Orin PVA VPU Programmer's Guide

Programmer Reference

SWE-PVA-076-PGRF | May 2025

Review Status

Title	Orin PVA VPU Programmers Guide-P
Author	chingh
Revision	May 29, 2025
Reviewers	jsankaran, nshigihalli
Status	Completed
Reviewed in	NVIDIA CR #23918

Document History

SWE-PVA-076-PGRF

Version	Date	Authors	Description of Change
0.5.1	May 15, 2025	chingh	> Branched from Orin_PVA_VPU_Programmings_Guide.docx into Orin_PVA_VPU_Programmers_Guide_pub.docx. > Restarted version number from 0.5.1, removed Proprietary/NDA tagging and cleaned up documentation control section. > Revised interpolated lookup example code to use PVA SDK Sampler API functions, in 8.5.4.
0.5.2	May 22, 2025	chingh	> Cindy Wilkinson imported into new template > Fixed footer and fixed some English mistakes in Chapters 1 through 4 > Refined description of partitioning a task into sub-tasks for concurrent prefetch and execution, in 4.1.2 > Removed reference to VPU config DMA feature for instruction cache invalidation for an address range, since it's not supported by PVA SDK, in 4.1.3.
0.5.3	May 22, 2025	chingh	> Revised DLUT programming example to one included in Thor PVA VPU Programmer's Guide, since the Thor version has gone through review involving PVA SW team members, in 8.5.4.
0.5.4	May 23, 2025	chingh	> Propagated applicable revisions suggested from Thor PVA VPU Programmer's Guide review (#15825) to the Orin version, various sections.
0.5.5	May 28, 2025	chingh	> Added review table > Per CR comments, remove references to PVA User's Guide, in 2, 2.2 > Per CR comments, fixed VPS block diagram, in 2.3 > Per CR comments, removed references to shallow pipeline & IA models, in 2.4

Version	Date	Authors	Description of Change
			> Per CR comments, elaborated Superbanks A/B/C, in 5.3 > Fixed code indentation issues, in chapter 8 sections. > Fixed instruction report white spaces so that execution count and cycle count numbers line up vertically, in chapter 8 sections. > Per CR comments, fixed instruction detail table indentation and width, in chapter 9 sections
0.5.6	May 28, 2025	chingh	> Fixed typos and issues identified from Thor version review, various sections
1.0	May 29, 2025	chingh	> CR completed, administrative update to fill review status as Completed

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

1.1 Document Scope

This document serves as a Programmer's Guide for PVA VPU. It covers VPU processor architecture, instruction set overview, example code, and instruction details.

1.2 References

1.2.1 Related External Documents

- PVA SDK Documentation

1.2.2 Related NVIDIA-Internal Documents

- PVA VPS IAS
- PVA Cluster IAS
- PVA DMA IAS
- PVA L1 RAMIC IAS
- PVA VPS MAS
- PVA DLUT MAS

1.3 Glossary and Acronyms

CV	Computer vision, field of study and application to recover 3D and motion information from camera views.
PVA	Programmable vision accelerator, a unit in Orin that accelerates computer vision algorithms in autonomous driving use cases, includes VPU, DMA, and Cortex R5 RISC processor.
SEC	Safety and Event Control at PVA top level. It collects safety error events in PVA, logs, aggregates, and forwarded as interrupts to the Cortex R5 processor.
VPU	Vector processing unit, the main data processing engine in PVA.

VMEM	VPU vector memory, the local/L1 data memory for VPU, also shared with DMA and DLUT
DMA	Direct memory access, a hardware block in charge of copying data between local memory and some other space in the system, which can be on-chip memory or system memory/DDR.
DLUT	Decoupled lookup unit
VPS	VPU subsystem, including VPU, its I-cache, DLUT and VMEM
Host1X	Command and synchronization unit that works with CPU, image/video processing and computer vision engines
ISP	Image Signal Processor, processes camera images
VIC	Video and Image Compositor, capable of affine/perspective image transformation and format conversion
OFA	Optical Flow Accelerator, capable of dense optical flow and stereo disparity
DLA	Deep Learning Accelerator

Chapter 2. Architecture Overview

A high-level overview of PVA, DMA, and VPS architecture is given in this chapter. For more in-depth coverage of PVA architecture and DMA programming details, please consult PVA SDK documentation.

2.1 PVA

PVA (programmable vision accelerator) is a computer vision (CV) processor targeting Autonomous Driving (AD) applications, including camera, LiDAR, RADAR processing and sensor fusion. PVA includes a control processor, Cortex R5, 2 copies of vector processing subsystems (VPS) as data processing engines, and 2 copies of directed memory access (DMA) as data movement engines. Orin PVA also includes an L2 SRAM memory to be shared between the 2 sets of VPS and DMA.

The Cortex R5 processor interacts with other SOC components (for example, ISP, VIC, OFA, DLA) through Host1X for control and synchronization at the subframe-application level. R5 configures the VPUs and DMAs at the task level.

The VPUs act like coprocessors in system-level programming model. For each VPU task, R5 configures DMA, optionally prefetches VPU program into VPU I-cache, and kicks off each VPU-DMA pair to process a task that runs for typically hundreds of micro-seconds to a few mini-seconds. Each VPU and DMA pair synchronize between themselves on tile granularity, and there are typically tens to hundreds of tiles per task.

For Orin, the second generation of PVA, we have one PVA having 2 VPUs, each VPU having 2 symmetrical vector functional units of 384-bit data path each.

For memory operations we have 3x32x16-bit throughput, having 3 memory slots and 3 superbanks, each superbank comprising of 32 banks of 16-bit-wide memories, and each superbank can perform both read and write in the same clock cycle.

2.2 DMA

DMA moves data among external memory, PVA L2 memory, the 2 VMEMs (one in each VPS), R5 TCM (tightly coupled memory), DMA descriptor memory, and PVA-level config registers.

Orin DMA contains the following resources

- 16 channels, each channel can be configured to move data from a source to a destination. The 16 channels work in parallel and can be optionally coordinated through programming.
- 64 descriptors, each descriptor includes up to 5 dimensions to advance source/destination address pointers. Descriptors can work in parallel or in sequence through programming.
- A set of internal buffers (ADB and VDB) to be allocated among channels. ADB, AXI data buffer, is for storing data read from the external memory controller temporarily, and VDB, VMEM data buffer, is for storing data read from the VMEM temporarily.

Please consult PVA SDK documentation for additional details in DMA programming.

2.3 VPS

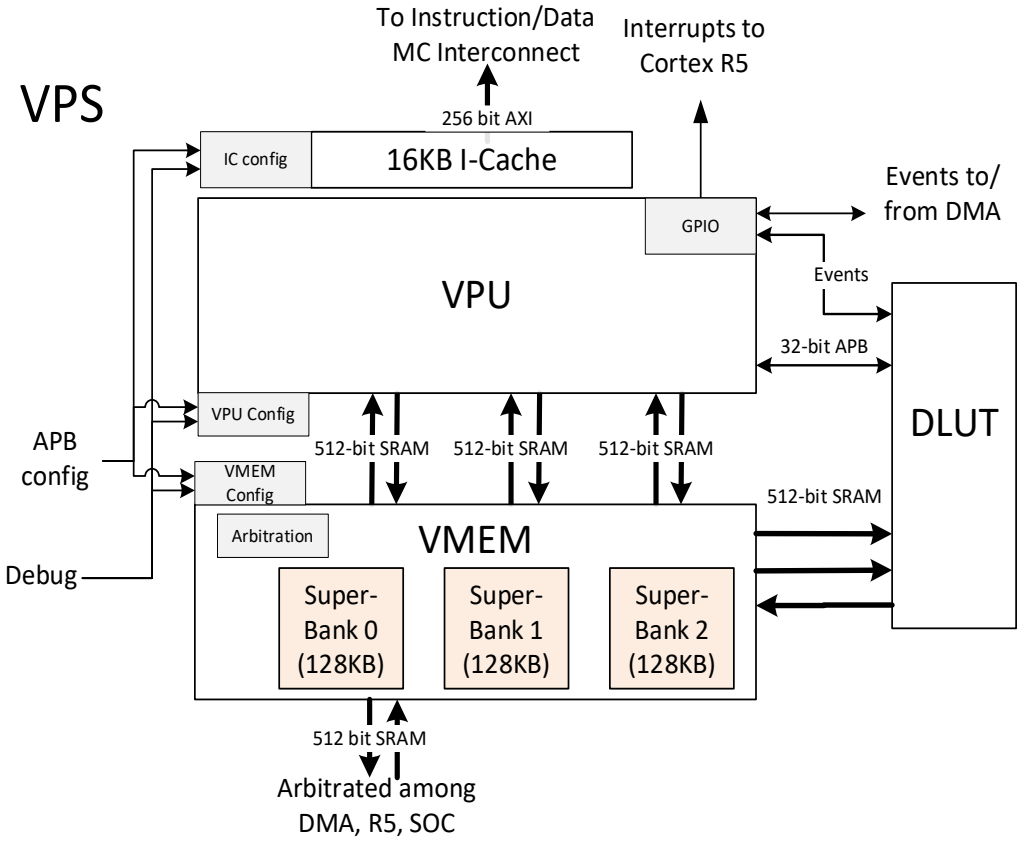
The VPU Subsystem (VPS) consists of the following major components:

- VPU core, the processor and main block of VPS.
- VPU instruction cache (I-cache) supplies instruction words to VPU and maintains temporary instruction storage, with prefetch/invalidation support and with interface to the system memory through MC interconnects.
- VPU vector memory (VMEM) houses data memory and supports various complex memory access functionalities, including transposition, table lookup, histogram, and vector addressed stores. It also supports accesses from outside-VPS hosts like DMA and R5, to allow data exchange with R5 and other system-level components.
- DLUT, decoupled lookup coprocessor, offloads lookup and interpolation tasks from VPU

Each major component will be described in more detail in subsequent chapters.

The following block diagram of VPS shows the major components in VPS and how they are connected.

Figure 1. VPU Subsystem (VPS) block diagram



2.4 VPU Processor Models and Differences

To facilitate model development as well as application software development, a number of VPU processor models have been constructed.

The most accurate model is the deep pipeline (Working) model. The VPU working model instruction set simulator (ISS) shall be cycle accurate with VPU processor inside Orin silicon.

There is a Native compilation model generated by the ASIP tool suite from the shallow pipeline model. It is mostly an application development platform. It is a collection of header files and C library that allows application code to be compiled in generic (thus named Native) environments, including Linux GCC and Windows Microsoft Visual Studio. It is functionally accurate with hardware for math operations. In memory operations it is mostly functionally accurate with hardware, but there are exceptions.

Because Native is compiled in a generic compute platform, there is no hard limit in data memory footprint, so is useful for early-stage software development. In this platform VPU code can access almost unlimited amount of memory, to directly process a whole frame of image, as opposed to processing one tile at a time through DMA.

Note that, depending on the physical memory size of the compute platform it is run on, large enough memory usage in Native simulation may still lead to excessive thrashing and slowdown.

Differences in behavior between Native compilation environment and final product, which is deep pipeline ISS and silicon are:

- There is no notion of clock cycles in Native compilation, thus Time Stamp Counter is not functional.
- There is no forced memory address alignment to 16-bit/32-bit with load/store of short/int types (see Memory Address Alignment).
- There is no forced memory address alignment to 512-bit with lookup, histogram, or vector-addressed stores (see Memory Address Alignment).
- There is no forced memory address alignment to 512-bit with agen circular buffer feature (see Circular Buffer Addressing).
- General purpose input and output (GPIO) in Native is non-functional, toggling GPO ignored and reading GPI returns 0. In ISS, the subset of GPIO pins that connect to the decoupled lookup coprocessor (DLUT) are functional for interaction between VPU and DLUT.

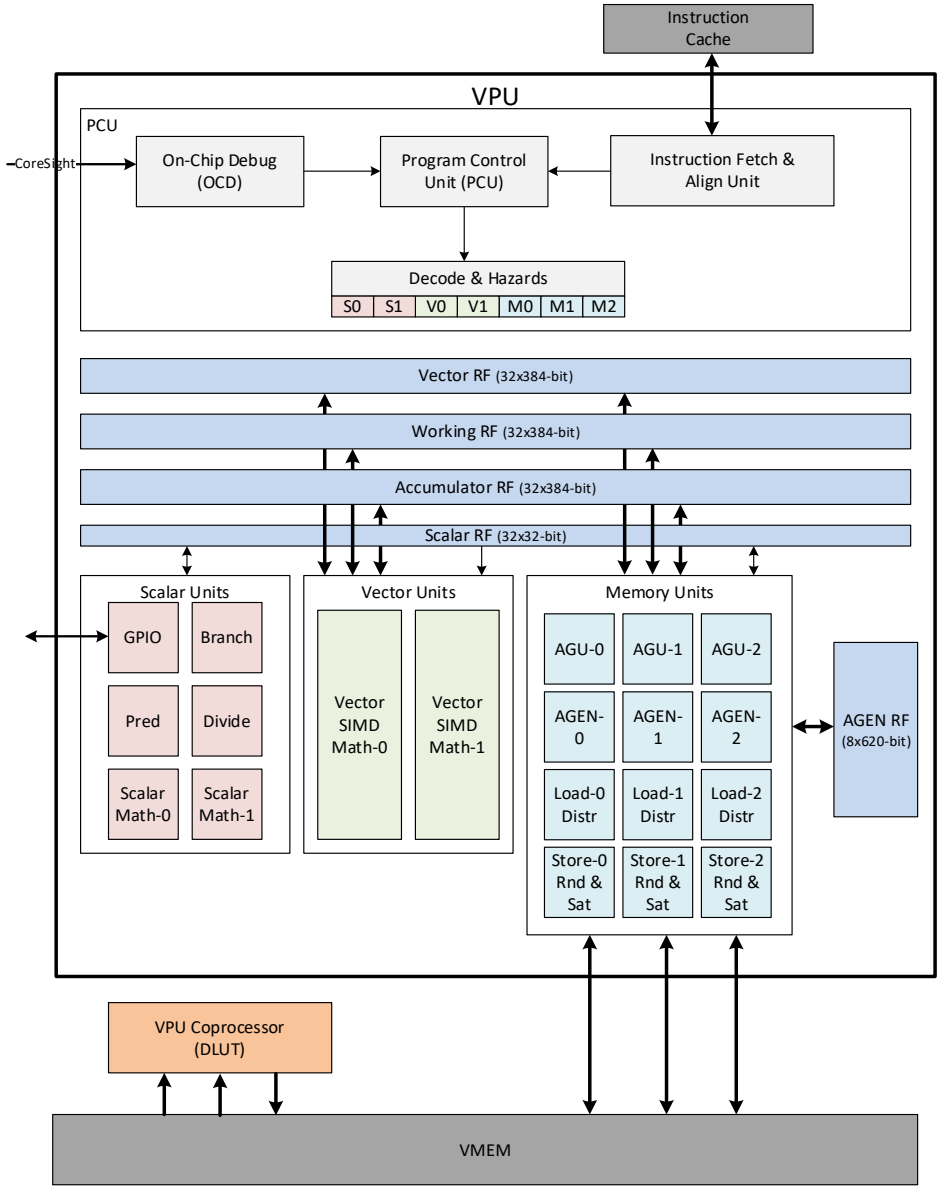
In working model ISS, the decoupled lookup table coprocessor (DLUT) is simulated functionally accurately (bit-exact), but is currently only cycle approximate, NOT cycle-accurate.

Although PVA-level simulator incorporating R5, DMA, VPU, DLUT component simulators does incorporate budgetary system-level latency, for example external memory controller latency, but it does not model components outside PVA so cannot predict actual latency. Thus, PVA-level simulation, even when incorporating cycle-accurate VPU working ISS, is NOT cycle-accurate with silicon when the simulation involves outside-VPU interactions, for example, reading from or writing to external memory.

Chapter 3. VPU Core

3.1 Block Diagram

Figure 2. VPU core block diagram



The VPU core is a vector SIMD VLIW DSP optimized for computer vision. It fetches instructions through the I-cache, and accesses data through the vector memory (VMEM). Major components inside VPU core are

- Processor front end including config/status interface and PC control
- Register files
- Scalar unit with two scalar ALUs
- Vector unit with two vector ALUs
- Memory unit with 3 load/store units and address generators

3.2 Processor Front End

The VPU instruction format is a 7-way VLIW, consisting of:

- 2 scalar operation slots (s0/s1)
- 2 vector operation slots (v0/v1)
- 3 memory slots (m0/m1/m2)

Each instruction is encoded in 32-bit, and the instruction encoding supports variable-length instructions, thus each execution packet contains between 1 and 7 32-bit words. The compressed instruction stream is decompressed to full 7 instructions per packet and dispatched to the scalar, vector, and memory units.

Example of compressed instruction packet:

```
ADD R1, R2, R3 || LDW *R4[20], R5
```

Corresponding decompressed instruction packet:

```
ADD R1, R2, R3 || s1_NOP || v0_NOP || v1_NOP || LDW *R4[20], R5 || m1_NOP || m2_NOP
```

The front end of the processor includes an interface to the instruction cache, 2-level hardware loops, loop instruction buffer, and fetch/decode stages of the processor pipeline.

The front end includes illegal instruction detection, both while expanding compressed variable-length instruction packet into full 7-instruction packet using leading few bits of each 32-bit instruction, and while decoding entire 32-bit instruction in scalar/vector/memory units.

3.3 Register Files

The following register files are in the VPU core:

- Scalar register file (Scalar RF): 32 entries x 32-bit
- Predicate register file (Predicate RF): 16 entries x 32-bit
- Main vector register file (VRF): 32 entries x 384-bit
- Working register file (WRF): 32 entries x 384-bit
- Extended accumulator register file (XARF): 32 entries x 512-bit

- Accumulator register file (ARF): 32 entries x 384-bit, part of XARF

> Agen register file: 8 entries x 620-bit

The vector register files VRF, WRF and ARF support the following data types:

> Word: each 384-bit entry is logically partitioned into 8 lanes x 48-bit
> Halfword: each 384-bit entry is logically partitioned into 16 lanes x 24-bit
> Byte: each 384-bit entry is logically partitioned into 32 lanes x 12-bit

XARF register file supports

> Word: each 512-bit entry partitioned into 16 lanes x 32-bit (for VFilt4x2x2BBW, VDotP4BBW, VDotP4x2BBW instructions)
> Halfword: each 512-bit entry partitioned into 32 lanes x 16-bit (for VXNorAnd8x4x2 instruction)

Support of operations in various register files is tabulated as follows:

Table 1. Support of scalar/vector operations in register files

	Scalar RF	Predicate RF	VRF	WRF	ARF/XARF
Scalar math	Yes	Yes, as src or dst of a few			
Instruction level predication		Yes			
Per-lane predication		Yes	V0~V15		
Vop11/12			Yes	Yes	
Vop21	Yes, as src2		Yes	Yes	
Vop31	some, as src2		Yes	some	
Vop31_CA, MAC	some, as src2	Yes, P0~P15 as predicate	Yes, as src1, src2, dst	some, as src2	Yes, as dst, src3dst
FP	Yes		Yes	Yes	
Load destination	Yes		Yes	Yes	
Store source	Yes		Yes		Yes

3.4 Scalar Unit

The scalar unit supports conventional scalar RISC instruction set, executing up to 2 scalar operations per cycle.

32-bit integer/fixed-point as well as 16/32-bit floating-point Add, Sub, Mul, MAdd, compare operations are supported through instructions. Some FP32 math functions (square root, reciprocal, reciprocal of square root, exp2, log2, sin, cos, tanh) and various FP/INT conversions are supported as well.

3.5 Vector Unit

The vector unit executes up to 2 vector math instructions per cycle. Various integer arithmetic and logic operations are implemented in the vector unit, with support for Byte (extended to 12-bit), Halfword (extended to 24-bit) and Word (extended to 48-bit) data types. Bitwise logic operations are also supported.

In addition to conventional arithmetic/logic operations, some larger or complex operations (e.g., 3-input min/max/median) as well as FP32/FP16 operations Add, Sub, Mul, MAdd, and compare are supported. Some FP32 math functions (square root, reciprocal, reciprocal of square root, exp2, log2, sin, cos, tanh) and various FP/INT conversions are supported as well.

3.6 Load/Store Unit

The load/store unit supports up to 3 load/store instructions per cycle. Word, Halfword, Byte, and selected promotion/demotion options are supported. For load, both signed/unsigned flavors are supported. Source and destination can be single scalar register, double scalar register, single vector register, or double vector register. Quad-vector-register store is also available to facilitate key filtering benchmarks. Load/store unit also supports various data distributions.

In Orin we have added load-and-permute instructions to manipulate/reorganize data from a double vector in memory to a double vector register destination in any permutation pattern. This enables various data access patterns to be efficiently carried out through such instructions.

In general, we would like memory transactions from load/store instructions to be executed in order through memory dependency checking and dynamic stalling. The VPU has a rich set of load/store features, and for some features it is cost prohibitive to implement the dependency checking. Scalar load/store instructions as well as consecutive-location vector load/store are included in the dependency checking, so they are guaranteed to execute in order. Transposing load/store, parallel table lookup, parallel histogram, and vector addressed stores are excluded in the checking, so they are not guaranteed to execute in order. A MemFence instruction is available to serialize memory transactions that hardware dependency checking does not cover. See Memory Coherency for additional details.

Chapter 4. VPU Instruction Cache (I-Cache)

4.1 Overview

The VPU Instruction Cache (I-cache) supplies instruction data to the VPU when requested, requests missing instruction data from system memory, and basically maintains temporary instruction storage for the VPU. It also implements the prefetch command to reduce cache misses, as well as the invalidation command as needed for error correction and debug.

Having an instruction cache allows for large total code size to be stored in the system memory, while having small physical memory footprint for area efficiency.

4.2 Functionality

The following table captures the characteristics of the I-cache.

Table 2. VPU I-cache characteristics

Characteristic	Configuration
Capacity	16KB
Associativity	2-way
Instruction width	256-bit
Instruction alignment	256-bit
Block size	128 bytes
Replacement policy	LRU
Write policy	None (I-cache read only)
Hit under miss (nonblocking, if/when VPU requests another instruction word that's available, go ahead and return hit)	No, fetch interface is in-order, so after a miss, if following fetch request hits, it's not possible to indicate so.
Miss under miss (if/when VPU requests another instruction word that's unavailable, request for that cache line as well)	Yes (request/ready pipelining allows following fetch request to be conveyed, and if it's a miss involving another cache line, request can be sent out as well)
Hit latency	2 cycles

Characteristic	Configuration
Prefetch (software request to fetch cache lines ahead of execution)	Yes, up to full cache capacity in a single R5/VPU interaction. Depending on outstanding transaction allocation may request in batches
Interface for misses	256-bit AXI, AR, R channels only
Prefetch request from R5 and VPU	Yes, will have separate config register entries for concurrency
Prefetch and fetch concurrency	Yes, giving fetch higher priority
ECC single error correction	Yes, corrected on the fly and sent back to VPU
ECC single/double error detection	Single errors are corrected but correction not written back to cache; single error handling software should invalidate cache line to initiate refetch when the line is requested again. Double errors are detected but not corrected.
Invalidation from R5	Yes, configurable address range
Invalidation from VPU	Yes, configurable address range

4.2.1 Preemption

The VPU fetch/align unit fetches ahead of execution, and thus may request some instruction data, but in the next few cycles branches to another PC location that renders the previous request unnecessary. In such cases, the fetch/align unit cancels a previous request and issues a request for the new PC location. This feature is called preemption and is particularly useful when one of the no-longer-needed requests triggered a cache miss. VPU execution would be stalled if hardware does not have the capability to cancel such requests.

The I-cache handles preemption by clearing the preempted request from the pipeline. In case the preempted request has been sent to the MC, the MC read request is not affected, and returned data from MC would be written to a cache line normally, possibly evicting instruction data on that cache line.

4.2.2 Prefetch

Prefetch capability is provided to both the R5 and the VPU. They use separate register entries and command queues to avoid any race conditions, although SW on both sides should be coherent and not attempt to request prefetch or invalidation at the same time.

When a program for a task fits the I-cache, the R5 should prefetch the whole task, then start VPU at its task PC. The VPU may initially see instruction-cache misses until the whole task is loaded.

When program for a task does not fit the I-cache, we recommend that the task code is partitioned into subtasks for concurrent execution and prefetch. Given the 2-way set

associativity characteristics of instruction cache and the cache capacity of 16KB, each subtask should ideally be under 8KB.

The R5 should prefetch just the first subtask before starting the VPU task. VPU code for a subtask should prefetch the next subtask at the appropriate time so that ideally the prefetch is hidden behind execution.

R5 SW should not start requesting prefetch for VPU's next task until VPU has completed its current task and is idle. This also ensures prefetches from R5 and VPU do not contend for cache lines.

4.2.3 Invalidation

The I-cache supports two concurrent invalidation interfaces through the config registers; one designated for R5 and the other designated for VPU. Each invalidation interface can selectively invalidate an address range or the whole cache.

Invalidating the whole cache by VPU is supported via GPO sideband signaling (see Summary of GPI/GPO Signals). Invalidating an address range by VPU is currently not supported by software.

Invalidation can be used to provide a clean slate for I-cache at the beginning of every task, and the R5 should be the one invalidating the entire cache.

Invalidation can also be used to handle I-cache single error detection. When a single error is detected (when the VPU requesting instruction(s) that contains an error), the I-cache sends the corrected instruction data back to VPU but does not write the corrected instruction data back to the I-cache's memory. The R5 software handling I-cache single error detection should invalidate the cache line to cause the line to be refreshed from DRAM, which we assume is ECC protected as well and contains the correct program data.

Invalidation is also needed for VPU debug software breakpoint, which is implemented by substituting code data at selected break point with SWBRK, software breakpoint instruction. As I-cache is read-only, code change is implemented by altering the code image in external memory and invalidating the corresponding cache line.

4.2.4 ECC (Single-Bit-Error-Correct Double-Bit-Error-Detect)

To reduce fault rate against memory cell transient faults, the VPU I-cache is protected by single bit error correction, double bit error detection scheme.

A single-bit error within a 256-bit instruction word is corrected on the fly, and an error event is sent to the PVA top-level SEC block, and from there it is forwarded to R5 and optionally to system-level error collator.

A double-bit error within a 256-bit instruction word is detected but not corrected. An error event is sent to the PVA top-level SEC block, and from there it is forwarded to R5

and optionally to system-level error collator. The erroneous instruction word is return to VPU, which continues to be executed.

Optionally, I-cache can be configured to suspend upon detection of double bit error, until R5 software comes in to query I-cache for the error and reset VPS. This feature may be useful during software development phase to differentiate RAM soft error from other error sources.

Chapter 5. VPU Vector Memory (VMEM)

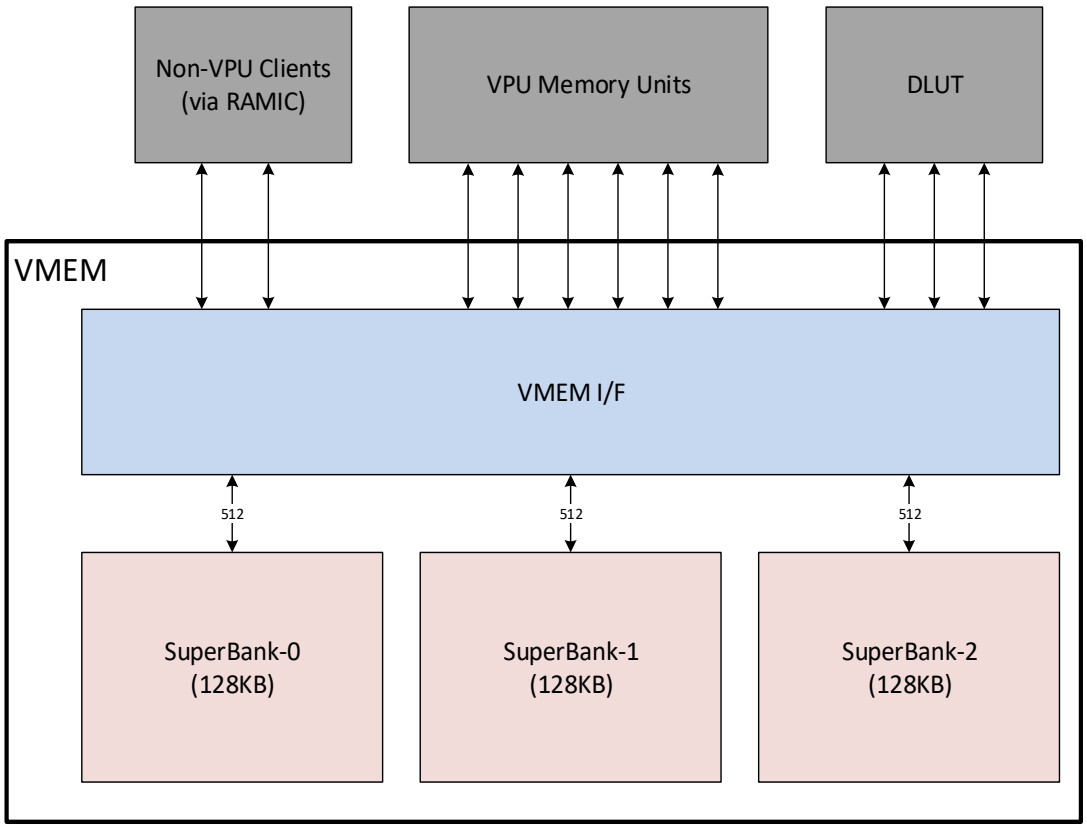
5.1 Overview

VPU vector memory (VMEM) houses local data memory for VPU to access so it can implement various image processing and computer vision algorithms efficiently. VMEM supports various complex memory access patterns from VPU, including consecutive read/write of various lengths, transposition, table lookup, histogram, vector addressed stores. It also supports accesses from outside-VPS hosts like DMA and R5, to allow data exchange with R5 and other system-level components.

VMEM includes VMEM I/F arbitration block and three VMEM superbanks of 128KB each. Each superbank incorporates dual port memory and supports one read AND one write per cycle. VMEM I/F arbitrates reads and writes separately for each superbank.

The VPU vector memory block diagram is as follows.

Figure 3. VMEM block diagram



5.2 VMEM Interface (VMEM I/F)

The VMEM I/F block performs arbitration among VPU load/store and external requests (including DMA), and handles histogram read-modify-write requests. Memory arbitration is carried out in superbank granularity and parallel between reads and writes. We have a fixed priority scheme as follows:

Read priority (highest to lowest)

- External (including DMA) read request-high
- VPU M0 load (including table lookup and histogram read)
- VPU M1 load
- VPU M2 load
- Stream0 read (DLUT lookup)
- Stream1 read (DLUT index/config)
- External (including DMA) read request-low

Write priority (highest to lowest)

- Histogram write
- External (including DMA) write request-high
- VPU M0 store
- VPU M1 store
- VPU M2 store
- Stream0 write (DLUT output)
- External (including DMA) write request-low

We have VPU load/store prioritized over stream read/write in VMEM arbitration. Stream read/write are driven by coprocessors. Normally coprocessors should have stream buffers so performance may not be affected by occasional stalls. In comparison, arbitration loss in VPU is likely to lead to performance loss.

Ideally, programmers should allocate memory objects to avoid VPU processing and coprocessor processing to compete for any superbank read/write. When that is not possible, programmers should consider coprocessor VMEM traffic and allocate memory objects to minimize VMEM contentions.

The cases when there are read(s) and write(s) in near execution packets to near address ranges in the memory are governed by memory coherency handling and are discussed separately in Memory Coherency. Here we are discussing VMEM arbitration for memory read/write transactions being executed at the same clock cycle.

Multiple memory transactions at the same clock cycle and going to the same superbank are executed sequentially following the above arbitration priority, when they are all reads or all writes. Mixed read/write cases (in the same execution packet) are:

- RW: Execute both in parallel, read will return the previous value
- RWW: Carry out the read and the first write in parallel, then the second write. The read will return the previous value,
- RRW: Carry out the first read, then the second read and the write in parallel. Both reads return the previous value.

A 2-bit QoS signal is sent with each external request, and the QoS is translated into a time-out count via VMEM config registers. Each external request is initially assigned to the external-low priority. If/when the request waits out the time-out count, it's escalated to the external-high priority, which prompts it to be served at next available cycle, thus ensuring some (configurable) minimal BW to VMEM for each QoS level.

The VPU supports memory accesses (table lookup, histogram, vector-addressed store, transposing load/store) that can potentially span a large address range. As each memory access is routed to a selected superbank based on the base address, no single memory access can straddle multiple superbanks.

5.3 VMEM Superbanks

The three memory superbanks appear as three memory regions in the VPU memory map, differentiated by high address bits to allow programmers to allocate, based on memory footprint and BW needs.

One simple way to allocate VMEM superbanks and avoid contention is

Superbank A write = DMA

Superbank A read = VPU

Superbank B read/write = VPU

Superbank C write = VPU

Superbank C read = DMA

This allows DMA to move data from system memory to Superbank A. VPU code would read that data for processing, and can use Superbank B for intermediate outcome, and write final outcome to Superbank C. DMA would then move data from Superbank C to system memory. The DMA input and output buffers can be ping-ponged to allow simultaneous read/write by VPU and DMA, without causing any contention.

This ideal, contention-free allocation scheme is only possible when DLUT is not involved, and DMA input/output buffer as well as intermediate buffers fit the 3 superbanks respectively.

When the buffer sizing does not work out, or when DLUT is involved, one will need to allocate buffers among superbanks carefully to minimize contention among the VPU, DMA and DLUT.

Each superbank has 128KB of capacity each. Each superbank sits in 256KB of space to allow for future expansion. 1 MB is allocated for the 3 superbanks (384KB total capacity). Address aliasing in the 1 MB space is as shown in the following table.

Table 3. VMEM address map

Byte address	Memory	Primary/Alias
0x00000 ~ 0x1FFFF	Superbank A first 128KB	Primary
0x20000 ~ 0x3FFFF	Superbank A second 128KB	Alias
0x40000 ~ 0x5FFFF	Superbank B first 128KB	Primary
0x60000 ~ 0x7FFFF	Superbank B second 128KB	Alias
0x80000 ~ 0x9FFFF	Superbank C first 128KB	Primary
0xA0000 ~ 0xBFFFF	Superbank C second 128KB	Alias
0xC0000 ~ 0xDFFFF	Superbank C third 128KB	Alias
0xE0000 ~ 0xFFFFF	Superbank C last 128KB	Alias

Note: Address aliasing is a side effect of address decoder logic and should not be taken advantage of in the software, as it is possible to set up address watch point via debugger to detect out-of-valid-range memory read/write and trigger error interrupts to PVA-top Cortex R5 processor.

Future generation hardware may change physical memory sizes and memory address mapping. Best practice for VPU software is to use A/B/C memory region naming (for example, `chess_segment(A/B/C)`) instead of hard-coding memory addresses, and to avoid using the alias memory regions.

5.4 Memory Banking and Read/Write Access Patterns

Each VMEM superbank consists of 32 banks of 16-bit wide RAMs. Each of the 32 memory banks are independently addressable per clock cycle. This enables a rich set of access patterns:

- Read/write one byte on any byte alignment
- Read/write one 16-bit half-word on any half-word alignment
- Read/write one 32-bit word on any word alignment
- Read/write 8 or 16 consecutive 32-bit words from any half-word alignment.
- Read/write 16, 24 or 32 consecutive 16-bit half-words from any half-word alignment
- Read/write 32 consecutive 8-bit bytes from any byte alignment
- Read/write 64 consecutive 8-bit bytes from any half-word alignment (starting odd byte is not supported, and shall be forcefully aligned to an even byte)
- Read/write in various transposed addressing patterns.
- Read/write independent memory rows in each 16-bit bank, leveraged by parallel table lookup, parallel histogram, and vector addressed store.

Various transposed load/store options, parallel table lookup, histogram, and vector addressed store options are discussed later. This is just describing access patterns from VMEM hardware capability point of view.

Example access patterns are shown in the following figures.

Figure 4. VMEM access pattern examples for consecutive accesses

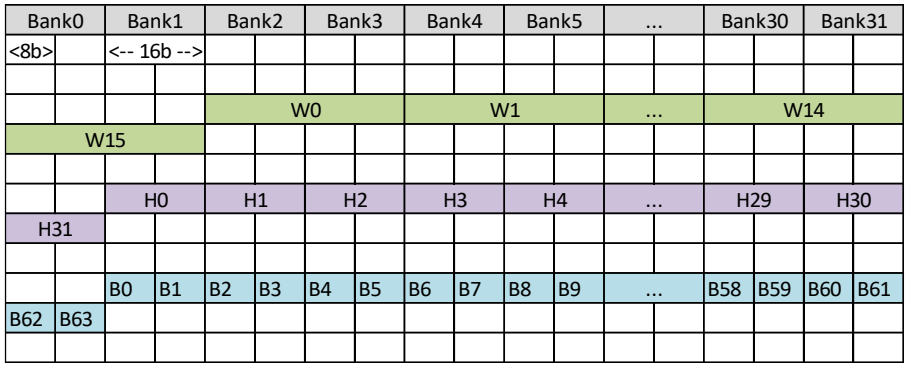


Figure 5. VMEM transposed access pattern examples

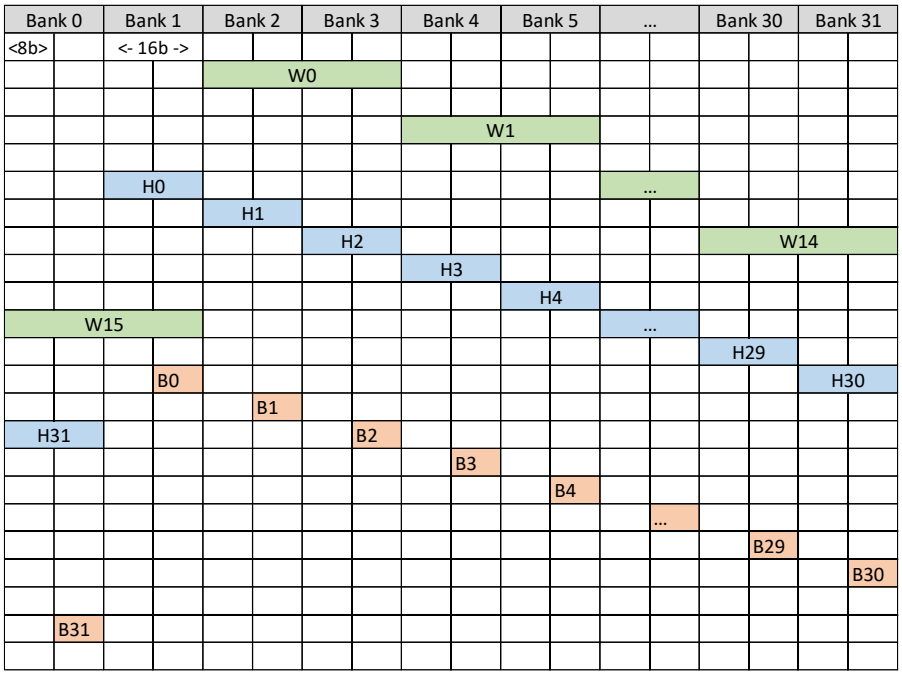
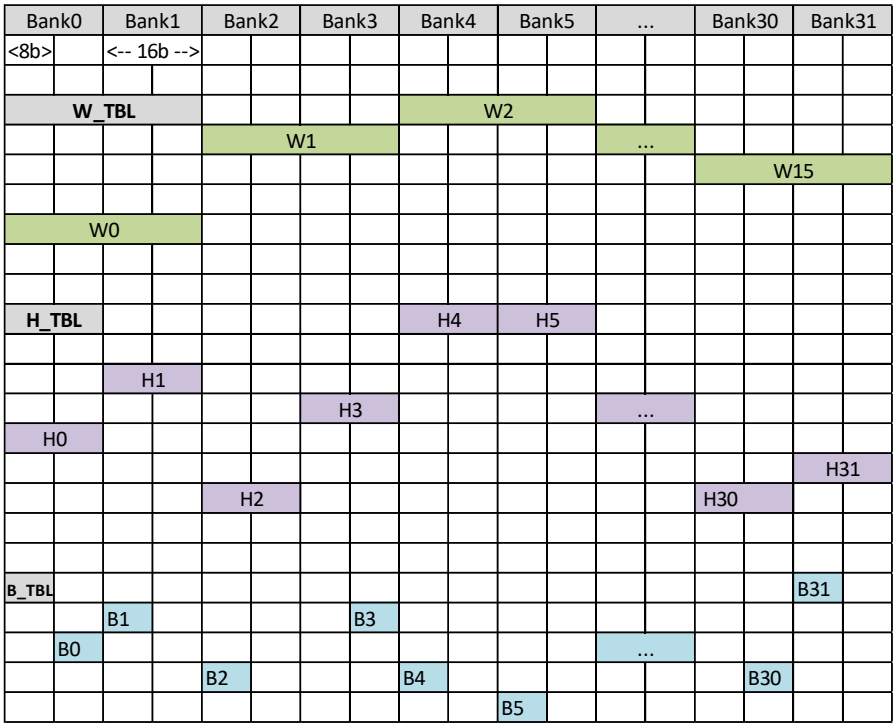


Figure 6. VMEM access pattern examples for parallel table lookup and histogram



5.5 Load Data Cache

Each VMEM memory bank contains a load data cache for power reduction. VPU processor pipeline including load timing accommodate both cache-hit and cache-miss cases. The cache can be enabled or disabled (see 9.6.16 for details) on a superbank granularity.

The load cache, when enabled, only caches the following types of memory read transactions:

- Single/double vector load of 32-byte or 64-byte consecutive data

The following types of memory read transactions are ignored by the load cache:

- Single vector WX-type load (48-byte transaction)
- Single/double vector S (scalar) and C2 (circulating 2 items) distributions
- Scalar register loads
- Lookups
- DMA reading VMEM
- DLUT reading VMEM

When enabled, the load cache monitors ALL memory transactions to invalidate cache entries when there's a hit, including

- Scalar/vector stores
- Histogram/VAST operations

- DMA writing VMEM
- DLUT writing VMEM

When cache is enabled and cache hit occurs, load data is retrieved from the cache and memory read transaction is not issued, saving some power. When the cache is enabled and a cache miss occurs, there is no performance penalty, but there is a small power penalty. Thus, enable/disable control is exposed to the programmer for power optimization. Load data cache should be enabled when there are repeated accesses to small localities, like in the case of filtering, and should otherwise be disabled.

5.6 Memory Allocation among VMEM Superbanks

VPU application code may use storage specifiers {DMb, RAM_Ab, RAM_Bb, RAM_Cb}, together with chess_segment(A/B/C) and optional alignment constraint to allocate scalar or array variables onto specific superbank.

Using RAM_Ab, RAM_Bb, RAM_Cb with chess_segment(A/B/C) causes the linker to allocate variable to superbank A, B, or C, respectively

Example 1:

```
short chess_storage(RAM_Ab % 512: chess_segment(A)) foo[256];
```

This allocates foo as a 256-element short array on superbank A with 512-byte alignment.

Example 2:

```
char chess_storage(RAM_Bb % 512: chess_segment(B)) bar[256];
```

This allocates bar as a 256-element char array on superbank B with 512-byte alignment.

Example 3:

```
int chess_storage(DMb % 4) more_foo[256];
```

Using DMb storage specifier causes the linker to allocate such variables to superbank A first, followed by B, then C, where it fits. Reserved regions between superbanks are skipped automatically. This allocates more_foo as a 256-element int array with 4-byte alignment in one of the superbanks.

Example 4:

```
int more_bar[128];
```

Not using any storage specifier causes the linker to allocate the variable to global memory (_global segment in BCF file). Application project can supply a custom BCF file to place _global segment in a valid memory range. Otherwise, the default BCF file applies and places _global segment in superbank A.

Chapter 6. VPU Instruction Set Architecture

6.1 Processor Architecture

6.1.1 Key Features

The VPU instruction set architecture has the following key features:

- VLIW and Wide SIMD vector processor, with multiple operations and multiple load/store slots.
- Multi-dimension address generation (6 dimensions).
- Multiple levels of zero-overhead hardware looping (2 levels).
- Instruction-level predication of certain vector operation, scalar load/store, vector load.
- Lane predication for vector store.
- Loop collapsing to reduce overhead across data block and filter kernel dimension, enabled by address generation and predication.
- Reduced code size and library construction effort for filtering and other windowing operations, enabled by zero-overhead nested looping through loop collapsing.
- Memory banking and parallel lookup, histogram, and vector addressed store.
- Memory bank address calculation to implement transposed vector load/store for various transposition options.
- Circular buffer addressing for memory-copy-free data/compute reuse.
- Rich set of load and store data distribution patterns.
- Vector load with permutation of loaded data.
- Vector-lane predication of selected store operations.
- Protected pipeline with sequential execution (except branch delay slots) and hardware dependency stalling.

The VPU instructions are scheduled in the following 7-way VLIW format. Each instruction word is 32-bit long, and up to 7 instruction words can be executed together as an execution packet.

Table 4. VLIW instruction format

S0	S1	V0	V1	M0	M1	M2
Scalar operation or Control (branch, call, return, SW breakpoint)	Scalar operation	Vector operation	Vector operation	Load/store, scalar or vector, lookup, histogram, vector-addressed stores	Load/store, scalar or vector	Load/store, scalar or vector

Variable-length packet encoding is supported, so that NOP (no operation) instructions are skipped and not taking up any code space. There is an exception though. Compiler may insert NOPs intentionally to align branch target, beginning of function, etc., execution packets to reduce branch penalty.

In general, control instructions are available only in S0 slot. Scalar operations are available in both scalar slots. Vector operations are available in both vector slots. Memory operations are available in all 3 memory slots, except lookup, histogram, and vector-addressed store are available only in M0. Additional details:

- Agen save/restore instructions are available only in M0 slot.
- Quad-vector store instructions are available only in M0 slot.
- Per-lane predicated store instructions via vector register file are available only in M0 slot.
- Per-lane rounding store (double vector only) instructions are available only in M0 slot.

6.1.2 Program and Data Memory Spaces

Program memory space is 32-bit byte address, with valid range [0 , 2^32 – 4], as instruction words are 32-bit each.

Data memory space is 20-bit byte address that spans 1MB, but only valid inside each of three 128KB superbanks, for a total of 384KB of physical memory. Please see VMEM Overview for the memory map.

Access outside the valid range would be wrapped back to the valid range. See Section VMEM Superbanks for details in address mapping. Programmers should not take advantage of this address wrapping, as data memory footprint and layout can change in the next generation.

Reading uninitialized memory locations WILL NOT be detected as an error but can trigger parity error. It's too expensive to implement such detection or automatic initialization in hardware. It is software's responsibility to either initialize the entire VMEM at the start of task or avoid referencing uninitialized memory locations.

6.1.3 Architecture Registers

6.1.3.1 Control and Scalar Registers

Program counter (PC), counting in 32-bit granularity so PC = 1 means byte address of 4.

VPU program space is 2^32 bytes, due to tool-chain constraints, and hardware conforms to this constraint. Although PC appears as a 32-bit register, upper 2 bits are not used. Upon task launch, VPU gets a starting PC specified in a 32-bit byte address config register by dropping lower 2 bits of the register. Also, the interface between VPU and I-cache carries 27-bit address in 256-bit (32-byte) granularity.

- Scalar registers R0..R31, 32-bit each. Special registers among them: R0 = constant zero
- SP (stack pointer) = R1
- LR (link register) = R15
- Global data page pointer = R16
- PL (64-bit product's low 32-bit, also quotient for DIV) = R12
- PH (64-bit product's high 32-bit, also remainder for DIV) = R13

All scalar registers are reset to 0.

Compiler is instructed to treat R0 as constant 0 and not modify R0. User assembly program can use R0 as a normal register and write non-zero to R0, but this would break compiled code so is highly inadvisable.

Stack grows by incrementing the stack pointer, so items in the local frame (already in the stack) are located with negative offset from the stack pointer. For example, the last int32 word pushed into the stack occupies SP – 4 ~ SP – 1 byte addresses, so is addressed by its starting byte address SP – 4. Compiled code uses load/store with base + immediate offset addressing mode to address items on the stack, and the immediate offset has range of [-2048, 2047]. Thus, if we use the stack pointer register R1 to represent the stack pointer itself, local frame size is limited to 2048 bytes.

In the model's compiler setting, we tell compiler to put an offset of -2048 between the logical stack pointer and the actual stack pointer register R1. In other words, we set SP_register (R1) = SP – 2048. This allows any local frame to take as much as 4096 bytes, thereby doubling the local frame size. This is because SP_reg + [-2048, 2047] = SP – 2048 + [-2048, 2047] = SP + [-4096, -1].

Hardware looping registers:

- LF: 2-bit loop level, -1, 0 or 1, indicating which loop level the execution is in, reset to -1 (which is encoded as binary "11").
- LS[0..1]: 32-bit loop start PC, reset to 0
- LE[0..1]: 32-bit loop end PC, reset to 0
- LC[0..1]: loop count, 32-bit, reset to 1

There is also a predicate register file to support instruction predication:

- Predication registers: P2..P15 each 32-bit (P0, P1 are unconditional), reset to -1 (all ones)

Additional miscellaneous registers are:

- GPI: general purpose input register, 32-bit
- GPO: general purpose output register, 32-bit, reset to 0
- TSC: Free running timestamp counter for performance instrumentation, 64-bit, reset to 0
- INV: floating-point invalid flag, 1-bit, reset to 0
- CFG_VMEM: 3 x 32-bit, 32-bit for each superbank, bit 0 for load cache enable, bits 31..1 reserved, reset to 0

6.1.3.2 Vector Registers

There are 3 vector register files: the main vector register file V0..V31; the working register file W0..W31; and the accumulator register file AC0..AC31. Each register 384-bit and can be partitioned as follows:

- 8 lanes x 48-bit (extended word, vintx)
- 16 lanes x 24-bit (extended half-word, vshortx)
- 32 lanes x 12-bit (extended byte, vcharx)

In addition, there is an extension register file, XRF, that extends precision of ARF on a lane-by-lane basis. It's used in selected MAC operation (VFilt4x2x2BBW) with 16 lanes x 32-bit per vector register entry, with lower 24-bit supplied by ARF, upper 8-bit supplied by XRF. The extended accumulator register file, XARF, XAC0..XAC31, is partitioned as

- 16 lanes x 32-bit (Further extended half-word, xvshortx)
- 32 lanes x 16-bit (Further extended byte, xvcharx)

VRF and WRF have extensive bypassing to reduce load-to-math and math-to-math latencies. ARF is accessible as accumulators. Compiler maps source code variables to these register files according to latency requirement and register capacity constraints.

Vector registers are not cleared during reset; it is software's responsibility to initialize each register before its value can be used.

6.1.3.3 Agen Registers

Each unit of the agen register file AGEN[0..7] has the following fields:

- Addr (32-bit, but only lower 20 bits are used in address calculation), reset to 0
- Transposition lane offset (12-bit), reset to 0
- Rounding/truncation option and number of bits (8-bit), reset to 0 (no rounding)
- Saturation option (2-bit), reset to 0 (saturation disabled)
- min/max option (2-bit)

- Auto predication off (1-bit), reset to 0, indicating agen loop has gone past max iteration count in all levels, so that subsequent stores should be automatically predicated off, overriding predicate register (or predicate vector register).
- Number of iterations (6 x 16-bit), reset to 1
- Address modifiers (6 x 18-bit), reset to 0
- Circular buffer start and size (2 x 16-bit), reset to 0
- Saturation parameters (4 x 32-bit), reset to 0
- Loop variables (6 x 16-bit), reset to 0
- Min/max values (2 x 32-bit), reset to 0, and initialized to signed/unsigned 32-bit MAX/MIN values depending on min/max option

Each Agen register has a parameter configuration portion, basically the first 16 words or 512-bit in memory and 428-bit in register (difference comes from 6 address modifiers, 32-bit in memory versus 18-bit in register). The last 6 words or 192-bit holds loop variables, auto_predicate_off and min/max values.

Agen configuration can be stored in memory in 512-bit, and when it's read back, loop variables and min/max values are reset, and this is useful to save and restore Agen configuration. In Orin there are instructions to save/restore the remaining part of Agen. The entire register entry can be copied from one agen register to another as well.

Data organization of the agen configuration in memory (from Agen configuration save, AgenCfgST) is as follows.

Figure 7. AGEN data format in memory

Word	31					0
0	Addr					
1	reserved (4-bit)	minmax _opt (2-bit)	sat_opt (2-bit)	round/truncate opt and bits (8-bit)	reserved (4-bit))	lane_offset (12-bit)
2	N2 (16-bit)				N1 (16-bit)	
3	N4 (16-bit)				N3 (16-bit)	
4	N6 (16-bit)				N5 (16-bit)	
5	reserved (14 upper bits)			MOD1 (18 LSBs)		
6	reserved (14 upper bits)			MOD2 (18 LSBs)		
7	reserved (14 upper bits)			MOD3 (18 LSBs)		
8	reserved (14 upper bits)			MOD4 (18 LSBs)		
9	reserved (14 upper bits)			MOD5 (18 LSBs)		
10	reserved (14 upper bits)			MOD6 (18 LSBs)		
11	CB_SIZE (16-bit)				CB_START (16-bit)	
12	SAT_LIM_LOW (comparison)					
13	SAT_VAL_LOW (replacement)					
14	SAT_LIM_HIGH (comparison)					
15	SAT_VAL_HIGH (replacement)					

Word	31	0
16	I2 (16-bit)	I1 (16-bit)
17	I4 (16-bit)	I3 (16-bit)
18	I6 (16-bit)	I5 (16-bit)
19	reserved (31-bit)	auto pred off (1-bit)
20	min_val (32-bit)	
21	max_val (32-bit)	

The rest of Agen data structure (6 x 32-bit = 192 bits) in the Agen register file, not directly visible but can be accessed one loop variable at a time through STH A<id>.I<level>:

In the data structure, ignored fields, basically upper bits of each address modifier, are writable via CfgAgen Mod instruction as well as CfgAgenLD instruction, but are not utilized in the address calculation.

Reserved fields are initialized to zero in InitAgen. They are not modifiable via any CfgAgen instructions and not utilized in any Agen functionality. Through CfgAgenLD, if corresponding contents in memory are non-zero, zero will be loaded into Agen data structure instead. When CfgAgenST is used to store out the whole Agen data structure, corresponding bits in memory will show zeros.

6.1.3.4 Floating-point Invalid Flag

To facilitate development of floating-pointing applications, in VPU we have a Boolean flag to for floating-point invalid, invalid_flag, that captures any invalid outcome (NaN) from FP32/FP16 operations. It's a sticky bit, so that when there is any invalid outcome from S0/S1/V0/V1 slots (as we support scalar as well as vector floating-point), the bit is set.

```
invalid_flag |= s0_invalid | s1_invalid | v0_invalid | v1_invalid
```

There are a pair of MOV instructions to move invalid_flag to/from scalar register, so that the flag can be cleared at beginning of applications and collected (and perhaps cleared) at key points in the application to check for unexpected outcomes.

Please see Exception for Instructions Accessing FP Invalid Flag for instruction execution ordering exceptions around FP invalid flag. Please see Floating-Point Invalid Flag for MOV instructions for FP invalid flag.

Note that the invalid flag read-modify-write dependency is hidden from the compiler, so that compiler can freely reorder, combine, and even optimize out unnecessary FP operations to achieve better performance. If, for whatever reason, certain FP operations should not be optimized out even when they are unnecessary, developer can add chess_keep_dead() compiler directive on the variable assigned to the FP operations.

For example,

```
float var3 = fadd(var1, var2);
chess_keep_dead(var3);
// no subsequent use of vars
```

6.1.4 Control Instructions

Control instructions include the following:

- Flow control instructions include jump, jump-and-link (call), and conditional branch.
- Zero-overhead hardware loop instruction
- Memory fence instruction
- Miscellaneous hardware control instructions involving GPI, GPO, coprocessor load/store, R5 interaction, time stamp counter, floating-point invalid flag, and load data cache
- Debug instructions

Control instructions are only supported on the S0 slot.

There are 2 delay slots following jump, jump-and-link, conditional branch, and hardware loop. For jump and branch, there are additional 2 to 3 cycles of gap before the first execution packet of the jump target can be executed, due to the fetch latency.

Memory fence takes variable number of cycles, up to 8 cycles, as it is stalled until preceding memory writes are committed to memory, to ensure memory coherency.

Hardware control instructions that interact with other hardware components (GPI, GPO, WFE_GPI/R5, SIG_R5, CPLD, CPST) take up to 16 cycles to execute, so that all preceding instructions complete their execution, to avoid any synchronization issues.

For example, VPU software might write some value in VMEM before toggling a GPO bit that triggers a DMA transfer to read from VMEM, so it's only prudent to allow the memory write to be completed before the GPO bit is toggled.

6.1.4.1 Hardware Looping

VPU supports 2 levels of zero-overhead hardware loops through the hardware loop instruction (RPT) and the following hardware looping registers:

- LF: 2-bit loop level, -1, 0 or 1, indicating which loop level the execution is in, reset to -1 (encoded as binary "11") to mean not being in any loop
- LS[0..1]: 32-bit loop start PC, reset to 0
- LE[0..1]: 32-bit loop end PC, reset to 0
- LC[0..1]: loop count, 32-bit, reset to 1

Behavior of hardware loop (RPT) that encodes a scalar register and an immediate value:

- LF++;
- LC[LF] = scalar register value, for the loop iteration count.
- LS[LF] = starting PC = PC(3 execution packets from RPT)

- LE[LF] = ending PC = PC(2 execution packets from RPT) + immediate

Hardware looping is carried out by RPT updating LF and corresponding LC, LF, LE entries, and by monitoring PC against LE[LF], the ending PC of current loop level.

LF is initialized to -1, so when RPT is first executed, LC[0], LS[0], LE[0] are filled.

Conditional branch-back from loop-end PC is carried out via:

```
if (LF >= 0) {
    if (PC == LE[LF]) {
     if (LC[LF] == 1) {
       LF--;
     } else {
       LC[LF]--;
       branch_target = LS[LF];  // take branch right away
                 // end-of-loop branch back has no delay slots
     }
   }
}
```

All these steps – detecting end of the loop body by matching PC against LE, checking the loop count register LC, making the decision to branch back to beginning of loop body (LS) or to decrement LC then fall out of the loop – occur in the background without incurring any explicit instruction, thus they feature **zero-overhead** looping.

There is a hardware loop buffer to store the first 3 execution packets of the loop body, so that branching back from loop-end to loop-start does not suffer the usual 2 ~ 3 cycles of pipeline bubble. Loop execution goes seamlessly from one iteration to the next iteration.

With the preceding hardware looping implementation, when nested hardware loops are used (up to 2 levels), the 2 loop levels should not share the same ending PC. Consequently, an NOP may be inserted by the compiler when there is no active processing between the end of two loop levels. For example:

```
add__sint_add___sint___sint___sint
    104 RPT R6,#7     || LHI #0,R7
    106 ADD R5,R4,R5   || ADDI R4,#0, R2
    108 LHI #0,R5     || ADD R5,R6,R3
    110 RPT R2,#1             // outer loop starts
    111 NOP
    112 NOP
    113 ADD R5,R4,R5   || ADD R3,R7,R7 // innerloop starts/ends
    115 NOP                // outer loop ends
    116 JR R15
    117 SUB R7,R5,R2
    118 NOP
```

In this example, the outer loop starts at PC 110, the inner loop starts at 113, two delay slots after the corresponding RPT instruction.

The immediate field of RPT encodes the PC difference between the 2nd delay slot (just before entering the loop) and the last packet of the loop. In the example above, the

outer loop ends at PC 115, so the RPT immediate field encodes 115 – 108 = 7. The inner loop ends at 113, so the RPT immediate field encodes 113 – 112 = 1.

Currently, the compiler does not generate code that branches into the middle of an execution packet, or into a delay slot of any execution-control instruction. Moreover, an assembly program that has such behavior would be rejected by the loader so it would not simulate. Due to the tool chain restriction, hardware behavior when supplied with such an assembly program is declared undefined.

In the case of nested hardware loops, the inner loop RPT shall not be placed in a delay slot of the outer loop RPT, as it complicates the VPU execution controller to support such looping structure. Compiler does not generate such a code sequence.

6.1.5 C Function Calling Convention

C functions shall adopt the following calling convention:

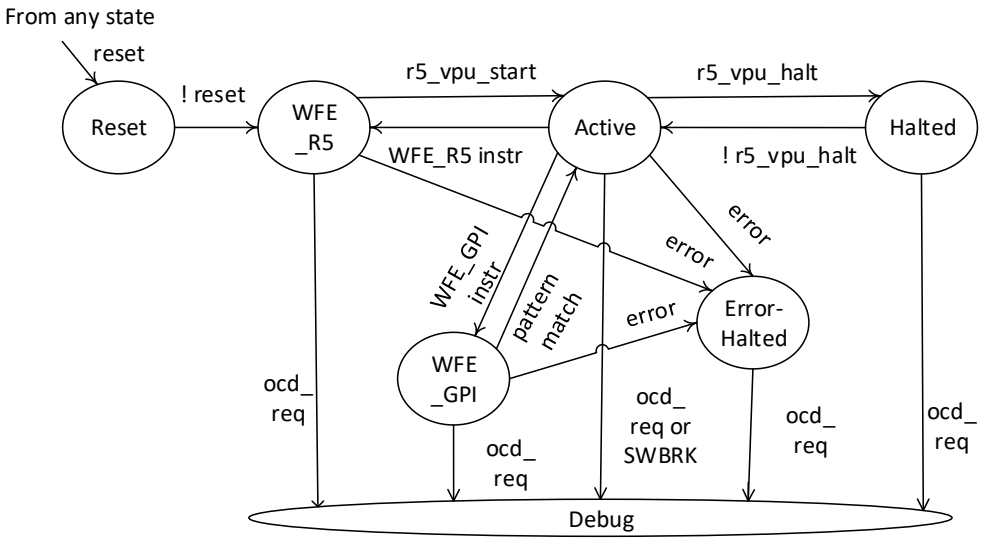
- Stack pointer = R1
- Link register = R15
- Global data page pointer = R16
- Scalar argument registers: R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14
- Scalar return value registers: R2, R3
- Vector argument registers: V0, V1, V2, V3, V4, V5, V6, V7
- Vector return value registers: V8, V9
- Double vector argument registers: V0:V1, V2:V3, V4:V5, V6:V7
- Double vector return value registers: V8:V9

In subsequent generations of VPU, it is likely that scalar and/or vector register file may be expanded, and the C function calling convention may change. No assembly backward compatibility is expected.

6.1.6 Processor Execution States

VPU execution state diagram is shown in the following figure. Description of the states, various state transitions and conditions follow the figure.

Figure 8. VPU execution state diagram



General priority for state transition, where applicable, is reset > VPU instruction > error > debug > VPS register programming (halt/unhalt/r5_vpu_start) > VPU internal state/detection.

Reset state: When reset is asserted, whatever state VPU is in, it shall transition to the Reset state. De-asserting reset signal would transition VPU to WFE_R5 state.

