RFC ls009 Simple-V REMAP Subsystem </>

- Funded by NLnet under the Privacy and Enhanced Trust Programme, EU Horizon2020 Grant 825310, and NGI0 Entrust No 101069594

- https://libre-soc.org/openpower/sv/rfc/ls009/
- https://bugs.libre-soc.org/show_bug.cgi?id=1060
- https://git.openpower.foundation/isa/PowerISA/issues/124

Severity: Major

Status: New

Date: 26 Mar 2023. v2 TODO

Target: v3.2B

Source: v3.0B

Books and Section affected:

```
Book I, new Zero-Overhead-Loop Chapter.
Appendix E Power ISA sorted by opcode
Appendix F Power ISA sorted by version
Appendix G Power ISA sorted by Compliancy Subset
Appendix H Power ISA sorted by mnemonic
```

Summary

```
svremap  - Re-Mapping of Register Element Offsets
svindex  - General-purpose setting of SHAPEs to be re-mapped
svshape  - Hardware-level setting of SHAPEs for element re-mapping
svshape2 - Hardware-level setting of SHAPEs for element re-mapping (v2)
```

Submitter: Luke Leighton (Libre-SOC)

Requester: Libre-SOC

Impact on processor:

```
Addition of four new "Zero-Overhead-Loop-Control" DSP-style Vector-style
Management Instructions which provide advanced features such as Matrix
FFT DCT Hardware-Assist Schedules and general-purpose Index reordering.
```

Impact on software:

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

Keywords:

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

Motivation

These REMAP Management instructions provide state-of-the-art advanced capabilities to dramatically decrease instruction count and power reduction whilst retaining unprecedented general-purpose capability and a standard Sequential Execution Model.

Notes and Observations:

1. Although costly the alternatives in SIMD-paradigm software result in huge algorithmic complexity and associated power consumption increases. Loop-unrolling compilers are prevalent as is thousands to tens of thousands of instructions.
2. Core inner kernels of Matrix DCT DFT FFT NTT are dramatically reduced to a handful of instructions.
3. No REMAP instructions with the exception of Indexed rely on registers for the establishment of REMAP capability.
4. Future EXT1xx variants and SVP64/VSX will dramatically expand the power and flexibility of REMAP.
5. The Simple-V Compliancy Subsets make REMAP optional except in the Advanced Levels. Like v3.1 MMA it is **not** necessary for **all** hardware to implement REMAP.

Changes

Add the following entries to:

- the Appendices of SV Book
- Instructions of SV Book as a new Section
- SVI, SVM, SVM2, SVRM Form of Book I Section 1.6.1.6 and 1.6.2

Rationale and background </>

In certain common algorithms there are clear patterns of behaviour which typically require inline loop-unrolled instructions comprising interleaved permute and arithmetic operations. REMAP was invented to split the permuting from the arithmetic, and to allow the Indexing to be done as a hardware Schedule.

A normal Vector Add:

```
for i in range(VL):
  GPR[RT+i] <= GPR[RA+i] + GPR[RB+i];
```

A Hardware-assisted REMAP Vector Add:

```
for i in range(VL):
  GPR[RT+remap1(i)] <= GPR[RA+remap2(i)] + GPR[RB+remap3(i)];
```

The result is a huge saving on register file accesses (no need to calculate Indices then use Permutation instructions), instruction count (Matrix Multiply up to 127 FMACs is 3 instructions), and programmer sanity.

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.

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}.

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.

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, currently under development).* 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

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 original motivation for Indexed REMAP was to mitigate the need to add an expensive `mv.x` to the Scalar ISA, which was highly 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 is the "Get out of Jail Free" card which (for a price) allows any general-purpose arbitrary Element Permutation not covered by the other Hardware Schedules.

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.

Parallel Reduction is normally done explicitly as a loop-unrolled operation, taking up a significant number of instructions. With REMAP it is just three instructions: two for setup and one Scalar Base.

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.

Parallel Reduction combined with Parallel Prefix Sum can be combined to help perform AI non-linear interpolation in around twelve to fifteen instructions.

Add the following to SV Book

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, independently on each Vector src or dest register. Up to four separate independent REMAPs may be applied to the registers of any instruction.

A normal Vector Add (no Element-width Overrides):

```
for i in range(VL):
  GPR[RT+i] <= GPR[RA+i] + GPR[RB+i];
```

A Hardware-assisted REMAP Vector Add:

```
for i in range(VL):
  GPR[RT+remap1(i)] <= GPR[RA+remap2(i)] + GPR[RB+remap3(i)];
```

Aside from Indexed REMAP this is entirely Hardware-accelerated reordering and consequently not costly in terms of register access for the Indices. 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.

