduce due to it creating overlapping operations in many of its uses. simple and saturate are however inter-element independent and may easily be parallelised to give high performance, regardless of the value of VL.

The Mode table for Arithmetic and Logical operations, being bits 19-23 of SVP64 `RM`, is laid out as follows:

0-1	2	3 4	description

0-1	2	3 4	description
00	0	dz sz	simple mode
00	1	0 RG	scalar reduce mode (mapreduce)
00	1	1 /	reserved
01	inv	CR-bit	Rc=1: ffirst CR sel
01	inv	VLi RC1	Rc=0: ffirst z/nonz
10	N	dz sz	sat mode: N=0/1 u/s
11	/	/ /	reserved

Fields:

- **sz / dz** source-zeroing, destination-zeroing. if predication is enabled will put zeros into the dest (or as src in the case of twin pred) when the predicate bit is zero. Otherwise the element is ignored or skipped, depending on context.
- **zz**: both sz and dz are set equal to this flag
- **inv CR bit** just as in branches (BO) these bits allow testing of a CR bit and whether it is set (inv=0) or unset (inv=1)
- **RG** inverts the Vector Loop order (VL-1 downto 0) rather than the normal 0..VL-1
- **N** sets signed/unsigned saturation.
- **RC1** as if Rc=1, enables access to `VLi`.
- **VLi** VL inclusive: in fail-first mode, the truncation of VL *includes* the current element at the failure point rather than excludes it from the count.

For LD/ST Modes, see {Load/Store Mode}. For Condition Registers see {Condition Register Fields Mode}. For Branch modes, see {Branch Mode}.

8.1.2 Rounding, clamp and saturate

See {Audio and Video Opcodes} for relevant opcodes and use-cases.

To help ensure for example that audio quality is not compromised by overflow, "saturation" is provided, as well as a way to detect when saturation occurred if desired (Rc=1). When Rc=1 there will be a *vector* of CRs, one CR per element in the result (Note: this is different from VSX which has a single CR per block).

When N=0 the result is saturated to within the maximum range of an unsigned value. For integer ops this will be 0 to 2^elwidth-1. Similar logic applies to FP operations, with the result being saturated to maximum rather than returning INF, and the minimum to +0.0

When N=1 the same occurs except that the result is saturated to the min or max of a signed result, and for FP to the min and max value rather than returning +/- INF.

When Rc=1, the CR "overflow" bit is set on the CR associated with the element, to indicate whether saturation occurred. Note that due to the hugely detrimental effect it has on parallel processing, XER.SO is **ignored** completely and is **not** brought into play here. The CR overflow bit is therefore simply set to zero if saturation did not occur, and to one if it did. This behaviour (ignoring XER.SO) is actually optional in the SFFS Compliancy Subset: for SVP64 it is made mandatory *but only on Vectorised instructions.*

Note also that saturate on operations that set OE=1 must raise an Illegal Instruction due to the conflicting use of the CR.so bit for storing if saturation occurred. Vectorised Integer Operations that produce a Carry-Out (CA, CA32): these two bits will be `UNDEFINED` if saturation is also requested.

Note that the operation takes place at the maximum bitwidth (max of src and dest elwidth) and that truncation occurs to the range of the dest elwidth.

Programmer's Note: Post-analysis of the Vector of CRs to find out if any given element hit saturation may be done using a mapreduced CR op (cror), or by using the new crrweird instruction with Rc=1, which will transfer the required CR bits to a scalar integer and update CR0, which will allow testing the scalar integer for nonzero. See {CR Weird ops}. Alternatively, a Data-Dependent Fail-First may be used to truncate the Vector Length to non-saturated elements, greatly increasing the productivity of parallelised inner hot-loops.

8.1.3 Reduce mode

Reduction in SVP64 is similar in essence to other Vector Processing ISAs, but leverages the underlying scalar Base v3.0B operations. Thus it is more a convention that the programmer may utilise to give the appearance and effect of a Horizontal Vector Reduction. Due to the unusual decoupling it is also possible to perform prefix-sum (Fibonacci Series) in certain circumstances. Details are in the {SVP64 Appendix}

Reduce Mode should not be confused with Parallel Reduction {REMAP subsystem}. As explained in the {SVP64 Appendix} Reduce Mode switches off the check which would normally stop looping if the result register is scalar. Thus, the result scalar register, if also used as a source scalar, may be used to perform sequential accumulation. This *deliberately* sets up a chain of Register Hazard Dependencies (which advanced hardware may optimise out), whereas Parallel Reduce {REMAP subsystem} deliberately issues a Tree-Schedule of operations that may be parallelised.

Hardware architectural note: implementations may optimise out the Hazard Dependency chain as long as Sequential Program Execution Order is preserved. Easy examples include Reduction on Logical OR or AND operations.

Horizontal Parallelism Hint

`SVSTATE.hphint` declares to hardware that groups of elements up to this size are 100% independent (free of all Hazards inter-element but not inter-group). With Reduction literally creating Dependency Hazards on every element-level sub-instruction it is pretty clear that setting `hphint` *at all* would cause data corruption. However `sv.add *r0, *r4, *r0` for example clearly leaves room for four parallel elements. Programmers must be aware of this and exercise caution.

8.1.4 Data-dependent Fail-on-first

Data-dependent fail-on-first is CR-field-driven and is completely separate and distinct from LD/ST Fail-First (also known as Fault-First). Note in each case the assumption is that vector elements are required to appear to be executed in sequential Program Order. When REMAP is not active, element 0 would be the first.

Data-driven (CR-field-driven) fail-on-first activates when Rc=1 or other CR-creating operation produces a result (including cmp). Similar to Branch-Conditional, an analysis of the CR is performed and if the test fails, the vector operation terminates and discards all element operations **at and above the current one**, and VL is truncated to either the *previous* element or the current one, depending on whether VLi (VL "inclusive") is clear or set, respectively.

Thus the new VL comprises a contiguous vector of results, all of which pass the testing criteria (equal to zero, less than zero etc as defined by the CR-bit test).

Note: when VLi is clear, the behaviour at first seems counter-intuitive. A result is calculated but if the test fails it is prohibited from being actually written. This becomes intuitive again when it is remembered that the length that VL is set to is the number of written* elements, and only when VLI is set will the current element be included in that count.*

The CR-based data-driven fail-on-first is "new" and not found in ARM SVE or RVV. At the same time it is "old" because it is almost identical to a generalised form of Z80's `CPIR` instruction. It is extremely useful for reducing instruction count, however requires speculative execution involving modifications of VL to get high performance implementations. An additional mode (RC1=1) effectively turns what would otherwise be an

arithmetic operation into a type of `cmp`. The CR is stored (and the CR.eq bit tested against the `inv` field). If the CR.eq bit is equal to `inv` then the Vector is truncated and the loop ends.

VLi is only available as an option when `Rc=0` (or for instructions which do not have Rc). When set, the current element is always also included in the count (the new length that VL will be set to). This may be useful in combination with "inv" to truncate the Vector to *exclude* elements that fail a test, or, in the case of implementations of strncpy, to include the terminating zero.

In CR-based data-driven fail-on-first there is only the option to select and test one bit of each CR (just as with branch BO). For more complex tests this may be insufficient. If that is the case, a vectorised crop such as crand, cror or {CR Weird ops} crweirder may be used, and ffirst applied to the crop instead of to the arithmetic vector. Note that crops are covered by the {Condition Register Fields Mode} Mode format.

Use of Fail-on-first with Vertical-First Mode is not prohibited but is not really recommended. The effect of truncating VL may have unintended and unexpected consequences on subsequent instructions. VLi set will be fine: it is when VLi is clear that problems may be faced.

Programmer's note: `VLi` is only accessible in normal operations which in turn limits the CR field bit-testing to only `EQ/NE`. {Condition Register Fields Mode} are not so limited. Thus it is possible to use for example `sv.cror/ff=gt/vli *0,*0,*0`, which is not a `nop` because it allows Fail-First Mode to perform a test and truncate VL.

*Hardware implementor's note: effective Sequential Program Order must be preserved. Speculative Execution is perfectly permitted as long as the speculative elements are held back from writing to register files (kept in Resevation Stations), until such time as the relevant CR Field bit(s) has been analysed. All Speculative elements sequentially beyond the test-failure point **MUST** be cancelled. This is no different from standard Out-of-Order Execution and the modification effort to efficiently support Data-Dependent Fail-First within a pre-existing Multi-Issue Out-of-Order Engine is anticipated to be minimal. In-Order systems on the other hand are expected, unavoidably, to be low-performance.*

Two extremely important aspects of ffirst are:

- LDST ffirst may never set VL equal to zero. This because on the first element an exception must be raised "as normal".
- CR-based data-dependent ffirst on the other hand **can** set VL equal to zero. When VL is set zero due to the first element failing the CR bit-test, all subsequent vectorised operations are effectively `nops` which is *precisely the desired and intended behaviour*.

The second crucial aspect, compared to LDST Ffirst:

- LD/ST Failfirst may (beyond the initial first element conditions) truncate VL for any architecturally suitable reason. Beyond the first element LD/ST Failfirst is arbitrarily speculative and 100% non-deterministic.
- CR-based data-dependent first on the other hand MUST NOT truncate VL arbitrarily to a length decided by the hardware: VL MUST only be truncated based explicitly on whether a test fails. This because it is a precise Deterministic test on which algorithms can and will rely.

Floating-point Exceptions

When Floating-point exceptions are enabled VL must be truncated at the point where the Exception appears not to have occurred. If `VLi` is set then VL must include the faulting element, and thus the faulting element will always raise its exception. If however `VLi` is clear then VL **excludes** the faulting element and thus the exception will **never** be raised.

Although very strongly discouraged the Exception Mode that permits Floating Point Exception notification to arrive too late to unwind is permitted (under protest, due it violating the otherwise 100% Deterministic nature of Data-dependent Fail-first).

Use of lax FP Exception Notification Mode could result in parallel computations proceeding with invalid results that have to be explicitly detected, whereas with the strict FP Execption Mode enabled, FFirst truncates VL, allows subsequent parallel computation to avoid the exceptions entirely

8.1.5 Data-dependent fail-first on CR operations (crand etc)

Operations that actually produce or alter CR Field as a result have their own SVP64 Mode, described in {Condition Register Fields Mode}.

[[!tag standards]]

Chapter 9

Load/Store Mode

9.1 SV Load and Store

Links:

- https://bugs.libre-soc.org/show_bug.cgi?id=561
- https://bugs.libre-soc.org/show_bug.cgi?id=572
- https://bugs.libre-soc.org/show_bug.cgi?id=571
- https://bugs.libre-soc.org/show_bug.cgi?id=940 post autoincrement mode
- https://bugs.libre-soc.org/show_bug.cgi?id=1047 Data-Dependent Fail-First
- https://llvm.org/devmtg/2016-11/Slides/Emerson-ScalableVectorizationinLLVMIR.pdf
- https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc#vector-loads-and-stores
- [[ldst/discussion]]

9.1.1 Rationale

All Vector ISAs dating back fifty years have extensive and comprehensive Load and Store operations that go far beyond the capabilities of Scalar RISC and most CISC processors, yet at their heart on an individual element basis may be found to be no different from RISC Scalar equivalents.

The resource savings from Vector LD/ST are significant and stem from the fact that one single instruction can trigger a dozen (or in some microarchitectures such as Cray or NEC SX Aurora) hundreds of element-level Memory accesses.

Additionally, and simply: if the Arithmetic side of an ISA supports Vector Operations, then in order to keep the ALUs 100% occupied the Memory infrastructure (and the ISA itself) correspondingly needs Vector Memory Operations as well.

Vectorised Load and Store also presents an extra dimension (literally) which creates scenarios unique to Vector applications, that a Scalar (and even a SIMD) ISA simply never encounters. SVP64 endeavours to add the modes typically found in *all* Scalable Vector ISAs, without changing the behaviour of the underlying Base (Scalar) v3.0B operations in any way. (The sole apparent exception is Post-Increment Mode on LD/ST-update instructions)

9.1.2 Modes overview

Vectorisation of Load and Store requires creation, from scalar operations, a number of different modes:

- **fixed aka "unit" stride** - contiguous sequence with no gaps
- **element strided** - sequential but regularly offset, with gaps

- **vector indexed** - vector of base addresses and vector of offsets
- **Speculative fail-first** - where it makes sense to do so
- **Structure Packing** - covered in SV by {REMAP subsystem} and Pack/Unpack Mode.

*Despite being constructed from Scalar LD/ST none of these Modes exist or make sense in any Scalar ISA. They **only** exist in Vector ISAs and are a critical part of its value.*

Also included in SVP64 LD/ST is both signed and unsigned Saturation, as well as Element-width overrides and Twin-Predication.

Note also that Indexed {REMAP subsystem} mode may be applied to both Scalar LD/ST Immediate Defined Words *and* LD/ST Indexed Defined Words. LD/ST-Indexed should not be conflated with Indexed REMAP mode: clarification is provided below.

Determining the LD/ST Modes

A minor complication (caused by the retro-fitting of modern Vector features to a Scalar ISA) is that certain features do not exactly make sense or are considered a security risk. Fail-first on Vector Indexed would allow attackers to probe large numbers of pages from userspace, where strided fail-first (by creating contiguous sequential LDs) does not.

In addition, reduce mode makes no sense. Realistically we need an alternative table definition for {SVP64 Chapter} `RM.MODE`. The following modes make sense:

- saturation
- simple (no augmentation)
- fail-first (where Vector Indexed is banned)
- Signed Effective Address computation (Vector Indexed only)

More than that however it is necessary to fit the usual Vector ISA capabilities onto both Power ISA LD/ST with immediate and to LD/ST Indexed. They present subtly different Mode tables, which, due to lack of space, have the following quirks:

- LD/ST Immediate has no individual control over src/dest zeroing, whereas LD/ST Indexed does.
- LD/ST Immediate has saturation but LD/ST Indexed does not.

9.1.3 Format and fields

Fields used in tables below:

- **sz / dz** if predication is enabled will put zeros into the dest (or as src in the case of twin pred) when the predicate bit is zero. otherwise the element is ignored or skipped, depending on context.
- **zz**: both sz and dz are set equal to this flag.
- **inv CR bit** just as in branches (BO) these bits allow testing of a CR bit and whether it is set (inv=0) or unset (inv=1)
- **N** sets signed/unsigned saturation.
- **RC1** as if Rc=1, stores CRs *but not the result*
- **SEA** - Signed Effective Address, if enabled performs sign-extension on registers that have been reduced due to elwidth overrides
- **PI** - post-increment mode (applies to LD/ST with update only). the Effective Address utilised is always just RA, i.e. the computation of EA is stored in RA **after** it is actually used.
- **LF** - Load/Store Fail or Fault First: for any reason Load or Store Vectors may be truncated to (at least) one element, and VL altered to indicate such.
- **VLi** - Inclusive Data-Dependent Fail-First: the failing element is included in the Truncated Vector.
- **els** - Element-strided Mode: the element index (after REMAP) is multiplied by the immediate offset (or Scalar RB for Indexed). Restrictions apply.

When VLi=0 on Store Operations the Memory update does **not** take place on the element that failed. EA does **not** update into RA on Load/Store with Update instructions either.

LD/ST immediate

The table for {SVP64 Chapter} for `immed(RA)` which is `RM.MODE` (bits 19:23 of `RM`) is:

0	1	2	3 4	description
0	0	0	zz els	simple mode
0	0	1	PI LF	post-increment and Fault-First
1	0	N	zz els	sat mode: N=0/1 u/s
VLi	1	inv	CR-bit	ffirst CR sel

The `els` bit is only relevant when `RA.isvec` is clear: this indicates whether stride is unit or element:

```
if RA.isvec:
    svctx.ldstmode = indexed
elif els == 0:
    svctx.ldstmode = unitstride
elif immediate != 0:
    svctx.ldstmode = elementstride
```

An immediate of zero is a safety-valve to allow `LD-VSPLAT`: in effect the multiplication of the immediate-offset by zero results in reading from the exact same memory location, *even with a Vector register*. (Normally this type of behaviour is reserved for the mapreduce modes)

For `LD-VSPLAT`, on non-cache-inhibited Loads, the read can occur just the once and be copied, rather than hitting the Data Cache multiple times with the same memory read at the same location. The benefit of Cache-inhibited LD-splats is that it allows for memory-mapped peripherals to have multiple data values read in quick succession and stored in sequentially numbered registers (but, see Note below).

For non-cache-inhibited ST from a vector source onto a scalar destination: with the Vector loop effectively creating multiple memory writes to the same location, we can deduce that the last of these will be the "successful" one. Thus, implementations are free and clear to optimise out the overwriting STs, leaving just the last one as the "winner". Bear in mind that predicate masks will skip some elements (in source non-zeroing mode). Cache-inhibited ST operations on the other hand **MUST** write out a Vector source multiple successive times to the exact same Scalar destination. Just like Cache-inhibited LDs, multiple values may be written out in quick succession to a memory-mapped peripheral from sequentially-numbered registers.

Note that any memory location may be Cache-inhibited (Power ISA v3.1, Book III, 1.6.1, p1033)

Programmer's Note: an immediate also with a Scalar source as a "VSPLAT" mode is simply not possible: there are not enough Mode bits. One single Scalar Load operation may be used instead, followed by any arithmetic operation (including a simple mv) in "Splat" mode.

LD/ST Indexed

The modes for `RA+RB` indexed version are slightly different but are the same `RM.MODE` bits (19:23 of `RM`):

0	1	2	3 4	description
els	0	SEA	dz sz	simple mode
VLi	1	inv	CR-bit	ffirst CR sel

Vector Indexed Strided Mode is qualified as follows:

```
if els and !RA.isvec and !RB.isvec:
    svctx.ldstmode = elementstride
```

A summary of the effect of Vectorisation of src or dest:

```
imm(RA)  RT.v   RA.v   no stride allowed
```

```
imm(RA)  RT.s   RA.v   no stride allowed
imm(RA)  RT.v   RA.s   stride-select allowed
imm(RA)  RT.s   RA.s   not vectorised
RA,RB    RT.v  {RA|RB}.v Standard Indexed
RA,RB    RT.s  {RA|RB}.v Indexed but single LD (no VSPLAT)
RA,RB    RT.v  {RA&RB}.s VSPLAT possible. stride selectable
RA,RB    RT.s  {RA&RB}.s not vectorised (scalar identity)
```

Signed Effective Address computation is only relevant for Vector Indexed Mode, when elwidth overrides are applied. The source override applies to RB, and before adding to RA in order to calculate the Effective Address, if SEA is set RB is sign-extended from elwidth bits to the full 64 bits. For other Modes (ffirst, saturate), all EA computation with elwidth overrides is unsigned. RA is *not* altered (not truncated) by element-width overrides.

Note that cache-inhibited LD/ST when VSPLAT is activated will perform **multiple** LD/ST operations, sequentially. Even with scalar src a Cache-inhibited LD will read the same memory location *multiple times*, storing the result in successive Vector destination registers. This because the cache-inhibit instructions are typically used to read and write memory-mapped peripherals. If a genuine cache-inhibited LD-VSPLAT is required then a single *scalar* cache-inhibited LD should be performed, followed by a VSPLAT-augmented mv, copying the one *scalar* value into multiple register destinations.

Note also that cache-inhibited VSPLAT with Data-Dependent Fail-First is possible. This allows for example to issue a massive batch of memory-mapped peripheral reads, stopping at the first NULL-terminated character and truncating VL to that point. No branch is needed to issue that large burst of LDs, which may be valuable in Embedded scenarios.

9.1.4 Vectorisation of Scalar Power ISA v3.0B

Scalar Power ISA Load/Store operations may be seen from [[isa/fixedload]] and [[isa/fixedstore]] pseudocode to be of the form:

```
lbux RT, RA, RB
EA <- (RA) + (RB)
RT <- MEM(EA)
```

and for immediate variants:

```
lb RT,D(RA)
EA <- RA + EXTS(D)
RT <- MEM(EA)
```

Thus in the first example, the source registers may each be independently marked as scalar or vector, and likewise the destination; in the second example only the one source and one dest may be marked as scalar or vector.

Thus we can see that Vector Indexed may be covered, and, as demonstrated with the pseudocode below, the immediate can be used to give unit stride or element stride. With there being no way to tell which from the Power v3.0B Scalar opcode alone, the choice is provided instead by the SV Context.

```
# LD not VLD!  format - ldop RT, immed(RA)
# op_width: lb=1, lh=2, lw=4, ld=8
op_load(RT, RA, op_width, immed, svctx, RAupdate):
  ps = get_pred_val(FALSE, RA); # predication on src
  pd = get_pred_val(FALSE, RT); # ... AND on dest
  for (i=0, j=0, u=0; i < VL && j < VL;):
    # skip nonpredicates elements
    if (RA.isvec) while (!(ps & 1<<i)) i++;
    if (RAupdate.isvec) while (!(ps & 1<<u)) u++;
    if (RT.isvec) while (!(pd & 1<<j)) j++;
    if postinc:
        offs = 0; # added afterwards
```

```
            if RA.isvec: srcbase = ireg[RA+i]
            else         srcbase = ireg[RA]
        elif svctx.ldstmode == elementstride:
          # element stride mode
          srcbase = ireg[RA]
          offs = i * immed              # j*immed for a ST
        elif svctx.ldstmode == unitstride:
          # unit stride mode
          srcbase = ireg[RA]
          offs = immed + (i * op_width) # j*op_width for ST
        elif RA.isvec:
          # quirky Vector indexed mode but with an immediate
          srcbase = ireg[RA+i]
          offs = immed;
        else
          # standard scalar mode (but predicated)
          # no stride multiplier means VSPLAT mode
          srcbase = ireg[RA]
          offs = immed

        # compute EA
        EA = srcbase + offs
        # load from memory
        ireg[RT+j] <= MEM[EA];
        # check post-increment of EA
        if postinc: EA = srcbase + immed;
        # update RA?
        if RAupdate: ireg[RAupdate+u] = EA;
        if (!RT.isvec)
            break # destination scalar, end now
        if (RA.isvec) i++;
        if (RAupdate.isvec) u++;
        if (RT.isvec) j++;
```

Indexed LD is:

```
    # format: ldop RT, RA, RB
    function op_ldx(RT, RA, RB, RAupdate=False) # LD not VLD!
      ps = get_pred_val(FALSE, RA); # predication on src
      pd = get_pred_val(FALSE, RT); # ... AND on dest
      for (i=0, j=0, k=0, u=0; i < VL && j < VL && k < VL):
        # skip nonpredicated RA, RB and RT
        if (RA.isvec) while (!(ps & 1<<i)) i++;
        if (RAupdate.isvec) while (!(ps & 1<<u)) u++;
        if (RB.isvec) while (!(ps & 1<<k)) k++;
        if (RT.isvec) while (!(pd & 1<<j)) j++;
        if svctx.ldstmode == elementstride:
            EA = ireg[RA] + ireg[RB]*j   # register-strided
        else
            EA = ireg[RA+i] + ireg[RB+k] # indexed address
        if RAupdate: ireg[RAupdate+u] = EA
        ireg[RT+j] <= MEM[EA];
        if (!RT.isvec)
            break # destination scalar, end immediately
        if (RA.isvec) i++;
        if (RAupdate.isvec) u++;
```

```
if (RB.isvec) k++;
if (RT.isvec) j++;
```

Note that Element-Strided uses the Destination Step because with both sources being Scalar as a prerequisite condition of activation of Element-Stride Mode, the source step (being Scalar) would never advance.

Note in both cases that {SVP64 Chapter} allows RA-as-a-dest in "update" mode (`ldux`) to be effectively a *completely different* register from RA-as-a-source. This because there is room in svp64 to extend RA-as-src as well as RA-as-dest, both independently as scalar or vector *and* independently extending their range.

Programmer's note: being able to set RA-as-a-source as separate from RA-as-a-destination as Scalar is ***extremely valuable*** *once it is remembered that Simple-V element operations must be in Program Order, especially in loops, for saving on multiple address computations. Care does have to be taken however that RA-as-src is not overwritten by RA-as-dest unless intentionally desired, especially in element-strided Mode.*

9.1.5 LD/ST Indexed vs Indexed REMAP

Unfortunately the word "Indexed" is used twice in completely different contexts, potentially causing confusion.

- There has existed instructions in the Power ISA `ld RT,RA,RB` since its creation: these are called "LD/ST Indexed" instructions and their name and meaning is well-established.
- There now exists, in Simple-V, a {REMAP subsystem} mode called "Indexed" Mode that can be applied to *any* instruction **including those named LD/ST Indexed**.

Whilst it may be costly in terms of register reads to allow REMAP Indexed Mode to be applied to any Vectorised LD/ST Indexed operation such as `sv.ld *RT,RA,*RB`, or even misleadingly labelled as redundant, firstly the strict application of the RISC Paradigm that Simple-V follows makes it awkward to consider *preventing* the application of Indexed REMAP to such operations, and secondly they are not actually the same at all.

Indexed REMAP, as applied to RB in the instruction `sv.ld *RT,RA,*RB` effectively performs an *in-place* re-ordering of the offsets, RB. To achieve the same effect without Indexed REMAP would require taking a *copy* of the Vector of offsets starting at RB, manually explicitly reordering them, and finally using the copy of re-ordered offsets in a non-REMAP'ed `sv.ld`. Using non-strided LD as an example, pseudocode showing what actually occurs, where the pseudocode for `indexed_remap` may be found in {REMAP subsystem}:

```
# sv.ld *RT,RA,*RB with Index REMAP applied to RB
for i in 0..VL-1:
    if remap.indexed:
        rb_idx = indexed_remap(i) # remap
    else:
        rb_idx = i # use the index as-is
    EA = GPR(RA) + GPR(RB+rb_idx)
    GPR(RT+i) = MEM(EA, 8)
```

Thus it can be seen that the use of Indexed REMAP saves copying and manual reordering of the Vector of RB offsets.

9.1.6 LD/ST ffirst (Fault-First)

LD/ST ffirst treats the first LD/ST in a vector (element 0 if REMAP is not active and predication is not applied) as an ordinary one, with all behaviour with respect to Interrupts Exceptions Page Faults Memory Management being identical in every regard to Scalar v3.0 Power ISA LD/ST. However for elements 1 and above, if an exception would occur, then VL is **truncated** to the previous element: the exception is **not** then raised because the LD/ST that would otherwise have caused an exception is *required* to be cancelled. Additionally an implementor may choose to truncate VL for any arbitrary reason *except for the very first.*

ffirst LD/ST to multiple pages via a Vectorised Index base is considered a security risk due to the abuse of probing multiple pages in rapid succession and getting speculative feedback on which pages would fail. Therefore

Vector Indexed LD/ST is prohibited entirely, and the Mode bit instead used for element-strided LD/ST. See https://bugs.libre-soc.org/show_bug.cgi?id=561

```
for(i = 0; i < VL; i++)
    reg[rt + i] = mem[reg[ra] + i * reg[rb]];
```

High security implementations where any kind of speculative probing of memory pages is considered a risk should take advantage of the fact that implementations may truncate VL at any point, without requiring software to be rewritten and made non-portable. Such implementations may choose to *always* set VL=1 which will have the effect of terminating any speculative probing (and also adversely affect performance), but will at least not require applications to be rewritten.

Low-performance simpler hardware implementations may also choose (always) to also set VL=1 as the bare minimum compliant implementation of LD/ST Fail-First. It is however critically important to remember that the first element LD/ST **MUST** be treated as an ordinary LD/ST, i.e. **MUST** raise exceptions exactly like an ordinary LD/ST.

For ffirst LD/STs, VL may be truncated arbitrarily to a nonzero value for any implementation-specific reason. For example: it is perfectly reasonable for implementations to alter VL when ffirst LD or ST operations are initiated on a nonaligned boundary, such that within a loop the subsequent iteration of that loop begins the following ffirst LD/ST operations on an aligned boundary such as the beginning of a cache line, or beginning of a Virtual Memory page. Likewise, to reduce workloads or balance resources.

When Predication is used, the "first" element is considered to be the first non-predicated element rather than specifically `srcstep=0`.

Vertical-First Mode is slightly strange in that only one element at a time is ever executed anyway. Given that programmers may legitimately choose to alter srcstep and dststep in non-sequential order as part of explicit loops, it is neither possible nor safe to make speculative assumptions about future LD/STs. Therefore, Fail-First LD/ST in Vertical-First is `UNDEFINED`. This is very different from Arithmetic (Data-dependent) FFirst where Vertical-First Mode is fully deterministic, not speculative.

9.1.7 Data-Dependent Fail-First (not Fail/Fault-First)

Not to be confused with Fail/Fault First, Data-Fail-First performs an additional check on the data, and if the test fails then VL is truncated and further looping terminates. This is precisely the same as Arithmetic Data-Dependent Fail-First, the only difference being that the result comes from the LD/ST rather than from an Arithmetic operation.

Also a crucial difference between Arithmetic and LD/ST Data-Dependent Fail-First: except for Store-Conditional a 4-bit Condition Register Field test is created for testing purposes *but not stored* (thus there is no RC1 Mode as there is in Arithmetic). The reason why a CR Field is not stored is because Load/Store, particularly the Update instructions, is already expensive in register terms, and adding an extra Vector write would be too costly in hardware.

Programmer's note: Programmers may use Data-Dependent Load with a test to truncate VL, and may then follow up with a `sv.cmpi` or other operation. The important aspect is that the Vector Load truncated on finding a NULL pointer, for example.

Programmer's note: Load-with-Update may be used to update the register used in Effective Address computation of th next element. This may be used to perform single-linked-list walking, where Data-Dependent Fail-First terminates and truncates the Vector at the first NULL.

Load/Store Data-Dependent Fail-First, VLi=0

In the case of Store operations there is a quirk when VLi (VL inclusive is "Valid") is clear. Bear in mind the criteria is that the truncated Vector of results, when VLi is clear, must all pass the "test", but when VLi is set the *current failed test* is permitted to be included. Thus, the actual update (store) to Memory is **not permitted to take place** should the test fail.

Additionally in any Load/Store with Update instruction, when VLi=0 and a test fails then RA does **not** receive a copy of the Effective Address. Hardware implementations with Out-of-Order Micro-Architectures should use speculative Shadow-Hold and Cancellation (or other Transactional Rollback mechanism) when the test fails.

Load/Store Data-Dependent Fail-First, VLi=1

By contrast if VLi=1 and the test fails, the Store may proceed *and then* looping terminates. In this way, when Inclusive the Vector of Truncated results contains the first-failed data (including RA on Updates)

Below is an example of loading the starting addresses of Linked-List nodes. If VLi=1 it will load the NULL pointer into the Vector of results. If however VLi=0 it will *exclude* the NULL pointer by truncating VL to one Element earlier.

Programmer's Note: by also setting the RC1 qualifier as well as setting VLi=1 it is possible to establish a Predicate Mask such that the first zero in the predicate will be the NULL pointer

```
RT=1 # vec - deliberately overlaps by one with RA
RA=0 # vec - first one is valid, contains ptr
imm = 8 # offset_of(ptr->next)
for i in range(VL):
    # this part is the Scalar Defined Word (standard scalar ld operation)
    EA = GPR(RA+i) + imm          # ptr + offset(next)
    data = MEM(EA, 8)             # 64-bit address of ptr->next
    # was a normal vector-ld up to this point. now the Data-Fail-First
    cr_test = conditions(data)
    if Rc=1 or RC1: CR.field(i) = cr_test # only store if Rc=1/RC1
    action_load = True
    if cr_test.EQ == testbit:             # check if zero
        if VLI then
           VL = i+1            # update VL, inclusive
        else
           VL = i              # update VL, exclusive current
           action_load = False # current load excluded
        stop = True            # stop looping
    if action_load:
       GPR(RT+i) = data        # happens to be read on next loop!
    if stop: break
```

Data-Dependent Fail-First on Store-Conditional (Rc=1)

There are very few instructions that allow Rc=1 for Load/Store: one of those is the `stdcx.` and other Atomic Store-Conditional instructions. With Simple-V being a loop around Scalar instructions strictly obeying Scalar Program Order a Horizontal-First Fail-First loop on an Atomic Store-Conditional will always fail the second and all other Store-Conditional instructions because Load-Reservation and Store-Conditional are required to be executed in pairs.

By contrast, in Vertical-First Mode it is in fact possible to issue the pairs, and consequently allowing Vectorised Data-Dependent Fail-First is useful.

Programmer's note: Care should be taken when VL is truncated in Vertical-First Mode.

Future potential

Although Rc=1 on LD/ST is a rare occurrence at present, future versions of Power ISA *might* conceivably have Rc=1 LD/ST Scalar instructions, and with the SVP64 Vectorisation Prefixing being itself a RISC-paradigm that is itself fully-independent of the Scalar Suffix Defined Words, prohibiting the possibility of Rc=1 Data-Dependent Mode on future potential LD/ST operations is not strategically sound.

9.1.8 LOAD/STORE Elwidths

Loads and Stores are almost unique in that the Power Scalar ISA provides a width for the operation (lb, lh, lw, ld). Only `extsb` and others like it provide an explicit operation width. There are therefore *three* widths involved:

- operation width (lb=8, lh=16, lw=32, ld=64)
- src element width override (8/16/32/default)
- destination element width override (8/16/32/default)

Some care is therefore needed to express and make clear the transformations, which are expressly in this order:

- Calculate the Effective Address from RA at full width but (on Indexed Load) allow srcwidth overrides on RB
- Load at the operation width (lb/lh/lw/ld) as usual
- byte-reversal as usual
- Non-saturated mode:
- zero-extension or truncation from operation width to dest elwidth
- place result in destination at dest elwidth
- Saturated mode:
- Sign-extension or truncation from operation width to dest width
- signed/unsigned saturation down to dest elwidth

In order to respect Power v3.0B Scalar behaviour the memory side is treated effectively as completely separate and distinct from SV augmentation. This is primarily down to quirks surrounding LE/BE and byte-reversal.

It is rather unfortunately possible to request an elwidth override on the memory side which does not mesh with the overridden operation width: these result in `UNDEFINED` behaviour. The reason is that the effect of attempting a 64-bit `sv.ld` operation with a source elwidth override of 8/16/32 would result in overlapping memory requests, particularly on unit and element strided operations. Thus it is `UNDEFINED` when the elwidth is smaller than the memory operation width. Examples include `sv.lw/sw=16/els` which requests (overlapping) 4-byte memory reads offset from each other at 2-byte intervals. Store likewise is also `UNDEFINED` where the dest elwidth override is less than the operation width.

Note the following regarding the pseudocode to follow:

- `scalar identity behaviour` SV Context parameter conditions turn this into a straight absolute fully-compliant Scalar v3.0B LD operation
- `brev` selects whether the operation is the byte-reversed variant (`ldbrx` rather than `ld`)
- `op_width` specifies the operation width (`lb`, `lh`, `lw`, `ld`) as a "normal" part of Scalar v3.0B LD
- `imm_offs` specifies the immediate offset `ld r3, imm_offs(r5)`, again as a "normal" part of Scalar v3.0B LD
- `svctx` specifies the SV Context and includes VL as well as source and destination elwidth overrides.

Below is the pseudocode for Unit-Strided LD (which includes Vector capability). Observe in particular that RA, as the base address in both Immediate and Indexed LD/ST, does not have element-width overriding applied to it.

Note that predication, predication-zeroing, and other modes except saturation have all been removed, for clarity and simplicity:

```
# LD not VLD!
# this covers unit stride mode and a type of vector offset
function op_ld(RT, RA, op_width, imm_offs, svctx)
  for (int i = 0, int j = 0; i < svctx.VL && j < svctx.VL):
    if not svctx.unit/el-strided:
        # strange vector mode, compute 64 bit address which is
        # not polymorphic! elwidth hardcoded to 64 here
        srcbase = get_polymorphed_reg(RA, 64, i)
    else:
        # unit / element stride mode, compute 64 bit address
        srcbase = get_polymorphed_reg(RA, 64, 0)
```

```
            # adjust for unit/el-stride
            srcbase += ....

        # read the underlying memory
        memread <= MEM(srcbase + imm_offs, op_width)

        # check saturation.
        if svpctx.saturation_mode:
            # ... saturation adjustment...
            memread = clamp(memread, op_width, svctx.dest_elwidth)
        else:
            # truncate/extend to over-ridden dest width.
            memread = adjust_wid(memread, op_width, svctx.dest_elwidth)

        # takes care of inserting memory-read (now correctly byteswapped)
        # into regfile underlying LE-defined order, into the right place
        # within the NEON-like register, respecting destination element
        # bitwidth, and the element index (j)
        set_polymorphed_reg(RT, svctx.dest_elwidth, j, memread)

        # increments both src and dest element indices (no predication here)
        i++;
        j++;
```

Note above that the source elwidth is *not used at all* in LD-immediate.

For LD/Indexed, the key is that in the calculation of the Effective Address, RA has no elwidth override but RB does. Pseudocode below is simplified for clarity: predication and all modes except saturation are removed:

```
    # LD not VLD! ld*rx if brev else ld*
    function op_ld(RT, RA, RB, op_width, svctx, brev)
      for (int i = 0, int j = 0; i < svctx.VL && j < svctx.VL):
        if not svctx.el-strided:
            # RA not polymorphic! elwidth hardcoded to 64 here
            srcbase = get_polymorphed_reg(RA, 64, i)
        else:
            # element stride mode, again RA not polymorphic
            srcbase = get_polymorphed_reg(RA, 64, 0)
        # RB *is* polymorphic
        offs = get_polymorphed_reg(RB, svctx.src_elwidth, i)
        # sign-extend
        if svctx.SEA: offs = sext(offs, svctx.src_elwidth, 64)

        # takes care of (merges) processor LE/BE and ld/ldbrx
        bytereverse = brev XNOR MSR.LE

        # read the underlying memory
        memread <= MEM(srcbase + offs, op_width)

        # optionally performs byteswap at op width
        if (bytereverse):
            memread = byteswap(memread, op_width)

        if svpctx.saturation_mode:
            # ... saturation adjustment...
            memread = clamp(memread, op_width, svctx.dest_elwidth)
        else:
```

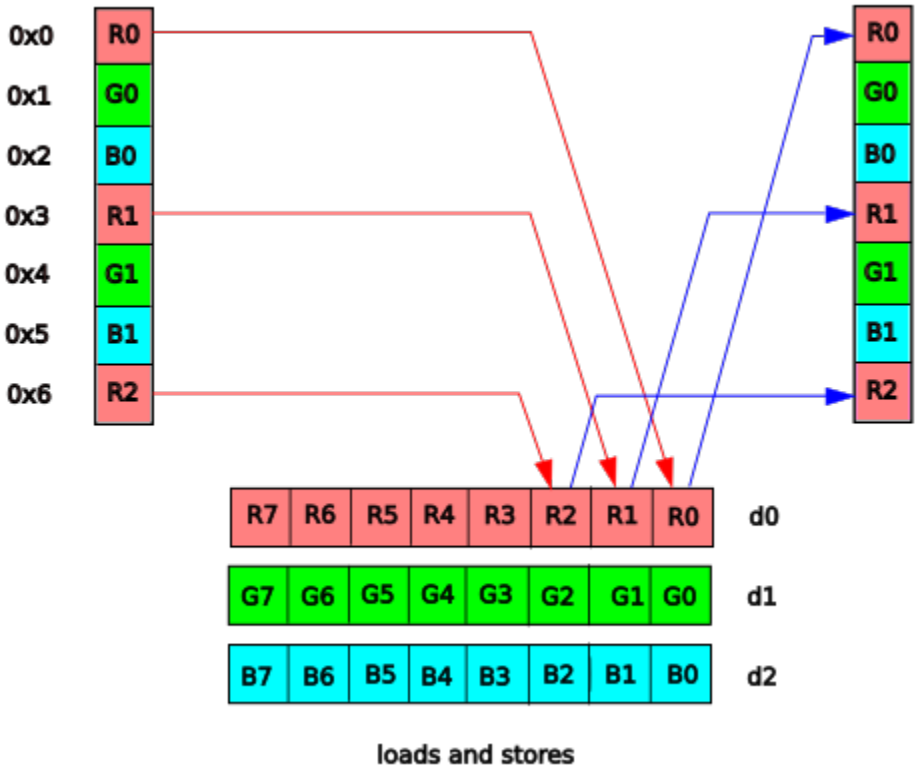

Figure 9.1: Load/Store remap

```
    # truncate/extend to over-ridden dest width.
    memread = adjust_wid(memread, op_width, svctx.dest_elwidth)

# takes care of inserting memory-read (now correctly byteswapped)
# into regfile underlying LE-defined order, into the right place
# within the NEON-like register, respecting destination element
# bitwidth, and the element index (j)
set_polymorphed_reg(RT, svctx.dest_elwidth, j, memread)

# increments both src and dest element indices (no predication here)
i++;
j++;
```

9.1.9 Remapped LD/ST

In the {REMAP subsystem} page the concept of "Remapping" is described. Whilst it is expensive to set up (2 64-bit opcodes minimum) it provides a way to arbitrarily perform 1D, 2D and 3D "remapping" of up to 64 elements worth of LDs or STs. The usual interest in such re-mapping is for example in separating out 24-bit RGB channel data into separate contiguous registers. NEON covers this as shown in the diagram below:

REMAP easily covers this capability, and with dest elwidth overrides and saturation may do so with built-in conversion that would normally require additional width-extension, sign-extension and min/max Vectorised instructions as post-processing stages.

Thus we do not need to provide specialist LD/ST "Structure Packed" opcodes because the generic abstracted concept of "Remapping", when applied to LD/ST, will give that same capability, with far more flexibility.

It is worth noting that Pack/Unpack Modes of SVSTATE, which may be established through `svstep`, are also an easy way to perform regular Structure Packing, at the vec2/vec3/vec4 granularity level. Beyond that, REMAP will need to be used.

Parallel Reduction REMAP

No REMAP Schedule is prohibited in SVP64 because the RISC-paradigm Prefix is completely separate from the RISC-paradigm Scalar Defined Words. Although obscure there does exist the outside possibility that a potential use for Parallel Reduction Schedules on LD/ST would find a use in Computer Science. Readers are invited to contact the authors of this document if one is ever found.

[[!tag standards]]

Chapter 10

Condition Register Fields Mode

10.1 Condition Register SVP64 Operations

DRAFT STATUS

Links:

- https://bugs.libre-soc.org/show_bug.cgi?id=687
- https://bugs.libre-soc.org/show_bug.cgi?id=936 write on failfirst
- {SVP64 Chapter}
- {Branch Mode}
- {CR Weird ops}
- [[openpower/isa/sprset]]
- [[openpower/isa/condition]]
- [[openpower/isa/comparefixed]]

Condition Register Fields are only 4 bits wide: this presents some interesting conceptual challenges for SVP64, which was designed primarily for vectors of arithmetic and logical operations. However if predicates may be bits of CR Fields it makes sense to extend Simple-V to cover CR Operations, especially given that Vectorised Rc=1 may be processed by Vectorised CR Operations that usefully in turn may become Predicate Masks to yet more Vector operations, like so:

```
sv.cmpi/ew=8 *B,*ra,0    # compare bytes against zero
sv.cmpi/ew=8 *B2,*ra,13. # and against newline
sv.cror PM.EQ,B.EQ,B2.EQ # OR compares to create mask
sv.stb/sm=EQ    ...      # store only nonzero/newline
```

Element width however is clearly meaningless for a 4-bit collation of Conditions, EQ LT GE SO. Likewise, arithmetic saturation (an important part of Arithmetic SVP64) has no meaning. An alternative Mode Format is required, and given that elwidths are meaningless for CR Fields the bits in SVP64 `RM` may be used for other purposes.

This alternative mapping **only** applies to instructions that **only** reference a CR Field or CR bit as the sole exclusive result. This section **does not** apply to instructions which primarily produce arithmetic results that also, as an aside, produce a corresponding CR Field (such as when Rc=1). Instructions that involve Rc=1 are definitively arithmetic in nature, where the corresponding Condition Register Field can be considered to be a "co-result". Such CR Field "co-result" arithmeric operations are firmly out of scope for this section, being covered fully by {Arithmetic Mode}.

- Examples of v3.0B instructions to which this section does apply is
- `mfcr` and `cmpi` (3 bit operands) and
- `crnor` and `crand` (5 bit operands).

- Examples to which this section does **not** apply include `fadds.` and `subf.` which both produce arithmetic results (and a CR Field co-result).
- `mtcr` is considered [[openpower/sv/normal]] because it refers to the entire 32-bit Condition Register rather than to CR Fields.

The CR Mode Format still applies to `sv.cmpi` because despite taking a GPR as input, the output from the Base Scalar v3.0B `cmpi` instruction is purely to a Condition Register Field.

Other modes are still applicable and include:

- **Data-dependent fail-first**. useful to truncate VL based on analysis of a Condition Register result bit.
- **Reduction**. Reduction is useful for analysing a Vector of Condition Register Fields and reducing it to one single Condition Register Field.

Predicate-result does not make any sense because when Rc=1 a co-result is created (a CR Field). Testing the co-result allows the decision to be made to store or not store the main result, and for CR Ops the CR Field result *is* the main result.

10.1.1 Format

SVP64 RM `MODE` (includes `ELWIDTH_SRC` bits) for CR-based operations:

6	7	19-20	21	22 23	description
/	/	0 RG	0	dz sz	simple mode
/	/	0 RG	1	dz sz	scalar reduce mode (mapreduce)
zz	SNZ	1 VLI	inv	CR-bit	Ffirst 3-bit mode
/	SNZ	1 VLI	inv	dz sz	Ffirst 5-bit mode (implies CR-bit from result)

Fields:

- **sz / dz** if predication is enabled will put zeros into the dest (or as src in the case of twin pred) when the predicate bit is zero. otherwise the element is ignored or skipped, depending on context.
- **zz** set both sz and dz equal to this flag
- **SNZ** In fail-first mode, on the bit being tested, when sz=1 and SNZ=1 a value "1" is put in place of "0".
- **inv CR-bit** just as in branches (BO) these bits allow testing of a CR bit and whether it is set (inv=0) or unset (inv=1)
- **RG** inverts the Vector Loop order (VL-1 downto 0) rather than the normal 0..VL-1
- **SVM** sets "subvector" reduce mode
- **VLi** VL inclusive: in fail-first mode, the truncation of VL *includes* the current element at the failure point rather than excludes it from the count.

10.1.2 Data-dependent fail-first on CR operations

The principle of data-dependent fail-first is that if, during the course of sequentially evaluating an element's Condition Test, one such test is encountered which fails, then VL (Vector Length) is truncated (set) at that point. In the case of Arithmetic SVP64 Operations the Condition Register Field generated from Rc=1 is used as the basis for the truncation decision. However with CR-based operations that CR Field result to be tested is provided *by the operation itself*.

Data-dependent SVP64 Vectorised Operations involving the creation or modification of a CR can require an extra two bits, which are not available in the compact space of the SVP64 RM `MODE` Field. With the concept of element width overrides being meaningless for CR Fields it is possible to use the `ELWIDTH` field for alternative purposes.

Condition Register based operations such as `sv.mfcr` and `sv.crand` can thus be made more flexible. However the rules that apply in this section also apply to future CR-based instructions.

There are two primary different types of CR operations:

- Those which have a 3-bit operand field (referring to a CR Field)
- Those which have a 5-bit operand (referring to a bit within the whole 32-bit CR)

Examining these two types it is observed that the difference may be considered to be that the 5-bit variant *already* provides the prerequisite information about which CR Field bit (EQ, GE, LT, SO) is to be operated on by the instruction. Thus, logically, we may set the following rule:

- When a 5-bit CR Result field is used in an instruction, the 5-bit variant of Data-Dependent Fail-First must be used. i.e. the bit of the CR field to be tested is the one that has just been modified (created) by the operation.
- When a 3-bit CR Result field is used the 3-bit variant must be used, providing as it does the missing `CRbit` field in order to select which CR Field bit of the result shall be tested (EQ, LE, GE, SO)

The reason why the 3-bit CR variant needs the additional CR-bit field should be obvious from the fact that the 3-bit CR Field from the base Power ISA v3.0B operation clearly does not contain and is missing the two CR Field Selector bits. Thus, these two bits (to select EQ, LE, GE or SO) must be provided in another way.

Examples of the former type:

- crand, cror, crnor. These all are 5-bit (BA, BB, BT). The bit to be tested against `inv` is the one selected by `BT`
- mcrf. This has only 3-bit (BF, BFA). In order to select the bit to be tested, the alternative encoding must be used. With `CRbit` coming from the SVP64 RM bits 22-23 the bit of BF to be tested is identified.

Just as with SVP64 {Branch Mode} there is the option to truncate VL to include the element being tested (`VLi=1`) and to exclude it (`VLi=0`).

Also exactly as with {Arithmetic Mode} fail-first, VL cannot, unlike {Load/Store Mode}, be set to an arbitrary value. Deterministic behaviour is *required*.

10.1.3 Reduction and Iteration

Bearing in mind as described in the {SVP64 Appendix} SVP64 Horizontal Reduction is a deterministic schedule on top of base Scalar v3.0 operations, the same rules apply to CR Operations, i.e. that programmers must follow certain conventions in order for an *end result* of a reduction to be achieved. Unlike other Vector ISAs *there are no explicit reduction opcodes* in SVP64: Schedules however achieve the same effect.

Due to these conventions only reduction on operations such as `crand` and `cror` are meaningful because these have Condition Register Fields as both input and output. Meaningless operations are not prohibited because the cost in hardware of doing so is prohibitive, but neither are they `UNDEFINED`. Implementations are still required to execute them but are at liberty to optimise out any operations that would ultimately be overwritten, as long as Strict Program Order is still obvservable by the programmer.

Also bear in mind that 'Reverse Gear' may be enabled, which can be used in combination with overlapping CR operations to iteratively accumulate results. Issuing a `sv.crand` operation for example with `BA` differing from `BB` by one Condition Register Field would result in a cascade effect, where the first-encountered CR Field would set the result to zero, and also all subsequent CR Field elements thereafter:

```
# sv.crand/mr/rg CR4.ge.v, CR5.ge.v, CR4.ge.v
for i in VL-1 downto 0 # reverse gear
     CR.field[4+i].ge &= CR.field[5+i].ge
```

`sv.crxor` with reduction would be particularly useful for parity calculation for example, although there are many ways in which the same calculation could be carried out (`parityw`) after transferring a vector of CR Fields to a GPR using crweird operations.

Implementations are free and clear to optimise these reductions in any way they see fit, as long as the end-result is compatible with Strict Program Order being observed, and Interrupt latency is not adversely impacted. Good

examples include `sv.cror/mr` which is a cumulative ORing of a Vector of CR Field bits, and consequently an easy target for parallelising.

10.1.4 Unusual and quirky CR operations

cmp and other compare ops

`cmp` and `cmpi` etc take GPRs as sources and create a CR Field as a result.

```
cmpli BF,L,RA,UI
cmpeqb BF,RA,RB
```

With `ELWIDTH` applying to the source GPR operands this is perfectly fine.

crweird operations

There are 4 weird CR-GPR operations and one reasonable one in the {CR Weird ops} set:

- crrweird
- mtcrweird
- crweirder
- crweird
- mcrfm - reasonably normal and referring to CR Fields for src and dest.

The "weird" operations have a non-standard behaviour, being able to treat *individual bits* of a GPR effectively as elements. They are expected to be Micro-coded by most Hardware implementations.

10.1.5 Effectively-separate Vector and Scalar Condition Register file

As mentioned in the introduction on {SVP64 Chapter} some prohibitions are made on instructions involving Condition Registers that allow implementors to actually consider the Scalar CR (fields CR0-CR7) as a completely separate register file from the Vector CRs (fields CR8-CR127).

The complications arise for existing Hardware implementations due to Power ISA not having had "Conditional Execution" added. Adding entirely new pipelines and a new Vector CR Register file is a much easier proposition to consider.

The prohibitions utilise the CR Field numbers implicitly to split out Vectorised CR operations to be considered completely separare and distinct from Scalar CR operations *even though they both use the same binary encoding.* This does in turn mean that at the Decode Phase it becomes necessary to examine not only the operation (`sv.crand`, `sv.cmp`) but also the CR Field numbers as well as whether, in the EXTRA2/3 Mode bits, the operands are Vectorised.

A future version of Power ISA, where SVP64Single is proposed, would in fact introduce "Conditional Execution", including for VSX. At which point this prohibition becomes moot as Predication would be required to be added into the existing Scalar (and PackedSIMD VSX) side of existing Power ISA implementations.

[[!tag standards]]

Chapter 11

Branch Mode

11.1 SVP64 Branch Conditional behaviour

DRAFT STATUS

Please note: although similar, SVP64 Branch instructions should be considered completely separate and distinct from standard scalar OpenPOWER-approved v3.0B branches. **v3.0B branches are in no way impacted, altered, changed or modified in any way, shape or form by the SVP64 Vectorised Variants**.

It is also extremely important to note that Branches are the sole pseudo-exception in SVP64 to `Scalar Identity Behaviour`. SVP64 Branches contain additional modes that are useful for scalar operations (i.e. even when VL=1 or when using single-bit predication).

Links

- https://bugs.libre-soc.org/show_bug.cgi?id=664
- http://lists.libre-soc.org/pipermail/libre-soc-dev/2021-August/003416.html
- https://lists.libre-soc.org/pipermail/libre-soc-dev/2022-April/004678.html
- Branch Divergence https://jbush001.github.io/2014/12/07/branch-divergence-in-parallel-kernels.html
- {Branch pseudocode}
- {CR Weird ops}
- TODO

11.1.1 Rationale

Scalar 3.0B Branch Conditional operations, `bc`, `bctar` etc. test a Condition Register. However for parallel processing it is simply impossible to perform multiple independent branches: the Program Counter simply cannot branch to multiple destinations based on multiple conditions. The best that can be done is to test multiple Conditions and make a decision of a *single* branch, based on analysis of a *Vector* of CR Fields which have just been calculated from a *Vector* of results.

In 3D Shader binaries, which are inherently parallelised and predicated, testing all or some results and branching based on multiple tests is extremely common, and a fundamental part of Shader Compilers. Example: without such multi-condition test-and-branch, if a predicate mask is all zeros a large batch of instructions may be masked out to `nop`, and it would waste CPU cycles to run them. 3D GPU ISAs can test for this scenario and, with the appropriate predicate-analysis instruction, jump over fully-masked-out operations, by spotting that *all* Conditions are false.

Unless Branches are aware and capable of such analysis, additional instructions would be required which perform Horizontal Cumulative analysis of Vectorised Condition Register Fields, in order to reduce the Vector of CR

Fields down to one single yes or no decision that a Scalar-only v3.0B Branch-Conditional could cope with. Such instructions would be unavoidable, required, and costly by comparison to a single Vector-aware Branch. Therefore, in order to be commercially competitive, `sv.bc` and other Vector-aware Branch Conditional instructions are a high priority for 3D GPU (and OpenCL-style) workloads.

Given that Power ISA v3.0B is already quite powerful, particularly the Condition Registers and their interaction with Branches, there are opportunities to create extremely flexible and compact Vectorised Branch behaviour. In addition, the side-effects (updating of CTR, truncation of VL, described below) make it a useful instruction even if the branch points to the next instruction (no actual branch).

11.1.2 Overview

When considering an "array" of branch-tests, there are four primarily-useful modes: AND, OR, NAND and NOR of all Conditions. NAND and NOR may be synthesised from AND and OR by inverting `BO[1]` which just leaves two modes:

- Branch takes place on the **first** CR Field test to succeed (a Great Big OR of all condition tests). Exit occurs on the first **successful** test.
- Branch takes place only if **all** CR field tests succeed: a Great Big AND of all condition tests. Exit occurs on the first **failed** test.

Early-exit is enacted such that the Vectorised Branch does not perform needless extra tests, which will help reduce reads on the Condition Register file.

*Note: Early-exit is **MANDATORY** (required) behaviour. Branches **MUST** exit at the first sequentially-encountered failure point, for exactly the same reasons for which it is mandatory in programming languages doing early-exit: to avoid damaging side-effects and to provide deterministic behaviour. Speculative testing of Condition Register Fields is permitted, as is speculative calculation of CTR, as long as, as usual in any Out-of-Order microarchitecture, that speculative testing is cancelled should an early-exit occur. i.e. the speculation must be "precise": Program Order must be preserved*

Also note that when early-exit occurs in Horizontal-first Mode, srcstep, dststep etc. are all reset, ready to begin looping from the beginning for the next instruction. However for Vertical-first Mode srcstep etc. are incremented "as usual" i.e. an early-exit has no special impact, regardless of whether the branch occurred or not. This can leave srcstep etc. in what may be considered an unusual state on exit from a loop and it is up to the programmer to reset srcstep, dststep etc. to known-good values *(easily achieved with `setvl`)*.

Additional useful behaviour involves two primary Modes (both of which may be enabled and combined):

- **VLSET Mode**: identical to Data-Dependent Fail-First Mode for Arithmetic SVP64 operations, with more flexibility and a close interaction and integration into the underlying base Scalar v3.0B Branch instruction. Truncation of VL takes place around the early-exit point.
- **CTR-test Mode**: gives much more flexibility over when and why CTR is decremented, including options to decrement if a Condition test succeeds *or if it fails*.

With these side-effects, basic Boolean Logic Analysis advises that it is important to provide a means to enact them each based on whether testing succeeds *or fails*. This results in a not-insignificant number of additional Mode Augmentation bits, accompanying VLSET and CTR-test Modes respectively.

Predicate skipping or zeroing may, as usual with SVP64, be controlled by `sz`. Where the predicate is masked out and zeroing is enabled, then in such circumstances the same Boolean Logic Analysis dictates that rather than testing only against zero, the option to test against one is also prudent. This introduces a new immediate field, `SNZ`, which works in conjunction with `sz`.

Vectorised Branches can be used in either SVP64 Horizontal-First or Vertical-First Mode. Essentially, at an element level, the behaviour is identical in both Modes, although the `ALL` bit is meaningless in Vertical-First Mode.

It is also important to bear in mind that, fundamentally, Vectorised Branch-Conditional is still extremely close to the Scalar v3.0B Branch-Conditional instructions, and that the same v3.0B Scalar Branch-Conditional

instructions are still *completely separate and independent*, being unaltered and unaffected by their SVP64 variants in every conceivable way.

*Programming note: One important point is that SVP64 instructions are 64 bit. (8 bytes not 4). This needs to be taken into consideration when computing branch offsets: the offset is relative to the start of the instruction, which **includes** the SVP64 Prefix*

11.1.3 Format and fields

With element-width overrides being meaningless for Condition Register Fields, bits 4 thru 7 of SVP64 RM may be used for additional Mode bits.

SVP64 RM `MODE` (includes repurposing `ELWIDTH` bits 4:5, and `ELWIDTH_SRC` bits 6-7 for *alternate* uses) for Branch Conditional:

4	5	6	7	17	18	19	20	21	22 23	description
ALL	SNZ	/	/	SL	SLu	0	0	/	LRu sz	simple mode
ALL	SNZ	/	VSb	SL	SLu	0	1	VLI	LRu sz	VLSET mode
ALL	SNZ	CTi	/	SL	SLu	1	0	/	LRu sz	CTR-test mode
ALL	SNZ	CTi	VSb	SL	SLu	1	1	VLI	LRu sz	CTR-test+VLSET mode

Brief description of fields:

- **sz=1** if predication is enabled and `sz=1` and a predicate element bit is zero, `SNZ` will be substituted in place of the CR bit selected by `BI`, as the Condition tested. Contrast this with normal SVP64 `sz=1` behaviour, where *only* a zero is put in place of masked-out predicate bits.
- **sz=0** When `sz=0` skipping occurs as usual on masked-out elements, but unlike all other SVP64 behaviour which entirely skips an element with no related side-effects at all, there are certain special circumstances where CTR may be decremented. See CTR-test Mode, below.
- **ALL** when set, all branch conditional tests must pass in order for the branch to succeed. When clear, it is the first sequentially encountered successful test that causes the branch to succeed. This is identical behaviour to how programming languages perform early-exit on Boolean Logic chains.
- **VLI** VLSET is identical to Data-dependent Fail-First mode. In VLSET mode, VL *may* (depending on `VSb`) be truncated. If VLI (Vector Length Inclusive) is clear, VL is truncated to *exclude* the current element, otherwise it is included. SVSTATE.MVL is not altered: only VL.
- **SL** identical to `LR` except applicable to SVSTATE. If `SL` is set, SVSTATE is transferred to SVLR (conditionally on whether `SLu` is set).
- **SLu**: SVSTATE Link Update, like `LRu` except applies to SVSTATE.
- **LRu**: Link Register Update, used in conjunction with LK=1 to make LR update conditional
- **VSb** In VLSET Mode, after testing, if VSb is set, VL is truncated if the test succeeds. If VSb is clear, VL is truncated if a test *fails*. Masked-out (skipped) bits are not considered part of testing when `sz=0`
- **CTi** CTR inversion. CTR-test Mode normally decrements per element tested. CTR inversion decrements if a test *fails*. Only relevant in CTR-test Mode.

LRu and CTR-test modes are where SVP64 Branches subtly differ from Scalar v3.0B Branches. `sv.bcl` for example will always update LR, whereas `sv.bcl/lru` will only update LR if the branch succeeds.

Of special interest is that when using ALL Mode (Great Big AND of all Condition Tests), if `VL=0`, which is rare but can occur in Data-Dependent Modes, the Branch will always take place because there will be no failing Condition Tests to prevent it. Likewise when not using ALL Mode (Great Big OR of all Condition Tests) and `VL=0` the Branch is guaranteed not to occur because there will be no *successful* Condition Tests to make it happen.

11.1.4 Vectorised CR Field numbering, and Scalar behaviour

It is important to keep in mind that just like all SVP64 instructions, the `BI` field of the base v3.0B Branch Conditional instruction may be extended by SVP64 EXTRA augmentation, as well as be marked as either Scalar or Vector. It is also crucially important to keep in mind that for CRs, SVP64 sequentially increments the CR *Field* numbers. CR *Fields* are treated as elements, not bit-numbers of the CR *register*.

The `BI` operand of Branch Conditional operations is five bits, in scalar v3.0B this would select one bit of the 32 bit CR, comprising eight CR Fields of 4 bits each. In SVP64 there are 16 32 bit CRs, containing 128 4-bit CR Fields. Therefore, the 2 LSBs of `BI` select the bit from the CR Field (EQ LT GT SO), and the top 3 bits are extended to either scalar or vector and to select CR Fields 0..127 as specified in SVP64 {SVP64 Appendix}.

When the CR Fields selected by SVP64-Augmented `BI` is marked as scalar, then as the usual SVP64 rules apply: the Vector loop ends at the first element tested (the first CR *Field*), after taking predication into consideration. Thus, also as usual, when a predicate mask is given, and `BI` marked as scalar, and `sz` is zero, srcstep skips forward to the first non-zero predicated element, and only that one element is tested.

In other words, the fact that this is a Branch Operation (instead of an arithmetic one) does not result, ultimately, in significant changes as to how SVP64 is fundamentally applied, except with respect to:

- the unique properties associated with conditionally changing the Program Counter (aka "a Branch"), resulting in early-out opportunities
- CTR-testing

Both are outlined below, in later sections.

11.1.5 Horizontal-First and Vertical-First Modes

In SVP64 Horizontal-First Mode, the first failure in ALL mode (Great Big AND) results in early exit: no more updates to CTR occur (if requested); no branch occurs, and LR is not updated (if requested). Likewise for non-ALL mode (Great Big Or) on first success early exit also occurs, however this time with the Branch proceeding. In both cases the testing of the Vector of CRs should be done in linear sequential order (or in REMAP re-sequenced order): such that tests that are sequentially beyond the exit point are *not* carried out. (*Note: it is standard practice in Programming languages to exit early from conditional tests, however a little unusual to consider in an ISA that is designed for Parallel Vector Processing. The reason is to have strictly-defined guaranteed behaviour*)

In Vertical-First Mode, setting the `ALL` bit results in `UNDEFINED` behaviour. Given that only one element is being tested at a time in Vertical-First Mode, a test designed to be done on multiple bits is meaningless.

11.1.6 Description and Modes

Predication in both INT and CR modes may be applied to `sv.bc` and other SVP64 Branch Conditional operations, exactly as they may be applied to other SVP64 operations. When `sz` is zero, any masked-out Branch-element operations are not included in condition testing, exactly like all other SVP64 operations, *including* side-effects such as potentially updating LR or CTR, which will also be skipped. There is *one* exception here, which is when `BO[2]=0, sz=0, CTR-test=0, CTi=1` and the relevant element predicate mask bit is also zero: under these special circumstances CTR will also decrement.

When `sz` is non-zero, this normally requests insertion of a zero in place of the input data, when the relevant predicate mask bit is zero. This would mean that a zero is inserted in place of `CR[BI+32]` for testing against `BO`, which may not be desirable in all circumstances. Therefore, an extra field is provided `SNZ`, which, if set, will insert a **one** in place of a masked-out element, instead of a zero.

(*Note: Both options are provided because it is useful to deliberately cause the Branch-Conditional Vector testing to fail at a specific point, controlled by the Predicate mask. This is particularly useful in `VLSET` mode, which will truncate SVSTATE.VL at the point of the first failed test.*)

Normally, CTR mode will decrement once per Condition Test, resulting under normal circumstances that CTR reduces by up to VL in Horizontal-First Mode. Just as when v3.0B Branch-Conditional saves at least one instruction on tight inner loops through auto-decrementation of CTR, likewise it is also possible to save instruction count for SVP64 loops in both Vertical-First and Horizontal-First Mode, particularly in circumstances where there is conditional interaction between the element computation and testing, and the continuation (or otherwise) of a given loop. The potential combinations of interactions is why CTR testing options have been added.

Also, the unconditional bit `BO[0]` is still relevant when Predication is applied to the Branch because in `ALL` mode all nonmasked bits have to be tested, and when `sz=0` skipping occurs. Even when VLSET mode is not used, CTR may still be decremented by the total number of nonmasked elements, acting in effect as either a popcount or cntlz depending on which mode bits are set. In short, Vectorised Branch becomes an extremely powerful tool.

Micro-Architectural Implementation Note: *when implemented on top of a Multi-Issue Out-of-Order Engine it is possible to pass a copy of the predicate and the prerequisite CR Fields to all Branch Units, as well as the current value of CTR at the time of multi-issue, and for each Branch Unit to compute how many times CTR would be subtracted, in a fully-deterministic and parallel fashion. A SIMD-based Branch Unit, receiving and processing multiple CR Fields covered by multiple predicate bits, would do the exact same thing. Obviously, however, if CTR is modified within any given loop (mtctr) the behaviour of CTR is no longer deterministic.*

11.1.6.1 Link Register Update

For a Scalar Branch, unconditional updating of the Link Register LR is useful and practical. However, if a loop of CR Fields is tested, unconditional updating of LR becomes problematic.

For example when using `bclr` with `LRu=1,LK=0` in Horizontal-First Mode, LR's value will be unconditionally overwritten after the first element, such that for execution (testing) of the second element, LR has the value `CIA+8`. This is covered in the `bclrl` example, in a later section.

The addition of a LRu bit modifies behaviour in conjunction with LK, as follows:

- `sv.bc` When LRu=0,LK=0, Link Register is not updated
- `sv.bcl` When LRu=0,LK=1, Link Register is updated unconditionally
- `sv.bcl/lru` When LRu=1,LK=1, Link Register will only be updated if the Branch Condition fails.
- `sv.bc/lru` When LRu=1,LK=0, Link Register will only be updated if the Branch Condition succeeds.

This avoids destruction of LR during loops (particularly Vertical-First ones).

SVLR and SVSTATE

For precisely the reasons why `LK=1` was added originally to the Power ISA, with SVSTATE being a peer of the Program Counter it becomes necessary to also add an SVLR (SVSTATE Link Register) and corresponding control bits `SL` and `SLu`.

11.1.6.2 CTR-test

Where a standard Scalar v3.0B branch unconditionally decrements CTR when `BO[2]` is clear, CTR-test Mode introduces more flexibility which allows CTR to be used for many more types of Vector loops constructs.

CTR-test mode and CTi interaction is as follows: note that `BO[2]` is still required to be clear for CTR decrements to be considered, exactly as is the case in Scalar Power ISA v3.0B

- **CTR-test=0, CTi=0**: CTR decrements on a per-element basis if `BO[2]` is zero. Masked-out elements when `sz=0` are skipped (i.e. CTR is *not* decremented when the predicate bit is zero and `sz=0`).
- **CTR-test=0, CTi=1**: CTR decrements on a per-element basis if `BO[2]` is zero and a masked-out element is skipped (`sz=0` and predicate bit is zero). This one special case is the **opposite** of other combinations, as well as being completely different from normal SVP64 `sz=0` behaviour)
- **CTR-test=1, CTi=0**: CTR decrements on a per-element basis if `BO[2]` is zero and the Condition Test succeeds. Masked-out elements when `sz=0` are skipped (including not decrementing CTR)

- **CTR-test=1, CTi=1**: CTR decrements on a per-element basis if `BO[2]` is zero and the Condition Test *fails*. Masked-out elements when `sz=0` are skipped (including not decrementing CTR)

`CTR-test=0, CTi=1, sz=0` requires special emphasis because it is the only time in the entirety of SVP64 that has side-effects when a predicate mask bit is clear. **All** other SVP64 operations entirely skip an element when sz=0 and a predicate mask bit is zero. It is also critical to emphasise that in this unusual mode, no other side-effects occur: **only** CTR is decremented, i.e. the rest of the Branch operation is skipped.

11.1.6.3 VLSET Mode

VLSET Mode truncates the Vector Length so that subsequent instructions operate on a reduced Vector Length. This is similar to Data-dependent Fail-First and LD/ST Fail-First, where for VLSET the truncation occurs at the Branch decision-point.

Interestingly, due to the side-effects of `VLSET` mode it is actually useful to use Branch Conditional even to perform no actual branch operation, i.e to point to the instruction after the branch. Truncation of VL would thus conditionally occur yet control flow alteration would not.

`VLSET` mode with Vertical-First is particularly unusual. Vertical-First is designed to be used for explicit looping, where an explicit call to `svstep` is required to move both srcstep and dststep on to the next element, until VL (or other condition) is reached. Vertical-First Looping is expected (required) to terminate if the end of the Vector, VL, is reached. If however that loop is terminated early because VL is truncated, VLSET with Vertical-First becomes meaningless. Resolving this would require two branches: one Conditional, the other branching unconditionally to create the loop, where the Conditional one jumps over it.

Therefore, with `VSb`, the option to decide whether truncation should occur if the branch succeeds *or* if the branch condition fails allows for the flexibility required. This allows a Vertical-First Branch to *either* be used as a branch-back (loop) *or* as part of a conditional exit or function call from *inside* a loop, and for VLSET to be integrated into both types of decision-making.

In the case of a Vertical-First branch-back (loop), with `VSb=0` the branch takes place if success conditions are met, but on exit from that loop (branch condition fails), VL will be truncated. This is extremely useful.

`VLSET` mode with Horizontal-First when `VSb=0` is still useful, because it can be used to truncate VL to the first predicated (non-masked-out) element.

The truncation point for VL, when VLi is clear, must not include skipped elements that preceded the current element being tested. Example: `sz=0, VLi=0, predicate mask = 0b110010` and the Condition Register failure point is at CR Field element 4.

- Testing at element 0 is skipped because its predicate bit is zero
- Testing at element 1 passed
- Testing elements 2 and 3 are skipped because their respective predicate mask bits are zero
- Testing element 4 fails therefore VL is truncated to **2** not 4 due to elements 2 and 3 being skipped.

If `sz=1` in the above example *then* VL would have been set to 4 because in non-zeroing mode the zero'd elements are still effectively part of the Vector (with their respective elements set to `SNZ`)

If `VLI=1` then VL would be set to 5 regardless of sz, due to being inclusive of the element actually being tested.

11.1.6.4 VLSET and CTR-test combined

If both CTR-test and VLSET Modes are requested, it is important to observe the correct order. What occurs depends on whether VLi is enabled, because VLi affects the length, VL.

If VLi (VL truncate inclusive) is set:

1. compute the test including whether CTR triggers
2. (optionally) decrement CTR
3. (optionally) truncate VL (VSb inverts the decision)

4. decide (based on step 1) whether to terminate looping (including not executing step 5)
5. decide whether to branch.

If VLi is clear, then when a test fails that element and any following it should **not** be considered part of the Vector. Consequently:

1. compute the branch test including whether CTR triggers
2. if the test fails against VSb, truncate VL to the *previous* element, and terminate looping. No further steps executed.
3. (optionally) decrement CTR
4. decide whether to branch.

11.1.7 Boolean Logic combinations

In a Scalar ISA, Branch-Conditional testing even of vector results may be performed through inversion of tests. NOR of all tests may be performed by inversion of the scalar condition and branching *out* from the scalar loop around elements, using scalar operations.

In a parallel (Vector) ISA it is the ISA itself which must perform the prerequisite logic manipulation. Thus for SVP64 there are an extraordinary number of nesessary combinations which provide completely different and useful behaviour. Available options to combine:

- `BO[0]` to make an unconditional branch would seem irrelevant if it were not for predication and for side-effects (CTR Mode for example)
- Enabling CTR-test Mode and setting `BO[2]` can still result in the Branch taking place, not because the Condition Test itself failed, but because CTR reached zero **because**, as required by CTR-test mode, CTR was decremented as a **result** of Condition Tests failing.
- `BO[1]` to select whether the CR bit being tested is zero or nonzero
- `R30` and `~R30` and other predicate mask options including CR and inverted CR bit testing
- `sz` and `SNZ` to insert either zeros or ones in place of masked-out predicate bits
- `ALL` or `ANY` behaviour corresponding to `AND` of all tests and `OR` of all tests, respectively.
- Predicate Mask bits, which combine in effect with the CR being tested.
- Inversion of Predicate Masks (`~r3` instead of `r3`, or using `NE` rather than `EQ`) which results in an additional level of possible ANDing, ORing etc. that would otherwise need explicit instructions.

The most obviously useful combinations here are to set `BO[1]` to zero in order to turn `ALL` into Great-Big-NAND and `ANY` into Great-Big-NOR. Other Mode bits which perform behavioural inversion then have to work round the fact that the Condition Testing is NOR or NAND. The alternative to not having additional behavioural inversion (`SNZ`, `VSb`, `CTi`) would be to have a second (unconditional) branch directly after the first, which the first branch jumps over. This contrivance is avoided by the behavioural inversion bits.

11.1.8 Pseudocode and examples

Please see {SVP64 Appendix} regarding CR bit ordering and for the definition of `CR{n}`

For comparative purposes this is a copy of the v3.0B `bc` pseudocode

```
if (mode_is_64bit) then M <- 0
else M <- 32
if ¬BO[2] then CTR <- CTR - 1
ctr_ok <- BO[2] | ((CTR[M:63] != 0) ^ BO[3])
cond_ok <- BO[0] | ¬(CR[BI+32] ^ BO[1])
if ctr_ok & cond_ok then
  if AA then NIA <-iea EXTS(BD || 0b00)
  else       NIA <-iea CIA + EXTS(BD || 0b00)
if LK then LR  <-iea  CIA + 4
```

Simplified pseudocode including LRu and CTR skipping, which illustrates clearly that SVP64 Scalar Branches (VL=1) are **not** identical to v3.0B Scalar Branches. The key areas where differences occur are the inclusion of predication (which can still be used when VL=1), in when and why CTR is decremented (CTRtest Mode) and whether LR is updated (which is unconditional in v3.0B when LK=1, and conditional in SVP64 when LRu=1).

Inline comments highlight the fact that the Scalar Branch behaviour and pseudocode is still clearly visible and embedded within the Vectorised variant:

```
if (mode_is_64bit) then M <- 0
else M <- 32
# the bit of CR to test, if the predicate bit is zero,
# is overridden
testbit = CR[BI+32]
if ¬predicate_bit then testbit = SVRMmode.SNZ
# otherwise apart from the override ctr_ok and cond_ok
# are exactly the same
ctr_ok <- BO[2] | ((CTR[M:63] != 0) ^ BO[3])
cond_ok <- BO[0] | ¬(testbit ^ BO[1])
if ¬predicate_bit & ¬SVRMmode.sz then
  # this is entirely new: CTR-test mode still decrements CTR
  # even when predicate-bits are zero
  if ¬BO[2] & CTRtest & ¬CTi then
    CTR = CTR - 1
  # instruction finishes here
else
  # usual BO[2] CTR-mode now under CTR-test mode as well
  if ¬BO[2] & ¬(CTRtest & (cond_ok ^ CTi)) then CTR <- CTR - 1
  # new VLset mode, conditional test truncates VL
  if VLSET and VSb = (cond_ok & ctr_ok) then
    if SVRMmode.VLI then SVSTATE.VL = srcstep+1
    else                 SVSTATE.VL = srcstep
  # usual LR is now conditional, but also joined by SVLR
  lr_ok <- LK
  svlr_ok <- SVRMmode.SL
  if ctr_ok & cond_ok then
    if AA then NIA <-iea EXTS(BD || 0b00)
    else       NIA <-iea CIA + EXTS(BD || 0b00)
    if SVRMmode.LRu then lr_ok <- ¬lr_ok
    if SVRMmode.SLu then svlr_ok <- ¬svlr_ok
  if lr_ok   then LR   <-iea CIA + 4
  if svlr_ok then SVLR <- SVSTATE
```

Below is the pseudocode for SVP64 Branches, which is a little less obvious but identical to the above. The lack of obviousness is down to the early-exit opportunities.