Debug state: When the ocd_req signal in VPU debug interface is asserted, VPU would transition from any state except the Reset state to the Debug state. The state VPU transitions from is saved in a shadow execution state (SES) register. If/when resuming execution is desired, it is debugger software's responsibility to restore VPU to the interrupted state (including all VMEM and register contents, except for first 64 bytes of VMEM), then de-assert ocd_req to allow VPU to resume execution by going to the state saved in the SES. Debugger software can optionally change the SES before de-asserting ocd_req to redirect VPU to a different state from the interrupted execution state. Please see 13.1 for details on debug features.

WFE_R5 state: This is when VPU is waiting for R5 to provide a starting PC (R5_vpu_start_PC). Once R5 writes the starting PC then writes 1 to the R5_vpu_start register field, VPU transitions out of WFE_R5 state and jumps to the starting PC. VPU software normally terminates a subframe-level task with WFE_R5 instruction, which takes VPU back to this state.

Active state: VPU normal execution is in the active state. From active state, VPU can be temporarily halted by vpu_halt register been written 1, to transition to the Halted state. VPU can transition to debug state by debug controller asserting ocd_req, or by executing SWBRK instruction. VPU can execute a WFE_R5 instruction to go to the WFE_R5 state. VPU can execute a WFE_GPI instruction to go to the WFE_GPI state.

Upon hardware error and when the error source is configured to error-halt, VPU goes to the Error-Halted state.

Priority on conditions to transition from active state is reset > error -> debug > r5_vpu_halt > instruction. Instructions can be WFE_R5 or WFE_GPI. Since both are control slot (S0) instructions, only one can be executed at any time.

WFE_GPI state: VPU executing a WFE_GPI instruction would transition VPU to this state. WFE_GPI allows a mask value and a match value as arguments, and hardware logic keeps VPU in this state until (GPI & mask) == match, upon which VPU is transitioned back to the Active state.

Note: WFE_GPI is not exclusive to interaction with DMA; it can be used for checking availability of I-cache prefetch and/or invalidate.

Halted state: R5 can temporarily halt VPU by writing 1 to the vpu_halt register field. When the field is written 0, VPU would go back to the Active state and resume execution. This mechanism can be used by R5 software to pause VPU execution upon watch-dog timer expiration, so VPU state can be saved for further diagnosis.

Error-Halted state: When one of the error conditions occurs and it's configured to error-halt in error handling (see 13.2, and for further details please see PVA VPS IAS), VPU is transitioned to the Error-Halted state. From this state, the debugger can drive the execution state to Debug, or R5 can reset VPU.

The transition from WFE_GPI and WFE_R5 to Error-halted. It is possible for an instruction causing the error to be close enough to WFE_GPI/WFE_R5 instruction that the execution state is temporarily transitioned to WFE_GPI/WFE_R5 states before ending up in Error-halted state.

6.2 Overview of Scalar/Vector Math Instructions

With VPU execution packets organized as 7-way VLIW, it is most convenient to describe the instructions in terms of instruction set grouping.

- Control instructions can only be placed in the first scalar slot, S0.
- Scalar math instructions can be placed in either of the scalar slots, S0 and S1.
- Vector math instructions can be placed in either of the vector math slots, V0 and V1. These 2 slots are symmetrical in functionality.
- Certain memory operations can only be placed in the first memory slot, M0.
- The remaining memory operations can be placed in any of the 3 memory slots, M0, M1 and M2.

This is a brief overview of scalar and vector math instructions, including general functionality description and latency. For a more detailed description of each instruction, consult Chapter 9 Instruction Set Reference. Memory instructions are better understood after some coverage on memory banking and address generator features.

The latency number of cycles specified in the following sub-sections are for back-to-back execution of the same class of instructions; for example, scalar integer math to scalar integer math. Latency across different classes of instructions is outside the scope of the VPU Programmer's Guide.

6.2.1 Scalar Integer Math Instructions

We support common arithmetic and logic operations in both scalar slots.

- Integer addition, subtraction, compare, and, or , exclusive or, sign/zero-extend
- Integer shift left/right, signed/unsigned min/max
- Integer multiplex (C select operator), shift-and-add, compare within, bit count
- 32-bit x 32-bit multiplication, keeping 32-bit product
- Signed/unsigned 32-bit x signed/unsigned 32-bit multiplication, keeping 64-bit product
- Integer division, taking up to 33 cycles depending on the dividend bit width

All scalar integer math instructions except for integer division have 1 cycle of latency.

6.2.2 Scalar Predicate Instructions

VPU has a predicate register file, and some vector math instructions are predicated, those with _CA postfix, to support periodically Clearing Accumulator in a filtering application for example. We support instructions to move between the predicate register file and scalar register file, as well as a few variations of modular increment instructions for periodic predication.

All predicate instructions have 2 cycles of latency.

6.2.3 Vector Math Instruction General Rules

We support many vector integer math instructions. There are multiple ways to group them into digestible chunks. The relevant section in the Instruction Reference chapter categorizes instructions by number of input/output operands. Here we categorize instructions by functionality:

- **ALU instructions**: move, bitwise and/or/exclusive-or/not, bitwise 3-input and/or, logical and/or/not, promote/demote, Hamming distance. All but Hamming distance have 1 cycle of latency; Hamming distance has 3 cycles of latency.
- **Bit manipulation instructions**: bit reverse, bit count, bit interleave/deinterleave, most significant bit detection. All have 1 cycle of latency.

- **Compare instructions**: 2/3-input min/max, 3-input median, min/max with LT/GT flag, Compare GE/GT/LE/LT/EQ/NE, multiplex (C select operator), 2-in/out sort, sort with payload, horizontal min/max. All have 1 cycle of latency.
- **Add/Subtract**: 1-cycle latency instructions are negation, sign-magnitude, apply sign, add/sub. 2-cycle latency instructions are add2sub (A+B-C), absolute difference, sum of absolute differences (SAD).
- **Shift instructions**: shift (left or right), shift-or, shift-add, shift right, shift left, round, extract bits, split bit sections, normalization. All have 2 cycles of latency.
- **Permutation instructions**: permute, collate index, expand index, compare bit-pack, bit unpack, bit transpose, select lane, SGM min-path-cost. All have 4 cycles of latency.
- **MAC (multiply-accumulate) instructions**: multiply, multiply-add, multiply-subtract, 2/4/2x2/4x2-term dot-product, 4/4x2/4x2x2-term filtering, blending, complex multiply, sum of squares, square of sum, 2x2 determinant, 8x4x2 term exclusive-not-or-add. All have 3 cycles of latency.

6.2.3.1 Extended Precision

The vector unit executes up to 2 vector operations per clock cycle. Various vector ALU instructions are available. A 32-entry 384-bit vector register file (VRF), a 32-entry 384-bit working register file (WRF) and a 32-entry 384-bit accumulator vector register file (ARF) supply the operands and store the outcomes.

There is a 128-bit extension for the ARF to extend each entry to 512-bit wide. The extended accumulator register file (XARF) is accessible only from selected MAC operation, VXNorAdd8x4x2, VFilt4x2x2BBW, VDotP4BBW, VDotP4x2BBW, and store operations.

Each 384-bit entry in VRF/WRF/ARF is logically partitioned into 32 x 12-bit (extended byte), 16 x 24-bit (extended halfword), or 8 x 48-bit (extended word). Each 512-bit entry in XARF is logically partitioned into 32 x 16-bit (short), or 16 x 32-bit (word).

VPU vector math instructions operate on extended precisions. Extended byte is 12-bit, versus standard byte being 8-bit. Extended halfword is 24-bit, versus standard halfword. Extended word is 48-bit, versus standard word being 32-bit.

The idea is that normally in C code, variables and arrays are declared with standard element type of char/uchar (8-bit), short/ushort (16-bit), and int/uint (32-bit). VPU compute kernels use signed or unsigned loads to load data from VMEM and sign-extend or zero-extend the values to place into destination vector registers. Processing occurs in the vector datapath via vector math instructions, reading from and writing back into vector register files. Eventually when a suitable chunk of the compute kernel is completed, results are written back to VMEM in standard precision.

It is possible for VPU programs to store the intermediate outcome in extended precision and load them back into vector register file. This can be through an extended-type load/store in the code or can be through the compiler automatically spilling vector

variables onto the stack, when size of variables involved in a compute kernel exceeds size of the vector register files (VRF/WRF/ARF/XARF).

In general, we would like to avoid spilling vector variables into the stack, as it generally degrades performance and consumes higher power consumption. Programmer should reduce size of variables involved in the computation by breaking up the computation in a loop into multiple loops, or by reducing the unrolling factor in the unroll_loop pragma.

Note that lane partitioning does not involve any conversion instruction but is accomplished via each vector math instruction specifying what precision it operates on. Vector math instructions are either type-agnostic – for example, bitwise operations – or have a type designation that can be:

- W: 48-bit word
- H: 24-bit half-word
- B: 12-bit byte
- W: 32-bit standard word in VFilt4x2x2BBW, VDotP4BBW, VDotP4x2BBW

For example, in VAddH, single vector addition half-word, the 'H' specifies that it operates on extended halfword precision and thus treats each source and destination vector register entry as 16 lane x 24-bit. Some instructions involve operands with multiple precisions. For example, VFilt4x2x2BBW involves extended byte (12-bit) source operands as well as word (32-bit) accumulator operand (which is both source and destination).

Many vector math instructions support one of the source operands coming from a scalar register, depending on the operation type, appropriate number of lower bits (number of bits specified in the operation) are extracted, or entire 32-bit value is signed-extended, then broadcast to all lanes to participate in the vector operation specified.

6.2.3.2 Signed/Unsigned Handling

Note: There are no signed/unsigned designations in vector math instructions. **All vector arithmetic operations where signed/unsigned make a difference, including comparison, min/max, right-shift, round, etc., are performed as signed operations.**

Signed and unsigned data may be stored in memory. Programmers are responsible for choosing signed/unsigned data type in the load instructions to read data into vector register file. Signed data type load (for example, VLDB) would cause the 8/16/32-bit data items in memory to be sign-extended to the 12/24/48-bit lanes in a vector register. Unsigned data type load (for example, VLDBU) would cause the 8/16/32-bit data items in memory to be zero-padded to the 12/24/48-bit lane in a vector register.

For storing data back to memory, writing to memory itself is type-agnostic; however, if it's an agen-based store, and rounding and/or saturation features are enabled, be aware that right-shift in store-path rounding is performed as **signed right-shift**, and comparisons in store-path saturation are performed as **signed comparison**. Thus, if a programmer intends to use full range of extended precision (12/24/48-bit) to store unsigned data, store-path rounding and saturation features should be disabled.

6.2.3.3 Data Types and Corresponding Bit Widths

Unless otherwise noted, the following lane partitioning scheme is followed in vector register:

- Word: 8 lanes x 48-bit, lane 0 in Vreg[47:0], lane 1 in Vreg[95:48], etc
- Half-word: 16 lanes x 24-bit, lane 0 in Vreg[23:0], lane 1 in Vreg[47:24], etc
- Byte: 32 lanes x 12-bit, lane 0 in Vreg[11:0], lane 1 in Vreg[23:12], etc
- No type: bitwise operation on whole 384-bit
- Standard Word: 16 lanes x 32-bit in XARF, lane 0 in XACreg[31:0], lane 1 in XACreg[63:32], etc.

Where a scalar register is used as an operand (can be src2 or src3), the general scalar operand bit width usage behavior is

- Word: whole 32-bit sign extended to 48-bit and broadcast to 8 x 48-bit lanes
- Half-word: lowest 24-bit broadcast to 16 x 24-bit lanes
- Byte: lowest 12-bit broadcast to 32 x 12-bit lanes
- No type: not applicable, as no-type operations do not allow scalar as operand

Exceptions to the above are stated in the specific instruction description. For example, for bitwise operations it makes more sense to zero-extend in case of Word type rather than sign-extend. As another example, VBitUnpk instruction uses its scalar operand one bit per lane, so it's 8-bit for Word type, 16-bit for Halfword type, and 32-bit for Byte type.

Some ALU instructions do not use the full lane, but just 8/16/32 or 9/17/33 LSBs of the lane, and they are specifically marked as such in the instruction table. Multiply and multiply-add/subtract and bit reverse are in this category.

6.2.3.4 Internal Bit Widths and Overflow

Arithmetic datapath implementing various instructions employ sufficient precision so that the functionality can be modeled as having infinite precision, but the final outcome is presented in the specified output width, so the hardware is not responsible for outcome overflow.

This style of functionality specification does not pin down internal details, leaving implementation flexible, while clearly defining the end-to-end behavior. The implementation flexibility allows sharing logic among various data types.

For example, VAdd adds 2 operands in each Byte/Half-word/Word lane. In case of Byte lane, inputs are 12-bit signed and output is 12-bit signed, and internal processing width can be any bit width greater than or equal to 12, so internally we can have

- 32 x 12-bit adders + 16 x 24-bit adders + 8 x 48-bit adders, each data type operates in separate datapath,
- 8 x 48-bit adders + 8 x 24-bit adders + 16 x 12-bit adders, carrying out half of half-word addition in 48-bit datapath, and half of byte addition in 24-bit and 48-bit datapaths, or

- 32 x 12-bit adders with carry logic to conditionally string together 24-bit and 48-bit additions based on type designation of the instruction.

For certain instructions, we do need internal bitwidth to be expanded to avoid internal overflow, but this does not mean the output would not overflow. VAbsDif and VSAD_CA are such instructions. Again, outcome is as if we use infinite arithmetic precision but only present the specified bit width to the output.

There is no out-of-range or overflow detection in VPU, and there is no automatic saturation. There is, however, free (not costing extra cycle) saturation in Agen-based vector store.

6.2.3.5 Application Vector Data Types

Various data types are referred to in the intrinsic field.

Note: Extended width types are all signed.

- vint: 8 x 32-bit vector (in memory)
- vuint: 8 x 32-bit vector (in memory, unsigned)
- dvint: 16 x 32-bit vector (in memory)
- dvuint: 16 x 32-bit vector (in memory, unsigned)
- vintx: 8 x 48-bit vector (mapped to register)
- dvintx: 16 x 48-bit vector (mapped to register)
- vfloat: 8 x 32-bit FP32 vector (in memory)
- dvfloat: 16 x 32-bit FP32 vector (in memory)
- vfloatx: 8 x 48-bit FP32 vector (mapped to register, sign-extended from FP32)
- dvfloatx: 16 x 48-bit FP32 vector (mapped to register, sign-extended from FP32)
- vshort: 16 x 16-bit vector (in memory)
- vushort: 16 x 16-bit vector (in memory, unsigned)
- dvshort: 32 x 16-bit vector (in memory)
- dvushort: 32 x 16-bit vector (in memory, unsigned)
- vshortx: 16 x 24-bit vector (mapped to register)
- dvshortx: 32 x 24-bit vector (mapped to register)
- xvshortx: 16 x 32-bit vector (mapped only to XARF)
- dxvshortx: 32 x 32-bit vector (mapped only to XARF)
- vhfloat: 16 x 16-bit FP16 vector (in memory)
- dvhfloat: 32 x 16-bit FP16 vector (in memory)
- vhfloatx: 16 x 24-bit FP16 vector (mapped to register, sign-extended from FP16)
- dvhfloatx: 32 x 24-bit FP16 vector (mapped to register, sign-extended from FP16)
- vchar: 32 x 8-bit vector (in memory)

- vuchar: 32 x 8-bit vector (in memory, unsigned)
- dvchar: 64 x 8-bit vector (in memory)
- dvuchar: 64 x 8-bit vector (in memory, unsigned)
- vcharx: 32 x 12-bit vector (mapped to register)
- dvcharx: 64 x 12-bit vector (mapped to register)
- xvcharx: 32 x 16-bit vector (mapped only to XARF)
- dxvcharx: 64 x 16-bit vector (mapped only to XARF)

There are two floating-point formats supported, FP32 and FP16. In vfloatx/dvfloatx, each 48-bit element contains one FP32 number with sign extended to fill the upper 16 bits. In vhfloatx/dvhfloatx, each 24-bit lane element contains one FP16 number with sign extended to fill the upper 8 bits.

For predication of lanes in vector stores, we use

- int: 8/16/32 bits of predication, mapped to one predicate register
- dpred: 64 bits of predication, mapped to two predicate registers

6.2.3.6 Data Ordering in Single and Double Vector Registers

Double vector data types have twice as many elements as the corresponding single vector data type. In vector register allocation, compiler would allocate even/odd register pairs (for example V2:V3) for double vector data type variables.

There are two schemes of element ordering in a double vector:

- **Sequential**: take dvintx for example, ascending elements are stored in dv.lo[0], dv.lo[1], ..., dv.lo[7], dv.hi[0], dv.hi[1], ..., dv.hi[7]
- **Interleaved**: take dvintx for example, ascending elements are stored in dv.lo[0], dv.hi[0], dv.lo[1], dv.hi[1], ..., dv.lo[7], dv.hi[7]

The interleaved format is the way physical design works, so it is supported throughout the instruction set. The sequential format is available only in load/store instructions and selected vector math operations.

Vector math operations mixing single and double vectors, typically due to 2x width expansion like VMulHHW, use deinterleaved ordering:

src1	a[0]	a[1]	a[2]	a[3]	...	a[14]	a[15]
src2	b[0]	b[1]	b[2]	b[3]		b[14]	b[15]
dst.lo	a[0] * b[0]		a[2] * b[2]			a[14] * b[14]	
dst.hi	a[1] * b[1]		a[3] * b[3]			a[15] * b[15]	

Vector demotion operations have both sequential (VDemote) and interleaving (VDemote_I) flavors, but promotion operation only has deinterleaving flavor (VPromote_DI).

See Vector Load/Store Distribution Options for sequential vs interleaving/deinterleaving flavors in load/store operations involving double and quad vectors.

6.2.3.7 Endianness

VPU adopts the Little Endian memory organization. In Little Endian, lower bytes are stored into lower addresses than upper bytes. For example, a vint vector {2, 3, 4, 5, 6, 7, 8, 9} in memory would look the same as a vshort vector {2, 0, 3, 0, 4, 0, 5, 0, 6, 0, 7, 0, 8, 0, 9, 0} in memory, or as a vchar vector {2, 0, 0, 0, 3, 0, 0, 0, ..., 9, 0, 0, 0} in memory.

Table 5. Little Endian layout of various data types

Word	0				1				2					7			
Content	2				3				4					9			
Halfword	0		1		2		3		4		5		...	14		15	
Content	2		0		3		0		4		0			9		0	
Byte	0	1	2	3	4	5	6	7	8	9	10	11	...	28	29	30	31
Content	2	0	0	0	3	0	0	0	4	0	0	0		9	0	0	0

The same Little Endianness is also observed in the lanes of vector registers. For example, a register holding vintx vector {2, 3, 4, 5, 6, 7, 8, 9} also has the same contents of another register holding vshortx vector {2, 0, 3, 0, 4, 0, 5, 0, 6, 0, 7, 0, 8, 0, 9, 0}. More generally, word lane i would occupy the same 48-bit section of storage in a vector register as short lanes 2*i and 2*i+1, with lane 2*i taking the lower 24-bit of that 48-bit section.

6.2.3.8 Intrinsic Functions/Operators Support

Most vector math instructions support single vector operands and have intrinsic functions or operators with single vector data type operands, for example, VBitRev instruction has the following single vector intrinsic functions:

```
vintx  vbitreverse(vintx  src);
vshortx vbitreverse(vshortx src);
vcharx vbitreverse(vcharx src);
```

For such instructions, double vector pseudo intrinsic functions/operators are also available to map to a pair of instructions, for example:

```
dvintx  dvbitreverse(dvintx  src);
dvshortx dvbitreverse(dvshortx src);
dvcharx dvbitreverse(dvcharx src);
```

The convention is to prefix the intrinsic function names with “d” so that it reads dv<something>.

Selected vector math instructions allow scalar operand to be broadcast to each lane before the operation takes place. Their intrinsic functions/operators support such operand type combinations as well. For example, for VAbsDif we support:

```
vintx  vabsdif(vintx  src1, vintx  src2);
vshortx vabsdif(vshortx src1, vshortx src2);
vcharx vabsdif(vcharx src1, vcharx src2);
vintx  vabsdif(vintx  src1, int   src2);
vshortx vabsdif(vshortx src1, int   src2);
```

```
vcharx vabsdif(vcharx src1, int   src2);
```

For such instructions, double vector pseudo intrinsics are also supported, for example:

```
dvintx  dvabsdif(dvintx  src1, dvintx  src2);
dvshortx dvabsdif(dvshortx src1, dvshortx src2);
dvcharx dvabsdif(dvcharx src1, dvcharx src2);
dvintx  dvabsdif(dvintx  src1, int   src2);
dvshortx dvabsdif(dvshortx src1, int   src2);
dvcharx dvabsdif(dvcharx src1, int   src2);
```

Note that in each function, the same int-type scalar operand is shared between the two single vectors.

A subset of vector math instructions has cross-lane dependency. For example, VMaxR does max reduction across 8 extended word lanes, 16 extended halfword lanes, or 32 extended byte lanes. For such instructions there is no double vector pseudo intrinsic support to avoid confusion.

Another subset of vector math instructions involved mixed size operands (between single and double vectors), for example, VMulBBH has two single vector vcharx type inputs, and its output is a double vector dvshortx type output. As we do not support quad vector data types, there is no double vector pseudo intrinsic support as well, and the intrinsics/operator field is similarly noted.

We also support various re-interpret type intrinsic functions:

Functionality	**Intrinsic**
Reinterpret as vcharx	`vcharx  as_vcharx (<vtype>);`
Reinterpret as vshortx	`vshortx as_vshortx (<vtype>);`
Reinterpret as vintx	`vintx  as_vintx  (<vtype>);`
Reinterpret as vfloatx	`vfloatx as_vfloatx (<vtype>);`
Reinterpret as vhfloatx	`vhfloatx as_vhfloatx(<vtype>);`
Reinterpret as dvcharx	`dvcharx as_dvcharx (<dvtype>);`
Reinterpret as dvshortx	`dvshortx as_dvshortx(<dvtype>);`
Reinterpret as dvintx	`dvintx  as_dvintx (<dvtype>);`
Reinterpret as dvfloatx	`dvfloatx as_dvfloatx(<dvtype>);`
Reinterpret as dvhfloatx	`dvhfloatx as_dvhfloatx(<dvtype>);`

With any of such re-interpret type intrinsics, there is no change in the variable value. The raw data is simply reinterpreted. For example, applying as_vshortx() on a vintx variable reinterpret each 48-bit lane i as a pair of 24-bit lanes 2*i and 2*i+1, lower 24-bit as the even lane, upper 24-bit as the odd lane.

For instructions sharing the same register entries (VRF, WRF, ARF) as source and destination, also known as read-modify-write operands, we expose functionality to the compiler in the form of intrinsic functions with return values.

For example, vector multiply-add of Byte type has this intrinsic function prototype:

```
vcharx vmaddb(vcharx src1, vcharx src2, vcharx src3, u3imm rnd_opt, int pred);
```

instead of

```
void vmaddb(vcharx src1, vcharx src2, vcharx & src3dst, u3imm rnd_opt, int pred);
```

The rationale for this choice is that return-value functions are more readable in application code.

Since such instructions normally have accumulator-like behavior, we expect programmers to use the same variable in the src2 fields as well as receiving return value of the function; for example:

```
acc = vmaddb(data, coef, acc, RND_R7, pred);
```

When the intrinsic functions are used this way, compiler usually achieves efficient register allocation without incurring additional register movements.

6.2.4 Scalar/Vector Floating-Point Math Instructions

The following floating-point instructions are supported in scalar and vector slots:

- FP16/FP32 add, subtract, multiply, multiply-add, multiply-subtract
- FP16/FP32 compare LT/LE/GT/GE/EQ/NE
- FP32 transcendental functions: square root, reciprocal, reciprocal of square root, log/exp base 2, sine, cosine, tanh
- Conversion functions among FP16/FP32/INT16 and INT32. FP-to-INT conversions include rounding and truncation options, and FP16-to/from-INT conversion includes fraction bit width to support fixed-point processing.

Scalar floating-point instructions have 2 latency cases. Scalar floating-point comparison instructions have 1 cycle of latency and remaining scalar floating-point instructions have 4 cycles of latency.

Vector floating-point has 3 latency cases. Vector floating-point comparison has 1 cycle of latency, conversion between FP16 and FP32 has 2 cycles of latency, and the remaining vector floating-point instructions have 3 cycles of latency.

VPU is an embedded processor that does not support exceptions. As an alternative, the floating-point invalid flag can be polled and set/reset by code explicitly.

The following features are also not supported:

- errno macro
- math_handling macro
- MATH_ERRNO macro
- MATH_ERREXCEPT macro
- EDOM or domain error
- ERANGE or poll error

6.2.4.1 FP Math Corner Cases

FP math outcome for various corner cases, x being a non-zero regular FP number:

Table 6. FP add/subtract/multiply corner cases

FAdd:

			src2			
src1	***-x***	***zero***	***-zero***	***inf***	***-inf***	***NaN***
x	zero	x	x	inf	-inf	NaN
zero	-x	zero	zero	inf	-inf	NaN
-zero	-x	zero	-zero	inf	-inf	NaN
inf	inf	inf	inf	inf	NaN	NaN
-inf	-inf	-inf	-inf	NaN	-inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

FSub:

			src2			
src1	***x***	***zero***	***-zero***	***inf***	***-inf***	***NaN***
x	zero	x	x	-inf	inf	NaN
zero	-x	zero	zero	-inf	inf	NaN
-zero	-x	-zero	zero	-inf	inf	NaN
inf	inf	inf	inf	NaN	inf	NaN
-inf	-inf	-inf	-inf	-inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

FMul:

				src2			
src1	***1***	***-1***	***zero***	***-zero***	***inf***	***-inf***	***NaN***
1	1	-1	zero	-zero	inf	-inf	NaN
-1	-1	1	-zero	zero	-inf	inf	NaN
zero	zero	-zero	zero	-zero	NaN	NaN	NaN
-zero	-zero	zero	-zero	zero	NaN	NaN	NaN
inf	inf	-inf	NaN	NaN	inf	-inf	NaN
-inf	-inf	inf	NaN	NaN	-inf	inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

The outcome of FMAdd(a, b, c) follows that of FAdd(FMul(a, b), c) for the above corner cases. Outcome of FMSub(a, b, c) follows that of FAdd(FMul(-a, b), c) for the corner

cases. For FMAdd, when multiplication a * b results in number too small to represent even as denormal, the product is represented as +zero or -zero, before the addition is performed. Similarly for FMSub with multiplication -a * b.

For combination of src2/src3 being zero/-zero , FMAdd and FMSub outcomes are:

Table 7. FP multiply-add/subtract corner cases

src1	*src2*	*src3*	*FMAdd*	*FMSub*
pos	zero	zero	zero	zero
pos	zero	-zero	zero	-zero
pos	-zero	zero	zero	zero
pos	-zero	-zero	-zero	zero
neg	zero	zero	zero	zero
neg	zero	-zero	zero	-zero
neg	-zero	zero	zero	zero
neg	-zero	-zero	-zero	zero

Note: These corner cases apply to scalar and vector, hfloat (FP16) and float (FP32) types.

FP multiply corner cases:

Table 8. FP multiply corner cases

src1	*src2*	*FMul*
zero	-zero	-zero
-zero	zero	-zero
zero	neg	-zero
neg	zero	-zero
-zero	pos	-zero
pos	-zero	-zero

6.2.4.2 FP MUFU Instruction Corner Cases

Corner cases of reciprocal, square root, reciprocal square root, exp2, log2, sine, cosine and tanh functions are documented in the corresponding instruction details.

6.2.4.3 FP Comparison Corner Cases

FP comparison always returns integer 0 (false) or 1 (true), and works as if FP numbers are placed into these categories that have strict ordering:

```
-inf < negative FP numbers < -0 == 0 < positive FP numbers < inf
```

Negative FP numbers and positive FP numbers compare normally.

Behavior of comparison involving inf or -inf is:

- inf is equivalent to inf, thus inf cmp_op inf is true for {==, <=, >=} and false for others
- -inf is equivalent to -inf, thus -inf cmp_op -inf is true for {==, <=, >=} and false for others

Behavior of comparison involving NaN is

- NaN cmp_op anything (including NaN itself) is false, for cmp_op = {<, <=, >, >=, ==}.
- NaN != anything (including NaN itself) is true.

Note that FP comparison produces an integer outcome, so it DOES NOT output NaN or set the sticky invalid status bit.

Note that the above behavior and corner cases apply both to scalar and vector, hfloat (FP16) type and float (FP32) type.

6.2.4.4 FP Conversion Corner Cases

FP conversion can produce +/- Inf in case of converting int24/int32/int48/fp32 into fp16 and can produce NaN in case of conversion between fp32 and fp16. However, FP conversion DOES NOT set the sticky invalid status bit even when outcome is NaN.

The following table shows scalar and vector floating-point conversion corner cases:

Table 9. FP/INT conversion corner cases

Conversion Function	+/- Inf	NaN
INT_FP VINT_FP	Output +/- Inf not possible, as INT32_MIN / INT32_MAX can be presented in normal FP32 numbers	Output NaN is not possible
INT_FP16 VINT_FP16	Output +/- Inf is possible from values not representable in FP16	Output NaN is not possible
VINTX_FP	Output +/- Inf not possible, as INT48_MIN / INT48_MAX can be presented in normal FP32 numbers	Output NaN is not possible
VINT24_FP16	Output +/- Inf is possible from values not representable in FP16	Output NaN is not possible
FP_INT_Trunc/Round VFP_INT_Trunc/Round	Input +/- Inf converts to output INT32_MIN / INT32_MAX	Input NaN converts to output INT32_MIN / INT32_MAX
FP16_INT_Trunc/Round VFP16_INT_Trunc/Round	Input +/- Inf converts output INT32_MIN / INT32_MAX	Input NaN converts to output INT32_MIN / INT32_MAX
VFP_INTX_Trunc/Round	Input +/- Inf converts to output INT48_MIN / INT48_MAX	Input NaN converts to output INT48_MIN / INT48_MAX
VFP16_INT24_Trunc/Round	Input +/- Inf converts to output INT24_MIN / INT24_MAX	Input NaN converts to output INT24_MIN / INT24_MAX
FP_FP16 VFP_FP16	Output +/- Inf is possible from +/- Inf and values not representable in FP16	Input NaN converts to output NaN
FP16_FP VFP16_FP	Input +/- Inf converts to output +/- Inf	Input NaN converts to output NaN

6.2.4.5 FP Conversion to/from Fixed-Point Formats

Some of the VPU scalar/vector FP/Integer conversion instructions support fixed-point conversion by having an argument that conveys qbit of the fixed-point format.

Fixed-point format is one that represents a number having fixed integer and fraction widths using integer representation. There is a qbit configuration parameter, sometimes referred to simply as Q, as in Q8, Q15, and so on, that programmer maintains in software to indicate width of the fraction portion. Qbit can be viewed as the bit position of an imaginary radix point, or boundary between integer bits and fraction bits.

Normally, variables in the same block of computation share the same qbit, so that fixed-point addition and subtraction are performed the same way as integer addition and

subtraction. Fixed-point multiplication is performed as integer multiplication followed by rounding to get back the same qbit configuration, or a different qbit configuration if desirable in the application.

To convert a floating-point number to a fixed-point, we multiply the floating-point number by 2^qbit. To convert from fixed-point to floating-point, we divide the fixed-point number by 2^qbit.

For example, numbers 1.125 and 5.0625 are represented in fixed-point with qbit = 8 as 1.125 * 2^8 = (1 + 1/8) * 256 = 256 + 32 = 288, and 5.0625 * 2^8 = (5 + 1/16) * 256 = 1280 + 16 = 1296.

The sum of the two numbers, 1.125 + 5.0625 = 6.1875, can be carried out as 288 + 1296 = 1584, and converted back to floating-point as 1584 / 256 = 6.1875.

With qbit argument as part of the conversion, the multiplication or division by 2^qbit is performed in hardware as part of the conversion, expanding precision and dynamic range internally in the process, and bring some acceleration to the conversion process.

Not all FP/INT conversions support the qbit argument though. Basically, only a conversion involving FP16 has this feature. FP16 format has relatively limited dynamic range, as its 5 bits of exponent gives +/- 14 range in the exponent in regular (not denormal) FP16 numbers. There are cases where the multiplication or division by 2^qbit involved, if carried out in FP16 math would have caused the number to become +/- Inf in FP16, and if carried out in integer would have overflown the integer representation. Without a qbit argument as part of the conversion, the programmer would have to go through FP32, that is, FP16 → FP32 → multiply 2^qbit in FP32 → Integer, or Integer → FP32 → multiply by 2^-qbit in FP32 → FP16, and would have taken much longer.

For example, the number 128.0 represented in Q8 fixed-point is integer 0x8000, or 2^15. It's representable in INT32 or INT24 (vector extended short lane). If we convert this number from fixed-point to FP16 using standard (no-qbit) conversion and FP16 math, we will convert it first to FP16 then multiplying by 2^-8 in FP16. The first step of converting INT24 or INT32 0x8000 to FP16 would result in +Inf (positive infinity), then +Inf * 2^-8 = +Inf. For this example, it seems we would want to first divide by 2^8 in INT24/INT32, before performing the standard INT24/INT32 to FP16 conversion. However, in general doing that would throw away fractional information that we work hard to obtain and would like to preserve as much and as long as possible in the computation.

Conversely, if we convert 128.0 represented in FP16 to Q8 fixed-point with standard (no-qbit) conversion and FP16 math, we see issues. 128.0 itself we can represent just fine in FP16. However, the multiplying by 2^qbit = 2^8 involved, if performed in FP16, we would see intermediate result becoming +Inf and cannot proceed to be accurately converted to Q8 fixed-point. For this example, it would work if we converted 128.0 in FP16 to INT24/INT32, then we left-shift by 8 bits in INT24/INT32. However, in this process we also throw away fractional portion of the input number, so it would not accurately convert, for example, 128.25, to fixed-point.

Converting FP32 to/from fixed-point would not have the same issue, as FP32 with its 8-bit exponent supports wider dynamic range, -126 ~ +126, much wider than integer side,

so inputs that cause intermediate outcome to become +/-Inf would cause the final converted outcome to be saturated to MAX/MIN integer value for that destination bit width, so there is no loss of information if the multiplying/dividing by 2^qbit is performed in FP32 before/after conversion to/from integer.

6.3 Memory Operations

6.3.1 Memory Coherency

There is memory dependency detection logic to stall the processor pipeline to keep memory coherent.

For this discussion, it is helpful to define coherent vs non-coherent memory operations.

Non-coherent memory operations:

- Transposing load/store
- Table lookup (load)
- Histogram (load and store)
- Vector-addressed store

Coherent memory operations: all other load/store. Each such load/store accesses consecutive memory contents whose size range from one byte to 64 bytes.

The non-coherent accesses are non-consecutive and thus have a wide address range, so it is too expensive to comprehend in the memory dependency stalling logic. Memory access for load is in EX5 stage, whereas memory access for store is later in EX9 stage. Thus, there should be 5 execution packets of separation between storing an item to memory before the loading of that element should be scheduled.

When a coherent store and the subsequent coherent load are detected by hardware to have “close enough” addresses and do not have enough execution packet separation in the code, processor will stall the load to create the separation, so that load would return memory contents after the store. The checking and stalling mechanism keeps the memory operations coherent, or consistent with sequential execution.

To reduce timing pressure, the address checking is simplified (exact for scalar load/store but use just starting row address for vector load/store) and is conservative. Thus, sometimes, a load can be stalled unnecessarily until memory transaction from a previous store is completed.

In case either or both memory operations are non-coherent, there is not enough execution packet separation, and even when there are overlaps in addresses, processors will not stall, causing RAW (read after write) and WAW (write after write) hazards. WAW does not happen between normal store and vector-addressed-store, but can happen between normal store and histogram update, as they occur on different pipeline stages.

To help achieve this separation between non-coherent memory operations, in Orin we have added a memory fence instruction (MemFence) that can be used to avoid memory

coherency issues. The MemFence instruction would inject stalls until all preceding memory store operations are committed. It is a broad (works on all memory operations) and blind (not based on address) fence, so should be used judiciously, otherwise performance may degrade too much.

Note that there is also available a compiler pragma chess_memory_fence() that works similarly as the MemFence instruction. With chess_memory_fence(), compiler inserts as many NOPs as necessary to ensure that memory store operations before the fence are committed before memory operations after the fence can start. One advantage over MemFence instructions is that each MemFence instruction simply inserts stall cycles, and with chess_memory_fence(), the compiler is supposed to schedule useful work when it's possible, so that some useful work may be accomplished while memory operations after the fence are delayed.

Histogram read/write has its own per-bank bypass mechanism (covering only histogram read/write) to implement correct histogram operation despite VMEM latency.

There is RAW hazard detection and handling built-in for the histogram functionality to ensure memory coherency among histogram updates. Note that there is no hazard detection between histogram read/write versus any other load/store accesses, thus the "non-coherent" memory operation designation for histogram.

6.3.2 Memory Address Alignment

Various scalar/vector load/store shall comply with the address alignment constraint and misalignment handling.

In the case of demoting/promoting load/store, we determine alignment based on the data type in memory, versus the data type in register file. For example, QVSTHB, quad vector demoting store from Halfword to Byte, is considered Byte-type store regarding to address alignment.

- Byte-type load/store:
  - Scalar load/store LDB, STB (based-offset, post-modify, agen-based) are 8-bit aligned.
  - Single vector (32 x 8-bit) load/store VLDB, VSTB (based-offset, post-modify, agen-based) are 8-bit aligned.
  - Double vector (64 x 8-bit) load/store DVLDB, DVSTB (post-modify, agen-based) are 16-bit aligned.
  - Promoting/demoting load/store resulting in 32 x 8-bit memory access, VLDBH, VLDBW, DVSTHB, are 8-bit aligned.
  - Demoting store resulting in 64 x 8-bit memory access, QVSTHB, are 16-bit aligned.
- Halfword-type scalar/vector load/store shall be 16-bit aligned.
- Word-type scalar/vector load/store shall be 32-bit aligned.
- Extended-word type vector load/store can be leveraged for extended Byte/Halfword/Word types (12/24/48-bit), shall be 16-bit aligned.

- Table lookup, histogram, vector-addressed store base address should be 512-bit aligned, so each 8-bit element is 8-bit aligned, 16-bit element 16-bit aligned, and 32-bit element 32-bit aligned.
  - VLUT_*, DVLUT_*
  - VHIST_*, DVHIST_*
  - DVAST_*
- Agen configuration (512-bit) load/store should be 32-bit aligned.
- AgenCfgLD, AgenCfgST
- Lane predicated vector stores would behave, in terms of address alignment, as unpredicated vector stores.
- Unsigned load would behave as the corresponding signed load (keeping all other attributes the same), in terms of address alignment.

The hardware enforces the alignment constraint by forcing the lowest {1, 2, 6} bits of the byte address to zero, based on the alignment requirement being 16-bit, 32-bit, or 256-bit. For 8-bit address alignment, the byte address is not altered.

6.3.3 Memory Address Range Constraints

Load/store addresses should be in valid range consistent with the address map:

- Superbank A: 0x00000 ~ 0x1FFFF
- Superbank B: 0x40000 ~ 0x5FFFF
- Superbank C: 0x80000 ~ 0x9FFFF

Any single-item load/store should have base address inside the valid range. Any multiple-item load/store should have base address sufficiently away from the end of each superbank range, such that no data item would fall out of the valid range. For example, software should avoid issuing a load or store starting 0x1FFE0 and spanning more than 32 bytes. An exception is lane-predicated store, if prediction is off for the part of store data going outside the valid range.

In case a multiple-item load/store falls partially or fully outside the valid range, hardware wraps around the access so that the part of load/store falling outside the valid range is mapped back in, to the superbank indicated by the base address.

In case the base address goes outside the valid range, hardware determines the superbank by:

- Address bits 19:18 == “00” → Superbank A
- Address bits 19:18 == “01” → Superbank B
- Address bits 19:18 == “10” or “11” → Superbank C

However, software should not take advantage of such a wrap-around, as address map changes in future generations can change the address wrap-around and make the software not work.

6.3.4 Scalar Data Types

Byte, half-word and word types are supported. Signed/unsigned flavors of load for byte and half-word are supported to properly sign or zero-extend into 32-bit scalar register entry. Store operations are signed/unsigned agnostic so there is just one flavor.

Table 10. Scalar load/store data types

Element type	Size in memory	Size in scalar register	Memory alignment
B/BU: signed/unsigned byte	8-bit	32-bit	8-bit
H/HU: signed/unsigned half-word	16-bit	32-bit	16-bit
W/WU: signed/unsigned word	32-bit	32-bit	32-bit

Note that hardware does not tag each scalar register carrying signed or unsigned data, where behavior is different, signed and unsigned flavors of scalar math operations are offered, so programmer should choose signed/unsigned flavors in scalar load and scalar math operations appropriately.

6.3.5 Vector Data Types and Promotion/Demotion

Scalar-based load/store can have immediate offset (10-bit) or can be post-modified with a second scalar register. Only parallel distribution mode is available, loading 256-bit or 512-bit from memory to write into single or double vector register, or storing single or double vector register into 256-bit or 512-bit in memory. The WX type allows storing the raw bits tightly packed as 384-bit, and can be used to load/store B, H, or W-type vector registers.

Data types supported for scalar-based vector load/store:

Table 11. Scalar-based vector load/store data types

Element type	Vector size	Size in memory	Size in vector register	Memory alignment
B/BU: signed/unsigned byte	single	32 x 8-bit vchar/vuchar	32 x 12-bit vcharx	8-bit
	double	2 x 32 x 8-bit dvchar/dvuchar	2 x 32 x 12-bit dvcharx	16-bit
H/HU: signed/unsigned half-word	single	16 x 16-bit vshort/vushort	16 x 24-bit vshortx	16-bit
	double	2 x 16 x 16-bit dvshort/dvushort	2 x 16 x 24-bit dvshortx	16-bit
W/WU: signed/unsigned word	single	8 x 32-bit vint/vuint	8 x 48-bit vintx	32-bit
	double	2 x 8 x 32-bit dvint/dvuint	2 x 8 x 48-bit dvintx	32-bit
WX: extended precision (VRF, WRF)	single	8 x 48-bit vintx	8 x 48-bit vintx	32-bit

Agen-based load/store offers more flexibility in data types. In addition to standard data bytes, some types of promotion and demotion cases are supported.

Note that Load-Permute instruction type designations DVLDPermHB/HBU are not included, as these type designations refer to data types in processing steps, permute as Halfword and zero/sign extend as Byte, and are not indicating type demotion functionality.

Table 12 Agen-based vector load/store data types

Type name	Size in memory	Size in vector register	Memory alignment
B/BU: signed/unsigned byte load B: signed byte store	single: 32 x 8-bit vchar/vuchar double: 64 x 8-bit dvchar/dvuchar	32 x 12-bit vcharx 2x 32 x 12-bit dvcharx	single: 8-bit double: 16-bit
H/HU: signed/unsigned half-word load H: signed half-word store	single: 16 x 16-bit vshort/vushrot vhfloat double: 32 x 16-bit dvshort/dvushort dvhfloat	16 x 24-bit vshortx vhfloatx 2x 16 x 24-bit dvshortx dvhfloatx	16-bit
W/WU: signed/unsigned word load W: signed word store	single: 8 x 32-bit vint/vuint	8 x 48-bit vintx	32-bit

Type name	Size in memory	Size in vector register	Memory alignment
	vfloat double: 16 x 32-bit dvint/dvuint dvfloat	vfloatx 2x 8 x 48-bit dvintx dvfloatx	
BH/BHU: byte to half-word promoting load	double: 32 x 8-bit vchar/vuchar	2x 16 x 24-bit dvshortx	8-bit
BW/BWU: byte to word promoting load	double: 16 x 8-bit n/a (half of vchar/vuchar)	2x 8 x 48-bit dvintx	8-bit
HW/HWU: half-word to word promoting load	double: 16 x 16-bit vshort/vushort	2x 8 x 48-bit dvintx	16-bit
BH: extended byte to half-word promoting store	single: 32 x 16-bit dvshort	32 x 12-bit vcharx	16-bit
HW: extended half-word to word promoting store	single: 16 x 32-bit dvint	16 x 24-bit vshortx	32-bit
HB: half-word to byte demoting store	quad: 64 x 8-bit dvchar double: 32 x 8-bit vchar	4 x 16 x 24-bit 2 x dvshortx 2 x 16 x 24-bit dvshortx	16-bit
WH: word to half word demoting store	quad: 32 x 16-bit dvshort double: 16 x 16-bit vshort	4 x 8 x 48-bit 2x dvintx 2 x 8 x 48-bit dvintx	16-bit
WH: word to half word demoting store from DXAC	double: 32 x 16-bit dvshort	2 x 16 x 32-bit dxvshortx	16-bit
WX: single vector register full 384-bit load/store (no rounding and saturation support)	single: 8 x 48-bit vintx	8 x 48-bit vintx	16-bit
W: single XARF full 512-bit store	single: 16 x 32-bit xvshortx	16 x 32-bit xvshortx	32-bit

While in scalar/vector math we use “F” and “HF” type designation to denote float and hfloat data types, in memory operations, float and hfloat are treated like int and short respectively and are thus mapped to “W” and “H” type designations.

6.3.6 Vector Load/Store Distribution Options

Various data distribution options are supported for vector load/store:

- S: scalar (load 1 element and broadcast to all lanes, store first lane), single register (storing first lane of vector register) or double register (storing first lane of .lo single vector and first lane of .hi single vector)

- P: parallel (1-to-1), single or double register
- T: transposing, having constant offset between elements, single or double register
- PDI: parallel double register deinterleaving (load-only)
- PI: parallel double register interleaving (store-only)
  - parallel quad register 4-way interleaving (store-only)
- TDI: transposing double register deinterleaving (load-only)
- TI: transposing double register interleaving (store-only)
- PI2: alternate form of quad register interleaving (store-only)
- C2: circulate between 2 data points, single register (load only)
- T2: transposing after every pair of elements (double Word vector load/store)
- T2DI/T2I: T2 with deinterleaving load or with interleaving store (double Word vector load/store, double Halfword vector load)
- T4: transpose every 4 data elements
- T8: transpose every 8 data elements
- T16: transpose every 16 data elements
- T32: transpose every 32 data elements

Interleaving/deinterleaving is to offer data access flexibility as well as to deal with MAC datapath interleaving in the lane-expanding cases. For double-register deinterleaving load, we take memory items and interleave (deal) into the two vector registers. For double-register interleaving store, we interleave (shuffle) data from two vector registers to sequential items in the memory. For quad-register interleaving store, we interleave each pair, then between the two pairs.

For example, “QVSTWH_P V0:V1, V2:V3, *A0++” would store out (indexing word lanes of each register):

```
V0[0], V0[1], …, V0[7], V1[0], V1[1], …, V1[7],
V2[0], V2[1], …, V2[7], V3[0], V3[1], …, V3[7]
```

The 4-way interleaving version, QVSTWH_PI V0:V1_V2:V3 would store out:

```
V0[0], V2[0], V1[0], V3[0], V0[1], V2[1], V1[1], V3[1], …, V0[7], V2[7], V1[7], V3[7]
```

where the lowest 16-bit of each word lane is stored out in half-word spacing.

The 4-way interleaving QVSTHB_PI V0:V1, V2:V3 has a similar data pattern, with input elements pulled from half-word (24-bit) lanes and stored out as bytes.

```
V0[0], V2[0], V1[0], V3[0], V0[1], V2[1], V1[1], V3[1], …, V0[15], V2[15], V1[15], V3[15]
```

Alternative interleaving pattern in QVSTWH_PI2 V0:V1, V2:V3, each element being 48-bit input from register, 16-bit output in memory:

```
V0[0], V1[0], V0[1], V1[1], … , V0[7], V1[7], V2[0], V3[0], V2[1], V3[1], … , V2[7], V3[7]
```

Alternative interleaving pattern in QVSTHB_PI2 V0:V1, V2:V3, each element being 24-bit input from register, 8-bit output in memory:

```
V0[0], V1[0], V0[1], V1[1], … , V0[15], V1[15], V2[0], V3[0], V2[1], V3[1], … , V2[15], V3[15]
```

Another way to compare with the “_P” distribution option is to look at V0, V1, V2, V3 each as an 8 (in case of WH type) or 16 (in case of HB type) -element array.

- QVST*_P stores out V0 + V1 + V2 + V3, “+” being concatenation.
- QVST*_PI stores out interleave(interleave(V0, V1), interleave(V2, V3)).
- QVST*_PI2 stores out interleave(V0, V1) + interleave(V2, V3).

A load with “C2” distribution, for example, “VLDW_C2 *A0++, V0” would read the first 2 32-bit words from the location pointed by agen A0, say x[0] and x[1], and distribute them such that

```
V0 = {x[0], x[1], x[0], x[1], x[0], x[1], x[0], x[1]}, seen as word (48-bit) lanes.
```

6.3.7 Transposing Load/Store

Transposing load/store accesses array elements vertically when the memory contents is viewed with the configured line pitch. Here, line pitch is defined by number of elements.

Six transposition modes are supported, designated as T, T2, T4, T8, T16 and T32. T is the normal transposition mode, and is supported broadly, for all Byte/Halfword/Word types and various promotion/demotion types, single and double vector load/store. T<n> transposition, n being a power of 2 from 2 to 32, reads/writes n consecutive data points before applying the line pitch address offset.

Not all line pitch values are possible. Constraints on the line pitch are dependent on the data type and the transposition mode, as shown in the following table.

Table 13. Line pitch constraint for various transposition modes

Trans-position mode	Single/double vector - type - load/store	Line pitch constraint	Programmed into lane_ofst (12-bit unsigned)
T	Single/double Word load (32-bit → 48-bit) Single/double Word store (48-bit → 32-bit) Single HW promoting store (24-bit → 32-bit)	16k + 1	k
	Single/double Halfword load (16-bit → 24-bit) Single/double Halfword store (24-bit → 16-bit) Double HW promoting load (16-bit → 24-bit) Double BH promoting store (12-bit → 16-bit) Double/quad WH demoting store (48-bit → 16-bit)	32k + 1	k
	Single Byte load (8-bit → 12-bit) Single Byte store (12-bit → 8-bit) Double BH promoting load (8-bit → 24-bit) Double BW promoting load (8-bit → 48-bit) Double/quad HB demoting store (24-bit → 8-bit)	64k + 2	k
T2	Double Word load (32-bit → 48-bit) Double Word store (48-bit → 32-bit)	16k + 2	k
	Double Halfword load (16-bit → 24-bit) Double Halfword store (24-bit → 16-bit)	32k + 2	k
T4	Double Halfword load (16-bit → 24-bit) Double Halfword store (24-bit → 16-bit)	32k + 4	k
T8	Double Word load (32-bit → 48-bit) Double Word store (48-bit → 32-bit)	16k + 8	k
	Double Halfword load (16-bit → 24-bit) Double Halfword store (24-bit → 16-bit)	32k + 8	k
T16	Double Halfword load (16-bit → 24-bit) Double Halfword store (24-bit → 16-bit)	32k + 16	k
T32	Double Byte load (8-bit → 12-bit) Double Byte store (12-bit → 8-bit)	64k + 32	k

It is allowed to program k = lane_ofst = 0, so that the transposing load/store behaves like normal (consecutive) load/store in address calculation. Behavior is still different than normal (consecutive) load/store, in the sense that degenerate transposing memory transactions are still noncoherent and can be used intentionally to avoid unnecessary memory stalls. Please see Memory Coherency for details.

In case there is a type promotion or demotion in transposing load/store, it's the data type in memory that dictates which line pitch constraint to use.

For Byte type we only support single vector T transposition load/store. For Halfword and Word types, both single vector and double vector T transposition load/store are supported.

In general, transposing load/store calculates byte addresses for each element as follows for the normal transposition (T):

```
M = data size in bytes, 1, 2 or 4 for Byte/Halfword/Word type
P = (M == 1) ? (64*K + 2) : (64*K + M)      // line pitch in bytes, K provided by agen lane_ofst
byte_address[i] = (base & SUPERBANK_SELECT)
                + alias_within_superbank((base + i*P), i = 0 .. num_lanes – 1
```

With this address calculation, adjacent lanes are P (pitch in bytes) apart in memory.

The first term of byte address is for superbank selection, which is affected only by the base address, not by any index. As each superbank occupies 256KB of space (256K = 2^18), including aliased region, and we have 4 superbanks, we look at bits 19 and 18 of byte address to select superbank:

```
SUPERBANK_SELECT = 0xC0000
```

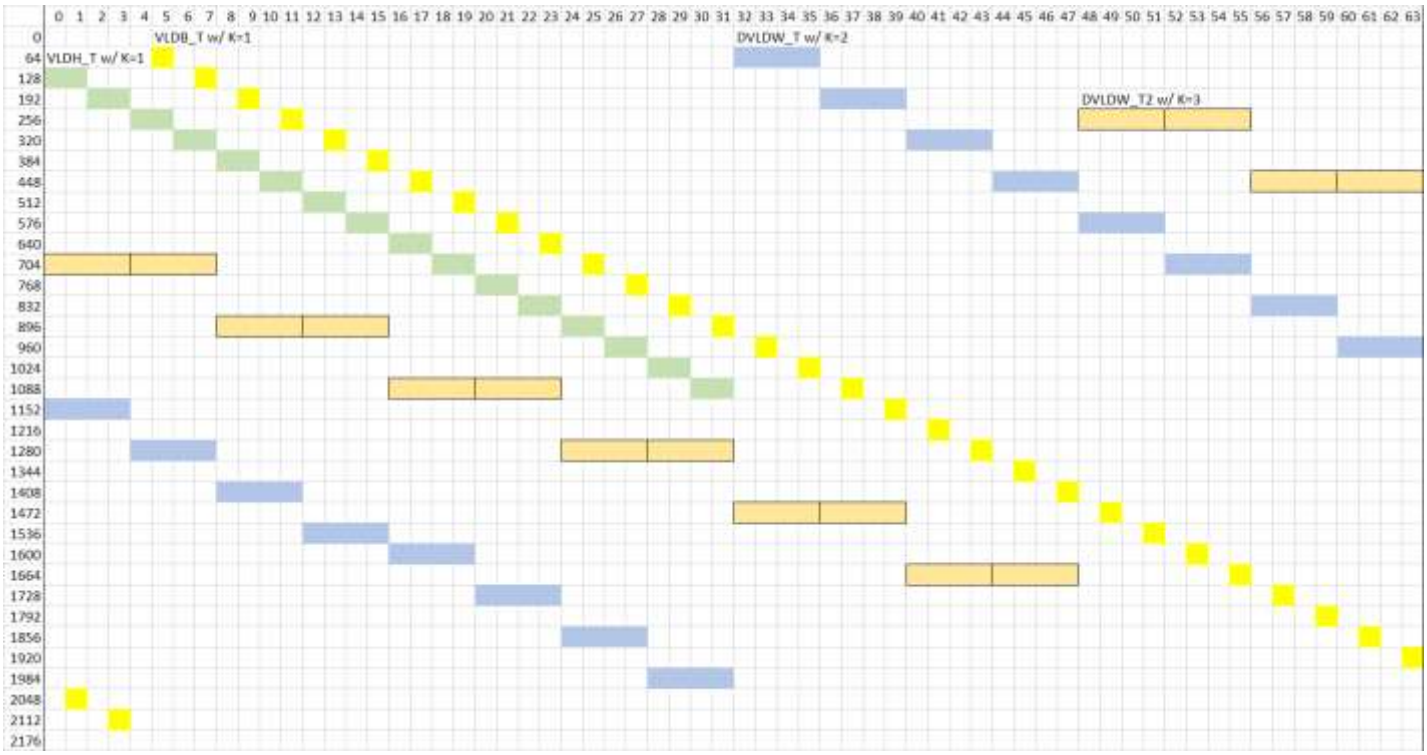
For the T2 element-pair transposition, we carry out the following address calculation:

```
M = 2 for Halfword type, 4 for Word type
P = 64*K + 2*M // line pitch in bytes, K provided by agen lane_ofst
byte_address[2*i] = (base & SUPERBANK_SELECT)
                  + alias_within_superbank ((base & BASE_MASK) + i*P)
byte_address[2*i+1] = (base & SUPERBANK_SELECT)
                    + alias_within_superbank ((base & BASE_MASK) + i*P + M)
                    i = 0 .. num_lanes/2 – 1, where BASE_MASK = 0x1FFC0.
```

With this address calculation, adjacent lanes are alternately M and 64*K+M apart in memory.

The following diagram shows examples of T and T2 transposition access patterns. Note that for Byte type, we write either all even bytes of every halfword or all odd bytes of every halfword, depending on the LSB of byte address.

Figure 9. Access patterns of transposition modes T and T2



T4, T8, T16, and T32 transposition modes are supported in selective load/store instructions. Halfword type is more heavily used than the other types in computer vision, and double vector load/store leverages full throughput of VMEM, so double vector Halfword load/store supports all the transposition modes. Other type-transposition combinations are supported where there is demand among use cases.

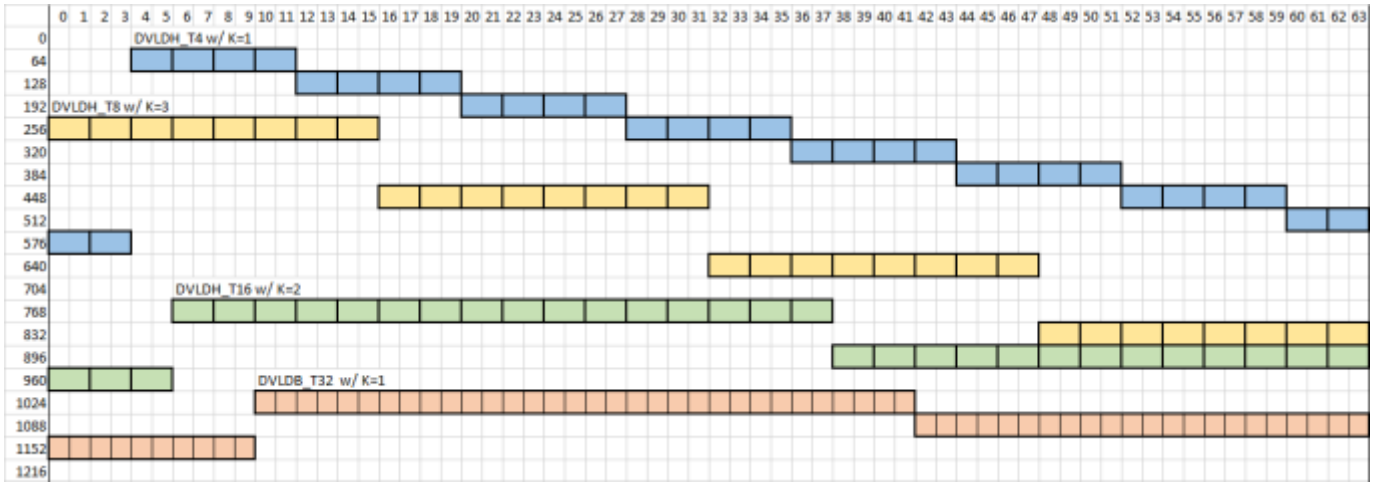
In general, line pitch in bytes for T<n> transposition is derived as

```
P = 64 * lane_offset + n * sizeof_data_type
```

The access pattern is that we would access consecutively n data elements before taking the line pitch address offset to move down to the next line.

The following diagram shows H_T4, H_T8, H_T16, B_T32 transposition access pattern.

Figure 10. Access patterns of transposition modes T4, T8, T16 and T32



The unsigned 12-bit lane offset is applied up to 31 times among the transposition options of a load/store instruction, and line pitch is 64 bytes times the lane offset, so the full range of unsigned 12-bit lane offset value can lead the raw address to map far outside the superbank the base address is pointed to. The extreme case is with single-vector byte-type T transpose, 31 * (64 * 4095 + 2), almost 8 Mega Bytes (with Mega being 1024^2).

It is allowed to program k = lane_ofst = 0, so that the transposing load/store behaves like normal (consecutive) load/store in address calculation in all cases except for byte-type T transposing load/store. Because we have 16-bit memory banks, byte-type T transposing load/store with zero lane offset would be reading/writing **every-other** byte instead of consecutive bytes.

In cases where address patterns of degenerate transposing load/store are identical with that of normal/consecutive load/store, the exact behavior is still different, in the sense that degenerate transposing memory transactions are still non-coherent transactions from memory coherence stall logic point of view, meaning there is no address proximity checks to stall memory transactions automatically. In certain cases, the degenerate transposing load/store can be used intentionally to avoid memory coherency stalls, but programmers should be extremely careful with its use.

The alias_within_superbank function in address calculation keeps subsequent accesses within one superbank. Only in B_T32/H_T16/W_T8 distribution options, where the line pitch is applied only once, would we make use of almost-full range of lane offset. It's also for future extension of VMEM capacity. Programmers are strongly discouraged to intentionally allow address to go outside physical memory and rely on address aliasing in the hardware. Such code may not work in the future when address map changes.

6.3.8 Parallel Lookup, Histogram and Vector-Addressed Store

PVA supports parallel table lookup and histogram through table/bin replication, taking advantage of the memory banking organization in VMEM superbanks.

Vector-addressed store, also called reverse parallel lookup, takes a scalar base address (512-bit = 64-byte aligned), a vector of indices, a vector of data values, and writes the data values into the indexed entries. Often, per-lane predication is required to perform store on selected lanes.

Table lookup:

- 1/2/4/8/16 x W/WU word index, word table entry
- 1/2/4/8/16/32 x H/HU halfword index, halfword table entry
- 1/2/4/8/16/32 x B/BU byte index, byte table entry
- 1/2/4/8/16/32 x HB/HBU halfword index, byte table entry

Histogram:

- 1/2/4/8/16 W word index, word histogram bin
- 1/2/4/8/16/32 H halfword index, halfword histogram bin

Vector-addressed store:

- 16W word index, word store data
- 32H halfword index, halfword store data

Note that only memory slot 0, M0, supports lookup, histogram, and vector-addressed store features.

6.3.8.1 Data Organization in Memory

Table/histogram/VAST data organization is as follows:

Figure 11. Parallel lookup, histogram and VAST data organization for various types and parallelism

16-parallel Word-type:

T0[0]	T1[0]	...	T15[0]
T0[1]	T1[1]	...	T15[1]

...

8-parallel Word-type:

T0[0..1]	T1[0..1]	...	T7[0..1]
T0[2..3]	T1[2..3]	...	T7[2..3]

...

4-parallel Word-type:

T0[0..3]	T1[0..3]	...	T3[0..3]
T0[4..7]	T1[4..7]	...	T3[4..7]

...

2-parallel Word-type:

T0[0..7]	T1[0..7]
T0[8..15]	T1[8..15]

...

1-parallel Word-type:

T0[0..15]
T0[16..31]

...

32-parallel Half-word-type:

T0[0]	T1[0]	T2[0]	...	T31[0]
T0[1]	T1[1]	T2[1]	...	T31[1]

...

16-parallel Half-word-type:

T0[0..1]	T1[0..1]	...	T15[0..1]
T0[2..3]	T1[2..3]	...	T15[2..3]

...

8-parallel Half-word -type:

T0[0..3]	T1[0..3]	...	T7[0..3]
T0[4..7]	T1[4..7]	...	T7[4..7]

...

4-parallel Half-word -type:

T0[0..7]	T1[0..7]	...	T3[0..7]
T0[8..15]	T1[8..15]	...	T3[8..15]

...