Hardware note: 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.*

There are five 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 (a future variant may introduce Inner Product).
- **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.

Further details on the Deterministic Precise-Interruptible algorithms used in these Schedules is found in the {REMAP Appendix}.

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.

Determining Register Hazards (hphint) </>

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.

Horizontal-Parallelism Hint

To help further in reducing Hazards, `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`, which greatly aids simplification of Multi-Issue implementations.

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.

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.

---

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` establishes the connection 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 immediately-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.

SHAPE Remapping SPRs </>

There are four "shape" SPRs, SHAPE0-3, 32-bits in each, which have the same format. It is possible to write directly to these SPRs but it is recommended to use the Management instructions `svshape`, `svshape2` or `svindex`.

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`

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.

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. The Schedule Algorithm may be found in the {REMAP Appendix}

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-instruction operations. 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).

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.

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 Word-instructions 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

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.

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.

Limitations and caveats

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 provide more comprehensive capacity and mitigate the need to write direct to the SVSHAPE SPRs.

Additionally there is not yet a way to set Matrix sizes from registers with `svshape`: this was an intentional decision to simplify Hardware, that may be corrected in a future version of SVP64. The limitation may presently be overcome by direct programming of the SVSHAPE SPRs.

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. Once solved Matrix REMAP will fit naturally.

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 {SVP64 Appendix} 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 (use the same GPRs), even if one SVSHAPE has different 2D dimensions and ordering from the others.

Caveats and Limitations

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.

Indexed REMAP **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.*

---

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 for Matrix Outer Product
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 instruction 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.

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.

---

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]]

Forms </>

Add `SVI, SVM, SVM2, SVRM` to `XO (26:31)` Field in Book I, 1.6.2

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

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

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

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

Add the following to Book I, 1.6.1, SVM2-Form

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

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

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

Add the following to Book I, 1.6.2

```
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
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
pst (21)
    Field used in REMAP to indicate "persistence" mode (REMAP
    continues to apply to multiple instructions)
    Formats: SVRM
rmm (11:15)
    REMAP Mode field for SVI-Form and SVM2-Form
    Formats: SVI, SVM2
sk (25)
    Field used to specify dimensional skipping in svindex
    Formats: SVI, SVM2
SVd (16:20)
    Immediate field used to specify the size of the REMAP dimension
    in the svindex and svshape2 instructions
    Formats: SVI, SVM2
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
    XO (21:23,26:31)
        Extended opcode field.  Note that bit 21 must be 1, 22 and 23
        must be zero, and bits 26-31 must be exactly the same as
        used for svshape.
        Formats: SVM2
```

Appendices </>

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

Form	Book	Page	Version	mnemonic	Description
SVRM	I	#	3.0B	svremap	REMAP enabling instruction
SVM	I	#	3.0B	svshape	REMAP shape instruction
SVM2	I	#	3.0B	svshape2	REMAP shape instruction (2)
SVI	I	#	3.0B	svindex	REMAP General-purpose Indexing

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 <a name="sv_remap.py_finite_state"> </>
# to end up being actually used in actual hardware <a name="sv_remap.py_to_end"> </>

# up to three dimensions permitted <a name="sv_remap.py_up_to"> </>
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 <a name="sv_remap.py_preprepare_the"> </>
# actually starting.  this algorithm can also be used for re-entrancy <a name="sv_remap.py_actually_starting"> <
# if exceptions occur and a REMAP has to be started from where the <a name="sv_remap.py_if_exceptions"> </>
# interrupt left off. <a name="sv_remap.py_interrupt_left"> </>
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.

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.

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.

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
```

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
```

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
```

Example Matrix 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 Vectorized 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

[[!tag standards]]

REMAP pseudocode </>

Written in python3 the following stand-alone executable source code is the Canonical Specification for each REMAP. Vectors of "loopends" are returned when Rc=1 in Vectors of CR Fields on `sv.svstep.`, or in Vertical-First Mode a single CR Field (CR0) on `svstep.`. The `SVSTATE.srcstep` or `SVSTATE.dststep` sequential offset is put through each algorithm to determine the actual Element Offset. Alternative implementations producing different ordering is prohibited as software will be critically relying on these Deterministic Schedules.

REMAP 2D/3D Matrix </>

The following stand-alone executable source code is the Canonical Specification for Matrix (2D/3D) REMAP. Hardware implementations are achievable with simple cascading counter-and-compares.