Effective pseudocode for Horizontal-First Mode:

```
if (mode_is_64bit) then M <- 0
else M <- 32
cond_ok = not SVRMmode.ALL
for srcstep in range(VL):
    # select predicate bit or zero/one
    if predicate[srcstep]:
        # get SVP64 extended CR field 0..127
        SVCRf = SVP64EXTRA(BI>>2)
        CRbits = CR{SVCRf}
        testbit = CRbits[BI & 0b11]
        # testbit = CR[BI+32+srcstep*4]
```

```
        else if not SVRMmode.sz:
            # inverted CTR test skip mode
            if ¬BO[2] & CTRtest & ¬CTI then
              CTR = CTR - 1
            continue # skip to next element
        else
            testbit = SVRMmode.SNZ
        # actual element test here
        ctr_ok <- BO[2] | ((CTR[M:63] != 0) ^ BO[3])
        el_cond_ok <- BO[0] | ¬(testbit ^ BO[1])
        # check if CTR dec should occur
        ctrdec = ¬BO[2]
        if CTRtest & (el_cond_ok ^ CTi) then
           ctrdec = 0b0
        if ctrdec then CTR <- CTR - 1
        # merge in the test
        if SVRMmode.ALL:
            cond_ok &= (el_cond_ok & ctr_ok)
        else
            cond_ok |= (el_cond_ok & ctr_ok)
        # test for VL to be set (and exit)
        if VLSET and VSb = (el_cond_ok & ctr_ok) then
            if SVRMmode.VLI then SVSTATE.VL = srcstep+1
            else                 SVSTATE.VL = srcstep
            break
        # early exit?
        if SVRMmode.ALL != (el_cond_ok & ctr_ok):
             break
        # SVP64 rules about Scalar registers still apply!
        if SVCRf.scalar:
           break
    # loop finally done, now test if branch (and update LR)
    lr_ok <- LK
    svlr_ok <- SVRMmode.SL
    if cond_ok then
        if AA then NIA <-iea EXTS(BD || 0b00)
        else       NIA <-iea CIA + EXTS(BD || 0b00)
        if SVRMmode.LRu then lr_ok <- ¬lr_ok
        if SVRMmode.SLu then svlr_ok <- ¬svlr_ok
    if lr_ok then LR <-iea CIA + 4
    if svlr_ok then SVLR <- SVSTATE
```

Pseudocode for Vertical-First Mode:

```
    # get SVP64 extended CR field 0..127
    SVCRf = SVP64EXTRA(BI>>2)
    CRbits = CR{SVCRf}
    # select predicate bit or zero/one
    if predicate[srcstep]:
        if BRc = 1 then # CR0 vectorised
            CR{SVCRf+srcstep} = CRbits
        testbit = CRbits[BI & 0b11]
    else if not SVRMmode.sz:
        # inverted CTR test skip mode
        if ¬BO[2] & CTRtest & ¬CTI then
           CTR = CTR - 1
```

```
        SVSTATE.srcstep = new_srcstep
        exit # no branch testing
    else
        testbit = SVRMmode.SNZ
    # actual element test here
    cond_ok <- BO[0] | ¬(testbit ^ BO[1])
    # test for VL to be set (and exit)
    if VLSET and cond_ok = VSb then
        if SVRMmode.VLI
            SVSTATE.VL = new_srcstep+1
        else
            SVSTATE.VL = new_srcstep
```

11.1.8.1 Example Shader code

```
// assume f() g() or h() modify a and/or b
while(a > 2) {
    if(b < 5)
        f();
    else
        g();
    h();
}
```

which compiles to something like:

```
vec<i32> a, b;
// ...
pred loop_pred = a > 2;
// loop continues while any of a elements greater than 2
while(loop_pred.any()) {
    // vector of predicate bits
    pred if_pred = loop_pred & (b < 5);
    // only call f() if at least 1 bit set
    if(if_pred.any()) {
        f(if_pred);
    }
label1:
    // loop mask ANDs with inverted if-test
    pred else_pred = loop_pred & ~if_pred;
    // only call g() if at least 1 bit set
    if(else_pred.any()) {
        g(else_pred);
    }
    h(loop_pred);
}
```

which will end up as:

```
   # start from while loop test point
   b looptest
while_loop:
   sv.cmpi CR80.v, b.v, 5     # vector compare b into CR64 Vector
   sv.bc/m=r30/~ALL/sz CR80.v.LT skip_f # skip when none
   # only calculate loop_pred & pred_b because needed in f()
   sv.crand CR80.v.SO, CR60.v.GT, CR80.V.LT # if = loop & pred_b
```

```
   f(CR80.v.SO)
skip_f:
   # illustrate inversion of pred_b. invert r30, test ALL
   # rather than SOME, but masked-out zero test would FAIL,
   # therefore masked-out instead is tested against 1 not 0
   sv.bc/m=~r30/ALL/SNZ CR80.v.LT skip_g
   # else = loop & ~pred_b, need this because used in g()
   sv.crternari(A&~B) CR80.v.SO, CR60.v.GT, CR80.V.LT
   g(CR80.v.SO)
skip_g:
   # conditionally call h(r30) if any loop pred set
   sv.bclr/m=r30/~ALL/sz BO[1]=1 h()
looptest:
   sv.cmpi CR60.v a.v, 2      # vector compare a into CR60 vector
   sv.crweird r30, CR60.GT # transfer GT vector to r30
   sv.bc/m=r30/~ALL/sz BO[1]=1 while_loop
end:
```

11.1.8.2 LRu example

show why LRu would be useful in a loop. Imagine the following c code:

```
for (int i = 0; i < 8; i++) {
    if (x < y) break;
}
```

Under these circumstances exiting from the loop is not only based on CTR it has become conditional on a CR result. Thus it is desirable that NIA *and* LR only be modified if the conditions are met

v3.0 pseudocode for `bclrl`:

```
if (mode_is_64bit) then M <- 0
else M <- 32
if ¬BO[2]  then CTR <- CTR - 1
ctr_ok <- BO[2] | ((CTR[M:63] != 0) ^ BO[3])
cond_ok <- BO[0] | ¬(CR[BI+32] ^  BO[1])
if ctr_ok & cond_ok then NIA <-iea LR[0:61] || 0b00
if LK then LR <-iea CIA + 4
```

the latter part for SVP64 `bclrl` becomes:

```
for i in 0 to VL-1:
    ...
    ...
    cond_ok <- BO[0] | ¬(CR[BI+32] ^  BO[1])
    lr_ok <- LK
    if ctr_ok & cond_ok then
       NIA <-iea LR[0:61] || 0b00
       if SVRMmode.LRu then lr_ok <- ¬lr_ok
    if lr_ok then LR <-iea CIA + 4
    # if NIA modified exit loop
```

The reason why should be clear from this being a Vector loop: unconditional destruction of LR when LK=1 makes `sv.bclrl` ineffective, because the intention going into the loop is that the branch should be to the copy of LR set at the *start* of the loop, not half way through it. However if the change to LR only occurs if the branch is taken then it becomes a useful instruction.

The following pseudocode should **not** be implemented because it violates the fundamental principle of SVP64

which is that SVP64 looping is a thin wrapper around Scalar Instructions. The pseducode below is more an actual Vector ISA Branch and as such is not at all appropriate:

```
for i in 0 to VL-1:
    ...
    ...
    cond_ok <- BO[0] | ¬(CR[BI+32] ^  BO[1])
    if ctr_ok & cond_ok then NIA <-iea LR[0:61] || 0b00
# only at the end of looping is LK checked.
# this completely violates the design principle of SVP64
# and would actually need to be a separate (scalar)
# instruction "set LR to CIA+4 but retrospectively"
# which is clearly impossible
if LK then LR <-iea CIA + 4
```

[[!tag standards]]

Chapter 12

setvl instruction

12.1 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

12.1.1 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. In future this will allow 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)

condition	effect
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)

12.1.2 Examples

12.1.2.1 Core concept loop

This example illustrates the Cray-style Loop concept. However where most Cray Vectors have a Max Vector Length hard-coded into the architecture, Simple-V allows MVL to be set, but only as a static immediate, so that compilers may embed the register resource allocation statically at compile-time.

```
loop:
    setvl a3, a0, MVL=8    #  update a3 with vl
                           # (# of elements this iteration)
                           # set MVL to 8 and
                           # set a3=VL=MIN(a0,MVL)
    # do vector operations at up to 8 length (MVL=8)
    # ...
    sub. a0, a0, a3   # Decrement count by vl, set CR0.eq
    bnez a0, loop    # Any more?
```

12.1.2.2 Loop using Rc=1

In this example, the setvl. instruction enabled Rc=1, which sets CR0.eq when VL becomes zero.

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

12.1.2.3 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` specifies which registers are set in a `VEXPAND` fashion.

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

[[!tag standards]]

Chapter 13

svstep instruction

13.1 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 although there are caveats for the Deterministic use of looping with Sub-Vectors 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.

Use of svstep with Vertical-First sub-vectors

Incrementing and iteration through subvector state ssubstep and dsubstep is possible with `sv.svstep/vecN` where as expected N may be 2/3/4. However it is necessary to use the exact same Sub-Vector qualifier on any Prefixed instructions, within any given Vertical-First loop: `vec2/3/4` is **not** automatically applied to all instructions, it must be explicitly applied on a per-instruction basis. Also valid is not specifying a Sub-vector qualifier at all, but it is critically important to note that operations will be repeated. For example if `sv.svstep/vec2` is not used on `sv.addi` then each Vector element operation is repeated twice. The reason is that whilst svstep will be iterating through both the SUBVL and VL loops, the addi instruction only uses `srcstep` and `dststep` (not ssubstep or dsubstep) Illustrated below:

```
def offset():
  for step in range(VL):
    for substep in range(SUBVL=2):
      yield step, substep
for i, j in offset():
    vec2_offs = i * SUBVL + j  # calculate vec2 offset
    addi RT+i, RA+i, 1      # but sv.addi is not vec2!
    muli/vec2 RT+vec2_offs, RA+vec2_offs, 2 # this is
```

Actual assembler would be:

```
loop:
    setvl VF=1, CTRmode
    sv.addi *RT, *RA, 1      # no vec2
    sv.muli/vec2 *RT, *RA, 2 # vec2
    sv.svstep/vec2           # must match the muli
    sv.bc CTRmode, loop      # subtracts VL from CTR
```

This illustrates the correct but seemingly-anomalous behaviour: `sv.svstep/vec2` is being requested to update `SVSTATE` to follow a vec2 loop construct. The anomalous `sv.addi` is not prohibited as it may in fact be desirable to execute operations twice, or to re-load data that was overwritten, and many other possibilities.

13.2 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]]

Chapter 14

REMAP subsystem

14.1 REMAP

REMAP is an advanced form of Vector "Structure Packing" that provides hardware-level support for commonly-used *nested* loop patterns that would otherwise require full inline loop unrolling. For more general reordering an Indexed REMAP mode is available (a RISC-paradigm abstracted analog to `xxperm`).

REMAP allows the usual sequential vector loop `0..VL-1` to be "reshaped" (re-mapped) from a linear form to a 2D or 3D transposed form, or "offset" to permit arbitrary access to elements (when elwidth overrides are used), independently on each Vector src or dest register. Aside from Indexed REMAP this is entirely Hardware-accelerated reordering and consequently not costly in terms of register access. It will however place a burden on Multi-Issue systems but no more than if the equivalent Scalar instructions were explicitly loop-unrolled without SVP64, and some advanced implementations may even find the Deterministic nature of the Scheduling to be easier on resources.

The initial primary motivation of REMAP was for Matrix Multiplication, reordering of sequential data in-place: in-place DCT and FFT were easily justified given the exceptionally high usage in Computer Science. Four SPRs are provided which may be applied to any GPR, FPR or CR Field so that for example a single FMAC may be used in a single hardware-controlled 100% Deterministic loop to perform 5x3 times 3x4 Matrix multiplication, generating 60 FMACs *without needing explicit assembler unrolling*. Additional uses include regular "Structure Packing" such as RGB pixel data extraction and reforming (although less costly vec2/3/4 reshaping is achievable with `PACK/UNPACK`).

Even once designed as an independent RISC-paradigm abstraction system it was realised that Matrix REMAP could be applied to min/max instructions to achieve Floyd-Warshall Graph computations, or to AND/OR Ternary bitmanipulation to compute Warshall Transitive Closure, or to perform Cryptographic Matrix operations with Galois Field variants of Multiply-Accumulate and many more uses expected to be discovered. This *without adding actual explicit Vector opcodes for any of the same.*

Thus it should be very clear: REMAP, like all of SV, is abstracted out, meaning that unlike traditional Vector ISAs which would typically only have a limited set of instructions that can be structure-packed (LD/ST and Move operations being the most common), REMAP may be applied to literally any instruction: CRs, Arithmetic, Logical, LD/ST, even Vectorised Branch-Conditional.

When SUBVL is greater than 1 a given group of Subvector elements are kept together: effectively the group becomes the element, and with REMAP applying to elements (not sub-elements) each group is REMAPed together. Swizzle *can* however be applied to the same instruction as REMAP, providing re-sequencing of Subvector elements which REMAP cannot. Also as explained in {Swizzle Move}, {Pack / Unpack} and the {SVP64 Appendix}, Pack and Unpack Mode bits can extend down into Sub-vector elements to influence vec2/vec3/vec4 sequential reordering, but even here, REMAP reordering is not *individually* extended down to the actual sub-vector elements themselves. This keeps the relevant Predicate Mask bit applicable to the Subvector group,

just as it does when REMAP is not active.

In its general form, REMAP is quite expensive to set up, and on some implementations may introduce latency, so should realistically be used only where it is worthwhile. Given that even with latency the fact that up to 127 operations can be Deterministically issued (from a single instruction) it should be clear that REMAP should not be dismissed for *possible* latency alone. Commonly-used patterns such as Matrix Multiply, DCT and FFT have helper instruction options which make REMAP easier to use.

Future specification note: future versions of the REMAP Management instructions will extend to EXT1xx Prefixed variants. This will overcome some of the limitations present in the 32-bit variants of the REMAP Management instructions that at present require direct writing to SVSHAPE0-3 SPRs. Additional REMAP Modes may also be introduced at that time.

There are four types of REMAP:

- **Matrix**, also known as 2D and 3D reshaping, can perform in-place Matrix transpose and rotate. The Shapes are set up for an "Outer Product" Matrix Multiply.
- **FFT/DCT**, with full triple-loop in-place support: limited to Power-2 RADIX
- **Indexing**, for any general-purpose reordering, also includes limited 2D reshaping as well as Element "offsetting".
- **Parallel Reduction**, for scheduling a sequence of operations in a Deterministic fashion, in a way that may be parallelised, to reduce a Vector down to a single value.
- **Parallel Prefix Sum**, implemented as a work-efficient Schedule, has several key Computer Science uses. Again Prefix Sum is 100% Deterministic.

Best implemented on top of a Multi-Issue Out-of-Order Micro-architecture, REMAP Schedules are 100% Deterministic **including Indexing** and are designed to be incorporated in between the Decode and Issue phases, directly into Register Hazard Management.

As long as the SVSHAPE SPRs are not written to directly, Hardware may treat REMAP as 100% Deterministic: all REMAP Management instructions take static operands (no dynamic register operands) with the exception of Indexed Mode, and even then Architectural State is permitted to assume that the Indices are cacheable from the point at which the `svindex` instruction is executed.

Parallel Reduction is unusual in that it requires a full vector array of results (not a scalar) and uses the rest of the result Vector for the purposes of storing intermediary calculations. As these intermediary results are Deterministically computed they may be useful. Additionally, because the intermediate results are always written out it is possible to service Precise Interrupts without affecting latency (a common limitation of Vector ISAs implementing explicit Parallel Reduction instructions, because their Architectural State cannot hold the partial results).

14.1.1 Basic principle

The following illustrates why REMAP was added.

- normal vector element read/write of operands would be sequential (0 1 2 3 ....)
- this is not appropriate for (e.g.) Matrix multiply which requires accessing elements in alternative sequences (0 3 6 1 4 7 ...)
- normal Vector ISAs use either Indexed-MV or Indexed-LD/ST to "cope" with this. both are expensive (copy large vectors, spill through memory) and very few Packed SIMD ISAs cope with non-Power-2 (Duplicate-data inline-loop-unrolling is the costly solution)
- REMAP **redefines** the order of access according to set (Deterministic) "Schedules".
- Matrix Schedules are not at all restricted to power-of-two boundaries making it unnecessary to have for example specialised 3x4 transpose instructions of other Vector ISAs.
- DCT and FFT REMAP are RADIX-2 limited but this is the case in existing Packed/Predicated SIMD ISAs anyway (and Bluestein Convolution is typically deployed to solve that).

Only the most commonly-used algorithms in computer science have REMAP support, due to the high cost in both the ISA and in hardware. For arbitrary remapping the `Indexed` REMAP may be used.

14.1.2 Example Usage

- `svshape` to set the type of reordering to be applied to an otherwise usual `0..VL-1` hardware for-loop
- `svremap` to set which registers a given reordering is to apply to (RA, RT etc)
- `sv.{instruction}` where any Vectorised register marked by `svremap` will have its ordering REMAPPED according to the schedule set by `svshape`.

The following illustrative example multiplies a 3x4 and a 5x3 matrix to create a 5x4 result:

```
svshape 5,4,3,0,0         # Outer Product 5x4 by 4x3
svremap 15,1,2,3,0,0,0,0  # link Schedule to registers
sv.fmadds *0,*32,*64,*0   # 60 FMACs get executed here
```

- svshape sets up the four SVSHAPE SPRS for a Matrix Schedule
- svremap activates four out of five registers RA RB RC RT RS (15)
- svremap requests:
- RA to use SVSHAPE1
- RB to use SVSHAPE2
- RC to use SVSHAPE3
- RT to use SVSHAPE0
- RS Remapping to not be activated
- sv.fmadds has vectors RT=0, RA=32, RB=64, RC=0
- With REMAP being active each register's element index is *independently* transformed using the specified SHAPEs.

Thus the Vector Loop is arranged such that the use of the multiply-and-accumulate instruction executes precisely the required Schedule to perform an in-place in-registers Outer Product Matrix Multiply with no need to perform additional Transpose or register copy instructions. The example above may be executed as a unit test and demo, here

Hardware Architectural note: with the Scheduling applying as a Phase between Decode and Issue in a Deterministic fashion the Register Hazards may be easily computed and a standard Out-of-Order Micro-Architecture exploited to good effect. Even an In-Order system may observe that for large Outer Product Schedules there will be no stalls, but if the Matrices are particularly small size an In-Order system would have to stall, just as it would if the operations were loop-unrolled without Simple-V. Thus: regardless of the Micro-Architecture the Hardware Engineer should first consider how best to process the exact same equivalent loop-unrolled instruction stream.

14.1.3 Horizontal-Parallelism Hint

`SVSTATE.hphint` is an indicator to hardware of how many elements are 100% fully independent. Hardware is permitted to assume that groups of elements up to `hphint` in size need not have Register (or Memory) Hazards created between them (including when `hphint > VL`).

If care is not taken in setting `hphint` correctly it may wreak havoc. For example Matrix Outer Product relies on the innermost loop computations being independent. If `hphint` is set to greater than the Outer Product depth then data corruption is guaranteed to occur.

Likewise on FFTs it is assumed that each layer of the RADIX2 triple-loop is independent, but that there is strict *inter-layer* Register Hazards. Therefore if `hphint` is set to greater than the RADIX2 width of the FFT, data corruption is guaranteed.

Thus the key message is that setting `hphint` requires in-depth knowledge of the REMAP Algorithm Schedules, given in the Appendix.

14.1.4 REMAP types

This section summarises the motivation for each REMAP Schedule and briefly goes over their characteristics and limitations. Further details on the Deterministic Precise-Interruptible algorithms used in these Schedules is found in the {REMAP Appendix}.

14.1.4.1 Matrix (1D/2D/3D shaping)

Matrix Multiplication is a huge part of High-Performance Compute, and 3D. In many PackedSIMD as well as Scalable Vector ISAs, non-power-of-two Matrix sizes are a serious challenge. PackedSIMD ISAs, in order to cope with for example 3x4 Matrices, recommend rolling data-repetition and loop-unrolling. Aside from the cost of the load on the L1 I-Cache, the trick only works if one of the dimensions X or Y are power-two. Prime Numbers (5x7, 3x5) become deeply problematic to unroll.

Even traditional Scalable Vector ISAs have issues with Matrices, often having to perform data Transpose by pushing out through Memory and back (costly), or computing Transposition Indices (costly) then copying to another Vector (costly).

Matrix REMAP was thus designed to solve these issues by providing Hardware Assisted "Schedules" that can view what would otherwise be limited to a strictly linear Vector as instead being 2D (even 3D) *in-place* reordered. With both Transposition and non-power-two being supported the issues faced by other ISAs are mitigated.

Limitations of Matrix REMAP are that the Vector Length (VL) is currently restricted to 127: up to 127 FMAs (or other operation) may be performed in total. Also given that it is in-registers only at present some care has to be taken on regfile resource utilisation. However it is perfectly possible to utilise Matrix REMAP to perform the three inner-most "kernel" loops of the usual 6-level "Tiled" large Matrix Multiply, without the usual difficulties associated with SIMD.

Also the `svshape` instruction only provides access to part of the Matrix REMAP capability. Rotation and mirroring need to be done by programming the SVSHAPE SPRs directly, which can take a lot more instructions. Future versions of SVP64 will include EXT1xx prefixed variants (`psvshape`) which provide more comprehensive capacity and mitigate the need to write direct to the SVSHAPE SPRs.

14.1.4.2 FFT/DCT Triple Loop

DCT and FFT are some of the most astonishingly used algorithms in Computer Science. Radar, Audio, Video, R.F. Baseband and dozens more. At least two DSPs, TMS320 and Hexagon, have VLIW instructions specially tailored to FFT.

An in-depth analysis showed that it is possible to do in-place in-register DCT and FFT as long as twin-result "butterfly" instructions are provided. These can be found in the [[openpower/isa/svfparith]] page if performing IEEE754 FP transforms. *(For fixed-point transforms, equivalent 3-in 2-out integer operations would be required).* These "butterfly" instructions avoid the need for a temporary register because the two array positions being overwritten will be "in-flight" in any In-Order or Out-of-Order micro-architecture.

DCT and FFT Schedules are currently limited to RADIX2 sizes and do not accept predicate masks. Given that it is common to perform recursive convolutions combining smaller Power-2 DCT/FFT to create larger DCT/FFTs in practice the RADIX2 limit is not a problem. A Bluestein convolution to compute arbitrary length is demonstrated by Project Nayuki

14.1.4.3 Indexed

The purpose of Indexing is to provide a generalised version of Vector ISA "Permute" instructions, such as VSX `vperm`. The Indexing is abstracted out and may be applied to much more than an element move/copy, and is not limited for example to the number of bytes that can fit into a VSX register. Indexing may be applied to LD/ST (even on Indexed LD/ST instructions such as `sv.lbzx`), arithmetic operations, extsw: there is no artificial limit.

The only major caveat is that the registers to be used as Indices must not be modified by any instruction after Indexed Mode is established, and neither must MAXVL be altered. Additionally, no register used as an Index may exceed MAXVL-1.

Failure to observe these conditions results in `UNDEFINED` behaviour. These conditions allow a Read-After-Write (RAW) Hazard to be created on the entire range of Indices to be subsequently used, but a corresponding Write-After-Read Hazard by any instruction that modifies the Indices **does not have to be created**. Given the large number of registers involved in Indexing this is a huge resource saving and reduction in micro-architectural complexity. MAXVL is likewise included in the RAW Hazards because it is involved in calculating how many registers are to be considered Indices.

With these Hazard Mitigations in place, high-performance implementations may read-cache the Indices at the point where a given `svindex` instruction is called (or SVSHAPE SPRs - and MAXVL - directly altered) by issuing background GPR register file reads whilst other instructions are being issued and executed.

The original motivation for Indexed REMAP was to mitigate the need to add an expensive `mv.x` to the Scalar ISA, which was likely to be rejected as a stand-alone instruction (`GPR(RT) <- GPR(GPR(RA))`). Usually a Vector ISA would add a non-conflicting variant (as in VSX `vperm`) but it is common to need to permute by source, with the risk of conflict, that has to be resolved, for example, in AVX-512 with `conflictd`.

Indexed REMAP on the other hand **does not prevent conflicts** (overlapping destinations), which on a superficial analysis may be perceived to be a problem, until it is recalled that, firstly, Simple-V is designed specifically to require Program Order to be respected, and that Matrix, DCT and FFT all *already* critically depend on overlapping Reads/Writes: Matrix uses overlapping registers as accumulators. Thus the Register Hazard Management needed by Indexed REMAP *has* to be in place anyway.

Programmer's Note: `hphint` may be used to help hardware identify parallelism opportunities but it is critical to remember that the groupings are by `FLOOR(step/MAXVL)` not `FLOOR(REMAP(step)/MAXVL)`.

The cost compared to Matrix and other REMAPs (and Pack/Unpack) is clearly that of the additional reading of the GPRs to be used as Indices, plus the setup cost associated with creating those same Indices. If any Deterministic REMAP can cover the required task, clearly it is adviseable to use it instead.

Programmer's note: some algorithms may require skipping of Indices exceeding VL-1, not MAXVL-1. This may be achieved programmatically by performing an `sv.cmp *BF,*RA,RB` where RA is the same GPRs used in the Indexed REMAP, and RB contains the value of VL returned from `setvl`. The resultant CR Fields may then be used as Predicate Masks to exclude those operations with an Index exceeding VL-1.

14.1.4.4 Parallel Reduction

Vector Reduce Mode issues a deterministic tree-reduction schedule to the underlying micro-architecture. Like Scalar reduction, the "Scalar Base" (Power ISA v3.0B) operation is leveraged, unmodified, to give the *appearance* and *effect* of Reduction. Parallel Reduction is not limited to Power-of-two but is limited as usual by the total number of element operations (127) as well as available register file size.

In Horizontal-First Mode, Vector-result reduction **requires** the destination to be a Vector, which will be used to store intermediary results, in order to achieve a correct final result.

Given that the tree-reduction schedule is deterministic, Interrupts and exceptions can therefore also be precise. The final result will be in the first non-predicate-masked-out destination element, but due again to the deterministic schedule programmers may find uses for the intermediate results, even for non-commutative Defined Word operations.

When Rc=1 a corresponding Vector of co-resultant CRs is also created. No special action is taken: the result *and its CR Field* are stored "as usual" exactly as all other SVP64 Rc=1 operations.

Note that the Schedule only makes sense on top of certain instructions: X-Form with a Register Profile of `RT,RA,RB` is fine because two sources and the destination are all the same type. Like Scalar Reduction, nothing is prohibited: the results of execution on an unsuitable instruction may simply not make sense. With care, even 3-input instructions (madd, fmadd, ternlogi) may be used, and whilst it is down to the Programmer to

walk through the process the Programmer can be confident that the Parallel-Reduction is guaranteed 100% Deterministic.

Critical to note regarding use of Parallel-Reduction REMAP is that, exactly as with all REMAP Modes, the `svshape` instruction *requests* a certain Vector Length (number of elements to reduce) and then sets VL and MAXVL at the number of **operations** needed to be carried out. Thus, equally as importantly, like Matrix REMAP the total number of operations is restricted to 127. Any Parallel-Reduction requiring more operations will need to be done manually in batches (hierarchical recursive Reduction).

Also important to note is that the Deterministic Schedule is arranged so that some implementations *may* parallelise it (as long as doing so respects Program Order and Register Hazards). Performance (speed) of any given implementation is neither strictly defined or guaranteed. As with the Vulkan(tm) Specification, strict compliance is paramount whilst performance is at the discretion of Implementors.

Parallel-Reduction with Predication

To avoid breaking the strict RISC-paradigm, keeping the Issue-Schedule completely separate from the actual element-level (scalar) operations, Move operations are **not** included in the Schedule. This means that the Schedule leaves the final (scalar) result in the first-non-masked element of the Vector used. With the predicate mask being dynamic (but deterministic) at a superficial glance it seems this result could be anywhere.

If that result is needed to be moved to a (single) scalar register then a follow-up `sv.mv/sm=predicate rt, *ra` instruction will be needed to get it, where the predicate is the exact same predicate used in the prior Parallel-Reduction instruction.

- If there was only a single bit in the predicate then the result will not have moved or been altered from the source vector prior to the Reduction
- If there was more than one bit the result will be in the first element with a predicate bit set.

In either case the result is in the element with the first bit set in the predicate mask. Thus, no move/copy *within the Reduction itself* was needed.

Programmer's Note: For *some* hardware implementations the vector-to-scalar copy may be a slow operation, as may the Predicated Parallel Reduction itself. It may be better to perform a pre-copy of the values, compressing them (VREDUCE-style) into a contiguous block, which will guarantee that the result goes into the very first element of the destination vector, in which case clearly no follow-up predicated vector-to-scalar MV operation is needed. A VREDUCE effect is achieved by setting just a source predicate mask on Twin-Predicated operations.

Usage conditions

The simplest usage is to perform an overwrite, specifying all three register operands the same.

```
svshape parallelreduce, 6
sv.add *8, *8, *8
```

The Reduction Schedule will issue the Parallel Tree Reduction spanning registers 8 through 13, by adjusting the offsets to RT, RA and RB as necessary (see "Parallel Reduction algorithm" in a later section).

A non-overwrite is possible as well but just as with the overwrite version, only those destination elements necessary for storing intermediary computations will be written to: the remaining elements will **not** be overwritten and will **not** be zero'd.

```
svshape parallelreduce, 6
sv.add *0, *8, *8
```

However it is critical to note that if the source and destination are not the same then the trick of using a follow-up vector-scalar MV will not work.

14.1.4.5 Sub-Vector Horizontal Reduction

To achieve Sub-Vector Horizontal Reduction, Pack/Unpack should be enabled, which will turn the Schedule around such that issuing of the Scalar Defined Words is done with SUBVL looping as the inner loop not the

outer loop. Rc=1 with Sub-Vectors (SUBVL=2,3,4) is `UNDEFINED` behaviour.

Programmer's Note: Overwrite Parallel Reduction with Sub-Vectors will clearly result in data corruption. It may be best to perform a Pack/Unpack Transposing copy of the data first

14.1.4.6 Parallel Prefix Sum

This is a work-efficient Parallel Schedule that for example produces Trangular or Factorial number sequences. Half of the Prefix Sum Schedule is near-identical to Parallel Reduction. Whilst the Arithmetic mapreduce Mode (`/mr`) may achieve the same end-result, implementations may only implement Mapreduce in serial form (or give the appearance to Programmers of the same). The Parallel Prefix Schedule is *required* to be implemented in such a way that its Deterministic Schedule may be parallelised. Like the Reduction Schedule it is 100% Deterministic and consequently may be used with non-commutative operations.

14.1.5 Determining Register Hazards

For high-performance (Multi-Issue, Out-of-Order) systems it is critical to be able to statically determine the extent of Vectors in order to allocate pre-emptive Hazard protection. The next task is to eliminate masked-out elements using predicate bits, freeing up the associated Hazards.

For non-REMAP situations `VL` is sufficient to ascertain early Hazard coverage, and with SVSTATE being a high priority cached quantity at the same level of MSR and PC this is not a problem.

The problems come when REMAP is enabled. Indexed REMAP must instead use `MAXVL` as the earliest (simplest) batch-level Hazard Reservation indicator (after taking element-width overriding on the Index source into consideration), but Matrix, FFT and Parallel Reduction must all use completely different schemes. The reason is that VL is used to step through the total number of *operations*, not the number of registers. The "Saving Grace" is that all of the REMAP Schedules are 100% Deterministic.

Advance-notice Parallel computation and subsequent cacheing of all of these complex Deterministic REMAP Schedules is *strongly recommended*, thus allowing clear and precise multi-issue batched Hazard coverage to be deployed, *even for Indexed Mode*. This is only possible for Indexed due to the strict guidelines given to Programmers.

In short, there exists solutions to the problem of Hazard Management, with varying degrees of refinement possible at correspondingly increasing levels of complexity in hardware.

A reminder: when Rc=1 each result register (element) has an associated co-result CR Field (one per result element). Thus above when determining the Write-Hazards for result registers the corresponding Write-Hazards for the corresponding associated co-result CR Field must not be forgotten, *including* when Predication is used.

14.1.6 REMAP area of SVSTATE SPR

The following bits of the SVSTATE SPR are used for REMAP:

```
|32:33|34:35|36:37|38:39|40:41| 42:46 | 62     |
| --  | --  | --  | --  | --  | ----- | ------ |
|mi0  |mi1  |mi2  |mo0  |mo1  | SVme  | RMpst  |
```

mi0-2 and mo0-1 each select SVSHAPE0-3 to apply to a given register. mi0-2 apply to RA, RB, RC respectively, as input registers, and likewise mo0-1 apply to output registers (RT/FRT, RS/FRS) respectively. SVme is 5 bits (one for each of mi0-2/mo0-1) and indicates whether the SVSHAPE is actively applied or not, and if so, to which registers.

- bit 4 of SVme indicates if mi0 is applied to source RA / FRA / BA / BFA / RT / FRT
- bit 3 of SVme indicates if mi1 is applied to source RB / FRB / BB
- bit 2 of SVme indicates if mi2 is applied to source RC / FRC / BC

- bit 1 of SVme indicates if mo0 is applied to result RT / FRT / BT / BF
- bit 0 of SVme indicates if mo1 is applied to result Effective Address / FRS / RS (LD/ST-with-update has an implicit 2nd write register, RA)

The “persistence” bit if set will result in all Active REMAPs being applied indefinitely.

14.2 svremap instruction

SVRM-Form:

0	6	11	13	15	17	19	21	22:25	26:31
PO	SVme	mi0	mi1	mi2	mo0	mo1	pst	rsvd	XO

- svremap SVme,mi0,mi1,mi2,mo0,mo1,pst

Pseudo-code:

```
# registers RA RB RC RT EA/FRS SVSHAPE0-3 indices
SVSTATE[32:33] <- mi0
SVSTATE[34:35] <- mi1
SVSTATE[36:37] <- mi2
SVSTATE[38:39] <- mo0
SVSTATE[40:41] <- mo1
# enable bit for RA RB RC RT EA/FRS
SVSTATE[42:46] <- SVme
# persistence bit (applies to more than one instruction)
SVSTATE[62] <- pst
```

Special Registers Altered:

```
SVSTATE
```

`svremap` determines the relationship between registers and SVSHAPE SPRs. The bitmask `SVme` determines which registers have a REMAP applied, and mi0-mo1 determine which shape is applied to an activated register. the `pst` bit if cleared indicated that the REMAP operation shall only apply to the immediately-following instruction. If set then REMAP remains permanently enabled until such time as it is explicitly disabled, either by `setvl` setting a new MAXVL, or with another `svremap` instruction. `svindex` and `svshape2` are also capable of setting or clearing persistence, as well as partially covering a subset of the capability of `svremap` to set register-to-SVSHAPE relationships.

Programmer's Note: applying non-persistent `svremap` to an instruction that has no REMAP enabled or is a Scalar operation will obviously have no effect but the bits 32 to 46 will at least have been set in SVSTATE. This may prove useful when using `svindex` or `svshape2`.

Hardware Architectural Note: when persistence is not set it is critically important to treat the `svremap` and the following SVP64 instruction as an indivisible fused operation. *No state* is stored in the SVSTATE SPR in order to allow continuation should an Interrupt occur between the two instructions. Thus, Interrupts must be prohibited from occurring or other workaround deployed. When persistence is set this issue is moot.

It is critical to note that if persistence is clear then `svremap` is the *only* way to activate REMAP on any given (following) instruction. If persistence is set however then **all** SVP64 instructions go through REMAP as long as `SVme` is non-zero.

14.3 SHAPE Remapping SPRs

There are four "shape" SPRs, SHAPE0-3, 32-bits in each, which have the same format.

Shape is 32-bits. When SHAPE is set entirely to zeros, remapping is disabled: the register's elements are a linear (1D) vector.

0:5	6:11	12:17	18:20	21:23	24:27	28:29	30:31	Mode
xdimsz	ydimsz	zdimsz	permute	invxyz	offset	skip	mode	Matrix
xdimsz	ydimsz	SVGPR	11/	sk1/invxy	offset	elwidth	0b00	Indexed
xdimsz	mode	zdimsz	submode2	invxyz	offset	submode	0b01	DCT/FFT
rsvd	rsvd	xdimsz	rsvd	invxyz	offset	submode	0b10	Red/Sum
							0b11	rsvd

`mode` sets different behaviours (straight matrix multiply, FFT, DCT).

- **mode=0b00** sets straight Matrix Mode
- **mode=0b00** with permute=0b110 or 0b111 sets Indexed Mode
- **mode=0b01** sets "FFT/DCT" mode and activates submodes
- **mode=0b10** sets "Parallel Reduction or Prefix-Sum" Schedules.

Architectural Resource Allocation note: the four SVSHAPE SPRs are best allocated sequentially and contiguously in order that `sv.mtspr` may be used. This is safe to do as long as `SVSTATE.SVme=0`

14.3.1 Parallel Reduction / Prefix-Sum Mode

Creates the Schedules for Parallel Tree Reduction and Prefix-Sum

- **submode=0b00** selects the left operand index for Reduction
- **submode=0b01** selects the right operand index for Reduction
- **submode=0b10** selects the left operand index for Prefix-Sum
- **submode=0b11** selects the right operand index for Prefix-Sum
- When bit 0 of `invxyz` is set, the order of the indices in the inner for-loop are reversed. This has the side-effect of placing the final reduced result in the last-predicated element. It also has the indirect side-effect of swapping the source registers: Left-operand index numbers will always exceed Right-operand indices. When clear, the reduced result will be in the first-predicated element, and Left-operand indices will always be *less* than Right-operand ones.
- When bit 1 of `invxyz` is set, the order of the outer loop step is inverted: stepping begins at the nearest power-of two to half of the vector length and reduces by half each time. When clear the step will begin at 2 and double on each inner loop.

14.3.2 FFT/DCT mode

submode2=0 is for FFT. For FFT submode the following schedules may be selected:

- **submode=0b00** selects the `j` offset of the innermost for-loop of Tukey-Cooley
- **submode=0b10** selects the `j+halfsize` offset of the innermost for-loop of Tukey-Cooley
- **submode=0b11** selects the `k` of exptable (which coefficient)

When submode2 is 1 or 2, for DCT inner butterfly submode the following schedules may be selected. When submode2 is 1, additional bit-reversing is also performed.

- **submode=0b00** selects the `j` offset of the innermost for-loop, in-place

- **submode=0b010** selects the `j+halfsize` offset of the innermost for-loop, in reverse-order, in-place
- **submode=0b10** selects the `ci` count of the innermost for-loop, useful for calculating the cosine coefficient
- **submode=0b11** selects the `size` offset of the outermost for-loop, useful for the cosine coefficient `cos(ci + 0.5) * pi / size`

When submode2 is 3 or 4, for DCT outer butterfly submode the following schedules may be selected. When submode is 3, additional bit-reversing is also performed.

- **submode=0b00** selects the `j` offset of the innermost for-loop,
- **submode=0b01** selects the `j+1` offset of the innermost for-loop,

`zdimsz` is used as an in-place "Stride", particularly useful for column-based in-place DCT/FFT.

14.3.3 Matrix Mode

In Matrix Mode, skip allows dimensions to be skipped from being included in the resultant output index. this allows sequences to be repeated: `0 0 0 1 1 1 2 2 2 ...` or in the case of skip=0b11 this results in modulo `0 1 2 0 1 2 ...`

- **skip=0b00** indicates no dimensions to be skipped
- **skip=0b01** sets "skip 1st dimension"
- **skip=0b10** sets "skip 2nd dimension"
- **skip=0b11** sets "skip 3rd dimension"

invxyz will invert the start index of each of x, y or z. If invxyz[0] is zero then x-dimensional counting begins from 0 and increments, otherwise it begins from xdimsz-1 and iterates down to zero. Likewise for y and z.

offset will have the effect of offsetting the result by `offset` elements:

```
for i in 0..VL-1:
    GPR(RT + remap(i) + SVSHAPE.offset) = ....
```

this appears redundant because the register RT could simply be changed by a compiler, until element width overrides are introduced. also bear in mind that unlike a static compiler SVSHAPE.offset may be set dynamically at runtime.

xdimsz, ydimsz and zdimsz are offset by 1, such that a value of 0 indicates that the array dimensionality for that dimension is 1. any dimension not intended to be used must have its value set to 0 (dimensionality of 1). A value of xdimsz=2 would indicate that in the first dimension there are 3 elements in the array. For example, to create a 2D array X,Y of dimensionality X=3 and Y=2, set xdimsz=2, ydimsz=1 and zdimsz=0

The format of the array is therefore as follows:

```
array[xdimsz+1][ydimsz+1][zdimsz+1]
```

However whilst illustrative of the dimensionality, that does not take the "permute" setting into account. "permute" may be any one of six values (0-5, with values of 6 and 7 indicating "Indexed" Mode). The table below shows how the permutation dimensionality order works:

permute	order	array format
000	0,1,2	(xdim+1)(ydim+1)(zdim+1)
001	0,2,1	(xdim+1)(zdim+1)(ydim+1)
010	1,0,2	(ydim+1)(xdim+1)(zdim+1)
011	1,2,0	(ydim+1)(zdim+1)(xdim+1)
100	2,0,1	(zdim+1)(xdim+1)(ydim+1)
101	2,1,0	(zdim+1)(ydim+1)(xdim+1)
110	0,1	Indexed (xdim+1)(ydim+1)
111	1,0	Indexed (ydim+1)(xdim+1)

In other words, the "permute" option changes the order in which nested for-loops over the array would be done. See executable python reference code for further details.

Note: permute=0b110 and permute=0b111 enable Indexed REMAP Mode, described below

With all these options it is possible to support in-place transpose, in-place rotate, Matrix Multiply and Convolutions, without being limited to Power-of-Two dimension sizes.

14.3.4 Indexed Mode

Indexed Mode activates reading of the element indices from the GPR and includes optional limited 2D reordering. In its simplest form (without elwidth overrides or other modes):

```
def index_remap(i):
    return GPR((SVSHAPE.SVGPR<<1)+i) + SVSHAPE.offset

for i in 0..VL-1:
    element_result = ....
    GPR(RT + indexed_remap(i)) = element_result
```

With element-width overrides included, and using the pseudocode from the SVP64 [[sv/svp64/appendix#elwidth]] elwidth section this becomes:

```
def index_remap(i):
    svreg = SVSHAPE.SVGPR << 1
    srcwid = elwid_to_bitwidth(SVSHAPE.elwid)
    offs = SVSHAPE.offset
    return get_polymorphed_reg(svreg, srcwid, i) + offs

for i in 0..VL-1:
    element_result = ....
    rt_idx = indexed_remap(i)
    set_polymorphed_reg(RT, destwid, rt_idx, element_result)
```

Matrix-style reordering still applies to the indices, except limited to up to 2 Dimensions (X,Y). Ordering is therefore limited to (X,Y) or (Y,X) for in-place Transposition. Only one dimension may optionally be skipped. Inversion of either X or Y or both is possible (2D mirroring). Pseudocode for Indexed Mode (including elwidth overrides) may be written in terms of Matrix Mode, specifically purposed to ensure that the 3rd dimension (Z) has no effect:

```
def index_remap(ISHAPE, i):
    MSHAPE.skip   = 0b0 || ISHAPE.sk1
    MSHAPE.invxyz = 0b0 || ISHAPE.invxy
    MSHAPE.xdimsz = ISHAPE.xdimsz
    MSHAPE.ydimsz = ISHAPE.ydimsz
    MSHAPE.zdimsz = 0 # disabled
    if ISHAPE.permute = 0b110 # 0,1
       MSHAPE.permute = 0b000 # 0,1,2
    if ISHAPE.permute = 0b111 # 1,0
       MSHAPE.permute = 0b010 # 1,0,2
    el_idx = remap_matrix(MSHAPE, i)
    svreg = ISHAPE.SVGPR << 1
    srcwid = elwid_to_bitwidth(ISHAPE.elwid)
    offs = ISHAPE.offset
    return get_polymorphed_reg(svreg, srcwid, el_idx) + offs
```

The most important observation above is that the Matrix-style remapping occurs first and the Index lookup second. Thus it becomes possible to perform in-place Transpose of Indices which may have been costly to set

up or costly to duplicate (waste register file space). In other words: it is fine for two or more SVSHAPEs to simultaneously use the same Indices, but one SVSHAPE has different 2D dimensions and ordering from the others.

14.4 svshape instruction

SVM-Form

```
svshape SVxd,SVyd,SVzd,SVRM,vf
```

0:5	6:10	11:15	16:20	21:24	25	26:31	name
PO	SVxd	SVyd	SVzd	SVRM	vf	XO	svshape

See {REMAP Appendix} for `svshape` pseudocode

Special Registers Altered:

```
SVSTATE, SVSHAPE0-3
```

`svshape` is a convenience instruction that reduces instruction count for common usage patterns, particularly Matrix, DCT and FFT. It sets up (overwrites) all required SVSHAPE SPRs and also modifies SVSTATE including VL and MAXVL. Using `svshape` therefore does not also require `setvl`.

Fields:

- **SVxd** - SV REMAP "xdim" (X-dimension)
- **SVyd** - SV REMAP "ydim" (Y-dimension, sometimes used for sub-mode selection)
- **SVzd** - SV REMAP "zdim" (Z-dimension)
- **SVRM** - SV REMAP Mode (0b00000 for Matrix, 0b00001 for FFT etc.)
- **vf** - sets "Vertical-First" mode
- **XO** - standard 6-bit XO field

Note: SVxd, SVyz and SVzd are all stored "off-by-one". In the assembler mnemonic the values `1-32` are stored in binary as `0b00000..0b11111`

There are 12 REMAP Modes (2 Modes are RESERVED for `svshape2`, 2 Modes are RESERVED)

SVRM	Remap Mode description
0b0000	Matrix 1/2/3D
0b0001	FFT Butterfly
0b0010	reserved
0b0011	DCT Outer butterfly
0b0100	DCT Inner butterfly, on-the-fly (Vertical-First Mode)
0b0101	DCT COS table index generation
0b0110	DCT half-swap
0b0111	Parallel Reduction and Prefix Sum
0b1000	reserved for svshape2
0b1001	reserved for svshape2
0b1010	reserved
0b1011	iDCT Outer butterfly
0b1100	iDCT Inner butterfly, on-the-fly (Vertical-First Mode)
0b1101	iDCT COS table index generation
0b1110	iDCT half-swap
0b1111	FFT half-swap

Examples showing how all of these Modes operate exists in the online SVP64 unit tests. Explaining these Modes further in detail is beyond the scope of this document.

In Indexed Mode, there are only 5 bits available to specify the GPR to use, out of 128 GPRs (7 bit numbering). Therefore, only the top 5 bits are given in the `SVxd` field: the bottom two implicit bits will be zero (`SVxd ||`

0b00).

svshape has *limited applicability* due to being a 32-bit instruction. The full capability of SVSHAPE SPRs may be accessed by directly writing to SVSHAPE0-3 with mtspr. Circumstances include Matrices with dimensions larger than 32, and in-place Transpose. Potentially a future v3.1 Prefixed instruction, psvshape, may extend the capability here.

Programmer's Note: Parallel Reduction Mode is selected by setting SVRM=7,SVyd=1. Prefix Sum Mode is selected by setting SVRM=7,SVyd=3:

```
# Vector length of 8.
svshape 8, 3, 1, 0x7, 0
# activate SVSHAPE0 (prefix-sum lhs) for RA
# activate SVSHAPE1 (prefix-sum rhs) for RT and RB
svremap 7, 0, 1, 0, 1, 0, 0
sv.add *10, *10, *10
```

Architectural Resource Allocation note: the SVRM field is carefully crafted to allocate two Modes, corresponding to bits 21-23 within the instruction being set to the value 0b100, to svshape2 (not svshape). These two Modes are considered "RESERVED" within the context of svshape but it is absolutely critical to allocate the exact same pattern in XO for both instructions in bits 26-31.

14.5 svindex instruction

SVI-Form

0:5	6:10	11:15	16:20	21:25	26:31	Form
PO	SVG	rmm	SVd	ew/yx/mm/sk	XO	SVI-Form

- svindex SVG,rmm,SVd,ew,SVyx,mm,sk

See {REMAP Appendix} for `svindex` pseudocode

Special Registers Altered:

```
SVSTATE, SVSHAPE0-3
```

`svindex` is a convenience instruction that reduces instruction count for Indexed REMAP Mode. It sets up (overwrites) all required SVSHAPE SPRs and **unlike** `svshape` can modify the REMAP area of the SVSTATE SPR as well, including setting persistence. The relevant SPRs *may* be directly programmed with `mtspr` however it is laborious to do so: svindex saves instructions covering much of Indexed REMAP capability.

Fields:

- **SVd** - SV REMAP x/y dim
- **rmm** - REMAP mask: sets remap mi0-2/mo0-1 and SVSHAPEs, controlled by mm
- **ew** - sets element width override on the Indices
- **SVG** - GPR SVG<<2 to be used for Indexing
- **yx** - 2D reordering to be used if yx=1
- **mm** - mask mode. determines how `rmm` is interpreted.
- **sk** - Dimension skipping enabled

Note: SVd, like SVxd, SVyz and SVzd of `svshape`, are all stored "off-by-one". In the assembler mnemonic the values `1-32` are stored in binary as `0b00000..0b11111`.

Note: when `yx=1,sk=0` the second dimension is calculated as `CEIL(MAXVL/SVd)`.

When `mm=0`:

- `rmm`, like REMAP.SVme, has bit 0 correspond to mi0, bit 1 to mi1, bit 2 to mi2, bit 3 to mo0 and bit 4 to mi1
- all SVSHAPEs and the REMAP parts of SVSHAPE are first reset (initialised to zero)
- for each bit set in the 5-bit `rmm`, in order, the first as-yet-unset SVSHAPE will be updated with the other operands in the instruction, and the REMAP SPR set.
- If all 5 bits of `rmm` are set then both mi0 and mo1 use SVSHAPE0.
- SVSTATE persistence bit is cleared
- No other alterations to SVSTATE are carried out

Example 1: if rmm=0b00110 then SVSHAPE0 and SVSHAPE1 are set up, and the REMAP SPR set so that mi1 uses SVSHAPE0 and mi2 uses mi2. REMAP.SVme is also set to 0b00110, REMAP.mi1=0 (SVSHAPE0) and REMAP.mi2=1 (SVSHAPE1)

Example 2: if rmm=0b10001 then again SVSHAPE0 and SVSHAPE1 are set up, but the REMAP SPR is set so that mi0 uses SVSHAPE0 and mo1 uses SVSHAPE1. REMAP.SVme=0b10001, REMAP.mi0=0, REMAP.mo1=1

Rough algorithmic form:

```
marray = [mi0, mi1, mi2, mo0, mo1]
idx = 0
for bit = 0 to 4:
    if not rmm[bit]: continue
    setup(SVSHAPE[idx])
    SVSTATE{marray[bit]} = idx
```

```
idx = (idx+1) modulo 4
```

When mm=1:

- bits 0-2 (MSB0 numbering) of rmm indicate an index selecting mi0-mo1
- bits 3-4 (MSB0 numbering) of rmm indicate which SVSHAPE 0-3 shall be updated
- only the selected SVSHAPE is overwritten
- only the relevant bits in the REMAP area of SVSTATE are updated
- REMAP persistence bit is set.

Example 1: if rmm=0b01110 then bits 0-2 (MSB0) are 0b011 and bits 3-4 are 0b10. thus, mo0 is selected and SVSHAPE2 to be updated. REMAP.SVme[3] will be set high and REMAP.mo0 set to 2 (SVSHAPE2).

Example 2: if rmm=0b10011 then bits 0-2 (MSB0) are 0b100 and bits 3-4 are 0b11. thus, mo1 is selected and SVSHAPE3 to be updated. REMAP.SVme[4] will be set high and REMAP.mo1 set to 3 (SVSHAPE3).

Rough algorithmic form:

```
marray = [mi0, mi1, mi2, mo0, mo1]
bit = rmm[0:2]
idx = rmm[3:4]
setup(SVSHAPE[idx])
SVSTATE{marray[bit]} = idx
SVSTATE.pst = 1
```

In essence, mm=0 is intended for use to set as much of the REMAP State SPRs as practical with a single instruction, whilst mm=1 is intended to be a little more refined.

Usage guidelines

- **Disable 2D mapping**: to only perform Indexing without reordering use SVd=1,sk=0,yx=0 (or set SVd to a value larger or equal to VL)
- **Modulo 1D mapping**: to perform Indexing cycling through the first N Indices use SVd=N,sk=0,yx=0 where VL>N. There is no requirement to set VL equal to a multiple of N.
- **Modulo 2D transposed**: SVd=M,sk=0,yx=1, sets xdim=M,ydim=CEIL(MAXVL/M).

Beyond these mappings it becomes necessary to write directly to the SVSTATE SPRs manually.

14.6 svshape2 (offset-priority)

SVM2-Form

0:5	6:9	10	11:15	16:20	21:24	25	26:31	Form
PO	offs	yx	rmm	SVd	100/mm	sk	XO	SVM2-Form

- svshape2 offs,yx,rmm,SVd,sk,mm

See {REMAP Appendix} for `svshape2` pseudocode

Special Registers Altered:

```
SVSTATE, SVSHAPE0-3
```

`svshape2` is an additional convenience instruction that prioritises setting `SVSHAPE.offset`. Its primary purpose is for use when element-width overrides are used. It has identical capabilities to `svindex` in terms of both options (skip, etc.) and ability to activate REMAP (rmm, mask mode) but unlike `svindex` it does not set GPR REMAP: only a 1D or 2D `svshape`, and unlike `svshape` it can set an arbitrary `SVSHAPE.offset` immediate.

One of the limitations of Simple-V is that Vector elements start on the boundary of the Scalar regfile, which is fine when element-width overrides are not needed. If the starting point of a Vector with smaller elwidths must begin in the middle of a register, normally there would be no way to do so except through costly LD/ST. `SVSHAPE.offset` caters for this scenario and `svshape2` makes it easier to access.

Operand Fields:

- **offs** (4 bits) - unsigned offset
- **yx** (1 bit) - swap XY to YX
- **SVd** dimension size
- **rmm** REMAP mask
- **mm** mask mode
- **sk** (1 bit) skips 1st dimension if set

Dimensions are calculated exactly as `svindex`. `rmm` and `mm` are as per `svindex`.

*Programmer's Note: offsets for `svshape2` may be specified in the range 0-15. Given that the principle of Simple-V is to fit on top of byte-addressable register files and that GPR and FPR are 64-bit (8 bytes) it should be clear that the offset may, when `elwidth=8`, begin an element-level operation starting element zero at any arbitrary byte. On cursory examination attempting to go beyond the range 0-7 seems unnecessary given that the **next GPR or FPR** is an alias for an offset in the range 8-15. Thus by simply increasing the starting Vector point of the operation to the next register it can be seen that the offset of 0-7 would be sufficient. Unfortunately however some operations are EXTRA2-encoded it is **not possible** to increase the GPR/FPR register number by one, because EXTRA2-encoding of GPR/FPR Vector numbers are restricted to even numbering. For CR Fields the EXTRA2 encoding is even more sparse. The additional offset range (8-15) helps overcome these limitations.*

*Hardware Implementor's note: with the offsets only being immediates and with register numbering being entirely immediate as well it is possible to correctly compute Register Hazards without requiring reading the contents of any SPRs. If however there are instructions that have directly written to the SVSTATE or SVSHAPE SPRs and those instructions are still in-flight then this position is clearly **invalid**. This is why Programmers are strongly discouraged from directly writing to these SPRs.*

Architectural Resource Allocation note: this instruction shares the space of `svshape`. Therefore it is critical that the two instructions, `svshape` and `svshape2` have the exact same XO in bits 26 thru 31. It is also critical that for `svshape2`, bit 21 of XO is a 1, bit 22 of XO is a 0, and bit 23 of XO is a 0.

[[!tag standards]]

Chapter 15

Swizzle Move

[[!tag standards]]

15.1 mv.swizzle

Links

- https://bugs.libre-soc.org/show_bug.cgi?id=139
- https://lists.libre-soc.org/pipermail/libre-soc-dev/2022-June/004913.html

Swizzle is a type of permute shorthand allowing arbitrary selection of elements from vec2/3/4 creating a new vec2/3/4. Their value lies in the high occurrence of Swizzle in 3D Shader Binaries (over 10% of all instructions). Swizzle is usually done on a per-vec-operand basis in 3D GPU ISAs, making for extremely long instructions (64 bits or greater), however it is not practical to add two or more sets of 12-bit prefixes into a single instruction. A compromise is to provide a Swizzle "Move": one such move is then required for each operand used in a subsequent instruction. The encoding for Swizzle Move embeds static predication into the swizzle as well as constants 1/1.0 and 0/0.0, and if Saturation is enabled maximum arithmetic constants may be placed into the destination as well.

An extremely important aspect of 3D GPU workloads is that the source and destination subvector lengths may be *different*. A vector of contiguous array of vec3 (XYZ) may only have 2 elements (ZY) swizzle-copied to a contiguous array of vec2. A contiguous array of vec2 sources may have multiple of each vec2 elements (XY) copied to a contiguous vec4 array (YYXX or XYXX). For this reason, *when Vectorised* Swizzle Moves support independent subvector lengths for both source and destination.

Although conceptually similar to `vpermd` of Packed SIMD VSX, Swizzle Moves come in immediate-only form with only up to four selectors, where VSX refers to individual bytes and may not copy constants to the destination. 3D Shader programs commonly use the letters "XYZW" when referring to the four swizzle indices, and also often use the letters "RGBA" if referring to pixel data. These designations are also part of both the OpenGL(TM) and Vulkan(TM) specifications.

As a standalone Scalar operation this instruction is valuable if Prefixed with SVP64Single (providing Predication). Combined with `cmpi` it synthesises Compare-and-Swap.

15.2 Format

0.5	6.10	11.15	16.27	28.31	name	Form
PO	RTp	RAp	imm	0011	mv.swiz	DQ-Form

0.5	6.10	11.15	16.27	28.31	name	Form
PO	RTp	RAp	imm	1011	fmv.swiz	DQ-Form

this gives a 12 bit immediate across bits 16 to 27. Each swizzle mnemonic (XYZW), commonly known from 3D GPU programming, has an associated index. 3 bits of the immediate are allocated to each:

imm	0.2	3.5	6.8	9.11
swizzle	X	Y	Z	W
pixel	R	G	B	A
index	0	1	2	3

The options for each Swizzle are:

- 0b000 to indicate "skip". this is equivalent to predicate masking
- 0b001 subvector length end marker (length=4 if not present)
- 0b010 to indicate "constant 0"
- 0b011 to indicate "constant 1" (or 1.0)
- 0b1NN index 0 thru 3 to copy from subelement in pos XYZW

In very simplistic terms the relationship between swizzle indices (NN, above), source, and destination is:

```
dest[i] = src[swiz[i]]
```

Note that 8 options are needed (not 6) because option 0b001 encodes the subvector length, and option 0b000 allows static predicate masking (skipping) to be encoded within the swizzle immediate. For example it allows "W.Y." to specify: "copy W to position X, and Y to position Z, leave the other two positions Y and W unaltered"

```
0    1    2    3
X    Y    Z    W  source
     |         |
     +----+    |
     .    |    |
+--------------+
|    .    |    .
W    .    Y    .  swizzle
|    .    |    .
|    Y    |    W  Y,W unmodified
|    .    |    .
W    Y    Y    W  dest
```

As a Scalar instruction

Given that XYZW Swizzle can select simultaneously between one *and four* register operands, a full version of this instruction would be an eye-popping 8 64-bit operands: 4-in, 4-out. As part of a Scalar ISA this not practical. A compromise is to cut the registers required by half, placing it on-par with `lq`, `stq` and Indexed Load-with-update instructions. When part of the Scalar Power ISA (not SVP64 Vectorised) mv.swiz and fmv.swiz operate on four 32-bit quantities, reducing this instruction to a feasible 2-in, 2-out pairs of 64-bit registers:

swizzle name	source	dest	half
X	RA	RT	lo-half
Y	RA	RT	hi-half
Z	RA+1	RT+1	lo-half
W	RA+1	RT+1	hi-half

When `RA=RT` (in-place swizzle) any portion of RT not covered by the Swizzle is unmodified. For example a Swizzle of "..XY" will copy the contents RA+1 into RT but leave RT+1 unmodified.

When `RA!=RT` any part of RT or RT+1 not set as a destination by the Swizzle will be set to zero. A Swizzle of "..XY" would copy the contents RA+1 into RT, but set RT+1 to zero.

Also, making life easier, RT and RA are only permitted to be even (no overlapping can occur). This makes RT (and RA) a "pair" exactly as in `lq` and `stq`. Scalar Swizzle instructions must be atomically indivisible: an Exception or Interrupt may not occur during the Moves.

Note that unlike the Vectorised variant, when `RT=RA` the Scalar variant *must* buffer (read) both 64-bit RA registers before writing to the RT pair (in an Out-of-Order Micro-architecture, both of the register pair must be "in-flight"). This ensures that register file corruption does not occur.

SVP64 Vectorised

Vectorised Swizzle may be considered to contain an extended static predicate mask for subvectors (SUBVL=2/3/4). Due to the skipping caused by the static predication capability, the destination subvector length can be *different* from the source subvector length, and consequently the destination subvector length is encoded into the Swizzle.

When Vectorised, given the use-case is for a High-performance GPU, the fundamental assumption is that Micro-coding or other technique will be deployed in hardware to issue multiple Scalar MV operations and full parallel crossbars, which would be impractical in a smaller Scalar-only Micro-architecture. Therefore the restriction imposed on the Scalar `mv.swiz` to 32-bit quantities as the default is lifted on `sv.mv.swiz`.

Additionally, in order to make life easier for implementers, some of whom may wish, especially for Embedded GPUs, to use multi-cycle Micro-coding, the usual strict Element-level Program Order is relaxed. An overlap between all and any Vectorised sources and destination Elements for the entirety of the Vector Loop `0..VL-1` is `UNDEFINED` behaviour.

This in turn implies that Traps and Exceptions are, as usual, permitted in between element-level moves, because due to there being no overlap there is no risk of destroying a source with an overwrite. This is *unlike* the Scalar variant which, when `RT=RA`, must buffer both halves of the RT pair.

Determining the source and destination subvector lengths is tricky. Swizzle Pseudocode:

```
swiz[0] = imm[0:3]   # X
swiz[1] = imm[3:6]   # Y
swiz[2] = imm[6:9]   # Z
swiz[3] = imm[9:12]  # W
# determine implied subvector length from Swizzle
dst_subvl = 4
for i in range(4):
    if swiz[i] == 0b001:
        dst_subvl = i+1
        break
```

What is going on here is that the option is provided to have different source and destination subvector lengths, by exploiting redundancy in the Swizzle Immediate. With the Swizzles marking what goes into each destination position, the marker "0b001" may be used to indicate the end. If no marker is present then the destination subvector length may be assumed to be 4. SUBVL is considered to be the "source" subvector length.

Pseudocode exploiting python "yield" for clarity: element-width overrides, Saturation and Predication also left out, for clarity:

```
def index_src():
    for i in range(VL):
        for j in range(SUBVL):
            if swiz[j] == 0b000: # skip
                continue
            if swiz[j] == 0b001: # end
                break
```

```
                if swiz[j] in [0b010, 0b011]:
                    yield (i*SUBVL, CONSTANT)
                else:
                    yield (i*SUBVL, swiz[j]-3)

        def index_dest():
            for i in range(VL):
                for j in range(dst_subvl):
                    if swiz[j] == 0b000: # skip
                        continue
                    yield i*dst_subvl+j

        # walk through both source and dest indices simultaneously
        for (src_idx, offs), dst_idx in zip(index_src(), index_dst()):
            if offs == CONSTANT:
                 set(RT+dst_idx, CONSTANT)
            else
                 move_operation(RT+dst_idx, RA+src_idx+offs)
```

Vertical-First Mode

It is important to appreciate that *only* the main loop VL is Vertical-First: the SUBVL loop is not. This makes sense from the perspective that the Swizzle Move is a group of moves, but is still a single instruction that happens to take vec2/3/4 as operands. Vertical-First only performing one of the *sub*-elements at a time rather than operating on the entire vec2/3/4 together would violate that expectation. The exceptions to this, explained later, are when Pack/Unpack is enabled.

Effect of Saturation on Vectorised Swizzle

A useful convenience for pixel data is to be able to insert values 0x7f or 0xff as magic constants for arbitrary R,G,B or A. Therefore, when Saturation is enabled and a Swizzle=0b011 (Constant 1) is requested, the maximum permitted Saturated value is inserted rather than Constant 1. `sv.mv.swiz/sats/vec2/ew=8 RT.v, RA.v, Y1` would insert the 2nd subelement (Y) into the first destination subelement and the signed-maximum constant 0x7f into the second. A Constant 0 Swizzle on the other hand still inserts zero because there is no encoding space to select between -1, 0 and 1, and 0 and max values are more useful.

15.3 Pack/Unpack Mode:

It is possible to apply Pack and Unpack to Vectorised swizzle moves. The interaction requires specific explanation because it involves the separate SUBVLs (with destination SUBVL being separate). Key to understanding is that the source and destination SUBVL be "outer" loops instead of inner loops, exactly as in {REMAP subsystem} Matrix mode, under the control of `SVSTATE.PACK` and `SVSTATE.UNPACK`.

Illustrating a "normal" SVP64 operation with `SUBVL!=1` (assuming no elwidth overrides):

```
        def index():
            for i in range(VL):
                for j in range(SUBVL):
                    yield i*SUBVL+j

        for idx in index():
            operation_on(RA+idx)
```

For a separate source/dest SUBVL (again, no elwidth overrides):

```
        # yield an outer-SUBVL or inner VL loop with SUBVL
        def index_dest(outer):
            if outer:
```

```
                for j in range(dst_subvl):
                    for i in range(VL):
                        yield j*VL+i
            else:
                for i in range(VL):
                    for j in range(dst_subvl):
                        yield i*dst_subvl+j

        # yield an outer-SUBVL or inner VL loop with SUBVL
        def index_src(outer):
            if outer:
                for j in range(SUBVL):
                    for i in range(VL):
                        yield j*VL+i
            else:
                for i in range(VL):
                    for j in range(SUBVL):
                        yield i*SUBVL+j
```

"yield" from python is used here for simplicity and clarity. The two Finite State Machines for the generation of the source and destination element offsets progress incrementally in lock-step.

Just as in {Pack / Unpack}, when `PACK_en` is set it is the source that swaps to Outer-subvector loops, and when `UNPACK_en` is set it is the destination that swaps its loop-order. Setting both `PACK_en` and `UNPACK_en` is neither prohibited nor `UNDEFINED` because the behaviour is fully deterministic.

However, in Vertical-First Mode, when both are enabled, with both source and destination being outer loops a **single** step of srstep and dststep is performed. Contrast this when one of `PACK_en` is set, it is the *destination* that is an inner subvector loop, and therefore Vertical-First runs through the entire `dst_subvl` group. Likewise when `UNPACK_en` is set it is the source subvector that is run through as a group.

```
    if VERTICAL_FIRST:
        # must run through SUBVL or dst_subvl elements, to keep
        # the subvector "together".  weirdness occurs due to
        # PACK_en/UNPACK_en
        num_runs = SUBVL # 1-4
        if PACK_en:
            num_runs = dst_subvl # destination still an inner loop
        if PACK_en and UNPACK_en:
            num_runs = 1 # both are outer loops
        for substep in num_runs:
            (src_idx, offs) = yield from index_src(PACK_en)
            dst_idx = yield from index_dst(UNPACK_en)
            move_operation(RT+dst_idx, RA+src_idx+offs)
```

Chapter 16

Pack / Unpack

[[!tag standards]]

16.1 Vector Pack/Unpack operations

In the SIMD VSX set, section 6.8.1 and 6.8.2 p254 of v3.0B has a series of pack and unpack operations. Additional pixel pack/unpack instructions also exist.

In SVP64, Pack and Unpack are achieved *in the abstract* for application on *all* Vectoriseable instructions.

- See https://bugs.libre-soc.org/show_bug.cgi?id=230#c30
- https://lists.libre-soc.org/pipermail/libre-soc-dev/2022-June/004911.html

The effect of Pack and unpack could be covered by {REMAP subsystem} by using Matrix 2D layouts on either source or destination but is quite expensive to do so. Additionally, with pressure on the Scalar 32-bit opcode space it is more appropriate to compromise by adding required capability in SVP64 as a high priority (part of the Management Instructions). {Swizzle Move} is sufficiently unusual to justify a base Scalar 32-bit instruction but pack/unpack is not. What, ultimately, was decided, was to make Pack/Unpack part of the `SVSTATE` [[sv/spr]].

16.2 SVSTATE Pack/unpack Mode bits

Described in {SVP64 Appendix} the Pack/Unpack Modes allow selective Transposition of Sub-vector elements, on both source and destination. {Swizzle Move} is unique in that the Subvector length may be different for source and destination.

Appendix A

SVP64 Appendix

A.1 Appendix

- https://bugs.libre-soc.org/show_bug.cgi?id=574 Saturation
- https://bugs.libre-soc.org/show_bug.cgi?id=558#c47 Parallel Prefix
- https://bugs.libre-soc.org/show_bug.cgi?id=697 Reduce Modes
- https://bugs.libre-soc.org/show_bug.cgi?id=864 parallel prefix simulator
- https://bugs.libre-soc.org/show_bug.cgi?id=809 OV sv.addex discussion
- ARM SVE Fault-first https://alastairreid.github.io/papers/sve-ieee-micro-2017.pdf

This is the appendix to {SVP64 Chapter}, providing explanations of modes etc. leaving the main svp64 page's primary purpose as outlining the instruction format.

Table of contents:

[[!toc]]

A.1.1 Partial Implementations

It is perfectly legal to implement subsets of SVP64 as long as illegal instruction traps are always raised on unimplemented features, so that soft-emulation is possible, even for future revisions of SVP64. With SVP64 being partly controlled through contextual SPRs, a little care has to be taken.

All SPRs not implemented including reserved ones for future use must raise an illegal instruction trap if read or written. This allows software the opportunity to emulate the context created by the given SPR.

See {Compliancy Levels} for full details.

A.1.2 XER, SO and other global flags

Vector systems are expected to be high performance. This is achieved through parallelism, which requires that elements in the vector be independent. XER SO/OV and other global "accumulation" flags (CR.SO) cause Read-Write Hazards on single-bit global resources, having a significant detrimental effect.

Consequently in SV, XER.SO behaviour is disregarded (including in `cmp` instructions). XER.SO is not read, but XER.OV may be written, breaking the Read-Modify-Write Hazard Chain that complicates microarchitectural implementations. This includes when `scalar identity behaviour` occurs. If precise OpenPOWER v3.0/1 scalar behaviour is desired then OpenPOWER v3.0/1 instructions should be used without an SV Prefix.

TODO jacob add about OV https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf

Of note here is that XER.SO and OV may already be disregarded in the Power ISA v3.0/1 SFFS (Scalar Fixed and Floating) Compliancy Subset. SVP64 simply makes it mandatory to disregard XER.SO even for other Subsets, but only for SVP64 Prefixed Operations.

XER.CA/CA32 on the other hand is expected and required to be implemented according to standard Power ISA Scalar behaviour. Interestingly, due to SVP64 being in effect a hardware for-loop around Scalar instructions executing in precise Program Order, a little thought shows that a Vectorised Carry-In-Out add is in effect a Big Integer Add, taking a single bit Carry In and producing, at the end, a single bit Carry out. High performance implementations may exploit this observation to deploy efficient Parallel Carry Lookahead.

```
# assume VL=4, this results in 4 sequential ops (below)
sv.adde r0.v, r4.v, r8.v

# instructions that get executed in backend hardware:
adde r0, r4, r8 # takes carry-in, produces carry-out
adde r1, r5, r9 # takes carry from previous
...
adde r3, r7, r11 # likewise
```

It can clearly be seen that the carry chains from one 64 bit add to the next, the end result being that a 256-bit "Big Integer Add with Carry" has been performed, and that CA contains the 257th bit. A one-instruction 512-bit Add-with-Carry may be performed by setting VL=8, and a one-instruction 1024-bit Add-with-Carry by setting VL=16, and so on. More on this in [[openpower/sv/biginteger]]

A.1.3 EXTRA Field Mapping

The purpose of the 9-bit EXTRA field mapping is to mark individual registers (RT, RA, BFA) as either scalar or vector, and to extend their numbering from 0..31 in Power ISA v3.0 to 0..127 in SVP64. Three of the 9 bits may also be used up for a 2nd Predicate (Twin Predication) leaving a mere 6 bits for qualifying registers. As can be seen there is significant pressure on these (and in fact all) SVP64 bits.

In Power ISA v3.1 prefixing there are bits which describe and classify the prefix in a fashion that is independent of the suffix. MLSS for example. For SVP64 there is insufficient space to make the SVP64 Prefix "self-describing", and consequently every single Scalar instruction had to be individually analysed, by rote, to craft an EXTRA Field Mapping. This process was semi-automated and is described in this section. The final results, which are part of the SVP64 Specification, are here: [[openpower/opcode_regs_deduped]]

- Firstly, every instruction's mnemonic (`add RT, RA, RB`) was analysed from reading the markdown formatted version of the Scalar pseudocode which is machine-readable and found in [[openpower/isatables]]. The analysis gives, by instruction, a "Register Profile". `add RT, RA, RB` for example is given a designation `RM-2R-1W` because it requires two GPR reads and one GPR write.
- Secondly, the total number of registers was added up (2R-1W is 3 registers) and if less than or equal to three then that instruction could be given an EXTRA3 designation. Four or more is given an EXTRA2 designation because there are only 9 bits available.
- Thirdly, the instruction was analysed to see if Twin or Single Predication was suitable. As a general rule this was if there was only a single operand and a single result (`extw` and LD/ST) however it was found that some 2 or 3 operand instructions also qualify. Given that 3 of the 9 bits of EXTRA had to be sacrificed for use in Twin Predication, some compromises were made, here. LDST is Twin but also has 3 operands in some operations, so only EXTRA2 can be used.
- Fourthly, a packing format was decided: for 2R-1W an EXTRA3 indexing could have been decided that RA would be indexed 0 (EXTRA bits 0-2), RB indexed 1 (EXTRA bits 3-5) and RT indexed 2 (EXTRA bits 6-8). In some cases (LD/ST with update) RA-as-a-source is given a **different** EXTRA index from RA-as-a-result (because it is possible to do, and perceived to be useful). Rc=1 co-results (CR0, CR1) are always given the same EXTRA index as their main result (RT, FRT).
- Fifthly, in an automated process the results of the analysis were outputted in CSV Format for use in machine-readable form by sv_analysis.py https://git.libre-soc.org/?p=openpower-isa.git;a=blob;f=src/openpower/sv/sv_analysis.py;hb=HEAD

This process was laborious but logical, and, crucially, once a decision is made (and ratified) cannot be reversed. Qualifying future Power ISA Scalar instructions for SVP64 is **strongly** advised to utilise this same process and the same sv_analysis.py program as a canonical method of maintaining the relationships. Alterations to that same program which change the Designation is **prohibited** once finalised (ratified through the Power ISA WG Process). It would be similar to deciding that `add` should be changed from X-Form to D-Form.

A.1.4 Single Predication

This is a standard mode normally found in Vector ISAs. every element in every source Vector and in the destination uses the same bit of one single predicate mask.

In SVSTATE, for Single-predication, implementors MUST increment both srcstep and dststep, but depending on whether sz and/or dz are set, srcstep and dststep can still potentially become different indices. Only when sz=dz is srcstep guaranteed to equal dststep at all times.

Note that in some Mode Formats there is only one flag (zz). This indicates that *both* sz *and* dz are set to the same.

Example 1:

- VL=4
- mask=0b1101
- sz=0, dz=1

The following schedule for srcstep and dststep will occur:

srcstep	dststep	comment
0	0	both mask[src=0] and mask[dst=0] are 1
1	2	sz=1 but dz=0: dst skips mask[1], src soes not
2	3	mask[src=2] and mask[dst=3] are 1
3	end	loop has ended because dst reached VL-1

Example 2:

- VL=4
- mask=0b1101
- sz=1, dz=0

The following schedule for srcstep and dststep will occur:

srcstep	dststep	comment
0	0	both mask[src=0] and mask[dst=0] are 1
2	1	sz=0 but dz=1: src skips mask[1], dst does not
3	2	mask[src=3] and mask[dst=2] are 1
end	3	loop has ended because src reached VL-1

In both these examples it is crucial to note that despite there being a single predicate mask, with sz and dz being different, srcstep and dststep are being requested to react differently.

Example 3:

- VL=4
- mask=0b1101
- sz=0, dz=0

The following schedule for srcstep and dststep will occur:

srcstep	dststep	comment
0	0	both mask[src=0] and mask[dst=0] are 1
2	2	sz=0 and dz=0: both src and dst skip mask[1]
3	3	mask[src=3] and mask[dst=3] are 1
end	end	loop has ended because src and dst reached VL-1

Here, both srcstep and dststep remain in lockstep because sz=dz=0

A.1.5 Twin Predication

This is a novel concept that allows predication to be applied to a single source and a single dest register. The following types of traditional Vector operations may be encoded with it, *without requiring explicit opcodes to do so*

- VSPLAT (a single scalar distributed across a vector)
- VEXTRACT (like LLVM IR `extractelement`)
- VINSERT (like LLVM IR `insertelement`)
- VCOMPRESS (like LLVM IR `llvm.masked.compressstore.*`)
- VEXPAND (like LLVM IR `llvm.masked.expandload.*`)

Those patterns (and more) may be applied to:

- mv (the usual way that V* ISA operations are created)
- exts* sign-extension
- rwlinm and other RS-RA shift operations (**note**: excluding those that take RA as both a src and dest. These are not 1-src 1-dest, they are 2-src, 1-dest)
- LD and ST (treating AGEN as one source)
- FP fclass, fsgn, fneg, fabs, fcvt, frecip, fsqrt etc.
- Condition Register ops mfcr, mtcr and other similar

This is a huge list that creates extremely powerful combinations, particularly given that one of the predicate options is `(1<<r3)`

Additional unusual capabilities of Twin Predication include a back-to-back version of VCOMPRESS-VEXPAND which is effectively the ability to do sequentially ordered multiple VINSERTs. The source predicate selects a sequentially ordered subset of elements to be inserted; the destination predicate specifies the sequentially ordered recipient locations. This is equivalent to `llvm.masked.compressstore.*` followed by `llvm.masked.expandload.*` with a single instruction, but abstracted out from Load/Store and applicable in general to any 2P instruction.

This extreme power and flexibility comes down to the fact that SVP64 is not actually a Vector ISA: it is a loop-abstraction-concept that is applied *in general* to Scalar operations, just like the x86 `REP` instruction (if put on steroids).