2-parallel Half-word -type:

T0[0..15]	T1[0..15]
T0[16..31]	T1[16..31]

...

1-parallel Half-word -type:

T0[0..31]
T0[32..63]

...

32-parallel Byte-type:

T0[0..1]	T1[0..1]	T2[0..1]	...	T31[0..1]
T0[2..3]	T1[2..3]	T2[2..3]	...	T31[2..3]

...

16-parallel Byte -type:

T0[0..3]	T1[0..3]	...	T15[0..3]
T0[4..7]	T1[4..7]	...	T15[4..7]

...

8-parallel Byte -type:

T0[0..7]	T1[0..7]	...	T7[0..7]
T0[8..15]	T1[8..15]	...	T7[8..15]

...

4-parallel Byte -type:

T0[0..15]	T1[0..15]	...	T3[0..15]
T0[16..31]	T1[16..31]	...	T3[16..31]

...

2-parallel Byte -type:

T0[0..31]	T1[0..31]
T0[32..63]	T1[32..63]

...

1-parallel Byte -type:

T0[0..63]
T0[64..127]

...

6.3.8.2 Table Lookup

VMEM Superbanks support parallel table lookup with the following data element size and parallelism combinations:

- For byte element size, 1/2/4/8/16/32 ways of parallelism
- For half-word (16-bit) element size, 1/2/4/8/16/32 ways of parallelism
- For word (32-bit) element size, 1/2/4/8/16 ways of parallelism

The VPU sends a table base address (512-bit or 64-byte aligned) and an index vector to the VMEM interface (VMEM I/F). The VPU also sends along addressing mode (to convey that it's a table lookup transaction), element size and parallelism as sideband signals. The first K elements of the index vector are consumed for K-way lookup; the rest are ignored.

The VMEM I/F decodes the upper bits of the base address and forwards all signals pertaining to the lookup access to the addressed superbank.

The superbank carries out the lookup, extracts the K table entries from memory according to the base address and the index vector, and sends an outcome vector through the VMEM I/F back to the VPU. The first K elements of the outcome vector are consumed by the VPU; the rest are ignored.

The 32 16-bit memory banks are divided evenly to support the various lookup parallelisms. For example, for 4-way word-size lookup, the 32 memory banks are evenly divided into 4 parallel tables, with each table residing in 8 16-bit memory banks. Please see Section 6.3.8.1 for table data organization for various data type and parallelism combinations.

In Gen-2 VPU we have added 2-point lookup and 2x2-point lookup.

Table 14. Table lookup 2-point and 2x2-point support

Instruction	Memory object	Index vector	Outcome vector
VLUT_*B VLUT_2pt_*B VLUT_2x2pt_*B	signed byte (8-bit)	sign-extended from byte lane (12-bit)	signed byte (12-bit)

Instruction	Memory object	Index vector	Outcome vector
VLUT_*BU VLUT_2pt_*BU VLUT_2x2pt_*BU	unsigned byte (8-bit)	sign-extended from byte lane (12-bit)	signed byte (12-bit)
[D]VLUT_*H [D]VLUT_2pt_*H [D]VLUT_2x2pt_*H	signed half-word (16-bit)	Up to 16 LSBs from half-word lane (24-bit)	signed half-word (24-bit)
[D]VLUT_*HU [D]VLUT_2pt_*HU [D]VLUT_2x2pt_*HU	unsigned half-word (16-bit)	Up to 16 LSBs from half-word lane (24-bit)	signed half-word (24-bit)
[D]VLUT_*W [D]VLUT_2pt_*W [D]VLUT_2x2pt_*W	signed word (32-bit)	Up to 15 LSBs from word lane (48-bit)	signed word (48-bit)
[D]VLUT_*WU [D]VLUT_2pt_*WU [D]VLUT_2x2pt_*WU	unsigned word (32-bit)	Up to 15 LSBs from word lane (48-bit)	signed word (48-bit)
VLUT_*HB VLUT_2pt_*HB VLUT_2x2pt_*HB	signed byte (8-bit)	Up to 17 LSBs from half-word lane (24-bit)	signed byte (12-bit)
VLUT_*HBU VLUT_2pt_*HBU VLUT_2x2pt_*HBU	unsigned byte (8-bit)	Up to 17 LSBs from half-word lane (24-bit)	signed byte (12-bit)

6.3.8.3 Histogram

VMEM Superbanks support parallel histogram with the following data element size and parallelism combinations:

- There is no byte element size support
- For half-word (16-bit) element size, 1/2/4/8/16/32 ways of parallelism
- For word (32-bit) element size, 1/2/4/8/16 ways of parallelism

Since each superbank supports one read transaction and one write transaction per cycle, histogram reads and writes are pipelined, to achieve up to 32 histogram updates per cycle, in case of 32-way half-word case.

The VPU sends a histogram base address (512-bit or 64-byte aligned), an index vector and an update vector to the VMEM interface (VMEM I/F). The VPU also sends along addressing mode (to convey that it's a histogram transaction), element size and parallelism as sideband signals. The first K elements of the index vector and the update vector respectively are consumed for K-way histogram; the rest are ignored.

The VMEM I/F decodes the upper bits of the base address and forwards all signals pertaining to the histogram access to the addressed superbank.

The superbank carries out the histogram update, reads the K histogram bins from memory according to the base address and the index vector, adds the update vector to the bins, writes the updated bins back to memory (where each bin came from), and sends the before-update bins as an outcome vector through the VMEM I/F back to the VPU. The first K elements of the outcome vector are consumed by the VPU; the rest are ignored.

The 32 16-bit memory banks are divided evenly to support the various histogram parallelisms. For example, for 4-way word-size histogram, the 32 memory banks are evenly divided into 4 parallel histograms, with each histogram residing in 8 16-bit memory banks. See Data Organization in Memory for histogram data organization for various data type and parallelism combinations.

Compared to conventional/normal histogram, VPU parallel histogram feature implements weighted histogram (by allowing an update vector to be added instead of only incrementing by one), and supports bin value read-back, which is useful in sorting and decision tree applications to bin records or features for further processing.

Table 15. Histogram support

Instruction	Memory object (input & outcome)	Index & weight vectors	Outcome vector
[D]VHIST_*H	signed half-word (16-bit)	Up to 16 LSBs from half-word lane (24-bit)	signed half-word (24-bit)
[D]VHIST_*W	signed word (32-bit)	Up to 15 LSBs from word lane (48-bit)	signed word (48-bit)
[D]VHIST_OR_*H	signed half-word (16-bit)	Up to 16 LSBs from half-word lane (24-bit)	signed half-word (24-bit)
[D]VHIST_OR_*W	signed word (32-bit)	Up to 15 LSBs from word lane (48-bit)	signed word (48-bit)

6.3.8.4 Vector Addressed Store

VMEM Superbanks support vector addressed store, which is also called reverse lookup, since instead of reading back indexed entries, data is written to the indexed entries. We support the maximal parallelism, 32 half-word and 16-word configurations.

Table 16. Vector addressed store support

Instruction	Memory object (outcome)	Index & data vectors	Outcome vector
DVAST_32H	signed half-word (16-bit)	11 LSBs from half-word lane (24-bit)	n/a
DVAST_16W	signed word (32-bit)	11 LSBs from word lane (48-bit)	n/a

Basically, each index lane is sign-extended where insufficient to cover a whole superbank, otherwise appropriate number of LSBs taken to cover a whole superbank.

When we are extending, it's always sign-extended, as opposed to complying with signed/unsigned designation in the lookup instruction (which is used to sign/zero-extend table/histogram entry).

In the case of byte indices (which is normally for byte entries), since a superbank has 128KB, 17 bits are needed for 1-way lookup, 16 bits for 2-way lookup (each way containing 64KB), and so on, to 12 bits needed for 32-way lookup (each way containing 4KB). We would sign-extend from 12-bit byte lane.

For the conventional lookup providing starting address of the table as the base, byte-indexed lookup can only cover 2KB for 1-way, 4KB for 2-way, and so on, to 64KB for 32-way. Due to the limited table size coverage, we also support using halfword indices for byte-entry table lookup.

In the case of halfword and word entry (which is only possible to go with halfword and word indices), we have more than sufficient bit width in each index lane to cover a full superbank, so only an appropriate number of LSBs are used. The address calculation is signed/unsigned agnostic (except when we need to sign/zero-extend for the case of byte indices), so it's safe to treat indices as unsigned, which is how table lookup is naturally implemented.

In case of VAST, only maximal parallelism is supported for each type (32H and 16W), so the index is used to point to each 64-byte-aligned wide memory word. Thus, there is just one bit width used, 11-bit, as superbank size 128KB is 2K x 64B.

The superbank to access is determined solely by the base address. There is no out-of-bound memory access detection; large index values can cause the resulting address to land outside the intended table or histogram object in the same superbank in VMEM.

Also, taking some LSBs of the indices, ignoring upper bits, is essentially performing index wrap-around in the same superbank, but not in the table/histogram/VAST-object, as there is no way to indicate size of the table/histogram/VAS-object to the processor. It is the programmer's responsibility to ensure that lookup/histogram/VAST operations do not index outside the intended memory range or suffer the consequences.

For example, a 4KB 32-way H-type lookup table has only 4KB/2/32 = 64 entries in each sub-table. If/when the base address is the starting address of the table, in conventional non-negative indexing, only [0, 63] in index range makes sense. If the base address is right in the middle of the table (starting address + 2KB), for a symmetrical signed indexing, only [-32, 31] range makes sense. A whole superbank can be reached by the lookup, with up to 128KB/2/32 = 2K entries. An index value of 2048 would behave the same as 0, and full range of index values in 24-bit vector lane would wrap 8192 times (ignoring upper 14 bits) around the superbank and can access data outside the allocated 4KB table.

Address Calculation

Parallel lookup, histogram, and vector address store addressing involves taking the prescribed number of indices, separating the indices into vertical and horizontal components, and accessing the table entry with the vertical/horizontal indices in the appropriate sub-table.

For example, 4-way parallel lookup of byte type would organize the table memory as 4 banks of 16 entries wide sub-tables, using the 4 LSBs of index horizontally within the row of 16 entries fetched for a sub-table, and the upper bits vertically to pick the row. Address calculation for the parallel lookup can be expressed as:

```
lut_out[i] = table[ (index[i] & 0xF) + i*16 + (index[i]>>4)*64 ], for i = 0..3
```

In general, for M bytes-per-point data type, N-way parallel lookup, we calculate stride K = (64/M)/N = 64/(M*N) = number of entries per table on the same memory line (512 bits = 64 bytes per line). Hardware accesses table entries at byte addresses

```
byte_offset[i] = ((index[i] modulo K) + i*K) * M + floor(index[i] / K)*64, for i = 0..N-1
```

Essentially, the table index is partitioned into two pieces, the modulo K piece for indexing consecutive entries in a memory line, and the quotient divided by K piece for addressing memory lines. As K is a power of two (since parallelism N, data size M and 64 are all powers of two), the modulo and the divide operations are implemented as bitwise AND and right shift.

```
byte_address[i] = (base & SUPERBANK_SELECT)
                  + ((base & BASE_MASK) + byte_offset[i] ) & SUPERBANK_MASK
```

The first term of byte address is for superbank selection, which is affected only by the base address, not by any index. For the first generation, we have

```
SUPERBANK_SELECT = 0xC0000
BASE_MASK = 0x1FFC0,
SUPEBANK_MASK = 0x1FFFF.
```

For two-point lookup, DVLUT_2pt, up to 16 indices (consistent with the parallelism designation) are used to calculate byte_offset and byte_address described above. Then, same number of additional indices, index[i] + 1, go through the same calculation to perform up to 32 lookups per DVLUT_2pt instruction. See 9.9.6.4 DVLUT_2pt instruction description for details.

For 2x2-point lookup, DVLUT_2x2pt, up to 8 indices (consistent with the parallelism designation) are used to calculate byte_offset and byte_address described above. Then, 3 times the number of additional indices, index[i] + 1, index[i] + LP, index[i] + LP + 1, go through the same calculation to perform up to 32 lookups per DVLUT_2pt instruction. LP here is line pitch and is derived from the PL register. See 9.9.6.5 DVLUT_2x2pt instruction description for details.

Vector addressed store is also called reverse lookup, as instead of retrieving indexed entries from memory, write values are to be written to the indexed locations. It is useful for list-based processing.

6.4 Address Generator Features

Address generator, or agen, is a unique feature in VPU instruction set architecture. Agen moves much of the multi-dimensional address calculation prominent in image and vision processing to the background and carried out by hardware, improving performance and power in common image and vision processing.

6.4.1 Multi-Dimensional Address Calculation

Agen configuration includes address generator and various other load/store parameters to accelerate regular load/store operations.

Each address generator supports up to 6-dimensional address calculation with its own set of n1..n6 number of iteration parameters, amod1..amod6 address modifiers, and loop variables i1..i6. Agen can be viewed as supporting 6-level nested for loop, with level 1 is being the inner-most loop, and level 6 being the outer-most loop.

For cases when we do not need all 6 dimensions, the convention is to use the lower-numbered variables and set the higher-numbered variables to default values. For example, 2D agen should have

```
n3 = n4 = n5 = n6 = 1
amod3 = amod4 = amod5 = amod6 = 0
```

The Agen supports 6-dimensional address calculation by realizing this function:

```
address(i1, i2, i3, i4, i5, i6) = base + item_size * (i1*w1 + i2*w2 + i3*w3 + i4*w4 + i5*w5 + i6*w6),
```

In this example, w1..w6 are the weights we place on the loop variables i1..i6. We can also visualize w1..w6 as the step amount, in data elements, for each dimension.

Instead of the programmer providing the weights and hardware computing the address via the sum of products expression, the programmer should provide the address modifiers (amod1~amod6), which is the delta of one address to the next address as the 6-dimensional iterator is advanced.

The address modifiers should be calculated as follows:

- Inside i1 loop: amod1 = w1.
- When i1 is reset and i2 is incremented: amod2 = w2 – (n1 – 1)*w1.
- When i1 and i2 are reset and i3 is incremented: amod3 = w3 – (n2 – 1) * w2 – (n1 – 1)*w1.
- When i1, i2 and i3 are reset and i4 is incremented: amod4 = w4 – (n3 – 1)*w3 – (n2 – 1) * w2 – (n1 – 1)*w1.
- When i1, i2, i3 and i4 are reset and i5 is incremented: amod5 = w5 – (n4 – 1)*w4 – (n3 – 1)*w3 – (n2 – 1) * w2 – (n1 – 1)*w1.
- When i1, i2, i3, i4 and i5 are reset and i6 is incremented: amod6 = w6 – (n5 – 1)*w5 – (n4 – 1)*w4 – (n3 – 1)*w3 – (n2 – 1) * w2 – (n1 – 1)*w1.

As the above expressions are tedious to program, there is a set of agen wrapper macros to translate from n1..n6 and w1..w6 into amod1..amod6. Example of programming with agen wrapper will be given in Optimization 2: Leveraging Agen to Collapse Nested Loops.

Agen data structure includes address modifiers as 18-bit fields, and CfgAgen Mod instruction reads 32-bit from the source scalar register and stores only 18 LSBs, dropping the upper 14 bits. Addresses generated from each agen is supposed to be confined within a superbank (128KB = 2^17), so address calculation does not require upper 14 bits.

Behavior of agen-based load/store is post-increment. Data is accessed from the current address and type, distribution option, etc., configuration. Then the address and loop variables i1..i6 are advanced, and address modifier chosen, according to following pseudo code:

```
  lpend1 = (i1 == (n1 - 1)) || (n1 == 0);
  lpend2 = (i2 == (n2 - 1)) || (n2 == 0);
  lpend3 = (i3 == (n3 - 1)) || (n3 == 0);
  lpend4 = (i4 == (n4 - 1)) || (n4 == 0);
  lpend5 = (i5 == (n5 - 1)) || (n5 == 0);
  lpend6 = (i6 == (n6 - 1)) || (n6 == 0);

if (lpend1 && lpend2 && lpend3 && lpend4 && lpend5 && lpend6) {
 amod = 0; // stay at last data point
} else if (lpend1 && lpend2 && lpend3 && lpend4 && lpend5) {
 i1 = i2 = i3 = i4 = i5 = 0;
 i6 = i6+1;
 amod = amod6;
} else if (lpend1 && lpend2 && lpend3 && lpend4) {
 i1 = i2 = i3 = i4 = 0;
 i5 = i5+1;
 amod = amod5;
} else if (lpend1 && lpend2 && lpend3) {
 i1 = i2 = i3 = 0;
 i4 = i4+1;
 amod = amod4;
} else if (lpend1 && lpend2) {
 i1 = i2 = 0;
 i3 = i3+1;
 amod = amod3;
} else if (lpend1) {
 i1 = 0;
 i2 = i2+1;
 amod = amod2;
} else {
 i1 = i1 + 1;
 amod = amod1;
}
```

If the agen functionality is implemented in scalar operations, it would take potentially many instructions.

Agen address calculation is post-modify. When executing an agen-based load/store operation, the lower 20-bit of Agen address field is used to address the load/store, amod is calculated as described above, address (unsigned 20-bit) is added with amod (signed 18-bit).

Consider the VMEM address map (see 5.3). In agen address update, it is NOT possible to jump from one superbank's primary region into another superbank's primary region, since the gap is 128KB, 2^17 bytes, thus minimal distance 2^17 + 1, while signed 18-bit of amod can encode a range of -2^17 ~ (2^17 – 1). It IS possible, however, for an agen

address to walk from one primary region to an aliased region, then onward into another superbank primary region. This is, however, strongly discouraged, as it may break software compatibility in the future.

See Circular Buffer Addressing for additional address calculation steps when circular buffer is configured.

With the reset default values of Addr = 0, amodi = 0, Ni = 1 and Ii = 0, uninitialized agen would have address fixed at 0 when it's used in agen-based load/store.

Additionally, Ni = 0 is treated like Ni = 1 with the way end-of-loop is detected, and maximal iteration count for any loop level is 65535.

Agen configuration also includes an optional lane_offset field for transposing load/store. For the basic T transposition mode, the lane_offset field provides a row offset scaled by the lane number. For lane i, relative to linear/consecutive access, the address offset is i * lane_offset * 64 Bytes.

See Transposing Load/Store for use of lane_offset in address calculation across various transposition modes.

6.4.2 Automatic Predication

When all loop variables reach their ending count, meaning the agen has executed the prescribed number of load/stores, all loop variables are stuck at the ending count. Any subsequent load with that Agen would repeat reading at the ending address. **Any subsequent store with that Agen will be predicated off**.

For example, for an Agen with N1 = 4, N2 = N3 = N4 = N5 = N6 = 1, its loop variable and predicate off status with respect to execution of the relevant load/store is as follows:

	I1	I2	I3	I4	I5	I6	auto_pred_off
Initial state	0	0	0	0	0	0	0
after 1 execution	1	0	0	0	0	0	0 (1st store allowed)
after 2 executions	2	0	0	0	0	0	0 (2nd store allowed)
after 3 executions	3	0	0	0	0	0	0 (3rd store allowed)
after 4 executions	3	0	0	0	0	0	1 (4th store allowed)
after 5 executions	3	0	0	0	0	0	1 (5th store blocked)

We can think of the auto_pred_off as an overflow bit of the Agen loop variables updated after the execution (like Agen loop variables), but its predication effect applies on the next memory store transaction.

This agen automatic predication works as an override of programmer-specified predication on vector or scalar store via predicate register or vector register. When auto_pred_off is 0, programmer-specified predication mechanism applies. When auto_pred_off is 1, entire memory write transaction is blocked.

The agen automatic predication does not affect loads. Any scalar or vector load using an Agen with exceeded iteration count (thus auto_pred_off = 1) will still have its memory transaction carried out and destination register write occurred, albeit with address stuck at the last valid address so memory read-back value should remain the same (except if/when there's another party, VPU, DLUT or DMA, writing to that address).

The use case for this feature is loop unrolling. Often VPU code uses pragma `chess_unroll_loop(K)` to indicate to compiler that the loop is to be unroll K times, for software pipelining.

```
for (i=0; i<niter; i++) chess_unroll_loop(K)
{
  // loop body
}
```

It is not required that iteration count (niter in the above example) be a multiple of K. Compiler generates code to check, and break up the loop into a "multiple" loop and a "remainder" loop to ensure that the generated code executes correctly.

If/when the programmer is certain that the iteration count is indeed a multiple of K, another pragma, `chess_unroll_loop_assuming_multiple(K)`, can be used. This pragma instructs compiler not to generate code to compute/check niter modulo K, and to not to generate the "remainder" loop.

The automatically predicate-off feature may allow `chess_unroll_loop_assuming_multiple(K)` to be used whether niter is a multiple of K, resulting in smaller code size and lower loop overhead.

```
quotient_ceil = (niter + K – 1) / K; // ceiling (niter / K)
for (i=0; i< quotient_ceil * K; i++) chess_unroll_loop_assuming_multiple(K)
{
  // loop body
}
```

This technique works for most common loops where outcomes are stored out in the loop, so extra iterations, as long as stores are predicated off, do not affect the outcome.

When there is accumulation over loop iterations using vector or scalar register, the Agen automatic predication feature does not quite work, as the predication applies only to stores, not to register writes. Also, the store must be driven by Agen, as there's no way to specify an ending iteration count using scalar-based (base + offset or post-modify) stores.

6.4.3 Rounding and Saturation

Agen-based store includes rounding and saturation features. Values from register file are first rounded, then saturated.

There are corresponding Agen configuration fields to convey the parameters:

- Rounding field includes 1-bit for round/truncation option and 7-bit for number of bits to round/truncate

- Saturation low/highs limit and values
- Saturation option field indicates whether saturation is enabled, and whether saturation limits are treated as signed or unsigned

When the number of bits to round/truncate exceeds source lane width (B=12, H=24, W=48), rounding leads to zero for all inputs, and truncation leads to zero for zero/positive inputs, and to -1 for negative inputs.

Rounding is performed by adding 1 to the bit position one bit lower than the bit count. For example, if we are rounding off 3 bits, we add (1 << 2) then right-shift by 3 bits. Truncation is performed by right-shift alone. Examples:

- round(6, 1) = (6 + (1 << 0)) >> 1 = 7 >> 1 = 3
- round(6, 2) = (6 + (1 << 1)) >> 2 = 8 >> 2 = 2
- round(-6, 3) = (-6 + (1 << 2)) >> 3 = -2 >> 3 = -1
- truncate(6, 1) = 6 >> 1 = 3
- truncate(6, 2) = 6 >> 2 = 1
- truncate(-6, 3) = -6 >> 3 = -1

For saturation, we support 4-parameter saturation. When enabled, hardware carries out the following:

```
store_val = (reg_val < SatLimLo) ? SatValLo : ((reg_val > SatLimHi) ? SatValHi : reg_val);
```

In this case, reg_val is 12/24/48-bit signed. SatLimLo and SatLimHi are sign/zero-extended from 32-bit values in Agen configuration. We have a 2-bit saturation option SatOpt to indicate whether to sign or zero extend the 32-bit configuration values. Note that vector lane values are always read as signed.

Rounding and saturation steps are performed with bit width accommodating both the data source bit width (12/24/48-bit lane width in vector registers) and comparison values (signed/unsigned 32-bit). Consequently,

- For promoting stores (12-bit into 16-bit, 24-bit into 32-bit), the source data values are in representation range of signed 12-bit or 24-bit.
- In the case of comparison values, SatLimLo and/or SatLimHi, exceed the source data representation range, reg_val < SatLimLo would never happen, so the source data is unchanged for that saturation bound. For example, SatLimLo (signed) = -0x8000_0000 when source data is 12-bit, having representation range [-0x800, 0x7FF], reg_val < SatLimLo is always false.

  For extended word type source data (48-bit), and when saturation is enabled, the comparison is carried out correctly as if it's carried out in signed 48-bit. For example, when reg_val = -0x8000_0000_0000 (min value in signed 48-bit) and SatLimLo (signed) = -0x8000_0000 (min value in signed 32-bit), reg_val < SatLimLo is true and the replacement occurs.

The saturation replacement values SatValLo and SatValHi are configured as 32-bit numbers. When the memory store type is 8-bit or 16-bit, and the replacement occurs, only the 8 or 16 LSBs of SatValLo or SatValHi are written out to memory; the upper 24 or 16 bits are ignored.

Rounding and saturation features are not available for these cases:

- WX type: can be 8 x 48-bit, 16 x 24-bit, or 32 x 12-bit
- W type on single vector XARF: 16 x 32-bit

Double-vector XARF store does include rounding and saturation.

Rounding and saturation operations are performed as integer operations, so if enabled on floating-point (FP32 or FP16) type store, they would interpret floating-point binary values as 48-bit/24-bit integer values, so the resulting values being stored may not make sense.

6.4.4 Min and Max Value Collection

There is a min and max value collection feature in agen-based scalar/vector stores. Min/max collection occurs after rounding/saturation and is predicated upon the lane being stored to the memory.

There is a 2-bit min/max option to encode

- 0: disable (default)
- 1: disable
- 2: enable for signed min/max
- 3: enable for unsigned min/max

This includes a 32-bit MinVal (min value) and a 32-bit MaxVal (max value) in the Agen register file.

Upon agen initialization, min/max option is initialized to 0 (disabled), and min/max values are initialized to 0.

Upon configuring the min/max option to 2 (enabled for signed min/max), the min value is initialized to MAX_INT32 = 0x7FFF_FFFF. The max value is initialized to MIN_INT32 = 0x8000_0000.

Upon configuring the min/max option to 3 (enabled for unsigned min/max), the min value is initialized to MAX_UINT32 = 0xFFFF_FFFF. The max value is initialized to MIN_UINT32 = 0.

Upon configuring the min/max option to 0 or 1 (disabled), the min/max values are reset to 0.

The min/max option is in the first 512-bit part of the Agen config, so is saved with AgenCfgST, and restored with AgenCfgLD. Upon AgenCfgLD, min/max values are initialized according to min/max option.

The min/max values are in the second part of the Agen config, so is saved with AgenCfgST_p2 and restored with AgenCfgLD_p2.

Note that with AgenCfgLD_p2, min/max values are loaded as-is from memory without checking to see if they make sense:

1. Min value can be larger than Max value according to the signed/unsigned option designated in MinMaxOpt.

2. Min value and/or Max value may fall outside the valid range of signed/unsigned option designated in MinMaxOpt and data type previously used agen-based store associated with the specific Agen.
3. Max/Max values can be non-zero, though MinMaxOpt indicates min/max collection is disabled.

(1) and (2) are because:

- The initialized values for Min/Max values are type-blind, and in fact fall out of valid char and short ranges in 3 of the 4 possible values (INT32MAX, INT32MIN, UINT32MAX).
- Agen data structure is type-neutral and does not record type of data being stored.

Because we cannot guarantee that min/max values make sense when min/max collection feature is enabled, we don't attempt to correct min/max values when the feature is disabled, presumably min/max values are not useful to the application in such cases.

Upon every Agen-based store (scalar or vector), if min/max feature is enabled, signed or unsigned min and max operations are carried out, so that the MinVal and MaxVal fields maintain the min and max values across all stored data. They can be read out after processing to query min and max values.

The min/max collection excludes WX type stores, and that if enabled on floating-point (FP32 or FP16) type store, would interpret floating-point binary values as 32-bit/24-bit integer values, so the resulting min/max values may not make sense. This is with rounding/saturation steps before min/max collection being disabled. If either rounding or saturation is enabled, input to min/max collection may not make sense.

6.4.5 Save and Restore to/from Memory

Once individual parameters in an agen are configured, the collection of all parameters can be saved to memory via AgenCfgST and restored back via AgenCfgLD. This allows calculation of parameters to be carried out during application initialization and be quickly restored to configure the agens during regular tile processing.

Reserved fields are written as zeros initialized to zero in InitAgen. They are not modifiable via any CfgAgen instructions and not utilized in any Agen functionality. Through CfgAgenLD, if corresponding contents in memory are non-zero, zero will be loaded into Agen data structure instead. When CfgAgenST is used to store out the whole Agen data structure, corresponding bits in memory will show zeros.

Consult Instruction Execution Ordering for various execution order exceptions regarding various instructions accessing Agen.

6.4.6 Circular Buffer Addressing

PVA supports circular buffer addressing to facilitate data reuse. Circular buffer addressing is available in agen-based load/store instructions by configuring optional

circular buffer starting address (cbuf_sa) and circular buffer size (cbuf_sz) parameters in the unit. Circular buffer is enabled when cbuf_sz is configured to be a nonzero value.

There's alignment constraint (consistent between DMA and VPU) that circular buffer should be 64-byte aligned. We allocate 16-bit for the starting address and the size parameters. We apply 6-bit up-shift before interpreting the parameters as a byte addressed to enforce the alignment.

Address is folded into the circular buffer via the following pseudo-code:

```
      CB_start = cbuf_sa << 6;
      CB_size = cbuf_sz << 6;
address = CB_start + ((address – CB_start) % CB_size);
// % = modulo operator, returns 0..CB_size-1
```

The circular buffer address calculation above is applied whenever agen-based load/store updates its address when each instance of such instruction is executed. The sequence of operations is as follows:

1. Prescribed load/store using the current address.
2. Address update using address modifiers, loop iteration count, and loop variables (see Multi-Dimensional Address Calculation).
3. When circular buffer is enabled, address is folded back to [CB_start, CB_start+CB_size-1] if it falls out of the range.

With circular buffer enabled (size > 0), address parameters should be constrained as follows:

- Base address and circular buffer should be inside a superbank.
- Base address should be within the buffer, i.e., CB_start <= base_addr < CB_start + CB_size.

Any address modifier must not have magnitude (absolute value) larger than the circular buffer size; i.e., |AMOD[i]| <= CB_size.

When circular buffer is enabled, every AGEN address update would be checked to see if it falls out of the circular buffer. If it falls under (addr < CB_start), it is adjusted with + CB_size. If it falls over (addr >= CB_start + CB_size), it is adjusted with – CB_size. If afterward it still falls out of the circular buffer, no error is reported. Note that when Agen parameters are properly constrained, this should not happen.

Details of circular buffer address calculation are as follows (this information is intended for verification, where parameters outside normal programming constraints may be used):

- Lower 20 bits of the AGEN address field is read as an unsigned number, **addr**
- **addr** is added with **amod**, lower 18 bits of one of the 6 address modifiers selected for this address increment. The addition outcome is kept as a signed 21-bit number, **addr1**, as the normally updated address without circular buffering
- Lower 14 bits of cbuf_start (Agen field) is left-shifted 6 bits to become CB_start (20-bit)

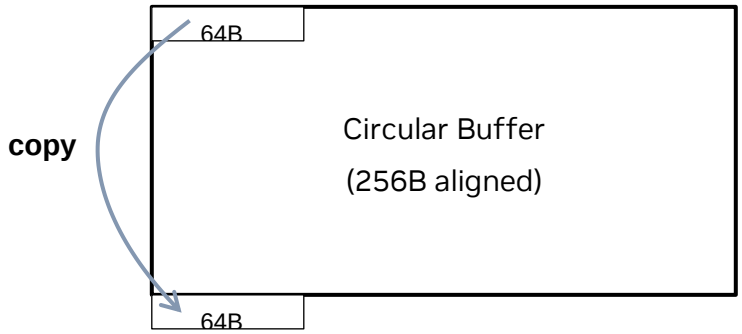
- Lower 14 bits of cbuf_size (Agen field) is left-shifted 6 bits to become CB_size (20-bit)
- **addr2** = **addr1** + CB_size, which is addr1 wrapped forward, kept as signed 21-bit number
- **addr3** = **addr1** – CB_size, which is addr1 wrapped backward, kept as signed 21-bit number
- If **amod** is negative:
  - If **addr1** is less than CB_start, meaning the negative address update makes it fall before circular buffer's start address, **wrapped_addr** is assigned **addr2** (addr1 wrapped forward)
  - Otherwise, **wrapped_addr** is assigned **addr1**
- Otherwise:
  - If **addr1** is greater than or equal to CB_size + CB_start, meaning the positive address update makes it fall after circular buffer's end address, **wrapped_addr** is assigned **addr3** (addr1 wrapped backward)
  - Otherwise, **wrapped_addr** is assigned **addr1**
- Lower 20 bits of **wrapped_addr** is read as an unsigned number and *written back zero-extended* to the 32-bit AGEN address field.

Circular buffer addressing is **NOT** applied **inside a single memory transaction** of single/double vector load/store. Thus, either vector load/store should avoid crossing the circular buffer boundary, or there should be software workaround.

One software workaround scheme where DMA supplies data to the circular buffer, and VPU consumes the data, is to allocate additional 64 bytes after the circular buffer as work-around areas. Before VPU starts consuming data in the circular buffer, the first 64 bytes of circular buffer data should be copied to fill the 64 bytes work-around area. This work-around only covers linear (consecutive) accesses though, not transposing load/store, table lookup, or histogram.

There is no easy workaround when VPU supplies data into the circular buffer, and DMA consumes it. Misaligned data access generally comes from spatial dependency and is only in reading data. It is usually feasible to size output block dimension so that data writes are compliant with reasonable alignment constraints. Thus, there is usually no need for such a workaround.

Figure 12. Workaround for vector accesses across circular buffer boundary



Superbanks are not consecutive in the data memory space (128KB in 256KB space). In normal application, circular buffer should not go out of any superbank. When it does, the address is wrapped around and mapped back to one of the VMEM superbanks without any error interrupt being raised.

Chapter 7. Decoupled Lookup Unit (DLUT)

In this chapter, an overview of the Decoupled Lookup Unit (DLUT) is provided. For a programming example, refer to Leveraging DLUT.

7.1 Overview

For Orin VPU, we extended VPU instruction set functionality in various areas within the scope of an embedded vector SIMD machine. There is one area that we cannot extend in this scope, that is resolving memory bank conflict in parallel lookup operations.

In the VPU instruction set, we do have various parallel lookup instructions (2/4/8/16/32-way parallelism), but these instructions require that we have correspondingly that many tables so that there is inherently no memory bank conflict. These tables are sometimes a replication of one table, and sometimes different tables, depending on the application.

For example, in image warping we transform one image tile at a time, and parallel lookup is only possible if we replicate it from that one image tile. For example, in the feature tracker, we perform gradient descent on many patches of an image, and parallel lookup can be performed on the many patches in parallel, if the patches are reformatted into parallel table organization. Either way, table lookup parallelism is constrained by memory footprint taken up by the parallel tables.

In applications involving table lookup, we often wish to perform parallel lookup with certain throughput, while we cannot afford memory footprint to replicate one table that many times, or load that many parallel tables into memory. Ideally, we want the processor to allow parallel lookup with just one copy of the table as part of the instruction set. However, such memory operations would result in data-dependent memory bank conflicts in execution. For example, 32-way parallel halfword lookup with one copy of the take may take up to 32 cycles just to carry out reading the table entries, if all 32 lanes happen to go to the same memory bank. VPU can handle some degree of data-dependent memory conflict, naming at superbank level. Handling memory bank level conflicts is simply too difficult to accomplish in an embedded processor pipeline with limited pipeline depth.

The decoupled lookup unit (DLUT) is architected to provide this functionality outside the processor pipeline and can operate concurrently and independently with the processor

pipeline, thus the term “decoupled unit”. The DLUT carries out parallel lookup with one common table by executing as many lookups in a cycle as it can in a decoupled pipeline.

Besides parallel lookup with one common table, DLUT also supports one configuration of contention free lookup/interpolation, which is most helpful in accelerating target workload in the Orin SOC plan. Although the functionality is supported in the VPU processor, but by adding this to DLUT, we offload VPU processing cycles so there are advantages in performance and power.

DLUT also supports table reformatting needed to bridge between DMA and DLUT or VPU lookup operations. Again, the table reformatting can be accomplished at the same throughput by the VPU processor, but by adding this to DLUT, we offload VPU processing cycles so there is advantages in performance and power, and the functionality in DLUT leverages datapath we need to have anyway for the main lookup functionality, so does not pose much area or power increase, just minor engineering effort.

7.2 DLUT Features

The DLUT provides these operation modes:

- 1D lookup
- 2D lookup
- 1D lookup and linear interpolation
- 2D lookup and bilinear interpolation
- Table reformatting
- Conflict free 2D lookup and bilinear interpolation (from parallel copies of table)
- 2D lookup and bilinear interpolation with auto-indexing, where the index data need not be supplied; indices are generated by DLUT from a few parameters

Other DLUT features:

- 1D/2D lookup from one common table, with conflict detection/resolution
- Optional integer only or fixed-point integer + fraction indices, via configurable number of fractional bits
- Out-of-range sentinel return value
- Out-of-range predicate off output write
- Configurable X/Y offset to translate between global coordinates and local coordinates
- Indices can be unsigned 16-bit, or 32-bit (each X or Y in case of 2D lookup)
- Table entries (and output) can be 8-bit, 16-bit, or 32-bit, signed or unsigned, and entry data type is independent of index data type

7.3 Task Structure and Operation Modes

We define a DLUT task as producing N2 * N1 outputs through lookup and optional post-lookup interpolation. A trimmed down agen (address generator) drives addressing of index read, and another agen drives output write. The table pointer can step linearly in the outer dimension of N2, so one task can be regarded as N2 rounds of lookup, with one table producing N1 outputs per round. These N2 rounds of lookup of one task share the same parameter block that specifies index/output data type, index read agen, output write agen, and so on.

Besides table lookup and post-lookup interpolation, DLUT also supports conflict-free 2D lookup with bilinear interpolation, and various table reformatting as separate tasks.

DLUT supports the following operation modes:

- 1D lookup: from linear indices, optionally perform rounding or truncation to convert to integer indices and 1D table lookup.
- 2D lookup: from 2D indices, optionally perform rounding or truncation to convert to integer indices and 2D table lookup
- 1D lookup with linear interpolation
- 2D lookup with bilinear interpolation
- 2D conflict-free lookup with bilinear interpolation, 32-bit index and 16-bit entry only
- Table reformatting
- 2D lookup and bilinear interpolation, with automatic index generation that supports starting X/Y and scaling step per round of lookup

DLUT in operation utilizes 3 memory streams, index read stream, lookup read stream, and output write stream. To simplify hardware design/verification, encourage efficient operation, and simplify DLUT/VPU/DMA interaction, each stream is tied to the superbank each task is configured with. Thus, address modification due to agen update and/or table address offset is performed in bits 17:0 of the respective address pointers, leaving bits 19:18 that identifies the superbank unchanged from the task-configured addresses.

Tor better DLUT performance, index and lookup should not be in the same superbank. However, such an allocation does not affect the correctness of the outcome.

We define DLUT group size being the number of outputs per clock the hardware can achieve ideally, when there is no conflict. The group size is basically set by either index read throughput or lookup throughput, as output write throughput is never lower than lookup throughput.

Group size for various modes is as follows:

- 1D/2D lookup (without interpolation): 32 for Byte/Halfword entries, 16 for Word entries.
- 1D lookup with interpolation: 16 for Byte/Halfword entries, 8 for Word entries.
- 2D lookup with interpolation: 8 for Byte/Halfword entries, 4 for Word entries.
- 2D conflict-free lookup with interpolation: 8 (since only Halfword entry type is supported).

> Table reformatting: 32 (since Halfword type is assumed).

Note that it is NOT required that the inner-loop output size, N1, should be a multiple of group size. Hardware handles optional partial-group operation in the last inner iteration by invalidating various index read, lookup, and output write lanes not being utilized. Note that even when task length N1 is a multiple of group size, we can still have partial transactions in index read and/or output write.

7.4 Task Sequencing and VPU/DLUT Interaction

DLUT execution time is dependent on bank conflict within the indices, so it is not constant. While it is possible to establish the average execution time given random number distribution of the indices, the actual execution time can be drastically different. For example, a task of 32 Halfword lookups can take between 1 and 32 cycles to execute, excluding any control and pipelining overhead.

In applications there can be multiple dependent or independent lookup tasks that we would like DLUT to execute sequentially, while VPU is executing some other compute tasks. Since DLUT execution time is data dependent and can be drastically different, it is not convenient for VPU to “check on” DLUT between compute tasks and kick off the next DLUT task one at a time. To facilitate parallel execution, we architect the DLUT interface to facilitate task sequencing.

VPU software prepares task parameters, allocate input/output regions, for multiple tasks at a time, and go through one interaction with DLUT. Parameters for each task is a fixed-sized data structure that links to the next task.

DLUT carries out the configured tasks sequentially without overlap. Each task is processed to completion (last output written) before the next task is started (first index read) to simplify hardware implementation.

Chapter 8. Programming Examples

In this chapter, we show a few relatively simple programming examples. The profiling instruction reports were generated at the time of the writing, and may not be accurate later, as the performance is subject to processor model revisions and ASIP tool updates.

8.1 Typical Test Case Organization

A recommended way to organize test source files for a typical algorithm/application VPU standalone test case, for example, array_add, is to have these source files:

- array_add.prx — project file listing source files, header include paths, compiler settings, etc.
- array_add_test.c: — containing main function and global input/output arrays
- array_add_ref.c — reference function, typically written in plain/scalar C code
- array_add_ref.h — reference function header
- array_add_opt.c — optimized function
- array_add_opt.h — optimized function header

The VPU standalone test case typically used to develop/optimize compute kernels. For developing a PVA application including DMA, one should follow the cuPVA development flow.

A sample testbench code in array_add_test.c follows:

```
#include "stdio.h"
#include "string.h"

#define TEST_SZ 4096

int chess_storage(RAM_Ab:chess_segment(A)) in1[TEST_SZ];
int chess_storage(RAM_Bb:chess_segment(B)) in2[TEST_SZ];
int chess_storage(RAM_Cb:chess_segment(C)) out_ref[TEST_SZ];
int chess_storage(RAM_Cb:chess_segment(C)) out_opt[TEST_SZ];

int main()
{
  test_mem_fill_int(in1, TEST_SZ, 0x80000000, 0x7FFFFFFF);
  test_mem_fill_int(in2, TEST_SZ, 0x80000000, 0x7FFFFFFF);
  memset(out_opt, 0, sizeof(out_opt));
```

```
  array_add_ref(in1, in2, out_ref, TEST_SZ);
  array_add_opt(in1, in2, out_opt, TEST_SZ);

  int fail = memcmp(out_ref, out_opt, sizeof(out_ref));
  return fail;
}
```

The arrays are allocated with chess_storage() pragma. In a VPU programming environment, the VMEM L1 data memory, which consists of 3 superbanks each 128KB, is a precious resource, so typically programmers would allocate manually into the 3 superbanks in a matter that minimize bank conflict during VPU compute kernel execution. See Memory Allocation among VMEM Superbanks about details in VMEM superbanks and storage specifiers.

For this specific compute kernel, array addition, we need 2 inputs being in different superbanks. The output array must be in a third superbank in Gen-1 VPU, since in Gen-1 VMEM, each superbank has one memory port that can support read or write, but not both. In Gen-2 VPU, the output array can be in any superbank since each VMEM superbank has one read port and one write port.

DMA and DLUT share VMEM superbanks as well, so can potentially conflict with VPU compute kernel accessing VMEM superbanks. In Gen-2 VPU, one can take advantage of the one-read-one-write ports of VMEM superbank to reduce conflicts. This is because typically we have the producer/consumer relationship between each pair of masters transmitting one array of data.

Typically, in the main program, input arrays are initialized with random values, and optimized outcome array is initialized to zero. Then, the reference function is called to produce expected outcome array, the optimized function is called to produce optimized outcome. Finally, the two arrays are matched to verify that optimized function carries out the intended functionality, and because of the matching, the convention being zero indicates pass, non-zero indicates fail, is returns from main().

8.2 1D Array Addition

We shall use a one-dimension array addition function to illustrate the process of taking some plain C code, and revise it step by step to achieve full performance.

8.2.1 Scalar Code

We start with the same code as the one shown in Hardware Looping to showcase the hardware looping feature. We often call this the scalar code, as the code is written without using vector data type, vector operation intrinsic functions, or vector load/store intrinsic functions. The code is translated into scalar math and scalar load/store instructions.

```
//*********************************************************************
// Function implemented with normal/scalar C code
//*********************************************************************
void array_add_ref(int * A, int * B, int * C, int len)
{
  for (int i=0 ; i<len; i++)
  {
    C[i] = A[i] + B[i];
  }
}
```

Instead of assembly listing, the profiling instruction report is shown next. The report has performance information annotated besides the assembly listing, so it is a lot more convenient to assess performance with, than assembly listing. There is also function-level PC range, code size, and cycle/instruction count information that are quite useful.

Some manual editing is done on the generated report to shorten labels and various fields so that the report can easily fit the page width for readability. Somehow tool generated instruction reports omit labels, and they are manually added back to make better sense of the control flow.

```
Function detail: array_add_ref void_array_add_ref___P__sint___P__sint___P__sint___sint

  Low PC        : 56
  High PC        : 71
  Size in program memory: 16
  Cycle-count      : 14352 (15.31%)
  Instruction-count   : 6154 ( 7.63%)
  Instruction Coverage : 100.00%

 PC   Assembly                                                                Exe-cnt  Cycs
---- ------------------------------------------------------------------------ ------- -------
 56 CMPLEI R7,#0,R2                                                             1       5
 57 BNEZ R2,#TGT_Fvoid_array_add_ref_12                                         1       1
 58 NOP                                                                         1       1
 59 NOP                                                                         1       1
 60 RPT R7,#LE_Fvoid_array_add_ref_11                                           1       1
 61 ORI R0,#4,R2                                                                1       1
 62 NOP || NOP                                                                  1       1
 64 LDW *R4+=R2,R8 || LDW *R5+=R2,R3                                         2048   10240
 66 ADD R3,R8,R9                                                             2048    2048
.label LE_Fvoid_array_add_ref_11
 67 STW R9,*R6+=R2                                                           2048    2048
.label TGT_Fvoid_array_add_ref_12
 68 JR R15                                                                      1       1
 69 NOP                                                                         1       1
 70 NOP || NOP                                                                  1       3
```

The number in the ‘Exe-Cnt’ column is execution count, or how many times that specific packet was executed, and number in the ‘Cycles’ column is the cycle count. Where the two numbers differ, usually the cycle count is an integer multiple of the execution count, with the ratio being the number of cycles each instance of the packet takes to execute, usually due to stalling in the execution.

By looking at either number, one can quickly tell the loop body from the rest of the code, as the loop body is iterated many times. In this example, the loop is iterated 2048 times, so each execution packet in the loop body is executed 2048 times.

The loop body consists of 4 instructions in 3 execution packets, performing, respectively, 2 loads, 1 operation, and 1 store, exactly as implied in the source code. In general, plain C code compiles cleanly into scalar instructions.

The first packet of the loop body is taking 10240 cycles to execute 2048 times, so 5 cycles each time. Note that the stalled execution packet is executing 2 parallel loads, and the very next packet is adding up the 2 destination registers of the loads. The stalling is due to the load-to-use latency of 5 cycles.

Also, the code has conditional branches, BNEZ, although the branch is not taken (otherwise the loop is completely bypassed and would get zero execution and cycle counts). The conditional branch is there in the assembly to guard against the case when the len (length) argument is zero, to truly implement the correct behavior of the C-language for loop.

Performance from this plain C code is quite poor, taking 5+1+1 = 7 cycles per iteration, with exactly one addition operation achieved per iteration. The whole function execution takes 14,352 cycles. In subsequent sections we will show how performance can be drastically improved.

8.2.2 Optimization 1: Vectorized Code

We make our first optimization revision by replacing scalar processing with vector processing, as shown in the following code:

```
//*********************************************************************
// Optimization 1: vectorization
//*********************************************************************
void array_add_opt1(int * A, int * B, int * C, int len)
{
  int  vecw = chess_elementsof(dvintx);
  dvint * vptrA = (dvint *) A;
  dvint * vptrB = (dvint *) B;
  dvint * vptrC = (dvint *) C;
  dvintx vA, vB, vC;

  for (int i=0 ; i<len/vecw; i++)
  {
    vA = sign_extend(*vptrA++);
    vB = sign_extend(*vptrB++);
    vC = vA + vB;
```

```
    *vptrC++ = extract(vC);
  }
}
```

We use a pragma chess_elementsof(dvintx) to acquire the vector width, as the number of elements in the dvintx type. Since source/destination arrays are of int type, we would use dvint as the vector data type in memory, and dvintx as the vector data type in register file. The two data types have the same number of elements, so it's just as valid to code vecw = chess_elementsof(dvint).

We cast each source and destination array points to dvint pointers, and declare vector variables vA, vB, vC, of dvintx type.

In the loop body, we perform signed vector loads via sign_extend() intrinsic function with vector pointer dereferencing with post-increment. Sign_extend is thus named to indicate that we are sign-extending from standard int (32-bit) type into extended word (48-bit) type for each element of the array.

We load the two source operands vA and vB, we add them up into vC, and we store out vC. The store is coded as vector pointer dereferencing and the extract() intrinsic function. Exact is thus names to indicate that we are extracting part of the extended word (48-bit) in each vector lane into a standard int type (32-bit) before storing into memory.

The generated (and cosmetically, manually edited) profiling instruction report that shows compiled assembly with execution count and cycle count information is as follows:

```
Function detail: array_add_opt1 void_array_add_opt1___P__sint___P__sint___P__sint___sint

  Low PC        : 72
  High PC        : 95
  Size in program memory: 24
  Cycle-count      : 1044 ( 1.11%)
  Instruction-count   : 398 ( 0.49%)
  Instruction Coverage : 100.00%

 PC   Assembly                                                                Exe-cnt     Cycs
---- ------------------------------------------------------------------------ ----------- -------
 72 SRAI R7,#31,R2                                                                 1        1
 73 ANDI R2,#15,R2                                                                 1        1
 74 ADD R2,R7,R7                                                                   1        1
 75 SRAI R7,#4,R2                                                                  1        1
 76 CMPLEI R2,#0,R7                                                                1        5
 77 BNEZ R7,#TGT_Fvoid_array_add_opt1_15                                           1        1
 78 NOP                                                                            1        1
 79 NOP                                                                            1        1
 80 RPT R2,#LE_Fvoid_array_add_opt1_14                                             1        1
 81 ORI R0,#64,R7                                                                  1        1
 82 NOP                                                                            1        1
 83 DVLDW_P *R4+=R7,V2:V3|| DVLDW_P *R5+=R7,V0:V1                                128      768
 85 VAddW V2:V3,V0:V1,V4:V5                                                      128      128
```

```
.label LE_Fvoid_array_add_opt1_14
 86 DVSTW_P V4:V5,*R6+=R7                                                    128      128
.label TGT_Fvoid_array_add_opt1_15
 87 JR R15                                                                    1        1
 88 NOP                                                                       1        1
 89 NOP || NOP || NOP || NOP || NOP || NOP || NOP                             1        3
```

The vectorized function takes 1044 cycles to execute and is about 13.7x the performance of the scalar code. Essentially, we gain a speedup of 16x by processing a dvint, 16 elements of 32-bit, per iteration, but the loop executes 6+1+1 = 8 cycles per iteration, versus 7 cycles per iteration in the scalar loop, so we give back some of the speedup from vectorization.

The compiled assembly is still relatively clean, and the loop body still has 4 instructions in 3 execution packets. The 4 instructions are respectively 2 vector loads, one vector addition, and one vector store. Here the de-reference of pointer with post-increment in the C code maps perfectly to the vector load/store instructions.

The higher stall count in the first execution packet of the loop body, 6 cycles in the vectorized loop, versus 5 cycles in the scalar loop, is due to processor pipelining. Vector addition happens to have its source operands forwarded from the load unit one cycle later than scalar addition can forward its source operands, so load-to-use latency for vector operations is one cycle longer.

8.2.3 Optimization 2: Unroll and Pipeline the Loop

Next, we tackle the inefficiency caused by load-to-use latency, as shown in the following optimized code:

```
//*********************************************************************
// Optimization 2: pipelining & unrolling
//*********************************************************************
void array_add_opt2(int * A, int * B, int * restrict C, int len)
{
  int  vecw = chess_elementsof(dvintx);
  dvint * vptrA = (dvint *) A;
  dvint * vptrB = (dvint *) B;
  dvint * restrict vptrC = (dvint *) C;
  dvintx vA, vB, vC;

  for (int i=0 ; i<len/vecw; i++) chess_unroll_loop(8)
            chess_prepare_for_pipelining chess_loop_range(16,)
  {
    vA = sign_extend(*vptrA++);
    vB = sign_extend(*vptrB++);
    vC = vA + vB;
    *vptrC++ = extract(vC);
  }
}
```

We cannot significantly reduce the latency. What we can do is to fill the pipeline with useful work while the latency is played out. Technique to do that is called software pipelining, and is enabled by the 3 pragma annotated on the for statement:

- **chess_unroll_loop(8)** tells the compiler to replicate the loop body 8 times and adjust the loop iteration count accordingly, by dividing it by 8.
- **chess_prepare_for_pipelining** tells the compiler to software pipeline this loop, causing the loop body code (which could be the original loop contents or already replicated through loop unrolling) to be folded and scheduled into multiple iterations, and consequently there will be a prolog of the loop and an epilog of the loop.

  Often, chess_unroll_loop() and chess_prepare_for_pipelining pragmas go hand-in-hand. Most loops would need both pragmas to achieve the best performance.
- **chess_loop_range(16,)** tells the compiler that this loop is guaranteed (by the programmer) to run at least 16 iterations. This pragma causes generated code to do without the “what if len is zero” checking and conditional branch, resulting in a more streamlined control flow in the compiled assembly.

One other thing to point out is the keyword **restrict** on address pointers C and vC that we use to write back to memory. This **restrict** keyword is telling the compiler that it is safe to perform these writes in any order relative to other memory reads and/or writes. Without the restrict keyword, compiler cannot overlap multiple instances of the original load/store operations to software-pipeline the loop effectively.

The corresponding profiling instruction report is shown next.

```
Function detail: array_add_opt2 void_array_add_opt2___P__sint___P__sint___P__sint___sint

  Low PC        : 96
  High PC        : 167
  Size in program memory: 72
  Cycle-count      : 141 ( 0.15%)
  Instruction-count   : 139 ( 0.17%)
  Instruction Coverage : 100.00%

 PC   Assembly                                                                  Exe-cnt Cycs
---- ---------- ------------------------------------------------------------------------ ------- -----
 96 SRAI R7,#31,R2                                                                    1     1
 97 ORI R0,#64,R2 || ANDI R2,#15,R3                                                   1     1
 99 ADD R3,R7,R7                                                                      1     1
 100 SRAI R7,#7,R7 || DVLDW_P *R4+=R2,V30:V31|| DVLDW_P *R5+=R2,V26:V27               1     1
 103 ADDI R7,#-1,R7 || DVLDW_P *R4+=R2,V22:V23|| DVLDW_P *R5+=R2,V18:V19              1     1
 106 DVLDW_P *R4+=R2,V14:V15|| DVLDW_P *R5+=R2,V10:V11                                1     1
 108 DVLDW_P *R4+=R2,V6:V7|| DVLDW_P *R5+=R2,V2:V3                                    1     1
 110 DVLDW_P *R4+=R2,V4:V5|| DVLDW_P *R5+=R2,V0:V1                                    1     1
 112 DVLDW_P *R4+=R2,V12:V13|| DVLDW_P *R5+=R2,V8:V9                                  1     1
 114 RPT R7,#LE_Fvoid_array_add_opt2_54                                               1     1
 115 VAddW V30:V31,V26:V27,V24:V25|| DVLDW_P *R4+=R2,V20:V21|| DVLDW_P *R5+=R2,V16:V17    1     1
 118 VAddW V22:V23,V18:V19,V18:V19|| DVLDW_P *R4+=R2,V28:V29|| DVSTW_P V24:V25,*R6+=R2||
     DVLDW_P *R5+=R2,V24:V25                                                          1     1
 122 VAddW V14:V15,V10:V11,V10:V11|| DVLDW_P *R4+=R2,V30:V31|| DVLDW_P *R5+=R2,V26:V27||
```

```
    DVSTW_P V18:V19,*R6+=R2                                                                  15   15
126 VAddW V6:V7,V2:V3,V2:V3|| DVLDW_P *R4+=R2,V22:V23|| DVLDW_P *R5+=R2,V18:V19||
    DVSTW_P V10:V11,*R6+=R2                                                                  15   15
130 VAddW V4:V5,V0:V1,V0:V1|| DVLDW_P *R4+=R2,V14:V15|| DVLDW_P *R5+=R2,V10:V11||
    DVSTW_P V2:V3,*R6+=R2                                                                    15   15
134 VAddW V12:V13,V8:V9,V8:V9|| DVLDW_P *R4+=R2,V6:V7|| DVLDW_P *R5+=R2,V2:V3||
    DVSTW_P V0:V1,*R6+=R2                                                                    15   15
138 VAddW V20:V21,V16:V17,V16:V17|| DVLDW_P *R4+=R2,V4:V5|| DVLDW_P *R5+=R2,V0:V1||
    DVSTW_P V8:V9,*R6+=R2                                                                    15   15
142 VAddW V28:V29,V24:V25,V28:V29|| DVLDW_P *R4+=R2,V12:V13|| DVLDW_P *R5+=R2,V8:V9||
    DVSTW_P V16:V17,*R6+=R2                                                                  15   15
146 VAddW V30:V31,V26:V27,V24:V25|| DVLDW_P *R4+=R2,V20:V21|| DVLDW_P *R5+=R2,V16:V17||
    DVSTW_P V28:V29,*R6+=R2                                                                  15   15
150 VAddW V22:V23,V18:V19,V18:V19|| DVLDW_P *R4+=R2,V28:V29|| DVSTW_P V24:V25,*R6+=R2||
    DVLDW_P *R5+=R2,V24:V25                                                                  15   15
154 VAddW V14:V15,V10:V11,V10:V11|| DVSTW_P V18:V19,*R6+=R2                                   1    1
156 VAddW V6:V7,V2:V3,V2:V3|| DVSTW_P V10:V11,*R6+=R2                                        1    1
158 VAddW V4:V5,V0:V1,V0:V1|| DVSTW_P V2:V3,*R6+=R2                                          1    1
160 VAddW V12:V13,V8:V9,V8:V9|| DVSTW_P V0:V1,*R6+=R2                                        1    1
162 JR R15 || VAddW V20:V21,V16:V17,V16:V17|| DVSTW_P V8:V9,*R6+=R2                           1    1
165 VAddW V28:V29,V24:V25,V28:V29|| DVSTW_P V16:V17,*R6+=R2                                  1    1
167 DVSTW_P V28:V29,*R6+=R2                                                                  1    3
```

This optimized function takes just 141 cycles to execute and achieves 7.4 times the performance of the previous code, which is vectorized but not yet software pipelined. If we compare it to the original plain C code, the speedup is 101.8 times.

This loop has a theoretical max throughput of one dvint vector addition, 16 lanes x 32-bit, per clock cycle. It's bounded by each dvint vector operation needing 2 loads and 1 store for input/output, saturating the 3 superbanks x 512-bit of VMEM bandwidth. Vector math throughout for addition is one dvintx addition per vector slot, so in this loop, vector math is only 50% utilized. Each execution packet in the loop body is packed with one VAddW (double vector addition), 2 DVLDW (double vector load word type), and one DVSTW (double vector store word type), confirming the math and memory utilization.

In terms of efficiency, 128/141 = 91%. Overhead comes from 13 cycles spent setting up the local frame on the stack, extracting arguments from the stack, setting up the loop, and finally for 2 cycles of pipeline bubble from executing a return instruction (JR R15) to the caller.

In reference to the loop unrolling factor: performance-wise, it's not necessary to unroll 8 times. It is convenient to constrain a compute function to limit loop iteration count to a power of 2, thus unrolling by 2, 4, 8, is more convenient than unrolling by 5, 6, 7, etc. The minimal number of times to unroll a loop depends on how much vacancy there is in a single iteration due to load to use latency and sometimes also vector math operation latency. With compiler and ISS (instruction set simulator), one can just experiment with different unrolling factors and find a factor that works.

For a simple, single-operation loop like in the array addition example, we need to unroll 6 times to achieve optimal performance. If unrolling by K times achieves the optimal

performance, unrolling more than K times should achieve the same performance, but would cause the compiled code size to grow. VPU Instruction Cache has a set capacity, 16K Bytes for the Orin generation, so we should not unnecessarily increase the code size.

8.3 2D Array Addition

Next, we shall use a two-dimension array addition function to illustrate how we leverage the multi-dimensional address calculation feature of agens to collapse nested for loops to minimize looping overhead and achieve optimal performance.

8.3.1 Scalar Code

The following code implements a two-dimension array addition.

```
//*****************************************************************
// Function implemented with normal/scalar C code
//*****************************************************************
void array2d_add_ref(int * A, int * B, int * C,
           int blkw, int blkh,
           int lofst_A, int lofst_B, int lofst_C)
{
  for (int i=0 ; i<blkh; i++)
    for (int j=0 ; j<blkw; j++)
    {
      C[i * lofst_C + j] = A[i * lofst_A + j] + B[i * lofst_B + j];
    }
}
```

As each source and operand array is two dimensional, in the function's arguments we convey block width and block height of the computation, and line offset for each operand array. This function uses two levels of nested for loops to iterate through rows and columns. In the loop body, the statement carrying out the addition operation indexes into each operand array with two-dimensional indexing to acquire each input data element and to store each output data element.

Compiled assembly, along with execution count and cycle count is shown next:

```
Function detail: array2d_add_ref
void_array2d_add_ref___P__sint___P__sint___P__sint___sint___sint___sint___sint___sint

  Low PC        : 168
  High PC        : 199
  Size in program memory: 32
  Cycle-count      : 14438 (13.01%)
  Instruction-count   : 6218 ( 7.16%)
  Instruction Coverage : 100.00%
```

```
PC    Assembly                                                                 Exe-cnt  Cycs
---- ------------------------------------------------------------------------ ------- ------
 168 CMPLE R7,R0,R2 || CMPLE R8,R0,R3                                             1      5
 170 BNEZ R3,#TGT_Fvoid_array2d_add_ref_23                                        1      1
 171 NOP                                                                          1      1
 172 NOP                                                                          1      1
 173 RPT R8,#LE_Fvoid_array2d_add_ref_22                                          1      1
 174 ORI R0,#4,R3                                                                 1      1
 175 NOP                                                                          1      1
 176 BNEZ R2,#TGT_Fvoid_array2d_add_ref_20                                        8      8
 177 NOP                                                                          8      8
 178 NOP                                                                          8     22
 179 RPT R7,#LE_Fvoid_array2d_add_ref_19                                          8      8
 180 ORI R6,#0,R13                                                                8      8
 181 MOV R5,R8 || MOV R4,R12 || NOP                                               8     16
 184 LDW *R12+=R3,R17 || LDW *R8+=R3,R14                                       2048  10240
 186 ADD R14,R17,R18                                                           2048   2048
.label #LE_Fvoid_array2d_add_ref_19
 187 STW R18,*R13+=R3                                                          2048   2048
.label TGT_Fvoid_array2d_add_ref_20
 188 SLLIADD R10,#2,R5,R5|| SLLIADD R9,#2,R4,R4                                   8      8
.label LE_Fvoid_array2d_add_ref_22
 190 SLLIADD R11,#2,R6,R6                                                         8      8
.label TGT_Fvoid_array2d_add_ref_23
 191 JR R15                                                                       1      1
 192 NOP                                                                          1      1
 193 NOP || NOP || NOP || NOP || NOP || NOP || NOP                                1      3
```

The block width and height are configured as 256 and 8 respectively. The execution count numbers show execution packets that are outside the loops (those with execution count of 1), between the loops (those with execution count of 8), and inside the inner-most loop (those with execution count of 2048).