```
# python "yield" can be iterated. use this to make it clear how <a name="ls009.mdwn_python_yield"> </>
# the indices are generated by using natural-looking nested loops <a name="ls009.mdwn_the_indices"> </>
def iterate_indices(SVSHAPE):
    # get indices to iterate over, in the required order
    xd = SVSHAPE.lims[0]
    yd = SVSHAPE.lims[1]
    zd = SVSHAPE.lims[2]
    # create lists of indices to iterate over in each dimension
    x_r = list(range(xd))
    y_r = list(range(yd))
    z_r = list(range(zd))
    # invert the indices if needed
    if SVSHAPE.invxyz[0]: x_r.reverse()
    if SVSHAPE.invxyz[1]: y_r.reverse()
    if SVSHAPE.invxyz[2]: z_r.reverse()
    # start an infinite (wrapping) loop
    step = 0 # track src/dst step
    while True:
        for z in z_r:   # loop over 1st order dimension
            z_end = z == z_r[-1]
            for y in y_r:       # loop over 2nd order dimension
                y_end = y == y_r[-1]
                for x in x_r:           # loop over 3rd order dimension
                    x_end = x == x_r[-1]
                    # ok work out which order to construct things in.
                    # start by creating a list of tuples of the dimension
                    # and its limit
                    vals = [(SVSHAPE.lims[0], x, "x"),
                            (SVSHAPE.lims[1], y, "y"),
                            (SVSHAPE.lims[2], z, "z")
                           ]
                    # now select those by order.  this allows us to
                    # create schedules for [z][x], [x][y], or [y][z]
                    # for matrix multiply.
                    vals = [vals[SVSHAPE.order[0]],
                            vals[SVSHAPE.order[1]],
                            vals[SVSHAPE.order[2]]
                           ]
                    # ok now we can construct the result, using bits of
                    # "order" to say which ones get stacked on
                    result = 0
                    mult = 1
                    for i in range(3):
                        lim, idx, dbg = vals[i]
                        # some of the dimensions can be "skipped".  the order
                        # was actually selected above on all 3 dimensions,
                        # e.g. [z][x][y] or [y][z][x].  "skip" allows one of
                        # those to be knocked out
                        if SVSHAPE.skip == i+1: continue
                        idx *= mult   # shifts up by previous dimension(s)
                        result += idx # adds on this dimension
                        mult *= lim   # for the next dimension

                    loopends = (x_end |
                                ((y_end and x_end)<<1) |
                                 ((y_end and x_end and z_end)<<2))

                    yield result + SVSHAPE.offset, loopends
                    step += 1

def demo():
```

```
    # set the dimension sizes here
    xdim = 3
    ydim = 2
    zdim = 4

    # set total (can repeat, e.g. VL=x*y*z*4)
    VL = xdim * ydim * zdim

    # set up an SVSHAPE
    class SVSHAPE:
        pass
    SVSHAPE0 = SVSHAPE()
    SVSHAPE0.lims = [xdim, ydim, zdim]
    SVSHAPE0.order = [1,0,2]  # experiment with different permutations, here
    SVSHAPE0.mode = 0b00
    SVSHAPE0.skip = 0b00
    SVSHAPE0.offset = 0       # experiment with different offset, here
    SVSHAPE0.invxyz = [0,0,0] # inversion if desired

    # enumerate over the iterator function, getting new indices
    for idx, (new_idx, end) in enumerate(iterate_indices(SVSHAPE0)):
        if idx >= VL:
            break
        print ("%d->%d" % (idx, new_idx), "end", bin(end)[2:])

# run the demo <a name="ls009.mdwn_run_the"> </>
if __name__ == '__main__':
    demo()
```

REMAP Parallel Reduction pseudocode </>

The python3 program below is stand-alone executable and is the Canonical Specification for Parallel Reduction REMAP. The Algorithm below is not limited to RADIX2 sizes, and Predicate sources, unlike in Matrix REMAP, apply to the Element Indices **after** REMAP has been applied, not before. MV operations are not required: the algorithm tracks positions of elements that would normally be moved and when applying an Element Reduction Operation sources the operands from their last-known (tracked) position.