A.1.6 Pack/Unpack

The pack/unpack concept of VSX `vpack` is abstracted out as Sub-Vector reordering. Two bits in the `SVSHAPE` [[sv/spr]] enable either "packing" or "unpacking" on the subvectors vec2/3/4.

First, illustrating a "normal" SVP64 operation with `SUBVL!=1:` (assuming no elwidth overrides), note that the VL loop is outer and the SUBVL loop inner:

```
def index():
    for i in range(VL):
        for j in range(SUBVL):
            yield i*SUBVL+j
```

```
for idx in index():
    operation_on(RA+idx)
```

For pack/unpack (again, no elwidth overrides), note that now there is the option to swap the SUBVL and VL loop orders. In effect the Pack/Unpack performs a Transpose of the subvector elements. Illustrated this time with a GPR mv operation:

```
# yield an outer-SUBVL or inner VL loop with SUBVL
def index_p(outer):
    if outer:
        for j in range(SUBVL):   # subvl is outer
            for i in range(VL):  # vl is inner
                yield i+VL*j
    else:
        for i in range(VL):        # vl is outer
            for j in range(SUBVL): # subvl is inner
                yield i*SUBVL+j

 # walk through both source and dest indices simultaneously
 for src_idx, dst_idx in zip(index_p(PACK), index_p(UNPACK)):
     move_operation(RT+dst_idx, RA+src_idx)
```

"yield" from python is used here for simplicity and clarity. The two Finite State Machines for the generation of the source and destination element offsets progress incrementally in lock-step.

Example VL=2, SUBVL=3, PACK_en=1 - elements grouped by vec3 will be redistributed such that Sub-elements 0 are packed together, Sub-elements 1 are packed together, as are Sub-elements 2.

```
srcstep=0   srcstep=1
0   1   2   3   4   5

dststep=0  dststep=1  dststep=2
0   3      1   4      2   5
```

Setting of both `PACK` and `UNPACK` is neither prohibited nor `UNDEFINED` because the reordering is fully deterministic, and additional REMAP reordering may be applied. Combined with Matrix REMAP this would give potentially up to 4 Dimensions of reordering.

Pack/Unpack has quirky interactions on {Swizzle Move} because it can set a different subvector length for destination, and has a slightly different pseudocode algorithm for Vertical-First Mode.

Ordering is as follows:

- SVSHAPE srcstep, dststep, ssubstep and dsubstep are advanced sequentially depending on PACK/UNPACK.
- srcstep and dststep are pushed through REMAP to compute actual Element offsets.
- Swizzle is independently applied to ssubstep and dsubstep

Pack/Unpack is enabled (set up) through {svstep instruction}.

A.1.7 Reduce modes

Reduction in SVP64 is deterministic and somewhat of a misnomer. A normal Vector ISA would have explicit Reduce opcodes with defined characteristics per operation: in SX Aurora there is even an additional scalar argument containing the initial reduction value, and the default is either 0 or 1 depending on the specifics of the explicit opcode. SVP64 fundamentally has to utilise *existing* Scalar Power ISA v3.0B operations, which presents some unique challenges.

The solution turns out to be to simply define reduction as permitting deterministic element-based schedules to be issued using the base Scalar operations, and to rely on the underlying microarchitecture to resolve Register

Hazards at the element level. This goes back to the fundamental principle that SV is nothing more than a Sub-Program-Counter sitting between Decode and Issue phases.

For Scalar Reduction, Microarchitectures *may* take opportunities to parallelise the reduction but only if in doing so they preserve strict Program Order at the Element Level. Opportunities where this is possible include an `OR` operation or a MIN/MAX operation: it may be possible to parallelise the reduction, but for Floating Point it is not permitted due to different results being obtained if the reduction is not executed in strict Program-Sequential Order.

In essence it becomes the programmer's responsibility to leverage the pre-determined schedules to desired effect.

A.1.7.1 Scalar result reduction and iteration

Scalar Reduction per se does not exist, instead is implemented in SVP64 as a simple and natural relaxation of the usual restriction on the Vector Looping which would terminate if the destination was marked as a Scalar. Scalar Reduction by contrast *keeps issuing Vector Element Operations* even though the destination register is marked as scalar *and* the same register is used as a source register. Thus it is up to the programmer to be aware of this, observe some conventions, and thus end up achieving the desired outcome of scalar reduction.

It is also important to appreciate that there is no actual imposition or restriction on how this mode is utilised: there will therefore be several valuable uses (including Vector Iteration and "Reverse-Gear") and it is up to the programmer to make best use of the (strictly deterministic) capability provided.

In this mode, which is suited to operations involving carry or overflow, one register must be assigned, by convention by the programmer to be the "accumulator". Scalar reduction is thus categorised by:

- One of the sources is a Vector
- the destination is a scalar
- optionally but most usefully when one source scalar register is also the scalar destination (which may be informally termed by convention the "accumulator")
- That the source register type is the same as the destination register type identified as the "accumulator". Scalar reduction on `cmp`, `setb` or `isel` makes no sense for example because of the mixture between CRs and GPRs.

*Note that issuing instructions in Scalar reduce mode such as `setb` are neither `UNDEFINED` nor prohibited, despite them not making much sense at first glance. Scalar reduce is strictly defined behaviour, and the cost in hardware terms of prohibition of seemingly non-sensical operations is too great. Therefore it is permitted and required to be executed successfully. Implementors **MAY** choose to optimise such instructions in instances where their use results in "extraneous execution", i.e. where it is clear that the sequence of operations, comprising multiple overwrites to a scalar destination **without** cumulative, iterative, or reductive behaviour (no "accumulator"), may discard all but the last element operation. Identification of such is trivial to do for `setb` and `cmp`: the source register type is a completely different register file from the destination. Likewise Scalar reduction when the destination is a Vector is as if the Reduction Mode was not requested. However it would clearly be unacceptable to perform such optimisations on cache-inhibited LD/ST, so some considerable care needs to be taken.*

Typical applications include simple operations such as `ADD r3, r10.v, r3` where, clearly, r3 is being used to accumulate the addition of all elements of the vector starting at r10.

```
# add RT, RA,RB but when RT==RA
for i in range(VL):
     iregs[RA] += iregs[RB+i] # RT==RA
```

However, *unless* the operation is marked as "mapreduce" (`sv.add/mr`) SV ordinarily **terminates** at the first scalar operation. Only by marking the operation as "mapreduce" will it continue to issue multiple sub-looped (element) instructions in `Program Order`.

To perform the loop in reverse order, the `RG` (reverse gear) bit must be set. This may be useful in situations where the results may be different (floating-point) if executed in a different order. Given that there is no actual prohibition on Reduce Mode being applied when the destination is a Vector, the "Reverse Gear" bit turns out

to be a way to apply Iterative or Cumulative Vector operations in reverse. `sv.add/rg r3.v, r4.v, r4.v` for example will start at the opposite end of the Vector and push a cumulative series of overlapping add operations into the Execution units of the underlying hardware.

Other examples include shift-mask operations where a Vector of inserts into a single destination register is required (see {Bitmanip ops}, bmset), as a way to construct a value quickly from multiple arbitrary bit-ranges and bit-offsets. Using the same register as both the source and destination, with Vectors of different offsets masks and values to be inserted has multiple applications including Video, cryptography and JIT compilation.

```
# assume VL=4:
# * Vector of shift-offsets contained in RC (r12.v)
# * Vector of masks contained in RB (r8.v)
# * Vector of values to be masked-in in RA (r4.v)
# * Scalar destination RT (r0) to receive all mask-offset values
sv.bmset/mr r0, r4.v, r8.v, r12.v
```

Due to the Deterministic Scheduling, Subtract and Divide are still permitted to be executed in this mode, although from an algorithmic perspective it is strongly discouraged. It would be better to use addition followed by one final subtract, or in the case of divide, to get better accuracy, to perform a multiply cascade followed by a final divide.

Note that single-operand or three-operand scalar-dest reduce is perfectly well permitted: the programmer may still declare one register, used as both a Vector source and Scalar destination, to be utilised as the "accumulator". In the case of `sv.fmadds` and `sv.maddhw` etc this naturally fits well with the normal expected usage of these operations.

If an interrupt or exception occurs in the middle of the scalar mapreduce, the scalar destination register **MUST** be updated with the current (intermediate) result, because this is how `Program Order` is preserved (Vector Loops are to be considered to be just another way of issuing instructions in Program Order). In this way, after return from interrupt, the scalar mapreduce may continue where it left off. This provides "precise" exception behaviour.

Note that hardware is perfectly permitted to perform multi-issue parallel optimisation of the scalar reduce operation: it's just that as far as the user is concerned, all exceptions and interrupts **MUST** be precise.

A.1.8 Fail-on-first

Data-dependent fail-on-first has two distinct variants: one for LD/ST (see {Load/Store Mode}, the other for arithmetic operations (actually, CR-driven) {Arithmetic Mode} and CR operations {Condition Register Fields Mode}. Note in each case the assumption is that vector elements are required appear to be executed in sequential Program Order, element 0 being the first.

- LD/ST ffirst (not to be confused with *Data-Dependent* LD/ST ffirst) treats the first LD/ST in a vector (element 0) as an ordinary one. Exceptions occur "as normal" on the first element. However for elements 1 and above, if an exception would occur, then VL is **truncated** to the previous element.
- Data-driven (CR-driven) fail-on-first activates when Rc=1 or other CR-creating operation produces a result (including cmp). Similar to branch, an analysis of the CR is performed and if the test fails, the vector operation terminates and discards all element operations above the current one (and the current one if VLi is not set), and VL is truncated to either the *previous* element or the current one, depending on whether VLi (VL "inclusive") is set.

Thus the new VL comprises a contiguous vector of results, all of which pass the testing criteria (equal to zero, less than zero).

The CR-based data-driven fail-on-first is new and not found in ARM SVE or RVV. At the same time it is also "old" because it is a generalisation of the Z80 Block compare instructions, especially CPIR which is based on CP (compare) as the ultimate "element" (suffix) operation to which the repeat (prefix) is applied. It is extremely useful for reducing instruction count, however requires speculative execution involving modifications of VL to get high performance implementations. An additional mode (RC1=1) effectively turns what would otherwise be an

arithmetic operation into a type of `cmp`. The CR is stored (and the CR.eq bit tested against the `inv` field). If the CR.eq bit is equal to `inv` then the Vector is truncated and the loop ends. Note that when RC1=1 the result elements are never stored, only the CRs.

VLi is only available as an option when `Rc=0` (or for instructions which do not have Rc). When set, the current element is always also included in the count (the new length that VL will be set to). This may be useful in combination with "inv" to truncate the Vector to *exclude* elements that fail a test, or, in the case of implementations of strncpy, to include the terminating zero.

In CR-based data-driven fail-on-first there is only the option to select and test one bit of each CR (just as with branch BO). For more complex tests this may be insufficient. If that is the case, a vectorised crops (crand, cror) may be used, and ffirst applied to the crop instead of to the arithmetic vector.

One extremely important aspect of ffirst is:

- LDST ffirst may never set VL equal to zero. This because on the first element an exception must be raised "as normal".
- CR-based data-dependent ffirst on the other hand **can** set VL equal to zero. This is the only means in the entirety of SV that VL may be set to zero (with the exception of via the SV.STATE SPR). When VL is set zero due to the first element failing the CR bit-test, all subsequent vectorised operations are effectively `nops` which is *precisely the desired and intended behaviour*.

Another aspect is that for ffirst LD/STs, VL may be truncated arbitrarily to a nonzero value for any implementation-specific reason. For example: it is perfectly reasonable for implementations to alter VL when ffirst LD or ST operations are initiated on a nonaligned boundary, such that within a loop the subsequent iteration of that loop begins subsequent ffirst LD/ST operations on an aligned boundary. Likewise, to reduce workloads or balance resources.

CR-based data-dependent first on the other hand MUST not truncate VL arbitrarily to a length decided by the hardware: VL MUST only be truncated based explicitly on whether a test fails. This because it is a precise test on which algorithms will rely.

Note: there is no reverse-direction for Data-dependent Fail-First. REMAP will need to be activated to invert the ordering of element traversal.

A.1.8.1 Data-dependent fail-first on CR operations (crand etc)

Operations that actually produce or alter CR Field as a result do not also in turn have an Rc=1 mode. However it makes no sense to try to test the 4 bits of a CR Field for being equal or not equal to zero. Moreover, the result is already in the form that is desired: it is a CR field. Therefore, CR-based operations have their own SVP64 Mode, described in {Condition Register Fields Mode}

There are two primary different types of CR operations:

- Those which have a 3-bit operand field (referring to a CR Field)
- Those which have a 5-bit operand (referring to a bit within the whole 32-bit CR)

More details can be found in {Condition Register Fields Mode}.

A.1.9 CR Operations

CRs are slightly more involved than INT or FP registers due to the possibility for indexing individual bits (crops BA/BB/BT). Again however the access pattern needs to be understandable in relation to v3.0B / v3.1B numbering, with a clear linear relationship and mapping existing when SV is applied.

A.1.9.1 CR EXTRA mapping table and algorithm

Numbering relationships for CR fields are already complex due to being in BE format (*the relationship is not clearly explained in the v3.0B or v3.1 specification*). However with some care and consideration the exact same mapping used for INT and FP regfiles may be applied, just to the upper bits, as explained below. Firstly and most importantly a new notation `CR{field number}` is used to indicate access to a particular Condition Register Field (as opposed to the notation `CR[bit]` which accesses one bit of the 32 bit Power ISA v3.0B Condition Register).

`CR{n}` refers to `CR0` when `n=0` and consequently, for CR0-7, is defined, in v3.0B pseudocode, as:

```
CR{n} = CR[32+n*4:35+n*4]
```

For SVP64 the relationship for the sequential numbering of elements is to the CR **fields** within the CR Register, not to individual bits within the CR register.

The `CR{n}` notation is designed to give *linear sequential numbering* in the Vector domain on a straight sequential Vector Loop.

In OpenPOWER v3.0/1, BF/BT/BA/BB are all 5 bits. The top 3 bits (0:2) select one of the 8 CRs; the bottom 2 bits (3:4) select one of 4 bits *in* that CR (EQ/LT/GT/SO). The numbering was determined (after 4 months of analysis and research) to be as follows:

```
CR_index = (BA>>2)      # top 3 bits
bit_index = (BA & 0b11) # low 2 bits
CR_reg = CR{CR_index}     # get the CR
# finally get the bit from the CR.
CR_bit = (CR_reg & (1<<bit_index)) != 0
```

When it comes to applying SV, it is the *CR Field* number `CR_reg` to which SV EXTRA2/3 applies, **not** the `CR_bit` portion (bits 3-4):

```
if extra3_mode:
    spec = EXTRA3
else:
    spec = EXTRA2<<1 | 0b0
if spec[0]:
   # vector constructs "BA[0:2] spec[1:2] 00 BA[3:4]"
   return ((BA >> 2)<<6) | # hi 3 bits shifted up
          (spec[1:2]<<4) | # to make room for these
          (BA & 0b11)      # CR_bit on the end
else:
   # scalar constructs "00 spec[1:2] BA[0:4]"
   return (spec[1:2] << 5) | BA
```

Thus, for example, to access a given bit for a CR in SV mode, the v3.0B algorithm to determine CR_reg is modified to as follows:

```
CR_index = (BA>>2)      # top 3 bits
if spec[0]:
    # vector mode, 0-124 increments of 4
    CR_index = (CR_index<<4) | (spec[1:2] << 2)
else:
    # scalar mode, 0-32 increments of 1
    CR_index = (spec[1:2]<<3) | CR_index
# same as for v3.0/v3.1 from this point onwards
bit_index = (BA & 0b11) # low 2 bits
CR_reg = CR{CR_index}     # get the CR
# finally get the bit from the CR.
CR_bit = (CR_reg & (1<<bit_index)) != 0
```

Note here that the decoding pattern to determine CR_bit does not change.

Note: high-performance implementations may read/write Vectors of CRs in batches of aligned 32-bit chunks (CR0-7, CR7-15). This is to greatly simplify internal design. If instructions are issued where CR Vectors do not start on a 32-bit aligned boundary, performance may be affected.

A.1.9.2 CR fields as inputs/outputs of vector operations

CRs (or, the arithmetic operations associated with them) may be marked as Vectorised or Scalar. When Rc=1 in arithmetic operations that have no explicit EXTRA to cover the CR, the CR is Vectorised if the destination is Vectorised. Likewise if the destination is scalar then so is the CR.

When vectorized, the CR inputs/outputs are sequentially read/written to 4-bit CR fields. Vectorised Integer results, when Rc=1, will begin writing to CR8 (TBD evaluate) and increase sequentially from there. This is so that:

- implementations may rely on the Vector CRs being aligned to 8. This means that CRs may be read or written in aligned batches of 32 bits (8 CRs per batch), for high performance implementations.
- scalar Rc=1 operation (CR0, CR1) and callee-saved CRs (CR2-4) are not overwritten by vector Rc=1 operations except for very large VL
- CR-based predication, from CR32, is also not interfered with (except by large VL).

However when the SV result (destination) is marked as a scalar by the EXTRA field the *standard* v3.0B behaviour applies: the accompanying CR when Rc=1 is written to. This is CR0 for integer operations and CR1 for FP operations.

Note that yes, the CR Fields are genuinely Vectorised. Unlike in SIMD VSX which has a single CR (CR6) for a given SIMD result, SV Vectorised OpenPOWER v3.0B scalar operations produce a **tuple** of element results: the result of the operation as one part of that element *and a corresponding CR element*. Greatly simplified pseudocode:

```
for i in range(VL):
     # calculate the vector result of an add
     iregs[RT+i] = iregs[RA+i] + iregs[RB+i]
     # now calculate CR bits
     CRs{8+i}.eq = iregs[RT+i] == 0
     CRs{8+i}.gt = iregs[RT+i] > 0
     ... etc
```

If a "cumulated" CR based analysis of results is desired (a la VSX CR6) then a followup instruction must be performed, setting "reduce" mode on the Vector of CRs, using cr ops (crand, crnor) to do so. This provides far more flexibility in analysing vectors than standard Vector ISAs. Normal Vector ISAs are typically restricted to "were all results nonzero" and "were some results nonzero". The application of mapreduce to Vectorised cr operations allows far more sophisticated analysis, particularly in conjunction with the new crweird operations see {CR Weird ops}.

Note in particular that the use of a separate instruction in this way ensures that high performance multi-issue OoO inplementations do not have the computation of the cumulative analysis CR as a bottleneck and hindrance, regardless of the length of VL.

Additionally, SVP64 {Branch Mode} may be used, even when the branch itself is to the following instruction. The combined side-effects of CTR reduction and VL truncation provide several benefits.

(see [[discussion]]. some alternative schemes are described there)

A.1.9.3 Rc=1 when SUBVL!=1

sub-vectors are effectively a form of Packed SIMD (length 2 to 4). Only 1 bit of predicate is allocated per subvector; likewise only one CR is allocated per subvector.

This leaves a conundrum as to how to apply CR computation per subvector, when normally Rc=1 is exclusively applied to scalar elements. A solution is to perform a bitwise OR or AND of the subvector tests. Given that OE is ignored in SVP64, this field may (when available) be used to select OR or AND behavior.

A.1.9.3.1 Table of CR fields

CRn is the notation used by the OpenPower spec to refer to CR field #i, so FP instructions with Rc=1 write to CR1 (n=1).

CRs are not stored in SPRs: they are registers in their own right. Therefore context-switching the full set of CRs involves a Vectorised mfcr or mtcr, using VL=8 to do so. This is exactly as how scalar OpenPOWER context-switches CRs: it is just that there are now more of them.

The 64 SV CRs are arranged similarly to the way the 128 integer registers are arranged. TODO a python program that auto-generates a CSV file which can be included in a table, which is in a new page (so as not to overwhelm this one). [[svp64/cr_names]]

A.1.10 Register Profiles

Instructions are broken down by Register Profiles as listed in the following auto-generated page: {SVP64 Augmentation Table}. These tables, despite being auto-generated, are part of the Specification.

A.1.11 SV pseudocode illustration

A.1.11.1 Single-predicated Instruction

illustration of normal mode add operation: zeroing not included, elwidth overrides not included. if there is no predicate, it is set to all 1s

```
function op_add(rd, rs1, rs2) # add not VADD!
  int i, id=0, irs1=0, irs2=0;
  predval = get_pred_val(FALSE, rd);
  for (i = 0; i < VL; i++)
    STATE.srcoffs = i # save context
    if (predval & 1<<i) # predication uses intregs
       ireg[rd+id] <= ireg[rs1+irs1] + ireg[rs2+irs2];
    if (!int_vec[rd].isvec) break;
    if (rd.isvec)  { id += 1; }
    if (rs1.isvec) { irs1 += 1; }
    if (rs2.isvec) { irs2 += 1; }
    if (id == VL or irs1 == VL or irs2 == VL) {
      # end VL hardware loop
      STATE.srcoffs = 0; # reset
      return;
    }
```

This has several modes:

- RT.v = RA.v RB.v
- RT.v = RA.v RB.s (and RA.s RB.v)
- RT.v = RA.s RB.s
- RT.s = RA.v RB.v
- RT.s = RA.v RB.s (and RA.s RB.v)
- RT.s = RA.s RB.s

All of these may be predicated. Vector-Vector is straightfoward. When one of source is a Vector and the other a Scalar, it is clear that each element of the Vector source should be added to the Scalar source, each result placed into the Vector (or, if the destination is a scalar, only the first nonpredicated result).

The one that is not obvious is RT=vector but both RA/RB=scalar. Here this acts as a "splat scalar result", copying the same result into all nonpredicated result elements. If a fixed destination scalar was intended, then an all-Scalar operation should be used.

See https://bugs.libre-soc.org/show_bug.cgi?id=552

A.1.12 Assembly Annotation

Assembly code annotation is required for SV to be able to successfully mark instructions as "prefixed".

A reasonable (prototype) starting point:

```
svp64 [field=value]*
```

Fields:

- ew=8/16/32 - element width
- sew=8/16/32 - source element width
- vec=2/3/4 - SUBVL
- mode=mr/satu/sats/crpred
- pred=1<<3/r3/$_{r3/r10/}$r10/r30/~r30/lt/gt/le/ge/eq/ne

similar to x86 "rex" prefix.

For actual assembler:

```
sv.asmcode/mode.vec{N}.ew=8,sw=16,m={pred},sm={pred} reg.v, src.s
```

Qualifiers:

- m={pred}: predicate mask mode
- sm={pred}: source-predicate mask mode (only allowed in Twin-predication)
- vec{N}: vec2 OR vec3 OR vec4 - sets SUBVL=2/3/4
- ew={N}: ew=8/16/32 - sets elwidth override
- sw={N}: sw=8/16/32 - sets source elwidth override
- ff={xx}: see fail-first mode
- sat{x}: satu / sats - see saturation mode
- mr: see map-reduce mode
- mrr: map-reduce, reverse-gear (VL-1 downto 0)
- mr.svm see map-reduce with sub-vector mode
- crm: see map-reduce CR mode
- crm.svm see map-reduce CR with sub-vector mode
- sz: predication with source-zeroing
- dz: predication with dest-zeroing

For modes:

- fail-first
- ff=lt/gt/le/ge/eq/ne/so/ns
- RC1 mode
- saturation:
- sats
- satu
- map-reduce:
- mr OR crm: "normal" map-reduce mode or CR-mode.
- mr.svm OR crm.svm: when vec2/3/4 set, sub-vector mapreduce is enabled

A.1.13 Parallel-reduction algorithm

The principle of SVP64 is that SVP64 is a fully-independent Abstraction of hardware-looping in between issue and execute phases that has no relation to the operation it issues. Additional state cannot be saved on context-switching beyond that of SVSTATE, making things slightly tricky.

Executable demo pseudocode, full version here

```
def preduce_yield(vl, vec, pred):
    step = 1
    ix = list(range(vl))
    while step < vl:
        step *= 2
        for i in range(0, vl, step):
            other = i + step // 2
            ci = ix[i]
            oi = ix[other] if other < vl else None
            other_pred = other < vl and pred[oi]
            if pred[ci] and other_pred:
                yield ci, oi
            elif other_pred:
                ix[i] = oi

def preduce_y(vl, vec, pred):
   for i, other in preduce_yield(vl, vec, pred):
       vec[i] += vec[other]
```

This algorithm works by noting when data remains in-place rather than being reduced, and referring to that alternative position on subsequent layers of reduction. It is re-entrant. If however interrupted and restored, some implementations may take longer to re-establish the context.

Its application by default is that:

- RA, FRA or BFA is the first register as the first operand (ci index offset in the above pseudocode)
- RB, FRB or BFB is the second (co index offset)
- RT (result) also uses ci **if RA==RT**

For more complex applications a REMAP Schedule must be used

Programmers's note: if passed a predicate mask with only one bit set, this algorithm takes no action, similar to when a predicate mask is all zero.

Implementor's Note: many SIMD-based Parallel Reduction Algorithms are implemented in hardware with MVs that ensure lane-crossing is minimised. The mistake which would be catastrophic to SVP64 to make is to then limit the Reduction Sequence for all implementors based solely and exclusively on what one specific internal microarchitecture does. In SIMD ISAs the internal SIMD Architectural design is exposed and imposed on the programmer. Cray-style Vector ISAs on the other hand provide convenient, compact and efficient encodings of abstract concepts. **It is the Implementor's responsibility to produce a design that complies with the above algorithm, utilising internal Micro-coding and other techniques to transparently insert micro-architectural lane-crossing Move operations if necessary or desired, to give the level of efficiency or performance required.**

A.1.14 Element-width overrides </>

Element-width overrides are best illustrated with a packed structure union in the c programming language. The following should be taken literally, and assume always a little-endian layout:

```
#pragma pack
typedef union {
```

```
    uint8_t  b[];
    uint16_t s[];
    uint32_t i[];
    uint64_t l[];
    uint8_t actual_bytes[8];
} el_reg_t;

elreg_t int_regfile[128];
```

Accessing (get and set) of registers given a value, register (in `elreg_t` form), and that all arithmetic, numbering and pseudo-Memory format is LE-endian and LSB0-numbered below:

```
elreg_t& get_polymorphed_reg(elreg_t const& reg, bitwidth, offset):
    el_reg_t res; // result
    res.l = 0; // TODO: going to need sign-extending / zero-extending
    if !reg.isvec: // scalar access has no element offset
        offset = 0
    if bitwidth == 8:
        reg.b = int_regfile[reg].b[offset]
    elif bitwidth == 16:
        reg.s = int_regfile[reg].s[offset]
    elif bitwidth == 32:
        reg.i = int_regfile[reg].i[offset]
    elif bitwidth == 64:
        reg.l = int_regfile[reg].l[offset]
    return reg

set_polymorphed_reg(elreg_t& reg, bitwidth, offset, val):
    if (!reg.isvec):
        # for safety mask out hi bits
        bytemask = (8 << bitwidth) - 1
        val &= bytemask
        # not a vector: first element only, overwrites high bits.
        # and with the *Architectural* definition being LE,
        # storing in the first DWORD works perfectly.
        int_regfile[reg].l[0] = val
    elif bitwidth == 8:
        int_regfile[reg].b[offset] = val
    elif bitwidth == 16:
        int_regfile[reg].s[offset] = val
    elif bitwidth == 32:
        int_regfile[reg].i[offset] = val
    elif bitwidth == 64:
        int_regfile[reg].l[offset] = val
```

In effect the GPR registers r0 to r127 (and corresponding FPRs fp0 to fp127) are reinterpreted to be "starting points" in a byte-addressable memory. Vectors - which become just a virtual naming construct - effectively overlap.

It is extremely important for implementors to note that the only circumstance where upper portions of an underlying 64-bit register are zero'd out is when the destination is a scalar. The ideal register file has byte-level write-enable lines, just like most SRAMs, in order to avoid READ-MODIFY-WRITE.

An example ADD operation with predication and element width overrides:

```
  for (i = 0; i < VL; i++)
    if (predval & 1<<i) # predication
       src1 = get_polymorphed_reg(RA, srcwid, irs1)
```

```
            src2 = get_polymorphed_reg(RB, srcwid, irs2)
            result = src1 + src2 # actual add here
            set_polymorphed_reg(RT, destwid, ird, result)
            if (!RT.isvec) break
         if (RT.isvec)  { id += 1; }
         if (RA.isvec)  { irs1 += 1; }
         if (RB.isvec)  { irs2 += 1; }
```

Thus it can be clearly seen that elements are packed by their element width, and the packing starts from the source (or destination) specified by the instruction.

A.1.15 Twin (implicit) result operations

Some operations in the Power ISA already target two 64-bit scalar registers: `lq` for example, and LD with update. Some mathematical algorithms are more efficient when there are two outputs rather than one, providing feedback loops between elements (the most well-known being add with carry). 64-bit multiply for example actually internally produces a 128 bit result, which clearly cannot be stored in a single 64 bit register. Some ISAs recommend "macro op fusion": the practice of setting a convention whereby if two commonly used instructions (mullo, mulhi) use the same ALU but one selects the low part of an identical operation and the other selects the high part, then optimised micro-architectures may "fuse" those two instructions together, using Micro-coding techniques, internally.

The practice and convention of macro-op fusion however is not compatible with SVP64 Horizontal-First, because Horizontal Mode may only be applied to a single instruction at a time, and SVP64 is based on the principle of strict Program Order even at the element level. Thus it becomes necessary to add explicit more complex single instructions with more operands than would normally be seen in the average RISC ISA (3-in, 2-out, in some cases). If it was not for Power ISA already having LD/ST with update as well as Condition Codes and `lq` this would be hard to justify.

With limited space in the `EXTRA` Field, and Power ISA opcodes being only 32 bit, 5 operands is quite an ask. `lq` however sets a precedent: `RTp` stands for "RT pair". In other words the result is stored in RT and RT+1. For Scalar operations, following this precedent is perfectly reasonable. In Scalar mode, `maddedu` therefore stores the two halves of the 128-bit multiply into RT and RT+1.

What, then, of `sv.maddedu`? If the destination is hard-coded to RT and RT+1 the instruction is not useful when Vectorised because the output will be overwritten on the next element. To solve this is easy: define the destination registers as RT and RT+MAXVL respectively. This makes it easy for compilers to statically allocate registers even when VL changes dynamically.

Bear in mind that both RT and RT+MAXVL are starting points for Vectors, and bear in mind that element-width overrides still have to be taken into consideration, the starting point for the implicit destination is best illustrated in pseudocode:

```
# demo of maddedu
for (i = 0; i < VL; i++)
   if (predval & 1<<i) # predication
      src1 = get_polymorphed_reg(RA, srcwid, irs1)
      src2 = get_polymorphed_reg(RB, srcwid, irs2)
      src2 = get_polymorphed_reg(RC, srcwid, irs3)
      result = src1*src2 + src2
      destmask = (2<<destwid)-1
      # store two halves of result, both start from RT.
      set_polymorphed_reg(RT, destwid, ird      , result&destmask)
      set_polymorphed_reg(RT, destwid, ird+MAXVL, result>>destwid)
      if (!RT.isvec) break
   if (RT.isvec)  { id += 1; }
   if (RA.isvec)  { irs1 += 1; }
```

```
if (RB.isvec)  { irs2 += 1; }
if (RC.isvec)  { irs3 += 1; }
```

The significant part here is that the second half is stored starting not from RT+MAXVL at all: it is the *element* index that is offset by MAXVL, both halves actually starting from RT. If VL is 3, MAXVL is 5, RT is 1, and dest elwidth is 32 then the elements RT0 to RT2 are stored:

LSB0:	63:32	31:0
MSB0:	0:31	32:63
r0	unchanged	unchanged
r1	RT1.lo	RT0.lo
r2	unchanged	RT2.lo
r3	RT0.hi	unchanged
r4	RT2.hi	RT1.hi
r5	unchanged	unchanged

Note that all of the LO halves start from r1, but that the HI halves start from half-way into r3. The reason is that with MAXVL bring 5 and elwidth being 32, this is the 5th element offset (in 32 bit quantities) counting from r1.

Programmer's note: accessing registers that have been placed starting on a non-contiguous boundary (half-way along a scalar register) can be inconvenient: REMAP can provide an offset but it requires extra instructions to set up. A simple solution is to ensure that MAXVL is rounded up such that the Vector ends cleanly on a contiguous register boundary. MAXVL=6 in the above example would achieve that

Additional DRAFT Scalar instructions in 3-in 2-out form with an implicit 2nd destination:

- {Fixed Point pseudocode}
- {Floating Point pseudocode}

[[!tag standards]]

Appendix B

SVP64 Quirks

B.1 The Rules

[[!toc]]

SVP64 is designed around fundamental and inviolate RISC principles. This gives a uniformity and regularity to the ISA, making implementation straightforward, which was why RISC as a concept became popular.

1. There are no actual Vector instructions: Scalar instructions are the sole exclusive bedrock.
2. No scalar instruction ever deviates in its encoding or meaning just because it is prefixed (semantic caveats below)
3. A hardware-level for-loop (the prefix) makes vector elements 100% synonymous with scalar instructions (the suffix)
4. Exactly as with Scalar RISC ISAs, the uniformity does produce "holes" in the encoding or some strange combinations.

How can a Vector ISA even exist when no actual Vector instructions are permitted to be added? It comes down to the strict RISC abstraction. First you start from a **scalar** instruction (32-bit). Second, the Prefixing is applied *in the abstract* to give the *appearance* and ultimately the same effect as if an explicit Vector instruction had also been added. Looking at the pseudocode of any Vector ISA (RVV, NEC SX Aurora, Cray) they always comprise (a) a for-loop around (b) element-based operations. It is perfectly reasonable and rational to separate (a) from (b) then find a powerful pre-existing Supercomputing-class ISA that qualifies for (b).

There are a few exceptional places where these rules get bent, and others where the rules take some explaining, and this page tracks them all.

The modification caveat in (2) above semantically exempts element width overrides, which still do not actually modify the meaning of the instruction: an add remains an add, even if its override makes it an 8-bit add rather than a 64-bit add. Even add-with-carry remains an add-with-carry: it's just that when elwidth=8 in the Prefix it's an *8-bit* add-with-carry where the 9th bit becomes Carry-out (not the 65th bit). In other words, elwidth overrides **definitely** do not fundamentally alter the actual Scalar v3.0 ISA encoding itself. Consequently we can still, in the strictest semantic sense, not be breaking rule (2).

Likewise, other "modifications" such as saturation or Data-dependent Fail-First likewise are actually post-augmentation or post-analysis, and do not fundamentally change an add operation into a subtract for example, and under absolutely no circumstances do the actual 32-bit Scalar v3.0 operand field bits change or the number of operands change.

In an early Draft of SVP64, an experiment was attempted, to modify LD-immediate instructions to include a third RC register i.e. reinterpret the normal v3.0 32-bit instruction as a completely different encoding if SVP64-prefixed. It did not go well. The complexity that resulted in the decode phase was too great. The lesson was learned, the hard way: it would be infinitely preferable to add a 32-bit Scalar Load-with-Shift instruction

first, which then inherently becomes Vectorised. Perhaps a future Power ISA spec will have this Load-with-Shift instruction: both ARM and x86 have it, because it saves greatly on instruction count in hot-loops.

The other reason for not adding an SVP64-Prefixed instruction without also having it as a Scalar un-prefixed instruction is that if the 32-bit encoding is ever allocated in a future revision of the Power ISA to a completely unrelated operation then how can a Vectorised version of that new instruction ever be added? The uniformity and RISC Abstraction is irreparably damaged. Bottom line here is that the fundamental RISC Principle is strictly adhered to, even though these are Advanced 64-bit Vector instructions. Advocates of the RISC Principle will appreciate the uniformity of SVP64 and the level of systematic abstraction kept between Prefix and Suffix.

B.2 Instruction Groups

The basic principle of SVP64 is the prefix, which contains mode as well as register augmentation and predicates. When thinking of instructions and Vectorising them, it is natural for arithmetic operations (ADD, OR) to be the first to spring to mind. Arithmetic instructions have registers, therefore augmentation applies, end of story, right?

Except, Load and Store deals also with Memory, not just registers. Power ISA has Condition Register Fields: how can element widths apply there? And branches: how can you have Saturation on something that does not return an arithmetic result? In short: there are actually four different categories (five including those for which Vectorisation makes no sense at all, such as `sc` or `mtmsr`). The categories are:

- arithmetic/logical including floating-point
- Load/Store
- Condition Register Field operations
- branch

Arithmetic

Arithmetic (known as "normal" mode) is where Scalar and Parallel Reduction can be done: Saturation as well, and a new innovative modes for Vector ISAs: data-dependent fail-first. Reduction and Saturation are common to see in Vector ISAs: it is just that they are usually added as explicit instructions, and NEC SX Aurora has even more iterative instructions. In SVP64 these concepts are applied in the abstract general form, which takes some getting used to.

Reduction may, when applied to non-commutative instructions incorrectly, result in invalid results, but ultimately it is critical to think in terms of the "rules", that everything is Scalar instructions in strict Program Order. Reduction on non-commutative Scalar Operations is not *prohibited*: the strict Program Order allows the programmer to think through what would happen and thus potentially actually come up with legitimate use.

Branches

Branch is the one and only place where the Scalar (non-prefixed) operations differ from the Vector (element) instructions (as explained in a separate section) although a case could be made for the perspective that they are identical, but the defaults for new parameters in the Scalar case makes branch identical to Power ISA v3.1 Scalar branches.

The RM bits can be used for other purposes because the Arithmetic modes make no sense at all for a Branch. Almost the entire SVP64 RM Field is interpreted differently from other Modes, in order to support a wide range of parallel boolean condition options which are expected of a Vector / GPU ISA. These save a considerable number of instructions in tight inner loop situations.

CR Field Ops

Condition Register Fields are 4-bit wide and consequently element-width overrides make absolutely no sense whatsoever. Therefore the elwidth override field bits can be used for other purposes when Vectorising CR Field instructions. Moreover, Rc=1 is completely invalid for CR operations such as `crand`: Rc=1 is for arithmetic operations, producing a "co-result" that goes into CR0 or CR1. Thus, Saturation makes no sense. All of these

differences, which require quite a lot of logical reasoning and deduction, help explain why there is an entirely different CR ops Vectorisation Category.

A particularly strange quirk of CR-based Vector Operations is that the Scalar Power ISA CR Register is 32-bits, but actually comprises eight CR Fields, CR0-CR7. With each CR Field being four bits (EQ, LT, GT, SO) this makes up 32 bits, and therefore a CR operand referring to one bit of the CR will be 5 bits in length (BA, BT). *However*, some instructions refer to a *CR Field* (CR0-CR7) and consequently these operands (BF, BFA etc) are only 3-bits.

(*It helps here to think of the top 3 bits of BA as referring to a CR Field, like BFA does, and the bottom 2 bits of BA referring to EQ/LT/GT/SO within that Field*)

With SVP64 extending the number of CR *Fields* to 128, the number of 32-bit CR *Registers* extends to 16, in order to hold all 128 CR *Fields* (8 per CR Register). Then, it gets even more strange, when it comes to Vectorisation, which applies to the CR Field *numbers*. The hardware-for-loop for Rc=1 for example starts at CR0 for element 0, and moves to CR1 for element 1, and so on. The reason here is quite simple: each element result has to have its own CR Field co-result.

In other words, the element is the 4-bit CR *Field*, not the bits *of* the 32-bit CR Register, and not the CR *Register* (of which there are now 16). All quite logical, but a little mind-bending.

Load/Store

LOAD/STORE is another area that has different needs: this time it is down to limitations in Scalar LD/ST. Vector ISAs have Load/Store modes which simply make no sense in a RISC Scalar ISA: element-stride and unit-stride and the entire concept of a stride itself (a spacing between elements) has no place at all in a Scalar ISA. The problems come when trying to *retrofit* the concept of "Vector Elements" onto a Scalar ISA. Consequently it required a couple of bits (Modes) in the SVP64 RM Prefix to convey the stride mode, changing the Effective Address computation as a result. Interestingly, worth noting for Hardware designers: it did turn out to be possible to perform pre-multiplication of the D/DS Immediate by the stride amount, making it possible to avoid actually modifying the LD/ST Pipeline itself.

Other areas where LD/ST went quirky: element-width overrides especially when combined with Saturation, given that LD/ST operations have byte, halfword, word, dword and quad variants. The interaction between these widths as part of the actual operation, and the source and destination elwidth overrides, was particularly obtuse and hard to derive: some care and attention is advised, here, when reading the specification, especially on arithmetic loads (lbarx, lharx etc.)

Non-vectorised

The concept of a Vectorised halt (`attn`) makes no sense. There are never going to be a Vector of global MSRs (Machine Status Register). `mtcr` on the other hand is a grey area: `mtspr` is clearly Vectoriseable. Even `td` and `tdi` makes a strange type of sense to permit it to be Vectorised, because a sequence of comparisons could be Vectorised. Vectorised System Calls (`sc`) or `tlbie` and other Cache or Virtual Nemory Management instructions, these make no sense to Vectorise.

However, it is really quite important to not be tempted to conclude that just because these instructions are un-vectoriseable, the Prefix opcode space must be free for reiterpretation and use for other purposes. This would be a serious mistake because a future revision of the specification might *retire* the Scalar instruction, and, worse, replace it with another. Again this comes down to being quite strict about the rules: only Scalar instructions get Vectorised: there are *no* actual explicit Vector instructions.

Summary

Where a traditional Vector ISA effectively duplicates the entirety of a Scalar ISA and then adds additional instructions which only make sense in a Vector Context, such as Vector Shuffle, SVP64 goes to considerable lengths to keep strictly to augmentation and embedding of an entire Scalar ISA's instructions into an abstract Vectorisation Context. That abstraction subdivides down into Categories appropriate for the type of operation (Branch, CRs, Memory, Arithmetic), and each Category has its own relevant but ultimately rational quirks.

B.3 Abstraction between Prefix and Suffix

In the introduction paragraph, a great fuss was made emphasising that the Prefix is kept separate from the Suffix. The whole idea there is that a Multi-issue Decoder and subsequent pipelines would in no way have "back-propagation" of state that can only be determined far too late. This *has* been preserved, however there is a hiccup.

Examining the Power ISA 3.1 a 64-bit Prefix was introduced, EXT001. The encoding of the prefix has 6 bits that are dedicated to letting the hardware know what the remainder of the Prefix bits mean: how they are formatted, even without having to examine the Suffix to which they are applied.

SVP64 has such pressure on its 24-bit encoding that it was simply not possible to perform the same trick used by Power ISA 3.1 Prefixing. Therefore, rather unfortunately, it becomes necessary to perform a *partial decoding* of the v3.0 Suffix before the 24-bit SVP64 RM Fields may be identified. Fortunately this is straightforward, and does not rely on any outside state, and even more fortunately for a Multi-Issue Execution decoder, the length 32/64 is also easy to identify by looking for the EXT001 pattern. Once identified the 32/64 bits may be passed independently to multiple Decoders in parallel.

B.4 Predication

Predication is entirely missing from the Power ISA. Adding it would be a costly mistake because it cannot be retrofitted to an ISA without literally duplicating all instructions. Prefixing is about the only sane way to go.

CR Fields as predicate masks could be spread across multiple register file entries, making them costly to read in one hit. Therefore the possibility exists that an instruction element writing to a CR Field could *overwrite* the Predicate mask CR Vector during the middle of a for-loop.

Clearly this is bad, so don't do it. If there are potential issues they can be avoided by using the crweird instructions to get CR Field bits into an Integer GPR (r3, r10 or r30) and use that GPR as a Predicate mask instead.

Even in Vertical-First Mode, which is a single Scalar instruction executed with "offset" registers (in effect), the rule still applies: don't write to the same register being used as the predicate, it's `UNDEFINED` behaviour.

B.4.1 Single Predication

So named because there is a Twin Predication concept as well, Single Predication is also unlike other Vector ISAs because it allows zeroing on both the source and destination. This takes some explaining.

In Vector ISAs, there is a Predicate Mask, it applies to the destination only, and there is a choice of actions when a Predicate Mask bit is zero:

- set the destination element to zero
- skip that element operation entirely, leaving the destination unmodified

The problem comes if the underlying register file SRAM is say 64-bit wide write granularity but the Vector elements are say 8-bit wide. Some Vector ISAs strongly advocate Zeroing because to leave one single element at a small bitwidth in amongst other elements where the register file does not have the prerequisite access granularity is very expensive, requiring a Read-Modify-Write cycle to preserve the untouched elements. Putting zero into the destination avoids that Read.

This is technically very easy to solve: use a Register File that does in fact have the smallest element-level write-enable granularity. If the elements are 8 bit then allow 8-bit writes!

With that technical issue solved there is nothing in the way of choosing to support both zeroing and non-zeroing (skipping) at the ISA level: SV chooses to further support both *on both the source and destination*. This can

result in the source and destination element indices getting "out-of-sync" even though the Predicate Mask is the same because the behaviour is different when zeros in the Predicate are encountered.

B.4.2 Twin Predication

Twin Predication is an entirely new concept not present in any commercial Vector ISA of the past forty years. To explain how normal Single-predication is applied in a standard Vector ISA:

- Predication on the **source** of a LOAD instruction creates something called "Vector Compressed Load" (VCOMPRESS).
- Predication on the **destination** of a STORE instruction creates something called "Vector Expanded Store" (VEXPAND).
- SVP64 allows the two to be put back-to-back: one on source, one on destination.

The above allows a reader familiar with VCOMPRESS and VEXPAND to conceptualise what the effect of Twin Predication is, but it actually goes much further: in *any* twin-predicated instruction (extsw, fmv) it is possible to apply one predicate to the source register (compressing the source element array) and another *completely separate* predicate to the destination register, not just on Load/Stores but on *arithmetic* operations.

No other Vector ISA in the world has this back-to-back capability. All true Vector ISAs have Predicate Masks: it is an absolutely essential characteristic. However none of them have abstracted dual predicates out to the extent where this VCOMPRESS-VEXPAND effect is applicable *in general* to a wide range of arithmetic instructions, as well as Load/Store.

It is however important to note that not all instructions can be Twin Predicated (2P): some remain only Single Predicated (1P), as is normally found in other Vector ISAs. Arithmetic operations with four registers (3-in, 1-out, VA-Form for example) are Single. The reason is that there just wasn't enough space in the 24-bits of the SVP64 Prefix. Consequently, when using a given instruction, it is necessary to look up in the ISA Tables whether it is 1P or 2P. caveat emptor!

Also worth a special mention: all Load/Store operations are Twin-Predicated. The underlying key to understanding:

- one Predicate effectively applies to the Array of Memory *Addresses*,
- the other Predicate effectively applies to the Array of Memory *Data*.

B.5 CR weird instructions

{CR Weird ops} is by far the biggest violator of the SVP64 rules, for good reasons. Transfers between Vectors of CR Fields and Integers for use as predicates is very awkward without them.

Normally, element width overrides allow the element width to be specified as 8, 16, 32 or default (64) bit. With CR weird instructions producing or consuming either 1 bit or 4 bit elements (in effect) some adaptation was required. When this perspective is taken (that results or sources are 1 or 4 bits) the weirdness starts to make sense, because the "elements", such as they are, are still packed sequentially.

From a hardware implementation perspective however they will need special handling as far as Hazard Dependencies are concerned, due to nonconformance (bit-level management)

B.6 mv.x (vector permute)

[[sv/mv.x]] aka `GPR(RT) = GPR(GPR(RA))` is so horrendous in terms of Register Hazard Management that its addition to any Scalar ISA is anathematic. In a Traditional Vector ISA however, where the indices are isolated behind a single Vector Hazard, there is no problem at all. `sv.mv.x` is also fraught, precisely because it sits on top of a Standard Scalar register paradigm, not a Vector ISA with separate and distinct Vector registers.

To help partly solve this, `sv.mv.x` would have had to have been made relative:

```
for i in range(VL):
    GPR(RT+i) = GPR(RT+MIN(GPR(RA+i), VL))
```

The reason for doing so is that MAXVL or VL may be used to limit the number of Register Hazards that need to be raised to a fixed quantity, at Issue time.

`mv.x` itself would still have to be added as a Scalar instruction, but the behaviour of `sv.mv.x` would have to be different from that Scalar version.

Normally, Scalar Instructions have a good justification for being added as Scalar instructions on their own merit. `mv.x` is the polar opposite, and in the end, the idea was thrown out, and Indexed REMAP added in its place. Indexed REMAP comes with its own quirks, solving the Hazard problem, described in a later section.

B.7 REMAP and other reordering

There are several places in Simple-V which apply some sort of reordering schedule to elements. srcstep and dststep do not themselves reorder: they continue to march in sequence (VL-1 downto 0 in the case of reverse-gear)

It is perfectly legal to apply Parallel-Reduction on top of any type of REMAP, for example, and it is possible to apply Pack/Unpack on a REMAP as well.

The order of application of REMAP combined with Parallel-Reduction should be logically obvious: REMAP has to come first because otherwise how can the Parallel-Reduction perform a tree-walk?

Pack/Unpack on the other hand is best implemented as applying first, because it is applied as the inversion of the for-loops which generate the steps and substeps. REMAP then applies to the src/dst-step indices (never to the subvl step indices: that is SWIZZLE's job).

It's all perfectly logical, just a lot going on.

B.8 Branch-Conditional

{Branch Mode} are a very special exception to the rule that there shall be no deviation from the corresponding Scalar instruction. This because of the tight integration with looping and the application of Boolean Logic manipulation needed for Parallel operations (predicate mask usage). This results in an extremely important observation that `scalar identity behaviour` is violated: the SV Prefixed variant of branch is **not** the same operation as the unprefixed 32-bit scalar version.

One key difference is that LR is only updated if certain additional conditions are met, whereas Scalar `bclrl` for example unconditionally overwrites LR.

Another is that the Vectorised Branch-Conditional instructions are the only ones where there are side-effects on predication when skipping is enabled. This is so as to be able to use CTR to count down *masked-out* elements.

Well over 500 Vectorised branch instructions exist in SVP64 due to the number of options available: close integration and interaction with the base Scalar Branch was unavoidable in order to create Conditional Branching suitable for parallel 3D / CUDA GPU workloads.

B.9 Saturation

The application of Saturation as a retro-fit to a Scalar ISA is challenging. It does help that within the SFFS Compliancy subset there are no Saturated operations at all: they are only added in VSX.

Saturation does not inherently change the instruction itself: it does however come with some fundamental implications, when applied. For example: a Floating-Point operation that would normally raise an exception will no longer do so, instead setting the CR1.SO Flag. Another quirky example: signed operations which produce a negative result will be truncated to zero if Unsigned Saturation is requested.

One very important aspect for implementors is that the operation in effect has to be considered to be performed at infinite precision, followed by saturation detection. In practice this does not actually require infinite precision hardware! Two 8-bit integers being added can only ever overflow into a 9-bit result.

Overall some care and consideration needs to be applied.

B.10 Fail-First

Fail-First (both the Load/Store and Data-Dependent variants) is worthy of a special mention in its own right. Where VL is normally forward-looking and may be part of a pre-decode phase in a (simplified) pipelined architecture with no Read-after-Write Hazards, Fail-First changes that because at any point during the execution of the element-level instructions, one of those elements may not only terminate further continuation of the hardware-for-looping but also effect a change of VL:

```
for i in range(VL):
    result = element_operation(GPR(RA+i), GPR(RB+i))
    if test(result):
        VL = i
        break
```

This is not exactly a violation of SVP64 Rules, more of a breakage of user expectations, particularly for LD/ST where exceptions would normally be expected to be raised, Fail-First provides for avoidance of those exceptions.

For Hardware implementers, a standard Out-of-Order micro-architecture allows for Cancellation of speculatively-executed elements that extended beyond the Vector Truncation point. In-order systems will have a slightly harder time and may choose to execute one element only at a time, reducing performance as a result.

B.11 OE=1

The hardware cost of Sticky Overflow in a parallel environment is immense. The SFFS Compliancy Level is permitted optionally to support XER.SO. Therefore the decision is made to make it mandatory **not** to support XER.SO. However, CR.SO *is* supported such that when Rc=1 is set the CR.SO flag will contain only the overflow of the current instruction, rather than being actually "sticky". Hardware Out-of-Order designers will recognise and appreciate that the Hazards are reduced to Read-After-Write (RAW) and that the WAR Hazard is removed.

This is sort-of a quirk and sort-of not, because the option to support XER.SO is already optional from the SFFS Compliancy Level.

B.12 Indexed REMAP and CR Field Predication Hazards

Normal Vector ISAs and those Packed SIMD ISAs inspired by them have Vector "Permute" or "Shuffle" instructions. These provide a Vector of indices whereby another Vector is reordered (permuted, shuffled) according to the indices. Register Hazard Managent here is trivial because there are three registers: indices source vector, elements source vector to be shuffled, result vector.

For SVP64 which is based on top of a Scalar Register File paradigm, combined with the hard requirement to respect full Register Hazard Management as if element instructions were actual Scalar instructions, the addition of a Vector permute instruction under these strict conditions would result in a catastrophic reduction in performance, due to having to consider Read-after-Write and Write-after-Read Hazards *at the element level*.

A little leniency and rule-bending is therefore required.

Rather than add explicit Vector permute instructions, the "Indexing" has been separated out into a REMAP Schedule. When an Indexed REMAP is requested, it is assumed (required, of software) that subsequent instructions intending to use those indices *will not* attempt to modify the indices. It is *Software* that must consider them to be read-only.

This simple relaxation of the rules releases Hardware from having the horrendous job of dynamically detecting Write-after-Read Hazards on a huge range of registers.

A similar Hazard problem exists for CR Field Predicates, in Vertical-First Mode. Instructions could modify CR Fields currently being used as Predicate Masks: detecting this is so horrendous for hardware resource utilisation and hardware complexity that, again, the decision is made to relax these constraints and for Software to take that into account.

B.13 Floating-Point "Single" becomes "Half"

In several places in the Power ISA there are operations that are on 32-bit quantities in 64-bit registers. The best example is FP which has 64-bit operations (`fadd`) and 32-bit operations (`fadds` or FP Add "single"). Element-width overrides it would seem to be unnecessary, under these circumstances.

However, it is not possible for `fadds` to fit two elements into 64-bit: that breaks the simplicity of SVP64. Bear in mind that the FP32 bits are spread out across a 64 bit register in FP64 format. The solution here was to consider the "s" at the end of each instruction to mean "half of the element's width". Thus, `sv.fadds/ew=32` actually stores an FP16 spread out across the 32 bits of an element, in FP32 format, where `sv.fadd/ew=32` stores a full FP32 result into the full 32 bits.

Where this breaks down is when attempting to do half-width on BF16 or FP16 operations: there does not exist a BF8 or an IEEE754 FP8 format, so these (`sv.fadds/ew=8`) should be avoided.

B.14 Word frequently becomes "half"

Again, related to "Single" becoming "half of element width", unless there are compelling reasons the same trick applies to Scalar GPR operations. With the pseudocode being "XLEN//2" then of course if XLEN=8 the operation becomes a 4-bit one.

Similarly byte operations which use "XLEN//8" when XLEN=8 actually become single-bit operations, which is very useful with `sv.extsb/w=8` for example. This instruction copies the LSB of each byte in a sequence of bytes, and expands it to all 8 bits in each result byte.

B.15 Vertical-First and Subvectors

Documented in the {setvl instruction} page, Vertical-First goes through elements second instructions first and requires an explicit {svstep instruction} instruction to move to the next element, (whereas Horizontal-First loops through elements in full first before moving on to the next instruction): *Subvectors are considered "elements"* in Vertical-First Mode.

This is conceptually quite easy to keep in mind that a Vertical-First instruction does one element at a time, and when SUBVL is set, that "element" in essence becomes a vec2/3/4.

B.16 Swizzle and Pack/Unpack

These are both so weird it's best to just read the pages in full and pay attention: {Swizzle Move} and {Pack / Unpack}. Swizzle Moves only engage with vec2/3/4, *reordering* the copying of the sub-vector elements (including allowing repeats and skips) based on an immediate supplied by the instruction. The fun comes when Pack/Unpack are enabled, and it is really important to be aware how the Arrays of vec2/3/4 become re-ordered *and swizzled at the same time.*

Pack/Unpack started out as {Pack / Unpack} but became its own distinct Mode over time. The main thing to keep in mind about Pack/Unpack is that it engages a swap of the ordering of the VL-SUBVL nested for-loops, in exactly the same way that Matrix REMAP can do. When Pack or Unpack is enabled it is the SUBVL for-loop that becomes outermost. A bit of thought shows that this is a 2D "Transpose" where Dimension X is VL and Dimension Y is SUBVL. However *both* source *and* destination may be independently "Transposed", which makes no sense at all until the fact that Swizzle can have a *different SUBVL* is taken into account.

Basically Pack/Unpack covers everything that VSX `vpkpx` and other ops can do, and then some: Saturation included, for arithmetic ops.

B.17 LD/ST with zero-immediate vs mapreduce mode

LD/ST operations with a zero immediate effectively means that on a Vector operation the element index to offset the memory location is multiplied by zero. Thus, a sequence of LD operations will load from the exact same address, and likewise STs to the exact same address.

Ordinarily this would make absolutely no sense whatsoever, except that Power ISA has cache-inhibited LD/STs (Power ISA v.1, Book III, 1.6.1, p1033), for accessing memory-mapped peripherals and other crucial uses. Thus, *despite not being a mapreduce mode*, zero-immediates cause multiple hits on the same element.

Mapreduce mode is not actually mapreduce at all: it is a relaxation of the normal rule where if the destination is a Scalar the Vector for-looping is not terminated on first write to the destination. Instead, the developer is expected to exploit the strict Program Order, make one of the sources the same as that Scalar destination, effectively making that Scalar register an "Accumulator", thus creating the *appearance* (effect) of Simple-V having a mapreduce capability, when in fact it is more of an artefact.

LD/ST zero-immediate has similar quirky overwriting as the "mapreduce" mode, but actually requires the registers to be Vectors. It is simply a mathematical artefact of multiplying by zero, which happens to be useful for cache-inhibited operations.

B.18 Limited space in LD/ST Mode

As pointed out in the {Load/Store Mode} page there is limited space in only 5 mode bits to fully express all potential modes of operation.

- LD/ST Immediate has no individual control over src/dest zeroing, whereas LD/ST Indexed does.
- Post-Increment is not possible with Saturation or Data-Dependent Fail-First
- Element-Strided LD/ST Indexed is not possible with Data-Dependent Fail-First.

Also, the LD/ST Indexed Mode can be element-strided (RB as a Scalar, times the element index), or, if that is not enough, although potentially costly it is possible to use `svstep` to compute a Vector RB sequence of Indices, then activate either `sz` or `dz` as required, as a workaround for LDST Immediate only having `zz`.

Simple-V is powerful but it cannot do everything! There is just not enough space and so some compromises had to be made.

B.19 sv.mtcr on entire 64-bit Condition Register

Normally, CR operations are either bit-based (where the element numbering actually applies to the CR Field) or field-based in which case the elements are still fields. The `sv.mtcr` and other instructions are actually full 64-bit Condition *Register* operations and are therefore qualified as Normal/Arithmetic not CRops.

This is to save on both Vector Length (VL of 16 is sufficient) as well as complexity in the Hazard Management when context-switching CR fields, as the entire batch of 128 CR Fields may be transferred to 8 GPRs with a VL of 16 and elwidth overriding of 32. Truncation is sufficent, dropping the top 32 bits of the Condition Register(s) which are always zero anyway.

B.20 Separate Scalar and Vector Condition Register files

As explained in the introduction {SVP64 Chapter} and {Condition Register Fields Mode} Scalar Power ISA lacks "Conditional Execution" present in ARM Scalar ISA of several decades. When Vectorised the fact that Rc=1 Vector results can immediately be used as a Predicate Mask back into the following instruction can result in large latency unless "Vector Chaining" is used in the Micro-Architecture.

But that aside is not the main problem faced by the introduction of Simple-V to the Power ISA: it's that the existing implementations (IBM) don't have "Conditional Execution" and to add it to their existing designs would be too disruptive a first step.

A compromise is to wipe blank certain entries in the Register Dependency Matrices by prohibiting some operations involving the two groups of CR Fields: those that fall into the existing Scalar 32-bit CR (fields CR0-CR7) and those that fall into the newly-introduced CR Fields, CR8-CR127.

This will drive compiler writers nuts, and give assembler writers headaches, but it gives IBM the opportunity to implement SVP64 without massive disruption. They can add an entirely new Vector CR register file, new pipelines etc safe in the knowledge that existing Scalar HDL needs no modification.

Appendix C

REMAP algorithms

C.0.1 REMAP Matrix pseudocode

The algorithm below shows how REMAP works more clearly, and may be executed as a python program:

```
# Finite State Machine version of the REMAP system.  much more likely
# to end up being actually used in actual hardware

# up to three dimensions permitted
xdim = 3
ydim = 2
zdim = 1

VL = xdim * ydim * zdim # set total (can repeat, e.g. VL=x*y*z*4)

lims = [xdim, ydim, zdim]
idxs = [0,0,0]   # starting indices
applydim = [1, 1]   # apply lower dims
order = [1,0,2]  # experiment with different permutations, here
offset = 0       # experiment with different offsetet, here
invxyz = [0,1,0] # inversion allowed

# pre-prepare the index state: run for "offset" times before
# actually starting.  this algorithm can also be used for re-entrancy
# if exceptions occur and a REMAP has to be started from where the
# interrupt left off.
for idx in range(offset):
    for i in range(3):
        idxs[order[i]] = idxs[order[i]] + 1
        if (idxs[order[i]] != lims[order[i]]):
            break
        idxs[order[i]] = 0

break_count = 0 # for pretty-printing

for idx in range(VL):
    ix = [0] * 3
    for i in range(3):
        ix[i] = idxs[i]
        if invxyz[i]:
```

```
            ix[i] = lims[i] - 1 - ix[i]
    new_idx = ix[2]
    if applydim[1]:
        new_idx = new_idx * ydim + ix[1]
    if applydim[0]:
        new_idx = new_idx * xdim + ix[0]
    print ("%d->%d" % (idx, new_idx)),
    break_count += 1
    if break_count == lims[order[0]]:
        print ("")
        break_count = 0
    # this is the exact same thing as the pre-preparation stage
    # above.  step 1: count up to the limit of the current dimension
    # step 2: if limit reached, zero it, and allow the *next* dimension
    # to increment.  repeat for 3 dimensions.
    for i in range(3):
        idxs[order[i]] = idxs[order[i]] + 1
        if (idxs[order[i]] != lims[order[i]]):
            break
        idxs[order[i]] = 0
```

An easier-to-read version (using python iterators) is given in a later section of this Appendix.

Each element index from the for-loop `0..VL-1` is run through the above algorithm to work out the **actual** element index, instead. Given that there are four possible SHAPE entries, up to four separate registers in any given operation may be simultaneously remapped:

```
function op_add(RT, RA, RB) # add not VADD!
  for (i=0,id=0,irs1=0,irs2=0; i < VL; i++)
    SVSTATE.srcstep = i # save context
    if (predval & 1<<i) # predication mask
       GPR[RT+remap1(id)] <= GPR[RA+remap2(irs1)] +
                             GPR[RB+remap3(irs2)];
       if (!RT.isvector) break;
    if (RT.isvector)  { id += 1; }
    if (RA.isvector)  { irs1 += 1; }
    if (RB.isvector)  { irs2 += 1; }
```

By changing remappings, 2D matrices may be transposed "in-place" for one operation, followed by setting a different permutation order without having to move the values in the registers to or from memory.

Note that:

- Over-running the register file clearly has to be detected and an illegal instruction exception thrown
- When non-default elwidths are set, the exact same algorithm still applies (i.e. it offsets *polymorphic* elements *within* registers rather than entire registers).
- If permute option 000 is utilised, the actual order of the reindexing does not change. However, modulo MVL still occurs which will result in repeated operations (use with caution).
- If two or more dimensions are set to zero, the actual order does not change!
- The above algorithm is pseudo-code **only**. Actual implementations will need to take into account the fact that the element for-looping must be **re-entrant**, due to the possibility of exceptions occurring. See SVSTATE SPR, which records the current element index. Continuing after return from an interrupt may introduce latency due to re-computation of the remapped offsets.
- Twin-predicated operations require **two** separate and distinct element offsets. The above pseudo-code algorithm will be applied separately and independently to each, should each of the two operands be remapped. *This even includes unit-strided LD/ST* and other operations in that category, where in that case it will be the **address offset** that is remapped: `EA <- (RA) + immediate * REMAP(elementoffset)`.
- Offset is especially useful, on its own, for accessing elements within the middle of a register. Without offsets,

it is necessary to either use a predicated MV, skipping the first elements, or performing a LOAD/STORE cycle to memory. With offsets, the data does not have to be moved.
- Setting the total elements (xdim+1) times (ydim+1) times (zdim+1) to less than MVL is **perfectly legal**, albeit very obscure. It permits entries to be regularly presented to operands **more than once**, thus allowing the same underlying registers to act as an accumulator of multiple vector or matrix operations, for example.
- Note especially that Program Order **must** still be respected even when overlaps occur that read or write the same register elements *including polymorphic ones*

Clearly here some considerable care needs to be taken as the remapping could hypothetically create arithmetic operations that target the exact same underlying registers, resulting in data corruption due to pipeline overlaps. Out-of-order / Superscalar micro-architectures with register-renaming will have an easier time dealing with this than DSP-style SIMD micro-architectures.

C.0.1.1 4x4 Matrix to vec4 Multiply (4x4 by 1x4)

The following settings will allow a 4x4 matrix (starting at f8), expressed as a sequence of 16 numbers first by row then by column, to be multiplied by a vector of length 4 (starting at f0), using a single FMAC instruction.

- SHAPE0: xdim=4, ydim=4, permute=yx, applied to f0
- SHAPE1: xdim=4, ydim=1, permute=xy, applied to f4
- VL=16, f4=vec, f0=vec, f8=vec
- FMAC f4, f0, f8, f4

The permutation on SHAPE0 will use f0 as a vec4 source. On the first four iterations through the hardware loop, the REMAPed index will not increment. On the second four, the index will increase by one. Likewise on each subsequent group of four.

The permutation on SHAPE1 will increment f4 continuously cycling through f4-f7 every four iterations of the hardware loop.

At the same time, VL will, because there is no SHAPE on f8, increment straight sequentially through the 16 values f8-f23 in the Matrix. The equivalent sequence thus is issued:

```
fmac f4, f0, f8, f4
fmac f5, f0, f9, f5
fmac f6, f0, f10, f6
fmac f7, f0, f11, f7
fmac f4, f1, f12, f4
fmac f5, f1, f13, f5
fmac f6, f1, f14, f6
fmac f7, f1, f15, f7
fmac f4, f2, f16, f4
fmac f5, f2, f17, f5
fmac f6, f2, f18, f6
fmac f7, f2, f19, f7
fmac f4, f3, f20, f4
fmac f5, f3, f21, f5
fmac f6, f3, f22, f6
fmac f7, f3, f23, f7
```

Hardware should easily pipeline the above FMACs and as long as each FMAC completes in 4 cycles or less there should be 100% sustained throughput, from the one single Vector FMAC.

The only other instruction required is to ensure that f4-f7 are initialised (usually to zero) however obviously if used as part of some other computation, which is frequently the case, then clearly the zeroing is not needed.

C.0.2 REMAP FFT, DFT, NTT

The algorithm from a later section of this Appendix shows how FFT REMAP works, and it may be executed as a standalone python3 program. The executable code is designed to illustrate how a hardware implementation may generate Indices which are completely independent of the Execution of element-level operations, even for something as complex as a Triple-loop Tukey-Cooley Schedule. A comprehensive demo and test suite may be found here including Complex Number FFT which deploys Vertical-First Mode on top of the REMAP Schedules.

Other uses include more than DFT and NTT: as abstracted RISC-paradigm the Schedules are not restricted in any way or tied to any particular instruction. If the programmer can find any algorithm which has identical triple nesting then the FFT Schedule may be used even there.