Compared to the one-dimensional array addition with the same number of element-wise additions, this function takes 14438 – 14352 = 86 cycles longer, or 0.6% slower. We can look at these additional number cycles as the cost of performing two-dimensional addressing. This cost strongly depends on the block width and height.

The additional number of cycles (86) depends only on the block height, as the compiled code has a fixed number of instructions between loop levels, and they are executed 8 times in this case because the outer loop is iterated 8 times.

The proportion of cycles (0.6%) spent between the loops roughly depends only on the block width. The compiled code has a fixed number of instructions in the innermost loop body as well, which is executed block_width * block_height = 2048 times. Thus, proportion of time spent between loop levels is some (K1 * block_height) / (K2 * block_width * block_height) = K1 / (K2 * block_width) = K3 / block_width. The wider the block width, the smaller proportion of time spent between loop levels.

In this code example, we do not see a large proportion of time spent handling two-dimensional addressing, but this is due to the block width being large enough for the inner loop to be unrolled 8 times and with sufficient iteration count to support the unrolling, as 16 * 16 = 256. If the block width is less than 256, we would see a larger proportion of processing time spent on two-dimensional addressing. Later in Performance Across 2D Array Dimensions, we will show cycle counts across different block dimension configurations.

8.3.2 Optimization 1: Vectorized, Unrolled and Pipelined Loop

Here we apply the vectorization and unrolling/pipelining techniques shown in Optimization 1: Vectorized Code and Optimization 2: Unroll and Pipeline the Loop respectively on the two-dimensional addition function.

```
//*********************************************************************
// Optimization 1: vectorized, unrolled and pipelined
//*********************************************************************
void array2d_add_opt1(int * A, int * B, int * restrict C,
           int blkw, int blkh,
           int lofst_A, int lofst_B, int lofst_C)
{
  dvintx vA, vB, vC;
  int idx_A, idx_B, idx_C;
  int vecw = chess_elementsof(dvint);
  dvint * vptrA = (dvint *) A;
  dvint * vptrB = (dvint *) B;
  dvint * restrict vptrC = (dvint *) C;

  for (int i=0 ; i<blkh; i++)
  {
    for (int j=0 ; j<blkw/vecw; j++) chess_loop_range(16,)
        chess_unroll_loop(8) chess_prepare_for_pipelining
    {
      vA = sign_extend(*vptrA++);
      vB = sign_extend(*vptrB++);
      vC = vA + vB;
      *vptrC++ = extract(vC);
    }
    A += lofst_A;
    B += lofst_B;
    C += lofst_C;
    vptrA = (dvint *) A;
    vptrB = (dvint *) B;
    vptrC = (dvint *) C;
  }
}
```

We still need nested for loops to iterate horizontally and vertically. After the inner loop, between loop levels, there is an update of pointers to adjust for the line offset so we can start the next row coming back to the inner loop.

The chess_loop_range, chess_unroll_loop, and chess_prepare_for_pipelining pragmas are applied only to the inner loop, as it is generally not improving performance to apply them on the outer loop as well.

The profiling instruction report is shown next:

```
Function detail: array2d_add_opt1
void_array2d_add_opt1___P__sint___P__sint___P__sint___sint___sint___sint___sint___sint

  Low PC        : 200
  High PC        : 287
  Size in program memory: 88
  Cycle-count      : 226 ( 0.20%)
  Instruction-count   : 204 ( 0.23%)
  Instruction Coverage : 100.00%

 PC   Assembly                                                                    Exe-cnt  Cycs
---- ------------------------------------------------------------------------- ------- ------
200 CMPLEI R8,#0,R2                                                                 1     5
 201 BNEZ R2,#TGT_Fvoid_array2d_add_opt1_82                                         1     1
 202 NOP                                                                            1     1
 203 NOP                                                                            1     1
 204 SRAI R7,#31,R3                                                                 1     1
 205 ANDI R3,#15,R3                                                                 1     1
 206 RPT R8,#LE_Fvoid_array2d_add_opt1_81 || ADD R3,R7,R7                           1     1
 208 SRAI R7,#7,R7                                                                  1     1
 209 ORI R0,#64,R2 || ADDI R7,#-1,R7                                                1     1
 211 MOV R5,R3 || MOV R4,R12                                                        8    16
 213 DVLDW_P *R12+=R2,V30:V31|| DVLDW_P *R3+=R2,V26:V27                             8     8
 215 DVLDW_P *R12+=R2,V22:V23|| DVLDW_P *R3+=R2,V18:V19                             8     8
 217 DVLDW_P *R12+=R2,V14:V15|| DVLDW_P *R3+=R2,V10:V11                             8     8
 219 DVLDW_P *R12+=R2,V6:V7|| DVLDW_P *R3+=R2,V2:V3                                 8     8
 221 DVLDW_P *R12+=R2,V4:V5|| DVLDW_P *R3+=R2,V0:V1                                 8     8
 223 RPT R7,#LE_Fvoid_array2d_add_opt1_62 || DVLDW_P *R12+=R2                       8     8
 226 ORI R6,#0,R8 || VAddW V30:V31,V26:V27,V24:V25|| DVLDW_P *R12+=R2,V20:V21||
     DVLDW_P *R3+=R2,V16:V17                                                        8    16
 230 VAddW V22:V23,V18:V19,V18:V19|| DVLDW_P *R12+=R2,V28:V29|| DVSTW_P V24:V25,*R8+=R2||
     DVLDW_P *R3+=R2,V24:V25                                                        8     8
 234 VAddW V14:V15,V10:V11,V10:V11|| DVLDW_P *R12+=R2,V30:V31|| DVLDW_P *R3+=R2,V26:V27||
     DVSTW_P V18:V19,*R8+=R2                                                        8     8
 238 VAddW V6:V7,V2:V3,V2:V3|| DVLDW_P *R12+=R2,V22:V23|| DVLDW_P *R3+=R2,V18:V19||
     DVSTW_P V10:V11,*R8+=R2                                                        8     8
 242 VAddW V4:V5,V0:V1,V0:V1|| DVLDW_P *R12+=R2,V14:V15|| DVLDW_P *R3+=R2,V10:V11||
     DVSTW_P V2:V3,*R8+=R2                                                          8     8
 246 VAddW V12:V13,V8:V9,V8:V9|| DVLDW_P *R12+=R2,V6:V7|| DVLDW_P *R3+=R2,V2:V3||
     DVSTW_P V0:V1,*R8+=R2                                                          8     8
 250 VAddW V20:V21,V16:V17,V16:V17|| DVLDW_P *R12+=R2,V4:V5|| DVLDW_P *R3+=R2,V0:V1||
```

```
     DVSTW_P V8:V9,*R8+=R2                                                                  8     8
 254 VAddW V28:V29,V24:V25,V28:V29|| DVLDW_P *R12+=R2,V12:V13|| DVLDW_P *R3+=R2,V8:V9||
     DVSTW_P V16:V17,*R8+=R2                                                                8     8
 258 VAddW V30:V31,V26:V27,V24:V25|| DVLDW_P *R12+=R2,V20:V21|| DVLDW_P *R3+=R2,V16:V17||
     DVSTW_P V28:V29,*R8+=R2                                                                8     8
.label LE_Fvoid_array2d_add_opt1_62
 262 VAddW V22:V23,V18:V19,V18:V19|| DVLDW_P *R12+=R2,V28:V29|| DVSTW_P V24:V25,*R8+=R2||
     DVLDW_P *R3+=R2,V24:V25                                                                8     8

 266 SLLIADD R9,#2,R4,R4|| SLLIADD R10,#2,R5,R5|| VAddW V14:V15,V10:V11,V10:V11||
     DVSTW_P V18:V19,*R8+=R2                                                                8     8
 270 SLLIADD R11,#2,R6,R6|| VAddW V6:V7,V2:V3,V2:V3|| DVSTW_P V10:V11,*R8+=R2           8     8
 273 VAddW V4:V5,V0:V1,V0:V1|| DVSTW_P V2:V3,*R8+=R2                                    8     8
 275 VAddW V12:V13,V8:V9,V8:V9|| DVSTW_P V0:V1,*R8+=R2                                  8     8
 277 VAddW V20:V21,V16:V17,V16:V17|| DVSTW_P V8:V9,*R8+=R2                              8     8
 279 VAddW V28:V29,V24:V25,V28:V29|| DVSTW_P V16:V17,*R8+=R2                            8     8
.label LE_Fvoid_array2d_add_opt1_81
 281 DVSTW_P V28:V29,*R8+=R2                                                            8     8
.label TGT_Fvoid_array2d_add_opt1_82
 282 JR R15                                                                             1     1
 283 NOP                                                                                1     1
 284 NOP || NOP || NOP || NOP                                                           1     3
```

It is not easy to spot the inner loop from the report, as the execution counts are 8 for both between-loop packets and inner loop packets. This is because of the inner-loop is unrolled 8 times, and with prolog and epilog together executing unrolled loop once, the actual inner loop body is executed just once, as 256 / (8 * chess_elementsof(dvint)) – 1 = 256 / 128 – 1 = 1. Blank lines are manually inserted to better visualize the innermost loop.

There is still a significant speedup from the scalar code, 14438/226 = 64.9 times. The inner loop is still packed with 1 VAddW, 2 DVLDW, and 1 DVSTW per execution packet, in all 8 execution packets.

The use of nested for loops and clock cycles spent between loop levels does add to the overhead. Compared to the ideal time spent, which is (256 * 8) / 16 = 128 cycles, the function execution time is only 128/226 = 57% efficient. There is relatively high overhead to handle 2D addressing, versus 91% efficient in the 1D array addition case.

As argued in the previous section, on scalar code performance, proportion of time spent between loop levels is mostly a function of the inner-loop iteration count. In the configuration where profiling instruction report is generated, we operate on 8 tall x 256 wide arrays. If it's not as "short-and-wide" in aspect ratio, say it's 16 tall x 128 wide or 32 tall x 64 wide, we don't have sufficient number of iterations for the inner-most loop to fully unroll and pipeline, and we have smaller iteration count on the inner loop, and both would contribute to reducing the overall efficiency of the code.

In Performance Across 2D Array Dimensions we will present function cycle count across various 2D array dimensions.

8.3.3 Optimization 2: Leveraging Agen to Collapse Nested Loops

In this section, we tackle the performance degradation from two-dimensional addressing.

In image and vision processing, we often need an even higher dimension of address calculation. For example, in 2D convolution, we have 2 dimensions from producing some block-width x block-height of output block, and we have kernel-width x kernel-height looping to perform convolution between points in the 2D convolution kernel and 2D neighborhood around each output pixel. The address generator, or agen, feature is there to support up to 6 dimensions of address calculation.

The following optimized code shows how agens are configured and utilized for the 2D array addition function:

```
//*********************************************************************
// Optimization 2: leverage agen, initialization
//*********************************************************************
void array2d_add_opt2_init(int * A, int * B, int * C,
              int blkw, int blkh,
              int lofst_A, int lofst_B, int lofst_C,
              int * niter, AgenCFG * agen_ptr)
{
  int vecw = chess_elementsof(dvint);
  dvintx vA, vB, vC;
  agen in0, in1, out;
  short niter1 = blkw/vecw;
  short niter2 = blkh;
  * niter = niter1 * niter2;
  agen_wrapper_t wrapper;

  in0 = init(A);
  wrapper.size = sizeof(int);
  wrapper.n1 = niter1;
  wrapper.n2 = niter2;
  wrapper.s1 = vecw;
  wrapper.s2 = lofst_A;
  INIT_AGEN2(in0, wrapper);

  in1 = init(B);
  wrapper.size = sizeof(int);
  wrapper.n1 = niter1;
  wrapper.n2 = niter2;
  wrapper.s1 = vecw;
  wrapper.s2 = lofst_B;
  INIT_AGEN2(in1, wrapper);

  out = init(C);
  wrapper.size = sizeof(int);
```

```
  wrapper.n1 = niter1;
  wrapper.n2 = niter2;
  wrapper.s1 = vecw;
  wrapper.s2 = lofst_C;
  INIT_AGEN2(out, wrapper);

  chess_separator_scheduler();

  *agen_ptr++ = extract_agen_cfg(in0);
  *agen_ptr++ = extract_agen_cfg(in1);
  *agen_ptr++ = extract_agen_cfg(out);
}

//*********************************************************************
// Optimization 2: leverage agen
//*********************************************************************
void array2d_add_opt2(int niter, AgenCFG * agen_ptr)
{
  agen_A in0 = init_agen_A_from_cfg(*agen_ptr++);
  agen_B in1 = init_agen_B_from_cfg(*agen_ptr++);
  agen_C out = init_agen_C_from_cfg(*agen_ptr++);
  dvintx vA, vB, vC;

  for (int i=0 ; i<niter; i++) chess_loop_range(16,)
              chess_unroll_loop(8) chess_prepare_for_pipelining
  {
    vA = dvint_load(in0);
    vB = dvint_load(in1);
    vC = vA + vB;
    vstore(vC, out);
  }
}
```

There are 2 functions, array2d_add_opt2_init() and array2d_add_opt2(). Agen parameter calculation and configuration is placed in an “init” function meant to be called just once or twice per application. By separating out the agen parameter calculation and configuration portion, we reduce the per-tile computation time.

The configured agens are saved to memory via the AgenCfgST instruction (see AgenCfgST) one at a time, and are restored from memory via AgenCfgLD instruction (see AgenCfgLD) one at a time before the compute loop.

In the init function, we still must calculate inner loop number of iterations, niter1, and outer loop number of iterations, niter2, but they are not used to iterate nested for loops. Instead, they are used in agen programming, as it’s agen that needs to know about these iteration counts to carry out the 2D addressing. Product of niter1 and niter2, niter, is returned to the main function, to supply to the compute function to iterate the collapsed for loop.

In the init function, we declare wrapper variable of agen_wrapper_t type. Using agen wrapper allows the programmer to specify the step size of various dimensions and use

macros like INIT_AGEN2 to carry out the complex expressions (see 6.4.1) to calculate the address modifiers, instead of coding the complex expressions directly. In general, we pre-determine dimension needed in the agens, say K dimensions, we program wrapper n1..nk, s1..sk, and then call INIT_AGENk to complete the agen programming.

There is a straightforward producess to convert the indexing expression in the scalar code into the step parameters s1..sk for the wrapper. For example, array A is indexed in the scalar code as:

```
A[i * lofst_A + j]
```

We map the inner loop variable j into loop level 1 of agen, and outer loop variable i into loop level 2.

Furthermore, in the process of vectorizing the 2D array addition, we process one dvint at a time, so the original indexing should be converted into loading from

```
A + i2 * lofst_A + i1 * vecw
```

with vecw = chess_elementsof(dvint). We take the vectorized indexing expression and basically fill step parameters s1..sk with whatever scaling factor is being multiplied with the corresponding loop variable i1..ik. Thus, we program them as

```
wrapper.s1 = vecw;
wrapper.s2 = lofst_A;
```

In the loop body, loading through agen-based load is performed via intrinsic function dvint_load(agen), and storing through agen-based store is performed via intrinsic function vstore(variable, agen). vstore() function is type-overloaded to handle various vector data types.

The 2 agens for load, in0 and in1, and the one agen for store, out, are declared as variables of agen_A/B/C types respectively. These _A/B/C suffixes are to denote superbank A/B/C. They do not really need to match the actual pointer values being in superbank A/B/C, but are there to guide compiler scheduling, so that we don't load from the same superbank to store to the same superbank multiple times in an execution packet and cause unnecessary performance degradation.

With the agen taking up the 2D address calculation, we can collapse the 2 levels of nested for loops into just one level and run it niter = niter1 * niter2 times. This also helps with unrolling and software pipelining, as the number of iterations being a multiple of 8 and being at least 16 are now constraints on the overall loop iteration count, and can apply to more array dimension cases.

The profiling instruction report is shown next:

```
Function detail: array2d_add_opt2 void_array2d_add_opt2___sint___Pdvuint

  Low PC        : 344
  High PC        : 415
  Size in program memory: 72
  Cycle-count      : 146 ( 0.13%)
  Instruction-count   : 141 ( 0.16%)
  Instruction Coverage : 100.00%
```

```
 PC   Assembly                                                                        Exe-cnt  Cycs
---- ------------------------------------------------------------------------------ ------- -----
 344 ORI R0,#64,R4 || SRAI R4,#3,R2                                                       1     1
 346 ADDI R2,#-1,R2                                                                       1     1
 347 AgenCfgLD *R5+=R4,A0                                                                 1     1
 348 AgenCfgLD *(R5+0),A2                                                                 1     1
 349 AgenCfgLD *(R5+64),A1                                                                1     4
 350 DVLDW_P *A0++,W12:W13                                                                1     1
 351 DVLDW_P *A0++,V10:V11|| DVLDW_P *A2++,V14:V15                                        1     1
 353 DVLDW_P *A0++,V6:V7|| DVLDW_P *A2++,W8:W9                                            1     1
 355 DVLDW_P *A0++,V2:V3|| DVLDW_P *A2++,W4:W5                                            1     1
 357 DVLDW_P *A0++,V0:V1|| DVLDW_P *A2++,W0:W1                                            1     1
 359 DVLDW_P *A0++,V4:V5|| DVLDW_P *A2++,W2:W3                                            1     1
 361 RPT R2,#LE_Fvoid_array2d_add_opt2_54|| DVLDW_P *A2++,W6:W7                           1     1
 363 VAddW W12:W13,V14:V15,V16:V17|| DVLDW_P *A0++,V8:V9|| DVLDW_P *A2++,W10:W11          1     1
 366 VAddW V10:V11,W8:W9,V18:V19|| DVLDW_P *A0++,V12:V13|| DVLDW_P *A2++,W14:W15||
     DVSTW_P V16:V17,*A1++                                                                1     1
 370 VAddW V6:V7,W4:W5,V20:V21|| DVSTW_P V18:V19,*A1++|| DVLDW_P *A0++,W12:W13||
     DVLDW_P *A2++,V14:V15                                                               15    15
 374 VAddW V2:V3,W0:W1,V22:V23|| DVLDW_P *A0++,V10:V11|| DVLDW_P *A2++,W8:W9||
     DVSTW_P V20:V21,*A1++                                                               15    15
 378 VAddW V0:V1,W2:W3,V24:V25|| DVLDW_P *A0++,V6:V7|| DVLDW_P *A2++,W4:W5||
     DVSTW_P V22:V23,*A1++                                                               15    15
 382 VAddW V4:V5,W6:W7,V26:V27|| DVLDW_P *A0++,V2:V3|| DVLDW_P *A2++,W0:W1||
     DVSTW_P V24:V25,*A1++                                                               15    15
 386 VAddW V8:V9,W10:W11,V28:V29|| DVLDW_P *A0++,V0:V1|| DVLDW_P *A2++,W2:W3||
     DVSTW_P V26:V27,*A1++                                                               15    15
 390 VAddW V12:V13,W14:W15,V30:V31|| DVLDW_P *A0++,V4:V5|| DVLDW_P *A2++,W6:W7||
     DVSTW_P V28:V29,*A1++                                                               15    15
 394 VAddW W12:W13,V14:V15,V16:V17|| DVLDW_P *A0++,V8:V9|| DVLDW_P *A2++,W10:W11||
     DVSTW_P V30:V31,*A1++                                                               15    15
.label LE_Fvoid_array2d_add_opt2_54
 398 VAddW V10:V11,W8:W9,V18:V19|| DVLDW_P *A0++,V12:V13|| DVLDW_P *A2++,W14:W15||
     DVSTW_P V16:V17,*A1++                                                               15    15
 402 VAddW V6:V7,W4:W5,V20:V21|| DVSTW_P V18:V19,*A1++                                    1     1
 404 VAddW V2:V3,W0:W1,V22:V23|| DVSTW_P V20:V21,*A1++                                    1     1
 406 VAddW V0:V1,W2:W3,V24:V25|| DVSTW_P V22:V23,*A1++                                    1     1
 408 VAddW V4:V5,W6:W7,V26:V27|| DVSTW_P V24:V25,*A1++                                    1     1
 410 JR R15 || VAddW V8:V9,W10:W11,V28:V29|| DVSTW_P V26:V27,*A1++                        1     1
 413 VAddW V12:V13,W14:W15,V30:V31|| DVSTW_P V28:V29,*A1++                                1     1
 415 DVSTW_P V30:V31,*A1++                                                                1     3
```

Now the loop body stands out, as there is just one loop level. Scalar code before the loop is relatively terse, as agen parameter calculation and configuration is moved to the init function, which takes 51 cycles (not shown here). We don't add these cycles to the tile compute function cycle count, as the init function is run just once per application.

It takes just 146 cycles to run the per-tile compute function, compared to 226 cycles in the vectorized and unrolled/pipelined version that still needs to deal with 2D address calculation. In this version, agen hardware takes care of 2D address calculation in the

background, so we are not spending any clock cycle. The efficiency of this code is 128/146 = 88%.

8.3.4 Performance Across 2D Array Dimensions

We vary the array dimension and collect cycle count, as follows. For optimization 1 and 2, efficiency ratios vs ideal cycle counts are also shown in parenthesis.

Table 17. Performance optimization across array dimensions

Array Height	Array Width	Scalar code cycles	Optimization 1 (vector, unroll/pipeline) cycles (efficiency %)	Optimization 2 (vector, unroll/pipeline, agen) cycles (efficiency %)
4	512	14,394	186 (69%)	146 (88%)
8	256	14,438	226 (57%)	146 (88%)
16	128	14,526	322 (40%)	146 (88%)
16	512	57,534	690 (74%)	530 (97%)
32	256	57,710	850 (60%)	530 (97%)
64	128	58,062	1,234 (41%)	530 (97%)

We can see that optimization 2 code's performance is not at all sensitive to block width versus height changes, only to the total number of data points, and efficiency is good. Scalar code performance is a weak function of the block width, wider blocks perform slightly better. Optimization 1 code's performance is better than scalar code, but is worse than optimization 2 code's performance, and the narrower the block width, the worse off it gets.

8.4 2D Convolution

Next, we see how 2D convolution, a common image processing step, is accelerated by leveraging the multi-dimension address calculation feature of agens, along with store-path rounding and predicated vector math instructions.

8.4.1 Scalar Code

A straightforward implementation of 2D convolution is as follows.

```
//*********************************************************************
// Filter implemented with natural C code to do 2D addressing
//*********************************************************************
void filter_short_ref(short *data, short *coef, short *out,
           int kw, int kh, int qbits, int blkw, int blkh,
           int lofst_data, int lofst_out)
{
```

```
  short sdata;
  short scoef;
  int  prod;
  long long acc;

  int rnd_add = (qbits == 0) ? 0 : (1 << (qbits-1));

  for (int i4=0 ; i4<blkh; i4++)
  for (int i3=0 ; i3<blkw; i3++) {
    acc = 0;
    for (int i2=0 ; i2<kh; i2++)
    for (int i1=0 ; i1<kw; i1++) {
      sdata = data[(i4 + i2)*lofst_data + i3 + i1];
      scoef = coef[i2*kw + i1];
      prod = sdata * scoef;
      acc += prod;
    }
    acc = (acc + rnd_add) >> qbits;
    out[i4*lofst_out + i3] = acc;
  }
}
```

The function carries out 2D convolution with 4 levels of nested for loop. The 4 levels of looping are needed to drive indexing of data and coefficient arrays and output array. Data indexing has 4 dimensions, horizontally and vertically to traverse in the kw x kh neighborhood to perform dot-product with the coefficient array, and then horizontally one vector width at a time, vertically one row at a time, to produce the 2D array output. Coefficient and output each have 2 dimensions of indexing.

There are statements between the outer 2 loop levels and the inner 8 loop levels. Before entering the inner 2 loop levels, we clear the accumulator. After exiting the inner 2 loop levels, having already accumulated kw * kh products to the accumulator, we perform rounding on the accumulated sum then store the rounded outcome to the output array.

The profiling instruction report of this scalar code is as follows:

```
Function detail: filter_short_ref void_filter_short_ref_ ...

  Low PC        :     56
  High PC        :    143
  Size in program memory:     88
  Cycle-count      :   285663 (71.77%)
  Instruction-count   :   191024 (73.10%)
  Instruction Coverage : 94.92%

 PC   Assembly                                                                Exe-cnt  Cycs
 --- ------------------------------------------------------------------------ ------- ------
 56 ADDI R9,#-1,R2 || ADDI R1,#20,R1                                            1      1
 58 CMPEQ R9,R0,R19 || ORI R0,#1,R3                                             1      1
 60 SLL R3,R2,R2 || CMPLTI R9,#32,R13|| STW R13,*(R1+2036)                      1      1
 63 ADDI R9,#-32,R18 || CMPLEI R7,#0,R14|| STW R10,*(R1+2028)                   1      1
```

```
66 ORI R0,#2,R20 || ORI R0,#32,R3                                    1      1
68 J #__ll4_void_filter_short_ref || ORI R0,#0,R23                   1      1
70 SUB R3,R9,R2 || MUX R19,R0,R2,R17|| STW R11,*(R1+2032)           1      1
73 SRAI R17,#31,R15 || CMPLEI R8,#0,R3|| STW R15,*(R1+2040)         1      3
76 J #__ll3_void_filter_short_ref                                   32     32
77 SLLIADD R10,#1,R6,R10                                            32     32
78 ORI R0,#0,R22 || MOV R6,R11                                      32     96
80 BNEZ R3,#TGT_J_Fvoid_filter_short_ref_81                        2048   2048
81 ADD R4,R22,R24                                                  2048   2048
82 NOP                                                             2048   2048
83 RPT R8,#TGT_Fvoid_filter_short_ref_48                           2048   2048
84 LHI #0,R28                                                      2048   2048
85 ORI R0,#0,R23 || MOV R5,R25                                     2048   2048
87 BNEZ R14,#TGT_Fvoid_filter_short_ref_48                         6144   6144
88 NOP                                                             6144   6144
89 NOP                                                             6144  14336
90 RPT R7,#LE_Fvoid_filter_short_ref_46                            6144   6144
91 MOV R24,R27 || MOV R25,R26                                      6144   6144
93 NOP                                                             6144   6144
94 LDH *R27+=R20,R30 || LDH *R26+=R20,R29                         18432  92160
96 MUL R29,R30,R30                                                18432  18432
97 ADD R23,R30,R30 || SRAI R30,#31,R31                            18432  18432
99 ORI R30,#0,R21                                                 18432  18432
100 ADD R28,R31,R30 || CMPLTU R30,R23,R31                         18432  18432
102 MOV R21,R23 || ADD R30,R31,R28                                18432  18432
104 SLLIADD R12,#1,R24,R24|| SLLIADD R7,#1,R25,R25                 6144   6144
106 ADD R15,R28,R25 || ADD R17,R23,R24                             2048   2048
108 SRL R24,R9,R23 || CMPLTU R24,R23,R26                           2048   2048
110 ADDI R22,#2,R22 || ADD R25,R26,R25                             2048   2048
112 SRA R25,R18,R25 || SLL R25,R2,R26                              2048   2048
114 OR R23,R26,R23                                                 2048   2048
115 MUX R13,R23,R25,R23                                            2048   2048
116 MUX R19,R24,R23,R23                                            2048   2048
117 STH R23,*R11+=R20                                              2048   2048
118 CMPLTU R11,R10,R23                                             2080  10400
119 BNEZ R23,#TGT_Fvoid_filter_short_ref_24                        2080   2080
120 NOP                                                            2080   2080
121 NOP                                                            2080   6176
122 SLLIADD R12,#1,R4,R4|| LDW *(R1+2044),R23                       32     32
124 LDW *(R1+2036),R22                                              32     32
125 LDW *(R1+2032),R11                                              32     32
126 LDW *(R1+2028),R10                                              32     64
127 ADDI R23,#1,R23                                                 32     32
128 SLLIADD R22,#1,R6,R6                                            32     32
129 CMPLT R23,R11,R11                                               33    165
130 BNEZ R11,#TGT_Fvoid_filter_short_ref_20                         33     33
131 STW R23,*(R1+2044)                                              33     33
132 NOP                                                             33     97
133 LDW *(R1+2040),R4                                                1      8
```

```
134 JR R4                                                                    1      1
135 ADDI R1,#-20,R1                                                          1      1
136 NOP                                                                      1      3
137 J #TGT_Fvoid_filter_short_ref_50                                         0      0
138 ORI R0,#0,R28 || ORI R0,#0,R23                                           0      0
140 NOP || NOP || NOP || NOP                                                 0      0
```

The function takes 285,663 cycles to compute a 64 wide x 32 tall outputs worth of 2D convolution, about 140 cycles per output, or about 15 cycles per data-coefficient product. From the rising then falling numbers in the execution count and cycle count, we can tell where the boundaries of 4 levels of for loop are.

In the innermost loop with execution count of 18432 (which is 64 * 32 * 9), we have a 10 cycle loop, as (92160 + 5*18432) / 18432 = 10. These 10 cycles are from 5 cycles of load and latency, multiply, add, then a few cycles to perform array indexing needed for the inner-most loop.

Later in the optimized code, we will see how various VPU instructions and agen features are leveraged, so that we perform all these, loading data/coefficient, multiply-add, index update, and in vectorized form so doing a double short vector worth thus 32 sets of these, in one cycle. Moreover, the 4 nested for loops are collapsed into one single loop, with periodic accumulator initialization and rounding and storing of output all absorbed into the loop body.

8.4.2 Optimization 1: Vectorized and Agen Optimized Loop

As we have learned in Optimization 2: Leveraging Agen to Collapse Nested Loops, besides vectorization and loop unrolling, software pipelining, we can leverage multi-dimensional addressing capability of agens to collapse nested for loops. The following example code includes two functions. There's an initialization function to calculate/configure agen parameters and save the agen configurations to memory. Then there is a run-time compute function to restore the agens and run the filtering loop.

```
//*********************************************************************
// Filter optimized, initialization function
//*********************************************************************
void filter_short_opt1_init(short *data, short *coef, short* restrict out,
         int kw, int kh, int qbits, int blkw, int blkh,
         int lofst_data, int lofst_out, int * niter_ptr,
         AgenCFG * cfg_ptr)
{
  int  vecw = chess_elementsof(dvshort);
  short niter1 = kw;
  short niter2 = kh;
  short niter3 = blkw/vecw;
```

```
  short niter4 = blkh;
  * niter_ptr++ = niter1 * niter2 * niter3 * niter4;
  * niter_ptr  = niter1 * niter2;
  agen data_agen, coef_agen, out_agen;
  agen_wrapper_t wrapper;

  data_agen = init((vshort*) data);
  wrapper.size = sizeof(short);
  wrapper.n1 = kw;
  wrapper.n2 = kh;
  wrapper.n3 = blkw/vecw;
  wrapper.n4 = blkh;
  wrapper.s1 = 1;
  wrapper.s2 = lofst_data;
  wrapper.s3 = vecw;
  wrapper.s4 = lofst_data;
  INIT_AGEN4(data_agen, wrapper);

  coef_agen = init((vshort*)coef);
  wrapper.size = sizeof(short);
  wrapper.n1 = kw * kh;
  wrapper.n2 = (blkw/vecw) * blkh;
  wrapper.s1 = 1;
  wrapper.s2 = 0;
  INIT_AGEN2(coef_agen, wrapper);

  out_agen = init((vshort*)out);
  wrapper.size = sizeof(short);
  wrapper.n1 = kw * kh;
  wrapper.n2 = blkw/vecw;
  wrapper.n3 = blkh;
  wrapper.s1 = 0;
  wrapper.s2 = vecw;
  wrapper.s3 = lofst_out;
  INIT_AGEN3(out_agen, wrapper);
  out_agen.round = qbits;

  chess_separator_scheduler();

  *cfg_ptr++ = extract_agen_cfg(data_agen);
  *cfg_ptr++ = extract_agen_cfg(coef_agen);
  *cfg_ptr++ = extract_agen_cfg(out_agen);
}

//*********************************************************************
// Filter optimized, run-time compute function
//*********************************************************************
void filter_short_opt1(int * niter_ptr, AgenCFG * cfg_ptr)
{
```

```
  int  count_madd = 0;
  int  count_store = 1;
  int  pred_madd = 0;
  int  pred_store = 0;
  int  niter = * niter_ptr++;
  int  niter_in = * niter_ptr;
  dvshortx dvdata;
  int   coef;
  dvintx dvacc0, dvacc1;

  agen_A data_agen = init_agen_A_from_cfg(*cfg_ptr++);
  agen_B coef_agen = init_agen_B_from_cfg(*cfg_ptr++);
  agen_C out_agen = init_agen_C_from_cfg(*cfg_ptr++);

  chess_separator_scheduler();

  for (int i=0; i<niter; i++) chess_prepare_for_pipelining
            chess_unroll_loop(8) chess_loop_range(16,) {

    dvdata = dvshort_load_di(data_agen);

    coef = short_load(coef_agen);

    dvacc0 = vmaddhw(dvdata.lo, coef, dvacc0, VPU_ROUND_0, pred_madd);
    dvacc1 = vmaddhw(dvdata.hi, coef, dvacc1, VPU_ROUND_0, pred_madd);

    vstore_i(dvacc0, dvacc1, out_agen, pred_store);

    count_madd = (count_madd == niter_in-1) ? 0 : (count_madd + 1);
    pred_madd  = (count_madd!=0) ? (int)0xFFFFFFFF : 0;
    count_store = (count_store == niter_in-1) ? 0 :(count_store + 1);
    pred_store = (count_store==0) ? (int)0xFFFFFFFF : 0;
  }
}
```

Agen programming for data, coefficients and outputs follow the nested loop iteration counts and data, coefficient, and output indexing in the scalar code.

Nested for loops in the scalar code:

```
  for (int i4=0 ; i4<blkh; i4++)
  for (int i3=0 ; i3<blkw; i3++) {
    ...
    for (int i2=0 ; i2<kh; i2++)
    for (int i1=0 ; i1<kw; i1++) {
      ...
    }
    ...
  }
```

Let's compare the data indexing in the scalar code:

```
sdata = data[(i4 + i2)*lofst_data + i3 + i1];
```

with data agen programming:

```
data_agen = init((vshort*) data);
wrapper.size = sizeof(short);
wrapper.n1 = kw;
wrapper.n2 = kh;
wrapper.n3 = blkw/vecw;
wrapper.n4 = blkh;
wrapper.s1 = 1;
wrapper.s2 = lofst_data;
wrapper.s3 = vecw;
wrapper.s4 = lofst_data;
INIT_AGEN4(data_agen, wrapper);
```

The iteration counts are translated directly to the iteration counts in data agen programming, except that in i3 loop, we run for blkw/vecw iterations instead of blkw, due to computing vecw elements of the output array in parallel through the vectorization process. The step amount is adjusted accordingly to vecw elements instead of one.

Coefficient indexing in the scalar code:

```
scoef = coef[i2*kw + i1];
```

Coefficient agen programming:

```
coef_agen = init((vshort*)coef);
wrapper.size = sizeof(short);
wrapper.n1 = kw * kh;
wrapper.n2 = (blkw/vecw) * blkh;
wrapper.s1 = 1;
wrapper.s2 = 0;
INIT_AGEN2(coef_agen, wrapper);
```

In this instance, we lump the scalar for loops i1 and i2 into just one dimension in coefficient agen. This is because coefficient indexing just advance by one element per iteration in the inner 2 loop levels. In coefficient agen programming we can just use one loop level with the combined number of iterations kw * kh to comprehend the inner 2 loop levels in the scalar code.

In the scalar code, coefficient indexing has no i3 or i4 components, the two outer loop variables. Consequently, coefficient addressing just repeats the same pattern when we iterate the outer loops. In agen programming, we accomplish this repeating pattern by configuring an outer dimension n2 parameter to the combined iteration count of the two outer loops, (blkw/vecw) * blkh, and with step amount s2 configured to 0.

Output indexing in the scalar code:

```
   out[i4*lofst_out + i3] = acc;
```

Output agen programming:

```
out_agen = init((vshort*)out);
```

```
wrapper.size = sizeof(short);
wrapper.n1 = kw * kh;
wrapper.n2 = blkw/vecw;
wrapper.n3 = blkh;
wrapper.s1 = 0;
wrapper.s2 = vecw;
wrapper.s3 = lofst_out;
INIT_AGEN3(out_agen, wrapper);
out_agen.round = qbits;
```

In the scalar code, output is stored out between the inner 2 loops and outer 2 loops, as in the inner 2 loops we are accumulating the products between data points and coefficients, and only when we are out of the inner 2 loops, we are ready to store the outcome to memory.

In the optimized code, the store is placed in the loop body instead of in an outer loop. Of course, it is functionally correct to move the store to the outer loop. but doing that would introduce much loop prolog/epilog time between loop levels and slow down the processing significantly. It is possible to avoid more of the loop prolog/epilog overhead if we fully unroll by kw * kh iterations so that the revised code has again single loop level, but doing that would hard-wire the code to a fixed convolution kernel size (if not kw x kh, at least the product kw * kh), which will have impact in code size if an application requires more than one convolution kernel size.

By moving the store inside the loop, we need to make two changes in the code. One is that the store should be predicated to execute periodically, once per kw * kh iterations. The other is that we need to change the output agen programming.

Store predication is accomplished through calculation of the predicate flag pred_store in the optimized code:

```
count_store = (count_store == niter_in-1) ? 0 : (count_store + 1);
pred_store = (count_store==0) ? (int)0xFFFFFFFF : 0;
```

The count_store is initialized to 1, and pred_store to 0, outside the loop. Inside the loop, count_store is modularly incremented, meaning it is incremented by 1 each time, until it reaches niter_in – 1, whereas it is reset to 0. pred_store flag is set -1 when count_store is zero and otherwise is set 0. With these statements, we implement a periodic pred_store with the following pattern:

```
0 0 … 0 1 0 0 … 1 …
```

Here, the period is niter_in, which is calculated in the initialization function to be kw * kh and stored to memory and restored in the run-time function. This achieves the objective of storing out once at the end of each period of kw * kh executions of the store.

These two lines of code involve many scalar operations, so it seems time-consuming to execute. To avoid predication becoming the bottleneck in compute loops, we have architected our predicate instructions to implement common periodic predication patterns, so these 2 lines of code map to just one predicate instruction, MODINC_NOTP. The “NOT” comes from the predication being derived negatively from the counter (true when counter is zero).

The output agen programming is also adjusted to account for placing the store inside the loop. An inner dimension n1 is inserted before the outer 2 dimensions iterating over output horizontally one vector width at a time, and vertically one row at a time. The inner dimension n1 is iterated kw * kh times with zero stepping, to implement a pattern that keeps the address static for kw * kh executions of the store before each advancement of the address.

The outer 2 dimensions of the output agen follow that of the scalar code, except that horizontally we are advancing by vector width at a time, due to vectorization, and vertically one row at a time.

The fact that we need an inner dimension for the output agen has to do with how predicated store is executed in the pipeline. In the processor pipeline, we need all memory operation to have address calculation early in the pipeline to deal with memory latency. Agen update is part of address calculation and thus is executed early and unconditionally even when there is predication on the store. The store predicate that controls whether a memory write is taking place is evaluated later in the pipeline, just in time to drive out to the VMEM interface along with data to be stored.

Similar predication is needed to implement accumulator initialization, which is also executed between loop levels. Through these 2 statements we implement another periodic predicate signal, pred_madd:

```
    count_madd = (count_madd == niter_in-1) ? 0 : (count_madd + 1);
    pred_madd  = (count_madd!=0) ? (int)0xFFFFFFFF : 0;
```

Both count_madd and pred_madd are initialized to 0 outside the loop. The pred_madd signal implemented has this pattern:

```
  0 1 1 … 1 0 1 1 … 1 …
```

Here, the period being also niter_in = kw * kh, matching the period of pred_store. pred_madd goes into the argument of vmaddhw, which is mapped to the predicated vector multiply-add instruction VMAddHHW_CA. When predicate is 0, the instruction does just multiplication, and when the predicate is non-zero, the instruction does multiply-add. Thus, the pred_madd pattern drives the MAC instruction to clear the accumulators for the first iteration in a period of kw * kh iterations.

These two lines of optimized code producing the pred_madd signal, although looking expensive, are mapped into just one predicate instruction, MODINCP.

Optimized code has loop body as follows:

```
    dvdata = dvshort_load_di(data_agen);
    coef = short_load(coef_agen);
    dvacc0 = vmaddhw(dvdata.lo, coef, dvacc0, VPU_ROUND_0, pred_madd);
    dvacc1 = vmaddhw(dvdata.hi, coef, dvacc1, VPU_ROUND_0, pred_madd);
    vstore_i(dvacc0, dvacc1, out_agen, pred_store);
```

Note the use of deinterleaving load, dvshort_load_di(), and interleaving store, vstore_i(). They are a matched pair to deal with data ordering when we use expanding MAC instructions, in this case VMAddHHW_CA, to produce outcome.

The expanding MAC instruction VMAddHHW_CA performs 17-bit x 17-bit multiplications (rather than 16-bit, so we can handle both signed 16-bit and unsigned 16-bit multiplication) and accumulates in 48-bit accumulators, to account for both product bit width and room for dynamic range growth in accumulating multiple products.

Here coefficients are loaded to a scalar variable/register, one at a time, and the scalar register is fed directly to the VMAddHHW_CA instruction and broadcast to all vector lanes performing the mutiply-add. In most vector math instructions, we support scalar source 2 operand optionally.

The profiling instruction report is as follows:

```
Function detail: filter_short_opt1 void_filter_short_opt1___P__sint___Pdvuint

  Low PC        :    232
  High PC        :    359
  Size in program memory:    128
  Cycle-count      :    597 ( 0.15%)
  Instruction-count   :    593 ( 0.23%)
  Instruction Coverage : 100.00%


 PC   Assembly                                                                  Exe-cnt  Cycs
 --- ------------------------------------------------------------------------- ------- -----
 232 ORI R0,#64,R3 || LDW *(R4+4),R2 || LDW *(R4+0),R4                            1  2
 235 AgenCfgLD *R5+=R3,A0                                                         1  1
 236 AgenCfgLD *(R5+64),A2                                                        1  2
 237 AgenCfgLD *(R5+0),A1                                                         1  1
 238 ORI R0,#0,R5 || SRAI R4,#3,R4                                                1  1
 240 ADDI R2,#-1,R2 || ADDI R4,#-1,R13                                            1  1
 242 ORI R0,#1,R4 || MOVSP R5,P2                                                  1  1
 244 MOVP P2,P9 || DVLDH_PDI *A0++,V12:V13                                        1  1
 246 DVLDH_PDI *A0++,V8:V9                                                        1  1
 247 LDH *A1++, R11 || DVLDH_PDI *A0++,V4:V5                                      1  1
 249 LDH *A1++, R9 || DVLDH_PDI *A0++,V0:V1                                       1  1
 251 LDH *A1++, R7 || DVLDH_PDI *A0++,V2:V3                                       1  1
 253 LDH *A1++, R3 || DVLDH_PDI *A0++,V6:V7                                       1  1
 255 LDH *A1++, R6                                                                1  1
 256 RPT R13,#LE_Fvoid_filter_short_opt1___P__sint___Pdvuint_89|| LDH *A1++, R8 ||      1  1
 258 MODINCP R2,R5,P10 || NOP || [P9] VMAddHHW_CA V13,R11,AC0:AC1||
     [P9] VMAddHHW_CA V12,R11,AC2:AC3 || DVLDH_PDI *A0++,V10:V11|| LDH *A1++, R10       1  1
 265 MODINC_NOTP R2,R4,P4||MODINCP R2,R5,P3||[P10] VMAddHHW_CA V9,R9,AC0:AC1||
     [P10] VMAddHHW_CA V8,R9,AC2:AC3||[P2] QVSTWH_PI AC2:AC3,AC0:AC1,*A2++||
     LDH *A1++, R12||DVLDH_PDI *A0++,V14:V15                                      1  1
 272 MODINCP R2,R5,P14 ||MODINC_NOTP R2,R4,P5||[P3] VMAddHHW_CA V4,R7,AC2:AC3||
     [P3] VMAddHHW_CA V5,R7,AC0:AC1||[P4] QVSTWH_PI AC2:AC3,AC0:AC1,*A2++||
     DVLDH_PDI *A0++,V12:V13||LDH *A1++,R11                                       71  71
 279 MODINCP R2,R5,P5 ||MODINC_NOTP R2,R4,P6||[P14] VMAddHHW_CA V0,R3,AC2:AC3||
     [P14] VMAddHHW_CA V1,R3,AC0:AC1||[P5] QVSTWH_PI AC2:AC3,AC0:AC1,*A2++||
     DVLDH_PDI *A0++,V8:V9||LDH *A1++,R9                                          71  71
 286 MODINCP R2,R5,P13 ||MODINC_NOTP R2,R4,P7||[P5] VMAddHHW_CA V2,R6,AC2:AC3||
```

```
    [P5] VMAddHHW_CA V3,R6,AC0:AC1||[P6] QVSTWH_PI AC2:AC3,AC0:AC1,*A2++||
    DVLDH_PDI *A0++,V4:V5||LDH *A1++,R7                                    71  71
293 MODINCP R2,R5,P7 ||MODINC_NOTP R2,R4,P15||[P13] VMAddHHW_CA V6,R8,AC2:AC3||
    [P13] VMAddHHW_CA V7,R8,AC0:AC1||[P7] QVSTWH_PI AC2:AC3,AC0:AC1,*A2++||
    DVLDH_PDI *A0++,V0:V1||LDH *A1++,R3                                    71  71
300 MODINCP R2,R5,P11 ||MODINC_NOTP R2,R4,P8||[P7] VMAddHHW_CA V10,R10,AC2:AC3||
    [P7] VMAddHHW_CA V11,R10,AC0:AC1||[P15] QVSTWH_PI AC2:AC3,AC0:AC1,*A2++||
    DVLDH_PDI *A0++,V2:V3||LDH *A1++,R6                                    71  71
307 MODINC_NOTP R2,R4,P12||MODINCP R2,R5,P9||[P11] VMAddHHW_CA V14,R12,AC2:AC3||
    [P11] VMAddHHW_CA V15,R12,AC0:AC1||[P8] QVSTWH_PI AC2:AC3,AC0:AC1,*A2++||
    DVLDH_PDI *A0++,V6:V7||LDH *A1++,8                                     71  71
314 MODINCP R2,R5,P10 ||MODINC_NOTP R2,R4,P2||[P9] VMAddHHW_CA V13,R11,AC0:AC1||
    [P9] VMAddHHW_CA V12,R11,AC2:AC3||[P12] QVSTWH_PI AC2:AC3,AC0:AC1,*A2++||
    DVLDH_PDI *A0++,V10:V11||LDH *A1++,R10                                 71  71
321 MODINCP R2,R5,P3 ||MODINC_NOTP R2,R4,P4||[P10] VMAddHHW_CA V8,R9,AC2:AC3||
    [P10] VMAddHHW_CA V9,R9,AC0:AC1||[P2] QVSTWH_PI AC2:AC3,AC0:AC1,*A2++||
    DVLDH_PDI *A0++,V14:V15||LDH *A1++,R12                                 71  71
328 MODINCP R2,R5,P14 || MODINC_NOTP R2,R4,P5|| [P3] VMAddHHW_CA V4,R7,AC2:AC3||
    [P3] VMAddHHW_CA V5,R7,AC0:AC1|| [P4] QVSTWH_PI AC2:AC3,AC0:AC1,*A2++          1   1
333 MODINC_NOTP R2,R4,P6|| MODINCP R2,R5,P5|| [P14] VMAddHHW_CA V1,R3,AC0:AC1||
    [P14] VMAddHHW_CA V0,R3,AC2:AC3|| [P5] QVSTWH_PI AC2:AC3,AC0:AC1,*A2++        1   1
338 MODINC_NOTP R2,R4,P7|| MODINCP R2,R5,P13|| [P5] VMAddHHW_CA V3,R6,AC0:AC1||
    [P5] VMAddHHW_CA V2,R6,AC2:AC3|| [P6] QVSTWH_PI AC2:AC3,AC0:AC1,*A2++         1   1
343 MODINC_NOTP R2,R4,P15|| MODINCP R2,R5,P7|| [P13] VMAddHHW_CA V7,R8,AC0:AC1||
    [P13] VMAddHHW_CA V6,R8,AC2:AC3|| [P7] QVSTWH_PI AC2:AC3,AC0:AC1,*A2++        1   1
348 MODINC_NOTP R2,R4,P8|| MODINCP R2,R5,P11|| [P7] VMAddHHW_CA V11,R10,AC0:AC1||
    [P7] VMAddHHW_CA V10,R10,AC2:AC3|| [P15] QVSTWH_PI AC2:AC3,AC0:AC1,*A2++      1   1
353 JR R15                                                                 1   1
354 MODINC_NOTP R2,R4,P12|| [P11] VMAddHHW_CA V14,R12,AC2:AC3||
    [P11] VMAddHHW_CA V15,R12,AC0:AC1|| [P8] QVSTWH_PI AC2:AC3,AC0:AC1,*A2++      1   1
358 NOP || [P12] QVSTWH_PI AC2:AC3,AC0:AC1,*A2++                           1   3
```

The loop body is scheduled optimally into 8 cycles, packing 2 scalar predicate instructions, 2 vector math instructions, and 3 memory operations (2 loads and 1 store) in every execution packet.

The loop prolog starting well ahead of the loop body, and the loop epilog ending well after the loop body, as the loop is unrolled 8 times and software pipelined.

The optimized function completes the same 64 wide x 32 tall output convolution task in 597 cycles. There is an almost 480x speedup compared to 285,663 cycles by the scalar code.

Next, we will see how we leverage a denser MAC instruction, VFilt4x2HHW_CA, to achieve further speedup.

8.4.3 Optimization 2: Leveraging Denser MAC Instruction

In VPU instruction set, besides vector multiply-add, we have denser MAC instructions. For 16-bit data, we have:

- VDotP2HHW_CA 2-term dot-product
- VDotP4 _CA 4-term dot-product
- VDotP4x2_CA 2 sets of 4-term dot-product
- VFilt4HHW_CA 4-tap filtering
- VFilt4x2HHW_CA 2 sets of 4-tap filtering

Of these, VDotP* instructions are suitable for dot-product. VFilt4HHW_CA delivers 4 MACs per halfword lane is very useful for 2D convolution.

We do have VFilt4x2HHW_CA that delivers 8 MACs per halfword lane so it has 2x raw MAC throughput of VFilt4HHW_CA. However, VFilt4x2HHW is more suitable for CNN or filter banks, where multiple output planes are produced. It is possible to leverage it for 2D convolution where a single output plane is produced, but there is some preprocessing and postprocessing steps involved to reformat data and output, and to avoid spending VPU cycles on pre- and post-processing, we will have to configure DMA to perform the reformatting while transferring data in and out of VMEM, so construction of the test case is much more involved.

VFilt4HHW_CA performs horizontal 4-tap filtering on 16 lanes of 16-bit data/coefficients and accumulates sum of products in 16 lanes of 48-bit accumulators. To leverage VFilt4HHW_CA, we need to zero-pad the coefficients horizontally into multiple of 4 kernel width.

Compared to VMAddHHW_CA that performs one MAC per halfword lane, VFilt4HHW_CA performs 4 MACs per halfword lane, so we need to feed 4 data points and 4 coefficient points to each lane to feed the MACs. The way we accomplish this, on the data feed, is to leverage the sliding-window dependency and provide 2 single vectors of data loaded with overlapping data. On the coefficient feed, we take advantage of the fact that in convolution we use the same filter kernel for all output data points to share coefficients within each group of lanes.

From instruction details in VFILT4_CA, we see that the intrinsic for VFilt4HHW_CA:

```
dvshortx vfilt4_bbh(vcharx src1a, vcharx src1b, vcharx src2, dvshortx src3dst, int pred);
```

This requires that data, coefficients, accumulators within each group of 4 lanes being laid out as:

src1a	D[0]	D[1]	D[2]	D[3]
src1b	D[4]	D[5]	D[6]	D[7]
src2	C[0]	C[1]	C[2]	C[3]
src3dst.lo	ACC[0]		ACC[2]	

src3dst.hi	ACC[1]	ACC[3]

In each group of 4 lanes, the instructions are carried out:

ACC[0] += D[0] * C[0] + D[1] * C[1] + D[2] * C[2] + D[3] * C[3];
ACC[1] += D[1] * C[0] + D[2] * C[1] + D[3] * C[2] + D[4] * C[3];
ACC[2] += D[2] * C[0] + D[3] * C[1] + D[4] * C[2] + D[5] * C[3];
ACC[3] += D[3] * C[0] + D[4] * C[1] + D[5] * C[2] + D[6] * C[3];

Like the other examples, we want to double-up vector math to take advantage of the double vector load/store throughput. Thus to feed 2 VFilt4HHW_CA instructions placed on both vector slots of the same execution packet, we would load from the data array 2 double vectors with 4 element offset for data, and either use VLDPerm to load from the coefficient array 4 elements and create the 4-term repeating pattern in the coefficient single vector, or we reformat the coefficients outside the compute kernel function to create this pattern.

However, if we use 2 loads for data, 1 load for coefficient, to feed the MACs, and together with predicated store to write outcome to VMEM when all product terms are accumulated, we spend 4 memory operations to feed 2 vector math operations, and would not be able to execute optimally as it would become memory-bound. To reduce the memory-to-vector-math ratio, we reuse data between 2 output rows; essentially working on 2 double vectors worth of output at a time, and the 2 double vectors are mapped to even and odd rows of the output array. By working on 2 rows of output at a time, we will also to zero-pad coefficients vertically and perform the 3x3 FIR filtering as 4x4 FIR filtering.

Derivation for number of iterations and step parameters for the agen is similar to the other examples, so here we shall just show program listings and profiling instruction report.

The filter_16b_filt4_init() function:

```
void filter_16b_filt4_init(short *data, short *coef, short* restrict out,
             int kw, int kh, int qbits, int blkw, int blkh,
             int lofst_data, int lofst_out,
             AGEN_PTR * agen_cfg, int * niter, int * niter_in)
{
  int  vecw = chess_elementsof(dvshort);
  short niter1 = (kw+3)/4;
  short niter2 = kh+1;
  short niter3 = blkw/vecw;
  short niter4 = blkh/2;
  agen_wrapper_t wrapper0, wrapper1, wrapper2;

  *niter  = niter1 * niter2 * niter3 * niter4;
  *niter_in = niter1 * niter2;
```

```
  agen a0 = init((dvshort*)data);
  wrapper0.size = sizeof(short);
  wrapper0.n1 = 2;
  wrapper0.n2 = niter1;
  wrapper0.n3 = niter2;
  wrapper0.n4 = niter3;
  wrapper0.n5 = niter4;
  wrapper0.s1 = 4;
  wrapper0.s2 = 4;
  wrapper0.s3 = lofst_data;
  wrapper0.s4 = vecw;
  wrapper0.s5 = 2 * lofst_data;
  INIT_AGEN5(a0, wrapper0);

  agen a1 = init((vshort*) coef);
  wrapper1.size = sizeof(vshort);
  wrapper1.n1 = 2 * niter1 * niter2;
  wrapper1.n2 = niter3 * niter4;
  wrapper1.s1 = 1;
  wrapper1.s2 = 0;
  INIT_AGEN2(a1, wrapper1);

  agen a2 = init((dvshort* restrict ) out);
  wrapper2.size = sizeof(short);
  wrapper2.n1 = niter1 * niter2;
  wrapper2.n2 = niter3;
  wrapper2.n3 = niter4;
  wrapper2.s1 = 0;
  wrapper2.s2 = vecw;
  wrapper2.s3 = 2 * lofst_out;
  INIT_AGEN3(a2, wrapper2);
  a2.round = qbits;

  agen a3 = a2;
  a3.a = (vint * restrict) (out + lofst_out);

  *agen_cfg++ = a0.get_cfg();
  *agen_cfg++ = a1.get_cfg();
  *agen_cfg++ = a2.get_cfg();
  *agen_cfg++ = a3.get_cfg();
}
```

The filter_16b_filter_exec() function, noting how the pair of double vectors for data are reused for 2 sets of accumulators.

```
void filter_16b_filt4_exec(AGEN_PTR * agen_cfg, int niter, int niter_in)
```

```
{
  dvshortx vdata0, vdata1;
  vshortx  vcoef0, vcoef1;
  dvintx vacc0, vacc1;
  dvintx vacc2, vacc3;
  int  count_madd = 0;
  int  count_store = 1;
  int  pred_madd  = 0;
  int  pred_store = 0;
  agen a0 = a0.expand_cfg(*agen_cfg++);
  agen a1 = a1.expand_cfg(*agen_cfg++);
  agen a2 = a2.expand_cfg(*agen_cfg++);
  agen a3 = a3.expand_cfg(*agen_cfg++);

  chess_separator_scheduler();

  for (int i=0; i<niter; i++) chess_prepare_for_pipelining
            chess_unroll_loop(4) chess_loop_range(12,)
  {
    vdata0 = dvshort_load(a0);
    vdata1 = dvshort_load(a0);
    vcoef0 = vshort_load(a1);
    vcoef1 = vshort_load(a1);

    vacc0 = vfilt4_hhw(vdata0.lo, vdata1.lo, vcoef0, vacc0, pred_madd);
    vacc1 = vfilt4_hhw(vdata0.hi, vdata1.hi, vcoef0, vacc1, pred_madd);

    vacc2 = vfilt4_hhw(vdata0.lo, vdata1.lo, vcoef1, vacc2, pred_madd);
    vacc3 = vfilt4_hhw(vdata0.hi, vdata1.hi, vcoef1, vacc3, pred_madd);

    vstore_i2(vacc0, vacc1, a2, pred_store);
    vstore_i2(vacc2, vacc3, a3, pred_store);

  }
}
```

The profiling instruction report of the _exec function is as follows.

```
Function detail: filter_16b_filt4_exec void_filter_16b_filt4_exec___Pdvuint___sint___sint

Low PC        :    168
High PC        :    279
Size in program memory:    112
Cycle-count      :    279 ( 5.41%)
Instruction-count   :    276 ( 9.98%)
Instruction Coverage :    100.00%

PC   Assembly                                                                  Exe-cnt Cycs
--- ------------------------------------------------------------------------- ------- ----
168 ORI R0,#64,R7                                                                1    2
169 AgenCfgLD *R4+=R7,A0                                                         1    1
```

```
170 AgenCfgLD *R4+=R7,A1                                                                  1    1
171 AgenCfgLD *(R4+64),A2                                                                1    1
172 AgenCfgLD *(R4+0),A3                                                                 1    1
173 GPO_SETLI #1                                                                         1    1
174 ORI R0,#0,R7  || SRAI R5,#2,R5                                                       1    1
176 ADDI R5,#-1,R4 || DVLDH_P *A0++,V10:V11                                              1    1
178 VLDH_P *A1++,W6 || DVLDH_P *A0++,V12:V13                                             1    1
180 VLDH_P *A1++,W4 || DVLDH_P *A0++,V2:V3                                               1    1
182 VLDH_P *A1++,W2 || DVLDH_P *A0++,V4:V5                                               1    1
184 VLDH_P *A1++,W1 || DVLDH_P *A0++,V0:V1                                               1    1
186 VLDH_P *A1++,W0 || DVLDH_P *A0++,V6:V7                                               1    1
188 RPT R4,#LE_Fvoid_filter_16b_filt4_exec || VLDH_P *A1++,W3                            1    1
190 MOVSP R7,P6 || ADDI R6,#-1,R6 || DVLDH_P *A0++,V8:V9|| VLDH_P *A1++,W5               1    1
194 MOVP P6,P2 || ORI R0,#1,R5 || [P6] VFilt4HHW_CA V11,V13,W6,AC0:AC1||
    [P6] VFilt4HHW_CA V10,V12,W6,AC2:AC3 || DVLDH_P *A0++,V14:V15|| VLDH_P *A1++,W7      1    1
200 MODINCP R6,R7,P7 || [P6] VFilt4HHW_CA V11,V13,W4,AC4:AC5||
    [P6] VFilt4HHW_CA V10,V12,W4,AC6:AC7 || [P2] QVSTWH_PI2 AC2:AC3,AC0:AC1,*A3++||
    DVLDH_P *A0++,V10:V11|| VLDH_P *A1++,W6                                             31   31
206 MODINC_NOTP R6,R5,P4|| [P7] VFilt4HHW_CA V3,V5,W2,AC0:AC1||
    [P7] VFilt4HHW_CA V2,V4,W2,AC2:AC3 || [P2] QVSTWH_PI2 AC6:AC7,AC4:AC5,*A2++||
    DVLDH_P *A0++,V12:V13|| VLDH_P *A1++,W4                                             31   31
212 MODINCP R6,R7,P8 || [P7] VFilt4HHW_CA V3,V5,W1,AC4:AC5||
    [P7] VFilt4HHW_CA V2,V4,W1,AC6:AC7 || [P4] QVSTWH_PI2 AC2:AC3,AC0:AC1,*A3++||
    DVLDH_P *A0++,V2:V3|| VLDH_P *A1++,W2                                               31   31
218 MODINC_NOTP R6,R5,P5|| [P8] VFilt4HHW_CA V1,V7,W0,AC0:AC1||
    [P8] VFilt4HHW_CA V0,V6,W0,AC2:AC3 || [P4] QVSTWH_PI2 AC6:AC7,AC4:AC5,*A2++||
    DVLDH_P *A0++,V4:V5|| VLDH_P *A1++,W1                                               31   31
224 MODINCP R6,R7,P9 || MODINC_NOTP R6,R5,P3|| [P8]VFilt4HHW_CA V1,V7,W3,AC4:AC5||
    [P8] VFilt4HHW_CA V0,V6,W3,AC6:AC7 || [P5] QVSTWH_PI2 AC2:AC3,AC0:AC1,*A3++||
    DVLDH_P *A0++,V0:V1|| VLDH_P *A1++,W0                                               31   31
231 MODINC_NOTP R6,R5,P2|| MODINCP R6,R7,P6|| [P9]VFilt4HHW_CA V9,V15,W5,AC0:AC1||
    [P9] VFilt4HHW_CA V8,V14,W5,AC2:AC3 || [P5] QVSTWH_PI2 AC6:AC7,AC4:AC5,*A2++||
    DVLDH_P *A0++,V6:V7|| VLDH_P *A1++,W3                                               31   31
238 [P9] VFilt4HHW_CA V9,V15,W7,AC4:AC5|| [P9] VFilt4HHW_CA V8,V14,W7,AC6:AC7 ||
    [P3] QVSTWH_PI2 AC2:AC3,AC0:AC1,*A3++|| DVLDH_P *A0++,V8:V9|| VLDH_P *A1++,W5       31   31
243 [P6] VFilt4HHW_CA V11,V13,W6,AC0:AC1|| [P6] VFilt4HHW_CA V10,V12,W6,AC2:AC3||
    [P3] QVSTWH_PI2 AC6:AC7,AC4:AC5,*A2++|| DVLDH_P *A0++,V14:V15|| VLDH_P *A1++,W7     31   31
248 MODINCP R6,R7,P7|| MODINC_NOTP R6,R5,P4||[P6]VFilt4HHW_CA V11,V13,W4,AC4:AC5||
    [P6]VFilt4HHW_CA V10,V12,W4,AC6:AC7 || [P2] QVSTWH_PI2 AC2:AC3,AC0:AC1,*A3++        1    1
253 MODINC_NOTP R6,R5,P5|| MODINCP R6,R7,P8|| [P7] VFilt4HHW_CA V2,V4,W1,AC6:AC7||
    [P7]VFilt4HHW_CA V3,V5,W1,AC4:AC5 || [P2] QVSTWH_PI2 AC6:AC7,AC4:AC5,*A2++          1    1
258 MODINC_NOTP R6,R5,P3|| MODINCP R6,R7,P9|| [P7] VFilt4HHW_CA V2,V4,W2,AC2:AC3||
    [P7]VFilt4HHW_CA V3,V5,W2,AC0:AC1 || [P4] QVSTWH_PI2 AC6:AC7,AC4:AC5,*A2++          1    1
263 [P8] VFilt4HHW_CA V0,V6,W3,AC6:AC7|| [P8] VFilt4HHW_CA V1,V7,W3,AC4:AC5||
    [P4] QVSTWH_PI2 AC2:AC3,AC0:AC1,*A3++                                               1    1
266 [P8] VFilt4HHW_CA V0,V6,W0,AC2:AC3|| [P8] VFilt4HHW_CA V1,V7,W0,AC0:AC1||
    [P5] QVSTWH_PI2 AC6:AC7,AC4:AC5,*A2++                                               1    1
269 [P9] VFilt4HHW_CA V8,V14,W7,AC6:AC7|| [P9] VFilt4HHW_CA V9,V15,W7,AC4:AC5||
    [P5] QVSTWH_PI2 AC2:AC3,AC0:AC1,*A3++                                               1    1
```

```
272 [P9] VFilt4HHW_CA V8,V14,W5,AC2:AC3|| [P9] VFilt4HHW_CA V9,V15,W5,AC0:AC1||
    [P3] QVSTWH_PI2 AC6:AC7,AC4:AC5,*A2++                                    1    1
275 [P3] QVSTWH_PI2 AC2:AC3,AC0:AC1,*A3++                                    1    1
276 GPO_CLRLI #1                                                             1    1
277 JR R15                                                                   1    1
278 NOP                                                                      1    1
279 NOP                                                                      1    3
```

The vector slots are fully utilized in the loop body, executing a pair of MAC instructions (VFilt4HHW) in every execution packet. Also, the 3 memory slots are also packed with one double vector load for data, one single vector load for coefficients, and one quad vector store. We could have loaded coefficients with a double vector load and left the memory slots less utilized. The key is that, if possible, we want to saturate the vector math slots to achieve the best performance. If to achieve full vector math utilization, we need to saturate memory slots as well, that's OK; however, if possible, if we can achieve full vector math utilization with less memory slots utilization, we would achieve better power efficiency as well.