```
# a "yield" version of the Parallel Reduction REMAP algorithm. <a name="ls009.mdwn_a_yield"> </>
# the algorithm is in-place. it does not perform "MV" operations. <a name="ls009.mdwn_the_algorithm"> </>
# instead, where a masked-out value *should* be read from is tracked <a name="ls009.mdwn_instead_where"> </>

def iterate_indices(SVSHAPE, pred=None):
    # get indices to iterate over, in the required order
    xd = SVSHAPE.lims[0]
    # create lists of indices to iterate over in each dimension
    ix = list(range(xd))
    # invert the indices if needed
    if SVSHAPE.invxyz[0]: ix.reverse()
    # start a loop from the lowest step
    step = 1
    steps = []
    while step < xd:
        step *= 2
        steps.append(step)
    # invert the indices if needed
    if SVSHAPE.invxyz[1]: steps.reverse()
    for step in steps:
        stepend = (step == steps[-1]) # note end of steps
        idxs = list(range(0, xd, step))
        results = []
        for i in idxs:
            other = i + step // 2
            ci = ix[i]
            oi = ix[other] if other < xd else None
            other_pred = other < xd and (pred is None or pred[oi])
            if (pred is None or pred[ci]) and other_pred:
                if SVSHAPE.skip == 0b00: # submode 00
                    result = ci
                elif SVSHAPE.skip == 0b01: # submode 01
                    result = oi
                results.append([result + SVSHAPE.offset, 0])
            elif other_pred:
                ix[i] = oi
        if results:
```

```
            results[-1][1] = (stependd<<1) | 1  # notify end of loops
        yield from results

def demo():
    # set the dimension sizes here
    xdim = 9

    # set up an SVSHAPE
    class SVSHAPE:
        pass
    SVSHAPE0 = SVSHAPE()
    SVSHAPE0.lims = [xdim, 0, 0]
    SVSHAPE0.order = [0,1,2]
    SVSHAPE0.mode = 0b10
    SVSHAPE0.skip = 0b00
    SVSHAPE0.offset = 0       # experiment with different offset, here
    SVSHAPE0.invxyz = [0,0,0] # inversion if desired

    SVSHAPE1 = SVSHAPE()
    SVSHAPE1.lims = [xdim, 0, 0]
    SVSHAPE1.order = [0,1,2]
    SVSHAPE1.mode = 0b10
    SVSHAPE1.skip = 0b01
    SVSHAPE1.offset = 0       # experiment with different offset, here
    SVSHAPE1.invxyz = [0,0,0] # inversion if desired

    # enumerate over the iterator function, getting new indices
    shapes = list(iterate_indices(SVSHAPE0)), \
              list(iterate_indices(SVSHAPE1))
    for idx in range(len(shapes[0])):
        l = shapes[0][idx]
        r = shapes[1][idx]
        (l_idx, lend) = l
        (r_idx, rend) = r
        print ("%d->%d:%d" % (idx, l_idx, r_idx),
               "end", bin(lend)[2:], bin(rend)[2:])

# run the demo <a name="ls009.mdwn_run_the"> </>
if __name__ == '__main__':
    demo()
```

REMAP FFT pseudocode </>

The FFT REMAP is RADIX2 only.

```
# a "yield" version of the REMAP algorithm, for FFT Tukey-Cooley schedules <a name="ls009.mdwn_a_yield"> </>
# original code for the FFT Tukey-Cooley schedule: <a name="ls009.mdwn_original_code"> </>
# https://www.nayuki.io/res/free-small-fft-in-multiple-languages/fft.py <a name="ls009.mdwn_httpswwwnayukiioresf
"""
    # Radix-2 decimation-in-time FFT (real, not complex)
    size = 2
    while size <= n:
        halfsize = size // 2
        tablestep = n // size
        for i in range(0, n, size):
            k = 0
            for j in range(i, i + halfsize):
                jh = j+halfsize
                jl = j
                temp1 = vec[jh] * exptable[k]
                temp2 = vec[jl]
                vec[jh] = temp2 - temp1
                vec[jl] = temp2 + temp1
                k += tablestep
        size *= 2
"""

# python "yield" can be iterated. use this to make it clear how <a name="ls009.mdwn_python_yield"> </>
# the indices are generated by using natural-looking nested loops <a name="ls009.mdwn_the_indices"> </>
def iterate_butterfly_indices(SVSHAPE):
    # get indices to iterate over, in the required order
    n = SVSHAPE.lims[0]
    stride = SVSHAPE.lims[2] # stride-multiplier on reg access
    # creating lists of indices to iterate over in each dimension
```