C.0.3 svshape pseudocode

```
# for convenience, VL to be calculated and stored in SVSTATE
vlen <- [0] * 7
mscale[0:5] <- 0b000001 # for scaling MAXVL
itercount[0:6] <- [0] * 7
SVSTATE[0:31] <- [0] * 32
# only overwrite REMAP if "persistence" is zero
if (SVSTATE[62] = 0b0) then
    SVSTATE[32:33] <- 0b00
    SVSTATE[34:35] <- 0b00
    SVSTATE[36:37] <- 0b00
    SVSTATE[38:39] <- 0b00
    SVSTATE[40:41] <- 0b00
    SVSTATE[42:46] <- 0b00000
    SVSTATE[62] <- 0b0
    SVSTATE[63] <- 0b0
# clear out all SVSHAPEs
SVSHAPE0[0:31] <- [0] * 32
SVSHAPE1[0:31] <- [0] * 32
SVSHAPE2[0:31] <- [0] * 32
SVSHAPE3[0:31] <- [0] * 32

# set schedule up for multiply
if (SVrm = 0b0000) then
    # VL in Matrix Multiply is xd*yd*zd
    xd <- (0b00 || SVxd) + 1
    yd <- (0b00 || SVyd) + 1
    zd <- (0b00 || SVzd) + 1
    n <- xd * yd * zd
    vlen[0:6] <- n[14:20]
    # set up template in SVSHAPE0, then copy to 1-3
    SVSHAPE0[0:5] <- (0b0 || SVxd)   # xdim
    SVSHAPE0[6:11] <- (0b0 || SVyd)   # ydim
    SVSHAPE0[12:17] <- (0b0 || SVzd)   # zdim
    SVSHAPE0[28:29] <- 0b11           # skip z
    # copy
    SVSHAPE1[0:31] <- SVSHAPE0[0:31]
    SVSHAPE2[0:31] <- SVSHAPE0[0:31]
    SVSHAPE3[0:31] <- SVSHAPE0[0:31]
    # set up FRA
    SVSHAPE1[18:20] <- 0b001          # permute x,z,y
    SVSHAPE1[28:29] <- 0b01           # skip z
```

```
    # FRC
    SVSHAPE2[18:20] <- 0b001          # permute x,z,y
    SVSHAPE2[28:29] <- 0b11           # skip y

# set schedule up for FFT butterfly
if (SVrm = 0b0001) then
    # calculate O(N log2 N)
    n <- [0] * 3
    do while n < 5
       if SVxd[4-n] = 0 then
           leave
       n <- n + 1
    n <- ((0b0 || SVxd) + 1) * n
    vlen[0:6] <- n[1:7]
    # set up template in SVSHAPE0, then copy to 1-3
    # for FRA and FRT
    SVSHAPE0[0:5] <- (0b0 || SVxd)   # xdim
    SVSHAPE0[12:17] <- (0b0 || SVzd)   # zdim - "striding" (2D FFT)
    mscale <- (0b0 || SVzd) + 1
    SVSHAPE0[30:31] <- 0b01          # Butterfly mode
    # copy
    SVSHAPE1[0:31] <- SVSHAPE0[0:31]
    SVSHAPE2[0:31] <- SVSHAPE0[0:31]
    # set up FRB and FRS
    SVSHAPE1[28:29] <- 0b01           # j+halfstep schedule
    # FRC (coefficients)
    SVSHAPE2[28:29] <- 0b10           # k schedule

# set schedule up for (i)DCT Inner butterfly
# SVrm Mode 4 (Mode 12 for iDCT) is for on-the-fly (Vertical-First Mode)
if ((SVrm = 0b0100) |
    (SVrm = 0b1100)) then
    # calculate O(N log2 N)
    n <- [0] * 3
    do while n < 5
       if SVxd[4-n] = 0 then
           leave
       n <- n + 1
    n <- ((0b0 || SVxd) + 1) * n
    vlen[0:6] <- n[1:7]
    # set up template in SVSHAPE0, then copy to 1-3
    # set up FRB and FRS
    SVSHAPE0[0:5] <- (0b0 || SVxd)   # xdim
    SVSHAPE0[12:17] <- (0b0 || SVzd)   # zdim - "striding" (2D DCT)
    mscale <- (0b0 || SVzd) + 1
    if (SVrm = 0b1100) then
        SVSHAPE0[30:31] <- 0b11          # iDCT mode
        SVSHAPE0[18:20] <- 0b011         # iDCT Inner Butterfly sub-mode
    else
        SVSHAPE0[30:31] <- 0b01          # DCT mode
        SVSHAPE0[18:20] <- 0b001         # DCT Inner Butterfly sub-mode
        SVSHAPE0[21:23] <- 0b001         # "inverse" on outer loop
    SVSHAPE0[6:11] <- 0b000011       # (i)DCT Inner Butterfly mode 4
    # copy
    SVSHAPE1[0:31] <- SVSHAPE0[0:31]
```

```
        SVSHAPE2[0:31] <- SVSHAPE0[0:31]
        if (SVrm != 0b0100) & (SVrm != 0b1100) then
            SVSHAPE3[0:31] <- SVSHAPE0[0:31]
        # for FRA and FRT
        SVSHAPE0[28:29] <- 0b01           # j+halfstep schedule
        # for cos coefficient
        SVSHAPE2[28:29] <- 0b10           # ci (k for mode 4) schedule
        SVSHAPE2[12:17] <- 0b000000       # reset costable "striding" to 1
        if (SVrm != 0b0100) & (SVrm != 0b1100) then
            SVSHAPE3[28:29] <- 0b11           # size schedule

    # set schedule up for (i)DCT Outer butterfly
    if (SVrm = 0b0011) | (SVrm = 0b1011) then
        # calculate O(N log2 N) number of outer butterfly overlapping adds
        vlen[0:6] <- [0] * 7
        n <- 0b000
        size <- 0b0000001
        itercount[0:6] <- (0b00 || SVxd) + 0b0000001
        itercount[0:6] <- (0b0 || itercount[0:5])
        do while n < 5
           if SVxd[4-n] = 0 then
               leave
           n <- n + 1
           count <- (itercount - 0b0000001) * size
           vlen[0:6] <- vlen + count[7:13]
           size[0:6] <- (size[1:6] || 0b0)
           itercount[0:6] <- (0b0 || itercount[0:5])
        # set up template in SVSHAPE0, then copy to 1-3
        # set up FRB and FRS
        SVSHAPE0[0:5] <- (0b0 || SVxd)   # xdim
        SVSHAPE0[12:17] <- (0b0 || SVzd)   # zdim - "striding" (2D DCT)
        mscale <- (0b0 || SVzd) + 1
        if (SVrm = 0b1011) then
            SVSHAPE0[30:31] <- 0b11      # iDCT mode
            SVSHAPE0[18:20] <- 0b011     # iDCT Outer Butterfly sub-mode
            SVSHAPE0[21:23] <- 0b101     # "inverse" on outer and inner loop
        else
            SVSHAPE0[30:31] <- 0b01      # DCT mode
            SVSHAPE0[18:20] <- 0b100     # DCT Outer Butterfly sub-mode
        SVSHAPE0[6:11] <- 0b000010       # DCT Butterfly mode
        # copy
        SVSHAPE1[0:31] <- SVSHAPE0[0:31] # j+halfstep schedule
        SVSHAPE2[0:31] <- SVSHAPE0[0:31] # costable coefficients
        # for FRA and FRT
        SVSHAPE1[28:29] <- 0b01           # j+halfstep schedule
        # reset costable "striding" to 1
        SVSHAPE2[12:17] <- 0b000000

    # set schedule up for DCT COS table generation
    if (SVrm = 0b0101) | (SVrm = 0b1101) then
        # calculate O(N log2 N)
        vlen[0:6] <- [0] * 7
        itercount[0:6] <- (0b00 || SVxd) + 0b0000001
        itercount[0:6] <- (0b0 || itercount[0:5])
        n <- [0] * 3
```

```
    do while n < 5
       if SVxd[4-n] = 0 then
           leave
       n <- n + 1
       vlen[0:6] <- vlen + itercount
       itercount[0:6] <- (0b0 || itercount[0:5])
    # set up template in SVSHAPE0, then copy to 1-3
    # set up FRB and FRS
    SVSHAPE0[0:5] <- (0b0 || SVxd)   # xdim
    SVSHAPE0[12:17] <- (0b0 || SVzd)   # zdim - "striding" (2D DCT)
    mscale <- (0b0 || SVzd) + 1
    SVSHAPE0[30:31] <- 0b01          # DCT/FFT mode
    SVSHAPE0[6:11] <- 0b000100       # DCT Inner Butterfly COS-gen mode
    if (SVrm = 0b0101) then
        SVSHAPE0[21:23] <- 0b001     # "inverse" on outer loop for DCT
    # copy
    SVSHAPE1[0:31] <- SVSHAPE0[0:31]
    SVSHAPE2[0:31] <- SVSHAPE0[0:31]
    # for cos coefficient
    SVSHAPE1[28:29] <- 0b10           # ci schedule
    SVSHAPE2[28:29] <- 0b11           # size schedule

# set schedule up for iDCT / DCT inverse of half-swapped ordering
if (SVrm = 0b0110) | (SVrm = 0b1110) | (SVrm = 0b1111) then
    vlen[0:6] <- (0b00 || SVxd) + 0b0000001
    # set up template in SVSHAPE0
    SVSHAPE0[0:5] <- (0b0 || SVxd)   # xdim
    SVSHAPE0[12:17] <- (0b0 || SVzd)   # zdim - "striding" (2D DCT)
    mscale <- (0b0 || SVzd) + 1
    if (SVrm = 0b1110) then
        SVSHAPE0[18:20] <- 0b001     # DCT opposite half-swap
    if (SVrm = 0b1111) then
        SVSHAPE0[30:31] <- 0b01          # FFT mode
    else
        SVSHAPE0[30:31] <- 0b11          # DCT mode
    SVSHAPE0[6:11] <- 0b000101       # DCT "half-swap" mode

# set schedule up for parallel reduction
if (SVrm = 0b0111) then
    # calculate the total number of operations (brute-force)
    vlen[0:6] <- [0] * 7
    itercount[0:6] <- (0b00 || SVxd) + 0b0000001
    step[0:6] <- 0b0000001
    i[0:6] <- 0b0000000
    do while step <u itercount
        newstep <- step[1:6] || 0b0
        j[0:6] <- 0b0000000
        do while (j+step <u itercount)
            j <- j + newstep
            i <- i + 1
        step <- newstep
    # VL in Parallel-Reduce is the number of operations
    vlen[0:6] <- i
    # set up template in SVSHAPE0, then copy to 1. only 2 needed
    SVSHAPE0[0:5] <- (0b0 || SVxd)   # xdim
```

```
    SVSHAPE0[12:17] <- (0b0 || SVzd)   # zdim - "striding" (2D DCT)
    mscale <- (0b0 || SVzd) + 1
    SVSHAPE0[30:31] <- 0b10          # parallel reduce submode
    # copy
    SVSHAPE1[0:31] <- SVSHAPE0[0:31]
    # set up right operand (left operand 28:29 is zero)
    SVSHAPE1[28:29] <- 0b01           # right operand

# set VL, MVL and Vertical-First
m[0:12] <- vlen * mscale
maxvl[0:6] <- m[6:12]
SVSTATE[0:6] <- maxvl  # MAVXL
SVSTATE[7:13] <- vlen  # VL
SVSTATE[63] <- vf
```

C.0.4 svindex pseudocode

```
# based on nearest MAXVL compute other dimension
MVL <- SVSTATE[0:6]
d <- [0] * 6
dim <- SVd+1
do while d*dim <u ([0]*4 || MVL)
   d <- d + 1

# set up template, then copy once location identified
shape <- [0]*32
shape[30:31] <- 0b00            # mode
if SVyx = 0 then
    shape[18:20] <- 0b110       # indexed xd/yd
    shape[0:5] <- (0b0 || SVd)  # xdim
    if sk = 0 then shape[6:11] <- 0 # ydim
    else           shape[6:11] <- 0b111111 # ydim max
else
    shape[18:20] <- 0b111       # indexed yd/xd
    if sk = 1 then shape[6:11] <- 0 # ydim
    else           shape[6:11] <- d-1 # ydim max
    shape[0:5] <- (0b0 || SVd) # ydim
shape[12:17] <- (0b0 || SVG)        # SVGPR
shape[28:29] <- ew                  # element-width override
shape[21] <- sk                     # skip 1st dimension

# select the mode for updating SVSHAPEs
SVSTATE[62] <- mm # set or clear persistence
if mm = 0 then
    # clear out all SVSHAPEs first
    SVSHAPE0[0:31] <- [0] * 32
    SVSHAPE1[0:31] <- [0] * 32
    SVSHAPE2[0:31] <- [0] * 32
    SVSHAPE3[0:31] <- [0] * 32
    SVSTATE[32:41] <- [0] * 10 # clear REMAP.mi/o
    SVSTATE[42:46] <- rmm # rmm exactly REMAP.SVme
    idx <- 0
    for bit = 0 to 4
        if rmm[4-bit] then
```

```
            # activate requested shape
            if idx = 0 then SVSHAPE0 <- shape
            if idx = 1 then SVSHAPE1 <- shape
            if idx = 2 then SVSHAPE2 <- shape
            if idx = 3 then SVSHAPE3 <- shape
            SVSTATE[bit*2+32:bit*2+33] <- idx
            # increment shape index, modulo 4
            if idx = 3 then idx <- 0
            else            idx <- idx + 1
else
    # refined SVSHAPE/REMAP update mode
    bit <- rmm[0:2]
    idx <- rmm[3:4]
    if idx = 0 then SVSHAPE0 <- shape
    if idx = 1 then SVSHAPE1 <- shape
    if idx = 2 then SVSHAPE2 <- shape
    if idx = 3 then SVSHAPE3 <- shape
    SVSTATE[bit*2+32:bit*2+33] <- idx
    SVSTATE[46-bit] <- 1
```

C.0.5 svshape2 pseudocode

```
# based on nearest MAXVL compute other dimension
MVL <- SVSTATE[0:6]
d <- [0] * 6
dim <- SVd+1
do while d*dim <u ([0]*4 || MVL)
   d <- d + 1
# set up template, then copy once location identified
shape <- [0]*32
shape[30:31] <- 0b00            # mode
shape[0:5] <- (0b0 || SVd)      # x/ydim
if SVyx = 0 then
    shape[18:20] <- 0b000       # ordering xd/yd(/zd)
    if sk = 0 then shape[6:11] <- 0 # ydim
    else           shape[6:11] <- 0b111111 # ydim max
else
    shape[18:20] <- 0b010       # ordering yd/xd(/zd)
    if sk = 1 then shape[6:11] <- 0 # ydim
    else           shape[6:11] <- d-1 # ydim max
# offset (the prime purpose of this instruction)
shape[24:27] <- SVo         # offset
if sk = 1 then shape[28:29] <- 0b01 # skip 1st dimension
else           shape[28:29] <- 0b00 # no skipping
# select the mode for updating SVSHAPEs
SVSTATE[62] <- mm # set or clear persistence
if mm = 0 then
    # clear out all SVSHAPEs first
    SVSHAPE0[0:31] <- [0] * 32
    SVSHAPE1[0:31] <- [0] * 32
    SVSHAPE2[0:31] <- [0] * 32
    SVSHAPE3[0:31] <- [0] * 32
    SVSTATE[32:41] <- [0] * 10 # clear REMAP.mi/o
    SVSTATE[42:46] <- rmm # rmm exactly REMAP.SVme
```

```
            idx <- 0
            for bit = 0 to 4
                if rmm[4-bit] then
                    # activate requested shape
                    if idx = 0 then SVSHAPE0 <- shape
                    if idx = 1 then SVSHAPE1 <- shape
                    if idx = 2 then SVSHAPE2 <- shape
                    if idx = 3 then SVSHAPE3 <- shape
                    SVSTATE[bit*2+32:bit*2+33] <- idx
                    # increment shape index, modulo 4
                    if idx = 3 then idx <- 0
                    else            idx <- idx + 1
        else
            # refined SVSHAPE/REMAP update mode
            bit <- rmm[0:2]
            idx <- rmm[3:4]
            if idx = 0 then SVSHAPE0 <- shape
            if idx = 1 then SVSHAPE1 <- shape
            if idx = 2 then SVSHAPE2 <- shape
            if idx = 3 then SVSHAPE3 <- shape
            SVSTATE[bit*2+32:bit*2+33] <- idx
            SVSTATE[46-bit] <- 1
```

[[!tag standards]]

Appendix D

Simple-V pseudocode

D.1 svstep

SVL-Form

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

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
    step <- SVSTATE_NEXT(SVi, vf)
    RT <- [0]*57 || step
```

Special Registers Altered:

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

D.2 setvl

SVL-Form

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

Pseudo-code:

```
overflow <- 0b0
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".
    if ms = 1 then
        SVSTATE[63] <- vf   # set Vertical-First mode
        SVSTATE[62] <- 0b0  # clear persist bit
```

Special Registers Altered:

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

D.3 svremap

SVRM-Form

- svremap SVme,mi0,mi1,mi2,mo0,mo1,pst

Pseudo-code:

```
    # registers RA RB RC RT EA/FRS SVSHAPE0-3 indices
    SVSTATE[32:33] <- mi0
    SVSTATE[34:35] <- mi1
    SVSTATE[36:37] <- mi2
    SVSTATE[38:39] <- mo0
    SVSTATE[40:41] <- mo1
    # enable bit for RA RB RC RT EA/FRS
    SVSTATE[42:46] <- SVme
    # persistence bit (applies to more than one instruction)
    SVSTATE[62] <- pst
```

Special Registers Altered:

```
    None
```

D.4 svshape

SVM-Form

- svshape SVxd,SVyd,SVzd,SVrm,vf

Pseudo-code:

```
# for convenience, VL to be calculated and stored in SVSTATE
vlen <- [0] * 7
mscale[0:5] <- 0b000001 # for scaling MAXVL
itercount[0:6] <- [0] * 7
SVSTATE[0:31] <- [0] * 32
# only overwrite REMAP if "persistence" is zero
if (SVSTATE[62] = 0b0) then
    SVSTATE[32:33] <- 0b00
    SVSTATE[34:35] <- 0b00
    SVSTATE[36:37] <- 0b00
    SVSTATE[38:39] <- 0b00
    SVSTATE[40:41] <- 0b00
    SVSTATE[42:46] <- 0b00000
    SVSTATE[62] <- 0b0
    SVSTATE[63] <- 0b0
# clear out all SVSHAPEs
SVSHAPE0[0:31] <- [0] * 32
SVSHAPE1[0:31] <- [0] * 32
SVSHAPE2[0:31] <- [0] * 32
SVSHAPE3[0:31] <- [0] * 32
# set schedule up for multiply
if (SVrm = 0b0000) then
    # VL in Matrix Multiply is xd*yd*zd
    xd <- (0b00 || SVxd) + 1
    yd <- (0b00 || SVyd) + 1
    zd <- (0b00 || SVzd) + 1
    n <- xd * yd * zd
    vlen[0:6] <- n[14:20]
    # set up template in SVSHAPE0, then copy to 1-3
    SVSHAPE0[0:5] <- (0b0 || SVxd)   # xdim
    SVSHAPE0[6:11] <- (0b0 || SVyd)   # ydim
    SVSHAPE0[12:17] <- (0b0 || SVzd)   # zdim
    SVSHAPE0[28:29] <- 0b11           # skip z
    # copy
    SVSHAPE1[0:31] <- SVSHAPE0[0:31]
    SVSHAPE2[0:31] <- SVSHAPE0[0:31]
    SVSHAPE3[0:31] <- SVSHAPE0[0:31]
    # set up FRA
    SVSHAPE1[18:20] <- 0b001          # permute x,z,y
    SVSHAPE1[28:29] <- 0b01           # skip z
    # FRC
    SVSHAPE2[18:20] <- 0b001          # permute x,z,y
    SVSHAPE2[28:29] <- 0b11           # skip y
# set schedule up for FFT butterfly
if (SVrm = 0b0001) then
    # calculate O(N log2 N)
    n <- [0] * 3
    do while n < 5
       if SVxd[4-n] = 0 then
           leave
       n <- n + 1
    n <- ((0b0 || SVxd) + 1) * n
    vlen[0:6] <- n[1:7]
    # set up template in SVSHAPE0, then copy to 1-3
    # for FRA and FRT
```

```
    SVSHAPE0[0:5] <- (0b0 || SVxd)   # xdim
    SVSHAPE0[12:17] <- (0b0 || SVzd)   # zdim - "striding" (2D FFT)
    mscale <- (0b0 || SVzd) + 1
    SVSHAPE0[30:31] <- 0b01          # Butterfly mode
    # copy
    SVSHAPE1[0:31] <- SVSHAPE0[0:31]
    SVSHAPE2[0:31] <- SVSHAPE0[0:31]
    # set up FRB and FRS
    SVSHAPE1[28:29] <- 0b01           # j+halfstep schedule
    # FRC (coefficients)
    SVSHAPE2[28:29] <- 0b10           # k schedule
# set schedule up for (i)DCT Inner butterfly
# SVrm Mode 4 (Mode 12 for iDCT) is for on-the-fly (Vertical-First Mode)
if ((SVrm = 0b0100) |
    (SVrm = 0b1100)) then
    # calculate O(N log2 N)
    n <- [0] * 3
    do while n < 5
       if SVxd[4-n] = 0 then
           leave
       n <- n + 1
    n <- ((0b0 || SVxd) + 1) * n
    vlen[0:6] <- n[1:7]
    # set up template in SVSHAPE0, then copy to 1-3
    # set up FRB and FRS
    SVSHAPE0[0:5] <- (0b0 || SVxd)   # xdim
    SVSHAPE0[12:17] <- (0b0 || SVzd)   # zdim - "striding" (2D DCT)
    mscale <- (0b0 || SVzd) + 1
    if (SVrm = 0b1100) then
        SVSHAPE0[30:31] <- 0b11          # iDCT mode
        SVSHAPE0[18:20] <- 0b011         # iDCT Inner Butterfly sub-mode
    else
        SVSHAPE0[30:31] <- 0b01          # DCT mode
        SVSHAPE0[18:20] <- 0b001         # DCT Inner Butterfly sub-mode
        SVSHAPE0[21:23] <- 0b001         # "inverse" on outer loop
    SVSHAPE0[6:11] <- 0b000011       # (i)DCT Inner Butterfly mode 4
    # copy
    SVSHAPE1[0:31] <- SVSHAPE0[0:31]
    SVSHAPE2[0:31] <- SVSHAPE0[0:31]
    if (SVrm != 0b0100) & (SVrm != 0b1100) then
        SVSHAPE3[0:31] <- SVSHAPE0[0:31]
    # for FRA and FRT
    SVSHAPE0[28:29] <- 0b01           # j+halfstep schedule
    # for cos coefficient
    SVSHAPE2[28:29] <- 0b10           # ci (k for mode 4) schedule
    SVSHAPE2[12:17] <- 0b000000       # reset costable "striding" to 1
    if (SVrm != 0b0100) & (SVrm != 0b1100) then
        SVSHAPE3[28:29] <- 0b11           # size schedule
# set schedule up for (i)DCT Outer butterfly
if (SVrm = 0b0011) | (SVrm = 0b1011) then
    # calculate O(N log2 N) number of outer butterfly overlapping adds
    vlen[0:6] <- [0] * 7
    n <- 0b000
    size <- 0b0000001
    itercount[0:6] <- (0b00 || SVxd) + 0b0000001
```

```
        itercount[0:6] <- (0b0 || itercount[0:5])
        do while n < 5
           if SVxd[4-n] = 0 then
               leave
           n <- n + 1
           count <- (itercount - 0b0000001) * size
           vlen[0:6] <- vlen + count[7:13]
           size[0:6] <- (size[1:6] || 0b0)
           itercount[0:6] <- (0b0 || itercount[0:5])
        # set up template in SVSHAPE0, then copy to 1-3
        # set up FRB and FRS
        SVSHAPE0[0:5] <- (0b0 || SVxd)   # xdim
        SVSHAPE0[12:17] <- (0b0 || SVzd)   # zdim - "striding" (2D DCT)
        mscale <- (0b0 || SVzd) + 1
        if (SVrm = 0b1011) then
            SVSHAPE0[30:31] <- 0b11      # iDCT mode
            SVSHAPE0[18:20] <- 0b011     # iDCT Outer Butterfly sub-mode
            SVSHAPE0[21:23] <- 0b101     # "inverse" on outer and inner loop
        else
            SVSHAPE0[30:31] <- 0b01      # DCT mode
            SVSHAPE0[18:20] <- 0b100     # DCT Outer Butterfly sub-mode
        SVSHAPE0[6:11] <- 0b000010       # DCT Butterfly mode
        # copy
        SVSHAPE1[0:31] <- SVSHAPE0[0:31] # j+halfstep schedule
        SVSHAPE2[0:31] <- SVSHAPE0[0:31] # costable coefficients
        # for FRA and FRT
        SVSHAPE1[28:29] <- 0b01           # j+halfstep schedule
        # reset costable "striding" to 1
        SVSHAPE2[12:17] <- 0b000000
    # set schedule up for DCT COS table generation
    if (SVrm = 0b0101) | (SVrm = 0b1101) then
        # calculate O(N log2 N)
        vlen[0:6] <- [0] * 7
        itercount[0:6] <- (0b00 || SVxd) + 0b0000001
        itercount[0:6] <- (0b0 || itercount[0:5])
        n <- [0] * 3
        do while n < 5
           if SVxd[4-n] = 0 then
               leave
           n <- n + 1
           vlen[0:6] <- vlen + itercount
           itercount[0:6] <- (0b0 || itercount[0:5])
        # set up template in SVSHAPE0, then copy to 1-3
        # set up FRB and FRS
        SVSHAPE0[0:5] <- (0b0 || SVxd)   # xdim
        SVSHAPE0[12:17] <- (0b0 || SVzd)   # zdim - "striding" (2D DCT)
        mscale <- (0b0 || SVzd) + 1
        SVSHAPE0[30:31] <- 0b01          # DCT/FFT mode
        SVSHAPE0[6:11] <- 0b000100       # DCT Inner Butterfly COS-gen mode
        if (SVrm = 0b0101) then
            SVSHAPE0[21:23] <- 0b001     # "inverse" on outer loop for DCT
        # copy
        SVSHAPE1[0:31] <- SVSHAPE0[0:31]
        SVSHAPE2[0:31] <- SVSHAPE0[0:31]
        # for cos coefficient
```

```
    SVSHAPE1[28:29] <- 0b10           # ci schedule
    SVSHAPE2[28:29] <- 0b11           # size schedule
# set schedule up for iDCT / DCT inverse of half-swapped ordering
if (SVrm = 0b0110) | (SVrm = 0b1110) | (SVrm = 0b1111) then
    vlen[0:6] <- (0b00 || SVxd) + 0b0000001
    # set up template in SVSHAPE0
    SVSHAPE0[0:5] <- (0b0 || SVxd)   # xdim
    SVSHAPE0[12:17] <- (0b0 || SVzd)   # zdim - "striding" (2D DCT)
    mscale <- (0b0 || SVzd) + 1
    if (SVrm = 0b1110) then
        SVSHAPE0[18:20] <- 0b001     # DCT opposite half-swap
    if (SVrm = 0b1111) then
        SVSHAPE0[30:31] <- 0b01          # FFT mode
    else
        SVSHAPE0[30:31] <- 0b11          # DCT mode
    SVSHAPE0[6:11] <- 0b000101       # DCT "half-swap" mode
# set schedule up for parallel reduction or prefix-sum
if (SVrm = 0b0111) then
    # is scan/prefix-sum
    is_scan <- SVyd = 2
    # calculate the total number of operations (brute-force)
    vlen[0:6] <- [0] * 7
    itercount[0:6] <- (0b00 || SVxd) + 0b0000001
    if is_scan then
        # prefix sum algorithm with operations replaced with
        # incrementing vlen
        dist <- 1
        vlen[0:6] <- 0
        do while dist <u itercount
            start <- dist * 2 - 1
            step <- dist * 2
            i <- start
            do while i <u itercount
                vlen[0:6] <- vlen[0:6] + 1
                i <- i + step
            dist <- dist * 2
        dist <- dist / 2
        do while dist != 0
            i <- dist * 3 - 1
            do while i <u itercount
                vlen[0:6] <- vlen[0:6] + 1
                i <- i + dist * 2
            dist <- dist / 2
    else
        step <- 0b0000001
        i <- 0b0000000
        do while step <u itercount
            newstep <- step[1:6] || 0b0
            j[0:6] <- 0b0000000
            do while (j+step <u itercount)
                j <- j + newstep
                i <- i + 1
            step <- newstep
        # VL in Parallel-Reduce is the number of operations
        vlen[0:6] <- i
```

```
        # set up template in SVSHAPE0, then copy to 1. only 2 needed
        SVSHAPE0[0:5] <- (0b0 || SVxd)   # xdim
        SVSHAPE0[12:17] <- (0b0 || SVzd)   # zdim - "striding" (2D DCT)
        mscale <- (0b0 || SVzd) + 1
        SVSHAPE0[30:31] <- 0b10          # parallel reduce/prefix submode
        # copy
        SVSHAPE1[0:31] <- SVSHAPE0[0:31]
        # set up submodes: parallel or prefix
        SVSHAPE0[28:29] <- 0b00   # left operand
        SVSHAPE1[28:29] <- 0b01   # right operand
        if is_scan then
            SVSHAPE0[28:29] <- 0b10   # left operand
            SVSHAPE1[28:29] <- 0b11   # right operand
    # set VL, MVL and Vertical-First
    m[0:12] <- vlen * mscale
    maxvl[0:6] <- m[6:12]
    SVSTATE[0:6] <- maxvl  # MAVXL
    SVSTATE[7:13] <- vlen  # VL
    SVSTATE[63] <- vf
```

Special Registers Altered:

```
None
```

D.5 svindex

SVI-Form

- svindex SVG,rmm,SVd,ew,SVyx,mm,sk

Pseudo-code:

```
# based on nearest MAXVL compute other dimension
MVL <- SVSTATE[0:6]
d <- [0] * 6
dim <- SVd+1
do while d*dim <u ([0]*4 || MVL)
   d <- d + 1
# set up template, then copy once location identified
shape <- [0]*32
shape[30:31] <- 0b00            # mode
if SVyx = 0 then
    shape[18:20] <- 0b110       # indexed xd/yd
    shape[0:5] <- (0b0 || SVd)  # xdim
    if sk = 0 then shape[6:11] <- 0 # ydim
    else           shape[6:11] <- 0b111111 # ydim max
else
    shape[18:20] <- 0b111       # indexed yd/xd
    if sk = 1 then shape[6:11] <- 0 # ydim
    else           shape[6:11] <- d-1 # ydim max
    shape[0:5] <- (0b0 || SVd) # ydim
shape[12:17] <- (0b0 || SVG)        # SVGPR
shape[28:29] <- ew                  # element-width override
shape[21] <- sk                     # skip 1st dimension
# select the mode for updating SVSHAPEs
SVSTATE[62] <- mm # set or clear persistence
```

```
if mm = 0 then
    # clear out all SVSHAPEs first
    SVSHAPE0[0:31] <- [0] * 32
    SVSHAPE1[0:31] <- [0] * 32
    SVSHAPE2[0:31] <- [0] * 32
    SVSHAPE3[0:31] <- [0] * 32
    SVSTATE[32:41] <- [0] * 10 # clear REMAP.mi/o
    SVSTATE[42:46] <- rmm # rmm exactly REMAP.SVme
    idx <- 0
    for bit = 0 to 4
        if rmm[4-bit] then
            # activate requested shape
            if idx = 0 then SVSHAPE0 <- shape
            if idx = 1 then SVSHAPE1 <- shape
            if idx = 2 then SVSHAPE2 <- shape
            if idx = 3 then SVSHAPE3 <- shape
            SVSTATE[bit*2+32:bit*2+33] <- idx
            # increment shape index, modulo 4
            if idx = 3 then idx <- 0
            else            idx <- idx + 1
else
    # refined SVSHAPE/REMAP update mode
    bit <- rmm[0:2]
    idx <- rmm[3:4]
    if idx = 0 then SVSHAPE0 <- shape
    if idx = 1 then SVSHAPE1 <- shape
    if idx = 2 then SVSHAPE2 <- shape
    if idx = 3 then SVSHAPE3 <- shape
    SVSTATE[bit*2+32:bit*2+33] <- idx
    SVSTATE[46-bit] <- 1
```

Special Registers Altered:

```
None
```

D.6 svshape2

SVM2-Form

- svshape2 SVo,SVyx,rmm,SVd,sk,mm

Pseudo-code:

```
# based on nearest MAXVL compute other dimension
MVL <- SVSTATE[0:6]
d <- [0] * 6
dim <- SVd+1
do while d*dim <u ([0]*4 || MVL)
   d <- d + 1
# set up template, then copy once location identified
shape <- [0]*32
shape[30:31] <- 0b00            # mode
shape[0:5] <- (0b0 || SVd)      # x/ydim
if SVyx = 0 then
    shape[18:20] <- 0b000       # ordering xd/yd(/zd)
    if sk = 0 then shape[6:11] <- 0 # ydim
```

```
        else           shape[6:11] <- 0b111111 # ydim max
    else
        shape[18:20] <- 0b010       # ordering yd/xd(/zd)
        if sk = 1 then shape[6:11] <- 0 # ydim
        else           shape[6:11] <- d-1 # ydim max
    # offset (the prime purpose of this instruction)
    shape[24:27] <- SVo         # offset
    if sk = 1 then shape[28:29] <- 0b01 # skip 1st dimension
    else           shape[28:29] <- 0b00 # no skipping
    # select the mode for updating SVSHAPEs
    SVSTATE[62] <- mm # set or clear persistence
    if mm = 0 then
        # clear out all SVSHAPEs first
        SVSHAPE0[0:31] <- [0] * 32
        SVSHAPE1[0:31] <- [0] * 32
        SVSHAPE2[0:31] <- [0] * 32
        SVSHAPE3[0:31] <- [0] * 32
        SVSTATE[32:41] <- [0] * 10 # clear REMAP.mi/o
        SVSTATE[42:46] <- rmm # rmm exactly REMAP.SVme
        idx <- 0
        for bit = 0 to 4
            if rmm[4-bit] then
                # activate requested shape
                if idx = 0 then SVSHAPE0 <- shape
                if idx = 1 then SVSHAPE1 <- shape
                if idx = 2 then SVSHAPE2 <- shape
                if idx = 3 then SVSHAPE3 <- shape
                SVSTATE[bit*2+32:bit*2+33] <- idx
                # increment shape index, modulo 4
                if idx = 3 then idx <- 0
                else            idx <- idx + 1
    else
        # refined SVSHAPE/REMAP update mode
        bit <- rmm[0:2]
        idx <- rmm[3:4]
        if idx = 0 then SVSHAPE0 <- shape
        if idx = 1 then SVSHAPE1 <- shape
        if idx = 2 then SVSHAPE2 <- shape
        if idx = 3 then SVSHAPE3 <- shape
        SVSTATE[bit*2+32:bit*2+33] <- idx
        SVSTATE[46-bit] <- 1
```

Special Registers Altered:

```
None
```

Appendix E

Simple-V Analysis

E.1 Simple-V Analysis

The creation and maintenance of SVP64 Categorisation is an automated process that uses "Register profiling", reading machine-readable versions of the Power ISA Specification and tables in order to make the Vectorisation Categorisation. To create this information by hand is neither sensible nor desirable: it may take far longer and introduce errors.

This in turn effectively makes that analysis program part of the Simple-V Specification. Its source code is therefore listed here

```
#!/usr/bin/env python2
#
# NOTE that this program is python2 compatible, please do not stop it
# from working by adding syntax that prevents that.
#
# Initial version written by lkcl Oct 2020
# This program analyses the Power 9 op codes and looks at in/out register uses
# The results are displayed:
#   https://libre-soc.org/openpower/opcode_regs_deduped/
#
# It finds .csv files in the directory isatables/
# then goes through the categories and creates svp64 CSV augmentation
# tables on a per-opcode basis
#
# NOTE: this program is effectively part of the Simple-V Specification.
# it encapsulates the relationships of what can be SVP64-encoded and
# holds all of the information on how to encode and decode SVP64.
# By auto-generating tables that go into the Simple-V Specification
# this program *is* the specification. do not be confused just because
# it is in python: if you do not understand please ask questions and
# help create patches with explanatory comments.

import argparse
import csv
import enum
import os
from os.path import dirname, join
from glob import glob
from collections import defaultdict
```

```
from collections import OrderedDict
from openpower.decoder.power_svp64 import SVP64RM
from openpower.decoder.power_enums import find_wiki_file, get_csv
from openpower.util import log


# Ignore those containing: valid test sprs
def glob_valid_csvs(root):
    def check_csv(fname):
        _, name = os.path.split(fname)
        if '-' in name:
            return False
        if 'valid' in fname:
            return False
        if 'test' in fname:
            return False
        if fname.endswith('insndb.csv'):
            return False
        if fname.endswith('sprs.csv'):
            return False
        if fname.endswith('minor_19_valid.csv'):
            return False
        if 'RM' in fname:
            return False
        return True

    yield from filter(check_csv, glob(root))


# Write an array of dictionaries to the CSV file name:
def write_csv(name, items, headers):
    file_path = find_wiki_file(name)
    with open(file_path, 'w') as csvfile:
        writer = csv.DictWriter(csvfile, headers, lineterminator="\n")
        writer.writeheader()
        writer.writerows(items)

# This will return True if all values are true.
# Not sure what this is about


def blank_key(row):
    # for v in row.values():
    #    if 'SPR' in v: # skip all SPRs
    #        return True
    for v in row.values():
        if v:
            return False
    return True

# General purpose registers have names like: RA, RT, R1, ...
# Floating point registers names like: FRT, FRA, FR1, ..., FRTp, ...
# Return True if field is a register
```

```
def isreg(field):
    return (field.startswith('R') or field.startswith('FR') or
            field == 'SPR')


# These are the attributes of the instructions,
# register names
keycolumns = ['unit', 'in1', 'in2', 'in3', 'out', 'CR in', 'CR out',
              ]  # don't think we need these: 'ldst len', 'rc', 'lk']

tablecols = ['unit', 'in', 'outcnt', 'CR in', 'CR out', 'imm'
             ]  # don't think we need these: 'ldst len', 'rc', 'lk']


def create_key(row):
    """ create an equivalent of a database key by which it is possible
    to easily categorise an instruction.  later this category is used
    to decide what kind of EXTRA encoding is to be done because the
    key contains the total number of input and output registers
    """
    res = OrderedDict()
    #print ("row", row)
    for key in keycolumns:
        # registers IN - special-case: count number of regs RA/RB/RC/RS
        if key in ['in1', 'in2', 'in3']:
            if 'in' not in res:
                res['in'] = 0
            if row['unit'] == 'BRANCH':  # branches must not include Vector SPRs
                continue
            if isreg(row[key]):
                res['in'] += 1

        # registers OUT
        if key == 'out':
            # If upd is 1 then increment the count of outputs
            if 'outcnt' not in res:
                res['outcnt'] = 0
            if isreg(row[key]):
                res['outcnt'] += 1
            if row['upd'] == '1':
                res['outcnt'] += 1

        # CRs (Condition Register) (CR0 .. CR7)
        if key.startswith('CR'):
            if row[key].startswith('NONE'):
                res[key] = '0'
            else:
                res[key] = '1'
            if row['comment'].startswith('cr'):
                res['crop'] = '1'
        # unit
        if key == 'unit':
            if row[key] == 'LDST':  # we care about LDST units
                res[key] = row[key]
            else:
```

```
                res[key] = 'OTHER'
            # LDST len (LoadStore length)
            if key.startswith('ldst'):
                if row[key].startswith('NONE'):
                    res[key] = '0'
                else:
                    res[key] = '1'
            # rc, lk
            if key in ['rc', 'lk']:
                if row[key] == 'ONE':
                    res[key] = '1'
                elif row[key] == 'NONE':
                    res[key] = '0'
                else:
                    res[key] = 'R'
            if key == 'lk':
                res[key] = row[key]

        # Convert the numerics 'in' & 'outcnt' to strings
        res['in'] = str(res['in'])
        res['outcnt'] = str(res['outcnt'])

        # constants
        if row['in2'].startswith('CONST_'):
            res['imm'] = "1"  # row['in2'].split("_")[1]
        else:
            res['imm'] = ''

        return res

    #


    def dformat(d):
        res = []
        for k, v in d.items():
            res.append("%s: %s" % (k, v))
        return ' '.join(res)


    def tformat(d):
        return "| " + ' | '.join(d) + " |"


    def keyname(row):
        """converts a key into a readable string. anything null or zero
        is skipped, shortening the readable string
        """
        res = []
        if row['unit'] != 'OTHER':
            res.append(row['unit'])
        if row['in'] != '0':
            res.append('%sR' % row['in'])
        if row['outcnt'] != '0':
            res.append('%sW' % row['outcnt'])
```

```
        if row['CR in'] == '1' and row['CR out'] == '1':
            if 'crop' in row:
                res.append("CR=2R1W")
            else:
                res.append("CRio")
        elif row['CR in'] == '1':
            res.append("CRi")
        elif row['CR out'] == '1':
            res.append("CRo")
        elif 'imm' in row and row['imm']:
            res.append("imm")
        return '-'.join(res)


    class Format(enum.Enum):
        BINUTILS = enum.auto()
        VHDL = enum.auto()

        @classmethod
        def _missing_(cls, value):
            return {
                "binutils": Format.BINUTILS,
                "vhdl": Format.VHDL,
            }[value.lower()]

        def __str__(self):
            return self.name.lower()

        def declarations(self, values, lens):
            def declaration_binutils(value, width):
                yield f"/* TODO: implement binutils declaration (value={value!r}, width={width!r}) */"

            def declaration_vhdl(value, width):
                yield f"    type sv_{value}_rom_array_t is " \
                    f"array(0 to {width}) of sv_decode_rom_t;"

            for value in values:
                if value not in lens:
                    todo = [f"TODO {value} (or no SVP64 augmentation)"]
                    todo = self.wrap_comment(todo)
                    yield from map(lambda line: f"    {line}", todo)
                else:
                    width = lens[value]
                    yield from {
                        Format.BINUTILS: declaration_binutils,
                        Format.VHDL: declaration_vhdl,
                    }[self](value, width)

        def definitions(self, entries_svp64, fullcols):
            def definitions_vhdl():
                for (value, entries) in entries_svp64.items():
                    yield ""
                    yield f"    constant sv_{value}_decode_rom_array :"
                    yield f"                sv_{value}_rom_array_t := ("
                    yield f"        -- {'  '.join(fullcols)}"
```

```
                for (op, insn, row) in entries:
                    yield f"    {op:>13} => ({', '.join(row)}), -- {insn}"

                yield f"    {'others':>13} => sv_illegal_inst"
                yield "    );"
                yield ""

            def definitions_binutils():
                yield f"/* TODO: implement binutils definitions */"

            yield from {
                Format.BINUTILS: definitions_binutils,
                Format.VHDL: definitions_vhdl,
            }[self]()

        def wrap_comment(self, lines):
            def wrap_comment_binutils(lines):
                lines = tuple(lines)
                if len(lines) == 1:
                    yield f"/* {lines[0]} */"
                else:
                    yield "/*"
                    yield from map(lambda line: f" * {line}", lines)
                    yield " */"

            def wrap_comment_vhdl(lines):
                yield from map(lambda line: f"-- {line}", lines)

            yield from {
                Format.BINUTILS: wrap_comment_binutils,
                Format.VHDL: wrap_comment_vhdl,
            }[self](lines)


    def read_csvs():
        csvs = {}
        csvs_svp64 = {}
        bykey = {}
        primarykeys = set()
        dictkeys = OrderedDict()
        immediates = {}
        insns = {}  # dictionary of CSV row, by instruction
        insn_to_csv = {}

        # Expand that (all .csv files)
        pth = find_wiki_file("*.csv")

        # Ignore those containing: valid test sprs
        for fname in glob_valid_csvs(pth):
            csvname = os.path.split(fname)[1]
            csvname_ = csvname.split(".")[0]
            # csvname is something like: minor_59.csv, fname the whole path
            csv = get_csv(fname)
            csvs[fname] = csv
```

```
            csvs_svp64[csvname_] = []
            for row in csv:
                if blank_key(row):
                    continue
                #print("row", row)
                insn_name = row['comment']
                condition = row['CONDITIONS']
                # skip instructions that are not suitable
                if insn_name.startswith("l") and insn_name.endswith("br"):
                    continue  # skip pseudo-alias lxxxbr
                if insn_name in ['mcrxr', 'mcrxrx', 'darn']:
                    continue
                if insn_name in ['bctar', 'bcctr']:  # for now. TODO
                    continue
                if 'rfid' in insn_name:
                    continue
                if 'addpcis' in insn_name:  # skip for now
                    continue

                # sv.bc is being classified as 2P-2S-1D by mistake due to SPRs
                if insn_name.startswith('bc'):
                    # whoops: remove out reg (SPRs CTR etc)
                    row['in1'] = 'NONE'
                    row['in2'] = 'NONE'
                    row['in3'] = 'NONE'
                    row['out'] = 'NONE'

                insns[(insn_name, condition)] = row  # accumulate csv data
                insn_to_csv[insn_name] = csvname_  # CSV file name by instruction
                dkey = create_key(row)
                key = tuple(dkey.values())
                #print("key=", key, dkey)
                dictkeys[key] = dkey
                primarykeys.add(key)
                if key not in bykey:
                    bykey[key] = []
                bykey[key].append((csvname, row['opcode'], insn_name, condition,
                                   row['form'].upper() + '-Form'))

                # detect immediates, collate them (useful info)
                if row['in2'].startswith('CONST_'):
                    imm = row['in2'].split("_")[1]
                    if key not in immediates:
                        immediates[key] = set()
                    immediates[key].add(imm)

        primarykeys = list(primarykeys)
        primarykeys.sort()

        return (csvs, csvs_svp64, primarykeys, bykey, insn_to_csv, insns,
                dictkeys, immediates)


    def regs_profile(insn, res):
        """get a more detailed register profile: 1st operand is RA,
```

```
        2nd is RB, etc. etc
        """
        regs = []
        for k in ['in1', 'in2', 'in3', 'out', 'CR in', 'CR out']:
            if insn[k].startswith('CONST'):
                res[k] = ''
                regs.append('')
            else:
                res[k] = insn[k]
                if insn[k] == 'RA_OR_ZERO':
                    regs.append('RA')
                elif insn[k] != 'NONE':
                    regs.append(insn[k])
                else:
                    regs.append('')
        return regs


    def extra_classifier(insn_name, value, name, res, regs):
        """extra_classifier: creates the SVP64.RM EXTRA2/3 classification.
        there is very little space (9 bits) to mark register operands
        (RT RA RB, BA BB, BFA, FRS etc.) with the "extra" information
        needed to tell if *EACH* operand (of which there can be up to five!)
        is Vectorised, and whether its numbering is extended into the
        0..127 range rather than the limited 3/5 bit of Scalar v3.0 Power ISA.

        thus begins the rather tedious but by-rote examination of EVERY
        Scalar instruction, working out how best to tell a decoder how to
        extend the registers.  EXTRA2 can have up to 4 slots (of 2 bit each)
        where due to RM.EXTRA being 9 bits, EXTRA3 can have up to 3 slots
        (of 3 bit each).  the index REGNAME says which slot the register
        named REGNAME must read its decoding from.  d: means destination,
        s: means source.  some are *shared slots* especially LDST update.
        some Rc=1 ops have the CR0/CR1 as a co-result which is also
        obviously Vectorised if the result is Vectorised.

        it is actually quite straightforward but the sheer quantity of
        Scalar Power ISA instructions made it prudent to do this in an
        intelligent way, almost by-rote, by analysing the register profiles.
        """
        # for LD/ST FP, use FRT/FRS not RT/RS, and use CR1 not CR0
        if insn_name.startswith("lf"):
            dRT = 'd:FRT'
            dCR = 'd:CR1'
        else:
            dRT = 'd:RT'
            dCR = 'd:CR0'
        if insn_name.startswith("stf"):
            sRS = 's:FRS'
            dCR = 'd:CR1'
        else:
            sRS = 's:RS'
            dCR = 'd:CR0'

        # sigh now the fun begins.  this isn't the sanest way to do it
```

```
# but the patterns are pretty regular.  we start with the "profile"
# because that determines how much space is available (total num
# regs to decode) then if necessary begin apecialising either
# by the instruction name or through more detailed register
# profiling. example:
#     if regs == ['RA', '', '', 'RT', '', '']:
# is in the order in1  in2  in3 out1 out2 Rc=1

# ********
# start with LD/ST

if value == 'LDSTRM-2P-1S1D':
    res['Etype'] = 'EXTRA3'  # RM EXTRA3 type
    res['0'] = dRT    # RT: Rdest_EXTRA3
    res['1'] = 's:RA'  # RA: Rsrc1_EXTRA3

elif value == 'LDSTRM-2P-1S2D':
    res['Etype'] = 'EXTRA2'  # RM EXTRA2 type
    res['0'] = dRT    # RT: Rdest_EXTRA3
    res['1'] = 'd:RA'  # RA: Rdest2_EXTRA2
    res['2'] = 's:RA'  # RA: Rsrc1_EXTRA2

elif value == 'LDSTRM-2P-2S':
    # stw, std, sth, stb
    res['Etype'] = 'EXTRA3'  # RM EXTRA3 type
    res['0'] = sRS    # RS: Rdest1_EXTRA3
    res['1'] = 's:RA'  # RA: Rsrc1_EXTRA3

elif value == 'LDSTRM-2P-2S1D':
    if 'st' in insn_name and 'x' not in insn_name:  # stwu/stbu etc
        res['Etype'] = 'EXTRA2'  # RM EXTRA2 type
        res['0'] = 'd:RA'  # RA: Rdest1_EXTRA2
        res['1'] = sRS    # RS: Rdsrc1_EXTRA2
        res['2'] = 's:RA'  # RA: Rsrc2_EXTRA2
    elif 'st' in insn_name and 'x' in insn_name:  # stwux
        res['Etype'] = 'EXTRA2'  # RM EXTRA2 type
        res['0'] = 'd:RA'  # RA: Rdest1_EXTRA2
        # RS: Rdest2_EXTRA2, RA: Rsrc1_EXTRA2
        res['1'] = "%s;%s" % (sRS, 's:RA')
        res['2'] = 's:RB'  # RB: Rsrc2_EXTRA2
    elif 'u' in insn_name:  # ldux etc.
        res['Etype'] = 'EXTRA2'  # RM EXTRA2 type
        res['0'] = dRT    # RT: Rdest1_EXTRA2
        res['1'] = 'd:RA'  # RA: Rdest2_EXTRA2
        res['2'] = 's:RB'  # RB: Rsrc1_EXTRA2
    else:
        res['Etype'] = 'EXTRA2'  # RM EXTRA2 type
        res['0'] = dRT     # RT: Rdest1_EXTRA2
        res['1'] = 's:RA'  # RA: Rsrc1_EXTRA2
        res['2'] = 's:RB'  # RB: Rsrc2_EXTRA2

elif value == 'LDSTRM-2P-3S':
    res['Etype'] = 'EXTRA2'  # RM EXTRA2 type
    if 'cx' in insn_name:
        res['0'] = "%s;%s" % (sRS, dCR)  # RS: Rsrc1_EXTRA2 CR0: dest
```

```
        else:
            res['0'] = sRS  # RS: Rsrc1_EXTRA2
        res['1'] = 's:RA'  # RA: Rsrc2_EXTRA2
        res['2'] = 's:RB'  # RA: Rsrc3_EXTRA2

    # **********
    # now begins,arithmetic

    elif value == 'RM-2P-1S1D':
        res['Etype'] = 'EXTRA3'  # RM EXTRA3 type
        if insn_name == 'mtspr':
            res['0'] = 'd:SPR'  # SPR: Rdest1_EXTRA3
            res['1'] = 's:RS'  # RS: Rsrc1_EXTRA3
        elif insn_name == 'mfspr':
            res['0'] = 'd:RS'  # RS: Rdest1_EXTRA3
            res['1'] = 's:SPR'  # SPR: Rsrc1_EXTRA3
        elif name == 'CRio' and insn_name == 'mcrf':
            res['0'] = 'd:BF'  # BFA: Rdest1_EXTRA3
            res['1'] = 's:BFA'  # BFA: Rsrc1_EXTRA3
        elif 'mfcr' in insn_name or 'mfocrf' in insn_name:
            res['0'] = 'd:RT'  # RT: Rdest1_EXTRA3
            res['1'] = 's:CR'  # CR: Rsrc1_EXTRA3
        elif insn_name == 'setb':
            res['0'] = 'd:RT'  # RT: Rdest1_EXTRA3
            res['1'] = 's:BFA'  # BFA: Rsrc1_EXTRA3
        elif insn_name.startswith('cmp'):  # cmpi
            res['0'] = 'd:BF'  # BF: Rdest1_EXTRA3
            res['1'] = 's:RA'  # RA: Rsrc1_EXTRA3
        elif regs == ['RA', '', '', 'RT', '', '']:
            res['0'] = 'd:RT'  # RT: Rdest1_EXTRA3
            res['1'] = 's:RA'  # RA: Rsrc1_EXTRA3
        elif regs == ['RA', '', '', 'RT', '', 'CR0']:
            res['0'] = 'd:RT;d:CR0'  # RT,CR0: Rdest1_EXTRA3
            res['1'] = 's:RA'  # RA: Rsrc1_EXTRA3
        elif (regs == ['RS', '', '', 'RA', '', 'CR0'] or
              regs == ['', '', 'RS', 'RA', '', 'CR0']):
            res['0'] = 'd:RA;d:CR0'  # RA,CR0: Rdest1_EXTRA3
            res['1'] = 's:RS'  # RS: Rsrc1_EXTRA3
        elif regs == ['RS', '', '', 'RA', '', '']:
            res['0'] = 'd:RA'  # RA: Rdest1_EXTRA3
            res['1'] = 's:RS'  # RS: Rsrc1_EXTRA3
        elif regs == ['', 'FRB', '', 'FRT', '0', 'CR1']:
            res['0'] = 'd:FRT;d:CR1'  # FRT,CR1: Rdest1_EXTRA3
            res['1'] = 's:FRA'  # FRA: Rsrc1_EXTRA3
        elif regs == ['', 'FRB', '', '', '', 'CR1']:
            res['0'] = 'd:CR1'  # CR1: Rdest1_EXTRA3
            res['1'] = 's:FRB'  # FRA: Rsrc1_EXTRA3
        elif regs == ['', 'FRB', '', '', '', 'BF']:
            res['0'] = 'd:BF'  # BF: Rdest1_EXTRA3
            res['1'] = 's:FRB'  # FRA: Rsrc1_EXTRA3
        elif regs == ['', 'FRB', '', 'FRT', '', 'CR1']:
            res['0'] = 'd:FRT;d:CR1'  # FRT,CR1: Rdest1_EXTRA3
            res['1'] = 's:FRB'  # FRB: Rsrc1_EXTRA3
        elif insn_name == 'fishmv':
            # an overwrite instruction
```

```
            res['0'] = 'd:FRS'  # FRS: Rdest1_EXTRA3
            res['1'] = 's:FRS'  # FRS: Rsrc1_EXTRA3
        elif insn_name == 'setvl':
            res['0'] = 'd:RT'  # RT: Rdest1_EXTRA3
            res['1'] = 's:RA'  # RS: Rsrc1_EXTRA3
        else:
            res['0'] = 'TODO'
            print("regs TODO", insn_name, regs)

    elif value == 'RM-1P-2S1D':
        res['Etype'] = 'EXTRA3'  # RM EXTRA3 type
        if insn_name.startswith('cr'):
            res['0'] = 'd:BT'  # BT: Rdest1_EXTRA3
            res['1'] = 's:BA'  # BA: Rsrc1_EXTRA3
            res['2'] = 's:BB'  # BB: Rsrc2_EXTRA3
        elif regs == ['FRA', '', 'FRC', 'FRT', '', 'CR1']:
            res['0'] = 'd:FRT;d:CR1'  # FRT,CR1: Rdest1_EXTRA3
            res['1'] = 's:FRA'  # FRA: Rsrc1_EXTRA3
            res['2'] = 's:FRC'  # FRC: Rsrc1_EXTRA3
        # should be for fcmp
        elif regs == ['FRA', 'FRB', '', '', '', 'BF']:
            res['0'] = 'd:BF'  # BF: Rdest1_EXTRA3
            res['1'] = 's:FRA'  # FRA: Rsrc1_EXTRA3
            res['2'] = 's:FRB'  # FRB: Rsrc1_EXTRA3
        elif regs == ['FRA', 'FRB', '', 'FRT', '', '']:
            res['0'] = 'd:FRT'  # FRT: Rdest1_EXTRA3
            res['1'] = 's:FRA'  # FRA: Rsrc1_EXTRA3
            res['2'] = 's:FRB'  # FRB: Rsrc1_EXTRA3
        elif regs == ['FRA', 'FRB', '', 'FRT', '', 'CR1']:
            res['0'] = 'd:FRT;d:CR1'  # FRT,CR1: Rdest1_EXTRA3
            res['1'] = 's:FRA'  # FRA: Rsrc1_EXTRA3
            res['2'] = 's:FRB'  # FRB: Rsrc1_EXTRA3
        elif regs == ['FRA', 'RB', '', 'FRT', '', 'CR1']:
            res['0'] = 'd:FRT;d:CR1'  # FRT,CR1: Rdest1_EXTRA3
            res['1'] = 's:FRA'  # FRA: Rsrc1_EXTRA3
            res['2'] = 's:RB'  # RB: Rsrc1_EXTRA3
        elif name == '2R-1W' or insn_name == 'cmpb':  # cmpb
            if insn_name in ['bpermd', 'cmpb']:
                res['0'] = 'd:RA'  # RA: Rdest1_EXTRA3
                res['1'] = 's:RS'  # RS: Rsrc1_EXTRA3
            else:
                res['0'] = 'd:RT'  # RT: Rdest1_EXTRA3
                res['1'] = 's:RA'  # RA: Rsrc1_EXTRA3
            res['2'] = 's:RB'  # RB: Rsrc1_EXTRA3
        elif insn_name.startswith('cmp'):  # cmp
            res['0'] = 'd:BF'  # BF: Rdest1_EXTRA3
            res['1'] = 's:RA'  # RA: Rsrc1_EXTRA3
            res['2'] = 's:RB'  # RB: Rsrc1_EXTRA3
        elif (regs == ['', 'RB', 'RS', 'RA', '', 'CR0'] or
              regs == ['RS', 'RB', '', 'RA', '', 'CR0']):
            res['0'] = 'd:RA;d:CR0'  # RA,CR0: Rdest1_EXTRA3
            res['1'] = 's:RB'  # RB: Rsrc1_EXTRA3
            res['2'] = 's:RS'  # RS: Rsrc1_EXTRA3
        elif regs == ['RA', 'RB', '', 'RT', '', 'CR0']:
            res['0'] = 'd:RT;d:CR0'  # RT,CR0: Rdest1_EXTRA3
```

```
            res['1'] = 's:RA'  # RA: Rsrc1_EXTRA3
            res['2'] = 's:RB'  # RB: Rsrc1_EXTRA3
        elif regs == ['RA', '', 'RS', 'RA', '', 'CR0']:
            res['0'] = 'd:RA;d:CR0'  # RA,CR0: Rdest1_EXTRA3
            res['1'] = 's:RA'  # RA: Rsrc1_EXTRA3
            res['2'] = 's:RS'  # RS: Rsrc1_EXTRA3
        else:
            res['0'] = 'TODO'

    elif value == 'RM-2P-2S1D':
        res['Etype'] = 'EXTRA2'  # RM EXTRA2 type
        if insn_name.startswith('mt'):  # mtcrf
            res['0'] = 'd:CR'  # CR: Rdest1_EXTRA2
            res['1'] = 's:RS'  # RS: Rsrc1_EXTRA2
            res['2'] = 's:CR'  # CR: Rsrc2_EXTRA2
        else:
            res['0'] = 'TODO'

    elif value == 'RM-1P-3S1D':
        res['Etype'] = 'EXTRA2'  # RM EXTRA2 type
        if regs == ['FRT', 'FRB', 'FRA', 'FRT', '', 'CR1']:  # ffmadds/fdmadds
            res['0'] = 'd:FRT;d:CR1'  # FRT,CR1: Rdest1_EXTRA2
            res['1'] = 's:FRT'  # FRT: Rsrc1_EXTRA2
            res['2'] = 's:FRB'  # FRB: Rsrc2_EXTRA2
            res['3'] = 's:FRA'  # FRA: Rsrc3_EXTRA2
        elif regs == ['RA', 'RB', 'RC', 'RT', '', '']:  # madd*
            res['0'] = 'd:RT'  # RT,CR0: Rdest1_EXTRA2
            res['1'] = 's:RA'  # RA: Rsrc1_EXTRA2
            res['2'] = 's:RB'  # RT: Rsrc2_EXTRA2
            res['3'] = 's:RC'  # RT: Rsrc3_EXTRA2
        elif regs == ['RA', 'RB', 'RC', 'RT', '', 'CR0']:  # pcdec
            res['0'] = 'd:RT;d:CR0'  # RT,CR0: Rdest1_EXTRA2
            res['1'] = 's:RA'  # RA: Rsrc1_EXTRA2
            res['2'] = 's:RB'  # RT: Rsrc2_EXTRA2
            res['3'] = 's:RC'  # RT: Rsrc3_EXTRA2
        elif regs == ['RA', 'RB', 'RT', 'RT', '', 'CR0']:  # overwrite 3-in
            res['0'] = 'd:RT;d:CR0'  # RT,CR0: Rdest1_EXTRA2
            res['1'] = 's:RA'  # RA: Rsrc1_EXTRA2
            res['2'] = 's:RB'  # RT: Rsrc2_EXTRA2
            res['3'] = 's:RT'  # RT: Rsrc3_EXTRA2
        elif regs == ['RA', 'RB', 'RT', 'RT', '', '']:  # maddsubrs
            res['0'] = 'd:RT'  # RT: Rdest1_EXTRA2
            res['1'] = 's:RA'  # RA: Rsrc1_EXTRA2
            res['2'] = 's:RB'  # RT: Rsrc2_EXTRA2
            res['3'] = 's:RT'  # RT: Rsrc3_EXTRA2
        elif insn_name == 'isel':
            res['0'] = 'd:RT'  # RT: Rdest1_EXTRA2
            res['1'] = 's:RA'  # RA: Rsrc1_EXTRA2
            res['2'] = 's:RB'  # RT: Rsrc2_EXTRA2
            res['3'] = 's:BC'  # BC: Rsrc3_EXTRA2
        else:  # fmadd*
            res['0'] = 'd:FRT;d:CR1'  # FRT, CR1: Rdest1_EXTRA2
            res['1'] = 's:FRA'  # FRA: Rsrc1_EXTRA2
            res['2'] = 's:FRB'  # FRB: Rsrc2_EXTRA2
            res['3'] = 's:FRC'  # FRC: Rsrc3_EXTRA2
```

```
        elif value == 'RM-1P-1D':
            res['Etype'] = 'EXTRA3'  # RM EXTRA3 type
            if insn_name == 'svstep':
                res['0'] = 'd:RT;d:CR0'  # RT,CR0: Rdest1_EXTRA3
            if insn_name == 'fmvis':
                res['0'] = 'd:FRS'  # FRS: Rdest1_EXTRA3

        # HACK! thos should be RM-1P-1S butvthere is a bug with sv.bc
        elif value == 'RM-2P-1S':
            res['Etype'] = 'EXTRA3'  # RM EXTRA3 type
            if insn_name.startswith('bc'):
                res['0'] = 's:BI'  # BI: Rsrc1_EXTRA3


    def process_csvs(format):

        print("# Draft SVP64 Power ISA register 'profile's")
        print('')
        print("this page is auto-generated, do not edit")
        print("created by http://libre-soc.org/openpower/sv_analysis.py")
        print('')

        (csvs, csvs_svp64, primarykeys, bykey, insn_to_csv, insns,
         dictkeys, immediates) = read_csvs()

        # mapping to old SVPrefix "Forms"
        mapsto = {'3R-1W-CRo': 'RM-1P-3S1D',
                  '3R-1W': 'RM-1P-3S1D',
                  '2R-1W-CRio': 'RM-1P-2S1D',
                  '2R-1W-CRi': 'RM-1P-3S1D',
                  '2R-1W-CRo': 'RM-1P-2S1D',
                  '2R': 'non-SV',
                  '2R-1W': 'RM-1P-2S1D',
                  '1R-CRio': 'RM-2P-2S1D',
                  '2R-CRio': 'RM-1P-2S1D',
                  '2R-CRo': 'RM-1P-2S1D',
                  '1R': 'non-SV',
                  '1R-1W-CRio': 'RM-2P-1S1D',
                  '1R-1W-CRo': 'RM-2P-1S1D',
                  '1R-1W': 'RM-2P-1S1D',
                  '1R-1W-imm': 'RM-2P-1S1D',
                  '1R-CRo': 'RM-2P-1S1D',
                  '1R-imm': 'RM-1P-1S',
                  '1W-CRo': 'RM-1P-1D',
                  '1W': 'non-SV',
                  '1W-imm': 'RM-1P-1D',
                  '1W-CRi': 'RM-2P-1S1D',
                  'CRio': 'RM-2P-1S1D',
                  'CR=2R1W': 'RM-1P-2S1D',
                  'CRi': 'RM-2P-1S',  # HACK, bc here, it should be 1P
                  'imm': 'non-SV',
                  '': 'non-SV',
                  'LDST-2R-imm': 'LDSTRM-2P-2S',
                  'LDST-2R-1W-imm': 'LDSTRM-2P-2S1D',
```

```
            'LDST-2R-1W': 'LDSTRM-2P-2S1D',
            'LDST-2R-2W': 'LDSTRM-2P-2S1D',
            'LDST-1R-1W-imm': 'LDSTRM-2P-1S1D',
            'LDST-1R-2W-imm': 'LDSTRM-2P-1S2D',
            'LDST-3R': 'LDSTRM-2P-3S',
            'LDST-3R-CRo': 'LDSTRM-2P-3S',  # st*x
            'LDST-3R-1W': 'LDSTRM-2P-2S1D',  # st*x
            }
print("# map to old SV Prefix")
print('')
print('|internal key | public name |')
print('|-----        | ----------  |')
for key in primarykeys:
    name = keyname(dictkeys[key])
    value = mapsto.get(name, "-")
    print(tformat([name, value + " "]))
print('')
print('')

print("# keys")
print('')
print(tformat(tablecols) + " imms | name |")
print(tformat([" - "] * (len(tablecols)+2)))

# print out the keys and the table from which they're derived
for key in primarykeys:
    name = keyname(dictkeys[key])
    row = tformat(dictkeys[key].values())
    imms = list(immediates.get(key, ""))
    imms.sort()
    row += " %s | " % ("/".join(imms))
    row += " %s |" % name
    print(row)
print('')
print('')

# print out, by remap name, all the instructions under that category
for key in primarykeys:
    name = keyname(dictkeys[key])
    value = mapsto.get(name, "-")
    print("## %s (%s)" % (name, value))
    print('')
    print(tformat(['CSV', 'opcode', 'asm', 'flags', 'form']))
    print(tformat(['---', '------', '---', '-----', '----']))
    rows = bykey[key]
    rows.sort()
    for row in rows:
        print(tformat(row))
    print('')
    print('')

# for fname, csv in csvs.items():
#    print (fname)

# for insn, row in insns.items():
```

```
#    print (insn, row)

print("# svp64 remaps")
svp64 = OrderedDict()
# create a CSV file, per category, with SV "augmentation" info
# XXX note: 'out2' not added here, needs to be added to CSV files
# KEEP TRACK OF THESE https://bugs.libre-soc.org/show_bug.cgi?id=619
csvcols = ['insn', 'mode', 'CONDITIONS', 'Ptype', 'Etype', 'SM']
csvcols += ['0', '1', '2', '3']
csvcols += ['in1', 'in2', 'in3', 'out', 'CR in', 'CR out']  # temporary
for key in primarykeys:
    # get the decoded key containing row-analysis, and name/value
    dkey = dictkeys[key]
    name = keyname(dkey)
    value = mapsto.get(name, "-")
    if value == 'non-SV':
        continue

    # print out svp64 tables by category
    print("* **%s**: %s" % (name, value))

    # store csv entries by svp64 RM category
    if value not in svp64:
        svp64[value] = []

    rows = bykey[key]
    rows.sort()

    for row in rows:
        # for idx in range(len(row)):
        #    if row[idx] == 'NONE':
        #        row[idx] = ''
        # get the instruction
        #print(key, row)
        insn_name = row[2]
        condition = row[3]
        insn = insns[(insn_name, condition)]

        # start constructing svp64 CSV row
        res = OrderedDict()
        res['insn'] = insn_name
        res['CONDITIONS'] = condition
        res['Ptype'] = value.split('-')[1]  # predication type (RM-xN-xxx)
        # get whether R_xxx_EXTRAn fields are 2-bit or 3-bit
        res['Etype'] = 'EXTRA2'
        # go through each register matching to Rxxxx_EXTRAx
        for k in ['0', '1', '2', '3']:
            res[k] = ''
        # create "fake" out2 (TODO, needs to be added to CSV files)
        # KEEP TRACK HERE https://bugs.libre-soc.org/show_bug.cgi?id=619
        res['out2'] = 'NONE'
        if insn['upd'] == '1':  # LD/ST with update has RA as out2
            res['out2'] = 'RA'

        # set the SVP64 mode to NORMAL, LDST, BRANCH or CR
```

```
            crops = ['mfcr', 'mfocrf', 'mtcrf', 'mtocrf',
                     ]
            mode = 'NORMAL'
            if value.startswith('LDST'):
                if 'x' in insn_name:  # Indexed detection
                    mode = 'LDST_IDX'
                else:
                    mode = 'LDST_IMM'
            elif insn_name.startswith('bc'):
                mode = 'BRANCH'
            elif insn_name.startswith('cmp') or insn_name.startswith('cr') or insn_name in crops:
                mode = 'CROP'
            res['mode'] = mode

            # create a register profile list (update res row as well)
            regs = regs_profile(insn, res)

            #print("regs", insn_name, regs)
            extra_classifier(insn_name, value, name, res, regs)

            # source-mask is hard to detect, it's part of RM-nn-nn.
            # to make style easier, create a yes/no decision here
            # see https://libre-soc.org/openpower/sv/svp64/#extra_remap
            # MASK_SRC
            vstripped = value.replace("LDST", "")
            if vstripped in ['RM-2P-1S1D', 'RM-2P-2S',
                             'RM-2P-2S1D', 'RM-2P-1S2D', 'RM-2P-3S',
                             ]:
                res['SM'] = 'EN'
            else:
                res['SM'] = 'NO'
            # add to svp64 csvs
            # for k in ['in1', 'in2', 'in3', 'out', 'CR in', 'CR out']:
            #    del res[k]
            # if res['0'] != 'TODO':
            for k in res:
                if k == 'CONDITIONS':
                    continue
                if res[k] == 'NONE' or res[k] == '':
                    res[k] = '0'
            svp64[value].append(res)
            # also add to by-CSV version
            csv_fname = insn_to_csv[insn_name]
            csvs_svp64[csv_fname].append(res)

    print('')

    # now write out the csv files
    for value, csv in svp64.items():
        if value == '-':
            continue
            from time import sleep
            print("WARNING, filename '-' should NOT exist. instrs missing")
            print("TODO: fix this (and put in the bugreport number here)")
            sleep(2)
```

```
            # print out svp64 tables by category
            print("## %s" % value)
            print('')
            cols = csvcols + ['out2']
            print(tformat(cols))
            print(tformat([" - "] * (len(cols))))
            for d in csv:
                row = []
                for k in cols:
                    row.append(d[k])
                print(tformat(row))
            print('')

            #csvcols = ['insn', 'Ptype', 'Etype', '0', '1', '2', '3']
            write_csv("%s.csv" % value, csv, csvcols + ['out2'])

        # okaaay, now we re-read them back in for producing microwatt SV

        # get SVP64 augmented CSV files
        svt = SVP64RM(microwatt_format=True)
        # Expand that (all .csv files)
        pth = find_wiki_file("*.csv")

        # Ignore those containing: valid test sprs
        for fname in glob_valid_csvs(pth):
            svp64_csv = svt.get_svp64_csv(fname)

        csvcols = ['insn', 'mode', 'Ptype', 'Etype', 'SM']
        csvcols += ['in1', 'in2', 'in3', 'out', 'out2', 'CR in', 'CR out']

        if format is Format.VHDL:
            # and a nice microwatt VHDL file
            file_path = find_wiki_file("sv_decode.vhdl")
        elif format is Format.BINUTILS:
            file_path = find_wiki_file("binutils.c")

        with open(file_path, 'w') as stream:
            output(format, svt, csvcols, insns, csvs_svp64, stream)


    def output_autogen_disclaimer(format, stream):
        lines = (
            "this file is auto-generated, do not edit",
            "http://libre-soc.org/openpower/sv_analysis.py",
            "part of Libre-SOC, sponsored by NLnet",
        )
        for line in format.wrap_comment(lines):
            stream.write(line)
            stream.write("\n")
        stream.write("\n")


    def output(format, svt, csvcols, insns, csvs_svp64, stream):
        lens = {
            'major': 63,
```

```
    'minor_4': 63,
    'minor_19': 7,
    'minor_30': 15,
    'minor_31': 1023,
    'minor_58': 63,
    'minor_59': 31,
    'minor_62': 63,
    'minor_63l': 511,
    'minor_63h': 16,
}

def svp64_canonicalize(item):
    (value, csv) = item
    value = value.lower().replace("-", "_")
    return (value, csv)

csvs_svp64_canon = dict(map(svp64_canonicalize, csvs_svp64.items()))

# disclaimer
output_autogen_disclaimer(format, stream)

# declarations
for line in format.declarations(csvs_svp64_canon.keys(), lens):
    stream.write(f"{line}\n")

# definitions
sv_cols = ['sv_in1', 'sv_in2', 'sv_in3', 'sv_out', 'sv_out2',
           'sv_cr_in', 'sv_cr_out']
fullcols = csvcols + sv_cols

entries_svp64 = defaultdict(list)
for (value, csv) in filter(lambda kv: kv[0] in lens, csvs_svp64_canon.items()):
    for entry in csv:
        insn = str(entry['insn'])
        condition = str(entry['CONDITIONS'])
        mode = str(entry['mode'])
        sventry = svt.svp64_instrs.get(insn, None)
        if sventry is not None:
            sventry['mode'] = mode
        op = insns[(insn, condition)]['opcode']
        # binary-to-vhdl-binary
        if op.startswith("0b"):
            op = "2#%s#" % op[2:]
        row = []
        for colname in csvcols[1:]:
            re = entry[colname]
            # zero replace with NONE
            if re == '0':
                re = 'NONE'
            # 1/2 predication
            re = re.replace("1P", "P1")
            re = re.replace("2P", "P2")
            row.append(re)
        #print("sventry", sventry)
        for colname in sv_cols:
```

```
                if sventry is None:
                    re = 'NONE'
                else:
                    re = sventry[colname]
                row.append(re)
            entries_svp64[value].append((op, insn, row))

    for line in format.definitions(entries_svp64, fullcols):
        stream.write(f"{line}\n")


def main():
    import os
    os.environ['SILENCELOG'] = '1'
    parser = argparse.ArgumentParser()
    parser.add_argument("-f", "--format",
                        type=Format, choices=Format, default=Format.VHDL,
                        help="format to be used (binutils or VHDL)")
    args = parser.parse_args()
    process_csvs(args.format)