The loop body portion executes for 8 * 31 = 248 cycles, compared with 8*71 = 568 cycles in the VMAdd implementation. The 4x MAC density is diluted somewhat from implementing 3x3 FIR filter as 4x4 FIR; 4 * 9 / 16 = 2.25x speedup. For larger FIR kernel, the diluting would not be as bad.

8.4.4 Further Optimization for Power

The proceeding programming examples are about techniques in performance optimization. While reducing processing time often leads to reduction in the energy exerted to implement specific functions, there are additional techniques one can follow to further optimize for power.

VPU has load data cache features that can help reduce power when used correctly. Load data cache reduces power consumption by bypassing VMEM superbank read for the memory banks that are read with the same row address. In a 2D convolution, both data and coefficient read may be implemented to have such address patterns and can leverage load data cache feature. In the 2D convolution optimization 1 and optimization 2 examples, we already have data read address pattern that works for load data cache.

Optimization 1 data agen initialization:

```
wrapper0.size = sizeof(short);
wrapper.n1 = kw;
wrapper.n2 = kh;
...
wrapper.s1 = 1;
wrapper.s2 = lofst_data;
```

For data read in optimization 1 agen innermost i1 loop, we move the read pointer 1 pixel at a time for kw reads from the agen, and each read is a double vector read. Enabling load data cache for data agen can save (kw-1)*31 out of every kw*32 memory bank read transactions for data read. In the next i2 loop, we move data pointer by one row of data,

which is usually greater than 64 bytes (as we vectorize processing we should process minimally the vector width).

Optimization 2 data agen initialization:

```
wrapper0.size = sizeof(short);
wrapper0.n1 = 2;
wrapper0.n2 = niter1; // (kw+3)/4
wrapper0.n3 = niter2; // kh+1
...
wrapper0.s1 = 4;
wrapper0.s2 = 4;
wrapper0.s3 = lofst_data;
```

For data read in optimization 2 agen innermost i1 loop, we move the read pointer 4 pixels at a time for 2 reads from the agen, and again each read is a double vector read. Enabling load data cache can save 1*28 out of every 2*32 memory bank read transactions. In case kw > 4, the pointer moves by 4 pixels, and there is further power saving.

For coefficients read in optimization 1, coefficients are read one element at a time into a scalar register. Scalar reads are not cached (see Load Data Cache), so optimization 1 coefficient read does not work for load cache.

Optimization 2 coefficient agen initialization:

```
wrapper1.size = sizeof(vshort);
wrapper1.n1 = 2 * niter1 * niter2;
wrapper1.n2 = niter3 * niter4;
wrapper1.s1 = 1;
wrapper1.s2 = 0;
```

For coefficient read in optimization 2, coefficients are reformatted outside the filtering loop so that in the filtering loop coefficients are read one single vector at a time (vshort) without repetition, so the pattern does not work for load cache.

It is possible to leverage load cache, but we will have to change the coefficient reformatting loop. To simplify the filtering loop in the example, we have the coefficient reformatting loop create the 2-output-rows-at-a-time zero-padded coefficient array AND repeated 4 times, as there are 4 4-lane groups in a single vector of halfwords. If we revise the coefficient reformatting loop to not repeat the coefficient data 4 times and revise the filtering loop to use VLDPerm to load and permute the coefficients with appropriate permutation pattern, we can leverage load data cache for coefficient reads as well, and further reduce power consumption for 2D convolution.

For example, if we have 3x3 filtering (kw = kh = 3), current optimization 2 code coefficient reformatting loop would produce:

```
coef[] = {C0, C1, C2, 0, C0, C1, C2, 0, C0, C1, C2, 0, C0, C1, C2, 0, // out 0 row 0
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,             // out 1 row 0
     C3, C4, C5, 0, C3, C4, C5, 0, C3, C4, C5, 0, C3, C4, C5, 0, // out 0 row 1
     C0, C1, C2, 0, C0, C1, C2, 0, C0, C1, C2, 0, C0, C1, C2, 0, // out 1 row 1
     C6, C7, C8, 0, C6, C7, C8, 0, C6, C7, C8, 0, C6, C7, C8, 0, // out 0 row 2
```

```
     C3, C4, C5, 0, C3, C4, C5, 0, C3, C4, C5, 0, C3, C4, C5, 0, // out 1 row 2
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,             // out 0 row 3
     C6, C7, C8, 0, C6, C7, C8, 0, C6, C7, C8, 0, C6, C7, C8, 0};// out 1 row 3
```

To leverage load data cache for coefficients, we would skip the 4-time repetition, so that the reformatted coefficients would be:

```
coef[] = {C0, C1, C2, 0, // out 0 row 0
     0, 0, 0, 0,    // out 1 row 0
     C3, C4, C5, 0, // out 0 row 1
     C0, C1, C2, 0, // out 1 row 1
     C6, C7, C8, 0, // out 0 row 2
     C3, C4, C5, 0, // out 1 row 2
     0, 0, 0, 0,    // out 0 row 3
     C6, C7, C8, 0};// out 1 row 3
```

Load data cache is enabled on a per VMEM superbank basis, so to have effective caching, we would need data and coefficient arrays be allocated in different VMEM superbanks. Otherwise, the load cache would be threshing from data, and coefficient reads giving different row addresses to the same memory banks, and the cache would have poor hit rate. When load data cache is enabled and when the read data pattern has few repeated read to the memory banks, we may end up consuming higher power, from additional activity in recording/matching the memory bank row addresses.

8.5 Interpolated 2D Lookup

In computer vision, we sometimes need to perform interpolated 2D lookup, typically to resize an image, to undistort an image, or to warp an image patch for object tracking. Most common interpolation scheme is bilinear interpolation. VPU instruction set includes instructions that accelerate interpolated 2D lookup.

In this example, we shall look at scalar/reference code, VPU optimized code, and code leveraging the DLUT (decoupled lookup table unit) to perform interpolated 2D lookup.

Here we skip the profiling instruction report.

8.5.1 Scalar Code

The following is the scalar/reference function performing interpolated 2D lookup:

```
//*********************************************************************
// interpolate_lookup2d_ref
// 2D table tblw wide
// index X/Y interleaved
//*********************************************************************
void interpolated_lookup2d_ref(int tblw, int len_out, int frac_bits,
                short * tbl, short * idx, short * out)
{
  int x, y, int_x, int_y, frac_x, frac_y;
  int lu_idx, entry0, entry1, entry2, entry3;
```

```
  int out01, out23, out0123;
  int frac_mask = (1 << frac_bits) - 1;
  int rnd_add  = 1 << (frac_bits - 1);

  for (int i = 0; i < len_out; i++) {
    x = *idx++;
    y = *idx++;
    int_x = x >> frac_bits;
    int_y = y >> frac_bits;
    frac_x = x & frac_mask;
    frac_y = y & frac_mask;
    lu_idx = int_y * tblw + int_x;
    entry0 = tbl[lu_idx];
    entry1 = tbl[lu_idx + 1];
    entry2 = tbl[lu_idx + tblw];
    entry3 = tbl[lu_idx + tblw + 1];
    out01 = entry0 + (((entry1 - entry0) * frac_x + rnd_add) >> frac_bits);
    out23 = entry2 + (((entry3 - entry2) * frac_x + rnd_add) >> frac_bits);
    out0123 = out01 + (((out23 - out01) * frac_y + rnd_add) >> frac_bits);
    *out++ = out0123;
  }
}
```

The index data is X/Y interleaved. Each element (carrying either X or Y) is a fixed-point number with number of fraction bits being frac_bits.

The reference code reads X & Y indices, separates out integer and fraction components, calculates a linear index using the integer X & Y components, performs the 2x2 lookup, then uses the fraction X & Y components to perform bilinear interpolation to produce one output value. Notice how we perform horizontal interpolation to blend entry0 with entry1 to produce out01, and entry2 with entry3 to produce out23. Then we perform vertical interpolation to blend out01 and out23 to produce the final output.

This scalar/reference function takes 63,504 cycles to produce 2048 outputs interpolating from a 66 x 34 image patch, averaging 31 cycles per output.

8.5.2 VPU Parallel Lookup

VPU has parallel lookup instructions to perform 2x2 lookup, up to a parallelism of 8. To leverage such instructions, we will need to replicate the original table containing the image patch 8 times.

Given the replicated lookup table, VPU optimized function performing interpolated 2D lookup is as follows:

```
void interpolated_lookup2d_opt(int tblw, int len_out, int frac_bits,
                short * tbl, short * idx, short * restrict out)
{
  dvshortx vidx;
  dvshortx bitpos;
```

```
  bitpos.lo = replicateh(frac_bits);
  bitpos.hi = replicateh(frac_bits);
  dvshortx xy_int, xy_frac, xy_frac_norm;
  vshortx lu_idx, lu_idx_even, lu_idx_odd;
  dvshortx entry_even, entry_odd;
  vshortx out01, out23, out0123;
  int lsh_bits = 15 - frac_bits;
  int lp_k = tblw/4;
  short even_arr[] = {0, 2, 4, 6, 8, 10, 12, 14, -1, -1, -1, -1, -1, -1, -1, -1};
  short odd_arr[] = {1, 3, 5, 7, 9, 11, 13, 15, -1, -1, -1, -1, -1, -1, -1, -1};
  vshortx pat_even = zero_extend(*((vushort *) even_arr));
  vshortx pat_odd = zero_extend(*((vushort *) odd_arr));
  int niter = len_out/16;
  agen_A aidx = init_A(idx);
  aidx.n1 = niter;
  aidx.mod1 = sizeof(dvshort);
  agen_C aout = init_C(out); // write 16 at a time
  aout.n1 = niter;
  aout.mod1 = sizeof(vshort);
  short chess_storage(DMb%64:chess_segment(B)) * tbl_ptr =
           (short chess_storage(DMb%64:chess_segment(B)) *) tbl;
#define UNROLL1 7
  // round up to multiple of unrolling factor
  niter = ((niter + UNROLL1 - 1)/UNROLL1) * UNROLL1;

  for (int i = 0; i < niter; i++) chess_prepare_for_pipelining
          chess_unroll_loop(UNROLL1) chess_loop_range(3 * UNROLL1,)
  {
    vidx = dvshort_load_di(aidx);                 // x/y intrlv -> lo/hi
    dvsplitbits(vidx, bitpos, xy_int, xy_frac);   // lo=x, hi=y
    lu_idx = vmaddh(xy_int.hi, tblw, xy_int.lo, (vpu_primitive::u3) 0);
                                                  // int_y * tblw + int_x
    xy_frac_norm = xy_frac << lsh_bits;
    lu_idx_even = vpermute(lu_idx, pat_even);     // 0, 2, 4, ..., 14
    lu_idx_odd = vpermute(lu_idx, pat_odd);       // 1, 3, 5, ..., 15
    entry_even = vlookup_2x2pt_8h(tbl_ptr, lu_idx_even, lp_k);
    entry_odd  = vlookup_2x2pt_8h(tbl_ptr, lu_idx_odd, lp_k);
    out01    = vhblend_i(entry_even.lo, entry_odd.lo, xy_frac_norm.lo);
    out23    = vhblend_i(entry_even.hi, entry_odd.hi, xy_frac_norm.lo);
    out0123   = vblend(out01, out23, xy_frac_norm.hi);
    vstore(out0123, aout);
  }
}
```

The use of vpermute() to reorganize elements in lu_idx vector to separate even and odd data points. Unfortunately, this is needed to work with vlookup_2x2pt_8h, as only 8 data points are needed in each index vector of vshortx type (which holds 16 lanes). After the 8 even/odd data points, rest of the lanes are zero-filled, by padding -1 in the even_arr[] and odd_arr[] arrays.

The loop has the following vector math operations:

- 2x vsplitbits
- vmaddh
- 2x vsla
- 2x vpermute
- 2x vhblend_i
- vblend

That's 10 operations, so we would say the SOL (speed of light/ideal) performance is 5 cycles per iteration.

Unrolling factors from 4 to 8 are tried, and with 7x loop unrolling, we found the best performance at 39 cycles per iteration, or 5. 57 cycles per original iteration.

The compiler often, but not always, achieves SOL performance. The loop body has relatively long latency due to cascading of long math and lookup latencies (vmaddh → vpermute → vlookup) and would need to unroll more to allow compiler to pack the vector math slots but unrolling more leads to more challenging register allocation.

The optimized function executes for 805 cycles for the same test configuration (2048 outputs, 66 x 34 image patch). This translates to 0.393 cycle per output, and roughly 78.9x speedup over scalar/reference function.

8.5.3 VPU Parallel Lookup in Two Loops

One optimization strategy we can try when we have a long string of math operations in the loop is to break it into 2 loops. For the VPU parallel lookup code 1 in the previous section, there is another advantage in breaking up the loop into two, in that the permutation operation in vector math we can get for free (of vector math operations) by leveraging the load with permute instruction. The resulting code is as follows:

```
void interpolated_lookup2d_opt2(int tblw, int len_out, int frac_bits,
              short * tbl, short * idx, short * temp_buf_idx,
              short * temp_buf_frac, short * out)
{
  dvshortx vidx;
  dvshortx bitpos;
  bitpos.lo = replicateh(frac_bits);
  bitpos.hi = replicateh(frac_bits);
  dvshortx xy_int, xy_frac, xy_frac_norm;
  vshortx lu_idx;
  dvshortx dv_lu_idx;
  dvshortx entry_even, entry_odd;
  vshortx out01, out23, out0123;
  int lp_k = tblw/4;
  char perm_pat_arr[] = {0, 2, 4, 6, 8, 10, 12, 14, -1,-1,-1,-1,-1,-1,-1,-1,
                         1, 3, 5, 7, 9, 11, 13, 15, -1,-1,-1,-1,-1,-1,-1,-1};
  vcharx perm_pat = sign_extend(*((vchar *) perm_pat_arr));
  short chess_storage(DMb%64:chess_segment(B)) * tbl_ptr =
```

```
          (short chess_storage(DMb%64:chess_segment(B)) *) tbl;

  int niter = len_out/16;
  agen_A aidx = init_A(idx);                // read 16 X + 16 Y at a time
  aidx.n1 = niter;
  aidx.mod1 = sizeof(dvshort);
  agen_C atemp_idxw = init_C(temp_buf_idx); // write 16 indices at a time
  atemp_idxw.n1 = niter;
  atemp_idxw.mod1 = sizeof(vshort);
  agen_C atemp_fracw = init_C(temp_buf_frac);// write 16 dX + 16 dY at a time
  atemp_fracw.n1 = niter;
  atemp_fracw.mod1 = sizeof(dvshort);
  agen_C atemp_idxr = init_C(temp_buf_idx); // read 16 indices at a time
  atemp_idxr.n1 = niter;
  atemp_idxr.mod1 = sizeof(vshort);
  agen_C atemp_fracr = init_C(temp_buf_frac);// read 16 dX + 16 dY at a time
  atemp_fracr.n1 = niter;
  atemp_fracr.mod1 = sizeof(dvshort);
  agen_C aout = init_C(out);                 // write 16 outputs at a time
  aout.n1 = niter;
  aout.mod1 = sizeof(vshort);
  chess_separator_scheduler();

#define UNROLL2 6
#define UNROLL3 5
  // round up to multiple of unrolling factor
  int niter1 = ((niter + UNROLL2 - 1)/UNROLL2) * UNROLL2;
  int niter2 = ((niter + UNROLL3 - 1)/UNROLL3) * UNROLL3;

  for (int i = 0; i < niter1; i++) chess_prepare_for_pipelining
          chess_unroll_loop(UNROLL2) chess_loop_range(3 * UNROLL2,)
  {
    vidx = dvshort_load_di(aidx);               // x/y intrlv -> lo/hi
    dvsplitbits(vidx, bitpos, xy_int, xy_frac); // lo=x, hi=y
    xy_frac_norm = xy_frac << (15 - frac_bits); // lo=x, hi=y
    lu_idx = vmaddh(xy_int.hi, tblw, xy_int.lo, (vpu_primitive::u3) 0);
                                                // int_y * tblw + int_x
    vstore(xy_frac_norm, atemp_fracw);          // 16 dX + 16 dY
    vstore(lu_idx, atemp_idxw);                 // 16 IDX
  }

  chess_separator_scheduler();

  for (int i = 0; i < niter2; i++) chess_prepare_for_pipelining
          chess_unroll_loop(UNROLL3) chess_loop_range(3 * UNROLL3,)
  {
    dv_lu_idx = dvshort_load_perm(atemp_idxr, perm_pat); // 8 even + 8 odd
    xy_frac_norm = dvshort_load(atemp_fracr);            // 16 dX + 16 dY
    entry_even = vlookup_2x2pt_8h(tbl_ptr, dv_lu_idx.lo, lp_k);
```

```
    entry_odd = vlookup_2x2pt_8h(tbl_ptr, dv_lu_idx.hi, lp_k);
    out01  = vhblend_i(entry_even.lo, entry_odd.lo, xy_frac_norm.lo);
    out23  = vhblend_i(entry_even.hi, entry_odd.hi, xy_frac_norm.lo);
    out0123 = vblend(out01, out23, xy_frac_norm.hi);
    vstore(out0123, aout);
  }
}
```

The two inner loops the following vector math operations, respectively:

- 2x vsplitbits
- vmaddh
- 2x vsla

- 2x vhblend_i
- vblend

After adding loads and stores to both loops to make them work, the resulting first loop is still vector math-bound at SOL of 2.5 cycles per iteration. The second loop becomes M0-slot and lookup-bound, at SOL of 2 cycles per iteration.

Again, unrolling factors from 4 to 8 are tried, and with 6x loop unrolling, we found the best performance at 17 cycles per iteration, or 2.83 cycles per original iteration. The second loop is 5x unrolled at 10 cycles per iteration, or 2 cycles per original iteration, meeting SOL.

The optimized function executes for 748 cycles for the same test configuration (2048 outputs, 66 x 34 image patch). The difference in inner-loop performance, 2.83 + 2 = 4.83 cycles per iteration versus 5.57 cycles per iteration, can lead to a bigger gap in cycle count if there is a bigger workload.

In breaking up the long sequence math into two loops, we achieve slightly faster compute function, but we also incur greater power consumption by having more VMEM read/write for the same application. The two-loop solution is also likely to have larger code size, which can lead to higher I-cache misses in an application. There are pros and cons in this implementation.

8.5.4 Leveraging DLUT

Interpolated 2D lookup is one of the operation modes supported by DLUT. To leverage DLUT, we need to leverage Sampler APIs in PVA SDK. For this particular problem, we configure the DLUT task with:

```
#include <cupva_device.h>

void dlut_setup_interp2D(CupvaSampler *restrict sampler,
                int tblw, int tblh, int len_out, int frac_bits,
                short * tbl, short * idx, short * out)
{
    CupvaSamplerInput2D const sampler_tbl = {
```

```
        .data           = tbl,
        .type           = SAMPLER_INPUT_TYPE_S16,
        .width          = tblw,
        .height         = tblh,
        .linePitch      = tblw,
        .outOfRangeMode = SAMPLER_OUT_OF_RANGE_CONSTANT,
        .outOfRangeVal  = 0, // don't care, not using on OOR feature
        .flags          = 0, // don't care, linePitch is specified
    };

    CupvaSamplerIndices2D const sampler_idx = {
        .data               = idx,
        .type               = SAMPLER_INDEX_TYPE_U16,
        .width              = len_out,  // idx & out are 1D
        .height             = 1,
        .linePitch          = 0,
        .fractionalBits     = frac_bits,
        .fractionalHandling = SAMPLER_FRAC_HANDLING_INTERPOLATE,
        .offsetX            = 0,
        .offsetY            = 0,
        .interleaving       = SAMPLER_INTERLEAVING_ELEMENTS,
    };

    CupvaSamplerOutput const sample_out = {
        .data      = out,
        .pitch     = 0,  // output 1D
        .transMode = TRANS_MODE_NONE,
    };

    cupvaSamplerSetup(sampler, &sampler_tbl, &sampler_idx, &sample_out);
}
```

This is setting up the DLUT task as 2D interpolation task mode and providing relevant parameters to the DLUT task.

In the main() function of this test case, DLUT is configured then invoked by this sequence of steps:

```
    CupvaSampler sampler_interp2D;

    // set up DLUT task via Sampler APIs
    dlut_setup_interp2D(&sampler_interp2D, tblw, tblh, len_out, frac_bits,
                        tbl, idx, out);

    // trigger DLUT to start
    cupvaSamplerStart(&config.z_reorder_sampler);

    // VPU can perform other processing in parallel with DLUT
```

```
    // wait for sampler to be done
    cupvaSamplerWait();
```

For common image processing tasks that process a constant-sized tile at a time, DLUT setup should ideally be performed in the application initialization time, perhaps with multiple sets of input, index, and output buffers for double-buffering.

Not counting the setup time, per-tile DLUT execution time is about 60 cycles of latency plus about len_out/4 cycles (for 16-bit 2D interpolated lookup), around 60 + 2048/4 = 572 cycles.

Besides faster processing speed than VPU, leveraging DLUT has the following advantages:

- The table does not need to be replicated, and this saves VMEM footprint, processing time and power consumption.
- While DLUT is busy performing the interpolated lookup, VPU can be potentially doing some useful work.
- DLUT generally consumes much less energy compared to VPU doing the same lookup or interpolated lookup workload.
- DLUT configuration and interaction code, in general, takes up less VPU code size than VPU doing the same lookup or interpolated lookup workload.
- DLUT provides table access out-of-bound handling without performance penalty.

Please see the PVA SDK documentation for full list of Sampler API functions.

Chapter 9. Instruction Set Reference

9.1 VPU Changes from Xavier to Orin

Changes in VPU from Xavier (Gen-1) to Orin (Gen-2) are as follows; throughput numbers are for one VPU:

- Doubled I-cache capacity from 8KB to 16KB
- Doubled VMEM capacity from 3 x 64KB to 3 x 128KB
- Double VMEM bandwidth from one read-or-write to one read and one write per superbank memory, from 3 x 512-bit per cycle to 3 x 2 x 512-bit per cycle in terms of max possible read/write transactions
- Additional vector register files: 32 x 384-bit WRF and 32 x 384-bit ARF, with ARF extended to 32 x 512-bit in select MAC and vector store instructions
- Doubled Predicate register file from 8 x 32-bit to 16 x 32-bit, from P0..P7 to P0..P15
- Integer MAC throughput boosted (see 9.2.1 for MAC instructions in Xavier/Orin)
  - 8-bit x 8-bit, from 128 MACs per cycle to 1024 MACs per cycle, 8x speedup
  - 16-bit x 16-bit, from 64 MACs per cycle to 256 MACs per cycle, 4x speedup
  - 32-bit x 16-bit, from 32 MACs per cycle to 64 MACs per cycle, 2x speedup
  - 32-bit x 32-bit, from 16 MACs per cycle to 64 MACs per cycle, 4x speedup
- Accelerated FFT (see 9.2.2 for FFT instructions in Xavier/Orin)
  - 16-bit x 16-bit complex multiply, from 16 per cycle to 32 per cycle, 2x speedup
  - 32-bit x 16-bit complex multiply, from 8 per cycle to 16 per cycle, 2x speedup
  - 32-bit x 32-bit complex multiply, from 4 per cycle to 16 per cycle, 4x speedup
  - 32-bit and 16-bit 4 x 2 add/sub
- Double throughput commonly vector operations (see 9.2.1 and 9.2.3 for such instructions)
  - Add, Sub, Compares, Min, Max, AbsDif
  - And, Or, Xor, BitCnt
  - Multiply, Multiply-add, Multiply-subtract
- Vector Blending
  - VBlend extended to cover Word type
  - New VHBlend_I to blend between even/odd lanes to work seamlessly with 2-point and 2x2-point lookup

- Enhance vector bitwise operations
  - Add scalar source 2 option and distinguish B/H/W types for bitwise And, Or, Xor
- Reduction operations going directly to scalar register destination
  - VSumR, VMinR, VMaxR, VAndR, VOrR, VXorR, VBitCmp
- Additional vector integer math instructions
  - VMinLT, VMaxLT, producing 2-input min/max and less-than/greater-than flag, to maintain min/max and index where min/max comes from
  - VSort2PL, sorting with payload, treating even lanes as keys and odd lanes as accompanying payload
  - VCollateIdx_Bits, fusing VCollateIdx (collate index) and bit-packing into scalar destination
  - VNormIdxFrac, fusing VNorm (normalization) and 2 VExtrBits (bit extraction) to produce table index and post-lookup interpolation fraction bits
  - VCmp*_AndL, VCmp*_OrL, using compare with logical and/or operations
  - VApplySign, apply positive/negative sign
  - VSelectLane, select a lane to write to scalar destination
  - VSplitBits, splitting a source into 2-bit sections
  - VXShiftL, VXShiftR, to work with an extra vector load to implement cross-lane left/right shift, for bit manipulation.
  - VHMin2ID, VHMax2ID, VMinSkip2RID, VMaxSkip2RID, Word type only, basically decomposition of VMinRID/VMaxRID Word type with vector destination into 2 instructions to avoid critical timing path.
  - VShuffle, shuffle permutation
- Vector floating-point support (see 9.2.3 for list of vector instructions added in Orin)
  - Vector FP16/FP32 FMA
  - Vector FP32/FP16 compare
  - Vector FP32 reciprocal, square root, reciprocal square root, sin, cos, log2, exp2, tanh
  - Vector FP32/FP16/INT48/INT32/INT24 conversions
- Scalar floating-point enhancement (see 9.2.4 for list of instructions)
  - In Xavier VPU there was just scalar FP32 FMA
  - All vector FP32/FP16 math instructions also offer scalar variation, except for conversion to/from IN48/INT24
- Agen features
  - Automatically predicate off stores when executed over configured number of iterations
  - Min/Max collection
  - Advance agen base
- Memory features
  - Load cache (see 5.5)

- Transpose modes T2/T4/T8/T16/T32 (see 6.3.7)
- Memory Fence (see 9.6.18)
- Load + permute (see 9.9.4.7 and 9.9.4.8)
- Per-lane rounding (see 9.9.4.9)
- 2-point and 2x2-point lookup (see 9.9.6.4 and 9.9.6.5)
- Histogram double throughput from VMEM upgrading to dual port memory
- OR-histogram (see 9.9.6.8 and 9.9.6.9)

> Decoupled coprocessor

- Coprocessors register interface (CPLD/CPST)
- Additional VMEM read/write ports to support coprocessors
- Decoupled lookup unit (DLUT)

9.2 VPU Math Operation Throughput

Math throughput is an important performance metric for a processor. The most important math operation for throughput comparison is multiply-add, especially for DSP processors. Multiplication is expensive in power, so it's useful to have a summary of various multiplication and MAC instructions to correlate performance and power consumption. Throughput numbers for a wider range of operations are also tabulated.

9.2.1 Multiply/MAC Instructions

Multiply/multiply-accumulate instructions, per instruction throughput, and per VPU MAC throughput are as follows. Instructions added in Orin are denoted in the "Added in Orin" column in the table below:

Table 18. Multiply/MAC instructions

Instruction	Function	Added in Orin	Thruput per slot	Mul / MAC Thruput per VPU (1)
VMulB	Multiply round_trunc(9b x 9b) = 12b		32	64 x 8b
VMulBBH	Multiply round_trunc(9b x 9b) = 24b		32	64 x 8b
VMulH	Multiply round_trunc(17b x 17b) = 24b		16	32 x 16b
VMulHHW	Multiply round_trunc(17b x 17b) = 48b		16	32 x 16b
VMulWHW	Multiply round_trunc(33b x 17b) = 48b		8	64 x 16b
VMulWWL	Multiply 33b x 33b = 48b : 32b		8	16 x 32b
VMulBBH (2x)	Multiply 9b x 9b = 24b	Y	64	128 x 8b
VMulHHW	Multiply 17b x 17b = 48b	Y	32	64 x 16b
VMulWHW	Multiply 33b x 17b = 48b	Y	16	64 x 16b
VMulW	Multiply trunc_16b(33b x 33b) = 48b	Y	16	32 x 32b

Instruction	Function	Added in Orin	Thruput per slot	Mul / MAC Thruput per VPU (1)
VMul2B	Multiply round_trunc(9b x 9b) = 12b		64	128 x 8b
VMul2H	Multiply round_trunc(17b x 17b) = 24b		32	64 x 16b
VMul2WHW	Multiply round_trunc(33b x 17b) = 48b		16	64 x 16b
VMAddB_CA	Multiply-add 12b + round_trunc(9b x 9b) = 12b		32	64 x 8b
VMAddBBH_CA	Multiply-add 24b + round_trunc(9b x 9b) = 24b		32	64 x 8b
VMAddH_CA	Multiply-add 24b + round_trunc(17b x 17b) = 24b		16	32 x 16b
VMAddHHW_CA	Multiply-add 48b + round_trunc(17b x 17b) = 48b		16	32 x 16b
VMAddWHW_CA	Multiply-add 48b + round_trunc(33b x 17b) = 48b		8	32 x 16b
VMAddB_CA (2x)	Multiply-add 12b + 9b x 9b = 12b	Y	64	128 x 8n
VMAddBBH_CA	Multiply-add 24b + 9b x 9b = 24b	Y	64	128 x 8b
VMAddH_CA	Multiply-add 24b + 17b x 17b = 24b	Y	32	64 x 16b
VMAddHHW_CA	Multiply-add 48b + 17b x 17b = 48b	Y	32	64 x 16b
VMAddWHW_CA	Multiply-add 48b + 33b x 17b = 48b		16	64 x 16b
VMAddW_CA	Multiply-add 48b + trunc_16b(33b x 33b) = 48b		16	32 x 32b
VDotP2BBH_CA	2-term dot product 24b + 9b x 9b + 9b x 9b = 24b		32	128 x 8b
VDotP2HHW_CA	2-term dot product 48b + 17b x 17b + 17b x 17b = 48b		16	64 x 16b
VDotP2WHW_CA	2-term dot product 48b + 33b x 17b + 33b x 17b = 48b		8	32 x 16b
VDotP2W_CA	2-term dot product 48b + trunc_16b(33b x 33b) + trunc_16b(33b x 33b) = 48b		8	32 x 32b
VDotP2x2W_CA (2x)	2-term dot product 48b + trunc_16b(33b x 33b) + trunc_16b(33b x 33b) = 48b	Y	16	64 x 32b
VDotP4BBH_CA (2x)	4-term dot product 24b + 9b x 9b + … + 9b x 9b = 24b	Y	32	256 x 8b
VDotP4BBW_CA	4-term dot product 32b + 9b x 9b + … + 9b x 9b = 32b	Y	32	256 x 8b
VDotP4HHW_CA	4-term dot product 48b + 17b x 17b + … + 17b x 17b = 48b	Y	16	128 x 16b
VDotP4WHW_CA	4-term dot product 48b + 33b x 17b + … + 33b x 17b = 48b	Y	8	128 x 16b
VDotP4x2BBH_CA (4x)	4-term dot product 24b + 9b x 9b + … + 9b x 9b = 24b	Y	64	512 x 8b
VDotP4x2BBW_CA	4-term dot product 24b + 9b x 9b + … + 9b x 9b = 32b	Y	64	512 x 8b
VDotP4x2HHW_CA	4-term dot product 48b + 17b x 17b + … + 17b x 17b = 48b	Y	32	256 x 16b
VFilt4BBH_CA (2x)	4-term filter 24b + 24b + 9b x 9b + … + 9b x 9b = 24b	Y	32	256 x 8b
VFilt4HHW_CA	4-term filter 48b + 17b x 17b + … + 17b x 17b = 48b	Y	16	128 x 16b
VFilt4x2BBH_CA (4x)	4-term filter 24b + 24b + 9b x 9b + … + 9b x 9b = 24b	Y	64	512 x 8b
VFilt4x2HHW_CA	4-term filter 48b + 17b x 17b + … + 17b x 17b = 48b	Y	32	256 x 16b
VFilt4x2x2BBH_CA (8x)	4x2-term filter 24b + 24b + 9b x 9b + … + 9b x 9b = 24b	Y	64	1024 x 8b
VFilt4x2x2BBW_CA	4x2-term filter 32b + 24b + 9b x 9b + … + 9b x 9b = 32b	Y	64	1024 x 8b

Note: Count conventional 8b/16b/32b multiplications or multiply-accumulates. 33b x 17b counted as 2 16b MACs.

9.2.2 MAC-Related Instructions

Additional instructions that leverage multiply-add or multiply-accumulate datapath:

Instruction	Function	Added in Orin	Thruput per slot	Mul / MAC Thruput per VPU (1)
VCMulH	Complex multiply round_trunc(17b x 17b) = 24b		8	64 x 16b
VCMulHHW	Complex multiply round_trunc(17b x 17b) = 48b		8	64 x 16b
VCMulHHW (2x)	Complex multiply 17b x 17b = 48b	Y	16	128 x 16b
VCMulWHW (2x)	Complex multiply 33b x 17b = 48b	Y	8	128 x 16b
VCMulW	Complex multiply trunc_16b(33b x 33b) = 48b	Y	8	64 x 32b
VMSubB_CA	Multiply-subtract 12b + round_trunc(9b x 9b) = 12b		32	64 x 8b
VMSubBBH_CA	Multiply-subtract 24b + round_trunc(9b x 9b) = 24b		32	64 x 8b
VMSubH_CA	Multiply-subtract 24b + round_trunc(17b x 17b) = 24b		16	32 x 16b
VMSubHHW_CA	Multiply-subtract 48b + round_trunc(17b x 17b) = 48b		16	32 x 16b
VMSubWHW_CA	Multiply-subtract 48b + round_trunc(33b x 17b) = 48b		8	32 x 16b
VMSubW_CA	Multiply-subtract 48b + trunc_16b(33b x 33b) = 48b		8	16 x 32b
VMSubB_CA (2x)	Multiply-subtract 12b + 9b x 9b = 12b	Y	64	128 x 8b
VMSubBBH_CA	Multiply-subtract 24b + 9b x 9b = 24b	Y	64	128 x 8b
VMSubH_CA	Multiply-subtract 24b + 17b x 17b = 24b	Y	32	64 x 16b
VMSubHHW_CA	Multiply-subtract 48b + 17b x 17b = 48b	Y	32	64 x 16b
VMSubWHW_CA	Multiply-subtract 48b + 33b x 17b = 48b	Y	16	64 x 16b
VMSubW_CA	Multiply-subtract 48b + trunc_16b(33b x 33b) = 48b	Y	16	32 x 32b
VBlendB	Blend 12b + round(9b x 8b – 9b x 8b) = 12b		32	128 x 8b
VBlendH	Blend 24b + round(17b x 16b – 17bb x 16b) = 24b		16	64 x 16b
VBlendW	Blend (48b << 16) + trunc_16b(33b x 32b) – trunc_16b(33b x 32b) = 48b	Y (W)	8	32 x 32b
VHBlend_IB	Blend 12b + round(9b x 8b – 9b x 8b) = 12b	Y	32	128 x 8b
VHBlend_IH	Blend 24b + round(17b x 16b – 17bb x 16b) = 24b	Y	16	64 x 16b
VHBlend_IW	Blend (48b << 16) + trunc_16b(33b x 32b) – trunc_16b(33b x 32b) = 48b	Y	8	32 x 32b
VHBlend_IBHB	Blend 12b + round(9b x 8b – 9b x 8b) = 12b	Y	32	128 x 8b
VXNorAdd8x4x2_CA	8x4-term XNorAdd 16b + 1b ^ 1b + … + 1b ^ 1b = 16b	Y	128	8192 x 1b
VSumSqBBH	Sum of square 9b x 9b + 9b x 9b = 24b	Y	32	128 x 8b
VSumSqHHW	Sum of square 17b x 17b + 17b x 17b = 48b	Y	16	64 x 16b
VSumSqW	Sum of square trunc_16b(33b x 33b) + trunc_16b(33b x 33b) = 48b	Y	8	32 x 32b
VSqSumBBH	Square of sum (9b + 9b) x (9b + 9b) = 24b	Y	32	192 x 8b
VSqSumHHW	Square of sum (17b + 17b) x (17b + 17b) = 48b	Y	16	96 x 16b
VDet2x2HHW	Determinant 2x2 17b x 17b + 17b x 17b = 48b	Y	16	64 x 16b
VDet2x2W	Determinant 2x2 trunc_16b(33b x 33b) – trunc_16b(33b x 33b) = 48b	Y	8	32 x 32b

9.2.3 Other Accelerated Vector Math Instructions

Selected math operations are accelerated over baseline 32 x 12-bit, 16 x 24-bit, or 8 x 48-bit per vector slot.

See Removed/Emulated Instructions for list of Xavier vector math instructions that we removed in Orin, where there is slowdown instead of speedup. We retain intrinsic functions to maintain source code compatibility through emulating the functionality with other instructions.

Instruction	Function	Added in Orin	Thruput per slot	Operation Thruput per VPU
VAddB/H/W	Addition			
VSubB/H/W	Subtraction			
VAndB/H/W	Bitwise and			
VOrB/H/W	Bitwise or			
VXorB/H/W	Bitwise exclusive-or			
VMinB/H/W	Min			
VMaxB/H/W	Max			
VCmpLTB/H/W	Compare less than			
VCmpLEB/H/W	Compare less than or equal to			
VCmpGTB/H/W	Compare greater than			
VCmpGEB/H/W	Compare greater than or equal to			
VCmpEQB/H/W	Compare equal			
VCmpNEB/H/W	Compare not equal			
VBitCntB/H/W	Bit count			
VAbsDifB/H/W	Absolute difference			
	12-bit operation		32	64 x 12-bit
	24-bit operation		16	32 x 24-bit
	48-bit operation		8	16 x 48-bit
2x perf of above	12-bit operation	Y	64	128 x 12-bit
	24-bit operation	Y	32	64 x 24-bit
	48-bit operation	Y	16	32 x 48-bit
VAdd2SubB	12-bit A + B – C		32	128 x 12-bit
VAdd2SubH	24-bit A + B – C		16	64 x 24-bit
VAdd2SubW	48-bit A + B – C		8	32 x 48-bit
	4-input-2-output add/subtract for radix-4 FFT			
VAddSub4x2B (3x)	12-bit	Y	32	192 x 12-bit
VAddSub4x2H	24-bit	Y	16	96 x 24-bit
VAddSub4x2W	48-bit	Y	8	48 x 48-bit
	4-input-2-output configurable add/subtract			
VCfgAddSub4x2B (3x)	12-bit	Y	32	192 x 12-bit

Instruction	Function	Added in Orin	Thruput per slot	Operation Thruput per VPU
VCfgAddSub4x2H	24-bit	Y	16	96 x 24-bit
VCfgAddSub4x2W	48-bit	Y	8	48 x 48-bit

9.2.4 Scalar/Vector Floating-point Instructions

In Xavier VPU we support only scalar FP32 instructions. In Orin VPU to extend floating support to both scalar and vector, and both FP32 and FP16.

Instruction	Function	Added in Orin	Thruput per slot	Operation Thruput per VPU
VAddF	FP32 addition	Y	8	16 x 32-bit
VSubF	FP32 subtraction	Y		
VMulF	FP32 multiplication	Y		
VMAddF	FP32 multiply-add	Y		
VMSubF	FP32 multiply-subtract	Y		
VCmp*F	FP32 comparison LT/LE/GT/GE/EQ/NE	Y		
VRCPF	FP32 reciprocal	Y		
VSQRTF	FP32 square root	Y		
VRSQF	FP32 reciprocal of square root	Y		
VEXP2F	FP32 exponent based 2	Y		
VLOG2F	FP32 log based 2	Y		
VSINF	FP32 sine	Y		
VCOSF	FP32 cosine	Y		
VTANHF	FP32 hyperbolic tangent	Y		
VAddHF	FP16 addition	Y	16	32 x 16-bit
VSubHF	FP16 subtraction	Y		
VMulHF	FP16 multiplication	Y		
VMAddHF	FP16 multiply-add	Y		
VMSubHF	FP16 multiply-subtract	Y		
VCmp*HF	FP16 comparison LT/LE/GT/GE/EQ/NE	Y		
VINT_FP	INT32 to FP32 conversion	Y	8	16 x 32/48-bit
VFP_INT_Trunc	FP32 to INT32 conversion with truncation	Y		
VFP_INT_Round	FP32 to INT32 conversion with rounding	Y		
VINTX_FP	INT48 to FP32 conversion	Y		
VFP_INTX_Trunc	FP32 to INT48 conversion with truncation	Y		
VFP_INTX_Round	FP32 to INT48 conversion with rounding	Y		
VINT_FP16	INT32 to FP16 conversion	Y	16	32 x 16/24-bit
VFP16_INT_Trunc	FP16 to INT32 conversion with truncation	Y		
VFP16_INT_Round	FP16 to INT32 conversion with rounding	Y		

Instruction	Function	Added in Orin	Thruput per slot	Operation Thruput per VPU
VINT24_FP16	INT24 to FP16 conversion	Y		
VFP16_INT24_Trunc	FP16 to INT24 conversion with truncation	Y		
VFP16_INT24_Round	FP16 to INT24 conversion with rounding	Y		
VFP16_FP	FP16 to FP32 conversion	Y		
VFP_FP16	FP32 to FP16 conversion	Y		
FAdd (Scalar)	FP32 addition		1	2 x 32-bit
FSub	FP32 subtraction			
FMul	FP32 multiplication			
FMAdd	FP32 multiply-add			
FMSub	FP32 multiply-subtract			
FCmp*	FP32 comparison LT/LE/GT/GE/EQ/NE	Y		
HFAdd (Scalar)	FP16 addition	Y	1	2 x 16-bit
HFSub	FP16 subtraction	Y		
HFMul	FP16 multiplication	Y		
HFMAdd	FP16 multiply-add	Y		
HFMSub	FP16 multiply-subtract	Y		
HFCmp*	FP16 comparison LT/LE/GT/GE/EQ/NE	Y		
FRCP (Scalar)	FP32 reciprocal	Y	1	2 x 32-bit
FSQRT	FP32 square root	Y		
FRSQ	FP32 reciprocal of square root	Y		
FEXP2	FP32 exponent based 2	Y		
FLOG2	FP32 log based 2	Y		
FSIN	FP32 sine	Y		
FCOS	FP32 cosine	Y		
FTANH	FP32 hyperbolic tangent	Y		
INT_FP (Scalar)	INT32 to FP32 conversion		1	2 x 32-bit
FP_INT_Trunc	FP32 to INT32 conversion with truncation	Y		
FP_INT_Round	FP32 to INT32 conversion with rounding			
INT_FP16	INT32 to FP16 conversion	Y		
FP16_INT_Trunc	FP16 to INT32 conversion with truncation	Y		
FP16_INT_Round	FP16 to INT32 conversion with rounding	Y		
FP16_FP	FP16 to FP32 conversion	Y		
FP_FP16	FP32 to FP16 conversion	Y		

9.2.5 Scalar Integer Math Instructions

In the 2 scalar math slots, we support a variety of integer math instructions as well:

Instruction	Function	Added in Orin
Add	Addition	
Sub	Subtraction	
And	Bitwise and	
Or	Bitwise or	
Xor	Bitwise exclusive-or	
SLL	Shift left logical	
SRL	Shift right logical/unsigned	
SRA	Shirt right arithmetic/signed	
SXTD	Sign-extend	
ZXTD	Zero-extend	
CmpEQ	Compare equal	
CmpNE	Compare not equal	
CmpGE (U)	Compare greater than (unsigned)	
CmpGT (U)	Compare greater than or equal to (unsigned)	
CmpLE (U)	Compare less than (unsigned)	
CmpLT (U)	Compare less than or equal to (unsigned)	
MIN (U)	Minimal (unsigned)	
MAX (U)	Maximal (unsigned)	
Mul	32-bit x 32-bit -> 32-bit multiply	
LMulSS	32-bit x 32-bit -> 64-bit multiply signed-signed	
LMulSU	32-bit x 32-bit -> 64-bit multiply signed-unsigned	
LMulUU	32-bit x 32-bit -> 64-bit multiply unsigned-unsigned	
Div	Integer division (variable # cycles)	
MODINC	Modular increment	
MODINCP	Modular increment and predicate if not zero	
MODINC_NOTP	Modular increment and predicate if zero	
DPMODINCP	Modular increment and predicate double if not zero	
DPMODINC_NOTP	Modular increment and predicate double if zero	
MUXP	Multiplex from predicate (C select operator)	
MUX	Multiplex from scalar register (C select operator)	
SLLIADD	Shift left immediately and add	Y
CMPWITHIN	Compare within low/high bounds	Y
BITCNT	Bit count	Y

9.3 VPU Compatibility

9.3.1 Compatibility Exceptions

We aim to maintain C source code backward compatibility with Xavier (Gen-1) VPU. We do not plan to support assembly code or binary compatibility.

There are a few cases where we need to break C source code compatibility in Orin VPU.

- Vector multiply-add rounding/truncating options, in Gen-1 we supported {.R0, .R7, .R15, .R16, .T0, .T7, .T15, .T16}. In Gen-2 we added .R4, taking up encoding space of .T0. Thus, hard-coded rounding/truncating option 4 in the application code, which was mapped to .T0, in Gen-2 will map to .R4.
- Some VMEM storage classes involving Word and Halfword types need to be revised to the base classes involving Byte type.
  - RAM_Aw, RAM_Ah → RAM_Ab
  - RAM_Bw, RAM_Bh → RAM_Bb
  - RAM_Cw, RAM_Ch → RAM_Cb
  - DMw, DMh → DMb
- CLRHWLP needs 3 instruction packets of gap to the Loop End instruction packet for the clear hardware loop (and exit loop) functionality to work.
- Agen auto predication features would predicate off any agen-based scalar/vector store past the configured iteration counts. For example, if N1/N2/N3/N4/N5/N6 are left unchanged after initializing an agen (which would set them to default value of 1), in Gen-1 ISS/silicon, multiple stores to the same location (as address would stick to last valid address), but in Gen-2 ISS/silicon, only the first store would be carried out; subsequent stores are blocked and thus not carried out.
- Address map difference and aliasing of address space means that code that addresses outside primary address regions would behave differently in Gen-1 ISS/silicon versus Gen-2 ISS/silicon. For example, reading 0x10024 would be aliased back to physical memory at 0x24 in Gen-1, and would be reading physical memory at 0x10024 in Gen-2.
- Gen-1 VPU supports floating-point math in scalar slots only and FP32 only, and functionality was implemented with Synopsys DesignWare floating-point fused multiply-add unit, Gen-2 VPU extends floating-point support to scalar/vector and FP16/FP32, and functionality was provided by reusing NVIDIA GPU SM floating-point unit. There can be differences in various corner case behavior around +/- zero, +/- infinity, and denormal numbers.
- vbool, vector Boolean type, was removed as it is ambiguous (as how many lanes of Boolean).
- Intrinsic functions for VMinRID/VMaxRID in Gen-1 was vminr()/vmaxr(), which are easy to confuse with intrinsic functions for VMinR/VMaxR. They are corrected in Gen-2 as vminrid()/vmaxrid().

- Agen configuration load/store syntax was revised to better support Native compilation. See AgenCfgST and AgenCfgLD for details.

9.3.2 Removed/Emulated Instructions

The following instructions are removed from the Orin VPU instruction set due to timing pressure:

- Removed VMinR, VMaxR with vector register destination, replaced with scalar destination
- Removed VMinRID, VMaxRID with vector register destinations (dst1 & dst2), replaced with scalar destinations (dst1 & dst2)
- Removed VPromote (without deinterleaving)

The intrinsic functions are still supported by emulating the functionality with multiple instructions. We do not regard this as breaking backward compatibility, but it is worth noting, in case programmers see compute kernels utilizing these instructions performing slower in Orin ISS/silicon versus Xavier ISS/silicon.

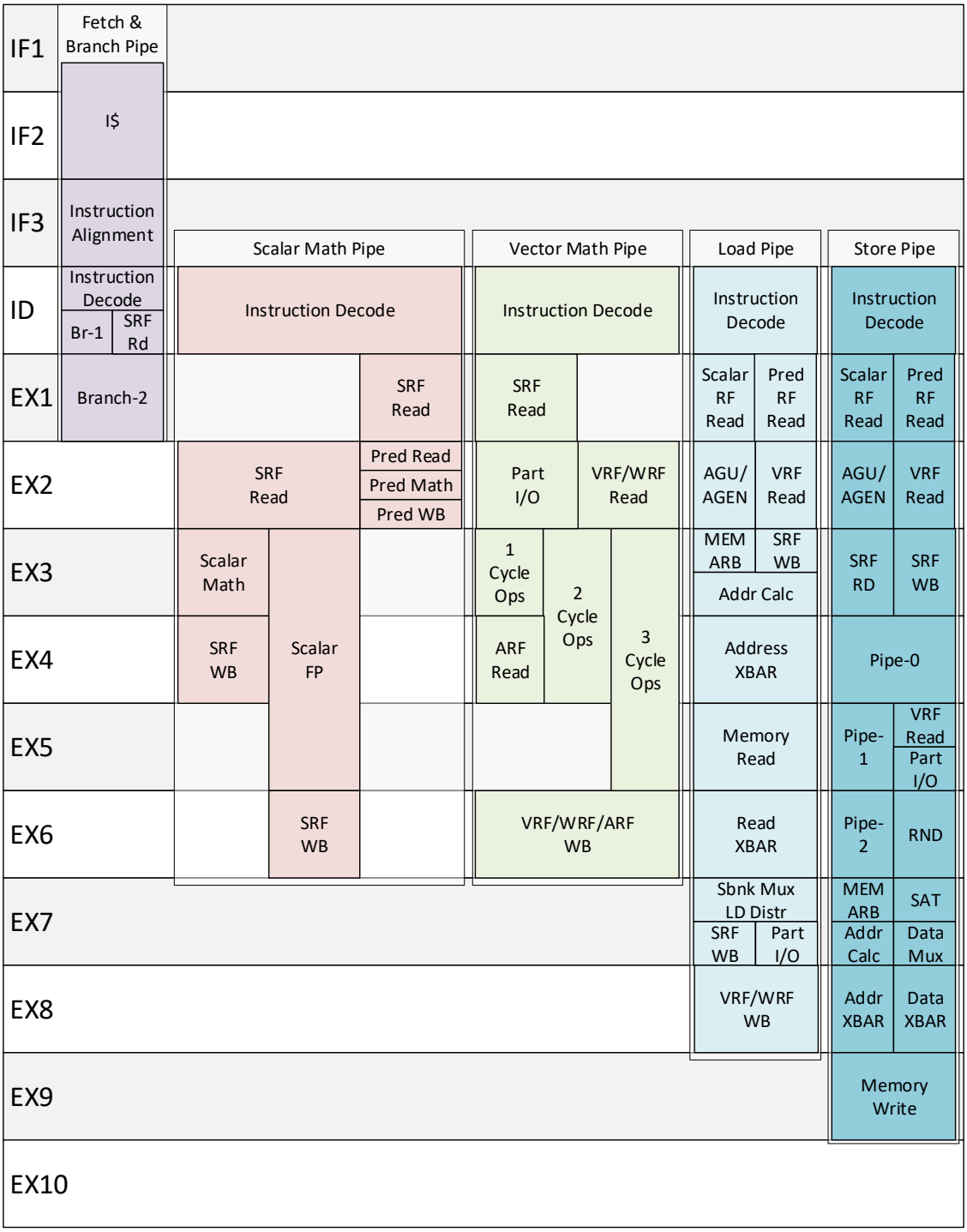
9.4 Instruction Execution Ordering

9.4.1 Processor Pipeline

Normally processor pipelining is behind the scenes, as execution packets appear to execute sequentially, and mostly one packet per cycle, with instructions in the same packet executed in parallel. However, to understand various conditions where the processor stalls, and the few exceptions to the sequential execution behavior better, we need to learn about the VPU processor pipeline stages:

- IF1..IF3: Instruction fetch stages
- ID: Instruction decode stage
- EX1 .. EX9: execution stages
- VPU pipeline diagram follows.

Figure 13. VPU processor pipeline



9.4.2 Default/General Behavior

The VPU instructions execute in the following general order consistent with assembly encoding:

1. Scalar and vector instructions in the same VLIW execution packet are executed in parallel.
2. Within the same VLIW execution packet, loads are executed before stores.
3. Multiple stores are executed in parallel if they go to different memory superbanks. Multiple writes going to the same memory superbanks are executed in slot order.
4. Reading the same register (scalar or vector) by multiple slots is supported.
5. Writing to same register (scalar or vector) by multiple slots is NOT allowed (compiler does not schedule such code, and such code would cause assembler to fail).
6. Same register (scalar or vector) can be read (multiple times) and written (only once) in the same execution packet, read preceding write.

9.4.3 Delay Slots for Branch Instructions

Branch and hardware loop instructions have delay slots, so they also appear as executing out of order; 2 packets after branch instruction are executed before taking the branch.

SWRBK, CLR_HWLP, STW HWLP, WFE_GPI, and WFE_R5 should not be placed in a branch delay slot.

Please see Control Instruction Summary for number of delay slots for each instruction.

9.4.4 Exception for Instructions Accessing Address Generator

Address Generator fields have the following read/write accesses:

- MovAgen reads and writes Agen in EX2
- Agen-based load/store reads and writes Agen in EX2 (reading most fields, writing base and loop variables)
- Agen-based store reads and writes Agen in EX7 (updating MinVal, MaxVal)
- InitAgen and CfgAgen write Agen in EX2
- Store Agen Loopvar reads Agen in EX2
- AgenCfgST/AgenCfgST_p2 reads Agen in EX7
- AgenCfgLD/AgenCfgST_p2 writes Agen in EX7

In the processor model we have, hardware stalls so that instructions appear to be executed sequentially. However, instructions from the same execution packet are executed or stalled together, except stalling for memory dependency. Thus, Agen read/write instructions that access Agen in different pipeline stages exhibit non-sequential behavior.

Write-EX2 + Read-EX7 in the same packet: would appear that write precedes read, violating rule #6. Possible combinations for this category are:

- MovAgen with AgenCfgST in same packet: Moved Agen contents are stored to memory.

- Agen-based load/store with AgenCfgST in the same packet: Updated Agen contents are stored to memory.
- InitAgen or CfgAgen with AgenCfgST in the same packet: Configured Agen contents are stored to memory.

Write-EX2 + Write-EX7 in the same packet: allowed, with Write-EX2 occurring before Write-EX7, so the outcome from Write-EX7 stays. This is violating rule #5. Possible combinations for this category are:

- MovAgen with AgenCfgLD in same packet: Moved Agen contents are lost, overridden by load outcome of AgenCfgLD.
- Agen-based load with AgenCfgLD in same packet: Agen-based load is carried out with current address value (since agen-update is post-modifying). Agen address update is lost, overridden by load outcome of AgenCfgLD.
- InitAgen or CfgAgen with AgenCfgLD in same packet: Configured Agen contents is lost, overridden by load outcome of AgenCfgLD.

Note that Read-EX2 + Write-EX7, Read-EX2 + Write-EX2, and Read-EX7 + Write-EX7 in same packet would appear that read precedes write and thus conform to the general instruction ordering (rule #6).

Agen-based load/store (reading agen configuration in EX2) and AgenCfgLD (writing agen configuration in EX7) in same packet: agen-based load/store uses configuration before AgenCfgLD

MovAgen (reading source agen in EX2) and InitAgen/AgenCfg (writing agen configuration in EX2) in same packet: source agen of MovAgen is read first, before being updated by InitAgen/AgenCfg

AgenCfgST_p2 (reading agen loop variables and min/max value in EX7) and agen-based store (reading/writing agen loop variables and min/max value in EX7, min/max value only when min/max collection is enabled) in same packet: AgenCfgST_p2 stores agen loop variables etc. before being updated by the agen-based store.

Agen-based load with AgenCfgLD in the same packet is allowed in Xavier VPU but is disallowed in Orin VPU. In Orin, we have added min/max collection feature, and both instructions are written into MinVal/MaxVal agen fields in EX7.

9.4.5 Exception for Instructions Accessing HW Loop Registers

The hardware zero-overhead looping utilizes the following registers:

- LF: 2-bit loop level, -1, 0 or 1, indicating which loop level the execution is in, reset to -1 (which is encoded as binary “11”).
- LS[0..1]: 32-bit loop start PC, reset to 0
- LE[0..1]: 32-bit loop end PC, reset to 0
- LC[0..1]: loop count, 32-bit, reset to 1

The hardware Loop instruction RPT accesses these registers (both read and write) in EX2 stage.

The PCU, program control unit, accesses these registers (both read and write) upon end of the loop (PC matching LE[LF]) to implement looping behavior.

These registers are written by CLR_HWLP instruction, to clear hardware loop context for a new algorithm task, and read by STW HWLP instruction, for debug. These instructions have placement restrictions with respect to hardware loop, to avoid hazards.

CLR_HWLP should not be placed:

- In two packets before RPT
- In the same packet as RPT
- In the two RPT delay slots
- In the first 2 packets of loop body
- In the last 2 packets of loop body
- In the first 2 packets after the loop.

Otherwise, hardware loop state is non-deterministic.

STW HWLP should not be placed:

- In two packets before RPT
- In the same packet as RPT
- In two RPT delay slots
- In the first 3 packets of loop body
- In the last 3 packets of loop body
- In the first 2 packets after the loop.

Otherwise, stored contents are non-deterministic.

These restrictions do not affect instructions injected through debug in Debug State, since such instructions are executed one instruction at a time through all pipeline stages.

9.4.6 Exception for Instructions Accessing FP Invalid Flag

With the scalar and vector unit FP instructions, we have an invalid flag that FP operations can set, and a pair of move instructions moving between the flag and a scalar register that we can use to acquire and clear the flag.

Interesting scenarios:

- When there are multiple FP operations in the same packet, the invalid outcome from any operation can set the invalid flag, and since the flag is sticky, the flag update can be represented as follows:

```
invalid_flag |= s0_invalid | s1_invalid | v0_invalid | v1_invalid
```

- When MOV R, INV instruction (only in S0 since it's classified as a control instruction) and FP operation(s) (in S1/V0/V1 slots) are placed in the same packet, writing of the flag from MOV R, INV instruction is ignored, overridden by the FP operation(s).
- When MOV INV, R instruction (in S0) and FP operation(s) (in S1/V0/V1 slots) are placed in the same packet, reading of the flag occurs before the FP operation(s) affect the flag. This case is consistent with the "read before write" general ordering rule.

9.4.7 Hardware Stalls to Comply with Sequential Execution Order

There is RAW (read after write) and WAW (write after write) data hazard detection on all register files (scalar, predicate, agen, VRF, WRF, ARF, XARF) to ensure sequential execution regarding dependency through registers.

Various control instructions interact with components external to the VPU processor in various pipeline stages:

- GPO_SET/CLR/WR affect GPO pins in the EX2 stage.
- GPI_RD reads GPI pins in the EX2 stage.
- CPST writes to coprocessor space via APB write transaction in the EX4 stage (address/write-request/write-data driven in EX4, wait for peripheral to be ready in EX5).
- CPLD reads from coprocessor space via APB read transaction in the EX5 stage (address/read-request driven in EX4, wait for peripheral to be ready and read-data in EX5).
- SIG_R5 raises vpu_start_r5 control signal to R5 in the EX3 stage.
- WFE_GPI and WFE_R5 waits for all proceeding instructions to exit pipeline before execution, so has their own mechanism to ensure sequential execution.

SIG_R5 and WFE_R5 are involved in R5/VPU communication. As R5 and VPU are two separate processor cores, we are not relying on fine timing of individual signals, but on the interaction protocol, to ensure coherent behavior.

Among the remaining external interface instructions, i.e. GPI/GPO/CPLD/CPST, GPI and CPLD are read actions, and GPO and CPST are write actions. We need to watch for potential RAW hazards:

- GPI after GPO: both execute in EX2, so execution order is preserved.
- GPI after CPST: CPST executes in EX4 and GPI in EX2, so potential RAW hazard. Hardware stalls GPI in EX2 (or earlier) until peripheral responds to readiness for the CPST transaction.
- CPLD after GPO: GPO executes in EX2 and CPLD in EX4, so execution order is preserved.
- CPLD after CPST: CPST executes in EX4 and CPLD in EX5, also APB bus is sequential, so execution order is preserved.

Potential WAW hazards:

- GPO after CPST: CPST executes in EX4 and GPO in EX2, so potential WAW hazard. Hardware stalls GPO in EX1 (or earlier) until peripheral responds to readiness for the CPST transaction.
- CPST after GPO: GPO executes in EX2 and CPST in EX4, so execution order is preserved.

9.5 Instruction Predication

The VPU has 14 32-bit predicate registers, P2… P15. P0 are P1 are reserved to indicate unpredicated (always-execute) instructions. In addition, the first half of the main vector register file, V0..V15, can be used for vector store lane predication.