```
    # has to be done dynamically, because it depends on the size
    # first, the size-based loop (which can be done statically)
    x_r = []
    size = 2
    while size <= n:
        x_r.append(size)
        size *= 2
    # invert order if requested
    if SVSHAPE.invxyz[0]: x_r.reverse()

    if len(x_r) == 0:
        return

    # start an infinite (wrapping) loop
    skip = 0
    while True:
        for size in x_r:           # loop over 3rd order dimension (size)
            x_end = size == x_r[-1]
            # y_r schedule depends on size
            halfsize = size // 2
            tablestep = n // size
            y_r = []
            for i in range(0, n, size):
                y_r.append(i)
            # invert if requested
            if SVSHAPE.invxyz[1]: y_r.reverse()
            for i in y_r:       # loop over 2nd order dimension
                y_end = i == y_r[-1]
                k_r = []
                j_r = []
                k = 0
                for j in range(i, i+halfsize):
                    k_r.append(k)
                    j_r.append(j)
                    k += tablestep
                # invert if requested
                if SVSHAPE.invxyz[2]: k_r.reverse()
                if SVSHAPE.invxyz[2]: j_r.reverse()
                for j, k in zip(j_r, k_r):   # loop over 1st order dimension
                    z_end = j == j_r[-1]
                    # now depending on MODE return the index
                    if SVSHAPE.skip == 0b00:
                        result = j              # for vec[j]
                    elif SVSHAPE.skip == 0b01:
                        result = j + halfsize   # for vec[j+halfsize]
                    elif SVSHAPE.skip == 0b10:
                        result = k              # for exptable[k]

                    loopends = (z_end |
                                ((y_end and z_end)<<1) |
                                 ((y_end and x_end and z_end)<<2))

                    yield (result * stride) + SVSHAPE.offset, loopends

def demo():
    # set the dimension sizes here
    xdim = 8
    ydim = 0 # not needed
    zdim = 1 # stride must be set to 1

    # set total. err don't know how to calculate how many there are...
    # do it manually for now
    VL = 0
    size = 2
    n = xdim
    while size <= n:
        halfsize = size // 2
        tablestep = n // size
        for i in range(0, n, size):
            for j in range(i, i + halfsize):
                VL += 1
        size *= 2
```

```
    # set up an SVSHAPE
    class SVSHAPE:
        pass
    # j schedule
    SVSHAPE0 = SVSHAPE()
    SVSHAPE0.lims = [xdim, ydim, zdim]
    SVSHAPE0.order = [0,1,2]  # experiment with different permutations, here
    SVSHAPE0.mode = 0b01
    SVSHAPE0.skip = 0b00
    SVSHAPE0.offset = 0       # experiment with different offset, here
    SVSHAPE0.invxyz = [0,0,0] # inversion if desired
    # j+halfstep schedule
    SVSHAPE1 = SVSHAPE()
    SVSHAPE1.lims = [xdim, ydim, zdim]
    SVSHAPE1.order = [0,1,2]  # experiment with different permutations, here
    SVSHAPE0.mode = 0b01
    SVSHAPE1.skip = 0b01
    SVSHAPE1.offset = 0       # experiment with different offset, here
    SVSHAPE1.invxyz = [0,0,0] # inversion if desired
    # k schedule
    SVSHAPE2 = SVSHAPE()
    SVSHAPE2.lims = [xdim, ydim, zdim]
    SVSHAPE2.order = [0,1,2]  # experiment with different permutations, here
    SVSHAPE0.mode = 0b01
    SVSHAPE2.skip = 0b10
    SVSHAPE2.offset = 0       # experiment with different offset, here
    SVSHAPE2.invxyz = [0,0,0] # inversion if desired

    # enumerate over the iterator function, getting new indices
    schedule = []
    for idx, (jl, jh, k) in enumerate(zip(iterate_butterfly_indices(SVSHAPE0),
                                          iterate_butterfly_indices(SVSHAPE1),
                                          iterate_butterfly_indices(SVSHAPE2))):
        if idx >= VL:
            break
        schedule.append((jl, jh, k))

    # ok now pretty-print the results, with some debug output
    size = 2
    idx = 0
    while size <= n:
        halfsize = size // 2
        tablestep = n // size
        print ("size %d halfsize %d tablestep %d" % \
                (size, halfsize, tablestep))
        for i in range(0, n, size):
            prefix = "i %d\t" % i
            k = 0
            for j in range(i, i + halfsize):
                (jl, je), (jh, he), (ks, ke) = schedule[idx]
                print ("  %-3d\t%s j=%-2d jh=%-2d k=%-2d -> "
                       "j[jl=%-2d] j[jh=%-2d] ex[k=%d]" % \
                               (idx, prefix, j, j+halfsize, k,
                                     jl, jh, ks,
                               ),
                               "end", bin(je)[2:], bin(je)[2:], bin(ke)[2:])
                k += tablestep
                idx += 1
        size *= 2

# run the demo <a name="ls009.mdwn_run_the"> </>
if __name__ == '__main__':
    demo()
```

DCT REMAP </>

DCT REMAP is RADIX2 only. Convolutions may be applied as usual to create non-RADIX2 DCT. Combined with appropriate Twin-butterfly instructions the algorithm below (written in python3), becomes part of an in-place in-registers Vectorized DCT. The algorithms work by loading data such that as the nested loops progress the result is sorted into correct sequential order.