if __name__ == '__main__':
    # don't do anything other than call main() here, cuz this code is bypassed
    # by the sv_analysis command created by setup.py
    main()
```

Appendix F

SVP64 Augmentation Table

F.1 Draft SVP64 Power ISA register 'profile's

this page is auto-generated, do not edit created by http://libre-soc.org/openpower/sv_analysis.py

F.2 map to old SV Prefix

internal key	public name
LDST-1R-1W-imm	LDSTRM-2P-1S1D
LDST-1R-2W-imm	LDSTRM-2P-1S2D
LDST-2R	-
LDST-2R-imm	LDSTRM-2P-2S
LDST-2R-1W	LDSTRM-2P-2S1D
LDST-2R-1W-imm	LDSTRM-2P-2S1D
LDST-2R-2W	LDSTRM-2P-2S1D
LDST-2R-2W-imm	-
LDST-3R	LDSTRM-2P-3S
LDST-3R-CRo	LDSTRM-2P-3S
LDST-3R-1W	LDSTRM-2P-2S1D
	non-SV
imm	non-SV
CRo	-
CRio	RM-2P-1S1D
CR=2R1W	RM-1P-2S1D
1W	non-SV
1W-imm	RM-1P-1D
1W-CRo	RM-1P-1D
1W-CRi	RM-2P-1S1D
1W-CRi	RM-2P-1S1D
1R	non-SV
1R-imm	RM-1P-1S
1R-CRo	RM-2P-1S1D
1R-CRo	RM-2P-1S1D
1R-CRio	RM-2P-2S1D
1R-1W	RM-2P-1S1D
1R-1W-imm	RM-2P-1S1D
1R-1W-CRo	RM-2P-1S1D
1R-1W-CRo	RM-2P-1S1D
2R	non-SV
2R-CRo	RM-1P-2S1D
2R-1W	RM-1P-2S1D
2R-1W-CRo	RM-1P-2S1D
2R-1W-CRo	RM-1P-2S1D
2R-1W-CRi	RM-1P-3S1D
3R-1W-CRo	RM-1P-3S1D

F.3 keys

unit	in	outcnt	CR in	CR out	imm	imms	name
LDST	1	1	0	0	1	DS/SI	LDST-1R-1W-imm
LDST	1	2	0	0	1	DS/SI	LDST-1R-2W-imm
LDST	2	0	0	0			LDST-2R
LDST	2	0	0	0	1	DS/SI	LDST-2R-imm
LDST	2	1	0	0			LDST-2R-1W
LDST	2	1	0	0	1	DS/SI/SVD	LDST-2R-1W-imm
LDST	2	2	0	0			LDST-2R-2W
LDST	2	2	0	0	1	SVD	LDST-2R-2W-imm
LDST	3	0	0	0			LDST-3R
LDST	3	0	0	1			LDST-3R-CRo
LDST	3	1	0	0			LDST-3R-1W
OTHER	0	0	0	0			
OTHER	0	0	0	0	1	LI	imm
OTHER	0	0	0	1			CRo
OTHER	0	0	1	1			CRio
OTHER	0	0	1	1	1		
OTHER	0	1	0	0			1W
OTHER	0	1	0	0	1	UI	1W-imm
OTHER	0	1	0	1			1W-CRo
OTHER	0	1	1	0			1W-CRi
OTHER	0	1	1	0	1	BD	1W-CRi
OTHER	1	0	0	0			1R
OTHER	1	0	0	0	1	SI	1R-imm
OTHER	1	0	0	1			1R-CRo
OTHER	1	0	0	1	1	SI/UI	1R-CRo
OTHER	1	0	1	1			1R-CRio
OTHER	1	1	0	0			1R-1W
OTHER	1	1	0	0	1	SI/UI	1R-1W-imm
OTHER	1	1	0	1			1R-1W-CRo
OTHER	1	1	0	1	1	M1/SH/SH32/SI/UI/XBI	1R-1W-CRo
OTHER	2	0	0	0			2R
OTHER	2	0	0	1			2R-CRo
OTHER	2	1	0	0			2R-1W
OTHER	2	1	0	1			2R-1W-CRo
OTHER	2	1	0	1	1	SH/SH32	2R-1W-CRo
OTHER	2	1	1	0			2R-1W-CRi
OTHER	3	1	0	1			3R-1W-CRo

F.3.1 LDST-1R-1W-imm (LDSTRM-2P-1S1D)

CSV	opcode	asm	flags	form
major.csv	32	lwz	~SVP64BREV	D-Form
major.csv	34	lbz	~SVP64BREV	D-Form
major.csv	40	lhz	~SVP64BREV	D-Form
major.csv	42	lha	~SVP64BREV	D-Form
major.csv	48	lfs	~SVP64BREV	D-Form
major.csv	50	lfd	~SVP64BREV	D-Form
minor_58.csv	0	ld		DS-Form

CSV	opcode	asm	flags	form
minor_58.csv	2	lwa		DS-Form

F.3.2 LDST-1R-2W-imm (LDSTRM-2P-1S2D)

CSV	opcode	asm	flags	form
major.csv	33	lwzu	~SVP64BREV	D-Form
major.csv	35	lbzu	~SVP64BREV	D-Form
major.csv	41	lhzu	~SVP64BREV	D-Form
major.csv	43	lhau	~SVP64BREV	D-Form
major.csv	49	lfsu	~SVP64BREV	D-Form
major.csv	51	lfdu	~SVP64BREV	D-Form
minor_58.csv	1	ldu		DS-Form

F.3.3 LDST-2R (-)

CSV	opcode	asm	flags	form
minor_31.csv	0b1111110110	dcbz		X-Form

F.3.4 LDST-2R-imm (LDSTRM-2P-2S)

CSV	opcode	asm	flags	form
major.csv	36	stw		D-Form
major.csv	38	stb		D-Form
major.csv	44	sth		D-Form
major.csv	52	stfs		D-Form
major.csv	54	stfd		D-Form
minor_62.csv	0	std		DS-Form

F.3.5 LDST-2R-1W (LDSTRM-2P-2S1D)

CSV	opcode	asm	flags	form
minor_31.csv	0b0000010100	lwarx		X-Form
minor_31.csv	0b0000010101	ldx		X-Form
minor_31.csv	0b0000010111	lwzx		X-Form
minor_31.csv	0b0000110100	lbarx		X-Form
minor_31.csv	0b0001010100	ldarx		X-Form

CSV	opcode	asm	flags	form
minor_31.csv	0b0001010111	lbzx		X-Form
minor_31.csv	0b0001110100	lharx		X-Form
minor_31.csv	0b0100010111	lhzx		X-Form
minor_31.csv	0b0101010101	lwax		X-Form
minor_31.csv	0b0101010111	lhax		X-Form
minor_31.csv	0b1000010100	ldbrx		X-Form
minor_31.csv	0b1000010110	lwbrx		X-Form
minor_31.csv	0b1000010111	lfsx		X-Form
minor_31.csv	0b1001010111	lfdx		X-Form
minor_31.csv	0b1100010101	lwzcix		X-Form
minor_31.csv	0b1100010110	lhbrx		X-Form
minor_31.csv	0b1100110101	lhzcix		X-Form
minor_31.csv	0b1101010101	lbzcix		X-Form
minor_31.csv	0b1101010111	lfiwax		X-Form
minor_31.csv	0b1101110101	ldcix		X-Form
minor_31.csv	0b1101110111	lfiwzx		X-Form

F.3.6 LDST-2R-1W-imm (LDSTRM-2P-2S1D)

CSV	opcode	asm	flags	form
major.csv	32	lwz	SVP64BREV	SVD-Form
major.csv	34	lbz	SVP64BREV	SVD-Form
major.csv	37	stwu		D-Form
major.csv	39	stbu		D-Form
major.csv	40	lhz	SVP64BREV	SVD-Form
major.csv	42	lha	SVP64BREV	SVD-Form
major.csv	45	sthu		D-Form
major.csv	48	lfs	SVP64BREV	SVD-Form
major.csv	50	lfd	SVP64BREV	SVD-Form
major.csv	53	stfsu		D-Form
major.csv	55	stfdu		D-Form
minor_62.csv	1	stdu		DS-Form

F.3.7 LDST-2R-2W (LDSTRM-2P-2S1D)

CSV	opcode	asm	flags	form
minor_31.csv	0b0000110101	ldux		X-Form
minor_31.csv	0b0000110111	lwzux		X-Form
minor_31.csv	0b0001110111	lbzux		X-Form
minor_31.csv	0b0100110111	lhzux		X-Form
minor_31.csv	0b0101110101	lwaux		X-Form
minor_31.csv	0b0101110111	lhaux		X-Form
minor_31.csv	0b1000110111	lfsux		X-Form
minor_31.csv	0b1001110111	lfdux		X-Form

F.3.8 LDST-2R-2W-imm (-)

CSV	opcode	asm	flags	form
major.csv	33	lwzu	SVP64BREV	SVD-Form
major.csv	35	lbzu	SVP64BREV	SVD-Form
major.csv	41	lhzu	SVP64BREV	SVD-Form
major.csv	43	lhau	SVP64BREV	SVD-Form
major.csv	49	lfsu	SVP64BREV	SVD-Form
major.csv	51	lfdu	SVP64BREV	SVD-Form

F.3.9 LDST-3R (LDSTRM-2P-3S)

CSV	opcode	asm	flags	form
minor_31.csv	0b0010010101	stdx		X-Form
minor_31.csv	0b0010010111	stwx		X-Form
minor_31.csv	0b0011010111	stbx		X-Form
minor_31.csv	0b0110010111	sthx		X-Form
minor_31.csv	0b1010010100	stdbrx		X-Form
minor_31.csv	0b1010010110	stwbrx		X-Form
minor_31.csv	0b1010010111	stfsx		X-Form
minor_31.csv	0b1011010111	stfdx		X-Form
minor_31.csv	0b1110010101	stwcix		X-Form
minor_31.csv	0b1110010110	sthbrx		X-Form
minor_31.csv	0b1110110101	sthcix		X-Form
minor_31.csv	0b1111010101	stbcix		X-Form
minor_31.csv	0b1111010111	stfiwx		X-Form
minor_31.csv	0b1111110101	stdcix		X-Form

F.3.10 LDST-3R-CRo (LDSTRM-2P-3S)

CSV	opcode	asm	flags	form
minor_31.csv	0b0010010110	stwcx		X-Form
minor_31.csv	0b0011010110	stdcx		X-Form
minor_31.csv	0b1010110110	stbcx		X-Form
minor_31.csv	0b1011010110	sthcx		X-Form

F.3.11 LDST-3R-1W (LDSTRM-2P-2S1D)

CSV	opcode	asm	flags	form
minor_31.csv	0b0010110101	stdux		X-Form

CSV	opcode	asm	flags	form
minor_31.csv	0b0010110111	stwux		X-Form
minor_31.csv	0b0011110111	stbux		X-Form
minor_31.csv	0b0110110111	sthux		X-Form
minor_31.csv	0b1010110111	stfsux		X-Form
minor_31.csv	0b1011110111	stfdux		X-Form

F.3.12 (non-SV)

CSV	opcode	asm	flags	form
extra.csv	000000----------0100000000-	attn		NONE-Form
extra.csv	01100000000000000000000000000000	nop		D-Form
major.csv	17	sc		SC-Form
minor_19.csv	0b0010010110	isync		XL-Form
minor_19_00000.csv	0b00010	add pcis not implemented yet		DX-Form
minor_22.csv	--011001	svshape		SVM-Form
minor_22.csv	--101001	svindex		SVI-Form
minor_22.csv	--111001	svremap		SVRM-Form
minor_31.csv	0b0000010110	icbt		X-Form
minor_31.csv	0b0000011110	wait		X-Form
minor_31.csv	0b0000110110	dcbst		X-Form
minor_31.csv	0b0001010110	dcbf		X-Form
minor_31.csv	0b0011110110	dcbtst		X-Form
minor_31.csv	0b0100010110	dcbt		X-Form
minor_31.csv	0b0111110010	slbia		X-Form
minor_31.csv	0b1000110110	tlbsync		X-Form
minor_31.csv	0b1001010110	sync		X-Form
minor_31.csv	0b1101010110	eieio		X-Form
minor_31.csv	0b1111010110	icbi		X-Form

F.3.13 imm (non-SV)

CSV	opcode	asm	flags	form
major.csv	18	b		I-Form

F.3.14 CRo (-)

CSV	opcode	asm	flags	form
minor_63.csv	0000100110	1/6=mtfsb1		X-Form
minor_63.csv	0001000000	2/0=mcrfs		X-Form
minor_63.csv	0001000110	2/6=mtfsb0		X-Form

CSV	opcode	asm	flags	form
minor_63.csv	0010000110	4/6=mtfsfi		X-Form

F.3.15 CRio (RM-2P-1S1D)

CSV	opcode	asm	flags	form
minor_19.csv	0b0000000000	mcrf		XL-Form

F.3.16 CR=2R1W (RM-1P-2S1D)

CSV	opcode	asm	flags	form
minor_19.csv	0b0000100001	crnor		XL-Form
minor_19.csv	0b0010000001	crandc		XL-Form
minor_19.csv	0b0011000001	crxor		XL-Form
minor_19.csv	0b0011100001	crnand		XL-Form
minor_19.csv	0b0100000001	crand		XL-Form
minor_19.csv	0b0100100001	creqv		XL-Form
minor_19.csv	0b0110100001	crorc		XL-Form
minor_19.csv	0b0111000001	cror		XL-Form

F.3.17 1W (non-SV)

CSV	opcode	asm	flags	form
extra.csv	000001----------0000000011-	sim_cfg		NONE-Form
minor_31.csv	0b0001010011	mfmsr		X-Form

F.3.18 1W-imm (RM-1P-1D)

CSV	opcode	asm	flags	form
minor_22.csv	----00011-	fmvis		DX-Form

F.3.19 1W-CRo (RM-1P-1D)

CSV	opcode	asm	flags	form
minor_22.csv	——10011-	svstep		SVL-Form

F.3.20 1W-CRi (RM-2P-1S1D)

CSV	opcode	asm	flags	form
minor_19.csv	0b0000010000	bclr		XL-Form
minor_31.csv	0b0000010011	mfcr/mfocrf		XFX-Form
minor_31.csv	0b0010000000	setb		VX-Form

F.3.21 1W-CRi (RM-2P-1S1D)

CSV	opcode	asm	flags	form
major.csv	16	bc		B-Form

F.3.22 1R (non-SV)

CSV	opcode	asm	flags	form
minor_31.csv	0b0010010010	mtmsr		X-Form
minor_31.csv	0b0010110010	mtmsrd		X-Form
minor_31.csv	0b0100010010	tlbiel		X-Form
minor_31.csv	0b0100110010	tlbie		X-Form

F.3.23 1R-imm (RM-1P-1S)

CSV	opcode	asm	flags	form
major.csv	2	tdi		D-Form
major.csv	3	twi		D-Form

F.3.24 1R-CRo (RM-2P-1S1D)

CSV	opcode	asm	flags	form
minor_63.csv	0010100000	5/0=ftsqrt		X-Form

CSV	opcode	asm	flags	form
minor_63.csv	1011000111	22/7=mtfsf		X-Form

F.3.25 1R-CRo (RM-2P-1S1D)

CSV	opcode	asm	flags	form
major.csv	10	cmpli		D-Form
major.csv	11	cmpi		D-Form

F.3.26 1R-CRio (RM-2P-2S1D)

CSV	opcode	asm	flags	form
minor_31.csv	0b0010010000	mtcrf/mtocrf		XFX-Form

F.3.27 1R-1W (RM-2P-1S1D)

CSV	opcode	asm	flags	form
minor_31.csv	0b0001101000	neg		XO-Form
minor_31.csv	0b0001111010	popcntb		X-Form
minor_31.csv	0b0010011010	prtyw		X-Form
minor_31.csv	0b0010111010	prtyd		X-Form
minor_31.csv	0b0100011010	cdtbcd		X-Form
minor_31.csv	0b0100111010	cbcdtd		X-Form
minor_31.csv	0b0101010011	mfspr		XFX-Form
minor_31.csv	0b0101111010	popcntw		X-Form
minor_31.csv	0b0111010011	mtspr		XFX-Form
minor_31.csv	0b0111111010	popcntd		X-Form
minor_31.csv	0b1001101000	nego		XO-Form

F.3.28 1R-1W-imm (RM-2P-1S1D)

CSV	opcode	asm	flags	form
major.csv	12	addic		D-Form
major.csv	14	addi		D-Form
major.csv	15	addis		D-Form
major.csv	24	ori		D-Form
major.csv	25	oris		D-Form

CSV	opcode	asm	flags	form
major.csv	26	xori		D-Form
major.csv	27	xoris		D-Form
major.csv	8	subfic		D-Form
minor_22.csv	——01011-	fishmv		DX-Form

F.3.29 1R-1W-CRo (RM-2P-1S1D)

CSV	opcode	asm	flags	form
minor_31.csv	0b0000011010	cntlzw		X-Form
minor_31.csv	0b0000111010	cntlzd		X-Form
minor_31.csv	0b0011001000	subfze		XO-Form
minor_31.csv	0b0011001010	addze		XO-Form
minor_31.csv	0b1000011010	cnttzw		X-Form
minor_31.csv	0b1000111010	cnttzd		X-Form
minor_31.csv	0b1011001000	subfzeo		XO-Form
minor_31.csv	0b1011001010	addzeo		XO-Form
minor_31.csv	0b1110011010	extsh		X-Form
minor_31.csv	0b1110111010	extsb		X-Form
minor_31.csv	0b1111011010	extsw		X-Form
minor_59.csv	——10110	fsqrts		A-Form
minor_59.csv	——11000	fres		A-Form
minor_59.csv	——11010	frsqrtes		A-Form
minor_59.csv	1000001110	fsins		X-Form
minor_59.csv	1000101110	fcoss		X-Form
minor_59.csv	1101001110	fcfids		X-Form
minor_59.csv	1111001110	fcfidus		X-Form
minor_63.csv	——10110	fsqrt		A-Form
minor_63.csv	——11000	fre		A-Form
minor_63.csv	——11010	frsqrte		A-Form
minor_63.csv	0000001100	0/12=frsp		X-Form
minor_63.csv	0000001110	0/14=fctiw		X-Form
minor_63.csv	0000001111	0/15=fctiwz		X-Form
minor_63.csv	0000101000	1/8=fneg		X-Form
minor_63.csv	0001001000	2/8=fmr		X-Form
minor_63.csv	0010001000	4/8=fnabs		X-Form
minor_63.csv	0010001110	4/14=fctiwu		X-Form
minor_63.csv	0010001111	4/15=fctiwuz		X-Form
minor_63.csv	0100001000	8/8=fabs		X-Form
minor_63.csv	0110001000	12/8=frin		X-Form
minor_63.csv	0110101000	13/8=friz		X-Form
minor_63.csv	0111001000	14/8=frip		X-Form
minor_63.csv	0111101000	15/8=frim		X-Form
minor_63.csv	1001000111	18/7=mffs		X-Form
minor_63.csv	1100101110	25/14=fctid		X-Form
minor_63.csv	1100101111	25/15=fctidz		X-Form
minor_63.csv	1101001110	26/14=fcfid		X-Form
minor_63.csv	1110101110	29/14=fctidu		X-Form
minor_63.csv	1110101111	29/15=fctiduz		X-Form

CSV	opcode	asm	flags	form
minor_63.csv	1111001110	30/14=fcfidu		X-Form

F.3.30 1R-1W-CRo (RM-2P-1S1D)

CSV	opcode	asm	flags	form
major.csv	13	addic.		D-Form
major.csv	21	rlwinm		M-Form
major.csv	28	andi.		B-Form
major.csv	29	andis.		B-Form
major.csv	7	mulli		D-Form
minor_30.csv	0b0000	rldicl		MDS-Form
minor_30.csv	0b0001	rldicl		MDS-Form
minor_30.csv	0b0010	rldicr		MD-Form
minor_30.csv	0b0011	rldicr		MD-Form
minor_30.csv	0b0100	rldic		MD-Form
minor_30.csv	0b0101	rldic		MD-Form
minor_31.csv	0b0011101000	subfme		XO-Form
minor_31.csv	0b0011101010	addme		XO-Form
minor_31.csv	0b1011101000	subfmeo		XO-Form
minor_31.csv	0b1011101010	addmeo		XO-Form
minor_31.csv	0b1100111000	srawi		X-Form
minor_31.csv	0b1100111010	sradi		XS-Form
minor_31.csv	0b1100111011	sradi		XS-Form
minor_31.csv	0b1101111010	extswsli		XS-Form
minor_31.csv	0b1101111011	extswsli		XS-Form
minor_5.csv	-011010110-	grevi		XB-Form
minor_5.csv	0011110110-	grevwi		X-Form

F.3.31 2R (non-SV)

CSV	opcode	asm	flags	form
minor_31.csv	0b0000000100	tw		X-Form
minor_31.csv	0b0001000100	td		X-Form

F.3.32 2R-CRo (RM-1P-2S1D)

CSV	opcode	asm	flags	form
minor_31.csv	0b0000000000	cmp		X-Form
minor_31.csv	0b0000100000	cmpl		X-Form
minor_31.csv	0b0011000000	cmprb		X-Form

CSV	opcode	asm	flags	form
minor_31.csv	0b0011100000	cmpeqb		X-Form
minor_63.csv	0000000000	0/0=fcmpu		X-Form
minor_63.csv	0000100000	1/0=fcmpo		X-Form
minor_63.csv	0010000000	4/0=ftdiv		X-Form

F.3.33 2R-1W (RM-1P-2S1D)

CSV	opcode	asm	flags	form
minor_22.csv	——10001	bmask		BM2-Form
minor_31.csv	0b0011111100	bpermd		X-Form
minor_31.csv	0b0100001001	modud		X-Form
minor_31.csv	0b0100001011	moduw		X-Form
minor_31.csv	0b0111111100	cmpb		X-Form
minor_31.csv	0b1100001001	modsd		X-Form
minor_31.csv	0b1100001011	modsw		X-Form
minor_63.csv	1101000110	26/6=fmrgow		X-Form
minor_63.csv	1111000110	30/6=fmrgew		X-Form

F.3.34 2R-1W-CRo (RM-1P-2S1D)

CSV	opcode	asm	flags	form
major.csv	23	rlwnm		M-Form
minor_22.csv	0001001110-	minu		X-Form
minor_22.csv	0011001110-	maxu		X-Form
minor_22.csv	0101001110-	mins		X-Form
minor_22.csv	0110001110-	cprop		X-Form
minor_22.csv	0111001110-	maxs		X-Form
minor_22.csv	1001110110-	absds		X-Form
minor_22.csv	1011110110-	absdu		X-Form
minor_22.csv	1101001110-	avgadd		X-Form
minor_30.csv	0b1000	rldcl		MD-Form
minor_30.csv	0b1001	rldcr		MD-Form
minor_31.csv	0b0000001000	subfc		XO-Form
minor_31.csv	0b0000001001	mulhdu		XO-Form
minor_31.csv	0b0000001010	addc		XO-Form
minor_31.csv	0b0000001011	mulhwu		XO-Form
minor_31.csv	0b0000011000	slw		X-Form
minor_31.csv	0b0000011011	sld		X-Form
minor_31.csv	0b0000011100	and		X-Form
minor_31.csv	0b0000101000	subf		XO-Form
minor_31.csv	0b0000111100	andc		X-Form
minor_31.csv	0b0001001001	mulhd		XO-Form
minor_31.csv	0b0001001010	addg6s		XO-Form
minor_31.csv	0b0001001011	mulhw		XO-Form

CSV	opcode	asm	flags	form
minor_31.csv	0b0001111100	nor		X-Form
minor_31.csv	0b0010001000	subfe		XO-Form
minor_31.csv	0b0010001010	adde		XO-Form
minor_31.csv	0b0011101001	mulld		XO-Form
minor_31.csv	0b0011101011	mullw		XO-Form
minor_31.csv	0b0100001010	add		XO-Form
minor_31.csv	0b0100011100	eqv		X-Form
minor_31.csv	0b0100111100	xor		X-Form
minor_31.csv	0b0110001001	divdeu		XO-Form
minor_31.csv	0b0110001011	divweu		XO-Form
minor_31.csv	0b0110011100	orc		X-Form
minor_31.csv	0b0110101001	divde		XO-Form
minor_31.csv	0b0110101011	divwe		XO-Form
minor_31.csv	0b0110111100	or		X-Form
minor_31.csv	0b0111001001	divdu		XO-Form
minor_31.csv	0b0111001011	divwu		XO-Form
minor_31.csv	0b0111011100	nand		X-Form
minor_31.csv	0b0111101001	divd		XO-Form
minor_31.csv	0b0111101011	divw		XO-Form
minor_31.csv	0b1000001000	subfco		XO-Form
minor_31.csv	0b1000001001	mulhdu		XO-Form
minor_31.csv	0b1000001010	addco		XO-Form
minor_31.csv	0b1000001011	mulhwu		XO-Form
minor_31.csv	0b1000011000	srw		X-Form
minor_31.csv	0b1000011011	srd		X-Form
minor_31.csv	0b1000101000	subfo		XO-Form
minor_31.csv	0b1001001001	mulhd		XO-Form
minor_31.csv	0b1001001011	mulhw		XO-Form
minor_31.csv	0b1010001000	subfeo		XO-Form
minor_31.csv	0b1010001010	addeo		XO-Form
minor_31.csv	0b1011101001	mulldo		XO-Form
minor_31.csv	0b1011101011	mullwo		XO-Form
minor_31.csv	0b1100001010	addo		XO-Form
minor_31.csv	0b1100011000	sraw		X-Form
minor_31.csv	0b1100011010	srad		X-Form
minor_31.csv	0b1110001001	divdeuo		XO-Form
minor_31.csv	0b1110001011	divweuo		XO-Form
minor_31.csv	0b1110101001	divdeo		XO-Form
minor_31.csv	0b1110101011	divweo		XO-Form
minor_31.csv	0b1111001001	divduo		XO-Form
minor_31.csv	0b1111001011	divwuo		XO-Form
minor_31.csv	0b1111101001	divdo		XO-Form
minor_31.csv	0b1111101011	divwo		XO-Form
minor_5.csv	0010010110-	grev		X-Form
minor_5.csv	0010110110-	grevw		X-Form
minor_59.csv	----01101	ffadds		A-Form
minor_59.csv	----10010	fdivs		A-Form
minor_59.csv	----10100	fsubs		A-Form
minor_59.csv	----10101	fadds		A-Form
minor_59.csv	----11001	fmuls		A-Form
minor_63.csv	----10010	fdiv		A-Form
minor_63.csv	----10100	fsub		A-Form

CSV	opcode	asm	flags	form
minor_63.csv	——10101	fadd		A-Form
minor_63.csv	——11001	fmul		A-Form
minor_63.csv	0000001000	0/8=fcpsgn		X-Form

F.3.35 2R-1W-CRo (RM-1P-2S1D)

CSV	opcode	asm	flags	form
major.csv	20	rlwimi		M-Form
minor_30.csv	0b0110	rldimi		MD-Form
minor_30.csv	0b0111	rldimi		MD-Form

F.3.36 2R-1W-CRi (RM-1P-3S1D)

CSV	opcode	asm	flags	form
minor_31.csv	0b0000001111	isel		A-Form
minor_31.csv	0b0000101111	isel		A-Form
minor_31.csv	0b0001001111	isel		A-Form
minor_31.csv	0b0001101111	isel		A-Form
minor_31.csv	0b0010001111	isel		A-Form
minor_31.csv	0b0010101111	isel		A-Form
minor_31.csv	0b0011001111	isel		A-Form
minor_31.csv	0b0011101111	isel		A-Form
minor_31.csv	0b0100001111	isel		A-Form
minor_31.csv	0b0100101111	isel		A-Form
minor_31.csv	0b0101001111	isel		A-Form
minor_31.csv	0b0101101111	isel		A-Form
minor_31.csv	0b0110001111	isel		A-Form
minor_31.csv	0b0110101111	isel		A-Form
minor_31.csv	0b0111001111	isel		A-Form
minor_31.csv	0b0111101111	isel		A-Form
minor_31.csv	0b1000001111	isel		A-Form
minor_31.csv	0b1000101111	isel		A-Form
minor_31.csv	0b1001001111	isel		A-Form
minor_31.csv	0b1001101111	isel		A-Form
minor_31.csv	0b1010001111	isel		A-Form
minor_31.csv	0b1010101111	isel		A-Form
minor_31.csv	0b1011001111	isel		A-Form
minor_31.csv	0b1011101111	isel		A-Form
minor_31.csv	0b1100001111	isel		A-Form
minor_31.csv	0b1100101111	isel		A-Form
minor_31.csv	0b1101001111	isel		A-Form
minor_31.csv	0b1101101111	isel		A-Form
minor_31.csv	0b1110001111	isel		A-Form
minor_31.csv	0b1110101111	isel		A-Form

CSV	opcode	asm	flags	form
minor_31.csv	0b1111001111	isel		A-Form
minor_31.csv	0b1111101111	isel		A-Form

F.3.37 3R-1W-CRo (RM-1P-3S1D)

CSV	opcode	asm	flags	form
minor_22.csv	0111110110-	absdacs		X-Form
minor_22.csv	1111110110-	absdacu		X-Form
minor_5.csv	———00-	ternlogi		TLI-Form
minor_59.csv	——00100	ffmsubs		A-Form
minor_59.csv	——00101	ffmadds		A-Form
minor_59.csv	——00110	ffnmsubs		A-Form
minor_59.csv	——00111	ffnmadds		A-Form
minor_59.csv	——01111	fdmadds		A-Form
minor_59.csv	——11100	fmsubs		A-Form
minor_59.csv	——11101	fmadds		A-Form
minor_59.csv	——11110	fnmsubs		A-Form
minor_59.csv	——11111	fnmadds		A-Form
minor_63.csv	——10111	fsel		A-Form
minor_63.csv	——11100	fmsub		A-Form
minor_63.csv	——11101	fmadd		A-Form
minor_63.csv	——11110	fnmsub		A-Form
minor_63.csv	——11111	fnmadd		A-Form

F.4 svp64 remaps

- **LDST-1R-1W-imm**: LDSTRM-2P-1S1D
- **LDST-1R-2W-imm**: LDSTRM-2P-1S2D
- **LDST-2R**: -
- **LDST-2R-imm**: LDSTRM-2P-2S
- **LDST-2R-1W**: LDSTRM-2P-2S1D
- **LDST-2R-1W-imm**: LDSTRM-2P-2S1D
- **LDST-2R-2W**: LDSTRM-2P-2S1D
- **LDST-2R-2W-imm**: -
- **LDST-3R**: LDSTRM-2P-3S
- **LDST-3R-CRo**: LDSTRM-2P-3S
- **LDST-3R-1W**: LDSTRM-2P-2S1D
- **CRo**: -
- **CRio**: RM-2P-1S1D
- **CR=2R1W**: RM-1P-2S1D
- **1W-imm**: RM-1P-1D
- **1W-CRo**: RM-1P-1D
- **1W-CRi**: RM-2P-1S1D
- **1W-CRi**: RM-2P-1S1D
- **1R-imm**: RM-1P-1S

- **1R-CRo**: RM-2P-1S1D
- **1R-CRo**: RM-2P-1S1D
- **1R-CRio**: RM-2P-2S1D
- **1R-1W**: RM-2P-1S1D
- **1R-1W-imm**: RM-2P-1S1D
- **1R-1W-CRo**: RM-2P-1S1D
- **1R-1W-CRo**: RM-2P-1S1D
- **2R-CRo**: RM-1P-2S1D
- **2R-1W**: RM-1P-2S1D
- **2R-1W-CRo**: RM-1P-2S1D
- **2R-1W-CRo**: RM-1P-2S1D
- **2R-1W-CRi**: RM-1P-3S1D
- **3R-1W-CRo**: RM-1P-3S1D

F.4.1 LDSTRM-2P-1S1D

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
lwz	LDST	~SVP64BREV	2P	EXTRA3	d:RT	s:RA	0	0	RA_OR_ZERO	0	0	RT	0	0	0
lbz	LDST	~SVP64BREV	2P	EXTRA3	d:RT	s:RA	0	0	RA_OR_ZERO	0	0	RT	0	0	0
lhz	LDST	~SVP64BREV	2P	EXTRA3	d:RT	s:RA	0	0	RA_OR_ZERO	0	0	RT	0	0	0
lha	LDST	~SVP64BREV	2P	EXTRA3	d:RT	s:RA	0	0	RA_OR_ZERO	0	0	RT	0	0	0
lfs	LDST	~SVP64BREV	2P	EXTRA3	d:FRT	s:RA	0	0	RA_OR_ZERO	0	0	FRT	0	0	0
lfd	LDST	~SVP64BREV	2P	EXTRA3	d:FRT	s:RA	0	0	RA_OR_ZERO	0	0	FRT	0	0	0
ld	LDST		2P	EXTRA3	d:RT	s:RA	0	0	RA_OR_ZERO	0	0	RT	0	0	0
lwa	LDST		2P	EXTRA3	d:RT	s:RA	0	0	RA_OR_ZERO	0	0	RT	0	0	0

F.4.2 LDSTRM-2P-1S2D

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
lwzu	LDST	~SVP64BREV	2P	EXTRA2	d:RT	d:RA	s:RA	0	RA_OR_ZERO	0	0	RT	0	0	RA
lbzu	LDST	~SVP64BREV	2P	EXTRA2	d:RT	d:RA	s:RA	0	RA_OR_ZERO	0	0	RT	0	0	RA
lhzu	LDST	~SVP64BREV	2P	EXTRA2	d:RT	d:RA	s:RA	0	RA_OR_ZERO	0	0	RT	0	0	RA
lhau	LDST	~SVP64BREV	2P	EXTRA2	d:RT	d:RA	s:RA	0	RA_OR_ZERO	0	0	RT	0	0	RA
lfsu	LDST	~SVP64BREV	2P	EXTRA2	d:FRT	d:RA	s:RA	0	RA	0	0	FRT	0	0	RA
lfdu	LDST	~SVP64BREV	2P	EXTRA2	d:FRT	d:RA	s:RA	0	RA	0	0	FRT	0	0	RA
ldu	LDST		2P	EXTRA2	d:RT	d:RA	s:RA	0	RA_OR_ZERO	0	0	RT	0	0	RA

F.4.3 LDSTRM-2P-2S

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
stw	LDST		2P	EXTRA3	s:RS	s:RA	0	0	RA_OR_ZERO	0	RS	0	0	0	0
stb	LDST		2P	EXTRA3	s:RS	s:RA	0	0	RA_OR_ZERO	0	RS	0	0	0	0
sth	LDST		2P	EXTRA3	s:RS	s:RA	0	0	RA_OR_ZERO	0	RS	0	0	0	0
stfs	LDST		2P	EXTRA3	s:FRS	s:RA	0	0	RA_OR_ZERO	0	FRS	0	0	0	0

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
stfd	LDST		2P	EXTRA3	s:FRS	s:RA	0	0	RA_OR_ZERO	0	FRS	0	0	0	0
std	LDST		2P	EXTRA3	s:RS	s:RA	0	0	RA_OR_ZERO	0	RS	0	0	0	0

F.4.4 LDSTRM-2P-2S1D

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
lwarx	LDST		2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	0
ldx	LDST		2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	0
lwzx	LDST		2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	0
lbarx	LDST		2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	0
ldarx	LDST		2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	0
lbzx	LDST		2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	0
lharx	LDST		2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	0
lhzx	LDST		2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	0
lwax	LDST		2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	0
lhax	LDST		2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	0
ldbrx	LDST		2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	0
lwbrx	LDST		2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	0
lfsx	LDST		2P	EXTRA2	d:FRT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	FRT	0	0	0
lfdx	LDST		2P	EXTRA2	d:FRT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	FRT	0	0	0
lwzcix	LDST		2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	0
lhbrx	LDST		2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	0
lhzcix	LDST		2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	0
lbzcix	LDST		2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	0
lfiwax	LDST		2P	EXTRA2	d:FRT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	FRT	0	0	0
ldcix	LDST		2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	0
lfiwzx	LDST		2P	EXTRA2	d:FRT	s:RA	s:RB	0	RA_OR_ZERO	RB	0	FRT	0	0	0
lwz	LDST	SVP64BREV	2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	0	RC	RT	0	0	0
lbz	LDST	SVP64BREV	2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	0	RC	RT	0	0	0
stwu	LDST		2P	EXTRA2	d:RA	s:RS	s:RA	0	RA_OR_ZERO	0	RS	0	0	0	RA
stbu	LDST		2P	EXTRA2	d:RA	s:RS	s:RA	0	RA_OR_ZERO	0	RS	0	0	0	RA
lhz	LDST	SVP64BREV	2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	0	RC	RT	0	0	0
lha	LDST	SVP64BREV	2P	EXTRA2	d:RT	s:RA	s:RB	0	RA_OR_ZERO	0	RC	RT	0	0	0
sthu	LDST		2P	EXTRA2	d:RA	s:RS	s:RA	0	RA_OR_ZERO	0	RS	0	0	0	RA
lfs	LDST	SVP64BREV	2P	EXTRA2	d:FRT	s:RA	s:RB	0	RA_OR_ZERO	0	RC	FRT	0	0	0
lfd	LDST	SVP64BREV	2P	EXTRA2	d:FRT	s:RA	s:RB	0	RA_OR_ZERO	0	RC	FRT	0	0	0
stfsu	LDST		2P	EXTRA2	d:RA	s:FRS	s:RA	0	RA	0	FRS	0	0	0	RA
stfdu	LDST		2P	EXTRA2	d:RA	s:FRS	s:RA	0	RA	0	FRS	0	0	0	RA
stdu	LDST		2P	EXTRA2	d:RA	s:RS	s:RA	0	RA_OR_ZERO	0	RS	0	0	0	RA
ldux	LDST		2P	EXTRA2	d:RT	d:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	RA
lwzux	LDST		2P	EXTRA2	d:RT	d:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	RA
lbzux	LDST		2P	EXTRA2	d:RT	d:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	RA
lhzux	LDST		2P	EXTRA2	d:RT	d:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	RA
lwaux	LDST		2P	EXTRA2	d:RT	d:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	RA
lhaux	LDST		2P	EXTRA2	d:RT	d:RA	s:RB	0	RA_OR_ZERO	RB	0	RT	0	0	RA
lfsux	LDST		2P	EXTRA2	d:FRT	d:RA	s:RB	0	RA	RB	0	FRT	0	0	RA
lfdux	LDST		2P	EXTRA2	d:FRT	d:RA	s:RB	0	RA	RB	0	FRT	0	0	RA
stdux	LDST		2P	EXTRA2	d:RA	s:RSs:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	0	RA

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
stwux	LDST		2P	EXTRA2	d:RA	s:RSs:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	0	RA
stbux	LDST		2P	EXTRA2	d:RA	s:RSs:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	0	RA
sthux	LDST		2P	EXTRA2	d:RA	s:RSs:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	0	RA
stfsux	LDST		2P	EXTRA2	d:RA	s:FRSs:RA	s:RB	0	RA	RB	FRS	0	0	0	RA
stfdux	LDST		2P	EXTRA2	d:RA	s:FRSs:RA	s:RB	0	RA	RB	FRS	0	0	0	RA

F.4.5 LDSTRM-2P-3S

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
stdx	LDST		2P	EXTRA2	s:RS	s:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	0	0
stwx	LDST		2P	EXTRA2	s:RS	s:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	0	0
stbx	LDST		2P	EXTRA2	s:RS	s:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	0	0
sthx	LDST		2P	EXTRA2	s:RS	s:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	0	0
stdbrx	LDST		2P	EXTRA2	s:RS	s:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	0	0
stwbrx	LDST		2P	EXTRA2	s:RS	s:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	0	0
stfsx	LDST		2P	EXTRA2	s:FRS	s:RA	s:RB	0	RA	RB	FRS	0	0	0	0
stfdx	LDST		2P	EXTRA2	s:FRS	s:RA	s:RB	0	RA_OR_ZERO	RB	FRS	0	0	0	0
stwcix	LDST		2P	EXTRA2	s:RS	s:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	0	0
sthbrx	LDST		2P	EXTRA2	s:RS	s:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	0	0
sthcix	LDST		2P	EXTRA2	s:RS	s:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	0	0
stbcix	LDST		2P	EXTRA2	s:RS	s:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	0	0
stfiwx	LDST		2P	EXTRA2	s:FRS	s:RA	s:RB	0	RA_OR_ZERO	RB	FRS	0	0	0	0
stdcix	LDST		2P	EXTRA2	s:RS	s:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	0	0
stwcx	LDST		2P	EXTRA2	s:RSd:CR0	s:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	CR0	0
stdcx	LDST		2P	EXTRA2	s:RSd:CR0	s:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	CR0	0
stbcx	LDST		2P	EXTRA2	s:RSd:CR0	s:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	CR0	0
sthcx	LDST		2P	EXTRA2	s:RSd:CR0	s:RA	s:RB	0	RA_OR_ZERO	RB	RS	0	0	CR0	0

F.4.6 RM-2P-1S1D

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
mcrf	NORMAL		2P	EXTRA3	d:BF	s:BFA	0	0	0	0	0	0	BFA	BF	0
bclr	BRANCH		2P	EXTRA3	d:BI	s:BI	0	0	SPR	SPR	0	SPR	BI	0	0
mfcr/mfocrf	NORMAL		2P	EXTRA3	d:RT	s:CR	0	0	0	0	0	RT	WHOLE_REG	0	0
setb	NORMAL		2P	EXTRA3	d:RT	s:BFA	0	0	0	0	0	RT	BFA	0	0
bc	BRANCH		2P	EXTRA3	d:BI	s:BI	0	0	SPR	0	0	SPR	BI	0	0
5/0=ftsqrt	NORMAL		2P	EXTRA3	d:BF	s:FRB	0	0	0	FRB	0	0	0	BF	0
22/7=mtfsf	NORMAL		2P	EXTRA3	d:CR1	s:FRB	0	0	0	FRB	0	0	0	CR1	0
cmpli	NORMAL		2P	EXTRA3	d:BF	s:RA	0	0	RA	0	0	0	0	BF	0
cmpi	NORMAL		2P	EXTRA3	d:BF	s:RA	0	0	RA	0	0	0	0	BF	0
neg	NORMAL		2P	EXTRA3	d:RT	s:RA	0	0	RA	0	0	RT	0	0	0
popcntb	NORMAL		2P	EXTRA3	d:RA	s:RS	0	0	RS	0	0	RA	0	0	0
prtyw	NORMAL		2P	EXTRA3	d:RA	s:RS	0	0	RS	0	0	RA	0	0	0
prtyd	NORMAL		2P	EXTRA3	d:RA	s:RS	0	0	RS	0	0	RA	0	0	0
cdtbcd	NORMAL		2P	EXTRA3	d:RA	s:RS	0	0	RS	0	0	RA	0	0	0

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
cbcdtd	NORMAL		2P	EXTRA3	d:RA	s:RS	0	0	RS	0	0	RA	0	0	0
mfspr	NORMAL		2P	EXTRA3	d:RS	s:SPR	0	0	SPR	0	0	RT	0	0	0
popcntw	NORMAL		2P	EXTRA3	d:RA	s:RS	0	0	RS	0	0	RA	0	0	0
mtspr	NORMAL		2P	EXTRA3	d:SPR	s:RS	0	0	RS	0	0	SPR	0	0	0
popcntd	NORMAL		2P	EXTRA3	d:RA	s:RS	0	0	RS	0	0	RA	0	0	0
nego	NORMAL		2P	EXTRA3	d:RT	s:RA	0	0	RA	0	0	RT	0	0	0
addic	NORMAL		2P	EXTRA3	d:RT	s:RA	0	0	RA	0	0	RT	0	0	0
addi	NORMAL		2P	EXTRA3	d:RT	s:RA	0	0	RA_OR_ZERO	0	0	RT	0	0	0
addis	NORMAL		2P	EXTRA3	d:RT	s:RA	0	0	RA_OR_ZERO	0	0	RT	0	0	0
ori	NORMAL		2P	EXTRA3	d:RA	s:RS	0	0	RS	0	0	RA	0	0	0
oris	NORMAL		2P	EXTRA3	d:RA	s:RS	0	0	RS	0	0	RA	0	0	0
xori	NORMAL		2P	EXTRA3	d:RA	s:RS	0	0	RS	0	0	RA	0	0	0
xoris	NORMAL		2P	EXTRA3	d:RA	s:RS	0	0	RS	0	0	RA	0	0	0
subfic	NORMAL		2P	EXTRA3	d:RT	s:RA	0	0	RA	0	0	RT	0	0	0
fishmv	NORMAL		2P	EXTRA3	TODO	0	0	0	FRS	0	0	FRS	0	0	0
cntlzw	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	RS	0	0	RA	0	CR0	0
cntlzd	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	RS	0	0	RA	0	CR0	0
subfze	NORMAL		2P	EXTRA3	d:RT;d:CR0	s:RA	0	0	RA	0	0	RT	0	CR0	0
addze	NORMAL		2P	EXTRA3	d:RT;d:CR0	s:RA	0	0	RA	0	0	RT	0	CR0	0
cnttzw	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	RS	0	0	RA	0	CR0	0
cnttzd	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	RS	0	0	RA	0	CR0	0
subfzeo	NORMAL		2P	EXTRA3	d:RT;d:CR0	s:RA	0	0	RA	0	0	RT	0	CR0	0
addzeo	NORMAL		2P	EXTRA3	d:RT;d:CR0	s:RA	0	0	RA	0	0	RT	0	CR0	0
extsh	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	RS	0	0	RA	0	CR0	0
extsb	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	RS	0	0	RA	0	CR0	0
extsw	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	RS	0	0	RA	0	CR0	0
fsqrts	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
fres	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
frsqrtes	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
fsins	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
fcoss	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
fcfids	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
fcfidus	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
fsqrt	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
fre	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
frsqrte	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
0/12=frsp	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
0/14=fctiw	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
0/15=fctiwz	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
1/8=fneg	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
2/8=fmr	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
4/8=fnabs	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
4/14=fctiwu	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
4/15=fctiwuz	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
8/8=fabs	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
12/8=frin	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
13/8=friz	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
14/8=frip	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
15/8=frim	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
18/7=mffs	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
25/14=fctid	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
25/15=fctidz	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
26/14=fcfid	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
29/14=fctidu	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
29/15=fctiduz	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
30/14=fcfidu	NORMAL		2P	EXTRA3	d:FRT;d:CR1	s:FRB	0	0	0	FRB	0	FRT	0	CR1	0
addic.	NORMAL		2P	EXTRA3	d:RT;d:CR0	s:RA	0	0	RA	0	0	RT	0	CR0	0
rlwinm	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	0	0	RS	RA	0	CR0	0
andi.	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	RS	0	0	RA	0	CR0	0
andis.	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	RS	0	0	RA	0	CR0	0
mulli	NORMAL		2P	EXTRA3	d:RT;d:CR0	s:RA	0	0	RA	0	0	RT	0	CR0	0
rldicl	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	0	0	RS	RA	0	CR0	0
rldicl	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	0	0	RS	RA	0	CR0	0
rldicr	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	0	0	RS	RA	0	CR0	0
rldicr	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	0	0	RS	RA	0	CR0	0
rldic	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	0	0	RS	RA	0	CR0	0
rldic	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	0	0	RS	RA	0	CR0	0
subfme	NORMAL		2P	EXTRA3	d:RT;d:CR0	s:RA	0	0	RA	0	0	RT	0	CR0	0
addme	NORMAL		2P	EXTRA3	d:RT;d:CR0	s:RA	0	0	RA	0	0	RT	0	CR0	0
subfmeo	NORMAL		2P	EXTRA3	d:RT;d:CR0	s:RA	0	0	RA	0	0	RT	0	CR0	0
addmeo	NORMAL		2P	EXTRA3	d:RT;d:CR0	s:RA	0	0	RA	0	0	RT	0	CR0	0
srawi	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	0	0	RS	RA	0	CR0	0
sradi	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	0	0	RS	RA	0	CR0	0
sradi	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	0	0	RS	RA	0	CR0	0
extswsli	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	0	0	RS	RA	0	CR0	0
extswsli	NORMAL		2P	EXTRA3	d:RA;d:CR0	s:RS	0	0	0	0	RS	RA	0	CR0	0
grevi	NORMAL		2P	EXTRA3	d:RT;d:CR0	s:RA	0	0	RA	0	0	RT	0	CR0	0
grevwi	NORMAL		2P	EXTRA3	d:RT;d:CR0	s:RA	0	0	RA	0	0	RT	0	CR0	0

F.4.7 RM-1P-2S1D

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
crnor	CROP		1P	EXTRA3	d:BT	s:BA	s:BB	0	0	0	0	0	BA_BB	BT	0
crandc	CROP		1P	EXTRA3	d:BT	s:BA	s:BB	0	0	0	0	0	BA_BB	BT	0
crxor	CROP		1P	EXTRA3	d:BT	s:BA	s:BB	0	0	0	0	0	BA_BB	BT	0
crnand	CROP		1P	EXTRA3	d:BT	s:BA	s:BB	0	0	0	0	0	BA_BB	BT	0
crand	CROP		1P	EXTRA3	d:BT	s:BA	s:BB	0	0	0	0	0	BA_BB	BT	0
creqv	CROP		1P	EXTRA3	d:BT	s:BA	s:BB	0	0	0	0	0	BA_BB	BT	0
crorc	CROP		1P	EXTRA3	d:BT	s:BA	s:BB	0	0	0	0	0	BA_BB	BT	0
cror	CROP		1P	EXTRA3	d:BT	s:BA	s:BB	0	0	0	0	0	BA_BB	BT	0
cmp	NORMAL		1P	EXTRA3	d:BF	s:RA	s:RB	0	RA	RB	0	0	0	BF	0
cmpl	NORMAL		1P	EXTRA3	d:BF	s:RA	s:RB	0	RA	RB	0	0	0	BF	0
cmprb	NORMAL		1P	EXTRA3	d:BF	s:RA	s:RB	0	RA	RB	0	0	0	BF	0
cmpeqb	NORMAL		1P	EXTRA3	d:BF	s:RA	s:RB	0	RA	RB	0	0	0	BF	0
0/0=fcmpu	NORMAL		1P	EXTRA3	d:BF	s:FRA	s:FRB	0	FRA	FRB	0	0	0	BF	0
1/0=fcmpo	NORMAL		1P	EXTRA3	d:BF	s:FRA	s:FRB	0	FRA	FRB	0	0	0	BF	0
4/0=ftdiv	NORMAL		1P	EXTRA3	d:BF	s:FRA	s:FRB	0	FRA	FRB	0	0	0	BF	0
bmask	NORMAL		1P	EXTRA3	d:RT	s:RA	s:RB	0	RA	RB	0	RT	0	0	0
bpermd	NORMAL		1P	EXTRA3	d:RA	s:RS	s:RB	0	RS	RB	0	RA	0	0	0
modud	NORMAL		1P	EXTRA3	d:RT	s:RA	s:RB	0	RA	RB	0	RT	0	0	0

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
moduw	NORMAL		1P	EXTRA3	d:RT	s:RA	s:RB	0	RA	RB	0	RT	0	0	0
cmpb	NORMAL		1P	EXTRA3	d:RA	s:RS	s:RB	0	RS	RB	0	RA	0	0	0
modsd	NORMAL		1P	EXTRA3	d:RT	s:RA	s:RB	0	RA	RB	0	RT	0	0	0
modsw	NORMAL		1P	EXTRA3	d:RT	s:RA	s:RB	0	RA	RB	0	RT	0	0	0
26/6=fmrgow	NORMAL		1P	EXTRA3	d:FRT	s:FRA	s:FRB	0	FRA	FRB	0	FRT	0	0	0
30/6=fmrgew	NORMAL		1P	EXTRA3	d:FRT	s:FRA	s:FRB	0	FRA	FRB	0	FRT	0	0	0
rlwnm	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RB	s:RS	0	0	RB	RS	RA	0	CR0	0
minu	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
maxu	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
mins	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
cprop	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
maxs	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
absds	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
absdu	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
avgadd	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
rldcl	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RB	s:RS	0	0	RB	RS	RA	0	CR0	0
rldcr	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RB	s:RS	0	0	RB	RS	RA	0	CR0	0
subfc	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
mulhdu	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
addc	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
mulhwu	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
slw	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RB	s:RS	0	0	RB	RS	RA	0	CR0	0
sld	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RB	s:RS	0	0	RB	RS	RA	0	CR0	0
and	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RB	s:RS	0	RS	RB	0	RA	0	CR0	0
subf	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
andc	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RB	s:RS	0	RS	RB	0	RA	0	CR0	0
mulhd	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
addg6s	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
mulhw	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
nor	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RB	s:RS	0	RS	RB	0	RA	0	CR0	0
subfe	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
adde	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
mulld	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
mullw	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
add	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
eqv	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RB	s:RS	0	RS	RB	0	RA	0	CR0	0
xor	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RB	s:RS	0	RS	RB	0	RA	0	CR0	0
divdeu	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
divweu	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
orc	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RB	s:RS	0	RS	RB	0	RA	0	CR0	0
divde	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
divwe	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
or	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RB	s:RS	0	RS	RB	0	RA	0	CR0	0
divdu	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
divwu	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
nand	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RB	s:RS	0	RS	RB	0	RA	0	CR0	0
divd	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
divw	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
subfco	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
mulhdu	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
addco	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
mulhwu	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
srw	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RB	s:RS	0	0	RB	RS	RA	0	CR0	0
srd	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RB	s:RS	0	0	RB	RS	RA	0	CR0	0
subfo	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
mulhd	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
mulhw	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
subfeo	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
addeo	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
mulldo	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
mullwo	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
addo	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
sraw	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RB	s:RS	0	0	RB	RS	RA	0	CR0	0
srad	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RB	s:RS	0	0	RB	RS	RA	0	CR0	0
divdeuo	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
divweuo	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
divdeo	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
divweo	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
divduo	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
divwuo	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
divdo	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
divwo	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
grev	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
grevw	NORMAL		1P	EXTRA3	d:RT;d:CR0	s:RA	s:RB	0	RA	RB	0	RT	0	CR0	0
ffadds	NORMAL		1P	EXTRA3	d:FRT;d:CR1	s:FRA	s:FRB	0	FRA	FRB	0	FRT	0	CR1	0
fdivs	NORMAL		1P	EXTRA3	d:FRT;d:CR1	s:FRA	s:FRB	0	FRA	FRB	0	FRT	0	CR1	0
fsubs	NORMAL		1P	EXTRA3	d:FRT;d:CR1	s:FRA	s:FRB	0	FRA	FRB	0	FRT	0	CR1	0
fadds	NORMAL		1P	EXTRA3	d:FRT;d:CR1	s:FRA	s:FRB	0	FRA	FRB	0	FRT	0	CR1	0
fmuls	NORMAL		1P	EXTRA3	d:FRT;d:CR1	s:FRA	s:FRC	0	FRA	0	FRC	FRT	0	CR1	0
fdiv	NORMAL		1P	EXTRA3	d:FRT;d:CR1	s:FRA	s:FRB	0	FRA	FRB	0	FRT	0	CR1	0
fsub	NORMAL		1P	EXTRA3	d:FRT;d:CR1	s:FRA	s:FRB	0	FRA	FRB	0	FRT	0	CR1	0
fadd	NORMAL		1P	EXTRA3	d:FRT;d:CR1	s:FRA	s:FRB	0	FRA	FRB	0	FRT	0	CR1	0
fmul	NORMAL		1P	EXTRA3	d:FRT;d:CR1	s:FRA	s:FRC	0	FRA	0	FRC	FRT	0	CR1	0
0/8=fcpsgn	NORMAL		1P	EXTRA3	d:FRT;d:CR1	s:FRA	s:FRB	0	FRA	FRB	0	FRT	0	CR1	0
rlwimi	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RA	s:RS	0	RA	0	RS	RA	0	CR0	0
rldimi	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RA	s:RS	0	RA	0	RS	RA	0	CR0	0
rldimi	NORMAL		1P	EXTRA3	d:RA;d:CR0	s:RA	s:RS	0	RA	0	RS	RA	0	CR0	0

F.4.8 RM-1P-1D

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
fmvis	NORMAL		1P	EXTRA3	d:FRS	0	0	0	0	0	0	FRS	0	0	0
svstep	NORMAL		1P	EXTRA3	d:RT;d:CR0	0	0	0	0	0	0	RT	0	CR0	0

F.4.9 RM-1P-1S

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
tdi	NORMAL		1P	EXTRA2	0	0	0	0	RA	0	0	0	0	0	0
twi	NORMAL		1P	EXTRA2	0	0	0	0	RA	0	0	0	0	0	0

F.4.10 RM-2P-2S1D

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
mtcrf/mtocrf	NORMAL		2P	EXTRA2	d:CR	s:RS	s:CR	0	RS	0	0	0	WHOLE_REG	WHOLE_REG	0

F.4.11 RM-1P-3S1D

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
isel	NORMAL		1P	EXTRA2	d:RT	s:RA	s:RB	s:BC	RA_OR_ZERO	RB	0	RT	BC	0	0
absdacs	NORMAL		1P	EXTRA2	d:RT;d:CR0	s:RA	s:RB	s:RT	RA	RB	RT	RT	0	CR0	0
absdacu	NORMAL		1P	EXTRA2	d:RT;d:CR0	s:RA	s:RB	s:RT	RA	RB	RT	RT	0	CR0	0

insn	mode	CONDITIONS	Ptype	Etype	0	1	2	3	in1	in2	in3	out	CR in	CR out	out2
ternlogi	NORMAL		1P	EXTRA2	d:RT;d:CR0	s:RA	s:RB	s:RT	RA	RB	RT	RT	0	CR0	0
ffmsubs	NORMAL		1P	EXTRA2	d:FRT;d:CR1	s:FRA	s:FRB	s:FRC	FRA	FRB	FRC	FRT	0	CR1	0
ffmadds	NORMAL		1P	EXTRA2	d:FRT;d:CR1	s:FRA	s:FRB	s:FRC	FRA	FRB	FRC	FRT	0	CR1	0
ffnmsubs	NORMAL		1P	EXTRA2	d:FRT;d:CR1	s:FRA	s:FRB	s:FRC	FRA	FRB	FRC	FRT	0	CR1	0
ffnmadds	NORMAL		1P	EXTRA2	d:FRT;d:CR1	s:FRA	s:FRB	s:FRC	FRA	FRB	FRC	FRT	0	CR1	0
fdmadds	NORMAL		1P	EXTRA2	d:FRT;d:CR1	s:FRA	s:FRB	s:FRC	FRA	FRB	FRC	FRT	0	CR1	0
fmsubs	NORMAL		1P	EXTRA2	d:FRT;d:CR1	s:FRA	s:FRB	s:FRC	FRA	FRB	FRC	FRT	0	CR1	0
fmadds	NORMAL		1P	EXTRA2	d:FRT;d:CR1	s:FRA	s:FRB	s:FRC	FRA	FRB	FRC	FRT	0	CR1	0
fnmsubs	NORMAL		1P	EXTRA2	d:FRT;d:CR1	s:FRA	s:FRB	s:FRC	FRA	FRB	FRC	FRT	0	CR1	0
fnmadds	NORMAL		1P	EXTRA2	d:FRT;d:CR1	s:FRA	s:FRB	s:FRC	FRA	FRB	FRC	FRT	0	CR1	0
fsel	NORMAL		1P	EXTRA2	d:FRT;d:CR1	s:FRA	s:FRB	s:FRC	FRA	FRB	FRC	FRT	0	CR1	0
fmsub	NORMAL		1P	EXTRA2	d:FRT;d:CR1	s:FRA	s:FRB	s:FRC	FRA	FRB	FRC	FRT	0	CR1	0
fmadd	NORMAL		1P	EXTRA2	d:FRT;d:CR1	s:FRA	s:FRB	s:FRC	FRA	FRB	FRC	FRT	0	CR1	0
fnmsub	NORMAL		1P	EXTRA2	d:FRT;d:CR1	s:FRA	s:FRB	s:FRC	FRA	FRB	FRC	FRT	0	CR1	0
fnmadd	NORMAL		1P	EXTRA2	d:FRT;d:CR1	s:FRA	s:FRB	s:FRC	FRA	FRB	FRC	FRT	0	CR1	0

Part III

Scalar Instructions

Preamble

As explained in the Simple-V introduction these are all intentionally and specifically Scalar instructions. Each section is free-standing, has no connection, dependence or relationship to any other section, including no direct critical dependence either way on Simple-V. They have with almost no exceptions been specifically crafted to have a justification for their inclusion in the Power ISA as Scalar instructions purely on their own merit.

- The biginteger multiply-and-add instruction is similar to Intel's mulx in that it produces a pair of results.
- JavaScript(tm) rounding is present in ARM as fjcvtzs and would save an astounding 35 instructions with 5 branches.
- Whilst there exist CR bit manipulation and copying instructions there are no CR Field manipulation instructions, putting pressure on GPRs if several CR fits need to be analysed.
- one single instruction, bmask, is proposed that covers the whole of x86 BMI1 and AMD TBM, combined, and provides more.

All of these have nothing to do with Simple-V at all: they make the Power ISA better at modern general-purpose compute, bringing it up-to-date.

That said: by a wonderful coincidence, should they be included, then Simple-V's capabilities increase significantly. For example the CRweird instructions combined with the bitmanip instructions, alongside Vectorised Rc=1 turn CR Fields into extremely powerful Predicate masks. bmask not only covers the BMI and TBM instructions of Intel and AMD it also includes the RVV set-before-first and set-after-first instructions.

The clean and clear separation between Vectorisation Prefix and Scalar Suffix is what makes it possible for both Scalar-only and Scalable-Vectors to benefit. It also makes proposal much easier, as there is no inter-dependence.

It is however important to note that the rationale for these instructions comes from a more general-purpose modern computing paradigm that is outside of IBM's much more focussed and specialist traditional customer base. We deeply respect IBM's curator role of the Power ISA of the past 25 years as much as we appreciate their courage in transferring that role to the OpenPOWER Foundation ISA Working Group.

Chapter 1

SV Vector-assist Scalar ops

[[!tag standards]]

1.1 SV Vector-assist Operations.

Links:

- [[discussion]]
- https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc#vector-register-gather-instructions
- https://lists.libre-soc.org/pipermail/libre-soc-dev/2022-May/004884.html
- https://bugs.libre-soc.org/show_bug.cgi?id=865 implementation in simulator
- https://bugs.libre-soc.org/show_bug.cgi?id=213
- https://bugs.libre-soc.org/show_bug.cgi?id=142 specialist vector ops out of scope for this document [[openpower/sv/3d_vector_ops]]
- [[simple_v_extension/specification/bitmanip]] previous version, contains pseudocode for sof, sif, sbf
- https://en.m.wikipedia.org/wiki/X86_Bit_manipulation_instruction_set#TBM_(Trailing_Bit_Manipulation)

The core Power ISA was designed as scalar: SV provides a level of abstraction to add variable-length element-independent parallelism. Therefore there are not that many cases where *actual* Vector instructions are needed. If they are, they are more "assistance" functions. Two traditional Vector instructions were initially considered (conflictd and vmiota) however they may be synthesised from existing SVP64 instructions: vmiota may use [[svstep]]. Details in [[discussion]]

Notes:

- Instructions suited to 3D GPU workloads (dotproduct, crossproduct, normalise) are out of scope: this document is for more general-purpose instructions that underpin and are critical to general-purpose Vector workloads (including GPU and VPU)
- Instructions related to the adaptation of CRs for use as predicate masks are covered separately, by crweird operations. See {CR Weird ops}.

1.1.1 Mask-suited Bitmanipulation

BM2-Form

0..5	6..10	11..15	16..20	21-25	26	27..31	Form
PO	RS	RA	RB	bm	L	XO	BM2-Form

- bmask RS,RA,RB,bm,L

Pseudo-code:

```
if _RB = 0 then mask <- [1] * XLEN
else            mask <- (RB)
ra <- (RA) & mask
a1 <- ra
if bm[4] = 0 then a1 <- ¬ra
mode2 <- bm[2:3]
if mode2 = 0 then a2 <- (¬ra)+1
if mode2 = 1 then a2 <- ra-1
if mode2 = 2 then a2 <- ra+1
if mode2 = 3 then a2 <- ¬(ra+1)
a1 <- a1 & mask
a2 <- a2 & mask
# select operator
mode3 <- bm[0:1]
if mode3 = 0 then result <- a1 | a2
if mode3 = 1 then result <- a1 & a2
if mode3 = 2 then result <- a1 ^ a2
if mode3 = 3 then result <- undefined([0]*XLEN)
# mask output
result <- result & mask
# optionally restore masked-out bits
if L = 1 then
    result <- result | (RA & ¬mask)
RT <- result
```

- first pattern A: two options `x` or `~x`
- second pattern B: three options `|` `&` or `^`
- third pattern C: four options `x+1`, `x-1`, `~(x+1)` or `(~x)+1`

The lower two bits of `bm` set to 0b11 are `RESERVED`. An illegal instruction trap must be raised.

Special Registers Altered:

```
None
```

1.1.2 Carry-lookahead

As a single scalar 32-bit instruction, up to 64 carry-propagation bits may be computed. When the output is then used as a Predicate mask it can be used to selectively perform the "add carry" of biginteger math, with `sv.addi/sm=rN RT.v, RA.v, 1`.

- cprop RT,RA,RB (Rc=0)
- cprop. RT,RA,RB (Rc=1)

pseudocode:

```
P = (RA)
G = (RB)
RT = ((P|G)+G)^P
```

X-Form

0:5	6:10	11:15	16:20	21:30	31	name	Form
PO	RT	RA	RB	XO	Rc	cprop	X-Form

used not just for carry lookahead, also a special type of predication mask operation.

Chapter 2

CR Weird ops

2.1 New instructions for CR/INT predication

See:

- main bugreport for crweirds https://bugs.libre-soc.org/show_bug.cgi?id=533
- https://bugs.libre-soc.org/show_bug.cgi?id=527
- https://bugs.libre-soc.org/show_bug.cgi?id=569
- https://bugs.libre-soc.org/show_bug.cgi?id=558#c47
- [[discussion]]

2.1.1 crrweird

CW2-Form

```
|0     |6   |9 |11|12   |16   |19  |22   |26   |31|
| PO   | RT    |M |fmsk |BFA  |XO  |fmap | XO  |Rc|
```

- crrweird RT,BFA,M,fmsk,fmap (Rc=0)
- crrweird. RT,BFA,M,fmsk,fmap (Rc=1)

```
creg <- CR[4*BFA+32:4*BFA+35]
n <- (¬fmap ^ creg) & fmsk
result <- (n != 0) if M else (n == fmsk)
RT <- [0] * 63 || result
if Rc then
    CR0 <- analyse(RT)
```

When used with SVP64 Prefixing this is a {Arithmetic Mode} SVP64 type operation and as such can use Rc=1 and RC1 Data-dependent Mode capability

Also as noted below, element-width override bits normally used on the source is instead used to allow multiple results to be packed sequentially into the destination. *Destination elwidth overrides still apply.*

Special registers altered:

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

2.1.2 mfcrrweird

CW2-Form

```
|0     |6   |9 |11|12   |16   |19  |22   |26   |31|
| PO   | RT    |M |fmsk |BFA  |XO  |fmap | XO  |Rc|
```

- mfcrrweird RT,BFA,fmsk,fmap (Rc=0)
- mfcrrweird. RT,BFA,fmsk,fmap (Rc=1)

```
creg = CR[4*BFA+32:4*BFA+35]
result = (¬fmap ^ creg) & fmsk
RT = [0] * 60 || result
If Rc:
    CR0 = analyse(RT)
```

When used with SVP64 Prefixing this is a {Arithmetic Mode} SVP64 type operation and as such can use Rc=1 and RC1 Data-dependent Mode capability.

Also as noted below, element-width override bits normally used on the source is instead used to allow multiple results to be packed into the destination. *Destination elwidth overrides still apply*

2.1.3 mtcrrweird

CW-Form

```
|0     |6   |9 |11|12   |16   |19  |22   |26   |31|
| PO   | RA    |M |fmsk |BF   |XO  |fmap | XO  |  |
| PO   | BT    |M |fmsk |BF   |XO  |fmap | XO  |  |
| PO   | BF |  |M |fmsk |BF   |XO  |fmap | XO  |  |
```

- mtcrrweird BF,RA,M,fmsk,fmap

```
a = (RA|0)
creg = a[60:63]
result = (¬fmap ^ creg) & fmsk
if M:
    result |= CR[4*BF+32:4*BF+35]  & ~fmsk
CR[4*BF+32:4*BF+35]  = result
```

When used with SVP64 Prefixing this is a {Arithmetic Mode} SVP64 type operation and as such can use RC1 Data-dependent Mode capability

Hardware Architectural Note: when M=1 this instruction is a Read-Modify-Write on the `BF` CR Field. When M=0 it is a more normal Write.

Special Registers Altered:

```
CR Field BF
```

2.1.4 mtcrweird

CW-Form

```
|0     |6   |9 |11|12   |16   |19  |22   |26   |31|
| PO   | RA    |M |fmsk |BF   |XO  |fmap | XO  |  |
| PO   | BT    |M |fmsk |BF   |XO  |fmap | XO  |  |
| PO   | BF |  |M |fmsk |BF   |XO  |fmap | XO  |  |
```

- mtcrweird BF,RA,M,fmsk,fmap

```
reg = (RA|0)
creg = reg[63] || reg[63] || reg[63] || reg[63]
result = (¬fmap ^ creg) & fmsk
```

```
if M:
    result |= CR[4*BF+32:4*BF+35] & ~fmsk
CR[4*BF+32:4*BF+35]  = result
```

Note that when M=1 this operation is a Read-Modify-Write on the CR Field BF. Masked-out bits of the 4-bit CR Field BF will not be changed when M=1. Correspondingly when M=0 this operation is an overwrite: no read of BF is required because the masked-out bits of the BF CR Field are set to zero.

When used with SVP64 Prefixing this is a {Condition Register Fields Mode} SVP64 type operation that has 3-bit Data-dependent and 3-bit Predicate-result capability (BF is 3 bits)

Special Registers Altered:

```
CR Field BF
```

2.1.5 mcrfm - Move CR Field, masked.

CW-Form

```
|0     |6   |9 |11|12   |16   |19  |22   |26   |31|
| PO   | BF |  |M |fmsk |BF   |XO  |fmap | XO     |
```

- mcrfm: BF,BFA,M,fmsk,fmap

```
result = fmsk & CR[4*BFA+32:4*BFA+35]
if M:
    result |= CR[4*BF+32:4*BF+35]  & ~fmsk
result ^= fmap
CR[4*BF+32:4*BF+35]  = result
```

This instruction copies, sets, or inverts parts of a CR Field into another CR Field. `mcrf` copies only one bit of the CR from any arbitrary bit to any other arbitrary bit, whereas `mcrfm` copies an entire 4-bit CR Field (or masked parts thereof). Unlike `mcrf` the bits of the CR Field may not change position: the EQ bit from the source may only go into the EQ bit of the destination (optionally inverted, set, or cleared).

When M=1 this operation is a Read-Modify-Write on the CR Field BF. Masked-out bits of the 4-bit CR Field BF will not be changed when M=1. Correspondingly when M=0 this operation is an overwrite: no read of BF is required because the masked-out bits of the BF CR Field are set to zero.

When used with SVP64 Prefixing this is a {Condition Register Fields Mode} SVP64 type operation that has 3-bit Data-dependent and 3-bit Predicate-result capability (BF is 3 bits)

*Programmer's note: **`fmap`** being XORed onto the result provides considerable flexibility. individual bits of BFA may be copied inverted to BF by ensuring that **`fmsk`** and **`fmap`** have the same bit set. Also, individual bits in BF may be set to 1 by ensuring that the required bit of **`fmsk`** is set to zero and the same bit in **`fmap`** is set to 1*

Special Registers Altered:

```
CR Field BF
```

2.1.6 crweirder

```
|0     |6   |9 |11|12   |16   |19  |22   |26   |31|
| PO   | BT |  |M |fmsk |BF   |XO  |fmap | XO     |
```

- crweirder: BT,BFA,fmsk,fmap

```
creg = CR[4*BFA+32:4*BFA+35]
n = (¬fmap ^ creg) & fmsk
result = (n != 0) if M else (n == fmsk)
CR[32+BT] = result
```

Special Registers Altered:

CR[BT+32]

When used with SVP64 Prefixing this is a {Condition Register Fields Mode} SVP64 type operation that has 5-bit Data-dependent capability (BT is 5 bits)

Hardware Architectural Note: this instruction is always a Read-Modify-Write on the CR Field containing BT.

Example Pseudo-ops:

```
mtcri BF, fmap    mtcrweird BF, r0, 0, 0b1111,~fmap
mtcrset BF, fmsk  mtcrweird BF, r0, 1, fmsk,0b0000
mtcrclr BF, fmsk  mtcrweird BF, r0, 1, fmsk,0b1111
```

2.2 Vectorised versions involving GPRs

The name "weird" refers to a minor violation of SV rules when it comes to deriving the Vectorised versions of these instructions.

Normally the progression of the SV for-loop would move on to the next register. Instead however in the scalar case these instructions **remain in the same register** and insert or transfer between **bits** of the scalar integer source or destination. The reason is that when using CR Fields as predicate masks and there is a need to transfer into a GPR, again for use as a predicate mask, the CR Field bits need to be efficiently packed into that one GPR (r3, r10 or r31).

Further useful violation of the normal SV Elwidth override rules allows for packing (or unpacking) of multiple CR test results into (or out of) an Integer Element. Note that the CR (source operand) elwidth field is utilised to determine the bit- packing size (1/2/4/8 with remaining bits within the Integer element set to zero) whilst the INT (dest operand) elwidth field still sets the Integer element size as usual (8/16/32/default)

sv.crrweird: RT, BB, fmsk, fmap

```
for i in range(VL):
    if BB.isvec: # Vector CR Field source?
        creg = CR{BB+i}
    else:
        creg = CR{BB}
    n = (¬fmap ^ creg) & fmsk
    result = (n != 0) if M else (n == fmsk)
    if RT.isvec:
        # TODO: RT.elwidth override to be also added here
        # note, yes, really, the CR's elwidth field determines
        # the bit-packing into the INT!
        if BB.elwidth == 0b00:
            # pack 1 result into 64-bit registers
            iregs[RT+i][0..62] = 0
            iregs[RT+i][63] = result # sets LSB to result
        if BB.elwidth == 0b01:
            # pack 2 results sequentially into INT registers
            iregs[RT+i//2][0..61] = 0
            iregs[RT+i//2][63-(i%2)] = result
        if BB.elwidth == 0b10:
            # pack 4 results sequentially into INT registers
            iregs[RT+i//4][0..59] = 0
            iregs[RT+i//4][63-(i%4)] = result
        if BB.elwidth == 0b11:
            # pack 8 results sequentially into INT registers
            iregs[RT+i//8][0..55] = 0
            iregs[RT+i//8][63-(i%8)] = result
    else:
        # scalar RT destination: exceeding VL=64 is UNDEFINED
        iregs[RT][63-i] = result # results also in scalar INT
        # only mapreduce mode (/mr) allows continuation here
        if not SVRM.mapreduce: break
```

Note that:

- in the scalar case the CR-Vector assessment is stored bit-wise starting at the LSB of the destination scalar INT
- in the INT-vector case the results are packed into LSBs of the INT Elements, the packing arrangement depending on both elwidth override settings.

mfcrrweird: RT, BFA, fmsk.fmap

Unlike `crrweird` the results are 4-bit wide, so the packing will begin to spill over to other destination elements. 8 results per destination at 4-bits each still fits into destination elwidth at 32-bit, but for 16-bit and 8-bit obviously this does not fit, and must split across to the next element

When for example destination elwidth is 16-bit (0b10) the following packing occurs:

- SVRM bits 6:7 equal to 0b00 - one 4-bit result element packed into the first 4-bits of the 16-bit destination element (in the first 4 LSBs)
- SVRM bits 6:7 equal to 0b01 - two 4-bit result elements packed into the first 8-bits of the 16-bit destination element (in the first 8 LSBs)
- SVRM bits 6:7 equal to 0b10 - four 4-bit result elements packed into each 16-bit destination element
- SVRM bits 6:7 equal to 0b11 - eight 4-bit result elements, the first four of which are packed into the first 16-bit destination element, the second four of which are packed into the second 16-bit destination element.

Pseudocode example: note that dest elwidth overrides affect the packing of results. BB.elwidth in effect requests how many 4-bit result elements would like to be packed, but RT.elwidth determines the limit. Any parts of the destination elements not containing results are set to zero.

```
for i in range(VL):
    if BB.isvec:
        creg = CR{BB+i}
    else:
        creg = CR{BB}
    result = (¬fmap ^ creg) & fmsk # 4-bit result
    if RT.isvec:
        # RT.elwidth override can affect the packing
        bwid = {0b00:64, 0b01:8, 0b10:16, 0b11:32}[RT.elwidth]
        t4, t8 = min(4, bwid//2), min(8, bwid//2)
        # yes, really, the CR's elwidth field determines
        # the bit-packing into the INT!
        if BB.elwidth == 0b00:
            # pack 1 result into 64-bit registers
            idx, boff = i, 0
        if BB.elwidth == 0b01:
            # pack 2 results sequentially into INT registers
            idx, boff = i//2, i%2
        if BB.elwidth == 0b10:
            # pack 4 results sequentially into INT registers
            idx, boff = i//t4, i%t4
        if BB.elwidth == 0b11:
            # pack 8 results sequentially into INT registers
            idx, boff = i//t8, i%t8
    else:
        # scalar RT destination: exceeding VL=16 is UNDEFINED
        idx, boff = 0, i
    # store 4-bit result in Vector starting from RT
    iregs[RT+idx][60-boff*4:63-boff*4] = result
    if not RT.isvec:
        # only mapreduce mode (/mr) allows continuation here
        if not SVRM.mapreduce: break
```

2.3 Predication Examples

Take the following example:

```
r10 = 0b00010
sv.mtcrweird/dm=r10/dz cr8.v, 0, 0b0011.0000
```

Here, RA is zero, so the source input is zero. The destination is CR Field 8, and the destination predicate mask indicates to target the first two elements. Destination predicate zeroing is enabled, and the destination predicate is only set in the 2nd bit. fmsk is 0b0011, fmap is all zeros.

Let us first consider what should go into element 0 (CR Field 8):

- The destination predicate bit is zero, and zeroing is enabled.
- Therefore, what is in the source is irrelevant: the result must be zero.
- Therefore all four bits of CR Field 8 are therefore set to zero.

Now the second element, CR Field 9 (CR9):

- Bit 2 of the destination predicate, r10, is 1. Therefore the computation of the result is relevant.
- RA is zero therefore bit 2 is zero. fmsk is 0b0011 and fmap is 0b0000
- When calculating n0 thru n3 we get n0=1, n1=2, n2=0, n3=0
- Therefore, CR9 is set (using LSB0 ordering) to 0b0011, i.e. to fmsk.

It should be clear that this instruction uses bits of the integer predicate to decide whether to set CR Fields to `(fmsk & ~fmap)` or to zero. Thus, in effect, it is the integer predicate that has been copied into the CR Fields.

By using twin predication, zeroing, and inversion (sm=~r3, dm=r10) for example, it becomes possible to combine two Integers together in order to set bits in CR Fields. Likewise there are dozens of ways that CR Predicates can be used, on the same sv.mtcrweird instruction.

[[!tag standards]]

Chapter 3

Bitmanip ops

[[!tag standards]]

[[!toc levels=1]]

3.1 Implementation Log

- ternlogi https://bugs.libre-soc.org/show_bug.cgi?id=745
- grev https://bugs.libre-soc.org/show_bug.cgi?id=755
- GF2^M https://bugs.libre-soc.org/show_bug.cgi?id=782
- binutils https://bugs.libre-soc.org/show_bug.cgi?id=836
- shift-and-add https://bugs.libre-soc.org/show_bug.cgi?id=968

3.2 bitmanipulation

DRAFT STATUS

pseudocode: [[openpower/isa/bitmanip]]

this extension amalgamates bitmanipulation primitives from many sources, including RISC-V bitmanip, Packed SIMD, AVX-512 and OpenPOWER VSX. Also included are DSP/Multimedia operations suitable for Audio/Video. Vectorisation and SIMD are removed: these are straight scalar (element) operations making them suitable for embedded applications. Vectorisation Context is provided by {Scalable Vectors for Power ISA}.

When combined with SV, scalar variants of bitmanip operations found in VSX are added so that the Packed SIMD aspects of VSX may be retired as "legacy" in the far future (10 to 20 years). Also, VSX is hundreds of opcodes, requires 128 bit pathways, and is wholly unsuited to low power or embedded scenarios.

ternlogv is experimental and is the only operation that may be considered a "Packed SIMD". It is added as a variant of the already well-justified ternlog operation (done in AVX512 as an immediate only) "because it looks fun". As it is based on the LUT4 concept it will allow accelerated emulation of FPGAs. Other vendors of ISAs are buying FPGA companies to achieve similar objectives.

general-purpose Galois Field 2^M operations are added so as to avoid huge custom opcode proliferation across many areas of Computer Science. however for convenience and also to avoid setup costs, some of the more common operations (clmul, crc32) are also added. The expectation is that these operations would all be covered by the same pipeline.

note that there are brownfield spaces below that could incorporate some of the set-before-first and other scalar operations listed in {Swizzle Move}, {SV Vector ops}, {FP/Int Conversion ops} and the {Audio and Video Opcodes} as well as {setvl instruction}, {svstep instruction}, {REMAP subsystem}

Useful resource:

- https://en.wikiversity.org/wiki/Reed%E2%80%93Solomon_codes_for_coders
- https://maths-people.anu.edu.au/~brent/pd/rpb232tr.pdf
- https://gist.github.com/animetosho/d3ca95da2131b5813e16b5bb1b137ca0
- https://github.com/HJLebbink/asm-dude/wiki/GF2P8AFFINEINVQB

3.3 Draft Opcode tables

[[sv/draft_opcode_tables]]

two major opcodes are needed

ternlog has its own major opcode

29.30	31	name	Form
0 0	Rc	ternlogi	TLI-Form
0 1		crternlogi	TLI-Form
1 iv		grevlogi	TLI-Form

2nd major opcode for other bitmanip: minor opcode allocation

28.30	31	name
-00	0	xpermi
-00	1	binary lut
-01	0	grevlog
-01	1	swizzle mv/fmv
010	Rc	bitmask
011		SVP64
110	Rc	1/2-op
111		bmrevi

`minmax` is allocated to PO 19 XO `000011`

1-op and variants

dest	src1	subop	op
RT	RA	..	bmatflip

2-op and variants

dest	src1	src2	subop	op
RT	RA	RB	or	bmatflip
RT	RA	RB	xor	bmatflip
RT	RA	RB		grev
RT	RA	RB		clmul*
RT	RA	RB		gorc

dest	src1	src2	subop	op
RT	RA	RB	shuf	shuffle
RT	RA	RB	unshuf	shuffle
RT	RA	RB	width	xperm
RT	RA	RB	MMM	`minmax`
RT	RA	RB		av abs avgadd
RT	RA	RB	type	vmask ops
RT	RA	RB	type	abs accumulate (overwrite)

3 ops

- grevlog[w]
- GF mul-add
- bitmask-reverse

TODO: convert all instructions to use RT and not RS

0.5	6.10	11.15	16.20	21..25	26....30	31	name	Form
NN	RT	RA	it/im57	im0-4	0 00 00	0	xpermi	TODO-Form
NN	RT	RA	RB	RC	sh 01 00	0	maddsubrs	BF-Form
NN					- 10 00	0	rsvd	rsvd
NN					- 11 00	0	rsvd	rsvd
NN	RT	RA	RB	RC	nh 00 00	1	binlut	VA-Form
NN	RT	RA	RB	/BFA/	0 01 00	1	bincrflut	VA-Form
NN					1 01 00	1	svindex	SVI-Form
NN	RT	RA	RB	mode	L 10 00	1	bmask	BM2-Form
NN					0 11 00	1	svshape	SVM-Form
NN					1 11 00	1	svremap	SVRM-Form
NN	RT	RA	RB	im0-4	im5-7 01	0	grevlog	TLI-Form
NN					- – 01	1	swizzle mv/f	TODO
NN	RT	RA	RB	RC	mode 010	Rc	bitmask*	VA2-Form
NN	FRS	d1	d0	d0	00 011	d2	fmvis	DX-Form
NN	FRS	d1	d0	d0	01 011	d2	fishmv	DX-Form
NN					10 011	Rc	svstep	SVL-Form
NN					11 011	Rc	setvl	SVL-Form
NN					—- 110		1/2 ops	other table [1]
NN	RT	RA	RB	RC	11 110	Rc	bmrev	VA2-Form
NN	RT	RA	RB	sh0-4	sh5 1 111	Rc	bmrevi	MDS-Form

[1] except bmrev

ops (note that av avg and abs as well as vec scalar mask are included here {SV Vector ops}, and the {Audio and Video Opcodes})

0.5	6.10	11.15	16.20	21	22.23	24....30	31	name	Form
NN	RS	me	sh	SH	ME 0	nn00 110	Rc	bmopsi	BM-Form
NN	RS	RA	sh	SH	0 1	nn00 110	Rc	bmopsi	XB-Form
NN	RS	RA	im04	im5	1 1	im67 00 110	Rc	bmatxori	TODO
NN	RT	RA	RB	1	00	0001 110	Rc	cldiv	X-Form
NN	RT	RA	RB	1	01	0001 110	Rc	clmod	X-Form
NN	RT	RA		1	10	0001 110	Rc	clmulh	X-Form
NN	RT	RA	RB	1	11	0001 110	Rc	clmul	X-Form
NN	RT	RA	RB	0	00	0001 110	Rc	rsvd	

0.5	6.10	11.15	16.20	21	22.23	24....30	31	name	Form
NN	RT	RA	RB	0	01	0001 110	Rc	rsvd	
NN	RT	RA	RB	0	10	0001 110	Rc	rsvd	
NN	RT	RA	RB	0	11	0001 110	Rc	vec cprop	X-Form
NN					00	0101 110	0	crfbinlog	{TODO}
NN					00	0101 110	1	rsvd	
NN					10	0101 110	Rc	rsvd	
NN	RT	RA	RB	sm0	sm1 1	0101 110	Rc	shaddw	X-Form
NN				0		1001 110	Rc	rsvd	
NN	RT	RA	RB	1	00	1001 110	Rc	av abss	X-Form
NN	RT	RA	RB	1	01	1001 110	Rc	av absu	X-Form
NN	RT	RA	RB	1	10	1001 110	Rc	av avgadd	X-Form
NN	RT	RA	RB	1	11	1001 110	Rc	grevlutr	X-Form
NN	RT	RA	RB	sm0	sm1 0	1101 110	Rc	shadd	X-Form
NN	RT	RA	RB	sm0	sm1 1	1101 110	Rc	shadduw	X-Form
NN	RT	RA	RB	0	00	0010 110	Rc	rsvd	
NN	RS	RA	sh	SH	00	1010 110	Rc	rsvd	
NN	RT	RA	RB	0	00	0110 110	Rc	rsvd	
NN	RS	RA	SH	0	00	1110 110	Rc	rsvd	
NN	RT	RA	RB	1	00	1110 110	Rc	absds	X-Form
NN	RT	RA	RB	0	01	0010 110	Rc	rsvd	
NN	RT	RA	RB	1	01	0010 110	Rc	clmulr	X-Form
NN	RS	RA	sh	SH	01	1010 110	Rc	rsvd	
NN	RT	RA	RB	0	01	0110 110	Rc	rsvd	
NN	RS	RA	SH	0	01	1110 110	Rc	rsvd	
NN	RT	RA	RB	1	01	1110 110	Rc	absdu	X-Form
NN	RS	RA	RB	0	10	0010 110	Rc	bmator	X-Form
NN	RS	RA	RB	0	10	0110 110	Rc	bmatand	X-Form
NN	RS	RA	RB	0	10	1010 110	Rc	bmatxor	X-Form
NN	RS	RA	RB	0	10	1110 110	Rc	bmatflip	X-Form
NN	RT	RA	RB	1	10	0010 110	Rc	xpermn	X-Form
NN	RT	RA	RB	1	10	0110 110	Rc	xpermb	X-Form
NN	RT	RA	RB	1	10	1010 110	Rc	xpermh	X-Form
NN	RT	RA	RB	1	10	1110 110	Rc	xpermw	X-Form
NN	RT	RA	RB	0	11	1110 110	Rc	absdacs	X-Form
NN	RT	RA	RB	1	11	1110 110	Rc	absdacu	X-Form
NN						–11 110	Rc	bmrev	VA2-Form

3.4 binary and ternary bitops

Similar to FPGA LUTs: for two (binary) or three (ternary) inputs take bits from each input, concatenate them and perform a lookup into a table using an 8-8-bit immediate (for the ternary instructions), or in another register (4-bit for the binary instructions). The binary lookup instructions have CR Field lookup variants due to CR Fields being 4 bit.

Like the x86 AVX512F vpternlogd/vpternlogq instructions.

3.4.1 ternlogi

0.5	6.10	11.15	16.20	21..28	29.30	31
NN	RT	RA	RB	im0-7	00	Rc

```
lut3(imm, a, b, c):
    idx = c << 2 | b << 1 | a
    return imm[idx] # idx by LSB0 order

for i in range(64):
    RT[i] = lut3(imm, RB[i], RA[i], RT[i])
```

3.4.2 binlut

Binary lookup is a dynamic LUT2 version of ternlogi. Firstly, the lookup table is 4 bits wide not 8 bits, and secondly the lookup table comes from a register not an immediate.

0.5	6.10	11.15	16.20	21..25	26..31	Form
NN	RT	RA	RB	RC	nh 00001	VA-Form
NN	RT	RA	RB	/BFA/	0 01001	VA-Form

For binlut, the 4-bit LUT may be selected from either the high nibble or the low nibble of the first byte of RC:

```
lut2(imm, a, b):
    idx = b << 1 | a
    return imm[idx] # idx by LSB0 order

imm = (RC>>(nh*4))&0b1111
for i in range(64):
    RT[i] = lut2(imm, RB[i], RA[i])
```

For bincrlut, `BFA` selects the 4-bit CR Field as the LUT2:

```
for i in range(64):
    RT[i] = lut2(CRs{BFA}, RB[i], RA[i])
```

When Vectorised with SVP64, as usual both source and destination may be Vector or Scalar.

Programmer's note: a dynamic ternary lookup may be synthesised from a pair of ***binlut*** *instructions followed by a* ***ternlogi*** *to select which to merge. Use* ***nh*** *to select which nibble to use as the lookup table from the RC source register (****nh=1*** *nibble high), i.e. keeping an 8-bit LUT3 in RC, the first* ***binlut*** *instruction may set nh=0 and the second nh=1.*

3.4.3 crternlogi

another mode selection would be CRs not Ints.

CRB-Form:

0.5	6.8	9.10	11.13	14.15	16.18	19.25	26.30	31
NN	BF	msk	BFA	msk	BFB	TLI	XO	TLI

```
for i in range(4):
    a,b,c = CRs[BF][i], CRs[BFA][i], CRs[BFB][i])
    if msk[i] CRs[BF][i] = lut3(imm, a, b, c)
```

This instruction is remarkably similar to the existing crops, `crand` etc. which have been noted to be a 4-bit (binary) LUT. In effect `crternlogi` is the ternary LUT version of crops, having an 8-bit LUT. However it is an overwrite instruction in order to save on register file ports, due to the mask requiring the contents of the BF to be both read and written.

Programmer's note: This instruction is useful when combined with Matrix REMAP in "Inner Product" Mode, creating Warshall Transitive Closure that has many applications in Computer Science.

3.4.4 crbinlog

With ternary (LUT3) dynamic instructions being very costly, and CR Fields being only 4 bit, a binary (LUT2) variant is better

CRB-Form:

0.5	6.8	9.10	11.13	14.15	16.18	19.25	26.30	31
NN	BF	msk	BFA	msk	BFB	//	XO	//

```
for i in range(4):
    a,b = CRs[BF][i], CRs[BF][i])
    if msk[i] CRs[BF][i] = lut2(CRs[BFB], a, b)
```

When SVP64 Vectorised any of the 4 operands may be Scalar or Vector, including `BFB` meaning that multiple different dynamic lookups may be performed with a single instruction. Note that this instruction is deliberately an overwrite in order to reduce the number of register file ports required: like `crternlogi` the contents of `BF` **must** be read due to the mask only writing back to non-masked-out bits of `BF`.

Programmer's note: just as with binlut and ternlogi, a pair of crbinlog instructions followed by a merging crternlogi may be deployed to synthesise dynamic ternary (LUT3) CR Field manipulation

3.5 int ops

3.5.1 min/m

required for the {Audio and Video Opcodes}

signed and unsigned min/max for integer.

signed/unsigned min/max gives more flexibility.

[un]signed min/max instructions are specifically needed for vector reduce min/max operations which are pretty common.

X-Form

- PO=19, XO=—-000011 `minmax RT, RA, RB, MMM`
- PO=19, XO=—-000011 `minmax. RT, RA, RB, MMM`

see [[openpower/sv/rfc/ls013]] for `MMM` definition and pseudo-code.

implements all of (and more):

```
uint_xlen_t mins(uint_xlen_t rs1, uint_xlen_t rs2)
{ return (int_xlen_t)rs1 < (int_xlen_t)rs2 ? rs1 : rs2;
}
uint_xlen_t maxs(uint_xlen_t rs1, uint_xlen_t rs2)
{ return (int_xlen_t)rs1 > (int_xlen_t)rs2 ? rs1 : rs2;
}
uint_xlen_t minu(uint_xlen_t rs1, uint_xlen_t rs2)
{ return rs1 < rs2 ? rs1 : rs2;
}
uint_xlen_t maxu(uint_xlen_t rs1, uint_xlen_t rs2)
```

```
{ return rs1 > rs2 ? rs1 : rs2;
}
```

3.5.2 average

required for the {Audio and Video Opcodes}, these exist in Packed SIMD (VSX) but not scalar

```
uint_xlen_t intavg(uint_xlen_t rs1, uint_xlen_t rs2) {
     return (rs1 + rs2 + 1) >> 1:
}
```

3.5.3 absdu

required for the {Audio and Video Opcodes}, these exist in Packed SIMD (VSX) but not scalar

```
uint_xlen_t absdu(uint_xlen_t rs1, uint_xlen_t rs2) {
     return (src1 > src2) ? (src1-src2) : (src2-src1)
}
```

3.5.4 abs-accumulate

required for the {Audio and Video Opcodes}, these are needed for motion estimation. both are overwrite on RS.