The following predication features are available:

- Vector math instruction-level predication.
- Vector load instruction-level predication.
- Scalar store instruction-level predication.
- Vector store lane predication.

9.5.1 Instruction-Level Predication for Register Moves

Scalar-to-scalar, scalar-to-vector, and vector-to-scalar are instruction-level predicated. When predication is on (nonzero), the register move is performed. When predication is off (zero), the register move is skipped.

Predicated register move can be used for conditional execution to avoid conditional branches.

9.5.2 Instruction-Level Predication for Vector Math

Selected vector ALU instructions are predicated on or off identically across lanes, MOVS (scalar-to-vector move) and those with "_CA" suffix in mnemonic. It's a common decision for all lanes to carry out one functionality or the other, with the predication-off functionality emulating clearing of the accumulator.

For example:

```
   [P2]  VMAddHHW_CA V0, V1, V2:V3   // if (P21==0), V2:V3 = V0*V1
// otherwise V2:V3 += V0*V1
```

Clearing of the accumulator typically happens periodically, once every K iterations, where K is number of items being accumulated, as in filtering. MODINCP can be used to implement a modulo K counter to control the periodic predication.

Please consult the description of individual instructions for additional details.

9.5.3 Predication for Load/Store

Predication support for various addressing modes of scalar/vector load/store is shown as follows:

Table 19. Scalar/vector load/store predication support

Predication feature	Base+offset	Post-modify	Agen-based
Scalar load	not available	instruction-level	not available
Scalar store	not available	instruction-level	instruction-level
Vector load	not available	not available	instruction-level
Vector store	not available	not available	per-lane

9.5.3.1 Instruction-Level Predication for Post-Modify Scalar Load

Scalar load with post-modify addressing mode is instruction-level predicated.

When predication is on, memory read, address register update, and destination write are carried out. Otherwise, none of these are carried out. Of course, predicate register will always be read for the predication.

Predicated scalar load/store is used to accelerate various conditional scalar processing.

9.5.3.2 Instruction-Level Predication for Post-Modify and Agen-Based Scalar Store

Scalar store, both post-modify and agen-based variations are instruction-level predicated.

For the post-modify scalar store, predication drives both memory write and the register update (base += modifier). When predication is on, both memory write and register update are carried out, otherwise, both are not carried out.

For the agen-based scalar store, predication drives only memory write. Agen update is always carried out. When predication is on, memory write is carried out, otherwise, memory write is not carried out.

In both kinds of scalar stores, source register read is carried out unconditionally, with any necessary hardware stalling to preserve source register dependency.

Predicated scalar load/store is used to accelerate various conditional scalar processing.

9.5.3.3 Instruction-Level Predication for Agen-Based Vector Load

Agen-based vector load instructions are instruction-level predicated. When predication is on, memory read and destination vector register write are performed. When predication is off, memory read and destination vector register write are skipped. Address update is carried out unconditionally.

A use case for predicated vector load is for **integral image**, where predication is used to **deal with boundary rows.**

9.5.3.4 Lane Predication for Agen-Based Vector Store

Agen-based vector store instructions are predicated per lane. Predication-on lanes are written to memory, predication-off lanes are skipped. Address update is carried out unconditionally.

Predication is conveyed via either predicate register(s) or a single vector register in VRF.

In case of predication via predicate register(s), as many bits of predicate register are used as the number of lanes, and up to 64 lanes, or two predicate registers, are used. The predication bits are the least significant bit justified.

For example, “[P2] DVSTW_PI V0, *A0++” stores 16-word lanes, with lane i predicated by bit i of the predicate register P2.

In case of predication via a single vector register in VRF, predicates are evenly spaced starting from bit 0. The VRF entry is regarded as a 384-bit vector, and a single bit is used for each lane. Bit position for each lane is lane_index * (384/num_lanes).

For vector store with scalar distribution, for example, VSTW_S, predication is supported only through predicate registers, and not through vector register. We are storing out just one or two values so there is little value in using vector register to convey predicates.

The following table shows bits of VRF used across variations of vector store:

Table 20. Vector register predicated vector store variations

Vector store	Number of source lanes	Bits used in predicate VRF entry	As bit 0 of array elements
VSTB_P/T VSTBH_P/T	32	0, 12, 24, ..., 372	arr_vcharx[0, 1, ..., 31]
VSTH_P/T VSTHW_P/T	16	0, 24, 48, ..., 360	arr_vshorx[0, 1, ..., 15]
VSTW_P/T VSTWX_P	8	0, 48, 96, ..., 336	arr_vintx[0, 1, ..., 7]
VSTB_S	1	Predication via VRF not supported	

Vector store	Number of source lanes	Bits used in predicate VRF entry	As bit 0 of array elements
VSTH_S VSTW_S			
DVSTB_P/PI	64	0, 6, 12, ..., 378	as bit 0 and bit 6 of arr_vcharx[0, 1, ..., 31]
DVSTH_P/PI/T/TI DVSTHB_P/PI	32	0, 12, 24, ..., 372	arr_vcharx[0, 1, ..., 31]
DVSTW_P/PI/T/TI/T2/T2I DVSTWH_P/PI/T/TI	16	0, 24, 48, ..., 360	arr_vshortx[0, 1, ..., 15]
DVSTB_S DVSTH_S DVSTW_S	2	Predication via VRF not supported	
QVSTHB_P/PI/PI2	64	Predication via VRF not supported	
QVSTWH_P/PI/PI2/T/TI2	32	Predication via VRF not supported	

For example, “[V2] DVSTW_PI V0, *A0++” stores 16 word lanes, with lane i predicated by bit i*24 of V2, or bit 0 of each element of a vshortx-type variable mapped to V2.

There is a behavior difference between predicate register file and vector register file for predication. With predicate register file, in case all lanes are predicated off, the memory transaction is **not issued**, conserving power consumption. With vector register file, to shorten the latency the VRF entry is read late in the pipeline, same stage as the store data, too late to block the memory transaction, so the predicated memory transaction is **always issued**.

Lane-predicated store via predicate register is supported in all types and distribution combinations of Agen-based scalar/vector stores as well as VAST, vector addressed stores, and in all memory slots.

Lane-predicated store via VRF is supported in agen-based single/double vector store of VRF, non-scalar distribution, and in M0 slot only.

9.6 Control Instructions

9.6.1 Instruction Summary

The following control instructions are supported. Most are available only in the S0 slot, except the following:

- RD_TSC.L/H can be issued in both S0 and S1 slots.
- CPLD, CPST, MemFence are available only in M0 slots.

In the table, delay slots refer to execution packets (one slot is one packet) following the control instructions that are executed before the control instruction takes place. For

example, the JR instruction has 2 delay slots, so two execution packets following the JR instruction's own packet are executed before the jump takes place.

Table 21. Control instructions

Function	Assembly Format	Comments
Jump to immediate	J imm20_addr	Jump to relative immediate address, with 2 delay slots.
Jump to register	JR Raddr	Jump to absolute address in register, with 2 delay slots.
Jump and link (call)	JAL imm20_addr	Call (jump and link) relative immediate address., with 2 delay slots.
Jump and link register (call)	JALR Raddr	Call absolute address in register, with 2 delay slots.
Branch if zero	BEQZ Rsrc, imm14_addr	Branch if Rsrc is zero to relative immediate address, 2 delay slots.
Branch if nonzero	BNEZ Rsrc, imm14_addr	Branch if Rsrc is not zero to relative immediate address, 2 delay slots.
Software break point	SWBRK	Software break point.
Hardware loop	RPT Rsrc, imm16	Hardware zero-overhead loop, with the Rsrc specifying number of iterations, and the immediate encoding size of the loop, with 2 delay slots.
Clear hardware loop registers	CLR_HWLP	Initialize hardware loop registers to default values
GP out set low	GPO_SETLI imm16	Set lower 16-bit of GPO according to immediate
GP out set high	GPO_SETHI imm16	Set higher 16-bit of GPO according to immediate
GP out clear low	GPO_CLRLI imm16	Clear lower 16-bit of GPO according to immediate
GP out clear high	GPO_CLRHI imm16	Clear higher 16-bit of GPO according to immediate
GP out set	GPO_SET Rsrc	Set 32-bit of GPO according to Rsrc
GP out clear	GPO_CLR Rsrc	Clear 32-bit of GPO according to Rsrc
GP out write	GPO_WR Rsrc	Copy 32-bit Rsrc to 32-bit GPO
GP out read	GPO_RD Rdst	Copy 32-bit GPO into Rdst
GP in read	GPI_RD Rdst	Sample 32-bit GPI into Rdst
Wait for GPI pattern	WFE_GPI Rsrc1, Rsrc2	Wait until (GPI & Rsrc1) == Rsrc2
Wait for R5 event	WFE_R5	Transition into low-power WFE_R5 state until R5 writes R5_vpu_start to dispatch next task
Signal R5	SIG_R5 Rsrc	Send software interrupt to R5; Rsrc carries a software-defined 32-bit data to write to a VPU config register, which R5 interrupt service routine can read.
Enable timestamp counter	ENABLE_TSC	Enable performance counter Once enabled, timer increments in Active state (and not increment in Reset, Debug, WFE_R5, WFE_GPI, Halted, Error-Halted states).

Function	Assembly Format	Comments
Read timestamp counter	RD_TSCL Rdst RD_TSCH Rdst	Copy performance counter lower/upper 32-bit to Rdst. S0 and S1 slots.
Move FP invalid flag	MOV INV-R MOV R-INV	Move floating-point invalid flag to/from scalar register
OCD load/store	OCD_LD PC/GPO OCD_ST PC/GPI/GPO	OCD (debug) load/store
Configure VMEM Superbanks	CFG_VMEM_SBA/B/C Rsrc RD_CFG_VMEM_SBA/B/C Rdst	Write configuration Read configuration
Coprocessor store	CPST Rsrc, Rdaddr CPST Rsrc, #imm12	
Coprocessor load	CPLD Rsaddr, Rdst CPLD #imm12, Rdst	M0 slot only
Memory fence	MemFence	M0 slot only

The VPU does not take interrupts, and thus there is no enable/disable interrupt, return from interrupt, etc., instructions available.

The PC is internally modeled to count in 32-bit increments. For example, PC = 1 means byte address of 4. The 20-bit absolute immediate field for J, JAL, the 14-bit relative immediate fields for BEQZ, BNEZ, the 16-bit immediate field for RPT, conform to this convention (count in 32-bit increments).

By default, the compiler aligns all branch targets to 256-bit = 32-byte = 8-word alignment, to avoid the instruction fetch interface spending an extra cycle to fetch a execution packet starting from target PC.

9.6.2 Branch/Jump/Call Delay Slots

For the processor pipeline to work, 2 execution packets after the branch/jump/call instructions are executed before taking the branch/jump/call. These 2 execution packets are called in the delay slots of the branch/jump/call instructions. Please see instruction summary or details in each branch/jump/call instruction for how many delay slots there are.

Note that the branching action is delayed but register read/write is still executed sequentially.

For example, case 1:

1 LDHI R5, #0

2 BEQZ R5, #42

3 ADDI R5, #-1, R5

4 NOP

5 HALT

In this case, R5 for instruction #2 is sampled and branch decision made accordingly. Subsequent instruction #3 that changes R5 does not change the branch decision.

For example, case 2:

1 LDHI R15, #0

2 JAL #42

3 ADDI R15, #-1, R15

4 NOP

5 HALT

In this case, R15, the link register, is changed in instruction #1, but JAL (jump and link) in instruction #2 would overwrite R15 with the return PC (after 2 delay slots, thus #5). R15 is then revised again by instruction #3 before taking the branch. Thus, when the called function returns via JR R15, execution starts at #4, rather than the normal behavior, 2 delay slots past the JAL, at #5.

9.6.3 Jump and Link (JAL, JALR)

Instruction name	**JAL**
Functionality	Jump and link (call)
Assembly format	JAL imm20_addr
Type and bit width	20-bit signed immediate
Predication	not available
Source options	not available
Destination options	not available (implicit: PC and LR)
Additional options	not available
Intrinsics/operator	not available
Additional details	Jump and link (call) relative immediate address. There're 2 delay slots. Immediate value is calculated as the PC offset from the 2nd delay slot to the destination. PC after the delay slot is written to the link register R15. This is where a subsequent JR R15 should jump to when returning from the called function.

Instruction name	**JALR**
Functionality	Jump and link register (call)
Assembly format	JALR Raddr
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	not available (implicit: PC and LR)
Additional options	not available

Intrinsics/operator	not available
Additional details	Jump and link (call) absolute address in register. There're 2 delay slots. Immediate value is calculated as the PC offset from the 2nd delay slot to the destination. PC after the delay slots is written to the link register R15. This is where a subsequent JR R15 should jump to when returning from the called function.

9.6.4 Jump (J, JR)

Instruction name	**J**
Functionality	Jump to immediate
Assembly format	J imm20_addr
Type and bit width	20-bit signed immediate
Predication	not available
Source options	not available
Destination options	not available
Additional options	not available
Intrinsics/operator	not available
Additional details	Jump to relative immediate address. There're 2 delay slots; one execution packet immediately following the jump would be executed before the jump takes place. Immediate value is calculated as the PC offset from the 2nd delay slot to the destination.

Instruction name	**JR**
Functionality	Jump to register
Assembly format	JR Raddr
Type and bit width	32-bit absolute address
Predication	not available
Source options	scalar register
Destination options	not available
Additional options	not available
Intrinsics/operator	not available
Additional details	Jump to absolute address in register. There are 2 delay slots.

9.6.5 Conditional Branch (BEQZ, BNEZ)

Instruction name	BEQZ
Functionality	Branch if zero
Assembly format	BEQZ Rsrc, imm14_addr
Type and bit width	14-bit signed immediate
Predication	not available
Source options	scalar register
Destination options	not available
Additional options	not available
Intrinsics/operator	not available
Additional details	Branch if Rsrc is zero to relative immediate address. There are 2 delay slots. Immediate value is calculated as the PC offset from the 2nd delay slot to the destination.

Instruction name	BNEZ
Functionality	Branch if not zero
Assembly format	BNEZ Rsrc, imm14_addr
Type and bit width	14-bit signed immediate
Predication	not available
Source options	scalar register
Destination options	not available
Additional options	not available
Intrinsics/operator	not available
Additional details	Branch if Rsrc is not zero to relative immediate address. There are 2 delay slots. Immediate value is calculated as the PC offset from the 2nd delay slot to the destination.

9.6.6 Software Break Point (SWBRK)

Instruction name	SWBRK
Functionality	Software break point
Assembly format	SWBRK
Type and bit width	not applicable
Predication	not available
Source options	not available
Destination options	not available
Additional options	not available

Instruction name	SWBRK
Intrinsics/operator	not available
Additional details	Upon executing this, VPU transitions into debug state. Only the debug controller can transition VPU back to active state. SWBRK should not be placed in any branch or hardware loop delay slots.

9.6.7 Hardware Zero-Overhead Loop (RPT)

Instruction name	RPT
Functionality	Hardware loop
Assembly format	RPT Rsrc, imm16
Type and bit width	Rsrc: 32-bit unsigned iteration count Imm16: 16-bit unsigned PC offset
Predication	not available
Source options	scalar register
Destination options	not available
Additional options	not available
Intrinsics/operator	not available
Additional details	Hardware zero-overhead loop, with Rsrc indicating number of iterations. There are 2 delay slots. The immediate field encodes loop size, which is the PC difference between the 2nd delay slot packet (very next packet is beginning of loop) and the last packet of the loop. Rsrc is checked at the end of the loop body, so loop is iterated at least one time. Loop with Rsrc = 0 will be executed one time (same behavior as Rsrc = 1).

Instruction name	CLR_HWLP
Functionality	Clear hardware loop registers
Assembly format	CLR_HWLP
Type and bit width	not applicable
Predication	not available
Source options	not available
Destination options	not available
Additional options	not available
Intrinsics/operator	`void clr_hwlp();`
Additional details	Initialize LF = -1 (2-bit binary 11), LC[0..1] = 1, LS[0..1] = 0, LE[0..1] = 0. Should be included in each task starting code to clear hardware loop registers for the new task. Should not be placed: > in the same packet as RPT > in RPT delay slots

Instruction name	CLR_HWLP
	> in first 2 packets of loop body > in last 2 packets of loop body

9.6.8 General Purpose Output (GPO_*)

The following instructions are available for GPO feature:

- GPO_SETLI
- GPO_SETHI
- GPO_CLRLI
- GPO_CLRHI
- GPO_SET
- GPO_CLR
- GPO_RD
- GPO_WR

GPO set/clear low/high immediate are used to set or clear a number of GPO bits at the same time, all in the lower 16 bits or upper 16 bits and known at compile time. For example, gpo_clrh(5) would map to “GPO_CLRHI #5” to clear GPO[18] and GPO[16], while leaving all other GPO pins unchanged.

GPO set/clear are used to set or clear a number of GPO bits at the same time, either not all in lower/upper 16 bits or unknown at compile time. The set/clear bit mask value is supplied by a scalar register. For example, gpo_set(val) would map to “GPO_SET R4” (assuming variable val is allocated to R4), to set GPO pins where bits of val are one, leaving all other GPO pins unchanged.

GPO read/write are used to replace (or not replace) a number of GPO bits at the same time, allowing any binary transition (0 → 0, 0 → 1, 1 → 0, 1 → 1) in each bit. For example, to replace GPO[7:4] with a 4-bit value in val, one would code:

```
temp = gpo_rd();
temp &= 0xFFFF_FF0F;
temp |= val << 4;
gpo_wr(temp);
```

which would map to (assuming val is allocated to R6):

```
GPO_RD R4
LHI #0xFFFF, R5
ORI R5, #0xFF0F, R5
AND R4, R5, R4
SLLI R6, #4, R5
OR R4, R5, R4
GPO_WR R4
```

Instruction name	GPO_SETLI
Functionality	General purpose output set low immediate

Instruction name	GPO_SETLI
Assembly format	GPO_SETLI imm16
Type and bit width	16-bit unsigned immediate
Predication	not available
Source options	not available
Destination options	not available
Additional options	not available
Intrinsics/operator	`void gpo_setl(unsigned short imm);`
Additional details	Set lower 16-bit of GPO according to immediate. When a bit of the immediate is on, the corresponding bit of GPO is set. The remaining GPO bits are left unchanged. For example, GPO_SETLI #0x11 would set bits 4 and 0 of GPO.

Instruction name	GPO_SETHI
Functionality	General purpose output set high immediate
Assembly format	GPO_SETHI imm16
Type and bit width	16-bit unsigned immediate
Predication	not available
Source options	not available
Destination options	not available
Additional options	not available
Intrinsics/operator	`void gpo_seth(unsigned short imm);`
Additional details	Set upper 16-bit of GPO according to immediate. When a bit of the immediate is on, the corresponding bit in upper 16 bits of GPO is set. The remaining GPO bits are left unchanged. For example, GPO_SETHI #0x11 would set bits 20 and 16 of GPO.

Instruction name	GPO_CLRLI
Functionality	General purpose output clear low immediate
Assembly format	GPO_CLRLI imm16
Type and bit width	16-bit unsigned immediate
Predication	not available
Source options	not available
Destination options	not available
Additional options	not available
Intrinsics/operator	`void gpo_clrl(unsigned short imm);`
Additional details	Clear lower 16-bit of GPO according to immediate. When a bit of the immediate is on, the corresponding bit of GPO is cleared. The remaining GPO bits are left unchanged. For example, GPO_CLRLI #0x11 would clear bits 4 and 0 of GPO.

Instruction name	GPO_CLRHI
Functionality	General purpose output clear high immediate
Assembly format	GPO_CLRHI imm16
Type and bit width	16-bit unsigned immediate
Predication	not available
Source options	not available
Destination options	not available
Additional options	not available
Intrinsics/operator	`void gpo_clrh(unsigned short imm);`
Additional details	Clear upper 16-bit of GPO according to immediate. When a bit of the immediate is on, the corresponding bit in upper 16 bits of GPO is cleared. The remaining GPO bits are left unchanged. For example, GPO_CLRHI #0x11 would clear bits 20 and 16 of GPO.

Instruction name	GPO_SET
Functionality	General purpose output set register
Assembly format	GPO_SET Rsrc
Type and bit width	32-bit unsigned
Predication	not available
Source options	scalar register
Destination options	not available
Additional options	not available
Intrinsics/operator	`void gpo_set(unsigned int);`
Additional details	Set 32-bit GPO according to register source. When a bit of the scalar register is on, the corresponding bit of GPO is set. The remaining GPO bits are left unchanged. For example, GPO_SET R1 with R1 = 0x11 would set bits 4 and 0 of GPO.

Instruction name	GPO_CLR
Functionality	General purpose output clear register
Assembly format	GPO_CLR Rsrc
Type and bit width	32-bit unsigned
Predication	not available
Source options	scalar register
Destination options	not available
Additional options	not available
Intrinsics/operator	`void gpo_clr(unsigned int);`
Additional details	Clear 32-bit GPO according to register source. When a bit of the scalar register is on, the corresponding bit of GPO is cleared. The remaining GPO bits are left unchanged. For example, GPO_CLR R1 with R1 = 0x11 would clear bits 4 and 0 of GPO.

Instruction name	GPO_RD
Functionality	General purpose output read
Assembly format	GPO_RD Rdst
Type and bit width	32-bit unsigned
Predication	not available
Source options	not available
Destination options	scalar register
Additional options	not available
Intrinsics/operator	`unsigned int gpo_rd();`
Additional details	Copy 32-bit GPO to destination register Rdst.

Instruction name	GPO_WR
Functionality	General purpose output write
Assembly format	GPO_WR Rsrc
Type and bit width	32-bit unsigned
Predication	not available
Source options	scalar register
Destination options	not available
Additional options	not available
Intrinsics/operator	`void gpo_wr(unsigned int var);`
Additional details	Copy 32-bit source register Rsrc to GPO.

9.6.9 General Purpose Input (GPI_RD)

Instruction name	GPI_RD
Functionality	General purpose input read
Assembly format	GPI_RD Rdst
Type and bit width	32-bit unsigned
Predication	not available
Source options	not available
Destination options	scalar register
Additional options	not available
Intrinsics/operator	`unsigned int gpi_rd();`
Additional details	Sample 32-bit GPI into destination register Rdst.

9.6.10 Wait for GPI Event (WFE_GPI)

Instruction name	WFE_GPI
Functionality	Wait for GPI pattern

Instruction name	WFE_GPI
Assembly format	WFE_GPI Rsrc1, Rsrc2
Type and bit width	32-bit unsigned
Predication	not available
Source options	Two scalar registers
Destination options	not available
Additional options	not available
Intrinsics/operator	`void wfe_gpi(unsigned int mask, unsigned int match);`
Additional details	Wait until (GPI & Rsrc1) == Rsrc2, Rsrc1 being the bit mask and Rsrc2 being the match pattern. For example, WFE_GPI R1, R2, with R1 = 3 and R2 = 1 would wait for GPI[0] = 1 and GPI[1] = 0 before proceeding to the next execution packet. Should not be placed in a branch delay slot.

9.6.11 Wait for R5 Event (WFE_R5)

Instruction name	WFE_R5
Functionality	Wait for R5 event
Assembly format	WFE_R5
Type and bit width	not applicable
Predication	not available
Source options	not available
Destination options	not available
Additional options	not available
Intrinsics/operator	`void wfe_r5();`
Additional details	Transition into low-power WFE_R5 state until R5 writes R5_vpu_start to dispatch next task. Should be included as the last statement in every task's exit code. Should not be placed in a loop. Should not be placed in a branch delay slot.

9.6.12 Signal R5 (SIG_R5)

Instruction name	SIG_R5
Functionality	Signal R5
Assembly format	SIG_R5 Rsrc
Type and bit width	not applicable
Predication	not available
Source options	scalar register
Destination options	not available

Instruction name	SIG_R5
Additional options	not available
Intrinsics/operator	`void sig_r5(unsigned int data);`
Additional details	Send software interrupt to R5; Rsrc carries a software-defined 32-bit data to write to a VPU config register, which R5 interrupt service routine can read.

9.6.13 Performance Counter (ENABLE/RD_TSC)

Instruction name	ENABLE_TSC
Functionality	Enable performance counter
Assembly format	ENABLE_TSC
Type and bit width	not applicable
Predication	not available
Source options	not available
Destination options	not available
Additional options	not available
Intrinsics/operator	`void enable_TSC();`
Additional details	Once enabled, the 64-bit counter increments in Active state (and not increment in Reset, Debug, WFE_R5, WFE_GPI, Halted, Error-Halted states). Once enabled, subsequent ENABLE TSC would be ignored. Though the counter is called TSC, it does not count in real-time scale, but in VPU clock cycles.

Instruction name	RD_TSC
Functionality	Read performance counter
Assembly format	RD_TSCL Rdst RD_TSCH Rdst
Type and bit width	not applicable
Predication	not available
Source options	not available
Destination options	scalar register
Additional options	not available

Intrinsics/operator	`unsigned long long read_TSC();//read lower/upper parts together`
	`unsigned int read_TSCL();  // read just lower part`
	`unsigned int read_TSCH();  // read just upper part`

Additional details	Copy TSC lower/upper 32-bit to Rdst. It's available on both S0 and S1 slots, and ideally should be schedule in both S0 and S1 to copy lower/upper parts to avoid skewed copy introducing inconsistency. **Intrinsic functions are supported to read just lower or upper part, or both parts. Intrinsic function reading both parts are implemented such that,**

Instruction name	RD_TSC

	RD_TSCL and RD_TSCH are executed in the same execution packet and with no other fused operations to avoid potential inconsistency.
	`unsigned long long start_time = read_TSC();`
	
	`// loop code`
	
	`unsigned long long end_time = read_TSC();`
	`printf("Loop XXX cycle count = %ld \n", end_time – start_time);`

9.6.14 Floating-Point Invalid Flag

Instruction name	MOV INV-R
Functionality	Move floating-point invalid flag to register
Assembly format	MOV INV, Rdst
Type and bit width	1-bit
Predication	not available
Source options	not available
Destination options	scalar register
Additional options	not available
Intrinsics/operator	`int invalid_flag();`
Additional details	Move floating-point invalid flag to scalar register. After the move, the scalar register becomes either 0 or 1. The invalid flag is set when any input or output floating-point value is NaN (not a number).

Instruction name	MOV R-INV
Functionality	Move register to floating-point invalid flag
Assembly format	MOV Rsrc, INV
Type and bit width	not applicable
Predication	not available
Source options	scalar register
Destination options	not available
Additional options	not available
Intrinsics/operator	`void set_invalid_flag(int var);`
Additional details	Move scalar register to floating-point invalid flag. Invalid flag is cleared if the scalar register is zero and set if the scalar register is non-zero.

9.6.15 OCD Load/Store

Instruction name	OCD_LD
Functionality	OCD (on-chip debug) load
Assembly format	OCD_LD PC OCD_LD GPO OCD_LD SES (shadow execution state)
Type and bit width	32-bit unsigned
Predication	not available
Source options	dedicated ocd_data register
Destination options	PC or GPO
Additional options	not available
Intrinsics/operator	not available
Additional details	Copy from ocd_data dedicated debug register to PC, GPO, or SES, for debug. Only take effect in debug mode; treated like NOP otherwise.

Instruction name	OCD_ST
Functionality	OCD (on-chip debug) store
Assembly format	OCD_ST PC OCD_ST GPI OCD_ST GPO OCD_ST SES (shadow execution state)
Type and bit width	32-bit unsigned
Predication	not available
Source options	PC, GPI or GPO
Destination options	dedicated ocd_data register
Additional options	not available
Intrinsics/operator	not available
Additional details	Copy from PC, GPI, GPO, SES to ocd_data dedicated debug register.

9.6.16 Configure VMEM Superbanks (CFG_VMEM_SBA/B/C)

Instruction name	CFG_VMEM_SBA/B/C
Functionality	Cofigure VMEM Superbanks
Assembly format	CFG_VMEM_SBA/B/C Rsrc
Type and bit width	not applicable
Predication	not available
Source options	32-bit scalar register

Instruction name	CFG_VMEM_SBA/B/C
Destination options	not available
Additional options	not available

Intrinsics/operator	`void cfg_vmem_sba(int data);`
	`void cfg_vmem_sbb(int data);`
	`void cfg_vmem_sbc(int data);`

Additional details	Write VMEM superbank A/B/C configuration data, 32-bit for each superbank. Bit 0: Load cache enable (0 = disable, 1 = enable) Bits 1 ~ 31: Reserved Reset value = 0 For example, cfg_vmem_sba(0) disables load cache in Superbank A, and cfg_vmem_sbb(1) enables load cache in Superbank B.

Instruction name	RD_CFG_VMEM_SBA/B/C
Functionality	Read cofiguration of VMEM Superbanks
Assembly format	RD_CFG_VMEM_SBA/B/C Rdst
Type and bit width	not applicable
Predication	not available
Source options	not available
Destination options	32-bit scalar register
Additional options	not available

Intrinsics/operator	`int rd_cfg_vmem_sba();`
	`int rd_cfg_vmem_sbb();`
	`int rd_cfg_vmem_sbc();`

Additional details	Read VMEM superbank A/B/C configuration data, 32-bit for each superbank and return in destination register. Bit 0: Load cache enable (0 = disable, 1 = enable) Bits 1 ~ 31: Reserved Reset value = 0

9.6.17 Coprocessor Control/Status Register Load/Store

Instruction name	CPST
Functionality	Coprocessor store
Assembly format	CPST Rsrc, Rdaddr CPST Rsrc, #imm12
Type and bit width	32-bit
Predication	not available
Source options	32-bit scalar register

Instruction name	CPST
Destination options	Coprocessor address supplied by bits 13:2 of Rdaddr or 12-bit immediate word address
Additional options	not available
Intrinsics/operator	`void cp_store(unsigned int src, int daddr);`
Additional details	Available in M0 slot

Instruction name	CPLD
Functionality	Coprocessor load
Assembly format	CPLD Rsaddr, Rdst CPLD #imm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	Coprocessor address supplied by bits 13:2 of Rdaddr or 12-bit immediate word address
Destination options	32-bit scalar register
Additional options	not available
Intrinsics/operator	`unsigned int cp_load(int saddr);`
Additional details	Available in M0 slot

9.6.18 Memory Fence

Instruction name	MemFence
Functionality	Memory fence
Assembly format	MemFence
Type and bit width	none
Predication	not available
Source options	none
Destination options	none
Additional options	not available
Intrinsics/operator	`void mem_fence();`
Additional details	Available in M0 slot Stall appropriately for any preceding memory write (scalar/vector store, histogram, VAST) to commit to memory before the execution packet where MemFence resides can execute, to ensure memory coherency and prevent RAW data hazards. See Memory Coherency for comparison between MemFence instruction and chess_memory_fence() pragma.

9.7 Scalar ALU Instructions

The scalar unit supports various common scalar arithmetic and logic operations in the S0 and S1 slots.

9.7.1 ALU RRR Instructions

9.7.1.1 Instruction Summary

These RRR (register-register-register) instructions have two source registers and one destination register. Unless otherwise noted, these are 32-bit operations.

Table 22. Scalar ALU RRR instructions

Function	Assembly Format	Comments
Add	ADD Rsrc1, Rsrc2, Rdst	
Subtract	SUB Rsrc1, Rsrc2, Rdst	
Multiply	MUL Rsrc1, Rsrc2, Rdst	
And	AND Rsrc1, Rsrc2, Rdst	Bitwise and
Or	OR Rsrc1, Rsrc2, Rdst	Bitwise or
Exclusive or	XOR Rsrc1, Rsrc2, Rdst	Bitwise exclusive or
Shift left logical	SLL Rsrc1, Rsrc2, Rdst	Rsrc2 carries the shift count, also works for shift left arithmetic. 6 LSBs of Rsrc2 are read as unsigned bit count; other bits are ignored.
Shift right logical	SRL Rsrc1, Rsrc2, Rdst	Rsrc2 carries the shift count. 6 LSBs of Rsrc2 are read as unsigned bit count; other bits are ignored.
Shift right arithmetic	SRA Rsrc1, Rsrc2, Rdst	Rsrc2 carries the shift count. 6 LSBs of Rsrc2 are read as unsigned bit count; other bits are ignored.
Sign extend	SXTD Rsrc1, Rsrc2, Rdst	Rsrc2 carries the bit width of Rsrc1 we want to sign extend from. 6 LSBs of Rsrc2 are read as unsigned bit width; other bits are ignored. When Rsrc2[5:0] is between 1 and 32, VPU does sh = 32 – Rsrc2[5:0]; Rdst = (Rsrc1 << sh) >> sh; Otherwise (0 or > 32), Rdst = 0.
Zero extend	ZXTD Rsrc1, Rsrc2, Rdst	Rsrc2 carries the bit width of Rsrc1 we want to zero extend from. 6 LSBs of Rsrc2 are read as unsigned bit width; other bits are ignored. When Rsrc2[5:0] is between 1 and 32, VPU does sh = 32 – Rsrc2[5:0];

Function	Assembly Format	Comments
		Rdst = (((unsigned) Rsrc1) << sh) >> sh; Otherwise (0 or > 32), Rdst = 0.
Compare equal	CMPEQ Rsrc1, Rsrc2, Rdst	
Compare not equal	CMPNE Rsrc1, Rsrc2, Rdst	
Compare greater than or equal	CMPGE Rsrc1, Rsrc2, Rdst	
Compare greater than or equal unsigned	CMPGEU Rsrc1, Rsrc2, Rdst	
Compare greater than	CMPGT Rsrc1, Rsrc2, Rdst	
Compare greater than unsigned	CMPGTU Rsrc1, Rsrc2, Rdst	
Compare less than or equal	CMPLE Rsrc1, Rsrc2, Rdst	
Compare less than or equal unsigned	CMPLEU Rsrc1, Rsrc2, Rdst	
Compare less than	CMPLT Rsrc1, Rsrc2, Rdst	
Compare less than unsigned	CMPLTU Rsrc1, Rsrc2, Rdst	
Modular increment	MODINC Rsrc1, Rsrc2, Rdst	Modular increment: Rdst = (Rsrc2 == Rsrc1) ? 0 : (Rsrc2 + 1); For example, with R4 = 3, R5 = 0, repeated execution of MODINC R4, R5, R5 results in R5 = 1, 2, 3, 0, 1, …
Min	MIN Rsrc1, Rsrc2, Rdst	
Min unsigned	MINU Rsrc1, Rsrc2, Rdst	
Max	MAX Rsrc1, Rsrc2, Rdst	
Max unsigned	MAXU Rsrc1, Rsrc2, Rdst	

9.7.1.2 ADD

Instruction name	ADD
Functionality	Add
Assembly format	ADD Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int operator+(int src1, int src2);`

Instruction name	ADD
Additional details	

9.7.1.3 SUB

Instruction name	SUB
Functionality	Subtract
Assembly format	SUB Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int operator-(int src1, int src2);`
Additional details	

9.7.1.4 MUL

Instruction name	MUL
Functionality	Multiply
Assembly format	MUL Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int operator*(int src1, int src2);`
Additional details	

9.7.1.5 AND

Instruction name	AND
Functionality	Bitwise and
Assembly format	AND Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register

Instruction name	AND
Additional options	
Intrinsics/operator	`int operator&(int src1, int src2);`
Additional details	

9.7.1.6 OR

Instruction name	OR
Functionality	Bitwise or
Assembly format	OR Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int operator|(int src1, int src2);`
Additional details	

9.7.1.7 XOR

Instruction name	XOR
Functionality	Bitwise exclusive or
Assembly format	XOR Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int operator^(int src1, int src2);`
Additional details	

9.7.1.8 SLL

Instruction name	SLL
Functionality	Shift left
Assembly format	SLL Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available

Instruction name	SLL
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int operator<<(int src1, int src2);` `unsigned int operator<<(unsigned int src1, int src2);`
Additional details	Rsrc2 carries the shift count, also works for shift left arithmetic. 6 LSBs of Rsrc2 are read as unsigned bit count; other bits are ignored.

9.7.1.9 SRL

Instruction name	SRL
Functionality	Shift right logical
Assembly format	SRL Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`unsigned int operator>>(unsigned int src1, int src2);`
Additional details	Rsrc2 carries the shift count. 6 LSBs of Rsrc2 are read as unsigned bit count; other bits are ignored. Zeroes are shifted into the most significant bits (logical vs arithmetic).

9.7.1.10 SRA

Instruction name	SRA
Functionality	Shift right arithmetic
Assembly format	SRA Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int operator>>(int src1, int src2);`
Additional details	Rsrc2 carries the shift count. 6 LSBs of Rsrc2 are read as unsigned bit count; other bits are ignored. Source 1 sign bit is into the most significant bits (arithmetic vs logic).

9.7.1.11 SXTD

Instruction name	SXTD
Functionality	Sign extend
Assembly format	SXTD Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int ext(int src1, int src2);`
Additional details	Rsrc2 carries the bit width of Rsrc1 we want to sign extend from. 6 LSBs of Rsrc2 are read as unsigned bit width; other bits are ignored. When Rsrc2[5:0] is between 1 and 32, VPU does: sh = 32 – Rsrc2[5:0]; Rdst = (Rsrc1 << sh) >> sh; Otherwise (0 or > 32), Rdst = 0. Examples: src1 = 0xF0 with src2 = 6 would take the lower 6 bits of src1, 0x30, sign-extend it to 0xFFFF_FFF0, and copy to dst. src1 = 0xF0 with src2 = 4 would take the lower 4 bits of src1, 0, sign-extend it to 0 and copy to dst.

9.7.1.12 ZXTD

Instruction name	ZXTD
Functionality	Zero extend
Assembly format	ZXTD Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int extu(int src1, int src2);`
Additional details	Rsrc2 carries the bit width of Rsrc1 we want to zero extend from. 6 LSBs of Rsrc2 are read as unsigned bit width; other bits are ignored. When Rsrc2[5:0] is between 1 and 32, VPU does: sh = 32 – Rsrc2[5:0]; Rdst = (((unsigned) Rsrc1) << sh) >> sh; Otherwise (0 or > 32), Rdst = 0. Examples:

Instruction name	ZXTD
	src1 = 0xF0 with src2 = 6 would take the lower 6 bits of src1, 0x30, zero-extend it to 0x30, and copy to dst. src1 = 0xF0 with src2 = 4 would take the lower 4 bits of src1, 0, zero-extend it to 0 and copy to dst.

9.7.1.13 CMPEQ

Instruction name	CMPEQ
Functionality	Compare equal
Assembly format	CMPEQ Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator==(int src1, int src2);`
Additional details	

9.7.1.14 CMPNE

Instruction name	CMPNE
Functionality	Compare not equal
Assembly format	CMPNE Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator!=(int src1, int src2);`
Additional details	

9.7.1.15 CMPGE

Instruction name	CMPGE
Functionality	Compare greater or equal
Assembly format	CMPGE Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available

Instruction name	CMPGE
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator>=(int src1, int src2);`
Additional details	

9.7.1.16 CMPGEU

Instruction name	CMPGEU
Functionality	Compare greater or equal unsigned
Assembly format	CMPGEU Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator>=(unsigned int src1, unsigned int src2);`
Additional details	

9.7.1.17 CMPGT

Instruction name	CMPGT
Functionality	Compare greater than
Assembly format	CMPGT Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator>(int src1, int src2);`
Additional details	

9.7.1.18 CMPGTU

Instruction name	CMPGTU
Functionality	Compare greater than unsigned
Assembly format	CMPGTU Rsrc1, Rsrc2, Rdst

Instruction name	CMPGTU
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator>(unsigned int src1, unsigned int src2);`
Additional details	

9.7.1.19 CMPLE

Instruction name	CMPLE
Functionality	Compare less or equal
Assembly format	CMPLE Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator<=(int src1, int src2);`
Additional details	

9.7.1.20 CMPLEU

Instruction name	CMPLEU
Functionality	Compare less or equal unsigned
Assembly format	CMPLEU Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator<=(unsigned int src1, unsigned int src2);`
Additional details	

9.7.1.21 CMPLT

Instruction name	CMPLT
Functionality	Compare less than
Assembly format	CMPLT Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator<(int src1, int src2);`
Additional details	

9.7.1.22 CMPLTU

Instruction name	CMPLTU
Functionality	Compare less than unsigned
Assembly format	CMPLTU Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator<(unsigned int src1, unsigned int src2);`
Additional details	

9.7.1.23 MODINC

Instruction name	MODINC
Functionality	Modular increment
Assembly format	MODINC Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	

Intrinsics/operator	`int dst = mod_inc(int src2, int src1);`
	`unsigned int dst = mod_inc(unsigned int src2, unsigned int src1);`
	`// note change in order vs assembly, src2 is the counter,`
	`// src1 is the max value`

Instruction name	MODINC
Additional details	Modular increment: Rdst = (Rsrc2 == Rsrc1) ? 0 : (Rsrc2 + 1); For example, with R4 = 3, R5 = 0, repeated execution of MODINC R4, R5, R5 results in R5 = 1, 2, 3, 0, 1, …

9.7.1.24 MIN

Instruction name	MIN
Functionality	Minimal
Assembly format	MIN Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int min(int src1, int src2);`
Additional details	

9.7.1.25 MINU

Instruction name	MINU
Functionality	Minimal unsigned
Assembly format	MINU Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit unsigned
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`unsigned int min(unsigned int src1, unsigned int src2);`
Additional details	

9.7.1.26 MAX

Instruction name	MAX
Functionality	Maximal
Assembly format	MAX Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit

Instruction name	MAX
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int max(int src1, int src2);`
Additional details	

9.7.1.27 MAXU

Instruction name	MAXU
Functionality	Maximal unsigned
Assembly format	MAXU Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit unsigned
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`unsigned int max(unsigned int src1, unsigned int src2);`
Additional details	

9.7.2 ALU RIR Instructions

9.7.2.1 Instruction Summary

These RIR (register-immediate-register) instructions have one source register, one 12-bit immediate, and one destination register. The immediate operand can be sign-extended (where designated as Imm12) or zero-padded (where designated as UImm12).

Table 23. Scalar ALU RIR instructions

Function	Assembly Format	Comments
Add	ADDI Rsrc1, Imm12, Rdst	
Add	ADDUI Rsrc1, UImm12, Rdst	
Subtract	SUBI Rsrc1, Imm12, Rdst	
Subtract	SUBUI Rsrc1, UImm12, Rdst	
And	ANDI Rsrc1, UImm12, Rdst	
Exclusive or	XORI Rsrc1, UImm12, Rdst	

Function	Assembly Format	Comments
Shift left logical	SLLI Rsrc1, UImm12, Rdst	Immediate carries the shift count, also works for shift left arithmetic. 6 LSBs of immediate are read as unsigned bit count; other bits are ignored.
Shift right logical	SRLI Rsrc1, UImm12, Rdst	Immediate carries the shift count. 6 LSBs of immediate are read as unsigned bit count; other bits are ignored.
Shift right arithmetic	SRAI Rsrc1, UImm12, Rdst	Immediate carries the shift count. 6 LSBs of immediate are read as unsigned bit count; other bits are ignored.
Sign extend	SXTDI Rsrc1, UImm12, Rdst	Immediate carries the bit width of Rsrc1 we want to sign extend from. 6 LSBs of Immediate are read as unsigned bit width; other bits are ignored. When Imm[5:0] is between 1 and 32, VPU does: sh = 32 – Imm[5:0]; Rdst = (Rsrc1 << sh) >> sh; Otherwise (0 or > 32), Rdst = 0.
Zero extend	ZXTDI Rsrc1, UImm12, Rdst	Immediate carries the bit width of Rsrc1 we want to zero extend from. 6 LSBs of Rsrc2 are read as unsigned bit width; other bits are ignored. When Imm[5:0] is between 1 and 32, VPU does: sh = 32 – Imm[5:0]; Rdst = (((unsigned) Rsrc1) << sh) >> sh; Otherwise (0 or > 32), Rdst = 0.
Compare equal	CMPEQI Rsrc1, Imm12, Rdst	
Compare not equal	CMPNEI Rsrc1, Imm12, Rdst	
Compare greater than or equal	CMPGEI Rsrc1, Imm12, Rdst	
Compare greater than or equal unsigned	CMPGEUI Rsrc1,UImm12, Rdst	
Compare greater than	CMPGTI Rsrc1, Imm12, Rdst	
Compare greater than unsigned	CMPGTUI Rsrc1, UImm12, Rdst	
Compare less than or equal	CMPLEI Rsrc1, Imm12, Rdst	
Compare less than or equal unsigned	CMPLEU Rsrc1, UImm12, Rdst	
Compare less than	CMPLTI Rsrc1, Imm12, Rdst	
Compare less than unsigned	CMPLTUI Rsrc1, UImm12, Rdst	
Min	MINI Rsrc1, Imm12, Rdst	
Min unsigned	MINUI Rsrc1, UImm12, Rdst	

Function	Assembly Format	Comments
Max	MAXI Rsrc1, Imm12, Rdst	
Max unsigned	MAXUI Rsrc1, UImm12, Rdst	

9.7.2.2 ADDI

Instruction name	ADDI
Functionality	Add immediate
Assembly format	ADDI Rsrc1, Imm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int operator+(int src1, int imm12);`
Additional details	Imm12 is signed-extended before the operation.

9.7.2.3 ADDUI

Instruction name	ADDUI
Functionality	Add unsigned immediate
Assembly format	ADDUI Rsrc1, UImm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int operator+(int src1, int uimm12);`
Additional details	UImm12 is zero-extended before the operation.

9.7.2.4 SUBI

Instruction name	SUBI
Functionality	Subtract immediate
Assembly format	SUBI Rsrc1, Imm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register

Instruction name	SUBI
Additional options	

Intrinsics/operator	`// Intrinsic functions are not needed for this instruction.`
	`// Compiler has freedom to leverage this and/or other`
	`// instructions to correctly implement expressions`
	`// involving scalar subtraction operation.`

Additional details	Imm12 is sign-extended before the operation.

9.7.2.5 SUBUI

Instruction name	SUBUI
Functionality	Subtract unsigned immediate
Assembly format	SUBUI Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	

Intrinsics/operator	`// Intrinsic functions are not needed for this instruction.`
	`// Compiler has freedom to leverage this and/or other`
	`// instructions to correctly implement expressions`
	`// involving scalar subtraction operation.`

Additional details	UImm12 is zero-extended before the operation.

9.7.2.6 ANDI

Instruction name	ANDI
Functionality	Bitwise and immediate
Assembly format	ANDI Rsrc1, UImm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int operator&(int src1, int uimm12);`
Additional details	UImm12 is zero-extended before the operation.

9.7.2.7 XORI

Instruction name	XORI
Functionality	Bitwise exclusive or immediate
Assembly format	XORI Rsrc1, UImm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int operator^(int src1, int uimm12);`
Additional details	UImm12 is zero-extended before the operation.

9.7.2.8 SLLI

Instruction name	SLLI
Functionality	Shift left immediate
Assembly format	SLLI Rsrc1, UImm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int operator<<(int src1, int uimm12);` `unsigned int operator<<(unsigned int src1, int src2);`
Additional details	Immediate carries the shift count, also works for shift left arithmetic. 6 LSBs of immediate are read as unsigned bit count; other bits are ignored.

9.7.2.9 SRLI

Instruction name	SRLI
Functionality	Shift right logical immediate
Assembly format	SRLI Rsrc1, UImm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`unsigned int operator>>(unsigned int src1, int uimm12);`
Additional details	Immediate carries the shift count.

Instruction name	SRLI
	6 LSBs of immediate are read as unsigned bit count; other bits are ignored.

9.7.2.10 SRAI

Instruction name	SRAI
Functionality	Shift right arithmetic immediate
Assembly format	SRAI Rsrc1, Ulmm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int operator>>(int src1, int uimm12);`
Additional details	Immediate carries the shift count. 6 LSBs of immediate are read as unsigned bit count; other bits are ignored.

9.7.2.11 SXTDI

Instruction name	SXTDI
Functionality	Sign extend immediate
Assembly format	SXTDI Rsrc1, Ulmm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int ext(int src1, int uimm12);`

Additional details	Immediate carries the bit width of Rsrc1 we want to sign extend from. 6 LSBs of Immediate are read as unsigned bit width; other bits are ignored.
	When Imm[5:0] is between 1 and 32, VPU does:
	`sh = 32 – Imm[5:0];`
	`Rdst = (Rsrc1 << sh) >> sh;`
	Otherwise (0 or > 32), Rdst = 0.

9.7.2.12 ZXTDI

Instruction name	ZXTDI
Functionality	Zero extend immediate
Assembly format	ZXTDI Rsrc1, Ulmm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int extu(int src1, int uimm12);`

Additional details	Immediate carries the bit width of Rsrc1 we want to zero extend from. 6 LSBs of Rsrc2 are read as unsigned bit width; other bits are ignored.
	When Imm[5:0] is between 1 and 32, VPU does:
	`sh = 32 – Imm[5:0];`
	`Rdst = (((unsigned) Rsrc1) << sh) >> sh;`
	Otherwise (0 or > 32), Rdst = 0.

9.7.2.13 CMPEQI

Instruction name	CMPEQI
Functionality	Compare equal immediate
Assembly format	CMPEQI Rsrc1, Imm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator==(int src1, int imm12);`
Additional details	Imm12 is signed-extended before the operation.

9.7.2.14 CMPNEI

Instruction name	CMPNEI
Functionality	Compare not equal immediate
Assembly format	CMPNE Rsrc1, Imm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register

Instruction name	CMPNEI
Additional options	
Intrinsics/operator	`bool operator!=(int src1, int imm12);`
Additional details	Imm12 is signed-extended before the operation.

9.7.2.15 CMPGEI

Instruction name	CMPGEI
Functionality	Compare greater or equal immediate
Assembly format	CMPGEI Rsrc1, Imm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator>=(int src1, int imm12);`
Additional details	Imm12 is signed-extended before the operation.

9.7.2.16 CMPGEUI

Instruction name	CMPGEUI
Functionality	Compare greater or equal unsigned immediate
Assembly format	CMPGEUI Rsrc1, UImm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator>=(unsigned int src1, unsigned int uimm12);`
Additional details	UImm12 is zero-extended before the operation.

9.7.2.17 CMPGTI

Instruction name	CMPGTI
Functionality	Compare greater than immediate
Assembly format	CMPGTI Rsrc1, Imm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register

Instruction name	CMPGTI
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator>(int src1, int imm12);`
Additional details	Imm12 is signed-extended before the operation.

9.7.2.18 CMPGTUI

Instruction name	CMPGTUI
Functionality	Compare greater than unsigned immediate
Assembly format	CMPGTUI Rsrc1, UImm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator>(unsigned int src1, unsigned int uimm12);`
Additional details	UImm12 is zero-extended before the operation.

9.7.2.19 CMPLEI

Instruction name	CMPLEI
Functionality	Compare less or equal immediate
Assembly format	CMPLEI Rsrc1, Imm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator<=(int src1, int imm12);`
Additional details	Imm12 is signed-extended before the operation.

9.7.2.20 CMPLEUI

Instruction name	CMPLEUI
Functionality	Compare less or equal unsigned immediate
Assembly format	CMPLEUI Rsrc1, UImm12, Rdst
Type and bit width	32-bit

Instruction name	CMPLEUI
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator<=(unsigned int src1, unsigned int uimm12);`
Additional details	UImm12 is zero-extended before the operation.

9.7.2.21 CMPLTI

Instruction name	CMPLTI
Functionality	Compare less than immediate
Assembly format	CMPLTI Rsrc1, Imm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator<(int src1, int imm12);`
Additional details	Imm12 is signed-extended before the operation.

9.7.2.22 CMPLTUI

Instruction name	CMPLTUI
Functionality	Compare less than unsigned immediate
Assembly format	CMPLTUI Rsrc1, UImm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator<(unsigned int src1, unsigned int uimm12);`
Additional details	UImm12 is zero-extended before the operation.

9.7.2.23 MINI

Instruction name	MINI
Functionality	Minimal immediate
Assembly format	MINI Rsrc1, Imm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int min(int src1, int imm12);`
Additional details	

9.7.2.24 MINUI

Instruction name	MINUI
Functionality	Minimal unsigned immediate
Assembly format	MINUI Rsrc1, Imm12, Rdst
Type and bit width	32-bit unsigned
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`unsigned int min(unsigned int src1, unsigned int uimm12);`
Additional details	

9.7.2.25 MAXI

Instruction name	MAXI
Functionality	Maximal Immediate
Assembly format	MAXI Rsrc1, Imm12, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int max(int src1, int imm12);`
Additional details	

9.7.2.26 MAXUI

Instruction name	MAXUI
Functionality	Maximal unsigned immediate
Assembly format	MAXUI Rsrc1, Imm12, Rdst
Type and bit width	32-bit unsigned
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`unsigned int max(unsigned int src1, unsigned int uimm12);`
Additional details	

9.7.3 Long Multiplication Instructions

9.7.3.1 Instruction Summary

The scalar ALU also supports long multiply, multiplication between two signed/unsigned 32-bit operands. Outcome is placed in the PL/PH special register pair.

Table 24. Scalar ALU long multiply instructions

Function	Assembly Format	Comments
Long multiply signed-signed	LMULSS Rsrc1, Rsrc2	Multiply into 64-bit product in PL:PH (dedicated product low/high registers)
Long multiply signed-unsigned	LMULSU Rsrc1, Rsrc2	Multiply into 64-bit product in PL:PH (dedicated product low/high registers)
Long multiply unsigned-unsigned	LMULUU Rsrc1, Rsrc2	Multiply into 64-bit product in PL:PH (dedicated product low/high registers)

9.7.3.2 LMULSS

Instruction name	LMULSS
Functionality	Long multiply signed-signed
Assembly format	LMULSS Rsrc1, Rsc2
Type and bit width	signed 32-bit x signed 32-bit → signed 64-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	

Instruction name	LMULSS
Intrinsics/operator	`long long mulwl_s(int src1, int src2);`
Additional details	Product is placed in PL (lower 32-bit) and PH (upper 32-bit).

9.7.3.3 LMULSU

Instruction name	LMULSU
Functionality	Long multiply signed-unsigned
Assembly format	LMULSU Rsrc1, Rsc2
Type and bit width	signed 32-bit x unsigned 32-bit → signed 64-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`long long mulwl_su(int src1, unsigned int src2);`
Additional details	Product is placed in PL (lower 32-bit) and PH (upper 32-bit).

9.7.3.4 LMULUU

Instruction name	LMULUU
Functionality	Long multiply unsigned-unsigned
Assembly format	LMULUU Rsrc1, Rsc2
Type and bit width	unsigned 32-bit x unsigned 32-bit → unsigned 64-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`unsigned long long mulwl_u(unsigned int src1, unsigned int src2);`
Additional details	Product is placed in PL (lower 32-bit) and PH (upper 32-bit).

9.7.4 Predicate Instructions

9.7.4.1 Instruction Summary

Moving between scalar register and predicate register, and modular increment/decrement on predicate register is also supported. These are used for periodic predication that enable loop collapsing.

Function	Assembly Format	Comments
Move scalar to predicate	MOVSP Rsrc, Pdst	Move scalar register to predicate register
Negate scalar to predicate	NOTSP Rsrc, Pdst	Negate scalar register to predicate register
Move predicate to scalar	MOVPS Psrc, Rdst	Move predicate register to scalar register
Move predicate	MOVP Psrc, Pdst MOVP DPsrc, DPdst	Move single/double predicate register
Modular increment	MODINC Rsrc1, Ps2d	Modular increment predicate register Ps2d. Rsrc1 conveys the max value.
Modular increment	MODINCP Rsrc1, Rs2d, Pdst	Modular increment scalar register Rs2d. Rsrc1 conveys the max value, and Pdst is set all 0 or all 1 based on Rs2d outcome being zero/non-zero
Modular increment NOT	MODINC_NOTP Rsrc1, Rs2d, Pdst	Modular increment scalar register Rs2d. Rsrc1 conveys the max value, and Pdst is set all 0 or all 1 based on Rs2d outcome being non-zero/zero, inversed w.r.t. MODINCP
Modular increment, double predicate	DPMODINCP Rsrc1, Rs2d, DPdst	Modular increment scalar register Rs2d. Rsrc1 conveys the max value, and DPdst is set all 0 or all 1 based on Rs2d outcome being zero/non-zero. Both destination predicate registers are set identically.
Modular increment NOT, double predicate	DPMODINC_NOTP Rsrc1, Rs2d, DPdst	Modular increment scalar register Rs2d. Rsrc1 conveys the max value, and DPdst is set all 0 or all 1 based on Rs2d outcome being non-zero/zero, inversed w.r.t. DPMODINCP. Both destination predicate registers are set identically.
Predicated Move	[Preg] MOV Rsrc, Rdst	Predicated scalar register move
Multiplex to predicate	MUXP Rsrc1, Rsrc2, Rsrc3, Pdst	Multiplex to predicate destination. For example, with Rsrc1 = 1, Rsrc2 = 2, Rsrc3 = 3, Pdst = (Rsrc1 != 0) ? Rsrc2 : Rsrc3, so would set Pdst to Rsrc2 = 2.

Table 25. Scalar predicate instructions

See Instruction Predication for use cases of instruction predication.

9.7.4.2 MOVSP

Instruction name	MOVSP
Functionality	Move scalar to predicate
Assembly format	MOVSP Rsrc, Pdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	predicate register
Additional options	

Intrinsics/operator	`// not needed, instantiated from an assignment statement`
	`// with destination variable mapped to a predicate register`
	`// example: int dst_predicate = int src;`

Additional details	P0 and P1 contain constant -1, and should not be a destination of MOVSP

9.7.4.3 NOTSP

Instruction name	NOTSP
Functionality	Negate (bitwise not) scalar to predicate
Assembly format	NOTSP Rsrc, Pdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	predicate register
Additional options	

Intrinsics/operator	`// Intrinsic functions are not needed for this instruction.`
	`// Compiler has freedom to leverage this and/or other`
	`// bitwise logic instructions to correctly implement`
	`// expressions involving bitwise not operations. NOTSP,`
	`// specifically, may be used when the outcome is mapped to`
	`// a predicate register.`

Additional details	P0 and P1 contain constant -1, and should not be a destination of NOTSP

9.7.4.4 MOVPS

Instruction name	MOVPS
Functionality	Move predicate to scalar
Assembly format	MOVPS Psrc, Rdst
Type and bit width	32-bit
Predication	not available
Source options	predicate register
Destination options	scalar register
Additional options	

Intrinsics/operator	`// not needed, instantiated from an assignment statement`
	`// with source variable mapped to a predicate register and`
	`// destination variable mapped to a scalar register`
	`// example: int dst = int src_predicate;`

Additional details	

9.7.4.5 MOVP

Instruction name	MOVP
Functionality	Move predicate register
Assembly format	MOVP Psrc, Pdst MOVP DPsrc, DPdst
Type and bit width	32-bit
Predication	not available
Source options	single or double predicate register
Destination options	single or double predicate register
Additional options	

Intrinsics/operator	`// not needed, instantiated from an assignment statement`
	`// with source and destination variables mapped to predicate`
	`// registers`
	`// example: int dst_predicate = int src_predicate;`

Additional details	P0 and P1 contain constant -1, and should not be a destination of MOVP

9.7.4.6 MODINC

Instruction name	MODINC
Functionality	Modular increment
Assembly format	MODINC Rsrc1, Ps2d
Type and bit width	32-bit
Predication	not available
Source options	scalar register and predicate register

Instruction name	MODINC
Destination options	predicate register
Additional options	

Intrinsics/operator	`int mod_inc(int s2d, int src1);`
	`unsigned int mod_inc(unsigned int s2d, unsigned int src1);`
	`// note the change in operand order vs assembly`
	`// s2d is the counter, src1 is the max value`

Additional details	Modular increment predicate register: Ps2d = (Ps2d == Rsrc1) ? 0 : (Ps2d + 1); For example, with R1 = 4, P2 = 0, a sequence of MODINC R1, P2 results in P2 = 1, 2, 3, 4, 0, 1, … This is useful for VMadd_CA to occasionally clear the accumulator.

9.7.4.7 MODINCP

Instruction name	MODINCP
Functionality	Modular increment predicate
Assembly format	MODINCP Rsrc1, Rs2d, Pdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register and predicate register
Additional options	

Intrinsics/operator	`int mod_inc_pred_nz(int s2d, int src1, int & pdst);`
	`unsigned int mod_inc_pred_nz(unsigned int s2d, unsigned int src1, int`
	`pdst);`
	`// Note the change in operand order compared to assembly`
	`// First argument is the modular counter input`
	`// Second argument src1 is the max counter value input`
	`// Third argument pdst is a reference argument output, and`
	`// is set -1 if the modular counter after the modular`
	`// increment is non-zero, otherwise is set 0`
	`// Return modular counter value after the increment`
	`// Typical usage:`
	`// count = mod_inc_pred_nz(count, period_mns_1, count_nz);`

Additional details	Modular increment scalar register Rs2d :
	Rs2d = (Rs2d == Rsrc1) ? 0 : (Rs2d + 1);
	Pdst = Rs2d ? -1 : 0; // set 0 or all 1s (-1)
	For example, with R1 = 4, initial R2 = 0, a sequence of
	MODINCP R1, R2, P2
	results in R2 = 1, 2, 3, 4, 0, 1, …
	P2 = -1, -1, -1, -1, 0, -1, …

9.7.4.8 MODINC_NOTP

Instruction name	MODINC_NOTP
Functionality	Modular increment not predicate
Assembly format	MODINC_NOTP Rsrc1, Rs2d, Pdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register and predicate register
Additional options	

Intrinsics/operator	`int mod_inc_pred_z(int s2d, int src1, int & pdst);`
	`unsigned int mod_inc_pred_z(unsigned int s2d, unsigned int src1, int pdst);`
	`// Note the change in operand order compared to assembly`
	`// First argument is the modular counter input`
	`// Second argument src1 is the max counter value input`
	`// Third argument pdst is a reference argument output, and`
	`// is set -1 if the modular counter after the modular`
	`// increment is 0, otherwise is set 0`
	`// Return modular counter value after the increment`
	`// Typical usage:`
	`// count = mod_inc_pred_nz(count, period_mns_1, count_z);`

Additional details	Modular increment scalar register Rs2d :
	Rs2d = (Rs2d == Rsrc1) ? 0 : (Rs2d + 1);
	Pdst = (Rs2d==0) ? -1 : 0; // set 0 or all 1s (-1)
	For example, with R1 = 4, initial R2 = 0, a sequence of
	MODINC_NOTP R1, R2, P2
	results in R2 = 1, 2, 3, 4, 0, 1, ...
	P2 = 0, 0, 0, 0, -1, 0, ...