```
# DCT "REMAP" scheduler to create an in-place iterative DCT. <a name="ls009.mdwn_dct_remap"> </>
# <a name="ls009.mdwn"> </>

# bits of the integer 'val' of width 'width' are reversed <a name="ls009.mdwn_bits_of"> </>
```

```
def reverse_bits(val, width):
    result = 0
    for _ in range(width):
        result = (result << 1) | (val & 1)
        val >>= 1
    return result


# iterative version of [recursively-applied] half-reversing <a name="ls009.mdwn_iterative_version"> </>
# turns out this is Gray-Encoding. <a name="ls009.mdwn_turns_out"> </>
def halfrev2(vec, pre_rev=True):
    res = []
    for i in range(len(vec)):
        if pre_rev:
            res.append(vec[i ^ (i>>1)])
        else:
            ri = i
            bl = i.bit_length()
            for ji in range(1, bl):
                ri ^= (i >> ji)
            res.append(vec[ri])
    return res


def iterate_dct_inner_halfswap_loadstore(SVSHAPE):
    # get indices to iterate over, in the required order
    n = SVSHAPE.lims[0]
    mode = SVSHAPE.lims[1]
    stride = SVSHAPE.lims[2]

    # reference list for not needing to do data-swaps, just swap what
    # *indices* are referenced (two levels of indirection at the moment)
    # pre-reverse the data-swap list so that it *ends up* in the order 0123..
    ji = list(range(n))

    levels = n.bit_length() - 1
    ri = [reverse_bits(i, levels) for i in range(n)]

    if SVSHAPE.mode == 0b01: # FFT, bitrev only
        ji = [ji[ri[i]] for i in range(n)]
    elif SVSHAPE.submode2 == 0b001:
        ji = [ji[ri[i]] for i in range(n)]
        ji = halfrev2(ji, True)
    else:
        ji = halfrev2(ji, False)
        ji = [ji[ri[i]] for i in range(n)]

    # invert order if requested
    if SVSHAPE.invxyz[0]:
        ji.reverse()

    for i, jl in enumerate(ji):
        y_end = jl == ji[-1]
        yield jl * stride, (0b111 if y_end else 0b000)

def iterate_dct_inner_costable_indices(SVSHAPE):
    # get indices to iterate over, in the required order
    n = SVSHAPE.lims[0]
    mode = SVSHAPE.lims[1]
    stride = SVSHAPE.lims[2]
    # creating lists of indices to iterate over in each dimension
    # has to be done dynamically, because it depends on the size
    # first, the size-based loop (which can be done statically)
    x_r = []
    size = 2
    while size <= n:
        x_r.append(size)
        size *= 2
    # invert order if requested
    if SVSHAPE.invxyz[0]:
        x_r.reverse()

    if len(x_r) == 0:
```

```
        return

    # start an infinite (wrapping) loop
    skip = 0
    z_end = 1 # doesn't exist in this, only 2 loops
    k = 0
    while True:
        for size in x_r:           # loop over 3rd order dimension (size)
            x_end = size == x_r[-1]
            # y_r schedule depends on size
            halfsize = size // 2
            y_r = []
            for i in range(0, n, size):
                y_r.append(i)
            # invert if requested
            if SVSHAPE.invxyz[1]: y_r.reverse()
            # two lists of half-range indices, e.g. j 0123, jr 7654
            j = list(range(0, halfsize))
            # invert if requested
            if SVSHAPE.invxyz[2]: j_r.reverse()
            # loop over 1st order dimension
            for ci, jl in enumerate(j):
                y_end = jl == j[-1]
                # now depending on MODE return the index.  inner butterfly
                if SVSHAPE.skip == 0b00: # in [0b00, 0b10]:
                    result = k  # offset into COS table
                elif SVSHAPE.skip == 0b10: #
                    result = ci # coefficient helper
                elif SVSHAPE.skip == 0b11: #
                    result = size # coefficient helper
                loopends = (z_end |
                            ((y_end and z_end)<<1) |
                             ((y_end and x_end and z_end)<<2))