```
uint_xlen_t uintabsacc(uint_xlen_t rs, uint_xlen_t ra, uint_xlen_t rb) {
     return rs + (src1 > src2) ? (src1-src2) : (src2-src1)
}
uint_xlen_t intabsacc(uint_xlen_t rs, int_xlen_t ra, int_xlen_t rb) {
     return rs + (src1 > src2) ? (src1-src2) : (src2-src1)
}
```

For SVP64, the twin Elwidths allows e.g. a 16 bit accumulator for 8 bit differences. Form is `RM-1P-3S1D` where RS-as-source has a separate SVP64 designation from RS-as-dest. This gives a limited range of non-overwrite capability.

3.6 shift-and-add

Power ISA is missing LD/ST with shift, which is present in both ARM and x86. Too complex to add more LD/ST, a compromise is to add shift-and-add. Replaces a pair of explicit instructions in hot-loops.

```
# 1.6.27 Z23-FORM
    |0     |6     |11    |15 |16     |21 |23    |31 |
    | PO   |  RT  |   RA     |   RB  |sm |   XO |Rc |
```

Pseudo-code (shadd):

```
n <- (RB)
m <- sm + 1
RT <- (n[m:XLEN-1] || [0]*m) + (RA)
```

Pseudo-code (shaddw):

```
shift <- sm + 1                # Shift is between 1-4
n <- EXTS((RB)[XLEN/2:XLEN-1]) # Only use lower XLEN/2-bits of RB
RT <- (n << shift) + (RA)      # Shift n, add RA
```

Pseudo-code (shadduw):

```
n <- ([0]*(XLEN/2)) || (RB)[XLEN/2:XLEN-1]
m <- sm + 1
RT <- (n[m:XLEN-1] || [0]*m) + (RA)
```

```
uint_xlen_t shadd(uint_xlen_t RA, uint_xlen_t RB, uint8_t sm) {
    sm = sm & 0x3;
    return (RB << (sm+1)) + RA;
}

uint_xlen_t shaddw(uint_xlen_t RA, uint_xlen_t RB, uint8_t sm) {
    uint_xlen_t n = (int_xlen_t)(RB << XLEN / 2) >> XLEN / 2;
    sm = sm & 0x3;
    return (n << (sm+1)) + RA;
}

uint_xlen_t shadduw(uint_xlen_t RA, uint_xlen_t RB, uint8_t sm) {
    uint_xlen_t n = RB & 0xFFFFFFFF;
    sm = sm & 0x3;
    return (n << (sm+1)) + RA;
}
```

3.7 bitmask set

based on RV bitmanip singlebit set, instruction format similar to shift [[isa/fixedshift]]. bmext is actually covered already (shift-with-mask rldicl but only immediate version). however bitmask-invert is not, and set/clr are not covered, although they can use the same Shift ALU.

bmext (RB) version is not the same as rldicl because bmext is a right shift by RC, where rldicl is a left rotate. for the immediate version this does not matter, so a bmexti is not required. bmrev however there is no direct equivalent and consequently a bmrevi is required.

bmset (register for mask amount) is particularly useful for creating predicate masks where the length is a dynamic runtime quantity. bmset(RA=0, RB=0, RC=mask) will produce a run of ones of length "mask" in a single instruction without needing to initialise or depend on any other registers.

0.5	6.10	11.15	16.20	21.25	26..30	31	name
NN	RS	RA	RB	RC	mode 010	Rc	bm*

Immediate-variant is an overwrite form:

0.5	6.10	11.15	16.20	21	22.23	24....30	31	name
NN	RS	RB	sh	SH	itype	1000 110	Rc	bm*i

```
def MASK(x, y):
     if x < y:
         x = x+1
         mask_a = ((1 << x) - 1) & ((1 << 64) - 1)
         mask_b = ((1 << y) - 1) & ((1 << 64) - 1)
     elif x == y:
         return 1 << x
     else:
         x = x+1
         mask_a = ((1 << x) - 1) & ((1 << 64) - 1)
```

```
            mask_b = (~((1 << y) - 1)) & ((1 << 64) - 1)
        return mask_a ^ mask_b


uint_xlen_t bmset(RS, RB, sh)
{
    int shamt = RB & (XLEN - 1);
    mask = (2<<sh)-1;
    return RS | (mask << shamt);
}

uint_xlen_t bmclr(RS, RB, sh)
{
    int shamt = RB & (XLEN - 1);
    mask = (2<<sh)-1;
    return RS & ~(mask << shamt);
}

uint_xlen_t bminv(RS, RB, sh)
{
    int shamt = RB & (XLEN - 1);
    mask = (2<<sh)-1;
    return RS ^ (mask << shamt);
}

uint_xlen_t bmext(RS, RB, sh)
{
    int shamt = RB & (XLEN - 1);
    mask = (2<<sh)-1;
    return mask & (RS >> shamt);
}
```

bitmask extract with reverse. can be done by bit-order-inverting all of RB and getting bits of RB from the opposite end.

when RA is zero, no shift occurs. this makes bmextrev useful for simply reversing all bits of a register.

```
msb = ra[5:0];
rev[0:msb] = rb[msb:0];
rt = ZE(rev[msb:0]);

uint_xlen_t bmrevi(RA, RB, sh)
{
    int shamt = XLEN-1;
    if (RA != 0) shamt = (GPR(RA) & (XLEN - 1));
    shamt = (XLEN-1)-shamt;  # shift other end
    brb = bitreverse(GPR(RB))     # swap LSB-MSB
    mask = (2<<sh)-1;
    return mask & (brb >> shamt);
}

uint_xlen_t bmrev(RA, RB, RC) {
    return bmrevi(RA, RB, GPR(RC) & 0b111111);
}
```

0.5	6.10	11.15	16.20	21.26	27..30	31	name	Form
NN	RT	RA	RB	sh	1111	Rc	bmrevi	MDS-Form

0.5	6.10	11.15	16.20	21.25	26..30	31	name	Form
NN	RT	RA	RB	RC	11110	Rc	bmrev	VA2-Form

3.8 grevlut

generalised reverse combined with a pair of LUT2s and allowing a constant `0b0101...0101` when RA=0, and an option to invert (including when RA=0, giving a constant 0b1010...1010 as the initial value) provides a wide range of instructions and a means to set hundreds of regular 64 bit patterns with one single 32 bit instruction.

the two LUT2s are applied left-half (when not swapping) and right-half (when swapping) so as to allow a wider range of options.

- A value of `0b11001010` for the immediate provides the functionality of a standard "grev".
- `0b11101110` provides gorc

grevlut should be arranged so as to produce the constants needed to put into bext (bitextract) so as in turn to be able to emulate x86 pmovmask instructions https://www.felixcloutier.com/x86/pmovmskb. This only requires 2 instructions (grevlut, bext).

Note that if the mask is required to be placed directly into CR Fields (for use as CR Predicate masks rather than a integer mask) then sv.cmpi or sv.ori may be used instead, bearing in mind that sv.ori is a 64-bit instruction, and `VL` must have been set to the required length:

```
sv.ori./elwid=8 r10.v, r10.v, 0
```

The following settings provide the required mask constants:

RA=0	RB	imm	iv	result
0x555..	0b10	0b01101100	0	0x111111...
0x555..	0b110	0b01101100	0	0x010101...
0x555..	0b1110	0b01101100	0	0x00010001...
0x555..	0b10	0b11000110	1	0x88888...
0x555..	0b110	0b11000110	1	0x808080...
0x555..	0b1110	0b11000110	1	0x80008000...

Better diagram showing the correct ordering of shamt (RB). A LUT2 is applied to all locations marked in red using the first 4 bits of the immediate, and a separate LUT2 applied to all locations in green using the upper 4 bits of the immediate.

demo code [[openpower/sv/grevlut.py]]

```
def lut2(imm, a, b):
    idx = b << 1 | a
    return (imm>>idx) & 1

def dorow(imm8, step_i, chunk_size):
    step_o = 0
    for j in range(64):
        if (j&chunk_size) == 0:
```

```
            imm = (imm8 & 0b1111)
        else:
            imm = (imm8>>4)
        a = (step_i>>j)&1
        b = (step_i>>(j ^ chunk_size))&1
        res = lut2(imm, a, b)
        #print(j, bin(imm), a, b, res)
        step_o |= (res<<j)
    #print ("  ", chunk_size, bin(step_o))
    return step_o

def grevlut64(RA, RB, imm, iv):
    x = 0
    if RA is None: # RA=0
        x = 0x5555555555555555
    else:
        x = RA
    if (iv): x = ~x;
    shamt = RB & 63;
    for i in range(6):
        step = 1<<i
        if (shamt & step):
            x = dorow(imm, x, step)
    return x & ((1<<64)-1)
```

A variant may specify different LUT-pairs per row, using one byte of RB for each. If it is desired that a particular row-crossover shall not be applied it is a simple matter to set the appropriate LUT-pair in RB to effect an identity transform for that row (`0b11001010`).

```
uint64_t grevlutr(uint64_t RA, uint64_t RB, bool iv, bool is32b)
{
    uint64_t x = 0x5555_5555_5555_5555;
    if (RA != 0) x = GPR(RA);
    if (iv) x = ~x;
    for i in 0 to (6-is32b)
        step = 1<<i
        imm = (RB>>(i*8))&0xff
        x = dorow(imm, x, step, is32b)
    return x;
}
```

0.5	6.10	11.15	16.20	21..28	29.30	31	name	Form
NN	RT	RA	s0-4	im0-7	1 iv	s5	grevlogi	
NN	RT	RA	RB	im0-7	01	0	grevlog	

An equivalent to `grevlogw` may be synthesised by setting the appropriate bits in RB to set the top half of RT to zero. Thus an explicit grevlogw instruction is not necessary.

3.9 xperm

based on RV bitmanip.

RA contains a vector of indices to select parts of RB to be copied to RT. The immediate-variant allows up to an 8 bit pattern (repeated) to be targetted at different parts of RT.

xperm shares some similarity with one of the uses of bmator in that xperm indices are binary addressing where bitmator may be considered to be unary addressing.

```
uint_xlen_t xpermi(uint8_t imm8, uint_xlen_t RB, int sz_log2)
{
    uint_xlen_t r = 0;
    uint_xlen_t sz = 1LL << sz_log2;
    uint_xlen_t mask = (1LL << sz) - 1;
    uint_xlen_t RA = imm8 | imm8<<8 | ... | imm8<<56;
    for (int i = 0; i < XLEN; i += sz) {
        uint_xlen_t pos = ((RA >> i) & mask) << sz_log2;
        if (pos < XLEN)
            r |= ((RB >> pos) & mask) << i;
    }
    return r;
}
uint_xlen_t xperm(uint_xlen_t RA, uint_xlen_t RB, int sz_log2)
{
    uint_xlen_t r = 0;
    uint_xlen_t sz = 1LL << sz_log2;
    uint_xlen_t mask = (1LL << sz) - 1;
    for (int i = 0; i < XLEN; i += sz) {
        uint_xlen_t pos = ((RA >> i) & mask) << sz_log2;
        if (pos < XLEN)
            r |= ((RB >> pos) & mask) << i;
    }
    return r;
}
uint_xlen_t xperm_n (uint_xlen_t RA, uint_xlen_t RB)
{  return xperm(RA, RB, 2); }
uint_xlen_t xperm_b (uint_xlen_t RA, uint_xlen_t RB)
{  return xperm(RA, RB, 3); }
uint_xlen_t xperm_h (uint_xlen_t RA, uint_xlen_t RB)
{  return xperm(RA, RB, 4); }
uint_xlen_t xperm_w (uint_xlen_t RA, uint_xlen_t RB)
{  return xperm(RA, RB, 5); }
```

3.10 bitmatrix

bmatflip and bmatxor is found in the Cray XMT, and in x86 is known as GF2P8AFFINEQB. uses:

- https://gist.github.com/animetosho/d3ca95da2131b5813e16b5bb1b137ca0
- SM4, Reed Solomon, RAID6 https://stackoverflow.com/questions/59124720/what-are-the-avx-512-galois-fie
- Vector bit-reverse https://reviews.llvm.org/D91515?id=305411
- Affine Inverse https://github.com/HJLebbink/asm-dude/wiki/GF2P8AFFINEINVQB

0.5	6.10	11.15	16.20	21	22.23	24....30	31	name	Form
NN	RS	RA	im04	im5	1 1	im67 00 110	Rc	bmatxori	TODO

```
uint64_t bmatflip(uint64_t RA)
{
    uint64_t x = RA;
    x = shfl64(x, 31);
    x = shfl64(x, 31);
```

```
    x = shfl64(x, 31);
    return x;
}

uint64_t bmatxori(uint64_t RS, uint64_t RA, uint8_t imm) {
    // transpose of RA
    uint64_t RAt = bmatflip(RA);
    uint8_t u[8]; // rows of RS
    uint8_t v[8]; // cols of RA
    for (int i = 0; i < 8; i++) {
        u[i] = RS >> (i*8);
        v[i] = RAt >> (i*8);
    }
    uint64_t bit, x = 0;
    for (int i = 0; i < 64; i++) {
        bit = (imm >> (i%8)) & 1;
        bit ^= pcnt(u[i / 8] & v[i % 8]) & 1;
        x |= bit << i;
    }
    return x;
}

uint64_t bmatxor(uint64_t RA, uint64_t RB) {
    return bmatxori(RA, RB, 0xff)
}

uint64_t bmator(uint64_t RA, uint64_t RB) {
    // transpose of RB
    uint64_t RBt = bmatflip(RB);
    uint8_t u[8]; // rows of RA
    uint8_t v[8]; // cols of RB
    for (int i = 0; i < 8; i++) {
        u[i] = RA >> (i*8);
        v[i] = RBt >> (i*8);
    }
    uint64_t x = 0;
    for (int i = 0; i < 64; i++) {
        if ((u[i / 8] & v[i % 8]) != 0)
            x |= 1LL << i;
    }
    return x;
}

uint64_t bmatand(uint64_t RA, uint64_t RB) {
    // transpose of RB
    uint64_t RBt = bmatflip(RB);
    uint8_t u[8]; // rows of RA
    uint8_t v[8]; // cols of RB
    for (int i = 0; i < 8; i++) {
        u[i] = RA >> (i*8);
        v[i] = RBt >> (i*8);
    }
    uint64_t x = 0;
    for (int i = 0; i < 64; i++) {
        if ((u[i / 8] & v[i % 8]) == 0xff)
```

```
            x |= 1LL << i;
        }
        return x;
    }
```

3.11 Introduction to Carry-less and GF arithmetic

- obligatory xkcd https://xkcd.com/2595/

There are three completely separate types of Galois-Field-based arithmetic that we implement which are not well explained even in introductory literature. A slightly oversimplified explanation is followed by more accurate descriptions:

- `GF(2)` carry-less binary arithmetic. this is not actually a Galois Field, but is accidentally referred to as GF(2) - see below as to why.
- `GF(p)` modulo arithmetic with a Prime number, these are "proper" Galois Fields
- `GF(2^N)` carry-less binary arithmetic with two limits: modulo a power-of-2 (2^N) and a second "reducing" polynomial (similar to a prime number), these are said to be GF(2^N) arithmetic.

further detailed and more precise explanations are provided below

- **Polynomials with coefficients in `GF(2)`** (aka. Carry-less arithmetic – the `cl*` instructions). This isn't actually a Galois Field, but its coefficients are. This is basically binary integer addition, subtraction, and multiplication like usual, except that carries aren't propagated at all, effectively turning both addition and subtraction into the bitwise xor operation. Division and remainder are defined to match how addition and multiplication works.
- **Galois Fields with a prime size** (aka. `GF(p)` or Prime Galois Fields – the `gfp*` instructions). This is basically just the integers mod `p`.
- **Galois Fields with a power-of-a-prime size** (aka. `GF(p^n)` or `GF(q)` where `q == p^n` for prime `p` and integer `n > 0`). We only implement these for `p == 2`, called Binary Galois Fields (`GF(2^n)` – the `gfb*` instructions). For any prime `p`, `GF(p^n)` is implemented as polynomials with coefficients in `GF(p)` and degree `< n`, where the polynomials are the remainders of dividing by a specificly chosen polynomial in `GF(p)` called the Reducing Polynomial (we will denote that by `red_poly`). The Reducing Polynomial must be an irreducable polynomial (like primes, but for polynomials), as well as have degree `n`. All `GF(p^n)` for the same `p` and `n` are isomorphic to each other – the choice of `red_poly` doesn't affect `GF(p^n)`'s mathematical shape, all that changes is the specific polynomials used to implement `GF(p^n)`.

Many implementations and much of the literature do not make a clear distinction between these three categories, which makes it confusing to understand what their purpose and value is.

- carry-less multiply is extremely common and is used for the ubiquitous CRC32 algorithm. [TODO add many others, helps justify to ISA WG]
- GF(2^N) forms the basis of Rijndael (the current AES standard) and has significant uses throughout cryptography
- GF(p) is the basis again of a significant quantity of algorithms (TODO, list them, jacob knows what they are), even though the modulo is limited to be below 64-bit (size of a scalar int)

3.12 Instructions for Carry-less Operations

aka. Polynomials with coefficients in `GF(2)`

Carry-less addition/subtraction is simply XOR, so a `cladd` instruction is not provided since the `xor[i]` instruction can be used instead.

These are operations on polynomials with coefficients in `GF(2)`, with the polynomial's coefficients packed into integers with the following algorithm:

```
    """Polynomials with GF(2) coefficients."""


    def pack_poly(poly):
        """`poly` is a list where `poly[i]` is the coefficient for `x ** i`"""
        retval = 0
        for i, v in enumerate(poly):
            retval |= v << i
        return retval


    def unpack_poly(v):
        """returns a list `poly`, where `poly[i]` is the coefficient for `x ** i`.
        """
        poly = []
        while v != 0:
            poly.append(v & 1)
            v >>= 1
        return poly
```

3.12.1 Carry-less Multiply Instructions

based on RV bitmanip see https://en.wikipedia.org/wiki/CLMUL_instruction_set and https://www.felixcloutier.com/x86/pclmulqdq and https://en.m.wikipedia.org/wiki/Carry-less_product

They are worth adding as their own non-overwrite operations (in the same pipeline).

3.12.1.1 `clmul` Carry-less Multiply

```
    def clmul(a, b):
        x = 0
        i = 0
        while b >> i != 0:
            if (b >> i) & 1:
                x ^= a << i
            i += 1
        return x
```

3.12.1.2 `clmulh` Carry-less Multiply High

```
    from nmigen_gf.reference.clmul import clmul


    def clmulh(a, b, XLEN):
        return clmul(a, b) >> XLEN
```

3.12.1.3 `clmulr` Carry-less Multiply (Reversed)

Useful for CRCs. Equivalent to bit-reversing the result of `clmul` on bit-reversed inputs.

```
    from nmigen_gf.reference.clmul import clmul
```

```
def clmulh(a, b, XLEN):
    return clmul(a, b) >> (XLEN - 1)
```

3.12.2 clmadd Carry-less Multiply-Add

```
clmadd RT, RA, RB, RC
```

```
(RT) = clmul((RA), (RB)) ^ (RC)
```

3.12.3 cltmadd Twin Carry-less Multiply-Add (for FFTs)

Used in combination with SV FFT REMAP to perform a full Discrete Fourier Transform of Polynomials over GF(2) in-place. Possible by having 3-in 2-out, to avoid the need for a temp register. RS is written to as well as RT.

Note: Polynomials over GF(2) are a Ring rather than a Field, so, because the definition of the Inverse Discrete Fourier Transform involves calculating a multiplicative inverse, which may not exist in every Ring, therefore the Inverse Discrete Fourier Transform may not exist. (AFAICT the number of inputs to the IDFT must be odd for the IDFT to be defined for Polynomials over GF(2). TODO: check with someone who knows for sure if that's correct.)

```
cltmadd RT, RA, RB, RC
```

TODO: add link to explanation for where RS comes from.

```
a = (RA)
c = (RC)
# read all inputs before writing to any outputs in case
# an input overlaps with an output register.
(RT) = clmul(a, (RB)) ^ c
(RS) = a ^ c
```

3.12.4 cldivrem Carry-less Division and Remainder

cldivrem isn't an actual instruction, but is just used in the pseudo-code for other instructions.

```
from nmigen_gf.reference.log2 import floor_log2


def cldivrem(n, d, width):
    """ Carry-less Division and Remainder.
        `n` and `d` are integers, `width` is the number of bits needed to hold
        each input/output.
        Returns a tuple `q, r` of the quotient and remainder.
    """
    assert d != 0, "TODO: decide what happens on division by zero"
    assert 0 <= n < 1 << width, f"bad n (doesn't fit in {width}-bit uint)"
    assert 0 <= d < 1 << width, f"bad d (doesn't fit in {width}-bit uint)"
    r = n
    q = 0
    d <<= width
    for _ in range(width):
        d >>= 1
        q <<= 1
        if degree(d) == degree(r):
            r ^= d
```

```
            q |= 1
    return q, r


def degree(v):
    """the degree of the GF(2) polynomial `v`. `v` is a non-negative integer.
    """
    if v == 0:
        return -1
    return floor_log2(v)
```

3.12.5 `cldiv` Carry-less Division

```
cldiv RT, RA, RB
```

```
n = (RA)
d = (RB)
q, r = cldivrem(n, d, width=XLEN)
(RT) = q
```

3.12.6 `clrem` Carry-less Remainder

```
clrem RT, RA, RB
```

```
n = (RA)
d = (RB)
q, r = cldivrem(n, d, width=XLEN)
(RT) = r
```

3.13 Instructions for Binary Galois Fields `GF(2^m)`

see:

- https://courses.csail.mit.edu/6.857/2016/files/ffield.py
- https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture7.pdf
- https://foss.heptapod.net/math/libgf2/-/blob/branch/default/src/libgf2/gf2.py

Binary Galois Field addition/subtraction is simply XOR, so a `gfbadd` instruction is not provided since the `xor[i]` instruction can be used instead.

3.13.1 `GFBREDPOLY` SPR – Reducing Polynomial

In order to save registers and to make operations orthogonal with standard arithmetic, the reducing polynomial is stored in a dedicated SPR `GFBREDPOLY`. This also allows hardware to pre-compute useful parameters (such as the degree, or look-up tables) based on the reducing polynomial, and store them alongside the SPR in hidden registers, only recomputing them whenever the SPR is written to, rather than having to recompute those values for every instruction.

Because Galois Fields require the reducing polynomial to be an irreducible polynomial, that guarantees that any polynomial of `degree > 1` must have the LSB set, since otherwise it would be divisible by the polynomial `x`, making it reducible, making whatever we're working on no longer a Field. Therefore, we can reuse the LSB to indicate `degree == XLEN`.

```
from nmigen_gf.reference.state import ST


def decode_reducing_polynomial():
    """returns the decoded reducing polynomial as an integer.
        Note: the returned integer is `XLEN + 1` bits wide.
    """
    v = ST.GFBREDPOLY & ((1 << ST.XLEN) - 1)  # mask to XLEN bits
    if v == 0 or v == 2:  # GF(2)
        return 0b10  # degree = 1, poly = x
    if (v & 1) == 0:
        # all reducing polynomials of degree > 1 must have the LSB set,
        # because they must be irreducible polynomials (meaning they
        # can't be factored), if the LSB was clear, then they would
        # have `x` as a factor. Therefore, we can reuse the LSB clear
        # to instead mean the polynomial has degree XLEN.
        v |= 1 << ST.XLEN
        v |= 1  # LSB must be set
    return v
```

3.13.2 gfbredpoly – Set the Reducing Polynomial SPR GFBREDPOLY

unless this is an immediate op, mtspr is completely sufficient.

```
from nmigen_gf.reference.state import ST


def gfbredpoly(immed):
    # TODO: figure out how `immed` should be encoded
    ST.GFBREDPOLY = immed
```

3.13.3 gfbmul – Binary Galois Field GF(2^m) Multiplication

```
gfbmul RT, RA, RB
```

```
from nmigen_gf.reference.state import ST
from nmigen_gf.reference.decode_reducing_polynomial import decode_reducing_polynomial
from nmigen_gf.reference.clmul import clmul
from nmigen_gf.reference.cldivrem import cldivrem


def gfbmul(a, b):
    product = clmul(a, b)
    red_poly = decode_reducing_polynomial()
    q, r = cldivrem(product, red_poly, width=ST.XLEN + 1)
    return r
```

3.13.4 gfbmadd – Binary Galois Field GF(2^m) Multiply-Add

```
gfbmadd RT, RA, RB, RC
```

```
from nmigen_gf.reference.state import ST
from nmigen_gf.reference.decode_reducing_polynomial import decode_reducing_polynomial
from nmigen_gf.reference.clmul import clmul
```

```
from nmigen_gf.reference.cldivrem import cldivrem


def gfbmadd(a, b, c):
    v = clmul(a, b) ^ c
    red_poly = decode_reducing_polynomial()
    q, r = cldivrem(v, red_poly, width=ST.XLEN + 1)
    return r
```

3.13.5 gfbtmadd – Binary Galois Field GF(2^m) Twin Multiply-Add (for FFT)

Used in combination with SV FFT REMAP to perform a full GF(2^m) Discrete Fourier Transform in-place. Possible by having 3-in 2-out, to avoid the need for a temp register. RS is written to as well as RT.

```
gfbtmadd RT, RA, RB, RC
```

TODO: add link to explanation for where RS comes from.

```
a = (RA)
c = (RC)
# read all inputs before writing to any outputs in case
# an input overlaps with an output register.
(RT) = gfbmadd(a, (RB), c)
# use gfbmadd again since it reduces the result
(RS) = gfbmadd(a, 1, c) # "a * 1 + c"
```

3.13.6 gfbinv – Binary Galois Field GF(2^m) Inverse

```
gfbinv RT, RA
```

```
from nmigen_gf.reference.decode_reducing_polynomial import decode_reducing_polynomial
from nmigen_gf.reference.cldivrem import degree


def gfbinv(a):
    """compute the GF(2^m) inverse of `a`."""
    # Derived from Algorithm 3, from [7] in:
    # https://ftp.libre-soc.org/ARITH18_Kobayashi.pdf

    s = decode_reducing_polynomial()
    m = degree(s)
    assert a >> m == 0, "`a` is out-of-range"
    r = a
    v = 0
    u = 1
    delta = 0

    for _ in range(2 * m):
        # could use count-leading-zeros here to skip ahead
        if r >> m == 0:  # if the MSB of `r` is zero
            r <<= 1
            u <<= 1
            delta += 1
        else:
            if s >> m != 0:  # if the MSB of `s` isn't zero
```

```
                s ^= r
                v ^= u
            s <<= 1
            if delta == 0:
                r, s = s, r  # swap r and s
                u, v = v << 1, u  # shift v and swap
                delta = 1
            else:
                u >>= 1
                delta -= 1
    if a == 0:
        # we specifically choose 0 as the result of inverting 0, rather than an
        # error or undefined, since that's what Rijndael needs.
        return 0
    return u
```

3.14 Instructions for Prime Galois Fields GF(p)

3.14.1 GFPRIME SPR – Prime Modulus For gfp* Instructions

3.14.2 gfpadd Prime Galois Field GF(p) Addition

```
    gfpadd RT, RA, RB
```

```
from nmigen_gf.reference.state import ST


def gfpadd(a, b):
    return (a + b) % ST.GFPRIME
```

the addition happens on infinite-precision integers

3.14.3 gfpsub Prime Galois Field GF(p) Subtraction

```
    gfpsub RT, RA, RB
```

```
from nmigen_gf.reference.state import ST


def gfpsub(a, b):
    return (a - b) % ST.GFPRIME
```

the subtraction happens on infinite-precision integers

3.14.4 gfpmul Prime Galois Field GF(p) Multiplication

```
    gfpmul RT, RA, RB
```

```
from nmigen_gf.reference.state import ST


def gfpmul(a, b):
    return (a * b) % ST.GFPRIME
```

the multiplication happens on infinite-precision integers

3.14.5 gfpinv Prime Galois Field GF(p) Invert

```
gfpinv RT, RA
```

Some potential hardware implementations are found in: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.5233&rep=rep1&type=pdf

```
from nmigen_gf.reference.state import ST


def gfpinv(a):
    # based on Algorithm ExtEucdInv from:
    # https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.5233&rep=rep1&type=pdf
    # designed to be dead-easy (and efficient) to implement in hardware
    p = ST.GFPRIME
    assert p >= 2, "GFPRIME isn't a prime"
    assert a != 0, "TODO: decide what happens for division by zero"
    assert isinstance(a, int) and 0 < a < p, "a out of range"
    if p == 2:
        return 1  # the only value possible

    # initial values`
    u = p
    v = a
    r = 0
    s = 1

    # main loop
    while v > 0:
        # implementations could use count-zeros on
        # both u and r to save cycles
        if u & 1 == 0:
            if r & 1 != 0:
                r += p
            u >>= 1
            r >>= 1
            continue # loop again
        # implementations could use count-zeros on
        # both v and s to save cycles
        if v & 1 == 0:
            if s & 1 != 0:
                s += p
            v >>= 1
            s >>= 1
            continue # loop again
        # both LSB of u and v are 1
        x = u - v
        if x > 0:
            u = x
            r -= s
            if r < 0:
                r += p
        else:
            v = -x
            s -= r
            if s < 0:
```

```
                s += p

        # make sure result r within modulo range 0 <= r <= p
        if r > p:
            r -= p
        if r < 0:
            r += p
        return r
```

3.14.6 gfpmadd Prime Galois Field GF(p) Multiply-Add

```
    gfpmadd RT, RA, RB, RC
```

```
from nmigen_gf.reference.state import ST


def gfpmadd(a, b, c):
    return (a * b + c) % ST.GFPRIME
```

the multiplication and addition happens on infinite-precision integers

3.14.7 gfpmsub Prime Galois Field GF(p) Multiply-Subtract

```
    gfpmsub RT, RA, RB, RC
```

```
from nmigen_gf.reference.state import ST


def gfpmsub(a, b, c):
    return (a * b - c) % ST.GFPRIME
```

the multiplication and subtraction happens on infinite-precision integers

3.14.8 gfpmsubr Prime Galois Field GF(p) Multiply-Subtract-Reversed

```
    gfpmsubr RT, RA, RB, RC
```

```
from nmigen_gf.reference.state import ST


def gfpmsubr(a, b, c):
    return (c - a * b) % ST.GFPRIME
```

the multiplication and subtraction happens on infinite-precision integers

3.14.9 gfpmaddsubr Prime Galois Field GF(p) Multiply-Add and Multiply-Sub-Reversed (for FFT)

Used in combination with SV FFT REMAP to perform a full Number-Theoretic-Transform in-place. Possible by having 3-in 2-out, to avoid the need for a temp register. RS is written to as well as RT.

```
    gfpmaddsubr RT, RA, RB, RC
```

TODO: add link to explanation for where RS comes from.

```
factor1 = (RA)
factor2 = (RB)
term = (RC)
# read all inputs before writing to any outputs in case
# an input overlaps with an output register.
(RT) = gfpmadd(factor1, factor2, term)
(RS) = gfpmsubr(factor1, factor2, term)
```

3.15 Already in POWER ISA or subsumed

Lists operations either included as part of other bitmanip operations, or are already in Power ISA.

3.15.1 cmix

based on RV bitmanip, covered by ternlog bitops

```
uint_xlen_t cmix(uint_xlen_t RA, uint_xlen_t RB, uint_xlen_t RC) {
    return (RA & RB) | (RC & ~RB);
}
```

3.15.2 count leading/trailing zeros with mask

in v3.1 p105

```
count = 0
do i = 0 to 63 if((RB)i=1) then do
if((RS)i=1) then break end end count ← count + 1
RA ← EXTZ64(count)
```

3.15.3 bit deposit

pdepd VRT,VRA,VRB, identical to RV bitmamip bdep, found already in v3.1 p106

```
do while(m < 64)
   if VSR[VRB+32].dword[i].bit[63-m]=1 then do
      result = VSR[VRA+32].dword[i].bit[63-k]
      VSR[VRT+32].dword[i].bit[63-m] = result
      k = k + 1
   m = m + 1


uint_xlen_t bdep(uint_xlen_t RA, uint_xlen_t RB)
{
    uint_xlen_t r = 0;
    for (int i = 0, j = 0; i < XLEN; i++)
        if ((RB >> i) & 1) {
            if ((RA >> j) & 1)
                r |= uint_xlen_t(1) << i;
            j++;
        }
    return r;
}
```

3.15.4 bit extract

other way round: identical to RV bext: pextd, found in v3.1 p196

```
uint_xlen_t bext(uint_xlen_t RA, uint_xlen_t RB)
{
    uint_xlen_t r = 0;
    for (int i = 0, j = 0; i < XLEN; i++)
        if ((RB >> i) & 1) {
            if ((RA >> i) & 1)
                r |= uint_xlen_t(1) << j;
            j++;
        }
    return r;
}
```

3.15.5 centrifuge

found in v3.1 p106 so not to be added here

```
ptr0 = 0
ptr1 = 0
do i = 0 to 63
    if((RB)i=0) then do
       resultptr0 = (RS)i
    end
    ptr0 = ptr0 + 1
    if((RB)63-i==1) then do
        result63-ptr1 = (RS)63-i
    end
    ptr1 = ptr1 + 1
RA = result
```

3.15.6 bit to byte permute

similar to matrix permute in RV bitmanip, which has XOR and OR variants, these perform a transpose (bmatflip). TODO this looks VSX is there a scalar variant in v3.0/1 already

```
do j = 0 to 7
  do k = 0 to 7
     b = VSR[VRB+32].dword[i].byte[k].bit[j]
     VSR[VRT+32].dword[i].byte[j].bit[k] = b
```

3.15.7 grev

superceded by grevlut

based on RV bitmanip, this is also known as a butterfly network. however where a butterfly network allows setting of every crossbar setting in every row and every column, generalised-reverse (grev) only allows a per-row decision: every entry in the same row must either switch or not-switch.

```
uint64_t grev64(uint64_t RA, uint64_t RB)
{
    uint64_t x = RA;
    int shamt = RB & 63;
```

```
    if (shamt & 1) x = ((x &  0x5555555555555555LL) <<  1) |
                       ((x & 0xAAAAAAAAAAAAAAAALL) >>  1);
    if (shamt & 2) x = ((x &  0x3333333333333333LL) <<  2) |
                       ((x & 0xCCCCCCCCCCCCCCCCLL) >>  2);
    if (shamt & 4) x = ((x &  0x0F0F0F0F0F0F0F0FLL) <<  4) |
                       ((x & 0xF0F0F0F0F0F0F0F0LL) >>  4);
    if (shamt & 8) x = ((x &  0x00FF00FF00FF00FFLL) <<  8) |
                       ((x & 0xFF00FF00FF00FF00LL) >>  8);
    if (shamt & 16) x = ((x & 0x0000FFFF0000FFFFLL) << 16) |
                       ((x & 0xFFFF0000FFFF0000LL) >> 16);
    if (shamt & 32) x = ((x & 0x00000000FFFFFFFFLL) << 32) |
                       ((x & 0xFFFFFFFF00000000LL) >> 32);
    return x;
}
```

3.15.8 gorc

based on RV bitmanip, gorc is superceded by grevlut

```
uint32_t gorc32(uint32_t RA, uint32_t RB)
{
    uint32_t x = RA;
    int shamt = RB & 31;
    if (shamt & 1) x |= ((x & 0x55555555) << 1)   |  ((x &  0xAAAAAAAA) >> 1);
    if (shamt & 2) x |= ((x & 0x33333333) << 2)   |  ((x &  0xCCCCCCCC) >> 2);
    if (shamt & 4) x |= ((x & 0x0F0F0F0F) << 4)   |  ((x &  0xF0F0F0F0) >> 4);
    if (shamt & 8) x |= ((x & 0x00FF00FF) << 8)   |  ((x &  0xFF00FF00) >> 8);
    if (shamt & 16) x |= ((x & 0x0000FFFF) << 16) |  ((x &  0xFFFF0000) >> 16);
    return x;
}
uint64_t gorc64(uint64_t RA, uint64_t RB)
{
    uint64_t x = RA;
    int shamt = RB & 63;
    if (shamt & 1) x |= ((x & 0x5555555555555555LL)   <<   1) |
                        ((x & 0xAAAAAAAAAAAAAAAALL)  >>  1);
    if (shamt & 2) x |= ((x & 0x3333333333333333LL)   <<   2) |
                        ((x & 0xCCCCCCCCCCCCCCCCLL)  >>  2);
    if (shamt & 4) x |= ((x & 0x0F0F0F0F0F0F0F0FLL)   <<   4) |
                        ((x & 0xF0F0F0F0F0F0F0F0LL)  >>  4);
    if (shamt & 8) x |= ((x & 0x00FF00FF00FF00FFLL)   <<   8) |
                        ((x & 0xFF00FF00FF00FF00LL)  >>  8);
    if (shamt & 16) x |= ((x & 0x0000FFFF0000FFFFLL)  << 16) |
                        ((x & 0xFFFF0000FFFF0000LL)  >> 16);
    if (shamt & 32) x |= ((x & 0x00000000FFFFFFFFLL)  << 32) |
                        ((x & 0xFFFFFFFF00000000LL)  >> 32);
    return x;
}
```

3.16 Appendix

see [[bitmanip/appendix]]

Chapter 4

FP/Int Conversion ops

[[!tag standards]]

Note on considered alternative naming schemes: we decided to switch to using the reduced mnemonic naming scheme (over some people's objections) since it would be 5 instructions instead of dozens, though we did consider trying to match PowerISA's existing naming scheme for the instructions rather than only for the instruction aliases. https://bugs.libre-soc.org/show_bug.cgi?id=1015#c7

4.1 FPR-to-GPR and GPR-to-FPR

TODO special constants instruction (e, tau/N, ln 2, sqrt 2, etc.) – exclude any constants available through fmvis

Draft Status under development, for submission as an RFC

Links:

- https://bugs.libre-soc.org/show_bug.cgi?id=650
- https://bugs.libre-soc.org/show_bug.cgi?id=230#c71
- https://bugs.libre-soc.org/show_bug.cgi?id=230#c74
- https://bugs.libre-soc.org/show_bug.cgi?id=230#c76
- https://bugs.libre-soc.org/show_bug.cgi?id=887 fmvis
- https://bugs.libre-soc.org/show_bug.cgi?id=1015 int-fp RFC
- [[int_fp_mv/appendix]]
- [[sv/rfc/ls002]] - `fmvis` and `fishmv` External RFC Formal Submission
- [[sv/rfc/ls006]] - int-fp-mv External RFC Formal Submission

Trademarks:

- Rust is a Trademark of the Rust Foundation
- Java and JavaScript are Trademarks of Oracle
- LLVM is a Trademark of the LLVM Foundation
- SPIR-V is a Trademark of the Khronos Group
- OpenCL is a Trademark of Apple, Inc.

Referring to these Trademarks within this document is by necessity, in order to put the semantics of each language into context, and is considered "fair use" under Trademark Law.

Introduction:

High-performance CPU/GPU software needs to often convert between integers and floating-point, therefore fast conversion/data-movement instructions are needed. Also given that initialisation of floats tends to take up considerable space (even to just load 0.0) the inclusion of two compact format float immediate instructions is

up for consideration using 16-bit immediates. BF16 is one of the formats: a second instruction allows a full accuracy FP32 to be constructed.

Libre-SOC will be compliant with the **Scalar Floating-Point Subset** (SFFS) i.e. is not implementing VMX/VSX, and with its focus on modern 3D GPU hybrid workloads represents an important new potential use-case for OpenPOWER.

Prior to the formation of the Compliancy Levels first introduced in v3.0C and v3.1 the progressive historic development of the Scalar parts of the Power ISA assumed that VSX would always be there to complement it. However With VMX/VSX **not available** in the newly-introduced SFFS Compliancy Level, the existing non-VSX conversion/data-movement instructions require a Vector of load/store instructions (slow and expensive) to transfer data between the FPRs and the GPRs. For a modern 3D GPU this kills any possibility of a competitive edge. Also, because SimpleV needs efficient scalar instructions in order to generate efficient vector instructions, adding new instructions for data-transfer/conversion between FPRs and GPRs multiplies the savings.

In addition, the vast majority of GPR <-> FPR data-transfers are as part of a FP <-> Integer conversion sequence, therefore reducing the number of instructions required is a priority.

Therefore, we are proposing adding:

- FPR load-immediate instructions, one equivalent to `BF16`, the other increasing accuracy to `FP32`
- FPR <-> GPR data-transfer instructions that just copy bits without conversion
- FPR <-> GPR combined data-transfer/conversion instructions that do Integer <-> FP conversions

If adding new Integer <-> FP conversion instructions, the opportunity may be taken to modernise the instructions and make them well-suited for common/important conversion sequences:

- Int -> Float
  - **standard IEEE754** - used by most languages and CPUs
- Float -> Int
  - **standard OpenPOWER** - saturation with NaN converted to minimum valid integer
  - **Java/Saturating** - saturation with NaN converted to 0
  - **JavaScript** - modulo wrapping with Inf/NaN converted to 0

The assembly listings in the [[int_fp_mv/appendix]] show how costly some of these language-specific conversions are: JavaScript, the worst case, is 32 scalar instructions including seven branch instructions.

4.2 Proposed New Scalar Instructions

All of the following instructions use the standard OpenPower conversion to/from 64-bit float format when reading/writing a 32-bit float from/to a FPR. All integers however are sourced/stored in the *GPR*.

Integer operands and results being in the GPR is the key differentiator between the proposed instructions (the entire rationale) compared to existing Scalar Power ISA. In all existing Power ISA Scalar conversion instructions, all operands are FPRs, even if the format of the source or destination data is actually a scalar integer.

(The existing Scalar instructions being FP-FP only is based on an assumption that VSX will be implemented, and VSX is not part of the SFFS Compliancy Level. An earlier version of the Power ISA used to have similar FPR<->GPR instructions to these: they were deprecated due to this incorrect assumption that VSX would always be present).

Note that source and destination widths can be overridden by SimpleV SVP64, and that SVP64 also has Saturation Modes *in addition* to those independently described here. SVP64 Overrides and Saturation work on *both* Fixed *and* Floating Point operands and results. The interactions with SVP64 are explained in the [[int_fp_mv/appendix]]

4.3 Float load immediate

These are like a variant of `fmvfg` and `oris`, combined. Power ISA currently requires a large number of instructions to get Floating Point constants into registers. `fmvis` on its own is equivalent to BF16 to FP32/64 conversion, but if followed up by `fishmv` an additional 16 bits of accuracy in the mantissa may be achieved.

These instructions **always** save resources compared to FP-load for exactly the same reason that `li` saves resources: an L1-Data-Cache and memory read is avoided.

IBM may consider it worthwhile to extend these two instructions to v3.1 Prefixed (`pfmvis` and `pfishmv`: 8RR, imm0 extended). If so it is recommended that `pfmvis` load a full FP32 immediate and `pfishmv` supplies the three high missing exponent bits (numbered 8 to 10) and the lower additional 29 mantissa bits (23 to 51) needed to construct a full FP64 immediate. Strictly speaking the sequence `fmvis fishmv pfishmv` achieves the same effect in the same number of bytes as `pfmvis pfishmv`, making `pfmvis` redundant.

Just as Floating-point Load does not set FP Flags neither does fmvis or fishmv. As fishmv is specifically intended to work in conjunction with fmvis to provide additional accuracy, all bits other than those which would have been set by a prior fmvis instruction are deliberately ignored. (If these instructions involved reading from registers rather than immediates it would be a different story).

4.3.1 Load BF16 Immediate

`fmvis FRS, D`

Reinterprets `D << 16` as a 32-bit float, which is then converted to a 64-bit float and written to `FRS`. This is equivalent to reinterpreting `D` as a `BF16` and converting to 64-bit float. There is no need for an Rc=1 variant because this is an immediate loading instruction.

Example:

```
# clearing a FPR
fmvis f4, 0 # writes +0.0 to f4
# loading handy constants
fmvis f4, 0x8000 # writes -0.0 to f4
fmvis f4, 0x3F80 # writes +1.0 to f4
fmvis f4, 0xBF80 # writes -1.0 to f4
fmvis f4, 0xBFC0 # writes -1.5 to f4
fmvis f4, 0x7FC0 # writes +qNaN to f4
fmvis f4, 0x7F80 # writes +Infinity to f4
fmvis f4, 0xFF80 # writes -Infinity to f4
fmvis f4, 0x3FFF # writes +1.9921875 to f4

# clearing 128 FPRs with 2 SVP64 instructions
# by issuing 32 vec4 (subvector length 4) ops
setvli VL=MVL=32
sv.fmvis/vec4 f0, 0 # writes +0.0 to f0-f127
```

Important: If the float load immediate instruction(s) are left out, change all GPR to FPR conversion instructions to instead write `+0.0` if `RA` is register `0`, at least allowing clearing FPRs.

`fmvis` fits with DX-Form:

0-5	6-10	11-15	16-25	26-30	31	Form
Major	FRS	d1	d0	XO	d2	DX-Form

Pseudocode:

```
bf16 = d0 || d1 || d2 # create BF16 immediate
fp32 = bf16 || [0]*16 # convert BF16 to FP32
FRS = DOUBLE(fp32)    # convert FP32 to FP64
```

Special registers altered:

```
None
```

4.3.2 Float Immediate Second-Half MV

`fishmv FRS, D`

DX-Form:

0-5	6-10	11-15	16-25	26-30	31	Form
Major	FRS	d1	d0	XO	d2	DX-Form

Strategically similar to how `oris` is used to construct 32-bit Integers, an additional 16-bits of immediate is inserted into `FRS` to extend its accuracy to a full FP32 (stored as usual in FP64 Format within the FPR). If a prior `fmvis` instruction had been used to set the upper 16-bits of an FP32 value, `fishmv` contains the lower 16-bits.

The key difference between using `li` and `oris` to construct 32-bit GPR Immediates and `fishmv` is that the `fmvis` will have converted the `BF16` immediate to FP64 (Double) format. This is taken into consideration as can be seen in the pseudocode below.

Pseudocode:

```
fp32 <- SINGLE((FRS))            # convert to FP32
fp32[16:31] <- d0 || d1 || d2    # replace LSB half
FRS <- DOUBLE(fp32)              # convert back to FP64
```

Special registers altered:

```
None
```

This instruction performs a Read-Modify-Write. *FRS is read, the additional 16 bit immediate inserted, and the result also written to FRS*

Example:

```
# these two combined instructions write 0x3f808000
# into f4 as an FP32 to be converted to an FP64.
# actual contents in f4 after conversion: 0x3ff0_1000_0000_0000
# first the upper bits, happens to be +1.0
fmvis f4, 0x3F80 # writes +1.0 to f4
# now write the lower 16 bits of an FP32
fishmv f4, 0x8000 # writes +1.00390625 to f4
```

[[!inline pages="openpower/sv/int_fp_mv/moves_and_conversions" raw=yes]]

Chapter 5

FP Class ops

5.1 fclass

based on xvtstdcsp v3.0B p760 the instruction performs analysis of the FP number to determine if it is Infinity, NaN, Denormalised or Zero and if so which sign. When VSX is not implemented these instructions become necessary.

unlike xvtstdcsp the result is stored in a Condition Register Field specified by BF. this allows it to be used as a predicate mask. setb may be used to create the equivalent of xvtstdcsp if desired.

The CR Field bits are set in a reasonably logical fashion:

- BF.EQ is set if FRB is zero
- BF.LE is set if FRB is non-normalises
- BF.GE is set if FRB is infinite
- BF.SO is set if FRB is NaN

0.5	6.8	9..15	16.20	21...30	31	name	Form
PO	BF	DCMX	FRB	XO	dm2	fptstsp	X-Form

```
dcmx <- DCMX || dm2
src  <- (FRB)[32:63]
sign  <- src[0]
exponent  <- src[1:8]
fraction  <- src[9:31]
class.Infinity  <- (exponent = 0xFF) & (fraction = 0)
class.NaN  <- (exponent = 0xFF) & (fraction != 0)
class.Zero  <- (exponent = 0x00) & (fraction = 0)
class.Denormal  <- (exponent = 0x00) & (fraction != 0)
CR{BF} <- ((dcmx[0] & class.NaN & !sign)  |
           (dcmx[1] & class.NaN & sign))  ||
          ((dcmx[2] & class.Infinity & !sign) |
           (dcmx[3] & class.Infinity & sign)) ||
          ((dcmx[6] & class.Denormal & !sign) |
           (dcmx[7] & class.Denormal & sign)) ||
          ((dcmx[4] & class.Zero & !sign) |
           (dcmx[5] & class.Zero & sign))
```

64 bit variant fptstdp is as follows:

```
src  <- (FRB)
```

```
sign <- src[0]
exponent <- src[1:11]
fraction <- src[12:63]
      exponent & 7FF
```

In SV just as with [[sv/fcvt]] single precision is to be considered half-of-elwidth precision. Thus when elwidth=FP32 fptstsp will test half that precision, at FP16.

Chapter 6

Audio and Video Opcodes

[[!tag standards]]

6.1 Scalar OpenPOWER Audio and Video Opcodes

the fundamental principle of SV is a hardware for-loop. therefore the first (and in nearly 100% of cases only) place to put Vector operations is first and foremost in the *scalar* ISA. However only by analysing those scalar opcodes *in* a SV Vectorisation context does it become clear why they are needed and how they may be designed.

This page therefore has accompanying discussion at https://bugs.libre-soc.org/show_bug.cgi?id=230 for evolution of suitable opcodes.

Links

- https://bugs.libre-soc.org/show_bug.cgi?id=915 add overflow to maxmin.
- https://bugs.libre-soc.org/show_bug.cgi?id=863 add pseudocode etc.
- https://bugs.libre-soc.org/show_bug.cgi?id=234 hardware implementation
- https://bugs.libre-soc.org/show_bug.cgi?id=910 mins/maxs zero-option?
- https://bugs.libre-soc.org/show_bug.cgi?id=1057 move all int/fp min/max to ls013
- [[vpu]]
- {FP/Int Conversion ops}
- [[openpower/isa/av]] pseudocode
- [[av_opcodes/analysis]]
- TODO review HP 1994-6 PA-RISC MAX https://en.m.wikipedia.org/wiki/Multimedia_Acceleration_eXtensions
- https://en.m.wikipedia.org/wiki/Sum_of_absolute_differences
- List of MMX instructions https://cs.fit.edu/~mmahoney/cse3101/mmx.html

6.2 Summary

In-advance, the summary of base scalar operations that need to be added is:

instruction	pseudocode
average-add.	result = (src1 + src2 + 1) >> 1
abs-diff	result = abs (src1-src2)
abs-accumulate	result += abs (src1-src2)
(un)signed min	result = (src1 < src2) ? src1 : src2 {RFC ls013}
(un)signed max	result = (src1 > src2) ? src1 : src2 {RFC ls013}

instruction	pseudocode
bitwise sel	(a ? b : c) - use {Bitmanip ops} ternary
int/fp move	covered by REMAP and Pack/Unpack

Implemented at the [[openpower/isa/av]] pseudocode page.

All other capabilities (saturate in particular) are achieved with {SVP64 Chapter} modes and swizzle. Note that minmax and ternary are added in bitmanip.

6.3 Instructions

6.3.1 Average Add

X-Form

- avgadd RT,RA,RB (Rc=0)
- avgadd. RT,RA,RB (Rc=1)

Pseudo-code:

```
a <- [0] * (XLEN+1)
b <- [0] * (XLEN+1)
a[1:XLEN] <- (RA)
b[1:XLEN] <- (RB)
r <- (a + b + 1)
RT <- r[0:XLEN-1]
```

Special Registers Altered:

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

6.3.2 Absolute Signed Difference

X-Form

- absds RT,RA,RB (Rc=0)
- absds. RT,RA,RB (Rc=1)

Pseudo-code:

```
if (RA) < (RB) then RT <- ¬(RA) + (RB) + 1
else                RT <- ¬(RB) + (RA) + 1
```

Special Registers Altered:

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

6.3.3 Absolute Unsigned Difference

X-Form

- absdu RT,RA,RB (Rc=0)
- absdu. RT,RA,RB (Rc=1)

Pseudo-code:

```
if (RA) <u (RB) then RT <- ¬(RA) + (RB) + 1
else                RT <- ¬(RB) + (RA) + 1
```

Special Registers Altered:

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

6.3.4 Absolute Accumulate Unsigned Difference

X-Form

- absdacu RT,RA,RB (Rc=0)
- absdacu. RT,RA,RB (Rc=1)

Pseudo-code:

```
if (RA) <u (RB) then r <- ¬(RA) + (RB) + 1
else                 r <- ¬(RB) + (RA) + 1
RT <- (RT) + r
```

Special Registers Altered:

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

6.3.5 Absolute Accumulate Signed Difference

X-Form

- absdacs RT,RA,RB (Rc=0)
- absdacs. RT,RA,RB (Rc=1)

Pseudo-code:

```
if (RA) < (RB) then r <- ¬(RA) + (RB) + 1
else                r <- ¬(RB) + (RA) + 1
RT <- (RT) + r
```

Special Registers Altered:

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

Chapter 7

Big Integer

[[!tag standards]]

7.1 Big Integer Arithmetic

DRAFT STATUS 19apr2022, last edited 23may2022

- [[discussion]] page for notes
- https://bugs.libre-soc.org/show_bug.cgi?id=817 bugreport
- https://bugs.libre-soc.org/show_bug.cgi?id=937 128/64 shifts
- [[biginteger/analysis]]
- [[openpower/isa/svfixedarith]] pseudocode

BigNum arithmetic is extremely common especially in cryptography, where for example RSA relies on arithmetic of 2048 or 4096 bits in length. The primary operations are add, multiply and divide (and modulo) with specialisations of subtract and signed multiply.

A reminder that a particular focus of SVP64 is that it is built on top of Scalar operations, where those scalar operations are useful in their own right without SVP64. Thus the operations here are proposed first as Scalar Extensions to the Power ISA.

A secondary focus is that if Vectorised, implementors may choose to deploy macro-op fusion targetting back-end 256-bit or greater Dynamic SIMD ALUs for maximum performance and effectiveness.

7.2 Analysis

Covered in [[biginteger/analysis]] the summary is that standard `adde` is sufficient for SVP64 Vectorisation of big-integer addition (and `subfe` for subtraction) but that big-integer shift, multiply and divide require an extra 3-in 2-out instructions, similar to Intel's shld and shrd, mulx and divq, to be efficient. The same instruction (`maddedu`) is used in both big-divide and big-multiply because 'maddedu"s primary purpose is to perform a fused 64-bit scalar multiply with a large vector, where that result is Big-Added for Big-Multiply, but Big-Subtracted for Big-Divide.

Chaining the operations together gives Scalar-by-Vector operations, except for `sv.adde` and `sv.subfe` which are Vector-by-Vector Chainable (through the `CA` flag). Macro-op Fusion and back-end massively-wide SIMD ALUs may be deployed in a fashion that is hidden from the user, behind a consistent, stable ISA API. The same macro-op fusion may theoretically be deployed even on Scalar operations.

7.3 DRAFT dsld

0. 5	6..10	11..15	16..20	21.25	26..30	31
EXT04	RT	RA	RB	RC	XO	Rc

VA2-Form

- dsld RT,RA,RB,RC (Rc=0)
- dsld. RT,RA,RB,RC (Rc=1)

Pseudo-code:

```
n <- (RB)[58:63]
v <- ROTL64((RA), n)
mask <- MASK(0, 63-n)
RT <- (v[0:63] & mask) | ((RC) & ¬mask)
RS <- v[0:63] & ¬mask
overflow = 0
if RS != [0]*64:
    overflow = 1
```

Special Registers Altered:

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

7.4 DRAFT dsrd

0. 5	6..10	11..15	16..20	21.25	26..30	31
EXT04	RT	RA	RB	RC	XO	Rc

VA2-Form

- dsrd RT,RA,RB,RC (Rc=0)
- dsrd. RT,RA,RB,RC (Rc=1)

Pseudo-code:

```
n <- (RB)[58:63]
v <- ROTL64((RA), 64-n)
mask <- MASK(n, 63)
RT <- (v[0:63] & mask) | ((RC) & ¬mask)
RS <- v[0:63] & ¬mask
overflow = 0
if RS != [0]*64:
    overflow = 1
```

Special Registers Altered:

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

7.5 maddedu

DRAFT

`maddedu` is similar to v3.0 `madd`, and is VA-Form despite having 2 outputs: the second destination register is implicit.

0.....5	6..10	11..15	16..20	21..25	26..31
EXT04	RT	RA	RB	RC	XO

The pseudocode for `maddedu RT, RA, RB, RC` is:

```
prod[0:127] = (RA) * (RB)
sum[0:127] = EXTZ(RC) + prod
RT <- sum[64:127]
RS <- sum[0:63] # RS implicit register, see below
```

RC is zero-extended (not shifted, not sign-extended), the 128-bit product added to it; the lower half of that result stored in RT and the upper half in RS.

The differences here to `maddhdu` are that `maddhdu` stores the upper half in RT, where `maddedu` stores the upper half in RS.

The value stored in RT is exactly equivalent to `maddld` despite `maddld` performing sign-extension on RC, because RT is the full mathematical result modulo 2^64 and sign/zero extension from 64 to 128 bits produces identical results modulo 2^64. This is why there is no maddldu instruction.

Programmer's Note: As a Scalar Power ISA operation, like `lq` and `stq`, RS=RT+1. To achieve the same big-integer rolling-accumulation effect as SVP64: assuming the scalar to multiply is in r0, the vector to multiply by starts at r4 and the result vector in r20, instructions may be issued `maddedu r20,r4,r0,r20` `maddedu r21,r5,r0,r21` etc. where the first `maddedu` will have stored the upper half of the 128-bit multiply into r21, such that it may be picked up by the second `maddedu`. Repeat inline to construct a larger bigint scalar-vector multiply, as Scalar GPR register file space permits.

SVP64 overrides the Scalar behaviour of what defines RS. For SVP64 EXTRA register extension, the `RM-1P-3S-1D` format is used with the additional bit set for determining RS.

Field Name	Field bits	Description
Rdest_EXTRA2	`10:11`	extends RT (R*_EXTRA2 Encoding)
Rsrc1_EXTRA2	`12:13`	extends RA (R*_EXTRA2 Encoding)
Rsrc2_EXTRA2	`14:15`	extends RB (R*_EXTRA2 Encoding)
Rsrc3_EXTRA2	`16:17`	extends RC (R*_EXTRA2 Encoding)
EXTRA2_MODE	`18`	used by `maddedu` for determining RS

When `EXTRA2_MODE` is set to zero, the implicit RS register takes its Vector/Scalar setting from Rdest_EXTRA2, and takes the register number from RT, but all numbering is offset by MAXVL. *Note that element-width overrides influence this offset* (see SVP64 {SVP64 Appendix} for full details).

When `EXTRA2_MODE` is set to one, the implicit RS register is identical to RC extended with SVP64 using `Rsrc3_EXTRA2` in every respect, including whether RC is set Scalar or Vector.

7.6 divmod2du RT,RA,RB,RC

DRAFT

Divide/Modulu Quad-Double Unsigned is another VA-Form instruction that is near-identical to `divdeu` except that:

- the lower 64 bits of the dividend, instead of being zero, contain a register, RC.

- it performs a fused divide and modulo in a single instruction, storing the modulo in an implicit RS (similar to `maddedu`)

RB, the divisor, remains 64 bit. The instruction is therefore a 128/64 division, producing a (pair) of 64 bit result(s), in the same way that Intel divq works. Overflow conditions are detected in exactly the same fashion as `divdeu`, except that rather than have `UNDEFINED` behaviour, RT is set to all ones and RS set to all zeros on overflow.

Programmer's note: there are no Rc variants of any of these VA-Form instructions. `cmpi` *will need to be used to detect overflow conditions: the saving in instruction count is that both RT and RS will have already been set to useful values (all 1s and all zeros respectively) needed as part of implementing Knuth's Algorithm D*

For SVP64, given that this instruction is also 3-in 2-out 64-bit registers, the exact same EXTRA format and setting of RS is used as for `sv.maddedu`. For Scalar usage, just as for `maddedu`, `RS=RT+1` (similar to `lq` and `stq`).

Pseudo-code:

```
if ((RA) <u (RB)) & ((RB) != [0]*XLEN) then
    dividend[0:(XLEN*2)-1] <- (RA) || (RC)
    divisor[0:(XLEN*2)-1] <- [0]*XLEN || (RB)
    result <- dividend / divisor
    modulo <- dividend % divisor
    RT <- result[XLEN:(XLEN*2)-1]
    RS <- modulo[XLEN:(XLEN*2)-1]
else
    RT <- [1]*XLEN
    RS <- [0]*XLEN
```

7.7 [DRAFT] EXT04 Proposed Map

For the Opcode map (XO Field) see Power ISA v3.1, Book III, Appendix D, Table 13 (sheet 7 of 8), p1357. Proposed is the addition of:

- `maddedu` in `110010`
- `divmod2du` in `111010`
- `pcdec` in `111000`

v >	000	001	010	011	100	101	110	111
110	maddhd	maddhdu	maddedu	maddld	rsvd	rsvd	rsvd	rsvd
111	pcdec.	rsvd	divmod2du	vpermr	vaddequm	vaddecuq	vsubeuqm	vsubecuq

Chapter 8

Transcendentals

8.1 DRAFT Scalar Transcendentals

Summary:

This proposal extends Power ISA scalar floating point operations to add IEEE754 transcendental functions (pow, log etc) and trigonometric functions (sin, cos etc). These functions are also 98% shared with the Khronos Group OpenCL Extended Instruction Set.

Authors/Contributors:

- Luke Kenneth Casson Leighton
- Jacob Lifshay
- Dan Petroski
- Mitch Alsup
- Allen Baum
- Andrew Waterman
- Luis Vitorio Cargnini

[[!toc levels=2]]

See:

- http://bugs.libre-soc.org/show_bug.cgi?id=127
- https://bugs.libre-soc.org/show_bug.cgi?id=899 transcendentals in simulator
- https://bugs.libre-soc.org/show_bug.cgi?id=923 under review
- https://www.khronos.org/registry/spir-v/specs/unified1/OpenCL.ExtendedInstructionSet.100.html
- [[power_trans_ops]] for opcode listing.

Extension subsets:

TODO: rename extension subsets – we're not on RISC-V anymore.

- **Zftrans**: standard transcendentals (best suited to 3D)
- **ZftransExt**: extra functions (useful, not generally needed for 3D, can be synthesised using Ztrans)
- **Ztrigpi**: trig. xxx-pi sinpi cospi tanpi
- **Ztrignpi**: trig non-xxx-pi sin cos tan
- **Zarctrigpi**: arc-trig. a-xxx-pi: atan2pi asinpi acospi
- **Zarctrignpi**: arc-trig. non-a-xxx-pi: atan2, asin, acos
- **Zfhyp**: hyperbolic/inverse-hyperbolic. sinh, cosh, tanh, asinh, acosh, atanh (can be synthesised - see below)
- **ZftransAdv**: much more complex to implement in hardware

- **Zfrsqrt**: Reciprocal square-root.
- **Zfminmax**: Min/Max.

Minimum recommended requirements for 3D: Zftrans, Ztrignpi, Zarctrignpi, with Ztrigpi and Zarctrigpi as augmentations.

Minimum recommended requirements for Mobile-Embedded 3D: Ztrignpi, Zftrans, with Ztrigpi as an augmentation.

The Platform Requirements for 3D are driven by cost competitive factors and it is the Trademarked Vulkan Specification that provides clear direction for 3D GPU markets, but nothing else (IEEE754). Implementors must note that minimum Compliance with the Third Party Vulkan Specification (for power-area competitive reasons with other 3D GPU manufacturers) will not qualify for strict IEEE754 accuracy Compliance or vice-versa.

Implementors **must** make it clear which accuracy level is implemented and provide a switching mechanism and throw Illegal Instruction traps if fully compliant accuracy cannot be achieved. It is also the Implementor's responsibility to comply with all Third Party Certification Marks and Trademarks (Vulkan, OpenCL). Nothing in this specification in any way implies that any Third Party Certification Mark Compliance is granted, nullified, altered or overridden by this document.

8.2 TODO:

- Decision on accuracy, moved to [[zfpacc_proposal]] http://lists.libre-riscv.org/pipermail/libre-riscv-dev/2019-August/002355.html
- Errors **MUST** be repeatable.
- How about four Platform Specifications? 3DUNIX, UNIX, 3DEmbedded and Embedded? http://lists.libre-riscv.org/pipermail/libre-riscv-dev/2019-August/002361.html Accuracy requirements for dual (triple) purpose implementations must meet the higher standard.
- Reciprocal Square-root is in its own separate extension (Zfrsqrt) as it is desirable on its own by other implementors. This to be evaluated.

8.3 Requirements

This proposal is designed to meet a wide range of extremely diverse needs, allowing implementors from all of them to benefit from the tools and hardware cost reductions associated with common standards adoption in Power ISA (primarily IEEE754 and Vulkan).

The use-cases are:

- 3D GPUs
- Numerical Computation
- (Potentially) A.I. / Machine-learning (1)

(1) although approximations suffice in this field, making it more likely to use a custom extension. High-end ML would inherently definitely be excluded.

The power and die-area requirements vary from:

- Ultra-low-power (smartwatches where GPU power budgets are in milliwatts)
- Mobile-Embedded (good performance with high efficiency for battery life)
- Desktop Computing
- Server / HPC / Supercomputing

The software requirements are:

- Full public integration into GNU math libraries (libm)
- Full public integration into well-known Numerical Computation systems (numpy)

- Full public integration into upstream GNU and LLVM Compiler toolchains
- Full public integration into Khronos OpenCL SPIR-V compatible Compilers seeking public Certification and Endorsement from the Khronos Group under their Trademarked Certification Programme.

8.4 Proposed Opcodes vs Khronos OpenCL vs IEEE754-2019

This list shows the (direct) equivalence between proposed opcodes, their Khronos OpenCL equivalents, and their IEEE754-2019 equivalents. 98% of the opcodes in this proposal that are in the IEEE754-2019 standard are present in the Khronos Extended Instruction Set.

See https://www.khronos.org/registry/spir-v/specs/unified1/OpenCL.ExtendedInstructionSet.100.html and https://ieeexplore.ieee.org/document/8766229

- Special FP16 opcodes are *not* being proposed, except by indirect / inherent use of elwidth overrides that is already present in the SVP64 Specification.
- "Native" opcodes are *not* being proposed: implementors will be expected to use the (equivalent) proposed opcode covering the same function.
- "Fast" opcodes are *not* being proposed, because the Khronos Specification fast_length, fast_normalise and fast_distance OpenCL opcodes require vectors (or can be done as scalar operations using other Power ISA instructions).

The OpenCL FP32 opcodes are **direct** equivalents to the proposed opcodes. Deviation from conformance with the Khronos Specification - including the Khronos Specification accuracy requirements - is not an option, as it results in non-compliance, and the vendor may not use the Trademarked words "Vulkan" etc. in conjunction with their product.

IEEE754-2019 Table 9.1 lists "additional mathematical operations". Interestingly the only functions missing when compared to OpenCL are compound, exp2m1, exp10m1, log2p1, log10p1, pown (integer power) and powr.

opcode	OpenCL FP32	OpenCL FP16	OpenCL native	IEEE754	Power ISA
fsin	sin	half_sin	native_sin	sin	NONE
fcos	cos	half_cos	native_cos	cos	NONE
ftan	tan	half_tan	native_tan	tan	NONE
NONE (1)	sincos	NONE	NONE	NONE	NONE
fasin	asin	NONE	NONE	asin	NONE
facos	acos	NONE	NONE	acos	NONE
fatan	atan	NONE	NONE	atan	NONE
fsinpi	sinpi	NONE	NONE	sinPi	NONE
fcospi	cospi	NONE	NONE	cosPi	NONE
ftanpi	tanpi	NONE	NONE	tanPi	NONE
fasinpi	asinpi	NONE	NONE	asinPi	NONE
facospi	acospi	NONE	NONE	acosPi	NONE
fatanpi	atanpi	NONE	NONE	atanPi	NONE
fsinh	sinh	NONE	NONE	sinh	NONE
fcosh	cosh	NONE	NONE	cosh	NONE
ftanh	tanh	NONE	NONE	tanh	NONE
fasinh	asinh	NONE	NONE	asinh	NONE
facosh	acosh	NONE	NONE	acosh	NONE
fatanh	atanh	NONE	NONE	atanh	NONE
fatan2	atan2	NONE	NONE	atan2	NONE
fatan2pi	atan2pi	NONE	NONE	atan2pi	NONE
frsqrt	rsqrt	half_rsqrt	native_rsqrt	rSqrt	fsqrte, fsqrtes (4)
fcbrt	cbrt	NONE	NONE	NONE (2)	NONE
fexp2	exp2	half_exp2	native_exp2	exp2	NONE
flog2	log2	half_log2	native_log2	log2	NONE

opcode	OpenCL FP32	OpenCL FP16	OpenCL native	IEEE754	Power ISA
fexpm1	expm1	NONE	NONE	expm1	NONE
flog1p	log1p	NONE	NONE	logp1	NONE
fexp	exp	half_exp	native_exp	exp	NONE
flog	log	half_log	native_log	log	NONE
fexp10	exp10	half_exp10	native_exp10	exp10	NONE
flog10	log10	half_log10	native_log10	log10	NONE
fpow	pow	NONE	NONE	pow	NONE
fpown	pown	NONE	NONE	pown	NONE
fpowr	powr	half_powr	native_powr	powr	NONE
frootn	rootn	NONE	NONE	rootn	NONE
fhypot	hypot	NONE	NONE	hypot	NONE
frecip	NONE	half_recip	native_recip	NONE (3)	fre, fres (4)
NONE	NONE	NONE	NONE	compound	NONE
fexp2m1	NONE	NONE	NONE	exp2m1	NONE
fexp10m1	NONE	NONE	NONE	exp10m1	NONE
flog2p1	NONE	NONE	NONE	log2p1	NONE
flog10p1	NONE	NONE	NONE	log10p1	NONE
fminnum08	fmin	fmin	NONE	minNum	xsmindp (5)
fmaxnum08	fmax	fmax	NONE	maxNum	xsmaxdp (5)
fmin19	fmin	fmin	NONE	minimum	NONE
fmax19	fmax	fmax	NONE	maximum	NONE
fminnum19	fmin	fmin	NONE	minimumNumber	vminfp (6), xsminjdp (
fmaxnum19	fmax	fmax	NONE	maximumNumber	vmaxfp (6), xsmaxjdp
fminc	fmin	fmin	NONE	NONE	xsmincdp (5)
fmaxc	fmax	fmax	NONE	NONE	xsmaxcdp (5)
fminmagnum08	minmag	minmag	NONE	minNumMag	NONE
fmaxmagnum08	maxmag	maxmag	NONE	maxNumMag	NONE
fminmag19	minmag	minmag	NONE	minimumMagnitude	NONE
fmaxmag19	maxmag	maxmag	NONE	maximumMagnitude	NONE
fminmagnum19	minmag	minmag	NONE	minimumMagnitudeNumber	NONE
fmaxmagnum19	maxmag	maxmag	NONE	maximumMagnitudeNumber	NONE
fminmagc	minmag	minmag	NONE	NONE	NONE
fmaxmagc	maxmag	maxmag	NONE	NONE	NONE
fmod	fmod	fmod		NONE	NONE
fremainder	remainder	remainder		remainder	NONE

from Mitch Alsup:

- Brian's LLVM compiler converts fminc and fmaxc into fmin and fmax instructions These are all IEEE 754-2019 compliant These are native instructions not extensions All listed functions are available in both F32 and F64 formats. THere is some confusion (in my head) abouot fmin and fmax. I intend both instruction to perform 754-2019 semantics– but I don know if this is minimum/maximum or minimum-Number/maximumNumber. fmad and remainder are a 2-instruction sequence–don't know how to "edit it in"

Note (1) fsincos is macro-op fused (see below).

Note (2) synthesised in IEEE754-2019 as "rootn(x, 3)"

Note (3) synthesised in IEEE754-2019 using "1.0 / x"

Note (4) these are estimate opcodes that help accelerate software emulation

Note (5) f64-only (though can be used on f32 stored in f64 format), requires VSX.

Note (6) 4xf32-only, requires VMX.

8.4.1 List of 2-arg opcodes

opcode	Description	pseudocode	Extension
fatan2	atan2 arc tangent	FRT = atan2(FRB, FRA)	Zarctrignpi
fatan2pi	atan2 arc tangent / pi	FRT = atan2(FRB, FRA) / pi	Zarctrigpi
fpow	x power of y	FRT = pow(FRA, FRB)	ZftransAdv
fpown	x power of n (n int)	FRT = pow(FRA, RB)	ZftransAdv
fpowr	x power of y (x +ve)	FRT = exp(FRA log(FRB))	ZftransAdv
frootn	x power 1/n (n integer)	FRT = pow(FRA, 1/RB)	ZftransAdv
fhypot	hypotenuse	FRT = sqrt(FRA^2 + FRB^2)	ZftransAdv
fminnum08	IEEE 754-2008 minNum	FRT = minNum(FRA, FRB) (1)	Zfminmax
fmaxnum08	IEEE 754-2008 maxNum	FRT = maxNum(FRA, FRB) (1)	Zfminmax
fmin19	IEEE 754-2019 minimum	FRT = minimum(FRA, FRB)	Zfminmax
fmax19	IEEE 754-2019 maximum	FRT = maximum(FRA, FRB)	Zfminmax
fminnum19	IEEE 754-2019 minimumNumber	FRT = minimumNumber(FRA, FRB)	Zfminmax
fmaxnum19	IEEE 754-2019 maximumNumber	FRT = maximumNumber(FRA, FRB)	Zfminmax
fminc	C ternary-op minimum	FRT = FRA < FRB ? FRA : FRB	Zfminmax
fmaxc	C ternary-op maximum	FRT = FRA > FRB ? FRA : FRB	Zfminmax
fminmagnum08	IEEE 754-2008 minNumMag	FRT = minmaxmag(FRA, FRB, False, fminnum08) (2)	Zfminmax
fmaxmagnum08	IEEE 754-2008 maxNumMag	FRT = minmaxmag(FRA, FRB, True, fmaxnum08) (2)	Zfminmax
fminmag19	IEEE 754-2019 minimumMagnitude	FRT = minmaxmag(FRA, FRB, False, fmin19) (2)	Zfminmax
fmaxmag19	IEEE 754-2019 maximumMagnitude	FRT = minmaxmag(FRA, FRB, True, fmax19) (2)	Zfminmax
fminmagnum19	IEEE 754-2019 minimumMagnitudeNumber	FRT = minmaxmag(FRA, FRB, False, fminnum19) (2)	Zfminmax

opcode	Description	pseudocode	Extension
fmaxmagnum19	IEEE 754-2019 maximumMagnitudeNumber	FRT = minmaxmag(FRA, FRB, True, fmaxnum19) (2)	Zfminmax
fminmagc	C ternary-op minimum magnitude	FRT = minmaxmag(FRA, FRB, False, fminc) (2)	Zfminmax
fmaxmagc	C ternary-op maximum magnitude	FRT = minmaxmag(FRA, FRB, True, fmaxc) (2)	Zfminmax
fmod	modulus	FRT = fmod(FRA, FRB)	ZftransExt
fremainder	IEEE 754 remainder	FRT = remainder(FRA, FRB)	ZftransExt

Note (1): for the purposes of minNum/maxNum, -0.0 is defined to be less than +0.0. This is left unspecified in IEEE 754-2008.

Note (2): minmaxmag(x, y, cmp, fallback) is defined as:

```
def minmaxmag(x, y, is_max, fallback):
    a = abs(x) < abs(y)
    b = abs(x) > abs(y)
    if is_max:
        a, b = b, a  # swap
    if a:
        return x
    if b:
        return y
    # equal magnitudes, or NaN input(s)
    return fallback(x, y)
```

8.4.2 List of 1-arg transcendental opcodes

opcode	Description	pseudocode	Extension
frsqrt	Reciprocal Square-root	FRT = sqrt(FRA)	Zfrsqrt
fcbrt	Cube Root	FRT = pow(FRA, 1.0 / 3)	ZftransAdv
frecip	Reciprocal	FRT = 1.0 / FRA	Zftrans
fexp2m1	power-2 minus 1	FRT = pow(2, FRA) - 1.0	ZftransExt
flog2p1	log2 plus 1	FRT = log(2, 1 + FRA)	ZftransExt
fexp2	power-of-2	FRT = pow(2, FRA)	Zftrans
flog2	log2	FRT = log(2. FRA)	Zftrans
fexpm1	exponential minus 1	FRT = pow(e, FRA) - 1.0	ZftransExt
flog1p	log plus 1	FRT = log(e, 1 + FRA)	ZftransExt
fexp	exponential	FRT = pow(e, FRA)	ZftransExt
flog	natural log (base e)	FRT = log(e, FRA)	ZftransExt
fexp10m1	power-10 minus 1	FRT = pow(10, FRA) - 1.0	ZftransExt
flog10p1	log10 plus 1	FRT = log(10, 1 + FRA)	ZftransExt
fexp10	power-of-10	FRT = pow(10, FRA)	ZftransExt
flog10	log base 10	FRT = log(10, FRA)	ZftransExt

8.4.3 List of 1-arg trigonometric opcodes

opcode	Description	pseudocode	Extension
fsin	sin (radians)	FRT = sin(FRA)	Ztrignpi
fcos	cos (radians)	FRT = cos(FRA)	Ztrignpi
ftan	tan (radians)	FRT = tan(FRA)	Ztrignpi
fasin	arcsin (radians)	FRT = asin(FRA)	Zarctrignpi
facos	arccos (radians)	FRT = acos(FRA)	Zarctrignpi
fatan	arctan (radians)	FRT = atan(FRA)	Zarctrignpi
fsinpi	sin times pi	FRT = sin(pi * FRA)	Ztrigpi
fcospi	cos times pi	FRT = cos(pi * FRA)	Ztrigpi
ftanpi	tan times pi	FRT = tan(pi * FRA)	Ztrigpi
fasinpi	arcsin / pi	FRT = asin(FRA) / pi	Zarctrigpi
facospi	arccos / pi	FRT = acos(FRA) / pi	Zarctrigpi
fatanpi	arctan / pi	FRT = atan(FRA) / pi	Zarctrigpi
fsinh	hyperbolic sin (radians)	FRT = sinh(FRA)	Zfhyp
fcosh	hyperbolic cos (radians)	FRT = cosh(FRA)	Zfhyp
ftanh	hyperbolic tan (radians)	FRT = tanh(FRA)	Zfhyp
fasinh	inverse hyperbolic sin	FRT = asinh(FRA)	Zfhyp
facosh	inverse hyperbolic cos	FRT = acosh(FRA)	Zfhyp
fatanh	inverse hyperbolic tan	FRT = atanh(FRA)	Zfhyp

8.5 Opcode Tables for PO=59/63 XO=1—011–

Power ISA v3.1B opcodes extracted from:

- Power ISA v3.1B Appendix D Table 23 sheet 2/3 of 4 page 1391/1392
- Power ISA v3.1B Appendix D Table 25 sheet 2/3 of 4 page 1399/1400

Parenthesized entries are not part of fptrans.

- Entries whose mnemonic ends in `s` are only in PO=59.
- Entries whose mnemonic does not end in `s` are only in PO=63.
- Entries whose mnemonic ends in `(s)` are in both PO=59 and PO=63.

XO LSB half → XO MSB half ↓	01100	01101	01110	01111
10000	`10000 01100` fcbrt(s) (draft)	`10000 01101` fsinpi(s) (draft)	`10000 01110` fatan2pi(s) (draft)	`10000 01111` fasinpi(s) (draft)
10001	`10001 01100` fcospi(s) (draft)	`10001 01101` ftanpi(s) (draft)	`10001 01110` facospi(s) (draft)	`10001 01111` fatanpi(s) (draft)
10010	`10010 01100` frsqrt(s) (draft)	`10010 01101` fsin(s) (draft)	`10010 01110` fatan2(s) (draft)	`10010 01111` fasin(s) (draft)
10011	`10011 01100` fcos(s) (draft)	`10011 01101` ftan(s) (draft)	`10011 01110` facos(s) (draft)	`10011 01111` fatan(s) (draft)
10100	`10100 01100` frecip(s) (draft)	`10100 01101` fsinh(s) (draft)	`10100 01110` fhypot(s) (draft)	`10100 01111` fasinh(s) (draft)
10101	`10101 01100` fcosh(s) (draft)	`10101 01101` ftanh(s) (draft)	`10101 01110` facosh(s) (draft)	`10101 01111` fatanh(s) (draft)
10110	`10110 01100`	`10110 01101`	`10110 01110`	`10110 01111`
10111	`10111 01100`	`10111 01101`	`10111 01110`	`10111 01111`

XO LSB half → XO MSB half ↓	01100	01101	01110	01111
11000	11000 01100 fexp2m1(s) (draft)	11000 01101 flog2p1(s) (draft)	11000 01110 (fcvttgo(s)) (draft)	11000 01111 (fcvtfg(s)) (draft)
11001	11001 01100 fexpm1(s) (draft)	11001 01101 flogp1(s) (draft)	11001 01110 (fctid)	11001 01111 (fctidz)
11010	11010 01100 fexp10m1(s) (draft)	11010 01101 flog10p1(s) (draft)	11010 01110 (fcfid(s))	11010 01111 fmod(s) (draft)
11011	11011 01100 fpown(s) (draft)	11011 01101 frootn(s) (draft)	11011 01110	11011 01111
11100	11100 01100 fexp2(s) (draft)	11100 01101 flog2(s) (draft)	11100 01110 (fmvtg(s)) (draft)	11100 01111 (fmvfg(s)) (draft)
11101	11101 01100 fexp(s) (draft)	11101 01101 flog(s) (draft)	11101 01110 (fctidu)	11101 01111 (fctiduz)
11110	11110 01100 fexp10(s) (draft)	11110 01101 flog10(s) (draft)	11110 01110 (fcfidu(s))	11110 01111 fremainder(s) (draft)
11111	11111 01100 fpowr(s) (draft)	11111 01101 fpow(s) (draft)	11111 01110	11111 01111

XO LSB half → XO MSB half ↓	10000	10001	10010	10011
////0	....0 10000 fminmax(s) (draft)	////0 10001	////0 10010 (fdiv(s))	////0 10011
////1	////1 10000	////1 10001	////1 10010 (fdiv(s))	////1 10011

8.6 DRAFT List of 2-arg opcodes

These are X-Form, recommended Major Opcode 63 for full-width and 59 for half-width (ending in s).

0.5	6.10	11.15	16.20	21..30	31	name	Form
NN	FRT	FRA	FRB	1xxxx011xx	Rc	transcendental	X-Form
NN	FRT	FRA	RB	1xxxx011xx	Rc	transcendental	X-Form
NN	FRT	FRA	FRB	xxxxx10000	Rc	transcendental	X-Form

Recommended 10-bit XO assignments:

opcode	Description	Major 59 and 63	bits 16..20
fatan2(s)	atan2 arc tangent	10010 01110	FRB
fatan2pi(s)	atan2 arc tangent / π	10000 01110	FRB
fpow(s)	x^y	11111 01101	FRB
fpown(s)	x^n ($n \in \mathbb{Z}$)	11011 01100	RB
fpowr(s)	x^y ($x >= 0$)	11111 01100	FRB
frootn(s)	$\sqrt[n]{x}$ ($n \in \mathbb{Z}$)	11011 01101	RB
fhypot(s)	$\sqrt{(x^2 + y^2)}$	10100 01110	FRB
fminmax(s)	min/max	....0 10000	FRB
fmod(s)	modulus	11010 01111	FRB

opcode	Description	Major 59 and 63	bits 16..20
fremainder(s)	IEEE 754 remainder	11110 01111	FRB

8.7 DRAFT List of 1-arg transcendental opcodes

These are X-Form, and are mostly identical in Special Registers Altered to `fsqrt` (the exact fp exceptions they can produce differ). Recommended Major Opcode 63 for full-width and 59 for half-width (ending in s).

Special Registers Altered (FIXME: come up with correct list):

```
FPRF FR FI FX OX UX XX
VXSNAN VXIMZ VXZDZ
CR1                    (if Rc=1)
```

0.5	6.10	11.15	16.20	21..30	31	name	Form
NN	FRT	///	FRB	1xxxx011xx	Rc	transcendental	X-Form

Recommended 10-bit XO assignments:

opcode	Description	Major 59 and 63
frsqrt(s)	$1 / \sqrt{x}$	10010 01100
fcbrt(s)	$\sqrt[3]{x}$	10000 01100
frecip(s)	$1 / x$	10100 01100
fexp2m1(s)	$2^x - 1$	11000 01100
flog2p1(s)	$\log_2 (x + 1)$	11000 01101
fexp2(s)	2^x	11100 01100
flog2(s)	$\log_2 x$	11100 01101
fexpm1(s)	$e^x - 1$	11001 01100
flogp1(s)	$\log_e (x + 1)$	11001 01101
fexp(s)	e^x	11101 01100
flog(s)	$\log_e x$	11101 01101
fexp10m1(s)	$10^x - 1$	11010 01100
flog10p1(s)	$\log_{10} (x + 1)$	11010 01101
fexp10(s)	10^x	11110 01100
flog10(s)	$\log_{10} x$	11110 01101

8.8 DRAFT List of 1-arg trigonometric opcodes

These are X-Form, and are mostly identical in Special Registers Altered to `fsqrt` (the exact fp exceptions they can produce differ). Recommended Major Opcode 63 for full-width and 59 for half-width (ending in s)

Special Registers Altered:

```
FPRF FR FI FX OX UX XX
VXSNAN VXIMZ VXZDZ
CR1                    (if Rc=1)
```

0.5	6.10	11.15	16.20	21..30	31	name	Form
NN	FRT	///	FRB	1xxxx011xx	Rc	trigonometric	X-Form

Recommended 10-bit XO assignments:

opcode	Description	Major 59 and 63
fsin(s)	sin (radians)	10010 01101
fcos(s)	cos (radians)	10011 01100
ftan(s)	tan (radians)	10011 01101
fasin(s)	arcsin (radians)	10010 01111
facos(s)	arccos (radians)	10011 01110
fatan(s)	arctan (radians)	10011 01111
fsinpi(s)	sin(π * x)	10000 01101
fcospi(s)	cos(π * x)	10001 01100
ftanpi(s)	tan(π * x)	10001 01101
fasinpi(s)	arcsin(x) / π	10000 01111
facospi(s)	arccos(x) / π	10001 01110
fatanpi(s)	arctan(x) / π	10001 01111
fsinh(s)	hyperbolic sin	10100 01101
fcosh(s)	hyperbolic cos	10101 01100
ftanh(s)	hyperbolic tan	10101 01101
fasinh(s)	inverse hyperbolic sin	10100 01111
facosh(s)	inverse hyperbolic cos	10101 01110
fatanh(s)	inverse hyperbolic tan	10101 01111

8.9 Subsets

The full set is based on the Khronos OpenCL opcodes. If implemented entirely it would be too much for both Embedded and also 3D.

The subsets are organised by hardware complexity, need (3D, HPC), however due to synthesis producing inaccurate results at the range limits, the less common subsets are still required for IEEE754 HPC.

MALI Midgard, an embedded / mobile 3D GPU, for example only has the following opcodes:

```
28 - fmin
2C - fmax
E8 - fatan_pt2
F0 - frcp (reciprocal)
F2 - frsqrt (inverse square root, 1/sqrt(x))
F3 - fsqrt (square root)
F4 - fexp2 (2^x)
F5 - flog2
F6 - fsin1pi
F7 - fcos1pi
F9 - fatan_pt1
```

These in FP32 and FP16 only: no FP64 hardware, at all.

Vivante Embedded/Mobile 3D (etnaviv https://github.com/laanwj/etna_viv/blob/master/rnndb/isa.xml) only has the following:

```
fmin/fmax (implemented using SELECT)
sin, cos2pi
cos, sin2pi
log2, exp
sqrt and rsqrt
recip.
```

It also has fast variants of some of these, as a CSR Mode.

AMD's R600 GPU (R600_Instruction_Set_Architecture.pdf) and the RDNA ISA (RDNA_Shader_ISA_5August2019.pdf, Table 22, Section 6.3) have:

```
MIN/MAX/MIN_DX10/MAX_DX10
COS2PI (appx)
EXP2
LOG (IEEE754)
RECIP
RSQRT
SQRT
SIN2PI (appx)
```

AMD RDNA has F16 and F32 variants of all the above, and also has F64 variants of SQRT, RSQRT, MIN, MAX, and RECIP. It is interesting that even the modern high-end AMD GPU does not have TAN or ATAN, where MALI Midgard does.

Also a general point, that customised optimised hardware targetting FP32 3D with less accuracy simply can neither be used for IEEE754 nor for FP64 (except as a starting point for hardware or software driven Newton Raphson or other iterative method).

Also in cost/area sensitive applications even the extra ROM lookup tables for certain algorithms may be too costly.

These wildly differing and incompatible driving factors lead to the subset subdivisions, below.

8.9.1 Transcendental Subsets

8.9.1.1 Zftrans

LOG2 EXP2 RECIP RSQRT

Zftrans contains the minimum standard transcendentals best suited to 3D. They are also the minimum subset for synthesising log10, exp10, exp1m, log1p, the hyperbolic trigonometric functions sinh and so on.

They are therefore considered "base" (essential) transcendentals.

8.9.1.2 ZftransExt

LOG, EXP, EXP10, LOG10, LOGP1, EXP1M, fmod, fremainder

These are extra transcendental functions that are useful, not generally needed for 3D, however for Numerical Computation they may be useful.

Although they can be synthesised using Ztrans (LOG2 multiplied by a constant), there is both a performance penalty as well as an accuracy penalty towards the limits, which for IEEE754 compliance is unacceptable. In particular, LOG(1+FRA) in hardware may give much better accuracy at the lower end (very small FRA) than LOG(FRA).

Their forced inclusion would be inappropriate as it would penalise embedded systems with tight power and area budgets. However if they were completely excluded the HPC applications would be penalised on performance and accuracy.

Therefore they are their own subset extension.

8.9.1.3 Zfhyp

SINH, COSH, TANH, ASINH, ACOSH, ATANH

These are the hyperbolic/inverse-hyperbolic functions. Their use in 3D is limited.

They can all be synthesised using LOG, SQRT and so on, so depend on Zftrans. However, once again, at the limits of the range, IEEE754 compliance becomes impossible, and thus a hardware implementation may be required.

HPC and high-end GPUs are likely markets for these.

8.9.1.4 ZftransAdv

CBRT, POW, POWN, POWR, ROOTN

These are simply much more complex to implement in hardware, and typically will only be put into HPC applications.

Note that `pow` is commonly used in Blinn-Phong shading (the shading model used by OpenGL 1.0 and commonly used by shader authors that need basic 3D graphics with specular highlights), however it can be sufficiently emulated using `pow(b, n) = exp2(n*log2(b))`.

- **Zfrsqrt**: Reciprocal square-root.

8.9.2 Trigonometric subsets

8.9.2.1 Ztrigpi vs Ztrignpi

- **Ztrigpi**: SINPI COSPI TANPI
- **Ztrignpi**: SIN COS TAN

Ztrignpi are the basic trigonometric functions through which all others could be synthesised, and they are typically the base trigonometrics provided by GPUs for 3D, warranting their own subset.

(programmerjake: actually, all other GPU ISAs mentioned in this document have sinpi/cospi or equivalent, and often not sin/cos, because sinpi/cospi are actually *waay* easier to implement because range reduction is simply a bitwise mask, whereas for sin/cos range reduction is a full division by pi)

(Mitch: My patent USPTO 10,761,806 shows that the above statement is no longer true.)

In the case of the Ztrigpi subset, these are commonly used in for loops with a power of two number of subdivisions, and the cost of multiplying by PI inside each loop (or cumulative addition, resulting in cumulative errors) is not acceptable.

In for example CORDIC the multiplication by PI may be moved outside of the hardware algorithm as a loop invariant, with no power or area penalty.

Again, therefore, if SINPI (etc.) were excluded, programmers would be penalised by being forced to divide by PI in some circumstances. Likewise if SIN were excluded, programmers would be penaslised by being forced to *multiply* by PI in some circumstances.

Thus again, a slightly different application of the same general argument applies to give Ztrignpi and Ztrigpi as subsets. 3D GPUs will almost certainly provide both.

8.9.2.2 Zarctrigpi and Zarctrignpi

- **Zarctrigpi**: ATAN2PI ASINPI ACOSPI
- **Zarctrignpi**: ATAN2 ACOS ASIN

These are extra trigonometric functions that are useful in some applications, but even for 3D GPUs, particularly embedded and mobile class GPUs, they are not so common and so are typically synthesised, there.

Although they can be synthesised using Ztrigpi and Ztrignpi, there is, once again, both a performance penalty as well as an accuracy penalty towards the limits, which for IEEE754 compliance is unacceptable, yet is acceptable for 3D.

Therefore they are their own subset extensions.

8.9.2.3 Zfminmax

- fminnum08 fmaxnum08
- fmin19 fmax19
- fminnum19 fmaxnum19
- fminc fmaxc
- fminmagnum08 fmaxmagnum08
- fminmag19 fmaxmag19
- fminmagnum19 fmaxmagnum19
- fminmagc fmaxmagc

These are commonly used for vector reductions, where having them be a single instruction is critical. They are also commonly used in GPU shaders, HPC, and general-purpose FP algorithms.

These min and max operations are quite cheap to implement hardware-wise, being comparable in cost to fcmp + some muxes. They're all in one extension because once you implement some of them, the rest require only slightly more hardware complexity.

Therefore they are their own subset extension.

8.10 Synthesis, Pseudo-code ops and macro-ops

The pseudo-ops are best left up to the compiler rather than being actual pseudo-ops, by allocating one scalar FP register for use as a constant (loop invariant) set to "1.0" at the beginning of a function or other suitable code block.

- fsincos - fused macro-op between fsin and fcos (issued in that order).
- fsincospi - fused macro-op between fsinpi and fcospi (issued in that order).

fatanpi example pseudo-code:

```
fmvis ft0, 0x3F80 // upper bits of f32 1.0 (BF16)
fatan2pis FRT, FRA, ft0
```

Hyperbolic function example (obviates need for Zfhyp except for high-performance or correctly-rounding):

```
ASINH( x ) = ln( x + SQRT(x**2+1))
```

`pow` sufficient for 3D Graphics:

```
pow(b, x) = exp2(x * log2(b))
```

8.11 Evaluation and commentary

Moved to [[discussion]]

Appendix G

Big Integer Analysis

[[!tag standards]]

G.1 Analysis

DRAFT SVP64

- Revision 0.0: 21apr2022 https://www.youtube.com/watch?v=8hrIG7-E77o
- Revision 0.01: 22apr2022 removal of msubed because sv.maddedu and sv.subfe works
- Revision 0.02: 22apr2022 128/64 scalar divide, investigate Goldschmidt
- Revision 0.03: 24apr2022 add 128/64 divmod2du, similar loop to maddedu
- Revision 0.04: 26apr2022 Knuth original uses overflow on scalar div
- Revision 0.05: 27apr2022 add vector shift section (no new instructions)

Introduction

This page covers an analysis of big integer operations, to work out optimal Scalar Instructions to propose be submitted to the OpenPOWER ISA WG, that when combined with Draft SVP64 give high performance compact Big Integer Vector Arithmetic. Leverage of existing Scalar Power ISA instructions is also explained.

Use of smaller sub-operations is a given: worst-case in a Scalar context, addition is O(N) whilst multiply and divide are O(N^2), and their Vectorisation would reduce those (for small N) to O(1) and O(N). Knuth's big-integer scalar algorithms provide useful real-world grounding into the types of operations needed, making it easy to demonstrate how they would be Vectorised.

The basic principle behind Knuth's algorithms is to break the problem down into a single scalar op against a Vector operand. *This fits naturally with a Scalable Vector ISA such as SVP64*. It only remains to exploit Carry (1-bit and 64-bit) in a Scalable Vector context and the picture is complete.

Links

- https://web.archive.org/web/20141021201141/https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf
- https://lists.libre-soc.org/pipermail/libre-soc-dev/2022-April/004700.html
- https://news.ycombinator.com/item?id=21151646
- https://twitter.com/lkcl/status/1517169267912986624
- https://www.youtube.com/watch?v=8hrIG7-E77o
- https://www.reddit.com/r/OpenPOWER/comments/u8r4vf/draft_svp64_biginteger_vector_arithmetic_for_the/
- https://bugs.libre-soc.org/show_bug.cgi?id=817

G.2 Vector Add and Subtract

Surprisingly, no new additional instructions are required to perform a straightforward big-integer add or subtract. Vectorised `adde` or `addex` is perfectly sufficient to produce arbitrary-length big-integer add due to the rules set in SVP64 that all Vector Operations are directly equivalent to the strict Program Order Execution of their element-level operations. Assuming that the two bigints (or a part thereof) have been loaded into sequentially-contiguous registers, with the least-significant bits being in the lowest-numbered register in each case:

```
R0,CA = A0+B0+CA  adde r0,a0,b0
    |
    +----------+
               |
R1,CA = A1+B1+CA  adde r1,a1,b1
    |
    +----------+
               |
R2,CA = A2+B2+CA  adde r2,a2,b2
```

This pattern - sequential execution of individual instructions with incrementing register numbers - is precisely the very definition of how SVP64 works! Thus, due to sequential execution of `adde` both consuming and producing a CA Flag, with no additions to SVP64 or to the v3.0 Power ISA, `sv.adde` is in effect an alias for Big-Integer Vectorised add. As such, implementors are entirely at liberty to recognise Horizontal-First Vector adds and send the vector of registers to a much larger and wider back-end ALU, and short-cut the intermediate storage of XER.CA on an element level in back-end hardware that need only:

- read the first incoming XER.CA
- implement a large Vector-aware carry propagation algorithm
- store the very last XER.CA in the batch

The size and implementation of the underlying back-end SIMD ALU is entirely at the discretion of the implementer, as is whether to deploy the above strategy. The only hard requirement for implementors of SVP64 is to comply with strict and precise Program Order even at the Element level.

If there is pressure on the register file (or multi-million-digit big integers) then a partial-sum may be carried out with LD and ST in a standard Cray-style Vector Loop:

```
  aptr = A address
  bptr = B address
  rptr = Result address
  li r0, 0        # used to help clear CA
  addic r0, r0, 0 # CA to zero as well
  setmvli 8       # set MAXVL to 8
loop:
  setvl t0, n         # n is the number of digits
  mulli t1, t0, 8     # 8 bytes per digit/element
  sv.ldu a0, aptr, t1 # update advances pointer
  sv.ldu b0, bptr, t1 # likewise
  sv.adde r0, a0, b0  # takes in CA, updates CA
  sv.stu rptr, r0, t1 # pointer advances too
  sub. n, n, t0       # should not alter CA
  bnz loop            # do more digits
```

This is not that different from a Scalar Big-Int add, it is just that like all Cray-style Vectorisation, a variable number of elements are covered by one instruction. Of interest to people unfamiliar with Cray-style Vectors: if VL is not permitted to exceed 1 (because MAXVL is set to 1) then the above actually becomes a Scalar Big-Int add algorithm.

G.3 Vector Shift

Like add and subtract, strictly speaking these need no new instructions. Keeping the shift amount within the range of the element (64 bit) a Vector bit-shift may be synthesised from a pair of shift operations and an OR, all of which are standard Scalar Power ISA instructions that when Vectorised are exactly what is needed.

```
void bigrsh(unsigned s, uint64_t r[], uint64_t un[], int n) {
    for (int i = 0; i < n - 1; i++)
        r[i] = (un[i] >> s) | (un[i + 1] << (64 - s));
    r[n - 1] = un[n - 1] >> s;
}
```

With SVP64 being on top of the standard scalar regfile the offset by one of the elements may be achieved simply by referencing the same vector data offset by one. Given that all three instructions (`srd`, `sld`, `or`) are an SVP64 type `RM-1P-2S1D` and are `EXTRA3`, it is possible to reference the full 128 64-bit registers (r0-r127):

```
subfic t1, t0, 64        # compute 64-s (s in t0)
sv.srd r8.v, r24.v, t0   # shift each element of r24.v up by s
sv.sld r16.v, r25.v, t1  # offset start of vector by one (r25)
sv.or  r8.v, r8.v, r16.v # OR two parts together
```

Predication with zeroing may be utilised on sld to ensure that the last element is zero, avoiding over-run.

The reason why three instructions are needed instead of one in the case of big-add is because multiple bits chain through to the next element, where for add it is a single bit (carry-in, carry-out), and this is precisely what `adde` already does. For multiply and divide as shown later it is worthwhile to use one scalar register effectively as a full 64-bit carry/chain but in the case of shift, an OR may glue things together, easily, and in parallel, because unlike `sv.adde`, down-chain carry-propagation through multiple elements does not occur.

With Scalar shift and rotate operations in the Power ISA already being complex and very comprehensive, it is hard to justify creating complex 3-in 2-out variants when a sequence of 3 simple instructions will suffice. However it is reasonably justifiable to have a 3-in 1-out instruction with an implicit source, based around the inner operation:

```
# r[i] = (un[i] >> s) | (un[i + 1] << (64 - s));
t <- ROT128(RA || RA1, RB[58:63])
RT <- t[64:127]
```

RA1 is implicitly (or explicitly, RC) greater than RA by one scalar register number, and like the other operations below, a 128/64 shift is performed, truncating to take the lower 64 bits. By taking a Vector source RA and assuming lower-numbered registers are lower-significant digits in the biginteger operation the entire biginteger source may be shifted by a scalar.

For larger shift amounts beyond an element bitwidth standard register move operations may be used, or, if the shift amount is static, to reference an alternate starting point in the registers containing the Vector elements because SVP64 sits on top of a standard Scalar register file. `sv.sld r16.v, r26.v, t1` for example is equivalent to shifting by an extra 64 bits, compared to `sv.sld r16.v, r25.v, t1`.

G.4 Vector Multiply

Long-multiply, assuming an O(N^2) algorithm, is performed by summing NxN separate smaller multiplications together. Karatsuba's algorithm reduces the number of small multiplies at the expense of increasing the number of additions. Some algorithms follow the Vedic Multiply pattern by grouping together all multiplies of the same magnitude/power (same column) whilst others perform row-based multiplication: a single digit of B multiplies the entirety of A, summed a row at a time. A Row-based algorithm is the basis of the analysis below (Knuth's Algorithm M).

Multiply is tricky: 64 bit operands actually produce a 128-bit result, which clearly cannot fit into an orthogonal register file. Most Scalar RISC ISAs have separate `mul-low-half` and `mul-hi-half` instructions, whilst some (OpenRISC) have "Accumulators" from which the results of the multiply must be explicitly extracted. High performance RISC advocates recommend "macro-op fusion" which is in effect where the second instruction gains access to the cached copy of the HI half of the multiply result, which had already been computed by the first. This approach quickly complicates the internal microarchitecture, especially at the decode phase.

Instead, Intel, in 2012, specifically added a `mulx` instruction, allowing both HI and LO halves of the multiply to reach registers with a single instruction. If however done as a multiply-and-accumulate this becomes quite an expensive operation: (3 64-Bit in, 2 64-bit registers out).

Long-multiplication may be performed a row at a time, starting with B0:

```
C4 C3 C2 C1 C0
         A0xB0
      A1xB0
   A2xB0
A3xB0
R4 R3 R2 R1 R0
```

- R0 contains C0 plus the LO half of A0 times B0
- R1 contains C1 plus the LO half of A1 times B0 plus the HI half of A0 times B0.
- R2 contains C2 plus the LO half of A2 times B0 plus the HI half of A1 times B0.

This would on the face of it be a 4-in operation: the upper half of a previous multiply, two new operands to multiply, and an additional accumulator (C). However if C is left out (and added afterwards with a Vector-Add) things become more manageable.

Demonstrating in c, a Row-based multiply using a temporary vector. Adapted from a simple implementation of Knuth M: https://git.libre-soc.org/?p=libreriscv.git;a=blob;f=openpower/sv/bitmanip/mulmnu.c;hb=HEAD

```
// this becomes the basis for sv.maddedu in RS=RC Mode,
// where k is RC. k takes the upper half of product
// and adds it in on the next iteration
k = 0;
for (i = 0; i < m; i++) {
   unsigned product = u[i]*v[j] + k;
   k = product>>16;
   plo[i] = product; // & 0xffff
}
// this is simply sv.adde where k is XER.CA
k = 0;
for (i = 0; i < m; i++) {
   t = plo[i] + w[i + j] + k;
   w[i + j] = t;          // (I.e., t & 0xFFFF).
   k = t >> 16; // carry: should only be 1 bit
}
```

We therefore propose an operation that is 3-in, 2-out, that, noting that the connection between successive mul-adds has the UPPER half of the previous operation as its input, writes the UPPER half of the current product into a second output register for exactly the purpose of letting it be added onto the next BigInt digit.

```
product = RA*RB+RC
RT = lowerhalf(product)
RC = upperhalf(product)
```

Horizontal-First Mode therefore may be applied to just this one instruction. Successive sequential iterations effectively use RC as a kind of 64-bit carry, and as noted by Intel in their notes on mulx, `RA*RB+RC+RD` cannot overflow, so does not require setting an additional CA flag. We first cover the chain of `RA*RB+RC` as follows:

```
RT0, RC0 = RA0 * RB0 + 0
      |
      +----------------+
                       |
RT1, RC1 = RA1 * RB1 + RC0
      |
      +----------------+
                       |
RT2, RC2 = RA2 * RB2 + RC1
```

Following up to add each partially-computed row to what will become the final result is achieved with a Vectorised big-int `sv.adde`. Thus, the key inner loop of Knuth's Algorithm M may be achieved in four instructions, two of which are scalar initialisation:

```
li r16, 0                     # zero accumulator
addic r16, r16, 0             # CA to zero as well
sv.madde r0.v, r8.v, r17, r16 # mul vector
sv.adde r24.v, r24.v, r0.v   # big-add row to result
```

Normally, in a Scalar ISA, the use of a register as both a source and destination like this would create costly Dependency Hazards, so such an instruction would never be proposed. However: it turns out that, just as with repeated chained application of `adde`, macro-op fusion may be internally applied to a sequence of these strange multiply operations. (*Such a trick works equally as well in a Scalar-only Out-of-Order microarchitecture, although the conditions are harder to detect*).

Application of SVP64

SVP64 has the means to mark registers as scalar or vector. However the available space in the prefix is extremely limited (9 bits). With effectively 5 operands (3 in, 2 out) some compromises are needed. A little thought gives a useful workaround: two modes, controlled by a single bit in `RM.EXTRA`, determine whether the 5th register is set to RC or whether to RT+MAXVL. This then leaves only 4 registers to qualify as scalar/vector, which can use four EXTRA2 designators and fits into the available 9-bit space.

RS=RT+MAXVL Mode:

```
product = RA*RB+RC
RT = lowerhalf(product)
RS=RT+MAXVL = upperhalf(product)
```

and RS=RC Mode:

```
product = RA*RB+RC
RT = lowerhalf(product)
RS=RC = upperhalf(product)
```

Now there is much more potential, including setting RC to a Scalar, which would be useful as a 64 bit Carry. RC as a Vector would produce a Vector of the HI halves of a Vector of multiplies. RS=RT+MAXVL Mode would allow that same Vector of HI halves to not be an overwrite of RC. Also it is possible to specify that any of RA, RB or RC are scalar or vector. Overall it is extremely powerful.

G.5 Vector Divide

The simplest implementation of big-int divide is the standard schoolbook "Long Division", set with RADIX 64 instead of Base 10. Donald Knuth's Algorithm D performs estimates which, if wrong, are compensated for afterwards. Essentially there are three phases:

- Calculation of the quotient estimate. This uses a single Scalar divide, which is covered separately in a later section
- Big Integer multiply and subtract.

- Carry-Correction with a big integer add, if the estimate from phase 1 was wrong by one digit.

From Knuth's Algorithm D, implemented in divmnu64.c, Phase 2 is expressed in c, as:

```
// Multiply and subtract.
k = 0;
for (i = 0; i < n; i++) {
   p = qhat*vn[i]; // 64-bit product
   t = un[i+j] - k - (p & 0xFFFFFFFFLL);
   un[i+j] = t;
   k = (p >> 32) - (t >> 32);
}
```

Where analysis of this algorithm, if a temporary vector is acceptable, shows that it can be split into two in exactly the same way as Algorithm M, this time using subtract instead of add.

```
uint32_t carry = 0;
// this is just sv.maddedu again
for (int i = 0; i <= n; i++) {
    uint64_t value = (uint64_t)vn[i] * (uint64_t)qhat + carry;
    carry = (uint32_t)(value >> 32); // upper half for next loop
    product[i] = (uint32_t)value;    // lower into vector
}
bool ca = true;
// this is simply sv.subfe where ca is XER.CA
for (int i = 0; i <= n; i++) {
    uint64_t value = (uint64_t)~product[i] + (uint64_t)un_j[i] + ca;
    ca = value >> 32 != 0;
    un_j[i] = value;
}
bool need_fixup = !ca; // for phase 3 correction
```

In essence then the primary focus of Vectorised Big-Int divide is in fact big-integer multiply

Detection of the fixup (phase 3) is determined by the Carry (borrow) bit at the end. Logically: if borrow was required then the qhat estimate was too large and the correction is required, which is, again, nothing more than a Vectorised big-integer add (one instruction). However this is not the full story

128/64-bit divisor

As mentioned above, the first part of the Knuth Algorithm D involves computing an estimate for the divisor. This involves using the three most significant digits, performing a scalar divide, and consequently requires a scalar division with *twice* the number of bits of the size of individual digits (for example, a 64-bit array). In this example taken from divmnu64.c the digits are 32 bit and, special-casing the overflow, a 64/32 divide is sufficient (64-bit dividend, 32-bit divisor):

```
// Compute estimate qhat of q[j] from top 2 digits
uint64_t dig2 = ((uint64_t)un[j + n] << 32) | un[j + n - 1];
if (un[j+n] >= vn[n-1]) {
    // rhat can be bigger than 32-bit when the division overflows
    qhat = UINT32_MAX;
    rhat = dig2 - (uint64_t)UINT32_MAX * vn[n - 1];
} else {
    qhat = dig2 / vn[n - 1]; // 64/32 divide
    rhat = dig2 % vn[n - 1]; // 64/32 modulo
}
// use 3rd-from-top digit to obtain better accuracy
b = 1UL<<32;
while (rhat < b || qhat * vn[n - 2] > b * rhat + un[j + n - 2]) {
    qhat = qhat - 1;
```

```
            rhat = rhat + vn[n - 1];
        }
```

However when moving to 64-bit digits (desirable because the algorithm is `O(N^2)`) this in turn means that the estimate has to be computed from a *128* bit dividend and a 64-bit divisor. Such an operation simply does not exist in most Scalar 64-bit ISAs. Although Power ISA comes close with `divdeu`, by placing one operand in the upper half of a 128-bit dividend, the lower half is zero. Again Power ISA has a Packed SIMD instruction `vdivuq` which is a 128/128 (quad) divide, not a 128/64, and its use would require considerable effort to move registers to and from GPRs. Some investigation into soft-implementations of 128/128 or 128/64 divide show it to be typically implemented bit-wise, with all that implies.

The irony is, therefore, that attempting to improve big-integer divide by moving to 64-bit digits in order to take advantage of the efficiency of 64-bit scalar multiply when Vectorised would instead lock up CPU time performing a 128/64 scalar division. With the Vector Multiply operations being critically dependent on that `qhat` estimate, and because that scalar is as an input into each of the vector digit multiples, as a Dependency Hazard it would cause *all* Parallel SIMD Multiply back-ends to sit 100% idle, waiting for that one scalar value.

Whilst one solution is to reduce the digit width to 32-bit in order to go back to 64/32 divide, this increases the completion time by a factor of 4 due to the algorithm being `O(N^2)`.

Reducing completion time of 128/64-bit Scalar division

Scalar division is a known computer science problem because, as even the Big-Int Divide shows, it requires looping around a multiply (or, if reduced to 1-bit per loop, a simple compare, shift, and subtract). If the simplest approach were deployed then the completion time for the 128/64 scalar divide would be a whopping 128 cycles. To be workable an alternative algorithm is required, and one of the fastest appears to be Goldschmidt Division. Whilst typically deployed for Floating Point, there is no reason why it should not be adapted to Fixed Point. In this way a Scalar Integer divide can be performed in the same time-order as Newton-Raphson, using two hardware multipliers and a subtract.

Back to Vector carry-looping

There is however another reason for having a 128/64 division instruction, and it's effectively the reverse of `maddedu`. Look closely at Algorithm D when the divisor is only a scalar (`v[0]`):

```
        k = 0; // the case of a
        for (j = m - 1; j >= 0; j--)
        {                                   // single-digit
            uint64_t dig2 = ((k << 32) | u[j]);
            q[j] = dig2 / v[0]; // divisor here.
            k = dig2 % v[0]; // modulo back into next loop
        }
```

Here, just as with `maddedu` which can put the hi-half of the 128 bit product back in as a form of 64-bit carry, a scalar divisor of a vector dividend puts the modulo back in as the hi-half of a 128/64-bit divide.

```
RT0      = ((  0<<64) | RA0) / RB0
     RC0 = ((  0<<64) | RA0) % RB0
      |
      +-------+
              |
RT1      = ((RC0<<64) | RA1) / RB1
     RC1 = ((RC0<<64) | RA1) % RB1
      |
      +-------+
              |
RT2      = ((RC1<<64) | RA2) / RB2
     RC2 = ((RC1<<64) | RA2) % RB2
```

By a nice coincidence this is exactly the same 128/64-bit operation needed (once, rather than chained) for the `qhat` estimate if it may produce both the quotient and the remainder. The pseudocode cleanly covering both

scenarios (leaving out overflow for clarity) can be written as:

```
divmod2du RT,RA,RB,RC

       dividend = (RC) || (RA)
       divisor = EXTZ128(RB)
       RT = UDIV(dividend, divisor)
       RS = UREM(dividend, divisor)
```

Again, in an SVP64 context, using EXTRA mode bit 8 allows for selecting whether `RS=RC` or `RS=RT+MAXVL`. Similar flexibility in the scalar-vector settings allows the instruction to perform full parallel vector div/mod, or act in loop-back mode for big-int division by a scalar, or for a single scalar 128/64 div/mod.

Again, just as with `sv.maddedu` and `sv.adde`, adventurous implementors may perform massively-wide DIV/MOD by transparently merging (fusing) the Vector element operations together, only inputting a single RC and outputting the last RC. Where efficient algorithms such as Goldschmidt are deployed internally this could dramatically reduce the cycle completion time for massive Vector DIV/MOD. Thus, just as with the other operations the apparent limitation of creating chains is overcome: SVP64 is, by design, an "expression of intent" where the implementor is free to achieve that intent in any way they see fit as long as strict precise-aware Program Order is preserved (even on the VL for-loops).

Just as with `divdeu` on which this instruction is based an overflow detection is required. When the divisor is too small compared to the dividend then the result may not fit into 64 bit. Knuth's original algorithm detects overflow and manually places 0xffffffff (all ones) into `qhat`. With there being so many operands already in `divmod2du` a `cmpl` instruction can be used instead to detect the overflow. This saves having to add an Rc=1 or OE=1 mode when the available space in VA-Form EXT04 is extremely limited.

Looking closely at the loop however we can see that overflow will not occur. The initial value k is zero: as long as a divide-by-zero is not requested this always fulfils the condition `RC < RA`, and on subsequent iterations the new k, being the modulo, is always less than the divisor as well. Thus the condition (the loop invariant) `RC < RA` is preserved, as long as RC starts at zero.

Limitations

One of the worst things for any ISA is that an algorithm's completion time is directly affected by different implementations having instructions take longer or shorter times. Knuth's Big-Integer division is unfortunately one such algorithm.

Assuming that the computation of qhat takes 128 cycles to complete on a small power-efficient embedded design, this time would dominate compared to the 64 bit multiplications. However if the element width was reduced to 8, such that the computation of qhat only took 16 cycles, the calculation of qhat would not dominate, but the number of multiplications would rise: somewhere in between there would be an elwidth and a Vector Length that would suit that particular embedded processor.

By contrast a high performance microarchitecture may deploy Goldschmidt or other efficient Scalar Division, which could complete 128/64 qhat computation in say only 5 to 8 cycles, which would be tolerable. Thus, for general-purpose software, it would be necessary to ship multiple implementations of the same algorithm and dynamically select the best one.

The very fact that programmers even have to consider multiple implementations and compare their performance is an unavoidable nuisance. SVP64 is supposed to be designed such that only one implementation of any given algorithm is needed. In some ways it is reassuring that some algorithms just don't fit. Slightly more reassuring is that Goldschmidt Divide, which uses two multiplications that can be performed in parallel, would be a much better fit with SVP64 (and Vector Processing in general), the only downside being that it is regarded as worthwhile for much larger integers.

G.6 Conclusion

TODO

Appendix H

Bitmanip pseudocode

H.1 Ternary Bitwise Logic Immediate

TLI-Form

- ternlogi RT,RA,RB,TLI (Rc=0)
- ternlogi. RT,RA,RB,TLI (Rc=1)

Pseudo-code:

```
result <- [0] * XLEN
do i = 0 to XLEN - 1
  idx <- (RT)[i] || (RA)[i] || (RB)[i]
  result[i] <- TLI[7-idx]
RT <- result
```

Special Registers Altered:

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

H.2 Generalized Bit-Reverse

X-Form

- grev RT,RA,RB (Rc=0)
- grev. RT,RA,RB (Rc=1)

Pseudo-code:

```
result <- [0] * XLEN
b <- EXTZ64(RB)
do i = 0 to XLEN - 1
  idx <- b[64-log2(XLEN):63] ^ i
  result[i] <- (RA)[idx]
RT <- result
```

Special Registers Altered:

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

H.3 Generalized Bit-Reverse Immediate

XB-Form

- grevi RT,RA,XBI (Rc=0)
- grevi. RT,RA,XBI (Rc=1)

Pseudo-code:

```
result <- [0] * XLEN
do i = 0 to XLEN - 1
  idx <- XBI[6-log2(XLEN):5] ^ i
  result[i] <- (RA)[idx]
RT <- result
```

Special Registers Altered:

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

H.4 Generalized Bit-Reverse Word

X-Form

- grevw RT,RA,RB (Rc=0)
- grevw. RT,RA,RB (Rc=1)

Pseudo-code:

```
result <- [0] * (XLEN / 2)
a <- (RA)[XLEN/2:XLEN-1]
b <- EXTZ64(RB)
do i = 0 to XLEN / 2 - 1
  idx <- b[64-log2(XLEN/2):63] ^ i
  result[i] <- a[idx]
RT <- ([0] * (XLEN / 2)) || result
```

Special Registers Altered:

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

H.5 Generalized Bit-Reverse Word Immediate

X-Form

- grevwi RT,RA,SH (Rc=0)
- grevwi. RT,RA,SH (Rc=1)

Pseudo-code:

```
result <- [0] * (XLEN / 2)
a <- (RA)[XLEN/2:XLEN-1]
do i = 0 to XLEN / 2 - 1
  idx <- SH[5-log2(XLEN/2):4] ^ i
  result[i] <- a[idx]
RT <- ([0] * (XLEN / 2)) || result
```

Special Registers Altered:

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

H.6 Add With Shift By Immediate

Z23-Form

- shadd RT,RA,RB,sm (Rc=0)
- shadd. RT,RA,RB,sm (Rc=1)

Pseudo-code:

```
n <- (RB)
m <- ((0b0 || sm) + 1)
RT <- (n[m:XLEN-1] || [0]*m) + (RA)
```

Special Registers Altered:

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

H.7 Add With Shift By Immediate Word

Z23-Form

- shaddw RT,RA,RB,sm (Rc=0)
- shaddw. RT,RA,RB,sm (Rc=1)

Pseudo-code:

```
n <- ([0]*(XLEN/2)) || (RB)[XLEN/2:XLEN-1]
if (RB)[XLEN/2] = 1 then
    n[0:XLEN/2-1] <- [1]*(XLEN/2)
m <- ((0b0 || sm) + 1)
RT <- (n[m:XLEN-1] || [0]*m) + (RA)
```

Special Registers Altered:

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

H.8 Add With Shift By Immediate Unsigned Word

Z23-Form

- shadduw RT,RA,RB,sm (Rc=0)
- shadduw. RT,RA,RB,sm (Rc=1)

Pseudo-code:

```
n <- ([0]*(XLEN/2)) || (RB)[XLEN/2:XLEN-1]
m <- ((0b0 || sm) + 1)
RT <- (n[m:XLEN-1] || [0]*m) + (RA)
```

Special Registers Altered:

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

Appendix I

Floating Point pseudocode

I.1 [DRAFT] Floating Add FFT/DCT [Single]

A-Form

- ffadds FRT,FRA,FRB (Rc=0)
- ffadds. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
FRT <- FPADD32(FRA, FRB)
FRS <- FPSUB32(FRB, FRA)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1          (if Rc=1)
```

I.2 [DRAFT] Floating Add FFT/DCT [Double]

A-Form

- ffadd FRT,FRA,FRB (Rc=0)
- ffadd. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
FRT <- FPADD64(FRA, FRB)
FRS <- FPSUB64(FRB, FRA)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1          (if Rc=1)
```

I.3 [DRAFT] Floating Subtract FFT/DCT [Single]

A-Form

- ffsubs FRT,FRA,FRB (Rc=0)
- ffsubs. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
FRT <- FPSUB32(FRB, FRA)
FRS <- FPADD32(FRA, FRB)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1          (if Rc=1)
```

I.4 [DRAFT] Floating Subtract FFT/DCT [Double]

A-Form

- ffsub FRT,FRA,FRB (Rc=0)
- ffsub. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
FRT <- FPSUB64(FRB, FRA)
FRS <- FPADD64(FRA, FRB)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1          (if Rc=1)
```

I.5 [DRAFT] Floating Multiply FFT/DCT [Single]

A-Form

- ffmuls FRT,FRA,FRC (Rc=0)
- ffmuls. FRT,FRA,FRC (Rc=1)

Pseudo-code:

```
FRT <- FPMUL32(FRA, FRC)
FRS <- FPMUL32(FRA, FRC, -1)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1          (if Rc=1)
```

I.6 [DRAFT] Floating Multiply FFT/DCT [Double]

A-Form

- ffmul FRT,FRA,FRC (Rc=0)
- ffmul. FRT,FRA,FRC (Rc=1)

Pseudo-code:

```
FRT <- FPMUL64(FRA, FRC)
FRS <- FPMUL64(FRA, FRC, -1)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1          (if Rc=1)
```

I.7 [DRAFT] Floating Divide FFT/DCT [Single]

A-Form

- ffdivs FRT,FRA,FRB (Rc=0)
- ffdivs. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
FRT <- FPDIV32(FRA, FRB)
FRS <- FPDIV32(FRA, FRB, -1)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1          (if Rc=1)
```

I.8 [DRAFT] Floating Divide FFT/DCT [Double]

A-Form

- ffdiv FRT,FRA,FRB (Rc=0)
- ffdiv. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
FRT <- FPDIV64(FRA, FRB)
FRS <- FPDIV64(FRA, FRB, -1)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1          (if Rc=1)
```

I.9 [DRAFT] Floating Twin Multiply-Add DCT [Single]

DCT-Form

- fdmadds FRT,FRA,FRB (Rc=0)
- fdmadds. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
FRS <- FPADD32(FRT, FRB)
sub <- FPSUB32(FRT, FRB)
FRT <- FPMUL32(FRA, sub)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1          (if Rc=1)
```

I.10 [DRAFT] Floating Multiply-Add FFT [Single]

A-Form

- ffmadds FRT,FRA,FRB (Rc=0)
- ffmadds. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
tmp <- FRT
FRT <- FPMULADD32(tmp, FRA, FRB, 1, 1)
FRS <- FPMULADD32(tmp, FRA, FRB, -1, 1)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1          (if Rc=1)
```

I.11 [DRAFT] Floating Multiply-Sub FFT [Single]

A-Form

- ffmsubs FRT,FRA,FRB (Rc=0)
- ffmsubs. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
tmp <- FRT
FRT <- FPMULADD32(tmp, FRA, FRB, 1, -1)
FRS <- FPMULADD32(tmp, FRA, FRB, -1, -1)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1          (if Rc=1)
```

I.12 [DRAFT] Floating Negative Multiply-Add FFT [Single]

A-Form

- ffnmadds FRT,FRA,FRB (Rc=0)
- ffnmadds. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
tmp <- FRT
FRT <- FPMULADD32(tmp, FRA, FRB, -1, -1)
FRS <- FPMULADD32(tmp, FRA, FRB, 1, -1)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1          (if Rc=1)
```

I.13 [DRAFT] Floating Negative Multiply-Sub FFT [Single]

A-Form

- ffnmsubs FRT,FRA,FRB (Rc=0)
- ffnmsubs. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
tmp <- FRT
FRT <- FPMULADD32(tmp, FRA, FRB, -1, 1)
FRS <- FPMULADD32(tmp, FRA, FRB, 1, 1)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1          (if Rc=1)
```

Appendix J

Fixed Point pseudocode

J.1 [DRAFT] Multiply and Add Extended Doubleword Unsigned

VA-Form

- maddedu RT,RA,RB,RC

Pseudo-code:

```
<!-- SVP64: RA,RB,RC,RT have EXTRA2, RS as below
<!-- bit 8 of EXTRA is set  : RS.[s|v]=RT.[s|v]+MAXVL
<!-- bit 8 of EXTRA is clear: RS.[s|v]=RC.[s|v]
prod[0:2*XLEN-1] <- (RA) * (RB)
sum[0:2*XLEN-1] <- ([0] * XLEN || (RC)) + prod
RT <- sum[XLEN:2*XLEN-1]
RS <- sum[0:XLEN-1]
```

Special Registers Altered:

```
None
```

J.2 [DRAFT] Multiply and Add Extended Doubleword Unsigned Signed

VA-Form

- maddedus RT,RA,RB,RC

Pseudo-code:

```
<!-- SVP64: RA,RB,RC,RT have EXTRA2, RS as below
<!-- bit 8 of EXTRA is set  : RS.[s|v]=RT.[s|v]+MAXVL
<!-- bit 8 of EXTRA is clear: RS.[s|v]=RC.[s|v]
<!-- no MULUS, so do it manually -->
prod[0:XLEN*2-1] <- [0] * (XLEN * 2)
if (RB)[0] != 0 then
    prod[0:XLEN*2-1] <- -((RA) * -(RB))
else
    prod[0:XLEN*2-1] <- (RA) * (RB)
<!-- no EXTS2XL, so do it manually -->
sum[0:XLEN*2-1] <- prod + (EXTSXL((RC)[0], 1) || (RC))
```

```
RT <- sum[XLEN:2*XLEN-1]
RS <- sum[0:XLEN-1]
```

Special Registers Altered:

```
None
```

J.3 [DRAFT] Divide/Modulo Double-width Doubleword Unsigned

VA-Form

- divmod2du RT,RA,RB,RC

Pseudo-code:

```
<!-- SVP64: RA,RB,RC,RT have EXTRA2, RS as below
<!-- bit 8 of EXTRA is set  : RS.[s|v]=RT.[s|v]+MAXVL
<!-- bit 8 of EXTRA is clear: RS.[s|v]=RC.[s|v]
if ((RC) <u (RB)) & ((RB) != [0]*XLEN) then
    dividend[0:(XLEN*2)-1] <- (RC) || (RA)
    divisor[0:(XLEN*2)-1] <- [0]*XLEN || (RB)
    result <- dividend / divisor
    modulo <- dividend % divisor
    RT <- result[XLEN:(XLEN*2)-1]
    RS <- modulo[XLEN:(XLEN*2)-1]
    overflow <- 0
else
    overflow <- 1
    RT <- [1]*XLEN
    RS <- [0]*XLEN
```

Special Registers Altered:

```
XER.OV
```

J.4 [DRAFT] Double-width Shift Left Doubleword

VA2-Form

- dsld RT,RA,RB,RC (Rc=0)
- dsld. RT,RA,RB,RC (Rc=1)

Pseudo-code:

```
n <- (RB)[58:63]
v <- ROTL64((RA), n)
mask <- MASK(0, 63-n)
RT <- (v[0:63] & mask) | ((RC) & ¬mask)
RS <- v[0:63] & ¬mask
overflow <- 0             # relevant only when Rc=1
if RS != [0]*64 then
    overflow <- 1         # relevant only when Rc=1
```

Special Registers Altered:

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

J.5 [DRAFT] Double-width Shift Right Doubleword

VA2-Form

- dsrd RT,RA,RB,RC (Rc=0)
- dsrd. RT,RA,RB,RC (Rc=1)

Pseudo-code:

```
n <- (RB)[58:63]
v <- ROTL64((RA), 64-n)
mask <- MASK(n, 63)
RT <- (v[0:63] & mask) | ((RC) & ¬mask)
RS <- v[0:63] & ¬mask
overflow <- 0             # relevant only when Rc=1
if RS != [0]*64 then
    overflow <- 1         # relevant only when Rc=1
```

Special Registers Altered:

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

Part IV

Scalar Power ISA pseudocode

Preamble

This section contains updated pseudocode from the Power ISA Specification v3.0B to be executable. Several bugfixes in Power ISA v3.0B have been found and reported as a direct result due to actually running the pseudocode as executable code in a Simulator. A Formal Correctness Proof Research Paper written by Boris Shingarov.

Additionally, with SVP64 performing element-width over-rides it is the *Scalar* pseudocode that needs adapting to variable-length (**XLEN**). Maintaining duplicate identical copies in every respect *except* for an XLEN as part of the Simple-V Specification is completely pointless and a waste of time: the updates to include XLEN need to be part of the Scalar Power ISA Specification. This has the added benefit that it makes life much easier for 32-bit implementors, and has an additional benefit of making it possible for the Scalar Power ISA to extend to 128-bit in future (like RV128).

Binary Coded Decimal pseudocode

J.1 Convert Declets To Binary Coded Decimal

X-Form

- cdtbcd RA,RS

Pseudo-code:

```
src <- [0]*64
src[64-XLEN:63] <- (RS)
result <- [0]*64
do i = 0 to 1
  n <- i * 32
  result[n+0:n+7] <- 0
  result[n+8:n+19] <- DPD_TO_BCD(src[n+12:n+21])
  result[n+20:n+31] <- DPD_TO_BCD(src[n+22:n+31])
RA <- result[64-XLEN:63]
```

Special Registers Altered:

```
None
```

J.2 Add and Generate Sixes

XO-Form

- addg6s RT,RA,RB

Pseudo-code:

```
sum <- (0b0000 || (RA)) + (0b0000 || (RB))
carries <- sum ^ (0b0000 || (RA)) ^ (0b0000 || (RB))
ones <- [0b0001] * (XLEN / 4)
nibbles_need_sixes <- ¬carries[0:XLEN-1] & ones
RT <- nibbles_need_sixes * 0b0110
```

Special Registers Altered:

```
None
```

J.3 Convert Binary Coded Decimal To Declets

X-Form

- cbcdtd RA,RS

Pseudo-code:

```
src <- [0]*64
src[64-XLEN:63] <- (RS)
result <- [0]*64
do i = 0 to 1
  n <- i * 32
  result[n+0:n+11] <- 0
  result[n+12:n+21] <- BCD_TO_DPD(src[n+8:n+19])
  result[n+22:n+31] <- BCD_TO_DPD(src[n+20:n+31])
RA <- result[64-XLEN:63]
```

Special Registers Altered:

```
None
```

Branch pseudocode

J.4 Branch

I-Form

- b target_addr (AA=0 LK=0)
- ba target_addr (AA=1 LK=0)
- bl target_addr (AA=0 LK=1)
- bla target_addr (AA=1 LK=1)

Pseudo-code:

```
if AA then NIA  <-iea EXTS(LI || 0b00)
else       NIA  <-iea CIA + EXTS(LI || 0b00)
if LK then LR <-iea  CIA + 4
```

Special Registers Altered:

```
LR                        (if LK=1)
```

J.5 Branch Conditional

B-Form

- bc BO,BI,target_addr (AA=0 LK=0)
- bca BO,BI,target_addr (AA=1 LK=0)
- bcl BO,BI,target_addr (AA=0 LK=1)
- bcla BO,BI,target_addr (AA=1 LK=1)

Pseudo-code:

```
if (mode_is_64bit) then M <- 0
else M <- 32
if ¬BO[2] then CTR <- CTR - 1
ctr_ok <- BO[2] | ((CTR[M:63] != 0) ^ BO[3])
cond_ok <- BO[0] | ¬(CR[BI+32] ^ BO[1])
if ctr_ok & cond_ok then
  if AA then NIA <-iea EXTS(BD || 0b00)
  else       NIA <-iea CIA + EXTS(BD || 0b00)
if LK then LR  <-iea  CIA + 4
```

Special Registers Altered:

```
CTR                     (if BO2=0)
LR                        (if LK=1)
```

J.6 Branch Conditional to Link Register

XL-Form

- bclr BO,BI,BH (LK=0)
- bclrl BO,BI,BH (LK=1)

Pseudo-code:

```
if (mode_is_64bit) then M <- 0
else M <- 32
if ¬BO[2]  then CTR <- CTR - 1
ctr_ok <- BO[2] | ((CTR[M:63] != 0) ^ BO[3])
cond_ok <- BO[0] | ¬(CR[BI+32] ^  BO[1])
if ctr_ok & cond_ok then NIA <-iea LR[0:61] || 0b00
if LK then LR <-iea CIA + 4
```

Special Registers Altered:

```
CTR                    (if BO2=0)
LR                       (if LK=1)
```

J.7 Branch Conditional to Count Register

XL-Form

- bcctr BO,BI,BH (LK=0)
- bcctrl BO,BI,BH (LK=1)

Pseudo-code:

```
cond_ok <- BO[0] | ¬(CR[BI+32] ^ BO[1])
if cond_ok then NIA <-iea CTR[0:61] || 0b00
if LK then LR <-iea CIA + 4
```

Special Registers Altered:

```
LR                     (if LK=1)
```

J.8 Branch Conditional to Branch Target Address Register

XL-Form

- bctar BO,BI,BH (LK=0)
- bctarl BO,BI,BH (LK=1)

Pseudo-code:

```
if (mode_is_64bit) then M <- 0
else M <- 32
if ¬BO[2] then CTR <- CTR - 1
ctr_ok <- BO[2]  | ((CTR[M:63] != 0) ^ BO[3])
cond_ok <- BO[0] | ¬(CR[BI+32] ^ BO[1])
if ctr_ok & cond_ok then NIA <-iea TAR[0:61]  || 0b00
if LK then LR <-iea  CIA + 4
```

Special Registers Altered:

```
CTR                     (if BO2=0)
LR                        (if LK=1)
```

Fixed Point Compare pseudocode

J.9 Compare Immediate

D-Form

- cmpi BF,L,RA,SI

Pseudo-code:

```
if L = 0 then a <- EXTS((RA)[XLEN/2:XLEN-1])
else a <-  (RA)
if      a < EXTS(SI) then c <- 0b100
else if a > EXTS(SI) then c <- 0b010
else                      c <- 0b001
CR[4*BF+32:4*BF+35] <- c || XER[SO]
```

Special Registers Altered:

```
CR field BF
```

J.10 Compare

X-Form

- cmp BF,L,RA,RB

Pseudo-code:

```
if L = 0 then
    a <- EXTS((RA)[XLEN/2:XLEN-1])
    b <- EXTS((RB)[XLEN/2:XLEN-1])
else
    a <- (RA)
    b <- (RB)
if      a < b then c <-  0b100
else if a > b then c <-  0b010
else               c <-  0b001
CR[4*BF+32:4*BF+35] <-  c || XER[SO]
```

Special Registers Altered:

```
CR field BF
```

J.11 Compare Logical Immediate

D-Form

- cmpli BF,L,RA,UI

Pseudo-code:

```
if L = 0 then a <- [0]*(XLEN/2) || (RA)[XLEN/2:XLEN-1]
else a <- (RA)
if      a <u ([0]*(XLEN-16) || UI) then c <- 0b100
else if a >u ([0]*(XLEN-16) || UI) then c <- 0b010
else                                    c <- 0b001
CR[4*BF+32:4*BF+35] <- c || XER[SO]
```

Special Registers Altered:

```
CR field BF
```

J.12 Compare Logical

X-Form

- cmpl BF,L,RA,RB

Pseudo-code:

```
if L = 0 then
    a <- [0]*(XLEN/2) || (RA)[XLEN/2:XLEN-1]
    b <- [0]*(XLEN/2) || (RB)[XLEN/2:XLEN-1]
else
    a <-  (RA)
    b <-  (RB)
if      a <u b then c <- 0b100
else if a >u b then c <- 0b010
else                c <-  0b001
CR[4*BF+32:4*BF+35] <- c || XER[SO]
```

Special Registers Altered:

```
CR field BF
```

J.13 Compare Ranged Byte

X-Form

- cmprb BF,L,RA,RB

Pseudo-code:

```
src1    <- EXTZ((RA)[XLEN-8:XLEN-1])
src21hi <- EXTZ((RB)[XLEN-32:XLEN-23])
src21lo <- EXTZ((RB)[XLEN-24:XLEN-17])
src22hi <- EXTZ((RB)[XLEN-16:XLEN-9])
src22lo <- EXTZ((RB)[XLEN-8:XLEN-1])
if L=0 then
   in_range <-  (src22lo  <= src1) & (src1 <=  src22hi)
else
```

```
    in_range <- (((src21lo  <= src1) & (src1 <=  src21hi)) |
                 ((src22lo  <= src1) & (src1 <=  src22hi)))
CR[4*BF+32] <- 0b0
CR[4*BF+33] <- in_range
CR[4*BF+34] <- 0b0
CR[4*BF+35] <- 0b0
```

Special Registers Altered:

```
CR field BF
```

J.14 Compare Equal Byte

X-Form

- cmpeqb BF,RA,RB

Pseudo-code:

```
src1 <- GPR[RA]
src1 <- src1[XLEN-8:XLEN-1]
match <- 0b0
for i = 0 to ((XLEN/8)-1)
    match <- (match | (src1 = (RB)[8*i:8*i+7]))
CR[4*BF+32] <- 0b0
CR[4*BF+33] <- match
CR[4*BF+34] <- 0b0
CR[4*BF+35] <- 0b0
```

Special Registers Altered:

```
CR field BF
```

Condition Register pseudocode

J.15 Condition Register AND

XL-Form

- crand BT,BA,BB

Pseudo-code:

```
CR[BT+32] <-  CR[BA+32] & CR[BB+32]
```

Special Registers Altered:

```
CR[BT+32]
```

J.16 Condition Register NAND

XL-Form

- crnand BT,BA,BB

Pseudo-code:

```
CR[BT+32] <- ¬(CR[BA+32] & CR[BB+32])
```

Special Registers Altered:

```
CR[BT+32]
```

J.17 Condition Register OR

XL-Form

- cror BT,BA,BB

Pseudo-code:

```
CR[BT+32] <-  CR[BA+32] | CR[BB+32]
```

Special Registers Altered:

```
CR[BT+32]
```

J.18 Condition Register XOR

XL-Form

- crxor BT,BA,BB

Pseudo-code:

```
CR[BT+32] <- CR[BA+32] ^ CR[BB+32]
```

Special Registers Altered:

```
CR[BT+32]
```

J.19 Condition Register NOR

XL-Form

- crnor BT,BA,BB

Pseudo-code:

```
CR[BT+32] <- ¬(CR[BA+32] | CR[BB+32])
```

Special Registers Altered:

```
CR[BT+32]
```

J.20 Condition Register Equivalent

XL-Form

- creqv BT,BA,BB

Pseudo-code:

```
CR[BT+32] <- ¬(CR[BA+32] ^ CR[BB+32])
```

Special Registers Altered:

```
CR[BT+32]
```

J.21 Condition Register AND with Complement

XL-Form

- crandc BT,BA,BB

Pseudo-code:

```
CR[BT+32] <- CR[BA+32] &  ¬CR[BB+32]
```

Special Registers Altered:

```
CR[BT+32]
```

J.22 Condition Register OR with Complement

XL-Form

- crorc BT,BA,BB

Pseudo-code:

```
CR[BT+32] <- CR[BA+32] |  ¬CR[BB+32]
```

Special Registers Altered:

```
CR[BT+32]
```

J.23 Move Condition Register Field

XL-Form

- mcrf BF,BFA

Pseudo-code:

```
CR[4*BF+32:4*BF+35] <- CR[4*BFA+32:4*BFA+35]
```

Special Registers Altered:

```
CR field BF
```

Fixed Point Arithmetic pseudocode

J.24 Add Immediate

D-Form

- addi RT,RA,SI

Pseudo-code:

```
RT <- (RA|0) + EXTS(SI)
```

Special Registers Altered:

```
None
```

J.25 Add Immediate Shifted

D-Form

- addis RT,RA,SI

Pseudo-code:

```
RT <- (RA|0) + EXTS(SI || [0]*16)
```

Special Registers Altered:

```
None
```

J.26 Add PC Immediate Shifted

DX-Form

- addpcis RT,D

Pseudo-code:

```
D <- d0||d1||d2
RT <- NIA + EXTS(D || [0]*16)
```

Special Registers Altered:

```
None
```

J.27 Add

XO-Form

- add RT,RA,RB (OE=0 Rc=0)
- add. RT,RA,RB (OE=0 Rc=1)
- addo RT,RA,RB (OE=1 Rc=0)
- addo. RT,RA,RB (OE=1 Rc=1)

Pseudo-code:

```
RT <- (RA) + (RB)
```

Special Registers Altered:

```
CR0                      (if Rc=1)
SO OV OV32               (if OE=1)
```

J.28 Subtract From

XO-Form

- subf RT,RA,RB (OE=0 Rc=0)
- subf. RT,RA,RB (OE=0 Rc=1)
- subfo RT,RA,RB (OE=1 Rc=0)
- subfo. RT,RA,RB (OE=1 Rc=1)

Pseudo-code:

```
RT <- ¬(RA) + (RB) + 1
```

Special Registers Altered:

```
CR0                      (if Rc=1)
SO OV OV32               (if OE=1)
```

J.29 Add Immediate Carrying

D-Form

- addic RT,RA,SI

Pseudo-code:

```
RT <- (RA) + EXTS(SI)
```

Special Registers Altered:

```
CA CA32
```

J.30 Add Immediate Carrying and Record

D-Form

- addic. RT,RA,SI

Pseudo-code:

```
RT <- (RA) + EXTS(SI)
```

Special Registers Altered:

```
CR0 CA CA32
```

J.31 Subtract From Immediate Carrying

D-Form

- subfic RT,RA,SI

Pseudo-code:

```
RT <- ¬(RA) + EXTS(SI) + 1
```

Special Registers Altered:

```
CA CA32
```

J.32 Add Carrying

XO-Form

- addc RT,RA,RB (OE=0 Rc=0)
- addc. RT,RA,RB (OE=0 Rc=1)
- addco RT,RA,RB (OE=1 Rc=0)
- addco. RT,RA,RB (OE=1 Rc=1)

Pseudo-code:

```
RT <- (RA) + (RB)
```

Special Registers Altered:

```
CA CA32
CR0                      (if Rc=1)
SO OV OV32              (if OE=1)
```

J.33 Subtract From Carrying

XO-Form

- subfc RT,RA,RB (OE=0 Rc=0)
- subfc. RT,RA,RB (OE=0 Rc=1)
- subfco RT,RA,RB (OE=1 Rc=0)
- subfco. RT,RA,RB (OE=1 Rc=1)

Pseudo-code:

```
RT <- ¬(RA) + (RB) + 1
```

Special Registers Altered:

```
CA CA32
CR0                      (if Rc=1)
SO OV OV32              (if OE=1)
```

J.34 Add Extended

XO-Form

- adde RT,RA,RB (OE=0 Rc=0)
- adde. RT,RA,RB (OE=0 Rc=1)
- addeo RT,RA,RB (OE=1 Rc=0)
- addeo. RT,RA,RB (OE=1 Rc=1)

Pseudo-code:

```
RT <- (RA) + (RB) + CA
```

Special Registers Altered:

```
CA CA32
CR0                      (if Rc=1)
SO OV OV32               (if OE=1)
```

J.35 Subtract From Extended

XO-Form

- subfe RT,RA,RB (OE=0 Rc=0)
- subfe. RT,RA,RB (OE=0 Rc=1)
- subfeo RT,RA,RB (OE=1 Rc=0)
- subfeo. RT,RA,RB (OE=1 Rc=1)

Pseudo-code:

```
RT <- ¬(RA) + (RB) + CA
```

Special Registers Altered:

```
CA CA32
CR0                      (if Rc=1)
SO OV OV32               (if OE=1)
```

J.36 Add to Minus One Extended

XO-Form

- addme RT,RA (OE=0 Rc=0)
- addme. RT,RA (OE=0 Rc=1)
- addmeo RT,RA (OE=1 Rc=0)
- addmeo. RT,RA (OE=1 Rc=1)

Pseudo-code:

```
RT <- (RA) + CA - 1
```

Special Registers Altered:

```
CA CA32
CR0                      (if Rc=1)
SO OV OV32               (if OE=1)
```

J.37 Subtract From Minus One Extended

XO-Form

- subfme RT,RA (OE=0 Rc=0)
- subfme. RT,RA (OE=0 Rc=1)
- subfmeo RT,RA (OE=1 Rc=0)
- subfmeo. RT,RA (OE=1 Rc=1)

Pseudo-code:

```
RT <- ¬(RA) + CA - 1
```

Special Registers Altered:

```
CA CA32
CR0                      (if Rc=1)
SO OV OV32              (if OE=1)
```

J.38 Add Extended using alternate carry bit

Z23-Form

- addex RT,RA,RB,CY

Pseudo-code:

```
if CY=0 then RT <- (RA) + (RB) + OV
```

Special Registers Altered:

```
OV OV32                 (if CY=0 )
```

J.39 Subtract From Zero Extended

XO-Form

- subfze RT,RA (OE=0 Rc=0)
- subfze. RT,RA (OE=0 Rc=1)
- subfzeo RT,RA (OE=1 Rc=0)
- subfzeo. RT,RA (OE=1 Rc=1)

Pseudo-code:

```
RT <- ¬(RA) + CA
```

Special Registers Altered:

```
CA CA32
CR0                      (if Rc=1)
SO OV OV32              (if OE=1)
```

J.40 Add to Zero Extended

XO-Form

- addze RT,RA (OE=0 Rc=0)
- addze. RT,RA (OE=0 Rc=1)

- addzeo RT,RA (OE=1 Rc=0)
- addzeo. RT,RA (OE=1 Rc=1)

Pseudo-code:

```
RT <- (RA) + CA
```

Special Registers Altered:

```
CA CA32
CR0                      (if Rc=1)
SO OV OV32              (if OE=1)
```

J.41 Negate

XO-Form

- neg RT,RA (OE=0 Rc=0)
- neg. RT,RA (OE=0 Rc=1)
- nego RT,RA (OE=1 Rc=0)
- nego. RT,RA (OE=1 Rc=1)

Pseudo-code:

```
RT <- ¬(RA) + 1
```

Special Registers Altered:

```
CR0                      (if Rc=1)
SO OV OV32              (if OE=1)
```

J.42 Multiply Low Immediate

D-Form

- mulli RT,RA,SI

Pseudo-code:

```
prod[0:(XLEN*2)-1] <- MULS((RA), EXTS(SI))
RT <- prod[XLEN:(XLEN*2)-1]
```

Special Registers Altered:

```
None
```

J.43 Multiply High Word

XO-Form

- mulhw RT,RA,RB (Rc=0)
- mulhw. RT,RA,RB (Rc=1)

Pseudo-code:

```
prod[0:XLEN-1] <- MULS((RA)[XLEN/2:XLEN-1], (RB)[XLEN/2:XLEN-1])
RT[XLEN/2:XLEN-1] <- prod[0:(XLEN/2)-1]
RT[0:(XLEN/2)-1] <- undefined(prod[0:(XLEN/2)-1])
```

Special Registers Altered:

```
CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
```

J.44 Multiply Low Word

XO-Form

- mullw RT,RA,RB (OE=0 Rc=0)
- mullw. RT,RA,RB (OE=0 Rc=1)
- mullwo RT,RA,RB (OE=1 Rc=0)
- mullwo. RT,RA,RB (OE=1 Rc=1)

Pseudo-code:

```
prod[0:XLEN-1] <- MULS((RA)[XLEN/2:XLEN-1], (RB)[XLEN/2:XLEN-1])
RT <- prod
overflow <- ((prod[0:XLEN/2] != [0]*((XLEN/2)+1)) &
             (prod[0:XLEN/2] != [1]*((XLEN/2)+1)))
```

Special Registers Altered:

```
CR0                     (if Rc=1)
SO OV OV32             (if OE=1)
```

J.45 Multiply High Word Unsigned

XO-Form

- mulhwu RT,RA,RB (Rc=0)
- mulhwu. RT,RA,RB (Rc=1)

Pseudo-code:

```
prod[0:XLEN-1] <- (RA)[XLEN/2:XLEN-1] * (RB)[XLEN/2:XLEN-1]
RT[XLEN/2:XLEN-1] <- prod[0:(XLEN/2)-1]
RT[0:(XLEN/2)-1] <- undefined(prod[0:(XLEN/2)-1])
```

Special Registers Altered:

```
CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
```

J.46 Divide Word

XO-Form

- divw RT,RA,RB (OE=0 Rc=0)
- divw. RT,RA,RB (OE=0 Rc=1)
- divwo RT,RA,RB (OE=1 Rc=0)
- divwo. RT,RA,RB (OE=1 Rc=1)

Pseudo-code:

```
dividend[0:(XLEN/2)-1] <- (RA)[XLEN/2:XLEN-1]
divisor[0:(XLEN/2)-1] <- (RB) [XLEN/2:XLEN-1]
if (((dividend = (0b1 || ([0b0] * ((XLEN/2)-1)))) &
     (divisor = [1]*(XLEN/2))) |
```

```
     (divisor = [0]*(XLEN/2))) then
    RT[0:XLEN-1] <- undefined([0]*XLEN)
    overflow <- 1
else
    RT[XLEN/2:XLEN-1] <- DIVS(dividend, divisor)
    RT[0:(XLEN/2)-1] <- undefined([0]*(XLEN/2))
    overflow <- 0
```

Special Registers Altered:

```
CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
SO OV OV32                              (if OE=1)
```

J.47 Divide Word Unsigned

XO-Form

- divwu RT,RA,RB (OE=0 Rc=0)
- divwu. RT,RA,RB (OE=0 Rc=1)
- divwuo RT,RA,RB (OE=1 Rc=0)
- divwuo. RT,RA,RB (OE=1 Rc=1)

Pseudo-code:

```
dividend[0:(XLEN/2)-1] <- (RA)[XLEN/2:XLEN-1]
divisor[0:(XLEN/2)-1] <- (RB)[XLEN/2:XLEN-1]
if divisor != 0 then
    RT[XLEN/2:XLEN-1] <- dividend / divisor
    RT[0:(XLEN/2)-1] <- undefined([0]*(XLEN/2))
    overflow <- 0
else
    RT[0:XLEN-1] <- undefined([0]*XLEN)
    overflow <- 1
```

Special Registers Altered:

```
CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
SO OV OV32                              (if OE=1)
```

J.48 Divide Word Extended

XO-Form

- divwe RT,RA,RB (OE=0 Rc=0)
- divwe. RT,RA,RB (OE=0 Rc=1)
- divweo RT,RA,RB (OE=1 Rc=0)
- divweo. RT,RA,RB (OE=1 Rc=1)

Pseudo-code:

```
dividend[0:XLEN-1] <- (RA)[XLEN/2:XLEN-1] || [0]*(XLEN/2)
divisor[0:XLEN-1] <- EXTS64((RB)[XLEN/2:XLEN-1])
if (((dividend = (0b1 || ([0b0] * (XLEN-1)))) &
     (divisor = [1]*XLEN)) |
     (divisor = [0]*XLEN)) then
    overflow <- 1
else
```

```
        result <- DIVS(dividend, divisor)
        result_half[0:XLEN-1] <- EXTS64(result[XLEN/2:XLEN-1])
        if (result_half = result) then
            RT[XLEN/2:XLEN-1] <- result[XLEN/2:XLEN-1]
            RT[0:(XLEN/2)-1] <- undefined([0]*(XLEN/2))
            overflow <- 0
        else
            overflow <- 1
    if overflow = 1 then
        RT[0:XLEN-1] <- undefined([0]*XLEN)
```