9.7.4.9 DPMODINCP

Instruction name	DPMODINCP
Functionality	Modular increment predicate, double predicate
Assembly format	DPMODINCP Rsrc1, Rs2d, DPdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register and double predicate register
Additional options	
Intrinsics/operator	`int mod_inc_pred_nz(int s2d, int src1, dpred & pdst);`

Instruction name	DPMODINCP

	`unsigned int mod_inc_pred_nz(unsigned int s2d, unsigned int src1, dp & pdst);`
	`// note the change in operand order`
	`// s2d is the counter, src1 is the max value, pdst is`
	`// set depending on counter value after modular increment`

Additional details	Modular increment scalar register Rs2d :
	Rs2d = (Rs2d == Rsrc1) ? 0 : (Rs2d + 1);
	Pdst = Rs2d ? -1 : 0; // set 0 or all 1s (-1)
	For example, with R1 = 4, initial R2 = 0, a sequence of
	DPMODINCP R1, R2, P2:P3
	results in R2 = 1, 2, 3, 4, 0, 1, …
	P2 = P3 = -1, -1, -1, -1, 0, -1, …

9.7.4.10 DPMODINC_NOTP

Instruction name	DPMODINC_NOTP
Functionality	Modular increment not predicate, double predicate
Assembly format	DPMODINC_NOTP Rsrc1, Rs2d, DPdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register and double predicate register
Additional options	

Intrinsics/operator	`int mod_inc_pred_z(int s2d, int src1, dpred & pdst);`
	`unsigned int mod_inc_pred_z(unsigned int s2d, unsigned int src1, dpr pdst);`
	`// note the change in operand order`
	`// s2d is the counter, src1 is the max value, pdst is`
	`// set depending on counter value after modular increment`

Additional details	Modular increment scalar register Rs2d : Rs2d = (Rs2d == Rsrc1) ? 0 : (Rs2d + 1); Pdst = (Rs2d==0) ? -1 : 0; // set 0 or all 1s (-1) For example, with R1 = 4, initial R2 = 0, a sequence of DPMODINC_NOTP R1, R2, P2:P3 results in R2 = 1, 2, 3, 4, 0, 1, … P2 = P3 = 0, 0, 0, 0, -1, 0, …

9.7.4.11 Predicated MOV

Instruction name	Predicated MOV
Functionality	Predicated scalar register move
Assembly format	[Preg] MOV Rsrc, Rdst
Type and bit width	32-bit
Predication	Instruction-level predication
Source options	scalar register and predicate register
Destination options	scalar register
Additional options	

Intrinsics/operator	`// not needed, instantiated from the following code`
	`// if (preg) chess_guard {`
	`//  int dst = int src;`
	`// }`

Additional details	

9.7.4.12 MUXP

Instruction name	Multiplex to predicate
Functionality	Multiplexing with scalar sources and predicate destination
Assembly format	MUXP Rsrc1, Rsrc2, Rsrc3, Pdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	predicate register
Additional options	

Intrinsics/operator	`// Intrinsic functions are not needed for this instruction.`
	`// Compiler may leverage MUXP to implement a ternary`
	`// conditional operator when the outcome variable is mapped`
	`// to a predicate register. For example,`
	`//  int chess_storage(PA2) dst = (a0 != 0) ? a1 : a2;`

Additional details	Multiplex to predicate destination. Pdst = (Rsrc1 != 0) ? Rsrc2 : Rsrc3; For example, with R1 = 1, R2 = 2, R3 = 3, MUXP R1, R2, R3, P4 would set P4 to R2, which is 2.

9.7.5 Scalar Floating-point Instructions

9.7.5.1 Instruction Summary

Floating-point add, subtract, multiply, multiply-add, and float-to-int, int-to-float conversion instructions are available in the S0 and S1 instruction slots. Scalar registers supply the sources and destination of FP instructions.

FP multiply-add is implemented with a fused multiply-add datapath that preserves full product precision and has higher precision than separate FP multiply and FP add operations.

A sticky invalid status bit, INV, is available, for software to read, set, or clear by moving between INV and a scalar register. We have (detailed in 9.3.14):

- MOV INV-R: moving from the invalid flag to a scalar register
- MOV R-INV: moving from a scalar register to the invalid flag

It's sticky in the sense that once a floating-point instruction produces an invalid (NaN) outcome, the flag is set if it's previously clear and remains set until a MOV R-INV instruction moves zero value to the flag.

The flag can also be set by software, by a MOV R-INV instruction moving a software-calculated invalid value to the flag. This is useful for software emulation of floating-point functions (reciprocal, square root, etc.).

R5 software can configure VPU to go to error-halted mode upon the flag being set, or to just continue execution.

FP instructions output a fixed NaN encoding value of 0x7FC0_0000, which is a quiet NaN (as opposed to a signaling NaN), as invalid output. Note that this is different behavior from X86 FP NaN output, going with some NaN propagation rule with priority among inputs to propagate input NaN value to the output.

Note that there is just one invalid status bit to indicate floating-point outcome being NaN.

Table 26. Scalar floating-point instructions

Function	Assembly Format	Comments
FP add	FAdd Rsrc1, Rsrc2, Rdst	
FP subtract	FSub Rsrc1, Rsrc2, Rdst	
FP multiply	FMul Rsrc1, Rsrc2, Rdst	
FP multiply-add	FMAdd Rsrc1, Rsrc2, Rsrc3, Rdst	
FP multiply-subtract	FMSub Rsrc1, Rsrc2, Rsrc3, Rdst	
FP16 add	HFAdd Rsrc1, Rsrc2, Rdst	
FP16 subtract	HFSub Rsrc1, Rsrc2, Rdst	
FP16 multiply	HFMul Rsrc1, Rsrc2, Rdst	

Function	Assembly Format	Comments
FP16 multiply-add	HFMAdd Rsrc1, Rsrc2, Rsrc3, Rdst	
FP16 multiply-subtract	HFMSub Rsrc1, Rsrc2, Rsrc3, Rdst	
INT to FP conversion	INT_FP Rsrc, Rdst	Integer to floating-point conversion
FP to INT conversion with truncation	FP_INT_Trunc Rsrc, Rdst	Floating-point to integer conversion with truncation (consistent with C float-to-int type casting)
FP to INT conversion with rounding	FP_INT_Round Rsrc, Rdst	Floating-point to integer conversion with rounding
INT to FP16 conversion	INT_FP16 Rsrc1, Rsrc2, Rdst	Rsrc2 conveys qbit for fixed-point representation.
FP16 to INT conversion wth truncation	FP16_INT_Trunc Rsrc1, Rsrc2, Rdst	Rsrc2 conveys qbit for fixed-point representation.
FP16 to INT conversion wth rounding	FP16_INT_Round Rsrc1, Rsrc2, Rdst	Rsrc2 conveys qbit for fixed-point representation.
FP16 to FP32 conversion	FP16_FP Rsrc, Rdst	
FP32 to FP16 conversion	FP_FP16 Rsrc, Rdst	
FP compare LT	FCmpLT Rsrc1, Rsrc2, Rdst	
FP compare LE	FCmpLE Rsrc1, Rsrc2, Rdst	
FP compare GT	FCmpGT Rsrc1, Rsrc2, Rdst	
FP compare GE	FCmpGE Rsrc1, Rsrc2, Rdst	
FP compare EQ	FCmpEQ Rsrc1, Rsrc2, Rdst	
FP compare NE	FCmpNE Rsrc1, Rsrc2, Rdst	
FP16 compare LT	HFCmpLT Rsrc1, Rsrc2, Rdst	
FP16 compare LE	HFCmpLE Rsrc1, Rsrc2, Rdst	
FP16 compare GT	HFCmpGT Rsrc1, Rsrc2, Rdst	
FP16 compare GE	HFCmpGE Rsrc1, Rsrc2, Rdst	
FP16 compare EQ	HFCmpEQ Rsrc1, Rsrc2, Rdst	
FP16 compare NE	HFCmpNE Rsrc1, Rsrc2, Rdst	
FP reciprocal	FRCP Vsrc, Vdst	
FP square root	FSQRT Vsrc, Vdst	
FP reciprocal square root	FRSQ Vsrc, Vdst	
FP exponential base-2	FEXP2 Vsrc, Vdst	
FP logarithm base-2	FLOG2 Vsrc, Vdst	
FP sine	FSIN Vsrc, Vdst	
FP cosine	FCOS Vsrc, Vdst	
FP hyperbolic tangent	FTANH Rsrc, Rdst	

9.7.5.2 FAdd

Instruction name	FAdd
Functionality	Floating-point add
Assembly format	FAdd Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit float
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	

Intrinsics/operator	`float fadd(float src1, float src2); //bit-exact between ISS & Native`
	`float operator+(float src1, float src2); // NOT bit-exact between`
	`// ISS and Native`

Additional details	IEEE compliant floating-point add. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN.

9.7.5.3 FSub

Instruction name	FSub
Functionality	Floating-point subtract
Assembly format	FSub Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit float
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	

Intrinsics/operator	`float fsub(float src1, float src2); //bit-exact between ISS & Native`
	`float operator-(float src1, float src2); // NOT bit-exact between`
	`// ISS and Native`

Additional details	IEEE compliant floating-point subtract. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN.

9.7.5.4 FMul

Instruction name	FMul
Functionality	Floating-point multiply
Assembly format	FMul Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit float
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	

Intrinsics/operator	`float fmul(float src1, float src2); //bit-exact between ISS & Native`
	`float operator*(float src1, float src2); // NOT bit-exact between`
	`// ISS and Native`

Additional details	IEEE compliant floating-point multiply. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN.

9.7.5.5 FMAdd

Instruction name	FMAdd
Functionality	Floating-point multiply-add
Assembly format	FMAdd Rsrc1, Rsrc2, Rsrc3, Rdst
Type and bit width	32-bit float
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`float fmadd(float src1, float src2, float src3);`
Additional details	Performing multiply-add with IEEE compliant floating-point multiply and add. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN. Example: FMAdd R1, R2, R3, R4 would perform R4 = R1 * R2 + R3, reading the source registers R1, R2, R3 as 32-bit floating-point numbers, and producing 32-bit floating-point outcome in R4.

9.7.5.6 FMSub

Instruction name	FMSub
Functionality	Floating-point multiply-subtract
Assembly format	FMSub Rsrc1, Rsrc2, Rsrc3, Rdst
Type and bit width	32-bit float
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`float fmsub(float src1, float src2, float src3);`
Additional details	Performing IEEE compliant floating-point multiply-subtract, src3 – src1 * src2. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN. Example: FMSub R1, R2, R3, R4 would perform R4 = R3 – R1 * R2, reading the source registers R1, R2, R3 as 32-bit floating-point numbers, and producing 32-bit floating-point outcome in R4.

9.7.5.7 HFAdd

Instruction name	HFAdd
Functionality	FP16 add
Assembly format	HFAdd Rsrc1, Rsrc2, Rdst
Type and bit width	16-bit float
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`hfloat hfadd(hfloat src1, hfloat src2);` `hfloat operator+(hfloat src1, hfloat src2);`
Additional details	Least significant 16 bits of sources registers are read as FP16 numbers, FP16 addition performed, and FP16 outcome is sign-extended to 32-bit in the destination register. IEEE compliant half-precision floating-point add. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN.

9.7.5.8 HFSub

Instruction name	HFSub
Functionality	FP16 subtract
Assembly format	HFSub Rsrc1, Rsrc2, Rdst
Type and bit width	16-bit float
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`hfloat hfsub(hfloat src1, hfloat src2);` `hfloat operator-(hfloat src1, hfloat src2);`
Additional details	Least significant 16 bits of sources registers are read as FP16 numbers, FP16 subtraction performed, and FP16 outcome is sign-extended to 32-bit in the destination register. IEEE compliant half-precision floating-point subtract. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN.

9.7.5.9 HFMul

Instruction name	HFMul
Functionality	FP16 multiply
Assembly format	HFMul Rsrc1, Rsrc2, Rdst
Type and bit width	16-bit float
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`hfloat hfmul(hfloat src1, hfloat src2);` `hfloat operator*(hfloat src1, hfloat src2);`
Additional details	Least significant 16 bits of sources registers are read as FP16 numbers, FP16 multiplication performed, and FP16 outcome is sign-extended to 32-bit in the destination register. IEEE compliant half-precision floating-point multiply. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN.

9.7.5.10 HFMAdd

Instruction name	HFMAdd
Functionality	FP16 multiply-add
Assembly format	HFMAdd Rsrc1, Rsrc2, Rsrc3, Rdst
Type and bit width	16-bit float
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`hfloat hfmadd(hfloat src1, hfloat src2, hfloat src3);`
Additional details	Least significant 16 bits of sources registers are read as FP16 numbers, FP16 multiply-add src1 * src2 + src3 performed, and FP16 outcome is sign-extended to 32-bit in the destination register. Fused multiply-add is performed, preserving intermediate precision as much as possible. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN.

9.7.5.11 HFMSub

Instruction name	HFMSub
Functionality	FP16 multiply-subtract
Assembly format	HFMSub Rsrc1, Rsrc2, Rsrc3, Rdst
Type and bit width	16-bit float
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`hfloat hfmsub(hfloat src1, hfloat src2, hfloat src3);`
Additional details	Least significant 16 bits of sources registers are read as FP16 numbers, FP16 multiply-subtract src3 - src1 * src2 performed, and FP16 outcome is sign-extended to 32-bit in the destination register. Fused multiply-subtract is performed, preserving intermediate precision as much as possible. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN.

9.7.5.12 INT_FP

Instruction name	INT_FP
Functionality	Integer to floating-point conversion
Assembly format	INT_FP Rsrc, Rdst
Type and bit width	32-bit signed integer input, 32-bit float output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	

Intrinsics/operator	`float int_fp(int src); //bit-exact between ISS & Native float`
	
	`// Type casting int into float also compiles into INT_FP,`
	`// but it's not bit-exact between ISS and Native. For example,`
	`float_var = (float) int_var;`

Additional details	Note that rounding is included in this instruction's functionality. Only rounding mode supported is round to nearest, ties to even.

9.7.5.13 FP_INT_Trunc

Instruction name	FP_INT_Trunc
Functionality	Floating-point to integer conversion
Assembly format	FP_INT_Trunc Rsrc, Rdst
Type and bit width	32-bit float input, 32-bit signed integer output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	

Intrinsics/operator	`int fp_int_trunc(float src); //bit-exact between ISS & Native float`
	
	`// Type casting float into int also compiles into FP_INT_Trunc,`
	`// but it's not bit-exact between ISS and Native. For example,`
	`int_var = (int) float_var;`

Additional details	FP32 to integer conversion with truncation. For example, if input is 0x3FC0_0000 (1.5 in FP32), output is trunc(1.5) = 1 Note that - truncation is used during the conversion, consistent with C float-to-int type casting. - Both zero and minus zero maps to zero. - Infinity maps to maximal 32-bit int value (0x7FFF_FFFF). - Minus infinity maps to minimal 32-bit int value (0x8000_0000).

Instruction name	FP_INT_Trunc
	- When output value exceeds 32-bit int representation range, output is saturated between 0x8000_0000 and x7FFF_FFFF. - NaN maps to either 0x8000_0000 or 0x7FFF_FFFF, preserving the sign. - The invalid status flag is NOT set when input is NaN.

9.7.5.14 FP_INT_Round

Instruction name	FP_INT_Round
Functionality	Floating-point to integer conversion
Assembly format	FP_INT_Round Rsrc, Rdst
Type and bit width	32-bit float input, 32-bit signed integer output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int fp_int_round(float src);` `int f32_to_i32_rte(float src); // Gen-1 legacy`
Additional details	FP32 to integer conversion with rounding. For example, if input is 0x3FC0_0000 (1.5 in FP32), output is round(1.5) = 2, as 1.5 is tied between 1 and 2, so we round to 2 (even). Note that - Rounding is used during the conversion. The only rounding mode supported is round to nearest, ties to even. - Both zero and minus zero maps to zero. - Infinity maps to maximal 32-bit int value (0x7FFF_FFFF). - Minus infinity maps to minimal 32-bit int value (0x8000_0000). - When output value exceeds 32-bit int representation range, output is saturated between 0x8000_0000 and x7FFF_FFFF. - NaN maps to either 0x8000_0000 or 0x7FFF_FFFF, preserving the sign. - The invalid status flag is NOT set when input is NaN. Gen-1 legacy intrinsic function f32_to_i32() is supported. As it implements rounding implicitly, programmers are strongly encouraged to switch to Gen-2 intrinsic function fp_int_round() to avoid confusion.

9.7.5.15 INT_FP16

Instruction name	INT_FP16
Functionality	Integer to 16-bit floating-point conversion
Assembly format	INT_FP16 Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit int input, 16-bit float output
Predication	not available

Instruction name	INT_FP16
Source options	src1: scalar register src2: scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`hfloat int_fp16(int src1, int src2);`
Additional details	src2 (read as sign number and saturated to [0, 31]) conveys qbit in source fixed-point representation. dst = src1 / 2^src2. Note that rounding is included in this instruction's functionality. Only rounding mode supported is round to nearest, ties to even. 16-bit floating-point output is sign-extended into the 32-bit container. Where output absolute value falls below normal FP16 range, denormal FP16 output is generated.

9.7.5.16 FP16_INT_Trunc

Instruction name	FP16_INT_Trunc
Functionality	Floating-point to integer conversion with truncation
Assembly format	FP16_INT_Trunc Rsrc1, Rsrc2, Rdst
Type and bit width	16-bit float input, 32-bit int output
Predication	not available
Source options	src1: scalar register src2: scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int fp16_int_trunc(hfloat src1, int src2);`
Additional details	src2 (read as sign number and saturated to [0, 31]) conveys qbit in destination fixed-point representation. dst = trunc(src1 * 2^src2). 16-bit floating-point input is read from 16 LSBs of the 32-bit input. Note that - truncation is used during the conversion. - Both zero and minus zero maps to zero. - Infinity maps to maximal 32-bit int value (0x7FFF_FFFF). - Minus infinity maps to minimal 32-bit int value (0x8000_0000). - When output value trunc(src1 * 2^src2) exceeds 32-bit int representation range, output is saturated between 0x8000_0000 and x7FFF_FFFF. - NaN maps to either 0x8000_0000 or 0x7FFF_FFFF, preserving the sign. - The invalid status flag is NOT set when input is NaN. - Denormal FP16 input value is supported.

9.7.5.17 FP16_INT_Round

Instruction name	FP16_INT_Round
Functionality	Floating-point to integer conversion with rounding
Assembly format	FP16_INT_Round Rsrc1, Rsrc2, Rdst
Type and bit width	16-bit float input, 32-bit int output
Predication	not available
Source options	src1: scalar register src2: scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int fp16_int_round(hfloat src1, int src2);`
Additional details	src2 (read as sign number and saturated to [0, 31]) conveys qbit in destination fixed-point representation. dst = round(src1 * 2^src2). 16-bit floating-point input is read from 16 LSBs of the 32-bit input. Note that - Rounding is used during the conversion. The only rounding mode supported is round to nearest, ties to even. - Both zero and minus zero maps to zero. - Infinity maps to maximal 32-bit int value (0x7FFF_FFFF). - Minus infinity maps to minimal 32-bit int value (0x8000_0000). - When output value round(src1 * 2^src2) exceeds 32-bit int representation range, output is saturated between 0x8000_0000 and x7FFF_FFFF. - NaN maps to either 0x8000_0000 or 0x7FFF_FFFF, preserving the sign. - The invalid status flag is NOT set when input is NaN. - Denormal FP16 input value is supported.

9.7.5.18 FP16_FP

Instruction name	FP16_FP
Functionality	Floating-point FP16 to floating-point FP32 conversion
Assembly format	FP16_FP Rsrc, Rdst
Type and bit width	16-bit float input, 32-bit float output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`float fp16_fp(hfloat src);`
Additional details	FP16 floating-point input is read from 16 LSBs of the 32-bit source, converted to FP32 floating-point outcome, and written to 32-bit destination. Note that the invalid status flag is NOT set when input is NaN.

9.7.5.19 FP_FP16

Instruction name	FP_FP16
Functionality	Floating-point FP32 to floating-point FP16 conversion
Assembly format	FP_FP16 Rsrc, Rdst
Type and bit width	32-bit float input, 16-bit float output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`hfloat fp_fp16(float src);`
Additional details	FP32 floating-point input is read from 32-bit source, converted to FP16 floating-point outcome, sign-extended and written to 32-bit destination. Note that the invalid status flag is NOT set when input is NaN.

9.7.5.20 FCmpLT

Instruction name	FCmpLT
Functionality	Floating-point compare less than
Assembly format	FCmpLT Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit float input, 32-bit int output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator<(float src1, float src2);`
Additional details	Always return 0 or 1 and never set invalid status flag. See 6.2.4.3 for corner cases.

9.7.5.21 FCmpLE

Instruction name	FCmpLE
Functionality	Floating-point compare less than or equal to
Assembly format	FCmpLE Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit float input, 32-bit int output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	

Instruction name	FCmpLE
Intrinsics/operator	`bool operator<=(float src1, float src2);`
Additional details	Always return 0 or 1 and never set invalid status flag. See 6.2.4.3 for corner cases.

9.7.5.22 FCmpGT

Instruction name	FCmpGT
Functionality	Floating-point compare greater than
Assembly format	FCmpGT Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit float input, 32-bit int output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator>(float src1, float src2);`
Additional details	Always return 0 or 1 and never set invalid status flag. See 6.2.4.3 for corner cases.

9.7.5.23 FCmpGE

Instruction name	FCmpGE
Functionality	Floating-point compare greater than or equal to
Assembly format	FCmpGE Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit float input, 32-bit int output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator>=(float src1, float src2);`
Additional details	Always return 0 or 1 and never set invalid status flag. See 6.2.4.3 for corner cases.

9.7.5.24 FCmpEQ

Instruction name	FCmpEQ
Functionality	Floating-point compare equal
Assembly format	FCmpEQ Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit float input, 32-bit int output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator==(float src1, float src2);`
Additional details	Always return 0 or 1 and never set invalid status flag. See 6.2.4.3 for corner cases.

9.7.5.25 FCmpNE

Instruction name	FCmpNE
Functionality	Floating-point compare not equal
Assembly format	FCmpNE Rsrc1, Rsrc2, Rdst
Type and bit width	32-bit float input, 32-bit int output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator!=(float src1, float src2);`
Additional details	Always return 0 or 1 and never set invalid status flag. See 6.2.4.3 for corner cases.

9.7.5.26 HFCmpLT

Instruction name	HFCmpLT
Functionality	FP16 compare less than
Assembly format	HFCmpLT Rsrc1, Rsrc2, Rdst
Type and bit width	16-bit float input, 32-bit int output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator<(hfloat src1, hfloat src2);`

Instruction name	HFCmpLT
Additional details	Always return 0 or 1 and never set invalid status flag. See 6.2.4.3 for corner cases.

9.7.5.27 HFCmpLE

Instruction name	HFCmpLE
Functionality	FP16 compare less than or equal
Assembly format	HFCmpLE Rsrc1, Rsrc2, Rdst
Type and bit width	16-bit float input, 32-bit int output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator<=(hfloat src1, hfloat src2);`
Additional details	Always return 0 or 1 and never set invalid status flag. See 6.2.4.3 for corner cases.

9.7.5.28 HFCmpGT

Instruction name	HFCmpGT
Functionality	FP16 compare greater than
Assembly format	HFCmpGT Rsrc1, Rsrc2, Rdst
Type and bit width	16-bit float input, 32-bit int output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator>(hfloat src1, hfloat src2);`
Additional details	Always return 0 or 1 and never set invalid status flag. See 6.2.4.3 for corner cases.

9.7.5.29 HFCmpGE

Instruction name	HFCmpGE
Functionality	FP16 compare greater than or equal
Assembly format	HFCmpGE Rsrc1, Rsrc2, Rdst
Type and bit width	16-bit float input, 32-bit int output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator>=(hfloat src1, hfloat src2);`
Additional details	Always return 0 or 1 and never set invalid status flag. See 6.2.4.3 for corner cases.

9.7.5.30 HFCmpEQ

Instruction name	HFCmpEQ
Functionality	FP16 compare equal
Assembly format	HFCmpEQ Rsrc1, Rsrc2, Rdst
Type and bit width	16-bit float input, 32-bit int output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator==(hfloat src1, hfloat src2);`
Additional details	Always return 0 or 1 and never set invalid status flag. See 6.2.4.3 for corner cases.

9.7.5.31 HFCmpNE

Instruction name	HFCmpNE
Functionality	FP16 compare not equal
Assembly format	HFCmpNE Rsrc1, Rsrc2, Rdst
Type and bit width	16-bit float input, 32-bit int output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`bool operator!=(hfloat src1, hfloat src2);`
Additional details	Always return 0 or 1 and never set invalid status flag. See 6.2.4.3 for corner cases.

9.7.5.32 FRCP

Instruction name	FRCP
Functionality	Floating-point reciprocal
Assembly format	FRCP Rsrc, Rdst
Type and bit width	32-bit float input, 32-bit float output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`float frcp(float src);`
Additional details	Performing FP32-input, FP32-output reciprocal. Set invalid status flag when output is NaN. Corner cases: RCP(+denorm) gives +Inf RCP(-denorm) gives -Inf RCP(+0.0) gives +Inf RCP(-0.0) gives -Inf RCP(+1.0) gives +1.0 RCP(-1.0) gives -1.0 RCP(+Inf) gives +0.0 RCP(-Inf) gives -0.0 RCP(NaN) gives NaN Max relative error is 2^-23 over entire normal floating-point range.

9.7.5.33 FSQRT

Instruction name	FSQRT
Functionality	Floating-point square root
Assembly format	FSQRT Rsrc, Rdst
Type and bit width	32-bit float input, 32-bit float output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`float fsqrt(float src);`
Additional details	Performing FP32-input, FP32-output square root. Set invalid status flag when output is NaN. Corner cases: SQRT(+denorm) gives +0.0 SQRT(-denorm) gives -0.0 SQRT(+0.0) gives +0.0 SQRT(-0.0) gives -0.0 SQRT(+1.0) gives +1.0 SQRT(-1.0) gives NaN SQRT(+Inf) gives +Inf SQRT(-Inf) gives NaN SQRT(NaN) gives NaN SQRT(negative) gives NaN (other than for -denorm or -0) Max relative error is 2^-23 over entire normal floating-point range.

9.7.5.34 FRSQ

Instruction name	FRSQ
Functionality	Floating-point reciprocal square root
Assembly format	FRSQ Rsrc, Rdst
Type and bit width	32-bit float input, 32-bit float output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`float frsq(float src);`
Additional details	Performing FP32-input, FP32-output reciprocal square root. Set invalid status flag when output is NaN. Corner cases:

Instruction name	FRSQ
	RSQ(+denorm) gives +Inf RSQ(-denorm) gives -Inf RSQ(+0.0) gives +Inf RSQ(-0.0) gives -Inf RSQ(+1.0) gives +1.0 RSQ(-1.0) gives NaN RSQ(+Inf) gives +0.0 RSQ(-Inf) gives NaN RSQ(NaN) gives NaN RSQ(negative) gives NaN (other than for -denorm or -0) Max relative error is 2^-22.4 over entire normal floating-point range.

9.7.5.35 FEXP2

Instruction name	FEXP2
Functionality	Floating-point exponential base-2
Assembly format	FEXP2 Rsrc, Rdst
Type and bit width	32-bit float input, 32-bit float output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`float fexp2(float src);`
Additional details	Performing FP32-input, FP32-output exponential base-2 function. Set invalid status flag when output is NaN. Corner cases: EXP2(+denorm) gives +1.0 EXP2(-denorm) gives +1.0 EXP2(+0.0) gives +1.0 EXP2(-0.0) gives +1.0 EXP2(+Inf) gives +Inf EXP2(-Inf) gives +0.0 EXP2(NaN) gives NaN Max relative error is 2^-22.5 over entire normal floating-point range.

9.7.5.36 FLOG2

Instruction name	FLOG2
Functionality	Floating-point logarithm base-2
Assembly format	FLOG2 Rsrc, Rdst
Type and bit width	32-bit float input, 32-bit float output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`float flog2(float src);`
Additional details	Performing FP32-input, FP32-output logarithm base-2 function. Set invalid status flag when output is NaN. Corner cases: LOG2(+denorm) gives -Inf LOG2(-denorm) gives -Inf LOG2(+0.0) gives -Inf LOG2(-0.0) gives -Inf LOG2(+Inf) gives +Inf LOG2(-Inf) gives NaN LOG2(NaN) gives NaN LOG2(negative) gives NaN (other than for -denorm or -0) Max absolute error is 2^-22 in range (0.5, 2.0). Max relative error can be as large as 0.9 in range (0.5, 2.0). Max relative error is 2^-22.5 in range [0, 0.5] and [2.0, +Inf].

9.7.5.37 FSIN

Instruction name	FSIN
Functionality	Floating-point sine
Assembly format	FSIN Rsrc, Rdst
Type and bit width	32-bit float input, 32-bit float output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`float fsin(float src);`
Additional details	Performing FP32-input, FP32-output sine function. Input in radians should be pre-normalized by multiplying 1.0/(2*pi). Input in degrees should be pre-normalized by multiplying 1.0/360. Set invalid status flag when output is NaN. Corner cases: SIN(+denorm) gives +0.0 SIN(-denorm) gives -0.0 SIN(+0.0) gives +0.0 SIN(-0.0) gives -0.0 SIN(+Inf) gives NaN SIN(-Inf) gives NaN SIN(NaN) gives NaN SIN(normal) is always in the range [-1, +1] Max absolute error is 2^-20.5 in range -2*pi ~ 2*pi. Max absolute error is 2^-14.7 in range -100*pi ~ 100*pi. The max error includes cumulative error of performing the required pre-normalization. Outside of range -100*pi ~ 100*pi, only best effort is provided; there are no defined error guarantees.

9.7.5.38 FCOS

Instruction name	FCOS
Functionality	Floating-point cosine
Assembly format	FCOS Rsrc, Rdst
Type and bit width	32-bit float input, 32-bit float output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`float fcos(float src);`
Additional details	Performing FP32-input, FP32-output cosine function. Input in radians should be pre-normalized by multiplying 1.0/(2*pi). Input in degrees should be pre-normalized by multiplying 1.0/360. Set invalid status flag when output is NaN. Corner cases: COS(+denorm) gives +1.0 COS(-denorm) gives +1.0 COS(+0.0) gives +1.0 COS(-0.0) gives +1.0 COS(+Inf) gives NaN COS(-Inf) gives NaN COS(NaN) gives NaN COS(normal) is always in the range [-1, +1] Max absolute error is 2^-20.9 in range -2*pi ~ 2*pi. Max absolute error is 2^-15.3 in range -100*pi ~ 100*pi. The max error includes cumulative error of performing the required pre-normalization. Outside of range -100*pi ~ 100*pi, only best effort is provided; there are no defined error guarantees.

9.7.5.39 FTANH

Instruction name	FTANH
Functionality	Floating-point hyperbolic tangent
Assembly format	FTANH Rsrc, Rdst
Type and bit width	32-bit float input, 32-bit float output
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`float ftanh(float src);`
Additional details	Performing FP32-input, FP32-output hyperbolic function. Set invalid status flag when output is NaN. Corner cases: TANH(-denorm) gives -0.0 TANH(-0.0) gives -0.0 TANH(+0.0) gives +0.0 TANH(+denorm) gives +0.0 TANH(-Inf) gives -1.0 TANH(+Inf) gives 1.0 TANH(NaN) gives NaN TANH(normal) is always in the range [-1.0 .. +1.0] Max relative error is 2^-11 over the entire normal floating-point range. Max absolute error is 2^-12 over the entire normal floating-point range.

9.7.6 Other Scalar ALU Instructions

9.7.6.1 Instruction Summary

Table 27. Other scalar ALU instructions

Function	Assembly Format	Comments
Count leading bits	CLB Rsrc, Rdst	If bit 31 is zero, count leading 0 bits, otherwise, count leading 1 bits
Load high	LHI imm16, Rdst	Set destination to (immediate << 16)
Or immediate	ORI Rsrc1, imm16, Rdst	Set destination to Rsrc1 OR immediate. LHI/ORI sequence is used to load a 32-bit immediate value into a scalar register.
Mux	MUX Rsrc1, Rsrc2, Rsrc3, Rdst	Select between 2 items Rdst = Rsrc1 ? Rsrc2 : Rsrc3
Divide	DIV Rsrc1, Rsrc2	Divide Rsrc1 by Rsrc2, resulting quotient into PL and remainder into PH, takes multiple cycles. Rsrc1 and Rsrc2 are regarded as unsigned 32-bit number. When Rsrc2 is zero, return quotient = 0xFFFF_FFFF (max value of unsigned 32-bit), and return remainder = Rsrc1. Divide-by-zero would generate error interrupt to R5. Only available in S0 slot.
Logical left shift and add	SLLIADD Rsrc1, UImm4, Rsrc2, Rdst	`dst = (src1 << imm) + src2;`
Compare within	CMPWITHIN Rsrc1, Rsrc2, Rsrc3, Rdst	`dst = (src1 <= src2) && (src2 < src3);`
Bit count	BITCNT Rsrc, Rdst	`Count number of bits set to one`

9.7.6.2 CLB

Instruction name	CLB
Functionality	Count leading bits
Assembly format	CLB Rsrc, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int clb(int src);`
Additional details	If bit 31 of the source is 0, count number of consecutive 0 bits from bit 31 down. Otherwise, count number of consecutive 1 bits from bit 31 down. Examples: clb(0) = 32 clb(0x1000_0000) = 3 clb(0x6000_0000) = 1 clb(0x8000_0000) = 1 clb(0xE000_0000) = 3 clb(0xFFF0_0000) = 12

9.7.6.3 LHI

Instruction name	LHI
Functionality	Load high. Load (immediate << 16) into scalar destination, and thus not just loading high, but clearing low at the same time.
Assembly format	LHI imm16, Rdst
Type and bit width	32-bit
Predication	not available
Source options	not available
Destination options	scalar register
Additional options	

Intrinsics/operator	`// not available, instantiated automatically when assigning`
	`// a value exceeding 12-bit to variable mapped to a scalar`
	`// register, for example,`
	`//   int var1 = 0x654321;`
	`// is compiled into`
	`//   LHI 0x65, R4`
	`//   ORI 0x4321, R4`
	`// when var1 is mapped to R4. When the value fits 12-bit,`
	`// compiler instantiates ADDI, for example,`
	`//   ADDI R0, #321, R4`

Additional details	Set destination to (immediate << 16)

9.7.6.4 ORI

Instruction name	ORI
Functionality	Bitwise OR with 16-bit immediate
Assembly format	ORI Rsrc1, imm16, Rdst
Type and bit width	32-bit
Predication	not available
Source options	not available
Destination options	scalar register
Additional options	

Intrinsics/operator	`// not available, instantiated automatically when assigning // value exceeding 16-bit to variable mapped to a scalar register, for example,`
	`//   int var1 = 0x654321;`
	`// is compiled into`
	`//   LHI 0x65, R4`
	`//   ORI 0x4321, R4`
	`// when var1 is mapped to R4. When the value fits 12-bit,`
	`// compiler instantiates ADDI, for example,`
	`//   ADDI R0, #321, R4`

Additional details	Set destination to Rsrc1 OR immediate. LHI/ORI sequence is used to load a 32-bit immediate value into a scalar register.

9.7.6.5 MUX

Instruction name	MUX
Functionality	Scalar multiplexing
Assembly format	MUX Rsrc1, Rsrc2, Rsrc3, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	

Intrinsics/operator	`char  mux(int src1, char  src2, char  src3);`
	`short mux(int src1, short src2, short src3);`
	`int  mux(int src1, int  src2, int  src3);`
	`hfloat mux(int src1, hfloat src2, hfloat src3);`
	`float mux(int src1, float src2, float src3);`
	`char  mux(bool src1, char  src2, char  src3);`
	`short mux(bool src1, short src2, short src3);`
	`int  mux(bool src1, int  src2, int  src3);`
	`hfloat mux(bool src1, hfloat src2, hfloat src3);`
	`float mux(bool src1, float src2, float src3);`

Additional details	Select between 2 data items, Rdst = Rsrc1 ? Rsrc2 : Rsrc3

9.7.6.6 DIV

Instruction name	DIV
Functionality	Scalar divide
Assembly format	DIV Rsrc1, Rsrc2
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	

Intrinsics/operator	`void udiv(unsigned int src1, unsigned int src2, unsigned int & dst1, unsigned int & dst2);`
	`unsigned int operator/( unsigned int src1, unsigned int src2);`
	`unsigned int operator%( unsigned int src1, unsigned int src2);`

Additional details	Divide Rsrc1 by Rsrc2, resulting quotient into PL and remainder into PH, takes multiple cycles. Rsrc1 and Rsrc2 are regarded as unsigned 32-bit number. When Rsrc2 is zero, return quotient = 0xFFFF_FFFF (max value of unsigned 32-bit), and return remainder = Rsrc1. Divide-by-zero would generate error interrupt to R5. This is a multi-cycle instruction, taking up to 33 cycles to complete. Subsequent instructions using PL/PH as source or destination shall be stalled until DIV completes. Also, to avoid task switch before PL/PH are written, subsequent HALT, WFE_R5, and GPO writes are stalled until DIV completes. Note that DIV is only available in the S0 slot.

9.7.6.7 SLLIADD

This is an instruction added in Gen-2 VPU to accelerate address calculation.

Instruction name	SLLIADD
Functionality	Scalar shift and add
Assembly format	SLLIADD Rsrc1, UImm4, Rsrc2, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int slladd(int src1, int shift_uimm4, int src2);`
Additional details	Unsigned 4-bit immediate value is used as left shift bit count, to shift left by up to 15 bits. `int dst = ((int src1) << imm) + (int src2);`

Why just 4-bit? The intention of this instruction is to support address calculation of base[index], byte_addr(base) + index * sizeof(base), when the size of the data type is a power of 2. 4-bit left shift would cover up to size of 2^15 = 32768 bytes, and is more than commonly needed.

9.7.6.8 CMPWITHIN

This is an instruction added in Gen-2 VPU to accelerate range checking.

Instruction name	CMPWITHIN
Functionality	Compare within
Assembly format	CMPWITHIN Rsrc1, Rsrc2, Rsrc3, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int cmpwithin(int src1, int src2, int src3);`
Additional details	Returns (src1 <= src2) && (src2 < src3); Note that signed comparison is carried out.

9.7.6.9 BITCNT

Instruction name	BITCNT
Functionality	Bit count
Assembly format	BITCNT Rsrc, Rdst
Type and bit width	32-bit
Predication	not available
Source options	scalar register
Destination options	scalar register
Additional options	
Intrinsics/operator	`int bitcount(int src);`
Additional details	Count number of bits set to one; a scalar version of VBitCnt. For example, bitcount(13) = 3, as the binary representation of 13 is "1101" and contains 3 ones.

9.8 Vector ALU Instructions

9.8.1 Move Instructions

9.8.1.1 Instruction Summary

Table 28 Scalar/vector move instructions

Function	Assembly Format	Comments
Vector move	VMov Vsrc/Wsrc, Vdst/Wdst VMov Vsrc, ACdst VMov ACsrc, Vdst VMov ACsrc, ACdst VMov XACsrc, XACdst	Move vector register
Move scalar to vector	<pred> VMovS<W/WU/H/B> Rsrc, Vdst/Wdst/ACdst	Broadcast scalar register to all W/H/B lanes of vector register
Vector move double	<pred> DVMov DVsrc/DWsrc, DVdst/DWdst <pred> DVMov DACsrc, DACdst <pred> DVMov DXACsrc, DXACdst DVMov DVsrc, DACdst DVMov DACsrc, DVdst	Move double vector register
Vector move pair	VMov2 Vsrc1, Vsrc2, Vdst1, Vdst2	Move 2 vector registers
Move from vector to scalar	<pred> MovVS<W/H/B/HU/BU> Vsrc, Rdst	Move vector lane 0 to scalar register

Table 29 vector register move support matrix

		Destination			
		VRF	**WRF**	**ARF**	**XARF**
Source	**VRF**	single/double	single/double	single/double	demote_i double-to-single
	WRF	single/double	single/double		
	ARF	single/double		single/double	
	XARF	promote_di single-to-double			single/double

9.8.1.2 VMOV

Instruction name	**VMOV**
Functionality	Vector move
Assembly format	VMov Vsrc/Wsrc, Vdst/Wsrc VMov Vsrc, ACdst VMov ACsrc, Vdst VMov ACsrc, ACdst VMov XACsrc, XACdst
Type and bit width	n/a: 384-bit
Predication	not available
Source options	Single vector register in VRF, WRF, ARF, XARF
Destination options	Single vector register in VRF, WRF, ARF, XARF
Additional options	

Intrinsics/operator	`// not needed; instantiated from assignment statement`
	`// between source and destination of same single vector`
	`// type, for example,`
	`// vintx dst = vintx src;`
	`// vshortx dst = vshortx src;`
	`// vcharx dst = vcharx src;`
	`// xvshortx dst = xvshortx src;`
	`// xvcharx dst = xvcharx src;`
	`// vfloatx dst = vfloatx src;`
	`// vhfloatx dst = vhfloatx src;`

Additional details	

9.8.1.3 VMOVS

Instruction name	VMOVS
Functionality	Move scalar to vector
Assembly format	<pred> VMovS<type> Rsrc, Vdst/Wdst/ACdst pred = none, [P2..P15]
Type and bit width	W: 32-bit sign-extended to 48-bit and broadcast to 8 x 48-bit WU: 32-bit zero-extended to 48-bit and broadcast to 8 x 48-bit H: lowest 24-bit broadcast to 16 x 24-bit B: lowest 12-bit broadcast to 32 x 12-bit Note that float/vfloatx type intrinsic function is mapped to W type instruction, and hfloat/vhfloatx type intrinsic function is mapped to the H type instruction.
Predication	Instruction-level predication
Source options	Scalar register
Destination options	Single vector register in VRF, WRF, ARF
Additional options	

Intrinsics/operator	`vintx  replicatew(int src);`
	`vintx  replicatew(unsigned int src);`
	`vshortx replicateh(int src);`
	`vcharx replicateb(int src);`
	`vfloatx replicatef(float src); // W type, float value`
	`// sign-extended to 48-bit`
	`vhfloatx replicatehf(hfloat src); // H type, hfloat value`
	`// sign-extended to 24-bit`

Additional details	Example:
	[P2] VMovSH R2, V3
	When P2 is non-zero, this would copy R2[23:0] to all 16 half-word lanes of V3. Otherwise, V3 is unchanged.
	
	The predication feature is not exposed through intrinsic functions, but with code pattern:
	if (condition) chess_guard {
	vector_var = replicatew(scalar_value);
	}

9.8.1.4 DVMOV

Instruction name	DVMOV
Functionality	Move double vector
Assembly format	<pred> DVMov DVsrc/DWsrc, DVdst/DWdst <pred> DVMov DACsrc, DACdst <pred> DVMov DXACsrc, DXACdst DVMov DVsrc, DACdst DVMov DACsrc, DVdst pred = none, [P2..P15]
Type and bit width	n/a: 768-bit
Predication	Instruction-level predication on DV/DW moves
Source options	Double vector register in VRF, WRF, ARF, XARF
Destination options	Double vector register in VRF, WRF, ARF, XARF
Additional options	

Intrinsics/operator	`// not needed; instantiated from assignment statement`
	`// between source and destination of same single vector`
	`// type, for example,`
	`// dvintx dst = dvintx src;`
	`// dvshortx dst = dvshortx src;`
	`// dvcharx dst = dvcharx src;`
	`// dxvshortx dst = dxvshortx src;`
	`// dxvcharx dst = dxvcharx src;`
	`// dvfloatx dst = dvfloatx src;`
	`// dvhfloatx dst = dvhfloatx src;`

Additional details	

9.8.1.5 VMOV2

Instruction name	VMOV2
Functionality	Move vector pair
Assembly format	VMov2 Vsrc1, Vsrc2, Vdst1, Vdst2
Type and bit width	n/a: 384-bit
Predication	not available
Source options	Two vector registers in VRF
Destination options	Two vector registers in VRF
Additional options	

Intrinsics/operator	`// Optional; instantiated from two assignments of the`
	`// same single vector data type, or one assignment of the`
	`// same double vector data type, for example,`
	`// vintx dst = vintx src;`
	`// vshortx dst = vshortx src;`
	`// vcharx dst = vcharx src;`
	`// vfloatx dst = vfloatx src;`
	`// vhfloatx dst = vhfloatx src;`
	`void dvmov(vfloatx src1, vfloatx src2, vfloatx &dst1, vfloatx &dst2);`
	`void dvmov(vhfloatx src1,vhfloatx src2,vhfloatx &dst1,vhfloatx &dst2);`
	`void dvmov(vintx src1, vintx src2, vintx &dst1, vintx &dst2);`
	`void dvmov(vshortx src1, vshortx src2, vshortx &dst1, vshortx &dst2);`
	`void dvmov(vcharx src1, vcharx src2, vcharx &dst1, vcharx &dst2);`

Additional details	

9.8.1.6 MOVVS

Instruction name	MOVVS
Functionality	Move vector lane 0 to scalar
Assembly format	<pred> MovVS<W/H/B/HU/BU> Vsrc, Rdst pred = none, [P2..P15]
Type and bit width	W: 32-bit H: 24-bit sign-extend to 32-bit B: 12-bit sign-extend to 32-bit HU: 24-bit zero-pad to 32-bit BU: 12-bit zero-pad to 32-bit Note that float/vfloatx type intrinsic function is mapped to W type instruction, and hfloat/vhfloatx type intrinsic function is mapped to the H type instruction.
Predication	Instruction-level predication
Source options	Vector register (lane 0) in VRF
Destination options	Scalar register
Additional options	n/a

Intrinsics/operator	`int smovw (vintx src);`
	`int smovh (vshortx src);`
	`int smovb (vcharx src);`
	`int smovhu (vshortx src);`
	`int smovbu (vcharx src);`
	`float smovf (vfloatx src);`
	`hfloat smovhf (vhfloatx src);`

Additional details	Available in memory slots.
	
	The predication feature is not exposed through intrinsic functions, but with code pattern:
	`if (condition) chess_guard {`
	`  scalar_var = smovw(vector_value);`
	`}`

9.8.2 Vector OP11 Instructions

These are one-source, one-destination operation vector instructions.

The double vector flavor is supported for selected operators.

9.8.2.1 Instruction Summary

Table 30. Vector OP11 instructions

Function	Assembly Format	Comments
Vector not bitwise	VNot Vsrc/Wsrc, Vdst/Wsrc	
Vector not logical	VNotL<W/H/B> Vsrc/Wsrc, Vdst/Wdst	
Vector bit reverse	VBitRev<W/H/B> Vsrc/Wsrc, Vdst/Wdst	Use standard (32/16/8) bit width
Vector negate	VNeg<W/H/B> Vsrc/Wsrc, Vdst/Wdst	
Vector sum reduction	VSumR<type> Vsrc/Wsrc, Vdst/Wdst/Rdst type = {BW, HW, W}	Sum of all lanes. Result stored across all lanes of Vdst
Vector min reduction	VMinR<W/H/B> Vsrc/Wsrc, Rdst	Min of all lanes. Scalar destination only
Vector max reduction	VMaxR<W/H/B> Vsrc/Wsrc, Rdst	Max of all lanes. Scalar destination only
Vector AND reduction	VAndR<W/H/B> Vsrc/Wsrc, Vdst/Wdst/Rdst	Bitwise AND across all lanes. Result stored across all lanes of Vdst
Vector OR reduction	VOrR<W/H/B> Vsrc/Wsrc, Vdst/Wdst/Rdst	Bitwise OR across all lanes. Result stored across all lanes of Vdst
Vector XOR reduction	VXorR<W/H/B> Vsrc/Wsrc, Vdst/Wdst/Rdst	Bitwise XOR across all lanes. Result stored across all lanes of Vdst
Vector bitunpack	VBitUnpk<W/H/B> Rsrc, Vdst/Wdst	Bit unpack from scalar
Vector bit-transpose	VBitTranspH Vsrc/Wsrc, Vdst/Wdst	Transpose 16-bit across 16 lanes. Used in LBP encode, speedup ~ 2x
Vector most-significant bit detect	VMSBD<W/H/B> Vsrc/Wsrc, Vdst/Wdst	Return most significant bit position, input must be non-negative, return -1 for zero input
Vector bit count	VBitCnt<W/H/B> Vsrc/Wsrc, Vdst/Wdst VBitCnt<W/H/B> DVsrc/DWsrc, DVdst/DWdst	Count number of 1 bits
Vector collate indices	VCollateIdx<W/H> Vsrc/Wsrc, Vdst/Wdst	Compute indices to nonzero lanes
Vector expand indices	VExpandIdx<W/H> Vsrc/Wsrc/Rsrc, Vdst/Wdst	Compute indices to expand collated data

Function	Assembly Format	Comments
Vector horizontal 2-term min-ID	VHMin2IDW Vsrc/Wsrc, Vdst/Wdst	Computer min & ID in each lane pair
Vector horizontal 2-term max-ID	VHMax2IDW Vsrc/Wsrc, Vdst/Wdst	Computer max & ID in each lane pair
Vector shuffle permutation	VShuffle<type> Vsrc/Wsrc, Vdst/Wdst	

9.8.2.2 VNOT

Instruction name	VNOT
Functionality	Vector inversion bitwise
Assembly format	VNot Vsrc/Wsrc, Vdst/Wsrc
Type and bit width	no type: 384-bit (bitwise)
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  operator~(vintx src);`
	`vshortx operator~(vshortx src);`
	`vcharx operator~(vcharx src);`
	`// double vector pseudo intrinsics`
	`dvintx  operator~(dvintx src);`
	`dvshortx operator~(dvshortx src);`
	`dvcharx operator~(dvcharx src);`

Additional details	

9.8.2.3 VNOTL

Instruction name	VNOTL
Functionality	Vector inversion logical
Assembly format	VNotL<type> Vsrc/Wsrc, Vdst/Wsrc
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  operator!(vintx src);`
	`vshortx operator!(vshortx src);`
	`vcharx operator!(vcharx src);`
	`// double vector pseudo intrinsics`
	`dvintx  operator!(dvintx src);`
	`dvshortx operator!(dvshortx src);`
	`dvcharx operator!(dvcharx src);`

Additional details	Example: VNotLB V1, V2 This would detect zero/non-zero of V1 byte lanes, and set a byte lane of V2 to 0 if the corresponding lane in V1 is non-zero, and 1 if the corresponding lane in V1 is zero.

9.8.2.4 VBITREV

Instruction name	VBITREV
Functionality	Vector bit reverse
Assembly format	VBitRev<type> Vsrc/Wsrc, Vdst/Wsrc
Type and bit width	W: 8 x 32-bit, H: 16 x 16-bit, B: 32 x 8-bit
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  vbitreverse(vintx  src);`
	`vshortx vbitreverse(vshortx src);`
	`vcharx vbitreverse(vcharx src);`
	`// double vector pseudo intrinsics`
	`dvintx  dvbitreverse(dvintx  src);`
	`dvshortx dvbitreverse(dvshortx src);`
	`dvcharx dvbitreverse(dvcharx src);`

Additional details	Reverse lower 8/16/32 bits of each lane; upper bits are dropped. Output lower 8/16/32 bits of each lane bit-reversed; upper bits are zero, and appear unsigned (or non-negative).

Instruction name	VBITREV

	Example:
	`vintx src = {0, 0x100, 0x200, 0x300, 0, 0, 0, 0};`
	`vintx dst = vbitreverse(src);`
	Expected dst = {0, 0x80_0000, 0x40_0000, 0xC0_0000, 0, 0, 0, 0}

9.8.2.5 VNEG

Instruction name	VNEG
Functionality	Vector negate
Assembly format	VNeg<type> Vsrc/Wsrc, Vdst/Wsrc
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx operator-(vintx src);`
	`vshortx operator-(vshortx src);`
	`vcharx operator-(vcharx src);`
	`// double vector pseudo intrinsics`
	`dvintx operator-(dvintx src);`
	`dvshortx operator-(dvshortx src);`
	`dvcharx operator-(dvcharx src);`

Additional details	

9.8.2.6 VSUMR

Instruction name	VSUMR
Functionality	Vector sum reduction
Assembly format	VSumR<type> Vsrc/Wsrc, Vdst/Wdst/Rdst
Type and bit width	W: 8 x 48-bit → 8 x 48-bit HW: 16 x 24-bit → 8 x 48-bit BW: 32 x 12-bit → 8 x 48-bit Note that sign extension is applied for HW and BW cases.
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single vector register in VRF or WRF, or scalar register
Additional options	

Intrinsics/operator	`vintx vsumr(vintx  src);`
	`vintx vsumr(vshortx src);`
	`vintx vsumr(vcharx src);`

Instruction name	VSUMR

	`int vsumr_s(vintx  src);`
	`int vsumr_s(vshortx src);`
	`int vsumr_s(vcharx src);`
	`// double vector pseudo intrinsics unavailable`

Additional details	Sum across all lanes of source is broadcast to all lanes of destination. Destination is of W-type to reduce chance of overflow. Note that number of lanes reduces for HW and BW variations. Programmer should be aware of possibility of overflow in the VSumRW case, and code accordingly. For scalar destination, in W-type, 32 LSBs of the sum is returned. Programmer should be aware of potential overflow in the outcome. In H-type and B-type, the sum is sign-extended to 32-bit.

9.8.2.7 VMINR

Instruction name	VMINR
Functionality	Vector min reduction
Assembly format	VMinR<type> Vsrc/Wsrc, Rdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single register in scalar register file
Additional options	

Intrinsics/operator	`int vminr_s(vintx  src);`
	`int vminr_s(vshortx src);`
	`int vminr_s(vcharx src);`
	`// Following gen-1 legacy intrinsics shall be emulated with multiple instructions`
	`vintx  vminr(vintx  src);`
	`vshortx vminr(vshortx src);`
	`vcharx vminr(vcharx src);`
	`// double vector pseudo intrinsics unavailable`

Additional details	Min across all lanes of source is stored in the scalar destination. For W-type, 32 LSBs of the min value is returned. Programmer should be aware of potential overflow in the outcome. In H-type and B-type, the min value is sign-extended to 32-bit. Gen-1 legacy intrinsics broadcast outcome across all lanes of destination register. For W-type 48-bit min value is output in each lane of the vector destination. For Halfword and Byte types, the emulation uses vminr_s() and replicateh/b(). For Word type, using just vminr_s() and replicatew() will not compute bits 47..32 of the extended word lane properly. Instead, the emulation uses vhmin2id() and vminskip2rid(). See 9.8.2.18 and 9.8.3.9 for details.

Instruction name	VMINR

	Examples:
	
	VMinRB V1, V2 is emulated as VMinRB V1, R2; VMovSB R2, V2.
	vminr(vcharx_src) as { replicateb(vminr_s(vcharx_src)); }
	
	VMinRH V1, V2 is emulated as VMinRH V1, R2; VMovSH R2, V2.
	vminr(vshortx_src) as { replicateh(vminr_s(vshortx_src)); }
	
	VMinRW V1, V2 is emulated as
	VHMin2IDW V1, V3; VMinSkip2RIDW V3, V2, R2.
	vminr(vintx_src) as {
	  vhmin2id(vintx_src, temp);
	  vminskip2rid(temp, vintx_dst1, id_dst2);
	  return vintx_dst1;
	}

9.8.2.8 VMAXR

Instruction name	VMAXR
Functionality	Vector max reduction
Assembly format	VMaxR<type> Vsrc/Wsrc, Rdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single register in scalar register file
Additional options	

Intrinsics/operator	int vmaxr_s(vintx src);
	int vmaxr_s(vshortx src);
	int vmaxr_s(vcharx src);
	// Following gen-1 legacy intrinsics shall be emulated with multiple instructions
	vintx vmaxr(vintx src);
	vshortx vmaxr(vshortx src);
	vcharx vmaxr(vcharx src);
	// double vector pseudo intrinsics unavailable

Additional details	Max across all lanes of source is stored in the scalar destination. For W-type, 32 LSBs of the max value is returned. Programmer should be aware of potential overflow in the outcome. In H-type and B-type, the max value is sign-extended to 32-bit. Gen-1 legacy intrinsics broadcast outcome across all lanes of destination register. For W-type 48-bit max value is output in each lane of the vector destination. For Halfword and Byte types, the emulation uses vmaxr_s() and replicateh/b().

Instruction name	VMAXR

	For Word type, using just vmaxr_s() and replicatew() will not compute bits 47..32 of the extended word lane properly. Instead, the emulation uses vhmax2id() and vmaxskip2rid(). See 9.8.2.18 and 9.8.3.10 for details.
	
	Examples:
	
	`VMaxRB V1, V2 is emulated as VMaxRB V1, R2; VMovSB R2, V2.`
	`vmaxr(vcharx_src) as { replicateb(vmaxr_s(vcharx_src)); }`
	
	`VMaxRH V1, V2 is emulated as VMaxRH V1, R2; VMovSH R2, V2.`
	`vmaxr(vshortx_src) as { replicateh(vmaxr_s(vshortx_src)); }`
	
	`VMaxRW V1, V2 is emulated as`
	`VHMax2IDW V1, V3; VMaxSkip2RIDW V3, V2, R2.`
	`vmaxr(vintx_src) as {`
	`  vhmax2id(vintx_src, temp);`
	`  vmaxskip2rid(temp, vintx_dst1, id_dst2);`
	`  return vintx_dst1;`
	`}`

9.8.2.9 VANDR

Instruction name	VANDR
Functionality	Vector bitwise AND reduction
Assembly format	VAndR<type> Vsrc/Wsrc, Vdst/Wdst/Rdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single vector register in VRF or WRF, or scalar register
Additional options	

Intrinsics/operator	`vintx vandr(vintx src);`
	`vshortx vandr(vshortx src);`
	`vcharx vandr(vcharx src);`
	`int vandr_s(vintx  src);`
	`int vandr_s(vshortx src);`
	`int vandr_s(vcharx src);`
	`// double vector pseudo intrinsics unavailable`

Additional details	Bitwise AND across all lanes of source is broadcast to all lanes of destination. `dst[i] = src[0] & src[1] & … & src[nlanes – 1]` For scalar destination, in W-type, 32 LSBs of the AND reduction value is returned. Programmer should be aware of potential overflow in the outcome. In H-type and B-type, the AND reduction value is zero-extended to 32-bit.

9.8.2.10 VORR

Instruction name	VORR
Functionality	Vector bitwise OR reduction
Assembly format	VOrR<type> Vsrc/Wsrc, Vdst/Wdst/Rdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single vector register in VRF or WRF, or scalar register
Additional options	

Intrinsics/operator	`vintx vorr(vintx src);`
	`vshortx vorr(vshortx src);`
	`vcharx vorr(vcharx src);`
	`int vorr_s(vintx  src);`
	`int vorr_s(vshortx src);`
	`int vorr_s(vcharx src);`
	`// double vector pseudo intrinsics unavailable`

Additional details	Bitwise OR across all lanes of source is broadcast to all lanes of destination. `dst[i] = src1[0] | src1[1] | … | src1[nlanes − 1]` For scalar destination, in W-type, 32 LSBs of the OR reduction value is returned. Programmer should be aware of potential overflow in the outcome. In H-type and B-type, the OR reduction value is zero-extended to 32-bit.

9.8.2.11 VXORR

Instruction name	VXORR
Functionality	Vector bitwise XOR reduction
Assembly format	VXorR<type> Vsrc/Wsrc, Vdst/Wdst/Rdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single vector register in VRF or WRF, or scalar register
Additional options	

Intrinsics/operator	`vintx  vxorr(vintx src);`
	`vshortx vxorr(vshortx src);`
	`vcharx vxorr(vcharx src);`
	`int vxorr_s(vintx  src);`
	`int vxorr_s(vshortx src);`
	`int vxorr_s(vcharx src);`
	`// double vector pseudo intrinsics unavailable`

Additional details	XOR across all lanes of source is broadcast to all lanes of destination. `dst[i] = src[0] ^ src[1] ^ … ^ src[nlanes − 1]`

Instruction name	VXORR
	For scalar destination, in W-type, 32 LSBs of the XOR reduction value is returned. Programmers should be aware of potential overflow in the outcome. In H-type and B-type, the XOR reduction value is zero-extended to 32-bit.

9.8.2.12 VBITUNPK

Instruction name	VBITUNPK
Functionality	Vector unpack from scalar
Assembly format	VBitUnpk<type> Rsrc, Vdst/Wdst
Type and bit width	W: take Rsrc[7:0], unpack into 8 x 48-bit, each lane gets 0 or 1 H: take Rsrc[15:0], unpack into 16 x 24-bit, each lane gets 0 or 1 B: take Rsrc[31:0], unpack into 32 x 12-bit, each lane gets 0 or 1
Predication	not available
Source options	Scalar register
Destination options	Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vcharx vbitunpackb(int src);`
	`vshortx vbitunpackh(int src);`
	`vintx  vbitunpackw(int src);`
	`// double vector pseudo intrinsics unavailable`

Additional details	Unpack lower 8/16/32-bit of source scalar register, one bit into each vector lane, bit i into lane i. For example, with R4 = 0xF0, “VBitUnpkW R4, V0” would result in V0 = {0, 0, 0, 0, 1, 1, 1, 1}

9.8.2.13 VBITTRANSP

Instruction name	VBITTRANSP
Functionality	Vector bit transpose
Assembly format	VBitTranspH Vsrc/Wsrc, Vdst/Wsrc
Type and bit width	H: 16 x 16-bit
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single vector register in VRF or WRF
Additional options	
Intrinsics/operator	`vshortx vbittranspose(vshortx src);` `// double vector pseudo intrinsics unavailable`
Additional details	Transpose between bit dimension (16 bits) and lane dimension (16 lanes), useful for census transform and rank transform

Example for VBitTranspH:

			Lane															
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Input	Value (dec)		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	Value (hex)		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	Bit	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
		2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
		3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
		4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bit-Transposed	Bit	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		3	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		4	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
		5	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
		6	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
		7	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
		8	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
		9	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
		10	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
		11	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
		12	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
		13	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
		14	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
		15	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	Value (dec)		43690	52428	61680	65280	0	0	0	0	0	0	0	0	0	0	0	0
	Value (hex)		AAAA	CCCC	F0F0	FF00	0	0	0	0	0	0	0	0	0	0	0	0

9.8.2.14 VMSBD

Instruction name	VMSBD
Functionality	Vector most significant bit detect
Assembly format	VMSBD<type> Vsrc/Wsrc, Vdst/Wsrc
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, unsigned
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  vmsbd(vintx  src);`
	`vshortx vmsbd(vshortx src);`
	`vcharx vmsbd(vcharx src);`
	`// double vector pseudo intrinsics`
	`dvintx  dvmsbd(dvintx  src);`
	`dvshortx dvmsbd(dvshortx src);`
	`dvcharx dvmsbd(dvcharx src);`

Additional details	Return most significant bit position, treat input as unsigned, return -1 for zero input. For example, value of 0x12 has leading bit in bit 4, thus MSBD would return 4.

9.8.2.15 VBITCNT

Instruction name	VBITCNT
Functionality	Vector bit count
Assembly format	VBitCnt<type> Vsrc/Wsrc, Vdst/Wsrc
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  vbitcount(vintx  src);`
	`vshortx vbitcount(vshortx src);`
	`vcharx vbitcount(vcharx src);`

Additional details	Count input “1” bits. For example, input of 12 = 0x0C would lead to bit count of 2.

Instruction name	VBITCNT
Functionality	Double vector bit count
Assembly format	VBitCnt<type> DVsrc/DWsrc, DVdst/DWsrc
Type and bit width	W: 2 x 8 x 48-bit, H: 2 x 16 x 24-bit, B: 2 x 32 x 12-bit, signed
Predication	not available

Instruction name	VBITCNT
Source options	Double vector register in VRF or WRF
Destination options	Double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`dvintx  dvbitcount(dvintx  src);`
	`dvshortx dvbitcount(dvshortx src);`
	`dvcharx dvbitcount(dvcharx src);`

Additional details	Count input “1” bits. For example, input of 12 = 0x0C would lead to bit count of 2.