                yield (result * stride) + SVSHAPE.offset, loopends
                k += 1

def iterate_dct_inner_butterfly_indices(SVSHAPE):
    # get indices to iterate over, in the required order
    n = SVSHAPE.lims[0]
    mode = SVSHAPE.lims[1]
    stride = SVSHAPE.lims[2]
    # creating lists of indices to iterate over in each dimension
    # has to be done dynamically, because it depends on the size
    # first, the size-based loop (which can be done statically)
    x_r = []
    size = 2
    while size <= n:
        x_r.append(size)
        size *= 2
    # invert order if requested
    if SVSHAPE.invxyz[0]:
        x_r.reverse()

    if len(x_r) == 0:
        return

    # reference (read/write) the in-place data in *reverse-bit-order*
    ri = list(range(n))
    if SVSHAPE.submode2 == 0b01:
        levels = n.bit_length() - 1
        ri = [ri[reverse_bits(i, levels)] for i in range(n)]

    # reference list for not needing to do data-swaps, just swap what
    # *indices* are referenced (two levels of indirection at the moment)
    # pre-reverse the data-swap list so that it *ends up* in the order 0123..
    ji = list(range(n))
    inplace_mode = True
    if inplace_mode and SVSHAPE.submode2 == 0b01:
        ji = halfrev2(ji, True)
    if inplace_mode and SVSHAPE.submode2 == 0b11:
        ji = halfrev2(ji, False)
```

```
    # start an infinite (wrapping) loop
    while True:
        k = 0
        k_start = 0
        for size in x_r:           # loop over 3rd order dimension (size)
            x_end = size == x_r[-1]
            # y_r schedule depends on size
            halfsize = size // 2
            y_r = []
            for i in range(0, n, size):
                y_r.append(i)
            # invert if requested
            if SVSHAPE.invxyz[1]: y_r.reverse()
            for i in y_r:       # loop over 2nd order dimension
                y_end = i == y_r[-1]
                # two lists of half-range indices, e.g. j 0123, jr 7654
                j = list(range(i, i + halfsize))
                jr = list(range(i+halfsize, i + size))
                jr.reverse()
                # invert if requested
                if SVSHAPE.invxyz[2]:
                    j.reverse()
                    jr.reverse()
                hz2 = halfsize // 2 # zero stops reversing 1-item lists
                # loop over 1st order dimension
                k = k_start
                for ci, (jl, jh) in enumerate(zip(j, jr)):
                    z_end = jl == j[-1]
                    # now depending on MODE return the index.  inner butterfly
                    if SVSHAPE.skip == 0b00: # in [0b00, 0b10]:
                        if SVSHAPE.submode2 == 0b11: # iDCT
                            result = ji[ri[jl]]        # lower half
                        else:
                            result = ri[ji[jl]]        # lower half
                    elif SVSHAPE.skip == 0b01: # in [0b01, 0b11]:
                        if SVSHAPE.submode2 == 0b11: # iDCT
                            result = ji[ri[jl+halfsize]] # upper half
                        else:
                            result = ri[ji[jh]] # upper half
                    elif mode == 4:
                        # COS table pre-generated mode
                        if SVSHAPE.skip == 0b10: #
                            result = k # cos table offset
                    else: # mode 2
                        # COS table generated on-demand ("Vertical-First") mode
                        if SVSHAPE.skip == 0b10: #
                            result = ci # coefficient helper
                        elif SVSHAPE.skip == 0b11: #
                            result = size # coefficient helper
                    loopends = (z_end |
                                ((y_end and z_end)<<1) |
                                 ((y_end and x_end and z_end)<<2))

                    yield (result * stride) + SVSHAPE.offset, loopends
                    k += 1

                # now in-place swap
                if inplace_mode:
                    for ci, (jl, jh) in enumerate(zip(j[:hz2], jr[:hz2])):
                        jlh = jl+halfsize
                        tmp1, tmp2 = ji[jlh], ji[jh]
                        ji[jlh], ji[jh] = tmp2, tmp1

            # new k_start point for cos tables( runs inside x_r loop NOT i loop)
            k_start += halfsize


# python "yield" can be iterated. use this to make it clear how <a name="ls009.mdwn_python_yield"> </>
# the indices are generated by using natural-looking nested loops <a name="ls009.mdwn_the_indices"> </>
def iterate_dct_outer_butterfly_indices(SVSHAPE):
    # get indices to iterate over, in the required order
    n = SVSHAPE.lims[0]
    mode = SVSHAPE.lims[1]
```