Special Registers Altered:

```
    CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
    SO OV OV32                              (if OE=1)
```

J.49 Divide Word Extended Unsigned

XO-Form

- divweu RT,RA,RB (OE=0 Rc=0)
- divweu. RT,RA,RB (OE=0 Rc=1)
- divweuo RT,RA,RB (OE=1 Rc=0)
- divweuo. RT,RA,RB (OE=1 Rc=1)

Pseudo-code:

```
    dividend[0:XLEN-1] <- (RA)[XLEN/2:XLEN-1] || [0]*(XLEN/2)
    divisor[0:XLEN-1] <- [0]*(XLEN/2) || (RB)[XLEN/2:XLEN-1]
    if (divisor = [0]*XLEN) then
        overflow <- 1
    else
        result <- dividend / divisor
        if RA[XLEN/2:XLEN-1] <u RB[XLEN/2:XLEN-1] then
            RT[XLEN/2:XLEN-1] <- result[XLEN/2:XLEN-1]
            RT[0:(XLEN/2)-1] <- undefined([0]*(XLEN/2))
            overflow <- 0
        else
            overflow <- 1
    if overflow = 1 then
        RT[0:XLEN-1] <- undefined([0]*XLEN)
```

Special Registers Altered:

```
    CR0 (bits 0:2 undefined in 64-bit mode) (if Rc=1)
    SO OV OV32                              (if OE=1)
```

J.50 Modulo Signed Word

X-Form

- modsw RT,RA,RB

Pseudo-code:

```
    dividend[0:(XLEN/2)-1] <- (RA)[XLEN/2:XLEN-1]
    divisor[0:(XLEN/2)-1] <- (RB)[XLEN/2:XLEN-1]
```

```
if (((dividend = (0b1 || ([0b0] * ((XLEN/2)-1)))) &
     (divisor = [1]*(XLEN/2))) |
     (divisor = [0]*(XLEN/2))) then
    RT[0:XLEN-1] <- undefined([0]*XLEN)
    overflow <- 1
else
    RT[0:XLEN-1] <- EXTS64(MODS(dividend, divisor))
    RT[0:(XLEN/2)-1] <- undefined(RT[0:(XLEN/2)-1])
    overflow <- 0
```

Special Registers Altered:

```
None
```

J.51 Modulo Unsigned Word

X-Form

- moduw RT,RA,RB

Pseudo-code:

```
dividend[0:(XLEN/2)-1] <- (RA)[XLEN/2:63]
divisor [0:(XLEN/2)-1] <- (RB)[XLEN/2:63]
if divisor = [0]*(XLEN/2) then
    RT[0:XLEN-1] <- undefined([0]*64)
    overflow <- 1
else
    RT[XLEN/2:XLEN-1] <- dividend % divisor
    RT[0:(XLEN/2)-1] <- undefined([0]*(XLEN/2))
    overflow <- 0
```

Special Registers Altered:

```
None
```

J.52 Deliver A Random Number

X-Form

- darn RT,L3

Pseudo-code:

```
RT <- random(L3)
```

Special Registers Altered:

```
none
```

J.53 Multiply Low Doubleword

XO-Form

- mulld RT,RA,RB (OE=0 Rc=0)
- mulld. RT,RA,RB (OE=0 Rc=1)
- mulldo RT,RA,RB (OE=1 Rc=0)

- mulldo. RT,RA,RB (OE=1 Rc=1)

Pseudo-code:

```
prod[0:(XLEN*2)-1] <- MULS((RA), (RB))
RT <- prod[XLEN:(XLEN*2)-1]
overflow <- ((prod[0:XLEN] != [0]*(XLEN+1)) &
             (prod[0:XLEN] != [1]*(XLEN+1)))
```

Special Registers Altered:

```
CR0                     (if Rc=1)
SO OV OV32             (if OE=1)
```

J.54 Multiply High Doubleword

XO-Form

- mulhd RT,RA,RB (Rc=0)
- mulhd. RT,RA,RB (Rc=1)

Pseudo-code:

```
prod[0:(XLEN*2)-1] <- MULS((RA), (RB))
RT <- prod[0:XLEN-1]
```

Special Registers Altered:

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

J.55 Multiply High Doubleword Unsigned

XO-Form

- mulhdu RT,RA,RB (Rc=0)
- mulhdu. RT,RA,RB (Rc=1)

Pseudo-code:

```
prod[0:(XLEN*2)-1] <- (RA) * (RB)
RT <- prod[0:XLEN-1]
```

Special Registers Altered:

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

J.56 Multiply-Add High Doubleword VA-Form

VA-Form

- maddhd RT,RA,RB,RC

Pseudo-code:

```
prod[0:(XLEN*2)-1] <- MULS((RA), (RB))
sum[0:(XLEN*2)-1] <- prod + EXTS(RC)[0:XLEN*2]
RT <- sum[0:XLEN-1]
```

Special Registers Altered:

```
None
```

J.57 Multiply-Add High Doubleword Unsigned

VA-Form

- maddhdu RT,RA,RB,RC

Pseudo-code:

```
prod[0:(XLEN*2)-1] <- (RA) * (RB)
sum[0:(XLEN*2)-1] <- prod + EXTZ(RC)
RT <- sum[0:XLEN-1]
```

Special Registers Altered:

```
None
```

J.58 Multiply-Add Low Doubleword

VA-Form

- maddld RT,RA,RB,RC

Pseudo-code:

```
prod[0:(XLEN*2)-1] <- MULS((RA), (RB))
sum[0:(XLEN*2)-1] <- prod + EXTS(RC)
RT <- sum[XLEN:(XLEN*2)-1]
```

Special Registers Altered:

```
None
```

J.59 Divide Doubleword

XO-Form

- divd RT,RA,RB (OE=0 Rc=0)
- divd. RT,RA,RB (OE=0 Rc=1)
- divdo RT,RA,RB (OE=1 Rc=0)
- divdo. RT,RA,RB (OE=1 Rc=1)

Pseudo-code:

```
dividend[0:XLEN-1] <- (RA)
divisor[0:XLEN-1] <- (RB)
if (((dividend = (0b1 || ([0b0] * (XLEN-1)))) &
     (divisor = [1]*XLEN)) |
     (divisor = [0]*XLEN)) then
    RT[0:XLEN-1] <- undefined([0]*XLEN)
    overflow <- 1
else
    RT <- DIVS(dividend, divisor)
    overflow <- 0
```

Special Registers Altered:

```
CR0                     (if Rc=1)
SO OV OV32              (if OE=1)
```

J.60 Divide Doubleword Unsigned

XO-Form

- divdu RT,RA,RB (OE=0 Rc=0)
- divdu. RT,RA,RB (OE=0 Rc=1)
- divduo RT,RA,RB (OE=1 Rc=0)
- divduo. RT,RA,RB (OE=1 Rc=1)

Pseudo-code:

```
dividend[0:XLEN-1] <- (RA)
divisor[0:XLEN-1] <- (RB)
if (divisor = [0]*XLEN) then
    RT[0:XLEN-1] <- undefined([0]*XLEN)
    overflow <- 1
else
    RT <- dividend / divisor
    overflow <- 0
```

Special Registers Altered:

```
CR0                     (if Rc=1)
SO OV OV32              (if OE=1)
```

J.61 Divide Doubleword Extended

XO-Form

- divde RT,RA,RB (OE=0 Rc=0)
- divde. RT,RA,RB (OE=0 Rc=1)
- divdeo RT,RA,RB (OE=1 Rc=0)
- divdeo. RT,RA,RB (OE=1 Rc=1)

Pseudo-code:

```
dividend[0:(XLEN*2)-1] <- (RA) || [0]*XLEN
divisor[0:(XLEN*2)-1] <- EXTS128((RB))
if (((dividend = (0b1 || ([0b0] * ((XLEN*2)-1)))) &
     (divisor = [1]*(XLEN*2))) |
     (divisor = [0]*(XLEN*2))) then
    overflow <- 1
else
    result <- DIVS(dividend, divisor)
    result_half[0:(XLEN*2)-1] <- EXTS128(result[XLEN:(XLEN*2)-1])
    if (result_half = result) then
        RT <- result[XLEN:(XLEN*2)-1]
        overflow <- 0
    else
        overflow <- 1
if overflow = 1 then
    RT[0:XLEN-1] <- undefined([0]*XLEN)
```

Special Registers Altered:

```
CR0                     (if Rc=1)
SO OV OV32              (if OE=1)
```

J.62 Divide Doubleword Extended Unsigned

XO-Form

- divdeu RT,RA,RB (OE=0 Rc=0)
- divdeu. RT,RA,RB (OE=0 Rc=1)
- divdeuo RT,RA,RB (OE=1 Rc=0)
- divdeuo. RT,RA,RB (OE=1 Rc=1)

Pseudo-code:

```
dividend[0:(XLEN*2)-1] <- (RA) || [0]*XLEN
divisor[0:(XLEN*2)-1] <- [0]*XLEN || (RB)
if divisor = [0]*(XLEN*2) then
    overflow <- 1
else
    result <- dividend / divisor
    if (RA) <u (RB) then
        RT <- result[XLEN:(XLEN*2)-1]
        overflow <- 0
    else
        overflow <- 1
if overflow = 1 then
    RT[0:XLEN-1] <- undefined([0]*XLEN)
```

Special Registers Altered:

```
CR0                     (if Rc=1)
SO OV OV32              (if OE=1)
```

J.63 Modulo Signed Doubleword

X-Form

- modsd RT,RA,RB

Pseudo-code:

```
dividend <- (RA)
divisor <- (RB)
if (((dividend = (0b1 || ([0b0] * (XLEN-1)))) &
     (divisor = [1]*XLEN)) |
     (divisor = [0]*XLEN)) then
    RT[0:63] <- undefined([0]*XLEN)
    overflow <- 1
else
    RT <- MODS(dividend, divisor)
    overflow <- 0
```

Special Registers Altered:

```
None
```

J.64 Modulo Unsigned Doubleword

X-Form

- modud RT,RA,RB

Pseudo-code:

```
dividend <- (RA)
divisor <- (RB)
if (divisor = [0]*XLEN) then
    RT[0:XLEN-1] <- undefined([0]*XLEN)
    overflow <- 1
else
    RT <- dividend % divisor
    overflow <- 0
```

Special Registers Altered:

```
None
```

Fixed Point Load pseudocode

J.65 Load Byte and Zero

D-Form

- lbz RT,D(RA)

Pseudo-code:

```
b <- (RA|0)
EA <- b + EXTS(D)
RT <- ([0] * (XLEN-8)) || MEM(EA, 1)
```

Special Registers Altered:

```
None
```

J.66 Load Byte and Zero Indexed

X-Form

- lbzx RT,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
RT <- ([0] * (XLEN-8)) || MEM(EA, 1)
```

Special Registers Altered:

```
None
```

J.67 Load Byte and Zero with Update

D-Form

- lbzu RT,D(RA)

Pseudo-code:

```
EA <- (RA) + EXTS(D)
RT <- ([0] * (XLEN-8)) || MEM(EA, 1)
RA <- EA
```

Special Registers Altered:

```
None
```

J.68 Load Byte and Zero with Update Indexed

X-Form

- lbzux RT,RA,RB

Pseudo-code:

```
EA <- (RA) + (RB)
RT <- ([0] * (XLEN-8)) || MEM(EA, 1)
RA <- EA
```

Special Registers Altered:

```
None
```

J.69 Load Halfword and Zero

D-Form

- lhz RT,D(RA)

Pseudo-code:

```
b <- (RA|0)
EA <- b + EXTS(D)
RT <- ([0] * (XLEN-16)) || MEM(EA, 2)
```

Special Registers Altered:

```
None
```

J.70 Load Halfword and Zero Indexed

X-Form

- lhzx RT,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
RT <- ([0] * (XLEN-16)) || MEM(EA, 2)
```

Special Registers Altered:

```
None
```

J.71 Load Halfword and Zero with Update

D-Form

- lhzu RT,D(RA)

Pseudo-code:

```
EA <- (RA) + EXTS(D)
RT <- ([0] * (XLEN-16)) || MEM(EA, 2)
RA <- EA
```

Special Registers Altered:

```
None
```

J.72 Load Halfword and Zero with Update Indexed

X-Form

- lhzux RT,RA,RB

Pseudo-code:

```
EA <- (RA) + (RB)
RT <- ([0] * (XLEN-16)) || MEM(EA, 2)
RA <- EA
```

Special Registers Altered:

```
None
```

J.73 Load Halfword Algebraic

D-Form

- lha RT,D(RA)

Pseudo-code:

```
b <- (RA|0)
EA <- b + EXTS(D)
RT <- EXTS(MEM(EA, 2))
```

Special Registers Altered:

```
None
```

J.74 Load Halfword Algebraic Indexed

X-Form

- lhax RT,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
RT <- EXTS(MEM(EA, 2))
```

Special Registers Altered:

```
None
```

J.75 Load Halfword Algebraic with Update

D-Form

- lhau RT,D(RA)

Pseudo-code:

```
EA <- (RA) + EXTS(D)
RT <- EXTS(MEM(EA, 2))
RA <- EA
```

Special Registers Altered:

```
None
```

J.76 Load Halfword Algebraic with Update Indexed

X-Form

- lhaux RT,RA,RB

Pseudo-code:

```
EA <- (RA) + (RB)
RT <- EXTS(MEM(EA, 2))
RA <- EA
```

Special Registers Altered:

```
None
```

J.77 Load Word and Zero

D-Form

- lwz RT,D(RA)

Pseudo-code:

```
b <- (RA|0)
EA <- b + EXTS(D)
RT <- [0] * 32 || MEM(EA, 4)
```

Special Registers Altered:

```
None
```

J.78 Load Word and Zero Indexed

X-Form

- lwzx RT,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
RT <- [0] * 32 || MEM(EA, 4)
```

Special Registers Altered:

```
None
```

J.79 Load Word and Zero with Update

D-Form

- lwzu RT,D(RA)

Pseudo-code:

```
EA <- (RA) + EXTS(D)
RT <- [0]*32 || MEM(EA, 4)
RA <- EA
```

Special Registers Altered:

```
None
```

J.80 Load Word and Zero with Update Indexed

X-Form

- lwzux RT,RA,RB

Pseudo-code:

```
EA <- (RA) + (RB)
RT <- [0] * 32 || MEM(EA, 4)
RA <- EA
```

Special Registers Altered:

```
None
```

J.81 Load Word Algebraic

DS-Form

- lwa RT,DS(RA)

Pseudo-code:

```
b <- (RA|0)
EA <- b + EXTS(DS || 0b00)
RT <- EXTS(MEM(EA, 4))
```

Special Registers Altered:

```
None
```

J.82 Load Word Algebraic Indexed

X-Form

- lwax RT,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
RT <- EXTS(MEM(EA, 4))
```

Special Registers Altered:

```
None
```

J.83 Load Word Algebraic with Update Indexed

X-Form

- lwaux RT,RA,RB

Pseudo-code:

```
EA <- (RA) + (RB)
RT <- EXTS(MEM(EA, 4))
RA <- EA
```

Special Registers Altered:

```
None
```

J.84 Load Doubleword

DS-Form

- ld RT,DS(RA)

Pseudo-code:

```
b <- (RA|0)
EA <- b + EXTS(DS || 0b00)
RT <- MEM(EA, 8)
```

Special Registers Altered:

```
None
```

J.85 Load Doubleword Indexed

X-Form

- ldx RT,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
RT <- MEM(EA, 8)
```

Special Registers Altered:

```
None
```

J.86 Load Doubleword with Update Indexed

DS-Form

- ldu RT,DS(RA)

Pseudo-code:

```
EA <- (RA) + EXTS(DS || 0b00)
RT <- MEM(EA, 8)
RA <- EA
```

Special Registers Altered:

```
None
```

J.87 Load Doubleword with Update Indexed

X-Form

- ldux RT,RA,RB

Pseudo-code:

```
EA <- (RA) + (RB)
RT <- MEM(EA, 8)
RA <- EA
```

Special Registers Altered:

```
None
```

J.88 Load Quadword

DQ-Form

- lq RTp,DQ(RA)

Pseudo-code:

```
b <- (RA|0)
EA <- b + EXTS(DQ || 0b0000)
RTp <- MEM(EA, 16)
```

Special Registers Altered:

```
None
```

J.89 Load Halfword Byte-Reverse Indexed

X-Form

- lhbrx RT,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
load_data <- MEM(EA, 2)
RT <- [0]*48 || load_data[8:15] || load_data[0:7]
```

Special Registers Altered:

```
None
```

J.90 Load Word Byte-Reverse Indexed

X-Form

- lwbrx RT,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
load_data <- MEM(EA, 4)
RT <- ([0] * 32 || load_data[24:31] || load_data[16:23]
                || load_data[8:15]  || load_data[0:7])
```

Special Registers Altered:

```
None
```

J.91 Load Doubleword Byte-Reverse Indexed

X-Form

- ldbrx RT,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
load_data <- MEM(EA, 8)
RT <- (load_data[56:63] || load_data[48:55]
    || load_data[40:47] || load_data[32:39]
    || load_data[24:31] || load_data[16:23]
    || load_data[8:15]  || load_data[0:7])
```

Special Registers Altered:

```
None
```

J.92 Load Multiple Word

DQ-Form

- lmw RT,D(RA)

Pseudo-code:

```
b <- (RA|0)
EA <- b + EXTS(D)
r <- RT[0:63]
do while r <=  31
    GPR(r) <- [0]*32 || MEM(EA, 4)
    r <- r + 1
    EA <- EA + 4
```

Special Registers Altered:

```
None
```

Fixed Point Logical pseudocode

J.93 AND Immediate

D-Form

- andi. RA,RS,UI

Pseudo-code:

```
RA <- (RS) & EXTZ(UI)
```

Special Registers Altered:

```
CR0
```

J.94 OR Immediate

D-Form

- ori RA,RS,UI

Pseudo-code:

```
RA <- (RS) | EXTZ(UI)
```

Special Registers Altered:

```
None
```

J.95 AND Immediate Shifted

D-Form

- andis. RA,RS,UI

Pseudo-code:

```
RA <- (RS) & EXTZ(UI || [0]*16)
```

Special Registers Altered:

```
CR0
```

J.96 OR Immediate Shifted

D-Form

- oris RA,RS,UI

Pseudo-code:

```
RA <- (RS) | EXTZ(UI || [0]*16)
```

Special Registers Altered:

```
None
```

J.97 XOR Immediate Shifted

D-Form

- xoris RA,RS,UI

Pseudo-code:

```
RA <- (RS) ^ EXTZ(UI || [0]*16)
```

Special Registers Altered:

```
None
```

J.98 XOR Immediate

D-Form

- xori RA,RS,UI

Pseudo-code:

```
RA <- (RS) ^ EXTZ(UI)
```

Special Registers Altered:

```
None
```

J.99 AND

X-Form

- and RA,RS,RB (Rc=0)
- and. RA,RS,RB (Rc=1)

Pseudo-code:

```
RA <- (RS) & (RB)
```

Special Registers Altered:

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

J.100 OR

X-Form

- or RA,RS,RB (Rc=0)
- or. RA,RS,RB (Rc=1)

Pseudo-code:

```
RA <- (RS) | (RB)
```

Special Registers Altered:

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

J.101 XOR

X-Form

- xor RA,RS,RB (Rc=0)
- xor. RA,RS,RB (Rc=1)

Pseudo-code:

```
RA <- (RS) ^ (RB)
```

Special Registers Altered:

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

J.102 NAND

X-Form

- nand RA,RS,RB (Rc=0)
- nand. RA,RS,RB (Rc=1)

Pseudo-code:

```
RA <- ¬((RS) & (RB))
```

Special Registers Altered:

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

J.103 NOR

X-Form

- nor RA,RS,RB (Rc=0)
- nor. RA,RS,RB (Rc=1)

Pseudo-code:

```
RA <- ¬((RS) | (RB))
```

Special Registers Altered:

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

J.104 Equivalent

X-Form

- eqv RA,RS,RB (Rc=0)
- eqv. RA,RS,RB (Rc=1)

Pseudo-code:

```
RA <- ¬((RS) ^ (RB))
```

Special Registers Altered:

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

J.105 AND with Complement

X-Form

- andc RA,RS,RB (Rc=0)
- andc. RA,RS,RB (Rc=1)

Pseudo-code:

```
RA <- (RS) &  ¬(RB)
```

Special Registers Altered:

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

J.106 OR with Complement

X-Form

- orc RA,RS,RB (Rc=0)
- orc. RA,RS,RB (Rc=1)

Pseudo-code:

```
RA <- (RS) |  ¬(RB)
```

Special Registers Altered:

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

J.107 Extend Sign Byte

X-Form

- extsb RA,RS (Rc=0)
- extsb. RA,RS (Rc=1)

Pseudo-code:

```
RA <- EXTSXL(RS, XLEN/8)
```

Special Registers Altered:

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

J.108 Extend Sign Halfword

X-Form

- extsh RA,RS (Rc=0)
- extsh. RA,RS (Rc=1)

Pseudo-code:

```
RA <- EXTSXL(RS, XLEN/4)
```

Special Registers Altered:

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

J.109 Count Leading Zeros Word

X-Form

- cntlzw RA,RS (Rc=0)
- cntlzw. RA,RS (Rc=1)

Pseudo-code:

```
n <- (XLEN/2)
do while n < XLEN
   if (RS)[n] = 1 then
       leave
   n <- n + 1
RA <- n - (XLEN/2)
```

Special Registers Altered:

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

J.110 Count Trailing Zeros Word

X-Form

- cnttzw RA,RS (Rc=0)
- cnttzw. RA,RS (Rc=1)

Pseudo-code:

```
n <- 0
do while n < XLEN/2
   if (RS)[XLEN-1-n] = 0b1 then
        leave
   n  <- n + 1
RA <- EXTZ(n)
```

Special Registers Altered:

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

J.111 Compare Bytes

X-Form

- cmpb RA,RS,RB

Pseudo-code:

```
do n = 0 to ((XLEN/8)-1)
    if RS[8*n:8* n+7] = (RB)[8*n:8*n+7] then
       RA[8*n:8* n+7] <- [1]*8
    else
       RA[8*n:8* n+7] <- [0]*8
```

Special Registers Altered:

```
None
```

J.112 Population Count Bytes

X-Form

- popcntb RA,RS

Pseudo-code:

```
do i = 0 to ((XLEN/8)-1)
   n <-  0
   do j = 0 to 7
      if (RS)[(i*8)+j] = 1 then
          n <- n+1
   RA[(i*8):(i*8)+7] <-  n
```

Special Registers Altered:

```
None
```

J.113 Population Count Words

X-Form

- popcntw RA,RS

Pseudo-code:

```
e <- (XLEN/2)-1
do i = 0 to 1
   s <- i*XLEN/2
   n <-  0
   do j = 0 to e
      if (RS)[s+j] = 1 then
          n <- n+1
   RA[s:s+e] <- n
```

Special Registers Altered:

```
None
```

J.114 Parity Doubleword

X-Form

- prtyd RA,RS

Pseudo-code:

```
s <- 0
do i = 0 to ((XLEN/8)-1)
    s <- s ^ (RS)[i*8+7]
RA <- [0] * (XLEN-1) || s
```

Special Registers Altered:

```
None
```

J.115 Parity Word

X-Form

- prtyw RA,RS

Pseudo-code:

```
s <- 0
t <- 0
do i = 0 to ((XLEN/8/2)-1)
    s <-  s ^ (RS)[i*8+7]
do i = 4 to ((XLEN/8)-1)
    t <-  t ^ (RS)[i*8+7]
RA[0:(XLEN/2)-1] <- [0]*((XLEN/2)-1) || s
RA[XLEN/2:XLEN-1] <- [0]*((XLEN/2)-1) || t
```

Special Registers Altered:

```
None
```

J.116 Extend Sign Word

X-Form

- extsw RA,RS (Rc=0)
- extsw. RA,RS (Rc=1)

Pseudo-code:

```
RA <- EXTSXL(RS, XLEN/2)
```

Special Registers Altered:

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

J.117 Population Count Doubleword

X-Form

- popcntd RA,RS

Pseudo-code:

```
n <- 0
do i = 0 to (XLEN-1)
   if (RS)[i] = 1 then
       n <-  n+1
RA <- n
```

Special Registers Altered:

```
None
```

J.118 Count Leading Zeros Doubleword

X-Form

- cntlzd RA,RS (Rc=0)
- cntlzd. RA,RS (Rc=1)

Pseudo-code:

```
n <- 0
do while n < XLEN
  if (RS)[n]  = 1 then
     leave
  n <- n + 1
RA <- n
```

Special Registers Altered:

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

J.119 Count Trailing Zeros Doubleword

X-Form

- cnttzd RA,RS (Rc=0)
- cnttzd. RA,RS (Rc=1)

Pseudo-code:

```
n  <- 0
do while n < XLEN
   if (RS)[XLEN-1-n] = 0b1 then
        leave
   n  <- n + 1
RA <- EXTZ(n)
```

Special Registers Altered:

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

J.120 Bit Permute Doubleword

X-Form

- bpermd RA,RS,RB

Pseudo-code:

```
perm <- [0] * (XLEN/8)
for i = 0 to ((XLEN/8)-1)
   index <- (RS)[8*i:8*i+7]
   if index <u XLEN then
        perm[i] <- (RB)[index]
   else
        perm[i] <- 0
RA <- [0]*(XLEN*7/8) || perm
```

Special Registers Altered:

```
None
```

Fixed Point Rotate pseudocode

J.121 Rotate Left Word Immediate then AND with Mask

M-Form

- rlwinm RA,RS,SH,MB,ME (Rc=0)
- rlwinm. RA,RS,SH,MB,ME (Rc=1)

Pseudo-code:

```
n <- SH
r <- ROTL32((RS)[XLEN/2:XLEN-1], n)
m <- MASK32(MB, ME)
RA <- r & m
```

Special Registers Altered:

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

J.122 Rotate Left Word then AND with Mask

M-Form

- rlwnm RA,RS,RB,MB,ME (Rc=0)
- rlwnm. RA,RS,RB,MB,ME (Rc=1)

Pseudo-code:

```
n <- (RB)[XLEN-5:XLEN-1]
r <- ROTL32((RS)[XLEN/2:XLEN-1], n)
m <- MASK32(MB, ME)
RA <- r & m
```

Special Registers Altered:

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

J.123 Rotate Left Word Immediate then Mask Insert

M-Form

- rlwimi RA,RS,SH,MB,ME (Rc=0)
- rlwimi. RA,RS,SH,MB,ME (Rc=1)

Pseudo-code:

```
n <- SH
r <- ROTL32((RS)[XLEN/2:XLEN-1], n)
m <- MASK32(MB, ME)
RA <- r&m | (RA) & ¬m
```

Special Registers Altered:

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

J.124 Rotate Left Doubleword Immediate then Clear Left

MD-Form

- rldicl RA,RS,sh,mb (Rc=0)
- rldicl. RA,RS,sh,mb (Rc=1)

Pseudo-code:

```
n <- sh
r <- ROTL64((RS), n)
b <- mb[5] || mb[0:4]
m <- MASK(b, (XLEN-1))
RA <- r & m
```

Special Registers Altered:

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

J.125 Rotate Left Doubleword Immediate then Clear Right

MD-Form

- rldicr RA,RS,sh,me (Rc=0)
- rldicr. RA,RS,sh,me (Rc=1)

Pseudo-code:

```
n <- sh
r <- ROTL64((RS), n)
e <- me[5] || me[0:4]
m <- MASK(0, e)
RA <- r & m
```

Special Registers Altered:

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

J.126 Rotate Left Doubleword Immediate then Clear

MD-Form

- rldic RA,RS,sh,mb (Rc=0)
- rldic. RA,RS,sh,mb (Rc=1)

Pseudo-code:

```
n <- sh
r <- ROTL64((RS), n)
b <- mb[5] || mb[0:4]
m <- MASK(b, ¬n)
RA <- r & m
```

Special Registers Altered:

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

J.127 Rotate Left Doubleword then Clear Left

MDS-Form

- rldcl RA,RS,RB,mb (Rc=0)
- rldcl. RA,RS,RB,mb (Rc=1)

Pseudo-code:

```
n <- (RB)[XLEN-5:XLEN-1]
r <- ROTL64((RS), n)
b <- mb[5] || mb[0:4]
m <- MASK(b, (XLEN-1))
RA <- r & m
```

Special Registers Altered:

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

J.128 Rotate Left Doubleword then Clear Right

MDS-Form

- rldcr RA,RS,RB,me (Rc=0)
- rldcr. RA,RS,RB,me (Rc=1)

Pseudo-code:

```
n <- (RB)[XLEN-5:XLEN-1]
r <- ROTL64((RS), n)
e <- me[5] || me[0:4]
m <- MASK(0, e)
RA <- r & m
```

Special Registers Altered:

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

J.129 Rotate Left Doubleword Immediate then Mask Insert

MD-Form

- rldimi RA,RS,sh,mb (Rc=0)
- rldimi. RA,RS,sh,mb (Rc=1)

Pseudo-code:

```
n <- sh
r <- ROTL64((RS), n)
b <- mb[5] || mb[0:4]
m <- MASK(b,  ¬n)
RA <- r&m | (RA)& ¬m
```

Special Registers Altered:

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

J.130 Shift Left Word

X-Form

- slw RA,RS,RB (Rc=0)
- slw. RA,RS,RB (Rc=1)

Pseudo-code:

```
n <- (RB)[XLEN-5:XLEN-1]
r <- ROTL32((RS)[XLEN/2:XLEN-1], n)
if (RB)[XLEN-6] = 0 then
     m <- MASK32(0, ((XLEN/2)-1-n))
else m <- [0]*XLEN
RA <- r & m
```

Special Registers Altered:

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

J.131 Shift Right Word

X-Form

- srw RA,RS,RB (Rc=0)
- srw. RA,RS,RB (Rc=1)

Pseudo-code:

```
n <- (RB)[XLEN-5:XLEN-1]
r <- ROTL32((RS)[XLEN/2:XLEN-1], XLEN-n)
if (RB)[XLEN-6] = 0 then
    m <- MASK32(n, ((XLEN/2)-1))
else m <- [0]*XLEN
RA <- r & m
```

Special Registers Altered:

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

J.132 Shift Right Algebraic Word Immediate

X-Form

- srawi RA,RS,SH (Rc=0)
- srawi. RA,RS,SH (Rc=1)

Pseudo-code:

```
n <- SH
r <- ROTL32((RS)[XLEN/2:XLEN-1], 64-n)
m <- MASK32(n, ((XLEN/2)-1))
s <- (RS)[XLEN/2]
RA <- r&m | ([s]*XLEN)& ¬m
carry <- s & ((r&¬m)[XLEN/2:XLEN-1] != 0)
CA    <- carry
CA32  <- carry
```

Special Registers Altered:

```
CA CA32
CR0                    (if Rc=1)
```

J.133 Shift Right Algebraic Word

X-Form

- sraw RA,RS,RB (Rc=0)
- sraw. RA,RS,RB (Rc=1)

Pseudo-code:

```
n <- (RB)[XLEN-5:XLEN-1]
r <- ROTL32((RS)[XLEN/2:XLEN-1], XLEN-n)
if (RB)[XLEN-6] = 0 then
    m <- MASK32(n, ((XLEN/2)-1))
else m <- [0]*XLEN
s <- (RS)[XLEN/2]
RA <- r&m | ([s]*XLEN)& ¬m
carry <-  s & ((r&¬m)[XLEN/2:XLEN-1] != 0)
CA    <-  carry
CA32  <-  carry
```

Special Registers Altered:

```
CA CA32
CR0                    (if Rc=1)
```

J.134 Shift Left Doubleword

X-Form

- sld RA,RS,RB (Rc=0)
- sld. RA,RS,RB (Rc=1)

Pseudo-code:

```
n <- (RB)[XLEN-6:XLEN-1]
r <- ROTL64((RS), n)
if (RB)[XLEN-7] = 0 then
     m <- MASK(0, XLEN-1-n)
else m <- [0]*XLEN
RA <- r & m
```

Special Registers Altered:

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

J.135 Shift Right Doubleword

X-Form

- srd RA,RS,RB (Rc=0)
- srd. RA,RS,RB (Rc=1)

Pseudo-code:

```
n <- (RB)[XLEN-6:XLEN-1]
r <- ROTL64((RS), XLEN-n)
if (RB)[XLEN-7] = 0 then
    m <-  MASK(n, (XLEN-1))
else m <- [0]*XLEN
RA <- r & m
```

Special Registers Altered:

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

J.136 Shift Right Algebraic Doubleword Immediate

XS-Form

- sradi RA,RS,sh (Rc=0)
- sradi. RA,RS,sh (Rc=1)

Pseudo-code:

```
n <- sh
r <- ROTL64((RS), XLEN-n)
m <- MASK(n, (XLEN-1))
s <- (RS)[0]
RA <- r&m | ([s]*XLEN)& ¬m
carry <- s & ((r& ¬m) != 0)
CA    <- carry
CA32  <- carry
```

Special Registers Altered:

```
CA CA32
CR0                    (if Rc=1)
```

J.137 Shift Right Algebraic Doubleword

X-Form

- srad RA,RS,RB (Rc=0)
- srad. RA,RS,RB (Rc=1)

Pseudo-code:

```
n <- (RB)[XLEN-6:XLEN-1]
r <- ROTL64((RS), XLEN-n)
if (RB)[XLEN-7] = 0 then
```

```
    m <-  MASK(n, (XLEN-1))
else m <- [0]*XLEN
s <- (RS)[0]
RA <- r&m | ([s]*XLEN)& ¬m
carry <-  s & ((r&¬m) != 0)
CA    <-  carry
CA32  <-  carry
```

Special Registers Altered:

```
CA CA32
CR0                    (if Rc=1)
```

J.138 Extend-Sign Word and Shift Left Immediate

XS-Form

- extswsli RA,RS,sh (Rc=0)
- extswsli. RA,RS,sh (Rc=1)

Pseudo-code:

```
n  <- sh
r  <- ROTL64(EXTS64(RS[XLEN/2:XLEN-1]), n)
m  <- MASK(0, XLEN-1-n)
RA <- r & m
```

Special Registers Altered:

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

Fixed Point Store pseudocode

J.139 Store Byte

D-Form

- stb RS,D(RA)

Pseudo-code:

```
b <- (RA|0)
EA <- b + EXTS(D)
MEM(EA, 1) <- (RS)[XLEN-8:XLEN-1]
```

Special Registers Altered:

```
None
```

J.140 Store Byte Indexed

X-Form

- stbx RS,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
MEM(EA, 1) <- (RS)[XLEN-8:XLEN-1]
```

Special Registers Altered:

```
None
```

J.141 Store Byte with Update

D-Form

- stbu RS,D(RA)

Pseudo-code:

```
EA <- (RA) + EXTS(D)
MEM(EA, 1) <- (RS)[XLEN-8:XLEN-1]
RA <- EA
```

Special Registers Altered:

```
None
```

J.142 Store Byte with Update Indexed

X-Form

- stbux RS,RA,RB

Pseudo-code:

```
EA <- (RA) + (RB)
MEM(EA, 1) <- (RS)[XLEN-8:XLEN-1]
RA <- EA
```

Special Registers Altered:

```
None
```

J.143 Store Halfword

D-Form

- sth RS,D(RA)

Pseudo-code:

```
b <- (RA|0)
EA <- b + EXTS(D)
MEM(EA, 2) <- (RS)[XLEN-16:XLEN-1]
```

Special Registers Altered:

```
None
```

J.144 Store Halfword Indexed

X-Form

- sthx RS,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
MEM(EA, 2) <- (RS)[XLEN-16:XLEN-1]
```

Special Registers Altered:

```
None
```

J.145 Store Halfword with Update

D-Form

- sthu RS,D(RA)

Pseudo-code:

```
EA <- (RA) + EXTS(D)
MEM(EA, 2) <- (RS)[XLEN-16:XLEN-1]
RA <- EA
```

Special Registers Altered:

```
None
```

J.146 Store Halfword with Update Indexed

X-Form

- sthux RS,RA,RB

Pseudo-code:

```
EA <- (RA) + (RB)
MEM(EA, 2) <- (RS)[XLEN-16:XLEN-1]
RA <- EA
```

Special Registers Altered:

```
None
```

J.147 Store Word

D-Form

- stw RS,D(RA)

Pseudo-code:

```
b <- (RA|0)
EA <- b + EXTS(D)
MEM(EA, 4) <- (RS)[XLEN-32:XLEN-1]
```

Special Registers Altered:

```
None
```

J.148 Store Word Indexed

X-Form

- stwx RS,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
MEM(EA, 4) <- (RS)[XLEN-32:XLEN-1]
```

Special Registers Altered:

```
None
```

J.149 Store Word with Update

D-Form

- stwu RS,D(RA)

Pseudo-code:

```
EA <- (RA) + EXTS(D)
MEM(EA, 4) <- (RS)[XLEN-32:XLEN-1]
RA <- EA
```

Special Registers Altered:

```
None
```

J.150 Store Word with Update Indexed

X-Form

- stwux RS,RA,RB

Pseudo-code:

```
EA <- (RA) + (RB)
MEM(EA, 4) <- (RS)[XLEN-32:XLEN-1]
RA <- EA
```

Special Registers Altered:

```
None
```

J.151 Store Doubleword

DS-Form

- std RS,DS(RA)

Pseudo-code:

```
b <- (RA|0)
EA <- b + EXTS(DS || 0b00)
MEM(EA, 8) <- (RS)
```

Special Registers Altered:

```
None
```

J.152 Store Doubleword Indexed

X-Form

- stdx RS,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
MEM(EA, 8) <- (RS)
```

Special Registers Altered:

```
None
```

J.153 Store Doubleword with Update

DS-Form

- stdu RS,DS(RA)

Pseudo-code:

```
EA <- (RA) + EXTS(DS || 0b00)
MEM(EA, 8) <- (RS)
RA <- EA
```

Special Registers Altered:

```
None
```

J.154 Store Doubleword with Update Indexed

X-Form

- stdux RS,RA,RB

Pseudo-code:

```
EA <- (RA) + (RB)
MEM(EA, 8) <- (RS)
RA <- EA
```

Special Registers Altered:

```
None
```

J.155 Store Quadword

DS-Form

- stq RSp,DS(RA)

Pseudo-code:

```
b <- (RA|0)
EA <- b + EXTS(DS || 0b00)
MEM(EA, 16) <- RSp
```

Special Registers Altered:

```
None
```

J.156 Store Halfword Byte-Reverse Indexed

X-Form

- sthbrx RS,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
MEM(EA, 2) <- (RS) [56:63] || (RS)[48:55]
```

Special Registers Altered:

```
None
```

J.157 Store Word Byte-Reverse Indexed

X-Form

- stwbrx RS,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
MEM(EA, 4) <- ((RS)[56:63] || (RS)[48:55] || (RS)[40:47]
               ||(RS)[32:39])
```

Special Registers Altered:

```
None
```

J.158 Store Doubleword Byte-Reverse Indexed

X-Form

- stdbrx RS,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
MEM(EA, 8) <- ((RS) [56:63] || (RS)[48:55]
                || (RS)[40:47] || (RS)[32:39]
                || (RS)[24:31] || (RS)[16:23]
                || (RS)[8:15]  || (RS)[0:7])
```

Special Registers Altered:

```
None
```

J.159 Store Multiple Word

D-Form

- stmw RS,D(RA)

Pseudo-code:

```
b <- (RA|0)
EA <- b + EXTS(D)
r <- RS[0:63]
do while r <= 31
    MEM(EA, 4) <- GPR(r)[32:63]
    r <-  r + 1
    EA <-  EA + 4
```

Special Registers Altered:

```
None
```

Fixed Point Trap pseudocode

J.160 Trap Word Immediate

D-Form

- twi TO,RA,SI

Pseudo-code:

```
a <- EXTS((RA)[XLEN/2:XLEN-1])
if (a < EXTS(SI)) & TO[0]  then TRAP
if (a > EXTS(SI)) & TO[1]  then TRAP
if (a = EXTS(SI)) & TO[2]  then TRAP
if (a <u EXTS(SI)) & TO[3] then TRAP
if (a >u EXTS(SI)) & TO[4] then TRAP
```

Special Registers Altered:

```
None
```

J.161 Trap Word

X-Form

- tw TO,RA,RB

Pseudo-code:

```
a <- EXTS((RA)[XLEN/2:XLEN-1])
b <- EXTS((RB)[XLEN/2:XLEN-1])
if (a < b) & TO[0] then TRAP
if (a > b) & TO[1] then TRAP
if (a = b) & TO[2] then TRAP
if (a <u b) & TO[3] then TRAP
if (a >u b) & TO[4] then TRAP
```

Special Registers Altered:

```
None
```

J.162 Trap Doubleword Immediate

D-Form

- tdi TO,RA,SI

Pseudo-code:

```
a <- (RA)
b <- EXTS(SI)
if (a < b) & TO[0] then TRAP
if (a > b) & TO[1] then TRAP
if (a = b) & TO[2] then TRAP
if (a <u  b) & TO[3] then TRAP
if (a >u  b) & TO[4]  then TRAP
```

Special Registers Altered:

```
None
```

J.163 Trap Doubleword

X-Form

- td TO,RA,RB

Pseudo-code:

```
a <- (RA)
b <- (RB)
if (a < b) & TO[0] then TRAP
if (a > b) & TO[1] then TRAP
if (a = b) & TO[2] then TRAP
if (a <u b) & TO[3] then TRAP
if (a >u b) & TO[4]  then TRAP
```

Special Registers Altered:

```
None
```

J.164 Integer Select

A-Form

- isel RT,RA,RB,BC

Pseudo-code:

```
if CR[BC+32]=1 then RT <- (RA|0)
else                RT <- (RB)
```

Special Registers Altered:

```
None
```

Special Purpose Register pseudocode

J.165 Move To Special Purpose Register

XFX-Form

- mtspr spr,RS

Pseudo-code:

```
n <- spr
switch (n)
  case(13): see(Book_III_p974)
  case(808, 809, 810, 811):
  default:
    if length(SPR(n)) = 64 then
      SPR(n) <- (RS)
    else
      SPR(n) <- (RS) [32:63]
```

Special Registers Altered:

```
See spec 3.3.17
```

J.166 Move From Special Purpose Register

XFX-Form

- mfspr RT,spr

Pseudo-code:

```
n <- spr
switch (n)
  case(129): see(Book_III_p975)
  case(808, 809, 810, 811):
  default:
    if length(SPR(n)) = 64 then
      RT <- SPR(n)
    else
      RT <- [0]*32 || SPR(n)
```

Special Registers Altered:

```
None
```

J.167 Move to CR from XER Extended

X-Form

- mcrxrx BF

Pseudo-code:

```
CR[4*BF+32:4*BF+35] <-  XER[OV] || XER[OV32] || XER[CA] || XER[CA32]
```

Special Registers Altered:

```
CR field BF
```

J.168 Move To One Condition Register Field

XFX-Form

- mtocrf FXM,RS

Pseudo-code:

```
n <- 7
do i = 7 to 0
  if FXM[i] = 1 then
    n <- i
CR[4*n+32:4*n+35] <- (RS)[4*n+32:4*n+35]
```

Special Registers Altered:

```
CR field selected by FXM
```

J.169 Move To Condition Register Fields

XFX-Form

- mtcrf FXM,RS

Pseudo-code:

```
do n = 0 to 7
  if FXM[n] = 1 then
    CR[4*n+32:4*n+35] <- (RS)[4*n+32:4*n+35]
```

Special Registers Altered:

```
CR fields selected by mask
```

J.170 Move From One Condition Register Field

XFX-Form

- mfocrf RT,FXM

Pseudo-code:

```
done <- 0
RT <- [0]*64
do n = 0 to 7
  if (done = 0) & (FXM[n] = 1) then
    RT[4*n+32:4*n+35] <- CR[4*n+32:4*n+35]
    done <- 1
```

Special Registers Altered:

```
None
```

J.171 Move From Condition Register

XFX-Form

- mfcr RT

Pseudo-code:

```
RT <- [0]*32 || CR
```

Special Registers Altered:

```
None
```

J.172 Set Boolean

X-Form

- setb RT,BFA

Pseudo-code:

```
if CR[4*BFA+32] = 1 then
   RT <- 0xFFFF_FFFF_FFFF_FFFF
else if CR[4*BFA+33]=1 then
   RT <- 0x0000_0000_0000_0001
else
   RT <- 0x0000_0000_0000_0000
```

Special Registers Altered:

```
None
```

J.173 Move To Machine State Register

X-Form

- mtmsr RS,L1

Pseudo-code:

```
if L1 = 0 then
    MSR[48] <- (RS)[48] | (RS)[49]
    MSR[58] <- (RS)[58] | (RS)[49]
    MSR[59] <- (RS)[59] | (RS)[49]
    MSR[32:40] <- (RS)[32:40]
    MSR[42:47] <- (RS)[42:47]
```

```
        MSR[49:50] <- (RS)[49:50]
        MSR[52:57] <- (RS)[52:57]
        MSR[60:62] <- (RS)[60:62]
    else
        MSR[48] <- (RS)[48]
        MSR[62] <- (RS)[62]
```

Special Registers Altered:

```
MSR
```

J.174 Move To Machine State Register

X-Form

- mtmsrd RS,L1

Pseudo-code:

```
if L1 = 0 then
    if (MSR[29:31] != 0b010) | ((RS)[29:31] != 0b000) then
        MSR[29:31] <- (RS)[29:31]
    MSR[48] <- (RS)[48] | (RS)[49]
    MSR[58] <- (RS)[58] | (RS)[49]
    MSR[59] <- (RS)[59] | (RS)[49]
    MSR[0:2] <- (RS)[0:2]
    MSR[4:28] <- (RS)[4:28]
    MSR[32:40] <- (RS)[32:40]
    MSR[42:47] <- (RS)[42:47]
    MSR[49:50] <- (RS)[49:50]
    MSR[52:57] <- (RS)[52:57]
    MSR[60:62] <- (RS)[60:62]
else
    MSR[48] <- (RS)[48]
    MSR[62] <- (RS)[62]
```

Special Registers Altered:

```
MSR
```

J.175 Move From Machine State Register

X-Form

- mfmsr RT

Pseudo-code:

```
RT <- MSR
```

Special Registers Altered:

```
None
```

J.176 Data Cache Block set to Zero

X-Form

- dcbz RA,RB

Pseudo-code:

```
if RA = 0 then b <- 0
else           b <-(RA)
EA <- b + (RB)
```

Special Registers Altered:

```
None
```

J.177 TLB Invalidate Entry

X-Form

- tlbie RB,RS,RIC,PRS,R

Pseudo-code:

```
IS <- (RB) [52:53]
```

Special Registers Altered:

```
None
```

String Load/Store pseudocode

J.178 Load String Word Immediate

X-Form

- lswi RT,RA,NB

Pseudo-code:

```
EA <- (RA|0)
if NB = 0 then n <-  32
else           n <-  NB
r <- RT - 1
i <- 32
do while n > 0
    if i = 32 then
        r <- (r + 1) % 32
        GPR(r) <- 0
    GPR(r)[i:i+7] <- MEM(EA, 1)
    i <- i + 8
    if i = 64 then i <- 32
    EA <- EA + 1
    n <- n - 1
```

Special Registers Altered:

```
None
```

J.179 Load String Word Indexed

X-Form

- lswx RT,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
n <- XER[57:63]
r <- RT - 1
i <- 32
RT <- undefined([0]*64)
do while n > 0
    if i = 32 then
        r <-  (r + 1) % 32
        GPR(r) <- 0
```

```
GPR(r)[i:i+7] <- MEM(EA, 1)
i <-  i + 8
if i = 64 then i <- 32
EA <-  EA + 1
n <- n - 1
```

Special Registers Altered:

```
None
```

J.180 Store String Word Immediate

X-Form

- stswi RS,RA,NB

Pseudo-code:

```
EA <- (RA|0)
if NB = 0 then n <-  32
else           n <-  NB
r <- RS - 1
i <- 32
do while n > 0
  if i = 32 then r <-  (r + 1) % 32
  MEM(EA, 1) <-  GPR(r)[i:i+7]
  i <-  i + 8
  if i = 64 then i <-  32
  EA <-  EA + 1
  n <- n - 1
```

Special Registers Altered:

```
None
```

J.181 Store String Word Indexed

X-Form

- stswx RS,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
n <- XER[57:63]
r <- RS - 1
i <- 32
do while n > 0
  if i = 32 then r <- (r + 1) % 32
  MEM(EA, 1) <- GPR(r)[i:i+7]
  i <-  i + 8
  if i = 64 then i <- 32
  EA <-  EA + 1
  n <- n - 1
```

Special Registers Altered:

```
None
```

System Call pseudocode

J.182 System Call

SC-Form

- sc LEV

Pseudo-code:

```
SRR0 <-iea CIA + 4
SRR1[33:36] <- 0
SRR1[42:47] <- 0
SRR1[0:32]  <- MSR[0:32]
SRR1[37:41] <- MSR[37:41]
SRR1[48:63] <- MSR[48:63]
MSR <- new_value
NIA <- 0x0000_0000_0000_0C00
```

Special Registers Altered:

```
SRR0 SRR1 MSR
```

J.183 System Call Vectored

SC-Form

- scv LEV

Pseudo-code:

```
LR <- CIA + 4
SRR1[33:36] <- undefined([0]*4)
SRR1[42:47] <- undefined([0]*6)
SRR1[0:32]  <- MSR[0:32]
SRR1[37:41] <- MSR[37:41]
SRR1[48:63] <- MSR[48:63]
MSR <- new_value
NIA <- vectored
```

Special Registers Altered:

```
LR CTR MSR
```

J.184 Return From System Call Vectored

XL-Form

- rfscv

Pseudo-code:

```
if (MSR[29:31] != 0b010) | (CTR[29:31] != 0b000) then
    MSR[29:31] <- CTR[29:31]
MSR[48] <- CTR[49]
MSR[58] <- CTR[49]
MSR[59] <- CTR[49]
MSR[0:2] <- CTR[0:2]
MSR[4:28] <- CTR[4:28]
MSR[32] <- CTR[32]
MSR[37:41] <- CTR[37:41]
MSR[49:50] <- CTR[49:50]
MSR[52:57] <- CTR[52:57]
MSR[60:63] <- CTR[60:63]
NIA <-iea LR[0:61] || 0b00
```

Special Registers Altered:

```
MSR
```

J.185 Return From Interrupt Doubleword

XL-Form

- rfid

Pseudo-code:

```
MSR[51] <- (MSR[3] & SRR1[51]) | ((¬MSR[3] & MSR[51]))
MSR[3] <- (MSR[3] & SRR1[3])
if (MSR[29:31] != 0b010) | (SRR1[29:31] != 0b000) then
    MSR[29:31] <- SRR1[29:31]
MSR[48] <- SRR1[48] | SRR1[49]
MSR[58] <- SRR1[58] | SRR1[49]
MSR[59] <- SRR1[59] | SRR1[49]
MSR[0:2] <- SRR1[0:2]
MSR[4:28] <- SRR1[4:28]
MSR[32] <- SRR1[32]
MSR[37:41] <- SRR1[37:41]
MSR[49:50] <- SRR1[49:50]
MSR[52:57] <- SRR1[52:57]
MSR[60:63] <- SRR1[60:63]
NIA <-iea SRR0[0:61] || 0b00
```

Special Registers Altered:

```
MSR
```

J.186 Hypervisor Return From Interrupt Doubleword

XL-Form

- hrfid

Pseudo-code:

```
if (MSR[29:31] != 0b010) | (HSRR1[29:31] != 0b000) then
    MSR[29:31] <- HSRR1[29:31]
MSR[48] <- HSRR1[48] | HSRR1[49]
MSR[58] <- HSRR1[58] | HSRR1[49]
MSR[59] <- HSRR1[59] | HSRR1[49]
MSR[0:28] <- HSRR1[0:28]
MSR[32] <- HSRR1[32]
MSR[37:41] <- HSRR1[37:41]
MSR[49:57] <- HSRR1[49:57]
MSR[60:63] <- HSRR1[60:63]
NIA <-iea HSRR0[0:61] || 0b00
```

Special Registers Altered:

```
MSR
```

Floating Point Load pseudocode

J.187 Load Floating-Point Single

D-Form

- lfs FRT,D(RA)

Pseudo-code:

```
EA <- (RA|0) + EXTS(D)
FRT <- DOUBLE(MEM(EA, 4))
```

Special Registers Altered:

```
None
```

J.188 Load Floating-Point Single Indexed

X-Form

- lfsx FRT,RA,RB

Pseudo-code:

```
EA <- (RA|0) + (RB)
FRT <- DOUBLE(MEM(EA, 4))
```

Special Registers Altered:

```
None
```

J.189 Load Floating-Point Single with Update

D-Form

- lfsu FRT,D(RA)

Pseudo-code:

```
EA <- (RA) + EXTS(D)
FRT <- DOUBLE(MEM(EA, 4))
RA <- EA
```

Special Registers Altered:

```
None
```

J.190 Load Floating-Point Single with Update Indexed

X-Form

- lfsux FRT,RA,RB

Pseudo-code:

```
EA <- (RA) + (RB)
FRT <- DOUBLE(MEM(EA, 4))
RA <- EA
```

Special Registers Altered:

```
None
```

J.191 Load Floating-Point Double

D-Form

- lfd FRT,D(RA)

Pseudo-code:

```
EA <- (RA|0) + EXTS(D)
FRT <- MEM(EA, 8)
```

Special Registers Altered:

```
None
```

J.192 Load Floating-Point Double Indexed

X-Form

- lfdx FRT,RA,RB

Pseudo-code:

```
EA <- (RA|0) + (RB)
FRT <- MEM(EA, 8)
```

Special Registers Altered:

```
None
```

J.193 Load Floating-Point Double with Update

D-Form

- lfdu FRT,D(RA)

Pseudo-code:

```
EA <- (RA) + EXTS(D)
FRT <- MEM(EA, 8)
RA <- EA
```

Special Registers Altered:

```
None
```

J.194 Load Floating-Point Double with Update Indexed

X-Form

- lfdux FRT,RA,RB

Pseudo-code:

```
EA <- (RA) + (RB)
FRT <- MEM(EA, 8)
RA <- EA
```

Special Registers Altered:

```
None
```

J.195 Load Floating-Point as Integer Word Algebraic Indexed

X-Form

- lfiwax FRT,RA,RB

Pseudo-code:

```
EA <- (RA|0) + (RB)
FRT <- EXTS(MEM(EA, 4))
```

Special Registers Altered:

```
None
```

J.196 Load Floating-Point as Integer Word Zero Indexed

X-Form

- lfiwzx FRT,RA,RB

Pseudo-code:

```
EA <- (RA|0) + (RB)
FRT <- [0]*32 || MEM(EA, 4)
```

Special Registers Altered:

```
None
```

Floating Point Store pseudocode

J.197 Store Floating-Point Single

D-Form

- stfs FRS,D(RA)

Pseudo-code:

```
EA <- (RA|0) + EXTS(D)
MEM(EA, 4)<- SINGLE( (FRS) )
```

Special Registers Altered:

```
None
```

J.198 Store Floating-Point Single Indexed

X-Form

- stfsx FRS,RA,RB

Pseudo-code:

```
EA <- (RA|0) + (RB)
MEM(EA, 4)<- SINGLE( (FRS) )
```

Special Registers Altered:

```
None
```

J.199 Store Floating-Point Single with Update

D-Form

- stfsu FRS,D(RA)

Pseudo-code:

```
EA <- (RA) + EXTS(D)
MEM(EA, 4)<- SINGLE( (FRS) )
RA <- EA
```

Special Registers Altered:

```
None
```

J.200 Store Floating-Point Single with Update Indexed

X-Form

- stfsux FRS,RA,RB

Pseudo-code:

```
EA <- (RA) + (RB)
MEM(EA, 4)<- SINGLE( (FRS) )
RA <- EA
```

Special Registers Altered:

```
None
```

J.201 Store Floating-Point Double

D-Form

- stfd FRS,D(RA)

Pseudo-code:

```
EA <- (RA|0) + EXTS(D)
MEM(EA, 8)<- (FRS)
```

Special Registers Altered:

```
None
```

J.202 Store Floating-Point Double Indexed

X-Form

- stfdx FRS,RA,RB

Pseudo-code:

```
EA <- (RA|0) + (RB)
MEM(EA, 8)<- (FRS)
```

Special Registers Altered:

```
None
```

J.203 Store Floating-Point Double with Update

D-Form

- stfdu FRS,D(RA)

Pseudo-code:

```
EA <- (RA) + EXTS(D)
MEM(EA, 8)<- (FRS)
RA <- EA
```

Special Registers Altered:

```
None
```

J.204 Store Floating-Point Double with Update Indexed

X-Form

- stfdux FRS,RA,RB

Pseudo-code:

```
EA <- (RA) + (RB)
MEM(EA, 8)<- (FRS)
RA <- EA
```

Special Registers Altered:

```
None
```

J.205 Store Floating-Point as Integer Word Indexed

X-Form

- stfiwx FRS,RA,RB

Pseudo-code:

```
b <- (RA|0)
EA <- b + (RB)
MEM(EA, 8)<- (FRS)[32:63]
```

Special Registers Altered:

```
None
```

Floating Point Move pseudocode

J.206 Floating Move Register

X-Form

- fmr FRT,FRB (Rc=0)
- fmr. FRT,FRB (Rc=1)

Pseudo-code:

```
FRT <- FRB[0:63]
```

Special Registers Altered:

```
CR1          (if Rc=1)
```

J.207 Floating Absolute Value Register

X-Form

- fabs FRT,FRB (Rc=0)
- fabs. FRT,FRB (Rc=1)

Pseudo-code:

```
FRT <- 0b0 || FRB[1:63]
```

Special Registers Altered:

```
CR1          (if Rc=1)
```

J.208 Floating Negative Absolute Value Register

X-Form

- fnabs FRT,FRB (Rc=0)
- fnabs. FRT,FRB (Rc=1)

Pseudo-code:

```
FRT <- 0b1 || FRB[1:63]
```

Special Registers Altered:

```
CR1          (if Rc=1)
```

J.209 Floating Negate Register

X-Form

- fneg FRT,FRB (Rc=0)
- fneg. FRT,FRB (Rc=1)

Pseudo-code:

```
FRT <- ¬FRB[0] || FRB[1:63]
```

Special Registers Altered:

```
CR1          (if Rc=1)
```

J.210 Floating Copy Sign Register

X-Form

- fcpsgn FRT,FRA,FRB (Rc=0)
- fcpsgn. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
FRT <- FRA[0] || FRB[1:63]
```

Special Registers Altered:

```
CR1          (if Rc=1)
```

J.211 [DRAFT] Floating Move To GPR

X-Form

- fmvtg RT,FRB (Rc=0)
- fmvtg. RT,FRB (Rc=1)

Pseudo-code:

```
RT <- (FRB)
```

Special Registers Altered:

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

J.212 [DRAFT] Floating Move To GPR Single

X-Form

- fmvtgs RT,FRB (Rc=0)
- fmvtgs. RT,FRB (Rc=1)

Pseudo-code:

```
RT <- [0] * (XLEN/2) || SINGLE((FRB))  # SINGLE since that's what stfs uses
```

Special Registers Altered:

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

J.213 [DRAFT] Floating Move From GPR

X-Form

- fmvfg FRT,RB (Rc=0)
- fmvfg. FRT,RB (Rc=1)

Pseudo-code:

```
FRT <- (RB)
```

Special Registers Altered:

```
CR1          (if Rc=1)
```

J.214 [DRAFT] Floating Move From GPR Single

X-Form

- fmvfgs FRT,RB (Rc=0)
- fmvfgs. FRT,RB (Rc=1)

Pseudo-code:

```
FRT <- DOUBLE((RB)[XLEN/2:XLEN-1])  # DOUBLE since that's what lfs uses
```

Special Registers Altered:

```
CR1          (if Rc=1)
```

Floating Point Arithmetic pseudocode

J.215 Floating Add [Single]

A-Form

- fadds FRT,FRA,FRB (Rc=0)
- fadds. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
FRT <- FPADD32(FRA, FRB)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1          (if Rc=1)
```

J.216 Floating Add [Double]

A-Form

- fadd FRT,FRA,FRB (Rc=0)
- fadd. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
FRT <- FPADD64(FRA, FRB)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1          (if Rc=1)
```

J.217 Floating Subtract [Single]

A-Form

- fsubs FRT,FRA,FRB (Rc=0)
- fsubs. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
FRT <- FPSUB32(FRA, FRB)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1          (if Rc=1)
```

J.218 Floating Subtract [Double]

A-Form

- fsub FRT,FRA,FRB (Rc=0)
- fsub. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
FRT <- FPSUB64(FRA, FRB)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1          (if Rc=1)
```

J.219 Floating Multiply [Single]

A-Form

- fmuls FRT,FRA,FRC (Rc=0)
- fmuls. FRT,FRA,FRC (Rc=1)

Pseudo-code:

```
FRT <- FPMUL32(FRA, FRC)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1          (if Rc=1)
```

J.220 Floating Multiply [Double]

A-Form

- fmul FRT,FRA,FRC (Rc=0)
- fmul. FRT,FRA,FRC (Rc=1)

Pseudo-code:

```
FRT <- FPMUL64(FRA, FRC)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1          (if Rc=1)
```

J.221 Floating Divide [Single]

A-Form

- fdivs FRT,FRA,FRB (Rc=0)
- fdivs. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
FRT <- FPDIV32(FRA, FRB)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1          (if Rc=1)
```

J.222 Floating Divide [Double]

A-Form

- fdiv FRT,FRA,FRB (Rc=0)
- fdiv. FRT,FRA,FRB (Rc=1)

Pseudo-code:

```
FRT <- FPDIV64(FRA, FRB)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI
CR1          (if Rc=1)
```

J.223 Floating Multiply-Add [Single]

A-Form

- fmadds FRT,FRA,FRC,FRB (Rc=0)
- fmadds. FRT,FRA,FRC,FRB (Rc=1)

Pseudo-code:

```
FRT <- FPMULADD32(FRA, FRC, FRB, 1, 1)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1          (if Rc=1)
```

J.224 Floating Multiply-Sub [Single]

A-Form

- fmsubs FRT,FRA,FRC,FRB (Rc=0)
- fmsubs. FRT,FRA,FRC,FRB (Rc=1)

Pseudo-code:

```
FRT <- FPMULADD32(FRA, FRC, FRB, 1, -1)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1          (if Rc=1)
```

J.225 Floating Negative Multiply-Add [Single]

A-Form

- fnmadds FRT,FRA,FRC,FRB (Rc=0)
- fnmadds. FRT,FRA,FRC,FRB (Rc=1)

Pseudo-code:

```
FRT <- FPMULADD32(FRA, FRC, FRB, -1, -1)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1          (if Rc=1)
```

J.226 Floating Negative Multiply-Sub [Single]

A-Form

- fnmsubs FRT,FRA,FRC,FRB (Rc=0)
- fnmsubs. FRT,FRA,FRC,FRB (Rc=1)

Pseudo-code:

```
FRT <- FPMULADD32(FRA, FRC, FRB, -1, 1)
```

Special Registers Altered:

```
FPRF FR FI
FX OX UX XX
VXSNAN VXISI VXIMZ
CR1          (if Rc=1)
```

Floating Point Integer Conversion pseudocode

J.227 Floating Convert with round Signed Doubleword to Single-Precision format

X-Form

- fcfids FRT,FRB (Rc=0)
- fcfids. FRT,FRB (Rc=1)

Pseudo-code:

```
FRT <- INT2FP(FRB, 'sint2single')
```

Special Registers Altered:

```
FPRF FR FI
FX XX
CR1        (if Rc=1)
```

J.228 [DRAFT] Floating Convert From Integer In GPR

X-Form

- fcvtfg FRT,RB,IT (Rc=0)
- fcvtfg. FRT,RB,IT (Rc=1)

Pseudo-code:

```
if IT[0] = 0 then  # 32-bit int -> 64-bit float
    # rounding never necessary, so don't touch FPSCR
    # based off xvcvsxwdp
    if IT = 0 then  # Signed 32-bit
        src <- bfp_CONVERT_FROM_SI32((RB)[32:63])
    else  # IT = 1 -- Unsigned 32-bit
        src <- bfp_CONVERT_FROM_UI32((RB)[32:63])
    FRT <- bfp64_CONVERT_FROM_BFP(src)
else
    # rounding may be necessary. based off xscvuxdsp
    reset_xflags()
    switch(IT)
        case(0):  # Signed 32-bit
            src <- bfp_CONVERT_FROM_SI32((RB)[32:63])
        case(1):  # Unsigned 32-bit
```

```
            src <- bfp_CONVERT_FROM_UI32((RB)[32:63])
        case(2):  # Signed 64-bit
            src <- bfp_CONVERT_FROM_SI64((RB))
        default:  # Unsigned 64-bit
            src <- bfp_CONVERT_FROM_UI64((RB))
    rnd <- bfp_ROUND_TO_BFP64(FPSCR[RN], src)
    result <- bfp64_CONVERT_FROM_BFP(rnd)
    cls <- fprf_CLASS_BFP64(result)
    if xx_flag = 1 then SetFX(FPSCR[XX])
    FRT <- result
    FPSCR[FPRF] <- cls
    FPSCR[FR] <- inc_flag
    FPSCR[FI] <- xx_flag
```

Special Registers Altered:

```
CR1          (if Rc=1)
FPRF FR FI FX XX  (if IT[0]=1)
```

J.229 [DRAFT] Floating Convert From Integer In GPR Single

X-Form

- fcvtfgs FRT,RB,IT (Rc=0)
- fcvtfgs. FRT,RB,IT (Rc=1)

Pseudo-code:

```
# rounding may be necessary. based off xscvuxdsp
reset_xflags()
switch(IT)
    case(0):  # Signed 32-bit
        src <- bfp_CONVERT_FROM_SI32((RB)[32:63])
    case(1):  # Unsigned 32-bit
        src <- bfp_CONVERT_FROM_UI32((RB)[32:63])
    case(2):  # Signed 64-bit
        src <- bfp_CONVERT_FROM_SI64((RB))
    default:  # Unsigned 64-bit
        src <- bfp_CONVERT_FROM_UI64((RB))
rnd <- bfp_ROUND_TO_BFP32(FPSCR[RN], src)
result32 <- bfp32_CONVERT_FROM_BFP(rnd)
cls <- fprf_CLASS_BFP32(result32)
result <- DOUBLE(result32)
if xx_flag = 1 then SetFX(FPSCR[XX])
FRT <- result
FPSCR[FPRF] <- cls
FPSCR[FR] <- inc_flag
FPSCR[FI] <- xx_flag
```

Special Registers Altered:

```
CR1          (if Rc=1)
FPRF FR FI FX XX
```

J.230 [DRAFT] Floating Convert To Integer In GPR

XO-Form

- fcvttg RT,FRB,CVM,IT (OE=0 Rc=0)
- fcvttg. RT,FRB,CVM,IT (OE=0 Rc=1)
- fcvttgo RT,FRB,CVM,IT (OE=1 Rc=0)
- fcvttgo. RT,FRB,CVM,IT (OE=1 Rc=1)

Pseudo-code:

```
# based on xscvdpuxws
reset_xflags()
src <- bfp_CONVERT_FROM_BFP64((FRB))
switch(IT)
    case(0):  # Signed 32-bit
        range_min <- bfp_CONVERT_FROM_SI32(0x8000_0000)
        range_max <- bfp_CONVERT_FROM_SI32(0x7FFF_FFFF)
        js_mask <- 0xFFFF_FFFF
    case(1):  # Unsigned 32-bit
        range_min <- bfp_CONVERT_FROM_UI32(0)
        range_max <- bfp_CONVERT_FROM_UI32(0xFFFF_FFFF)
        js_mask <- 0xFFFF_FFFF
    case(2):  # Signed 64-bit
        range_min <- bfp_CONVERT_FROM_SI64(-0x8000_0000_0000_0000)
        range_max <- bfp_CONVERT_FROM_SI64(0x7FFF_FFFF_FFFF_FFFF)
        js_mask <- 0xFFFF_FFFF_FFFF_FFFF
    default:  # Unsigned 64-bit
        range_min <- bfp_CONVERT_FROM_UI64(0)
        range_max <- bfp_CONVERT_FROM_UI64(0xFFFF_FFFF_FFFF_FFFF)
        js_mask <- 0xFFFF_FFFF_FFFF_FFFF
if (CVM[2] = 1) | (FPSCR[RN] = 0b01) then
    rnd <- bfp_ROUND_TO_INTEGER_TRUNC(src)
else if FPSCR[RN] = 0b00 then
    rnd <- bfp_ROUND_TO_INTEGER_NEAR_EVEN(src)
else if FPSCR[RN] = 0b10 then
    rnd <- bfp_ROUND_TO_INTEGER_CEIL(src)
else if FPSCR[RN] = 0b11 then
    rnd <- bfp_ROUND_TO_INTEGER_FLOOR(src)
switch(CVM)
    case(0, 1):  # OpenPower semantics
        if IsNaN(rnd) then
            result <- si64_CONVERT_FROM_BFP(range_min)
        else if bfp_COMPARE_GT(rnd, range_max) then
            result <- ui64_CONVERT_FROM_BFP(range_max)
        else if bfp_COMPARE_LT(rnd, range_min) then
            result <- si64_CONVERT_FROM_BFP(range_min)
        else if IT[1] = 1 then  # Unsigned 32/64-bit
            result <- ui64_CONVERT_FROM_BFP(range_max)
        else  # Signed 32/64-bit
            result <- si64_CONVERT_FROM_BFP(range_max)
    case(2, 3):  # Java/Saturating semantics
        if IsNaN(rnd) then
            result <- [0] * 64
        else if bfp_COMPARE_GT(rnd, range_max) then
            result <- ui64_CONVERT_FROM_BFP(range_max)
```

```
            else if bfp_COMPARE_LT(rnd, range_min) then
                result <- si64_CONVERT_FROM_BFP(range_min)
            else if IT[1] = 1 then  # Unsigned 32/64-bit
                result <- ui64_CONVERT_FROM_BFP(range_max)
            else  # Signed 32/64-bit
                result <- si64_CONVERT_FROM_BFP(range_max)
        default:  # JavaScript semantics
            # CVM = 6, 7 are illegal instructions
            # this works because the largest type we try to convert from has
            # 53 significand bits, and the largest type we try to convert to
            # has 64 bits, and the sum of those is strictly less than the 128
            # bits of the intermediate result.
            limit <- bfp_CONVERT_FROM_UI128([1] * 128)
            if IsInf(rnd) | IsNaN(rnd) then
                result <- [0] * 64
            else if bfp_COMPARE_GT(bfp_ABSOLUTE(rnd), limit) then
                result <- [0] * 64
            else
                result128 <- si128_CONVERT_FROM_BFP(rnd)
                result <- result128[64:127] & js_mask
    switch(IT)
        case(0):  # Signed 32-bit
            result <- EXTS64(result[32:63])
            result_bfp <- bfp_CONVERT_FROM_SI32(result[32:63])
        case(1):  # Unsigned 32-bit
            result <- EXTZ64(result[32:63])
            result_bfp <- bfp_CONVERT_FROM_UI32(result[32:63])
        case(2):  # Signed 64-bit
            result_bfp <- bfp_CONVERT_FROM_SI64(result)
        default:  # Unsigned 64-bit
            result_bfp <- bfp_CONVERT_FROM_UI64(result)
    if vxsnan_flag = 1 then SetFX(FPSCR[VXSNAN])
    if vxcvi_flag = 1 then SetFX(FPSCR[VXCVI])
    if xx_flag = 1 then SetFX(FPSCR[XX])
    vx_flag <- vxsnan_flag | vxcvi_flag
    vex_flag <- FPSCR[VE] & vx_flag
    if vex_flag = 0 then
        RT <- result
        FPSCR[FPRF] <- undefined
        FPSCR[FR] <- inc_flag
        FPSCR[FI] <- xx_flag
        if IsNaN(src) | ¬bfp_COMPARE_EQ(src, result_bfp) then
            overflow <- 1  # signals SO only when OE = 1
    else
        FPSCR[FR] <- 0
        FPSCR[FI] <- 0
```

Special Registers Altered:

```
    CR0                     (if Rc=1)
    SO OV OV32              (if OE=1)
    FPRF=0bUUUUU FR FI FX XX VXSNAN VXCV
```

J.231 [DRAFT] Floating Convert To Integer In GPR Single

XO-Form

- fcvttgs RT,FRB,CVM,IT (OE=0 Rc=0)
- fcvttgs. RT,FRB,CVM,IT (OE=0 Rc=1)
- fcvttgso RT,FRB,CVM,IT (OE=1 Rc=0)
- fcvttgso. RT,FRB,CVM,IT (OE=1 Rc=1)

Pseudo-code:

```
# based on xscvdpuxws
reset_xflags()
src <- bfp_CONVERT_FROM_BFP32(SINGLE((FRB)))
switch(IT)
    case(0):  # Signed 32-bit
        range_min <- bfp_CONVERT_FROM_SI32(0x8000_0000)
        range_max <- bfp_CONVERT_FROM_SI32(0x7FFF_FFFF)
        js_mask <- 0xFFFF_FFFF
    case(1):  # Unsigned 32-bit
        range_min <- bfp_CONVERT_FROM_UI32(0)
        range_max <- bfp_CONVERT_FROM_UI32(0xFFFF_FFFF)
        js_mask <- 0xFFFF_FFFF
    case(2):  # Signed 64-bit
        range_min <- bfp_CONVERT_FROM_SI64(-0x8000_0000_0000_0000)
        range_max <- bfp_CONVERT_FROM_SI64(0x7FFF_FFFF_FFFF_FFFF)
        js_mask <- 0xFFFF_FFFF_FFFF_FFFF
    default:  # Unsigned 64-bit
        range_min <- bfp_CONVERT_FROM_UI64(0)
        range_max <- bfp_CONVERT_FROM_UI64(0xFFFF_FFFF_FFFF_FFFF)
        js_mask <- 0xFFFF_FFFF_FFFF_FFFF
if (CVM[2] = 1) | (FPSCR[RN] = 0b01) then
    rnd <- bfp_ROUND_TO_INTEGER_TRUNC(src)
else if FPSCR[RN] = 0b00 then
    rnd <- bfp_ROUND_TO_INTEGER_NEAR_EVEN(src)
else if FPSCR[RN] = 0b10 then
    rnd <- bfp_ROUND_TO_INTEGER_CEIL(src)
else if FPSCR[RN] = 0b11 then
    rnd <- bfp_ROUND_TO_INTEGER_FLOOR(src)
switch(CVM)
    case(0, 1):  # OpenPower semantics
        if IsNaN(rnd) then
            result <- si64_CONVERT_FROM_BFP(range_min)
        else if bfp_COMPARE_GT(rnd, range_max) then
            result <- ui64_CONVERT_FROM_BFP(range_max)
        else if bfp_COMPARE_LT(rnd, range_min) then
            result <- si64_CONVERT_FROM_BFP(range_min)
        else if IT[1] = 1 then  # Unsigned 32/64-bit
            result <- ui64_CONVERT_FROM_BFP(range_max)
        else  # Signed 32/64-bit
            result <- si64_CONVERT_FROM_BFP(range_max)
    case(2, 3):  # Java/Saturating semantics
        if IsNaN(rnd) then
            result <- [0] * 64
        else if bfp_COMPARE_GT(rnd, range_max) then
            result <- ui64_CONVERT_FROM_BFP(range_max)
```

```
            else if bfp_COMPARE_LT(rnd, range_min) then
                result <- si64_CONVERT_FROM_BFP(range_min)
            else if IT[1] = 1 then  # Unsigned 32/64-bit
                result <- ui64_CONVERT_FROM_BFP(range_max)
            else  # Signed 32/64-bit
                result <- si64_CONVERT_FROM_BFP(range_max)
        default:  # JavaScript semantics
            # CVM = 6, 7 are illegal instructions
            # this works because the largest type we try to convert from has
            # 53 significand bits, and the largest type we try to convert to
            # has 64 bits, and the sum of those is strictly less than the 128
            # bits of the intermediate result.
            limit <- bfp_CONVERT_FROM_UI128([1] * 128)
            if IsInf(rnd) | IsNaN(rnd) then
                result <- [0] * 64
            else if bfp_COMPARE_GT(bfp_ABSOLUTE(rnd), limit) then
                result <- [0] * 64
            else
                result128 <- si128_CONVERT_FROM_BFP(rnd)
                result <- result128[64:127] & js_mask
    switch(IT)
        case(0):  # Signed 32-bit
            result <- EXTS64(result[32:63])
            result_bfp <- bfp_CONVERT_FROM_SI32(result[32:63])
        case(1):  # Unsigned 32-bit
            result <- EXTZ64(result[32:63])
            result_bfp <- bfp_CONVERT_FROM_UI32(result[32:63])
        case(2):  # Signed 64-bit
            result_bfp <- bfp_CONVERT_FROM_SI64(result)
        default:  # Unsigned 64-bit
            result_bfp <- bfp_CONVERT_FROM_UI64(result)
    if vxsnan_flag = 1 then SetFX(FPSCR[VXSNAN])
    if vxcvi_flag = 1 then SetFX(FPSCR[VXCVI])
    if xx_flag = 1 then SetFX(FPSCR[XX])
    vx_flag <- vxsnan_flag | vxcvi_flag
    vex_flag <- FPSCR[VE] & vx_flag
    if vex_flag = 0 then
        RT <- result
        FPSCR[FPRF] <- undefined
        FPSCR[FR] <- inc_flag
        FPSCR[FI] <- xx_flag
        if IsNaN(src) | ¬bfp_COMPARE_EQ(src, result_bfp) then
            overflow <- 1  # signals SO only when OE = 1
    else
        FPSCR[FR] <- 0
        FPSCR[FI] <- 0
```

Special Registers Altered:

```
CR0                     (if Rc=1)
SO OV OV32              (if OE=1)
FPRF=0bUUUUU FR FI FX XX VXSNAN VXCV
```