```
stride = SVSHAPE.lims[2]
# creating lists of indices to iterate over in each dimension
# has to be done dynamically, because it depends on the size
# first, the size-based loop (which can be done statically)
x_r = []
size = n // 2
while size >= 2:
    x_r.append(size)
    size //= 2
# invert order if requested
if SVSHAPE.invxyz[0]:
    x_r.reverse()

if len(x_r) == 0:
    return

# I-DCT, reference (read/write) the in-place data in *reverse-bit-order*
ri = list(range(n))
if SVSHAPE.submode2 in [0b11, 0b01]:
    levels = n.bit_length() - 1
    ri = [ri[reverse_bits(i, levels)] for i in range(n)]

# reference list for not needing to do data-swaps, just swap what
# *indices* are referenced (two levels of indirection at the moment)
# pre-reverse the data-swap list so that it *ends up* in the order 0123..
ji = list(range(n))
inplace_mode = False # need the space... SVSHAPE.skip in [0b10, 0b11]
if SVSHAPE.submode2 == 0b11:
    ji = halfrev2(ji, False)

# start an infinite (wrapping) loop
while True:
    k = 0
    k_start = 0
    for size in x_r:           # loop over 3rd order dimension (size)
        halfsize = size//2
        x_end = size == x_r[-1]
        y_r = list(range(0, halfsize))
        # invert if requested
        if SVSHAPE.invxyz[1]: y_r.reverse()
        for i in y_r:       # loop over 2nd order dimension
            y_end = i == y_r[-1]
            # one list to create iterative-sum schedule
            jr = list(range(i+halfsize, i+n-halfsize, size))
            # invert if requested
            if SVSHAPE.invxyz[2]: jr.reverse()
            hz2 = halfsize // 2 # zero stops reversing 1-item lists
            k = k_start
            for ci, jh in enumerate(jr):   # loop over 1st order dimension
                z_end = jh == jr[-1]
                if mode == 4:
                    # COS table pre-generated mode
                    if SVSHAPE.skip == 0b00: # in [0b00, 0b10]:
                        if SVSHAPE.submode2 == 0b11: # iDCT
                            result = ji[ri[jh]] # upper half
                        else:
                            result = ri[ji[jh]]        # lower half
                    elif SVSHAPE.skip == 0b01: # in [0b01, 0b11]:
                        if SVSHAPE.submode2 == 0b11: # iDCT
                            result = ji[ri[jh+size]] # upper half
                        else:
                            result = ri[ji[jh+size]] # upper half
                    elif SVSHAPE.skip == 0b10: #
                        result = k # cos table offset
                else:
                    # COS table generated on-demand ("Vertical-First") mode
                    if SVSHAPE.skip == 0b00: # in [0b00, 0b10]:
                        if SVSHAPE.submode2 == 0b11: # iDCT
                            result = ji[ri[jh]]        # lower half
                        else:
                            result = ri[ji[jh]]        # lower half
                    elif SVSHAPE.skip == 0b01: # in [0b01, 0b11]:
                        if SVSHAPE.submode2 == 0b11: # iDCT
```

```
                    result = ji[ri[jh+size]] # upper half
                else:
                    result = ri[ji[jh+size]] # upper half
            elif SVSHAPE.skip == 0b10: #
                result = ci # coefficient helper
            elif SVSHAPE.skip == 0b11: #
                result = size # coefficient helper
        loopends = (z_end |
                    ((y_end and z_end)<<1) |
                     ((y_end and x_end and z_end)<<2))

        yield (result * stride) + SVSHAPE.offset, loopends
        k += 1

    # new k_start point for cos tables( runs inside x_r loop NOT i loop)
    k_start += halfsize
```

REMAP selector </>

Selecting which REMAP Schedule to use is shown by the pseudocode below. Each SVSHAPE 0-3 goes through this selection process.

```
if SVSHAPEn.mode == 0b00:             iterate_fn = iterate_indices
if SVSHAPEn.mode == 0b10:             iterate_fn = iterate_preduce_indices
if SVSHAPEn.mode in [0b01, 0b11]:
  # further sub-selection
  if SVSHAPEn.ydimsz == 1:            iterate_fn = iterate_butterfly_indices
  if SVSHAPEn.ydimsz == 2:            iterate_fn = iterate_dct_inner_butterfly_indices
  if SVSHAPEn.ydimsz == 3:            iterate_fn = iterate_dct_outer_butterfly_indices
  if SVSHAPEn.ydimsz in [5, 13]:      iterate_fn = iterate_dct_inner_costable_indices
  if SVSHAPEn.ydimsz in [6, 14, 15]:  iterate_fn = iterate_dct_inner_halfswap_loadstore
```

[[!tag opf_rfc]]