9.8.2.16 VCOLLATEIDX

Instruction name	VCOLLATEIDX
Functionality	Vector collate
Assembly format	VCollateIdx<type> Vsrc/Wsrc, Vdst/Wdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit (no byte type support)
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single vector registers in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  vcollate_idx(vintx  src);`
	`vshortx vcollate_idx(vshortx src);`
	`// double vector pseudo intrinsics unavailable`

Additional details	Vdst/Wdst gets indices to nonzero lanes (sequentially from lane 0) of Vsrc/Wsrc, followed by indices to zero lanes of Vsrc/Wsrc. For example, VCollateIdxW V0, V1, with V0 = {0, -1, 2, -3, 0, 0, 0, 4}. Non-zero lanes are lane 1, 2, 3, and 7. Expected outcome V1 = {1, 2, 3, 7, 0, 4, 5, 6}. The idea is that a subsequent VPermW would use V1 as indices to extract/compact V0 nonzero and zero lanes into {-1, 2, -3, 4, 0, 0, 0, 0}.

9.8.2.17 VEXPANDIDX

Instruction name	VEXPANDIDX
Functionality	Vector expand, the inverse operation of vector collate
Assembly format	VExpandIdx<type> Vsrc/Wsrc/Rsrc, Vdst/Wdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit (no byte type support)
Predication	not available
Source options	Single vector register in VRF or WRF, or scalar register
Destination options	Single vector registers in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  vexpand_idx(vintx  src);`
	`vshortx vexpand_idx(vshortx src);`
	`vintx  vexpand_idxw(int  src);`
	`vshortx vexpand_idxh(int  src);`
	`// double vector pseudo intrinsics unavailable`

Additional details	Identify nonzero lanes of Vsrc/Wsrc (sequentially from lane 0) and replace these lanes with incrementing indices. Zero lanes continue the indexing from non-zero lanes. When scalar register source is used, extract zero/nonzero directly from the scalar, bit i = 1 indicating lane i is nonzero. The source contains a Boolean array. The expanded index can be used to expand, or uncollated, a collated array back to original data. For example, VExpandIdxW V0, V1, with V0 = {0, 1, 1, 1, 0, 0, 0, 1}. Non-zero lanes are 1, 2, 3, and 7. Expected outcome V1 = {4, 0, 1, 2, 5, 6, 7, 3}. The idea is that a subsequent VPermW would use V1 as indices to expand collated data, for example, {-1, 2, -3, 4, 0, 0, 0, 0} to {0, -1, 2, -3, 0, 0, 0, 4}.

9.8.2.18 VHMIN2ID

Instruction name	VHMIN2ID
Functionality	Vector horizontal (between lane) min and ID
Assembly format	VHMin2ID<type> Vsrc/Wsrc, Vdst/Wdst
Type and bit width	W: 8 x 48-bit signed
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single vector register in VRF or WRF
Additional options	
Intrinsics/operator	`vintx vhmin2id(vintx  src);`

Additional details	Min-ID between even/odd lanes in each lane pair. even destination lane gets the minimal value, odd destination lane gets the lane ID (0 or 1) that has the minimal value, in case of tie, ID = 0.
	`dst[2*i] = min(src[2*i], src[2*i+1]);`
	`dst[2*i+1] = (src[2*i] <= src[2*i+1]) ? 0 : 1;`
	This operation is supported only for the Word type, to handle potential overflow of Word-type VMinRID with scalar register destinations.

9.8.2.19 VHMAX2ID

Instruction name	VHMAX2ID
Functionality	Vector horizontal (between lane) max and ID
Assembly format	VHMax2ID<type> Vsrc/Wsrc, Vdst/Wdst
Type and bit width	W: 8 x 48-bit signed
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single vector register in VRF or WRF
Additional options	
Intrinsics/operator	`vintx vhmax2id(vintx src);`

Additional details	Min-ID between even/odd lanes in each lane pair. even destination lane gets the maximal value, odd destination lane gets the lane ID (0 or 1) that has the maximal value, in case of tie, ID = 0.
	`dst[2*i] = max(src[2*i], src[2*i+1]);`
	`dst[2*i+1] = (src[2*i] >= src[2*i+1]) ? 0 : 1;`
	This operation is supported only for the Word type, to handle potential overflow of Word-type VMaxRID with scalar register destinations.

9.8.2.20 VSHUFFLE

Instruction name	VSHUFFLE
Functionality	Vector shuffle permutation
Assembly format	VShuffle<type> Vsrc/Wsrc, Vdst/Wdst
Type and bit width	B: 32 x 12-bit H: 16 x 24-bit W: 8 x 48-bit
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vcharx vshuffle(vcharx src);`
	`vshortx vshuffle(vshortx src);`
	`vintx vshuffle(vintx src);`

Additional details	Perform shuffle permutation among byte/halfword/word lanes. Equivalent to VPerm with pattern: Byte: {0, 16, 1, 17, 2, 18, 3, 19, 4, 20, 5, 21, 6, 22, 7, 23, 8, 24, 9, 25, 10, 26, 11, 27, 12, 28, 13, 29, 14, 30, 15, 31} Halfword: {0, 8, 1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 14, 7, 15} Word: {0, 4, 1, 5, 2, 6, 3, 7}

9.8.3 Vector OP12 Instructions

These are one-source, two-destination operation vector instructions.

9.8.3.1 Instruction Summary

Table 31. Vector OP12 instructions

Function	Assembly Format	Comments
Vector sign-magnitude	VSignMag<W/H/B> Vsrc/Wsrc, Vdst1/Wdst1, Vdst2/Wdst2	Vdst1/Wdst2 gets sign values. Vdst2/Wdst2 gets magnitude values.
Vector min reduction & ID	VMinRID<type> Vsrc/Wsrc, Rdst1, Rdst2	dst1 gets the min value. dst2 gets the min ID.
Vector max reduction & ID	VMaxRID<type> Vsrc/Wsrc, Rdst1, Rdst2	dst1 gets the max value. dst2 gets the max ID.
Vector type promotion	VPromote_DI<type> Vsrc/Wsrc, Vdst1/Wdst1, Vdst2/Wdst2 VPromote_DI<type> XACsrc, Vdst1, Vdst2 type = {H, W}	With and without deinterleaving
Vector bit deinterleaving	VBitDeIntrlvW Vsrc/Wsrc, Vdst1/Wdst1, Vdst2/Wdst2 VBitDeIntrlv21W Vsrc/Wsrc, Vdst1/Wdst1, Vdst2/Wdst2	1:1 and 2:1 deinterleaving
Collate index and bits	VCollateIdx_Bits<type> Vsrc/Wsrc, Vdst1/Wdst1, Rdst2	
Vector min skip2 reduction-ID	VMinSkip2RIDW Vsrc/Wsrc, Vdst1/Wdst1, Rdst2	Complete min reduction-ID, assuming src is outcome from VHMin2ID
Vector max skip2 reduction-ID	VMaxSkip2RIDW Vsrc/Wsrc, Vdst1/Wdst1, Rdst2	Complete max reduction-ID, assuming src is outcome from VHMax2ID

9.8.3.2 VSIGNMAG

Instruction name	VSIGNMAG
Functionality	Vector sign magnitude
Assembly format	VSignMag<type> Vsrc/Wsrc, Vdst1/Wdst1, Vdst2/Wdst2
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	dst1: Single vector register in VRF or WRF dst2: single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`void vsignmag(vintx  src, vintx  & dst1, vintx  & dst2);`
	`void vsignmag(vshortx src, vshortx & dst1, vshortx & dst2);`
	`void vsignmag(vcharx src, vcharx & dst1, vcharx & dst2);`
	`// double vector pseudo intrinsics`
	`void dvsignmag(dvintx  src, dvintx  & dst1, dvintx  & dst2);`
	`void dvsignmag(dvshortx src, dvshortx & dst1, dvshortx & dst2);`
	`void dvsignmag(dvcharx src, dvcharx & dst1, dvcharx & dst2);`

Additional details	dst1 gets the sign, 0 for zero/positive and 1 for negative. dst2 gets the magnitude (absolute value).

9.8.3.3 VMINRID

Instruction name	VMINRID
Functionality	Vector min reduction with ID
Assembly format	VMinRID<type> Vsrc/Wsrc, Rdst1, Rdst2
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	dst1, dst2: scalar registers
Additional options	

Intrinsics/operator	`void vminrid_s(vintx  src, int  & dst1, int   & dst2);`
	`void vminrid_s(vshortx src, int  & dst1, int   & dst2);`
	`void vminrid_s(vcharx src, int  & dst1, int   & dst2);`
	`// Following gen-1 legacy intrinsics shall be emulated with multiple instructions`
	`void vminrid(vintx  src, vintx  & dst1, vintx  & dst2);`
	`void vminrid(vshortx src, vshortx & dst1, vshortx & dst2);`
	`void vminrid(vcharx src, vcharx & dst1, vcharx & dst2);`

Additional details	dst1 gets the min value among lanes, 12-bit/24-bit outcome is sign-extended to 32-bit, and 48-bit outcome has 32 LSBs written to the destination with upper 16 bits dropped. dst2 gets lane ID (0 ~ 7/15/31) where the min value is found, lowest lane when there's a tie.

	Gen-1 legacy intrinsics broadcast outcomes across all lanes of destination registers. For W-type 48-bit min value is output in each lane of the first vector destination.
	For Halfword and Byte types, the emulation uses vminrid_s() and replicateh/b().
	For Word type, using just vminrid_s() and replicatew() will not compute bits 47..32 of the extended word lane properly. Instead, the emulation uses vhmin2id() and vminskip2rid(). See 9.8.2.18 and 9.8.3.9 for details.
	
	Examples:
	`VMinRIDB V1, V2, V3 is emulated as`
	`VMinRIDB V1, R2, R3; VMovSB R2, V2; VMovSB R3, V3.`
	`vminrid(vcharx_src, vcharx_dst1, vcharx_dst2) as {`
	`  vminrid_s(vcharx_src, min_dst, id_dst);`
	`  vcharx_dst1 = replicateb(min_dst);`
	`  vcharx_dst2 = replicateb(id_idst);`
	`}`
	
	`VMinRIDH V1, V2, V3 is emulated as`
	`VMinRIDH V1, R2, R3; VMovSH R2, V2; VMovSH R3, V3.`
	`vminrid(vshortx_src, vshortx_dst1, vshortx_dst2) as {`
	`  vminrid_s(vshortx_src, min_dst, id_dst);`
	`  vshortx_dst1 = replicateh(min_dst);`
	`  vshortx_dst2 = replicateh(id_idst);`
	`}`
	
	`VMinRIDW V1, V2, V3 is emulated as`
	`VHMin2IDW V1, V4; VMinSkip2RIDW V4, V2, R2; VMovS R2, V3.`
	`vminrid(vintx_src, vintx_dst1, vintx_dst2) as {`
	`  vhmin2id(vintx_src, temp);`
	`  vminskip2rid(temp, vintx_dst1, id_dst2);`
	`  vintx_dst2 = replicatew(id_dst2);`
	`}`

9.8.3.4 VMAXRID

Instruction name	**VMAXRID**
Functionality	Vector max reduction with ID
Assembly format	VMaxRID<type> Vsrc/Wsrc, Rdst1, Rdst2
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	dst1, dst2: scalar registers
Additional options	
Intrinsics/operator	`void vmaxrid_s(vintx  src, int  & dst1, int   & dst2);` `void vmaxrid_s(vshortx src, int  & dst1, int   & dst2);`

Instruction name	VMAXRID

	`void vmaxrid_s(vcharx src, int  & dst1, int   & dst2);`
	`// Following gen-1 legacy intrinsics shall be emulated with multiple instructions`
	`void vmaxrid(vintx  src, vintx  & dst1, vintx  & dst2);`
	`void vmaxrid(vshortx src, vshortx & dst1, vshortx & dst2);`
	`void vmaxrid(vcharx src, vcharx & dst1, vcharx & dst2);`

Additional details	dst1 gets the max value among lanes, 12-bit/24-bit outcome is sign-extended to 32-bit, and 48-bit outcome has 32 LSBs written to the destination with upper 16 bits dropped.
	dst2 gets lane ID (0 ~ 7/15/31) where the max value is found, lowest lane when there's a tie.
	Gen-1 legacy intrinsics broadcast outcomes across all lanes of destination registers. For W-type 48-bit min value is output in each lane of the first vector destination.
	For Halfword and Byte types, the emulation uses vmaxrid_s() and replicateh/b().
	For Word type, using just vmaxrid_s() and replicatew() will not compute bits 47..32 of the extended word lane properly. Instead, the emulation uses vhmax2id() and vmaxskip2rid(). See 9.8.2.19 and 9.8.3.10 for details.
	
	Examples:
	
	`VMaxRIDB V1, V2, V3 is emulated as`
	`VMaxRIDB V1, R2, R3; VMovSB R2, V2; VMovSB R3, V3.`
	`vmaxrid(vcharx_src, vcharx_dst1, vcharx_dst2) as {`
	`  vmaxrid_s(vcharx_src, max_dst, id_dst);`
	`  vcharx_dst1 = replicateb(max_dst);`
	`  vcharx_dst2 = replicateb(id_idst);`
	`}`
	
	`VMaxRIDH V1, V2, V3 is emulated as`
	`VMaxRIDH V1, R2, R3; VMovSH R2, V2; VMovSH R3, V3.`
	`vmaxrid(vshortx_src, vshortx_dst1, vshortx_dst2) as {`
	`  vmaxrid_s(vshortx_src, max_dst, id_dst);`
	`  vshortx_dst1 = replicateh(max_dst);`
	`  vshortx_dst2 = replicateh(id_idst);`
	`}`
	
	`VMaxRIDW V1, V2, V3 is emulated as`
	`VHMax2IDW V1, V4; VMaxSkip2RIDW V4, V2, R2; VMovS R2, V3.`
	`vmaxrid(vintx_src, vintx_dst1, vintx_dst2) as {`
	`  vhmax2id(vintx_src, temp);`
	`  vmaxskip2rid(temp, vintx_dst1, id_dst2);`
	`  vintx_dst2 = replicatew(id_dst2);`
	`}`

9.8.3.5 VPROMOTE_DI

Instruction name	VPROMOTE_DI
Functionality	Vector type promotion with deinterleaving
Assembly format	VPromote_DI<type> Vsrc/Wsrc, Vdst1/Wdst1, Vdst2/Wdst2
Type and bit width	BH: 32 x 12-bit → 2 x 16 x 24-bit, HW: 16 x 24-bit → 2 x 8 x 48-bit, with sign extension
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	dst1: Single vector register in VRF or WRF dst2: single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`void vpromote_di(vcharx src, vshortx & dst1, vshortx & dst2);`
	`void vpromote_di(vshortx src, vintx  & dst1, vintx  & dst2);`
	`dvshortx vpromote_di(vcharx src);`
	`dvintx vpromote_di(vshortx src);`
	`// double vector pseudo intrinsics unavailable`

Additional details	Expand byte to half-word or half-word to word, with 2 single registers as destination and with deinterleaving. Example: VPromote_DIHW V1, V2, V3 with V1 = {0, 1, 2, …, 15} would copy V1's contents to V2 and V3 deinterleavingly, so that V2 = {0, 2, 4, …, 14} and V3 = {1, 3, 5, …, 15}

Instruction name	VPROMOTE_DI (Gen-2 from XARF to VRF)
Functionality	Vector type promotion with deinterleaving
Assembly format	VPromote_DI<type> XACsrc, Vdst1, Vdst2
Type and bit width	H: 32 x 16-bit → 2 x 16 x 24-bit, W: 16 x 32-bit → 2 x 8 x 48-bit
Predication	not available
Source options	Single vector register in XARF
Destination options	dst1: Single vector register in VRF dst2: single vector register in VRF
Additional options	

Intrinsics/operator	`void vpromote_di(xvcharx src, vshortx& dst1, vshortx& dst2);`
	`void vpromote_di(xvshortx src, vintx&  dst1, vintx&  dst2);`
	`dvshortx vpromote_di(xvcharx src);`
	`dvintx vpromote_di(xvshortx src);`
	`// double vector pseudo intrinsics unavailable`

Additional details	Expand halfword to extended halfword or word to extended word. Example: VPromote_DIH XAC0, V2, V3

Instruction name	VPROMOTE_DI (Gen-2 from XARF to VRF)
	with XAC0 = {0, 1, 2, ..., 31} would copy XAC0 contents to V2 and V3 deinterleavingly, so that V2 = {0, 2, 4, ..., 30} and V3 = {1, 3, 5, ..., 31}

9.8.3.6 VPROMOTE (Emulated)

Instruction name	VPROMOTE (Emulated)
Functionality	Vector type promotion (without deinterleaving)
Assembly format	VPromote<type> Vsrc, Vdst1, Vdst2
Type and bit width	BH: 32 x 12-bit → 2 x 16 x 24-bit, HW: 16 x 24-bit → 2 x 8 x 48-bit, with sign extension
Predication	not available
Source options	Single vector register
Destination options	Single vector register
Additional options	

Intrinsics/operator	`void vpromote(vshortx src, vintx  & dst1, vintx  & dst2);`
	`void vpromote(vcharx src, vshortx & dst1, vshortx & dst2);`
	`dvintx vpromote(vshortx src);`
	`dvshortx vpromote(vcharx src);`
	`// double vector pseudo intrinsics unavailable`

Additional details	Expand byte to half-word or half-word to word, with 2 single registers as destination and without deinterleaving.
	It's an instruction in Gen-1 VPU only. In Gen-2, it was removed to reduce timing pressure in cross-lane unit. The intrinsic is supported by implementing the same functionality using multiple instructions.
	The emulation uses vshuffle() and vpromote_di() intrinsic functions. See 9.8.2.20 and 9.8.3.5 for details.
	
	Examples:
	`VPromoteBH V1, V2, V3 is emulated as`
	`VShuffleB V1, V4; VPromote_DIBH V4, V2, V3.`
	`vpromote(vcharx_src, vshortx_dst1, vshortx_dst2) as {`
	`vpromote_di(vshuffle(vcharx_src), vshortx_dst1, vshortx_dst2); }`
	
	`VPromoteHW V1, V2, V3 is emulated as`
	`VShuffleH V1, V4; VPromote_DIHW V4, V2, V3.`
	`vpromote(vcharx_src, vintx_dst1, vintx_dst2) as {`
	`vpromote_di(vshuffle(vshortx_src), vintx_dst1, vintx_dst2); }`

9.8.3.7 VBITDEINTRLV

Instruction name	VBITDEINTRLV
Functionality	Vector bit deinterleave
Assembly format	VBitDeintrlv<type> Vsrc/Wsrc, Vdst1/Wdst1, Vdst2/Wdst2
Type and bit width	W: 8 x 32-bit → 8 x 16-bit + 8 x 16-bit
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	dst1: Single vector register in VRF or WRF dst2: single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`void vbit_deinterleave(vintx src, vintx & dst1, vintx & dst2);`
	`// double vector pseudo intrinsics`
	`void dvbit_deinterleave(dvintx src, dvintx & dst1, dvintx & dst2);`

Additional details	In each 48-bit W lane, bit-deinterleave src[31:0] into dst1[15:0] and dst2[15:0] dst1[15] = src[31], dst2[15] = src[30], dst1[14] = src[29], dst2[14] = src[28], and so on. dst1[47:16] = dst2[47:16] = 0

Instruction name	VBITDEINTRLV21
Functionality	Vector bit deinterleave 2:1
Assembly format	VBitDeIntrlv21<type> Vsrc/Wsrc, Vdst1/Wdst1, Vdst2/Wdst2
Type and bit width	W: 8 x 48-bit → 8 x 32-bit + 8 x 16-bit
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	dst1: Single vector register in VRF or WRF dst2: single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`void vbit_deinterleave_21(vintx src, vintx & dst1, vintx & dst2);`
	`// double vector pseudo intrinsics`
	`void dvbit_deinterleave_21(dvintx src, dvintx & dst1, dvintx & dst2);`

Additional details	In each 48-bit W lane, bit-deinterleave src[47:0] into dst1[31:0] and dst2[15:0] dst1[31:30] = src[47:46], dst2[15] = src[45], dst1[29:28] = src[44:43], dst2[14] = src[42], and so on. dst1[47:32] = dst2[47:16] = 0

9.8.3.8 VCOLLATEIDX_BITS

Instruction name	VCOLLATEIDX_BITS
Functionality	Vector collate index and bits
Assembly format	VCollateIdx_Bits<type> Vsrc/Wsrc, Vdst1/Wdst1, Rdst2
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit (no byte type support)
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	Single vector registers in VRF or WRF Scalar register
Additional options	

Intrinsics/operator	`void vcollate_idx_bits(vintx src, vintx& dst1, int& dst2);`
	`void vcollate_idx_bits(vshortx src, vshortx& dst1, int& dst2);`
	`// double vector pseudo intrinsics unavailable`

Additional details	Vdst1/Wdst1 gets indices to nonzero lanes (sequentially from lane 0) of Vsrc/Wsrc, followed by indices to zero lanes of Vsrc/Wsrc. Rdst2 gets bit-packed Boolean vector indicating nonzero lanes of Vsrc/Wsrc. For example, VCollateIdxW V0, V1, R2, with V0 = {0, -1, 2, -3, 0, 0, 0, 4}. Non-zero lanes are lane 1, 2, 3, and 7. Expected outcome V1 = {1, 2, 3, 7, 0, 4, 5, 6}, R2 = 0x8E (bits 1, 2, 3, 7 are ones). The idea is that a subsequent VPermW would use V1 as indices to extract/compact V0 nonzero and zero lanes into {-1, 2, -3, 4, 0, 0, 0, 0}. R2 is saved for later-on expanding the nonzeros back to original data array.

9.8.3.9 VMINSkip2RID

Instruction name	VMINSKIP2RID
Functionality	Vector every-other-lane horizontal min reduction and ID
Assembly format	VMinSkip2RID<type> Vsrc/Wsrc, Vdst1/Wdst1, Rdst2
Type and bit width	W: 8 x 48-bit signed
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	dst1: Single vector register in VRF or WRF dst2: scalar register
Additional options	
Intrinsics/operator	`void vminskip2rid(vintx src, vintx& dst1, int& dst2);` `// double vector pseudo intrinsics unavailable`

Additional details	Complete min reduction-ID functionality cross lanes, assuming the source is outcome of VHMin2ID, with even lanes containing min values and LSB of odd lanes containing 1-bit min-ID (between lanes 2*I and 2*I+1).
	`min_val = min(src[0], src[2], src[4], src[6]);`
	`id_val = (src[0] == min_val) ? src[1][0] :`
	`((src[2] == min_val) ? (2 + src[3][0]) :`

Instruction name	VMINSKIP2RID

	((src[4] == min_val) ? (4 + src[5][0]) : (6 + src[7][0])));
	dst1[0..7] = min_val;
	dst2 = id_val;
	
	src[i][0] means bit 0 of src[i].
	This operation is supported only for the Word type, to handle potential overflow of Word-type VMinRID with scalar register destinations.
	For example, start with V0 = {3, 1, 4, 0, 2, 5, 9, 1}, holding 8 int48 values.
	After VHMin2IDW V0, V1, we shall have
	V1 = {1, 1, 0, 1, 2, 0, 1, 1}, this is because
	min(3, 1) = 1 from odd lane, min(4, 0) = 0 from odd lane, and so on.
	While executing VMinSkip2RID V1, V2, R3, we have
	min_val = min(1, 0, 2, 1) = 0,
	and we have src1[2] == min_val, so id_val = 2 + src[3][0] = 2+1 = 3.
	We return
	V2 = {0, 0, 0, 0, 0, 0, 0, 0} and R3 = 3

9.8.3.10 VMAXSkip2RID

Instruction name	VMAXSKIP2RID
Functionality	Vector every-other-lane horizontal max reduction and ID
Assembly format	VMaxSkip2RID<type> Vsrc/Wsrc, Vdst1/Wdst1, Rdst2
Type and bit width	W: 8 x 48-bit signed
Predication	not available
Source options	Single vector register in VRF or WRF
Destination options	dst1: Single vector register in VRF or WRF dst2: scalar register
Additional options	
Intrinsics/operator	void vmaxskip2rid(vintx src, vintx& dst1, int& dst2); // double vector pseudo intrinsics unavailable

Additional details	Complete max reduction-ID functionality cross lanes, assuming the source is outcome of VHMax2ID, with even lanes containing max values and LSB of odd lanes containing 1-bit max-ID (between lanes 2*I and 2*I+1).
	max_val = max(src[0], src[2], src[4], src[6]);
	id_val = (src[0] == max_val) ? src[1] :
	((src[2] == max_val) ? (2 + src[3]) :
	((src[4] == max_val) ? (4 + src[5]) : (6 + src[7])));
	dst1[0..7] = max_val;
	dst2 = id_val;
	This operation is supported only for the Word type, to handle potential overflow of Word-type MaxRID with scalar register destinations.
	See example in VMinSkip2RID instruction description.

9.8.4 Vector OP21 Instructions

These are two-source, one-destination operation vector instructions.

9.8.4.1 Instruction Summary

For some of these two-source, one-destination instructions, source 2 can be vector register or scalar register. In case of scalar register, its value is broadcast to all lanes before the operation, depending on the type:

- For Byte-type operations, bits 11:0 of scalar register is broadcast to all extended byte lanes (12-bit each).
- For Halfword-type operations, bits 23:0 of scalar register is broadcast to all extended halfword lanes (24-bit each).
- For Word-type operations, the 32-bit scalar register is sign-extended to extended word lanes (48-bit each).

Double vector flavor is supported for selected operators.

Table 32 Vector OP21 instructions

Function	Assembly Format	Comments
Vector and bitwise	VAnd<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst VAnd<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VAnd<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst	
Vector and logical	VAndL<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	
Vector or bitwise	VOr<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst VOr<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VOr<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst	
Vector or logical	VOrL<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	
Vector exclusive or	VXor<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst VXor<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VXor<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst	
Vector add	VAdd<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	

Function	Assembly Format	Comments
	VAdd<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VAdd<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst	
Vector subtract	VSub<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst VSub<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VSub<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst	
Vector absolute difference	VAbsDif<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst VAbsDif<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VAbsDif<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst	
Vector min	VMin<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst VMin<W/H/B> DVsrc1/Wsrc1, DVsrc2/Wsrc2/Rsrc2, DVdst/Wdst	
Vector max	VMax<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst VMax<W/H/B> DVsrc1/Wsrc1, DVsrc2/Wsrc2/Rsrc2, DVdst/Wdst	
Vector shift	VShift<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	src1 carries data. src2 carries bit counts, when positive shift left, when negative shift right. Bit counts are saturated to [-12, 12], [-24, 24] or [-48, 48] range before applying the shift.
Vector shift left	VSLA<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	src1 carries data. src2 carries bit counts, saturated to [0, 12], [0, 24], [0, 48] before applying the left shift.
Vector shift right	VSRA<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	src1 carries data. src2 carries bit counts, saturated to [0, 12], [0, 24], [0, 48] range before applying the right shift.
Vector round	VRound<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	src1 carries data. src2 carries bit counts, saturated to [0, 12], [0, 24], [0, 48] range before applying the right shift.
Vector permute	VPerm<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	src1 carries data.

Function	Assembly Format	Comments
		src2 carries permute pattern in corresponding lane, value i for lane i. Only 5/4/3 LSBs are read as unsigned indices for W/H/B type.
Vector compare GT	VCmpGT<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst VCmpGT<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VCmpGT<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst	
Vector compare GE	VCmpGE<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst VCmpGE<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VCmpGE<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst	
Vector compare LT	VCmpLT<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst VCmpLT<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VCmpLT<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst	
Vector compare LE	VCmpLE<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst VCmpLE<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VCmpLE<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst	
Vector compare EQ	VCmpEQ<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst VCmpEQ<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VCmpEQ<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst	
Vector compare NE	VCmpNE<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst VCmpNE<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VCmpNE<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst	

Function	Assembly Format	Comments
Vector compare and bit-pack	VBitCmp<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst/Rdst type = {BBW, H, WWB}	Compare src1 >= src2, bit-pack outcome, then broadcast to all lanes
Vector normalize	VNorm<W/H/B> Vsrc1/Wsrc1, Vsrc2/Wsrc2, Vdst/Wdst	Normalize src1 data with most-significant bit position src2
Vector octant detect	VOctDetH Vsrc1/Wsrc1, Vsrc2/Wsrc2, Vdst/Wdst	src1 = Y values, src2 = X values, detect octant of (X, Y) vectors.
Vector type demotion	VDemote_I<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2, Vdst/Wdst VDemote<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2, Vdst/Wdst type = {HB, WH} VDemote_I<type> Vsrc1, Vsrc2, XACdst type = {H, W}	Type demotion with and without interleaving
Vector bit interleaving	VBitIntrlvW Vsrc1/Wsrc1, Vsrc2/Wsrc2, Vdst/Wdst VBitIntrlv21W Vsrc1/Wsrc1, Vsrc2/Wsrc2, Vdst/Wdst	Bit interleaving, 1:1 and 2:1
Vector apply sign	VApplySign<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2, Vdst/Wdst	Inverse operation of VSignMag, treating src1 as sign (0 for zero/positive and 1 for negative), and src2 as magnitude.
Vector select lane	VSelectLane<type> Vsrc1/Wsrc1, Rsrc2, Rdst	

9.8.4.2 VAND

Instruction name	VAND
Functionality	Vector bitwise AND
Assembly format	VAnd<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit Scalar operand: W: full 32-bit **zero-extended** to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  operator&(vintx  src1, vintx src2);`
	`vshortx operator&(vshortx src1, vshortx src2);`
	`vcharx operator&(vcharx src1, vcharx src2);`
	`vintx  operator&(vintx  src1, unsigned int src2);`
	`vshortx operator&(vshortx src1, unsigned int src2);`
	`vcharx operator&(vcharx src1, unsigned int src2);`

Additional details	

Instruction name	VAND
Functionality	Double vector bitwise AND
Assembly format	VAnd<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VAnd<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst
Type and bit width	Vector operand: W: 2 x 8 x 48-bit, H: 2 x 16 x 24-bit, B: 2 x 32 x 12-bit Scalar operand: W: full 32-bit zero-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: double vector register in VRF or WRF src2: double vector register in VRF or WRF, or scalar register
Destination options	dst: double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`dvintx  operator&(dvintx  src1, dvintx src2);`
	`dvshortx operator&(dvshortx src1, dvshortx src2);`
	`dvcharx operator&(dvcharx src1, dvcharx src2);`
	`dvintx  operator&(dvintx  src1, unsigned int src2);`
	`dvshortx operator&(dvshortx src1, unsigned int src2);`
	`dvcharx operator&(dvcharx src1, unsigned int src2);`

Additional details	

9.8.4.3 VANDL

Instruction name	VANDL
Functionality	Vector logical AND
Assembly format	VAndL<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit Scalar operand: W/H/B: **full 32-bit** detected logically then broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  operator&&(vintx  src1, vintx src2);`
	`vshortx operator&&(vshortx src1, vshortx src2);`
	`vcharx operator&&(vcharx src1, vcharx src2);`
	`vintx  operator&&(vintx  src1, unsigned int src2);`
	`vshortx operator&&(vshortx src1, unsigned int src2);`
	`vcharx operator&&(vcharx src1, unsigned int src2);`
	`// double vector pseudo intrinsics`
	`dvintx  operator&&(dvintx  src1, dvintx src2);`
	`dvshortx operator&&(dvshortx src1, dvshortx src2);`
	`dvcharx operator&&(dvcharx src1, dvcharx src2);`
	`dvintx  operator&&(dvintx  src1, unsigned int src2);`
	`dvshortx operator&&(dvshortx src1, unsigned int src2);`
	`dvcharx operator&&(dvcharx src1, unsigned int src2);`

Additional details	

9.8.4.4 VOR

Instruction name	VOR
Functionality	Vector bitwise OR
Assembly format	VOr<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit Scalar operand: W: full 32-bit **zero-extended** to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	
Intrinsics/operator	`vintx  operator|(vintx  src1, vintx src2);` `vshortx operator|(vshortx src1, vshortx src2);`

Instruction name	VOR

	`vcharx operator|(vcharx src1, vcharx src2);`
	`vintx  operator|(vintx  src1, unsigned int src2);`
	`vshortx operator|(vshortx src1, unsigned int src2);`
	`vcharx operator|(vcharx src1, unsigned int src2);`

Additional details	

Instruction name	VOR
Functionality	Double vector bitwise OR
Assembly format	VOr<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VOr<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst
Type and bit width	Vector operand: W: 2 x 8 x 48-bit, H: 2 x 16 x 24-bit, B: 2 x 32 x 12-bit Scalar operand: W: full 32-bit zero-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: double vector register in VRF or WRF src2: double vector register in VRF or WRF, or scalar register
Destination options	dst: double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`dvintx  operator|(dvintx  src1, dvintx src2);`
	`dvshortx operator|(dvshortx src1, dvshortx src2);`
	`dvcharx operator|(dvcharx src1, dvcharx src2);`
	`dvintx  operator|(dvintx  src1, unsigned int src2);`
	`dvshortx operator|(dvshortx src1, unsigned int src2);`
	`dvcharx operator|(dvcharx src1, unsigned int src2);`

Additional details	

9.8.4.5 VORL

Instruction name	VORL
Functionality	Vector logical OR
Assembly format	VOrL Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit Scalar operand: W/H/B: **full 32-bit** detected logically then broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  operator||(vintx  src1, vintx src2);`
	`vshortx operator||(vshortx src1, vshortx src2);`
	`vcharx operator||(vcharx src1, vcharx src2);`

Instruction name	VORL

	`vintx  operator||(vintx  src1, unsigned int src2);`
	`vshortx operator||(vshortx src1, unsigned int src2);`
	`vcharx operator||(vcharx src1, unsigned int src2);`
	`// double vector pseudo intrinsics`
	`dvintx  operator||(dvintx  src1, dvintx src2);`
	`dvshortx operator||(dvshortx src1, dvshortx src2);`
	`dvcharx operator||(dvcharx src1, dvcharx src2);`
	`dvintx  operator||(dvintx  src1, unsigned int src2);`
	`dvshortx operator||(dvshortx src1, unsigned int src2);`
	`dvcharx operator||(dvcharx src1, unsigned int src2);`

Additional details	

9.8.4.6 VXOR

Instruction name	VXOR
Functionality	Vector bitwise exclusive or
Assembly format	VXor<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit Scalar operand: W: full 32-bit **zero-extended** to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  operator^(vintx  src1, vintx src2);`
	`vshortx operator^(vshortx src1, vshortx src2);`
	`vcharx operator^(vcharx src1, vcharx src2);`
	`vintx  operator^(vintx  src1, unsigned int src2);`
	`vshortx operator^(vshortx src1, unsigned int src2);`
	`vcharx operator^(vcharx src1, unsigned int src2);`

Additional details	

Instruction name	VXOR
Functionality	Double vector bitwise exclusive or
Assembly format	VXor<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VXor<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst
Type and bit width	Vector operand: W: 2 x 8 x 48-bit, H: 2 x 16 x 24-bit, B: 2 x 32 x 12-bit Scalar operand: W: full 32-bit zero-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: double vector register in VRF or WRF

Instruction name	VXOR
	src2: double vector register in VRF or WRF, or scalar register
Destination options	dst: double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`dvintx  operator^(dvintx  src1, dvintx src2);`
	`dvshortx operator^(dvshortx src1, dvshortx src2);`
	`dvcharx operator^(dvcharx src1, dvcharx src2);`
	`dvintx  operator^(dvintx  src1, unsigned int src2);`
	`dvshortx operator^(dvshortx src1, unsigned int src2);`
	`dvcharx operator^(dvcharx src1, unsigned int src2);`

Additional details	

9.8.4.7 VADD

Instruction name	VADD
Functionality	Vector add
Assembly format	VAdd<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  operator+(vintx  src1, vintx src2);`
	`vshortx operator+(vshortx src1, vshortx src2);`
	`vcharx operator+(vcharx src1, vcharx src2);`
	`vintx  operator+(vintx  src1, int src2);`
	`vshortx operator+(vshortx src1, int src2);`
	`vcharx operator+(vcharx src1, int src2);`

Additional details	

Instruction name	VADD
Functionality	Double vector add
Assembly format	VAdd<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VAdd<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst
Type and bit width	Vector operand: W: 2 x 8 x 48-bit, H: 2 x 16 x 24-bit, B: 2 x 32 x 12-bit Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: double vector register in VRF or WRF

Instruction name	VADD
	src2: double vector register in VRF or WRF, or scalar register
Destination options	dst: double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`dvintx  operator+(dvintx  src1, dvintx src2);`
	`dvshortx operator+(dvshortx src1, dvshortx src2);`
	`dvcharx operator+(dvcharx src1, dvcharx src2);`
	`dvintx  operator+(dvintx  src1, int src2);`
	`dvshortx operator+(dvshortx src1, int src2);`
	`dvcharx operator+(dvcharx src1, int src2);`

Additional details	

9.8.4.8 VSUB

Instruction name	VSUB
Functionality	Vector subtract
Assembly format	VSub<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  operator-(vintx  src1, vintx src2);`
	`vshortx operator-(vshortx src1, vshortx src2);`
	`vcharx operator-(vcharx src1, vcharx src2);`
	`vintx  operator-(vintx  src1, int src2);`
	`vshortx operator-(vshortx src1, int src2);`
	`vcharx operator-(vcharx src1, int src2);`

Additional details	

Instruction name	VSUB
Functionality	Double vector subtract
Assembly format	VSub<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VSub<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst
Type and bit width	Vector operand: W: 2 x 8 x 48-bit, H: 2 x 16 x 24-bit, B: 2 x 32 x 12-bit Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: double vector register in VRF or WRF

Instruction name	VSUB
	src2: double vector register in VRF or WRF, or scalar register
Destination options	dst: double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`dvintx  operator-(dvintx  src1, dvintx src2);`
	`dvshortx operator-(dvshortx src1, dvshortx src2);`
	`dvcharx operator-(dvcharx src1, dvcharx src2);`
	`dvintx  operator-(dvintx  src1, int src2);`
	`dvshortx operator-(dvshortx src1, int src2);`
	`dvcharx operator-(dvcharx src1, int src2);`

Additional details	

9.8.4.9 VABSDIF

Instruction name	VABSDIF
Functionality	Vector absolute difference
Assembly format	VAbsDif<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  vabsdif(vintx  src1, vintx  src2);`
	`vshortx vabsdif(vshortx src1, vshortx src2);`
	`vcharx vabsdif(vcharx src1, vcharx src2);`
	`vintx  vabsdif(vintx  src1, int   src2);`
	`vshortx vabsdif(vshortx src1, int   src2);`
	`vcharx vabsdif(vcharx src1, int   src2);`

Additional details	

Instruction name	VABSDIF
Functionality	Double vector absolute difference
Assembly format	VAbsDif<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VAbsDif<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst
Type and bit width	Vector operand: W: 2 x 8 x 48-bit, H: 2 x 16 x 24-bit, B: 2 x 32 x 12-bit Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: double vector register in VRF or WRF

Instruction name	VABSDIF
	src2: double vector register in VRF or WRF, or scalar register
Destination options	dst: double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`dvintx  dvabsdif(dvintx  src1, dvintx  src2);`
	`dvshortx dvabsdif(dvshortx src1, dvshortx src2);`
	`dvcharx dvabsdif(dvcharx src1, dvcharx src2);`
	`dvintx  dvabsdif(dvintx  src1, int   src2);`
	`dvshortx dvabsdif(dvshortx src1, int   src2);`
	`dvcharx dvabsdif(dvcharx src1, int   src2);`

Additional details	

9.8.4.10 VMIN

Instruction name	VMIN
Functionality	Vector min
Assembly format	VMin<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  vmin(vintx  src1, vintx  src2);`
	`vshortx vmin(vshortx src1, vshortx src2);`
	`vcharx vmin(vcharx src1, vcharx src2);`
	`vintx  vmin(vintx  src1, int   src2);`
	`vshortx vmin(vshortx src1, int   src2);`
	`vcharx vmin(vcharx src1, int   src2);`

Additional details	Return minimal of 2 inputs

Instruction name	VMIN
Functionality	Double vector min
Assembly format	VMin<type> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VMin<type> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst
Type and bit width	Vector operand: W: 16 x 48-bit, H: 32 x 24-bit, B: 64 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: double vector register in VRF or WRF src2: double vector register in VRF or WRF, or scalar register

Instruction name	VMIN
Destination options	dst: double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`dvintx  dvmin(dvintx  src1, dvintx  src2);`
	`dvshortx dvmin(dvshortx src1, dvshortx src2);`
	`dvcharx dvmin(dvcharx src1, dvcharx src2);`
	`dvintx  dvmin(dvintx  src1, int   src2);`
	`dvshortx dvmin(dvshortx src1, int   src2);`
	`dvcharx dvmin(dvcharx src1, int   src2);`

Additional details	Return minimal of 2 inputs

9.8.4.11 VMAX

Instruction name	VMAX
Functionality	Vector max
Assembly format	VMax<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  vmax(vintx  src1, vintx  src2);`
	`vshortx vmax(vshortx src1, vshortx src2);`
	`vcharx vmax(vcharx src1, vcharx src2);`
	`vintx  vmax(vintx  src1, int   src2);`
	`vshortx vmax(vshortx src1, int   src2);`
	`vcharx vmax(vcharx src1, int   src2);`

Additional details	Return maximal of 2 inputs

Instruction name	VMAX
Functionality	Double vector max
Assembly format	VMax<type> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VMax<type> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst
Type and bit width	Vector operand: W: 16 x 48-bit, H: 32 x 24-bit, B: 64 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: double vector register in VRF or WRF src2: double vector register in VRF or WRF, or scalar register

Instruction name	VMAX
Destination options	dst: double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`dvintx  dvmax(dvintx  src1, dvintx  src2);`
	`dvshortx dvmax(dvshortx src1, dvshortx src2);`
	`dvcharx dvmax(dvcharx src1, dvcharx src2);`
	`dvintx  dvmax(dvintx  src1, int   src2);`
	`dvshortx dvmax(dvshortx src1, int   src2);`
	`dvcharx dvmax(dvcharx src1, int   src2);`

Additional details	Return minimal of 2 inputs

9.8.4.12 VSHIFT

Instruction name	VSHIFT
Functionality	Vector shift
Assembly format	VShift<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  vshift(vintx  src1, vintx  src2);`
	`vshortx vshift(vshortx src1, vshortx src2);`
	`vcharx vshift(vcharx src1, vcharx src2);`
	`vintx  vshift(vintx  src1, int   src2);`
	`vshortx vshift(vshortx src1, int   src2);`
	`vcharx vshift(vcharx src1, int   src2);`
	`// double vector pseudo intrinsics`
	`dvintx  dvshift(dvintx  src1, dvintx  src2);`
	`dvshortx dvshift(dvshortx src1, dvshortx src2);`
	`dvcharx dvshift(dvcharx src1, dvcharx src2);`
	`dvintx  dvshift(dvintx  src1, int   src2);`
	`dvshortx dvshift(dvshortx src1, int   src2);`
	`dvcharx dvshift(dvcharx src1, int   src2);`

Additional details	When the lane value in src2 is positive, perform left shift, otherwise perform right shift, -k indicating >> k. Each 12/24/48-bit lane of Vsrc2/Wsrc2 or lower 12/24/32-bit of Rsrc2 is read as a signed number, and saturated to [-12, 12], [-24, 24], [-48, 48] range before detecting sign and applying the shift.

9.8.4.13 VSLA

Instruction name	VSLA
Functionality	Vector shift left
Assembly format	VSLA<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  operator<<(vintx  src1, vintx  src2);`
	`vshortx operator<<(vshortx src1, vshortx src2);`
	`vcharx operator<<(vcharx src1, vcharx src2);`
	`vintx  operator<<(vintx  src1, int   src2);`
	`vshortx operator<<(vshortx src1, int   src2);`
	`vcharx operator<<(vcharx src1, int   src2);`
	`// double vector pseudo intrinsics`
	`dvintx  operator<<(dvintx  src1, dvintx  src2);`
	`dvshortx operator<<(dvshortx src1, dvshortx src2);`
	`dvcharx operator<<(dvcharx src1, dvcharx src2);`
	`dvintx  operator<<(dvintx  src1, int   src2);`
	`dvshortx operator<<(dvshortx src1, int   src2);`
	`dvcharx operator<<(dvcharx src1, int   src2);`

Additional details	Each 12/24/48-bit lane of Vsrc2/Wsrc2 or lower 12/24/32-bit of Rsrc2 is read as a signed number, and saturated to [0, 12], [0, 24], [0, 48] range before applying the shift.

9.8.4.14 VSRA

Instruction name	VSRA
Functionality	Vector shift right
Assembly format	VSRA<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  operator>>(vintx  src1, vintx  src2);`
	`vshortx operator>>(vshortx src1, vshortx src2);`
	`vcharx operator>>(vcharx src1, vcharx src2);`
	`vintx  operator>>(vintx  src1, int   src2);`
	`vshortx operator>>(vshortx src1, int   src2);`
	`vcharx operator>>(vcharx src1, int   src2);`
	`// double vector pseudo intrinsics`
	`dvintx  operator>>(dvintx  src1, dvintx  src2);`
	`dvshortx operator>>(dvshortx src1, dvshortx src2);`
	`dvcharx operator>>(dvcharx src1, dvcharx src2);`
	`dvintx  operator>>(dvintx  src1, int   src2);`
	`dvshortx operator>>(dvshortx src1, int   src2);`
	`dvcharx operator>>(dvcharx src1, int   src2);`

Additional details	Shift right arithmetic (preserving sign). Unsigned data should be loaded with appropriate unsigned type of vector load, and zero-padded when loading into extended vector lanes. Each 12/24/48-bit lane of Vsrc2/Wsrc2 or lower 12/24/32-bit of Rsrc2 is read as a signed number, and saturated to [0, 12], [0, 24], [0, 48] range before applying the shift.

9.8.4.15 VROUND

Instruction name	VROUND
Functionality	Vector round
Assembly format	VRound<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  vround(vintx  src1, vintx  src2);`
	`vshortx vround(vshortx src1, vshortx src2);`
	`vcharx vround(vcharx src1, vcharx src2);`
	`vintx  vround(vintx  src1, int   src2);`
	`vshortx vround(vshortx src1, int   src2);`
	`vcharx vround(vcharx src1, int   src2);`
	`// double vector pseudo intrinsics`
	`dvintx  dvround(dvintx  src1, dvintx  src2);`
	`dvshortx dvround(dvshortx src1, dvshortx src2);`
	`dvcharx dvround(dvcharx src1, dvcharx src2);`
	`dvintx  dvround(dvintx  src1, int   src2);`
	`dvshortx dvround(dvshortx src1, int   src2);`
	`dvcharx dvround(dvcharx src1, int   src2);`

Additional details	In case of Vsrc2/Wsrc2, each lane gets an independent number of bits to round. In case of Rsrc2, lower 12/24/32 bits are broadcast so all lanes are rounded by the same number of bits.
	Number of bits to round is read as a signed number and saturated to [0, 12], [0, 24], or [0, 48] range before being applied to the rounding.
	Let rnd_bits be number of bits to round. Rounding is performed in each lane where rnd_bits >= 1, by
	`temp1 = src1 >> (rnd_bits – 1);`
	`dst = (temp1 + 1) >> 1;`
	In lanes where rnd_bits <= 0, dst = src1 (no rounding).
	Note that rounding by the lane width or more bits would result in 0 for both positive and negative inputs.

9.8.4.16 VPERM

Instruction name	VPERM
Functionality	Vector permute
Assembly format	VPerm<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit Scalar operand: W: full 32-bit **zero-extended** to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  vpermute(vintx  src1, vintx  src2);`
	`vshortx vpermute(vshortx src1, vshortx src2);`
	`vcharx vpermute(vcharx src1, vcharx src2);`
	`vfloatx vpermute(vfloatx src1, vintx  src2);`
	`vhfloatx vpermute(vhfloatx src1, vshortx  src2);`
	
	`vintx  vpermute(vintx  src1, int   src2);`
	`vshortx vpermute(vshortx src1, int   src2);`
	`vcharx vpermute(vcharx src1, int   src2);`
	`vfloatx vpermute(vfloatx src1,  int  src2);`
	`vhfloatx vpermute(vhfloatx src1, int  src2);`
	
	`// double vector pseudo intrinsics unavailable`

Additional details	Treat src1 as lane data and src2 as lane indices. For each lane, return value of the lane pointed to by the index. Only 3/4/5 LSBs are read as unsigned indices for W/H/B type, rest are ignored. For example, say if we start with V0 = {1, 3, 5, 7, 9, 11, 13, 15} in W lanes V1 = {4, 5, 6, 7, 0, 0, 1, 1} in W lanes VPermW V0, V1, V2 would result in V2 = {9, 11, 13, 15, 1, 1, 3, 3} in W lanes When using scalar register as src2, the value in 3/4/5 LSBs of the scalar register is used to select one of 8/16/32 W/H/B lanes of src1, and value in the selected lane is replicated in all lanes of the destination.

9.8.4.17 VCMPGT

Instruction name	VCMPGT
Functionality	Vector compare greater than
Assembly format	VCmpGT<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  operator>(vintx  src1, vintx  src2);`
	`vshortx operator>(vshortx src1, vshortx src2);`
	`vcharx operator>(vcharx src1, vcharx src2);`
	`vintx  operator>(vintx  src1, int src2);`
	`vshortx operator>(vshortx src1, int src2);`
	`vcharx operator>(vcharx src1, int src2);`

Additional details	

Instruction name	VCMPGT
Functionality	Double vector compare greater than
Assembly format	VCmpGT<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VCmpGT<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst
Type and bit width	Vector operand: W: 2 x 8 x 48-bit, H: 2 x 16 x 24-bit, B: 2 x 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: double vector register in VRF or WRF src2: double vector register in VRF or WRF, or scalar register
Destination options	dst: double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`dvintx  operator>(dvintx  src1, dvintx  src2);`
	`dvshortx operator>(dvshortx src1, dvshortx src2);`
	`dvcharx operator>(dvcharx src1, dvcharx src2);`
	`dvintx  operator>(dvintx  src1, int src2);`
	`dvshortx operator>(dvshortx src1, int src2);`
	`dvcharx operator>(dvcharx src1, int src2);`

Additional details	

9.8.4.18 VCMPGE

Instruction name	VCMPGE
Functionality	Vector compare greater than or equal
Assembly format	VCmpGE<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  operator>=(vintx  src1, vintx  src2);`
	`vshortx operator>=(vshortx src1, vshortx src2);`
	`vcharx operator>=(vcharx src1, vcharx src2);`
	`vintx  operator>=(vintx  src1, int src2);`
	`vshortx operator>=(vshortx src1, int src2);`
	`vcharx operator>=(vcharx src1, int src2);`

Additional details	

Instruction name	VCMPGE
Functionality	Double vector compare greater than or equal
Assembly format	VCmpGE<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VCmpGE<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst
Type and bit width	Vector operand: W: 2 x 8 x 48-bit, H: 2 x 16 x 24-bit, B: 2 x 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: double vector register in VRF or WRF src2: double vector register in VRF or WRF, or scalar register
Destination options	dst: double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`dvintx  operator>=(dvintx  src1, dvintx  src2);`
	`dvshortx operator>=(dvshortx src1, dvshortx src2);`
	`dvcharx operator>=(dvcharx src1, dvcharx src2);`
	`dvintx  operator>=(dvintx  src1, int src2);`
	`dvshortx operator>=(dvshortx src1, int src2);`
	`dvcharx operator>=(dvcharx src1, int src2);`

Additional details	

9.8.4.19 VCMPLT

Instruction name	VCMPLT
Functionality	Vector compare less than
Assembly format	VCmpLT<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  operator<(vintx  src1, vintx  src2);`
	`vshortx operator<(vshortx src1, vshortx src2);`
	`vcharx operator<(vcharx src1, vcharx src2);`
	`vintx  operator<(vintx  src1, int src2);`
	`vshortx operator<(vshortx src1, int src2);`
	`vcharx operator<(vcharx src1, int src2);`

Additional details	

Instruction name	VCMPLT
Functionality	Double vector compare less than
Assembly format	VCmpLT<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VCmpLT<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst
Type and bit width	Vector operand: W: 2 x 8 x 48-bit, H: 2 x 16 x 24-bit, B: 2 x 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: double vector register in VRF or WRF src2: double vector register in VRF or WRF, or scalar register
Destination options	dst: double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`dvintx  operator<(dvintx  src1, dvintx  src2);`
	`dvshortx operator<(dvshortx src1, dvshortx src2);`
	`dvcharx operator<(dvcharx src1, dvcharx src2);`
	`dvintx  operator<(dvintx  src1, int src2);`
	`dvshortx operator<(dvshortx src1, int src2);`
	`dvcharx operator<(dvcharx src1, int src2);`

Additional details	

9.8.4.20 VCMPLE

Instruction name	VCMPLE
Functionality	Vector compare less than or equal
Assembly format	VCmpLE<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  operator<=(vintx  src1, vintx  src2);`
	`vshortx operator<=(vshortx src1, vshortx src2);`
	`vcharx operator<=(vcharx src1, vcharx src2);`
	`vintx  operator<=(vintx  src1, int src2);`
	`vshortx operator<=(vshortx src1, int src2);`
	`vcharx operator<=(vcharx src1, int src2);`

Additional details	

Instruction name	VCMPLE
Functionality	Double vector compare less than or equal
Assembly format	VCmpLE<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VCmpLE<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst
Type and bit width	Vector operand: W: 2 x 8 x 48-bit, H: 2 x 16 x 24-bit, B: 2 x 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: double vector register in VRF or WRF src2: double vector register in VRF or WRF, or scalar register
Destination options	dst: double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`dvintx  operator<=(dvintx  src1, dvintx  src2);`
	`dvshortx operator<=(dvshortx src1, dvshortx src2);`
	`dvcharx operator<=(dvcharx src1, dvcharx src2);`
	`dvintx  operator<=(dvintx  src1, int src2);`
	`dvshortx operator<=(dvshortx src1, int src2);`
	`dvcharx operator<=(dvcharx src1, int src2);`

Additional details	

9.8.4.21 VCMPEQ

Instruction name	VCMPEQ
Functionality	Vector compare equal
Assembly format	VCmpEQ<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  operator==(vintx  src1, vintx  src2);`
	`vshortx operator==(vshortx src1, vshortx src2);`
	`vcharx operator==(vcharx src1, vcharx src2);`
	`vintx  operator==(vintx  src1, int src2);`
	`vshortx operator==(vshortx src1, int src2);`
	`vcharx operator==(vcharx src1, int src2);`

Additional details	

Instruction name	VCMPEQ
Functionality	Double vector compare equal
Assembly format	VCmpEQ<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VCmpEQ<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst
Type and bit width	Vector operand: W: 2 x 8 x 48-bit, H: 2 x 16 x 24-bit, B: 2 x 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: double vector register in VRF or WRF src2: double vector register in VRF or WRF, or scalar register
Destination options	dst: double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`dvintx  operator==(dvintx  src1, dvintx  src2);`
	`dvshortx operator==(dvshortx src1, dvshortx src2);`
	`dvcharx operator==(dvcharx src1, dvcharx src2);`
	`dvintx  operator==(dvintx  src1, int src2);`
	`dvshortx operator==(dvshortx src1, int src2);`
	`dvcharx operator==(dvcharx src1, int src2);`

Additional details	

9.8.4.22 VCMPNE

Instruction name	VCMPNE
Functionality	Vector compare not equal
Assembly format	VCmpNE<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx  operator!=(vintx  src1, vintx  src2);`
	`vshortx operator!=(vshortx src1, vshortx src2);`
	`vcharx operator!=(vcharx src1, vcharx src2);`
	`vintx  operator!=(vintx  src1, int src2);`
	`vshortx operator!=(vshortx src1, int src2);`
	`vcharx operator!=(vcharx src1, int src2);`

Additional details	

Instruction name	VCMPNE
Functionality	Double vector compare not equal
Assembly format	VCmpNE<W/H/B> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VCmpNE<W/H/B> DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst
Type and bit width	Vector operand: W: 2 x 8 x 48-bit, H: 2 x 16 x 24-bit, B: 2 x 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: double vector register in VRF or WRF src2: double vector register in VRF or WRF, or scalar register
Destination options	dst: double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`dvintx  operator!=(dvintx  src1, dvintx  src2);`
	`dvshortx operator!=(dvshortx src1, dvshortx src2);`
	`dvcharx operator!=(dvcharx src1, dvcharx src2);`
	`dvintx  operator!=(dvintx  src1, int src2);`
	`dvshortx operator!=(dvshortx src1, int src2);`
	`dvcharx operator!=(dvcharx src1, int src2);`

Additional details	

9.8.4.23 VBITCMP

Instruction name	VBITCMP
Functionality	Vector compare and bit-pack
Assembly format	VBitCmp<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst/Rdst
Type and bit width	Vector operand: WWB: 8 x 48-bit compare as signed → 8-bit → broadcast to B lanes. H: 16 x 24-bit compare as signed → 16-bit → broadcast to H lanes. BBW: 32 x 12-bit compare as signed → 32-bit → broadcast to W lanes. Scalar operand: WWB: full 32-bit sign-extended to 48-bit, H: 24 LSBs, BBW: 12 LSBs, compare as signed.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF, or scalar register
Destination options	dst: Single vector register in VRF or WRF, or scalar register
Additional options	

Intrinsics/operator	`vintx  vbitcmp(vcharx vsrc1, vcharx vsrc2);`
	`vshortx vbitcmp(vshortx vsrc1, vshortx vsrc2);`
	`vcharx vbitcmp(vintx  vsrc1, vintx  vsrc2);`
	`vintx  vbitcmp(vcharx vsrc1, int   vsrc2);`
	`vshortx vbitcmp(vshortx vsrc1, int   vsrc2);`
	`vcharx vbitcmp(vintx  vsrc1, int   vsrc2);`
	`int vbitcmp_s(vcharx vsrc1, vcharx vsrc2);`
	`int vbitcmp_s(vshortx vsrc1, vshortx vsrc2);`
	`int vbitcmp_s(vintx  vsrc1, vintx  vsrc2);`
	`int vbitcmp_s(vcharx vsrc1, int   vsrc2);`
	`int vbitcmp_s(vshortx vsrc1, int   vsrc2);`
	`int vbitcmp_s(vintx  vsrc1, int   vsrc2);`
	`// double vector pseudo intrinsics unavailable`

Additional details	Compare src1 >= src2 in each W/H/B lane, compact to 8/16/32-bit scalar, broadcast to all destination B/H/W lanes. For example, say if we start with V0 = {1, 3, 5, 7, 9, 11, 13, 15} in W lanes V1 = {5, 5, 5, 5, 10, 10, 10, 10} in W lanes VBitCmpWWB V0, V1, V2 would result in {0,0,1,1,0,1,1,1} = 0xEC, V2 = {0xEC, 0xEC, …, 0xEC} in B lanes For scalar destination, in WWB-type, the 8-bit scalar is zero-extended to 32-bit and returned. In H-type, the 16-bit scalar is zero-extended to 32-bit and returned. In BBW-type, the 32-bit scalar is returned.

9.8.4.24 VNORM

Instruction name	VNORM
Functionality	Vector normalize
Assembly format	VNorm<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2, Vdst/Wdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vcharx vnorm(vcharx vsrc1, vcharx vsrc2);`
	`vshortx vnorm(vshortx vsrc1, vshortx vsrc2);`
	`vintx  vnorm(vintx  vsrc1, vintx  vsrc2);`
	`// double vector pseudo intrinsics`
	`dvintx  dvnorm(dvintx  vsrc1, dvintx  vsrc2);`
	`dvshortx dvnorm(dvshortx vsrc1, dvshortx vsrc2);`
	`dvcharx dvnorm(dvcharx vsrc1, dvcharx vsrc2);`

Additional details	Each 12/24/48-bit lane of Vsrc2/Wsrc2 is read as an signed number, 7 – src2, 15 – src2, or 31 – src2 is performed, outcome saturated to [-12, 12], [-24, 24], [-48, 48] range, then src1 is shifted by this many bits. Arithmetic shift is performed to preserve sign bit when shifting right. The intention is to precede VNorm with VMSBD, so that src2 holds the most significant bit position of src1. VNorm would then shift the most significant bit (left or right) to bit 7 for B, bit 15 for H, bit 31 for W. For example, when src2 = 11, the shift amount is 7 – 11 = -4, to shift src1 right by 4 bits. When src2 = 5, the shift amount is 7 – 5 = 2, to shift src1 left by 2 bits.

9.8.4.25 VOCTDET

Instruction name	**VOCTDET**
Functionality	Vector octant detection for atan2(Y, X)
Assembly format	VOctDetH Vsrc1/Wsrc1, Vsrc2/Wsrc2, Vdst/Wdst
Type and bit width	H: 16 x 24-bit, signed
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vshortx voct_detect(vshortx vsrc1, vshortx vsrc2);`
	`// double vector pseudo intrinsics`
	`dvshortx dvoct_detect(dvshortx vsrc1, dvshortx vsrc2);`

Additional details	Treat Vsrc1/Wsrc1 as Y, Vsrc2/Wsrc2 as X, return octant of (X, Y) in 2D plane, 0 ~ 7. First (0^{th}) octant from 0 to 44.999 degree, second (1th) from 45 to 89.999 degree, etc, 0 degree being the X axis. Condition Octant and angle range X>=0, Y>=0, |Y|<=|X| 0: [0 ~ 0.25 pi] X>=0, Y>=0, |Y|>|X| 1: (0.25 pi ~ 0.5 pi) X<0, Y>=0, |Y|>|X| 2: (0.5 pi ~ 0.75 pi) X<0, Y>=0, |Y|<=|X| 3: [0.75 pi ~ pi] X<0, Y<0, |Y|<=|X| 4: (pi ~ 1.25 pi) X<0, Y<0, |Y|>|X| 5: (1.25 pi ~ 1.5 pi) X>=0,Y<0, |Y|>|X| 6: [1.5 pi ~ 1.75 pi] X>=0,Y<0, |Y|<=|X| 7: [1.75 pi ~ 2 pi]

9.8.4.26 VDEMOTE_I

Instruction name	VDEMOTE_I
Functionality	Vector type demotion with interleaving
Assembly format	VDemote_I<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2, Vdst/Wdst
Type and bit width	HB: 2 x 16 x 24-bit → 32 x 12-bit, WH: 2 x 8 x 48-bit → 16 x 24-bit
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`void vdemote_i(vshortx src1, vshortx src2, vcharx & dst);`
	`void vdemote_i(vintx  src1, vintx  src2, vshortx & dst);`
	`vcharx vdemote_i(dvshortx src);`
	`vshortx vdemote_i(dvintx src);`
	`// double vector pseudo intrinsics unavailable`

Additional details	Compress half-word to byte or word to half-word, with 2 single registers as source and with interleaving. For HB, lower 12 bits of the source lane is copied to the destination. For WH, lower 24 bits. Programmer should be aware of the possibility of overflow. Example: VDemote_IWH V1, V2, V3 with V1 = {0, 1, 2, ..., 7} and V2 = {8, 9, ..., 15} would copy V1 and V2 contents to V3 interleavingly, such that V3 = {0, 8, 1, 9, ..., 7, 15}

Instruction name	VDEMOTE_I (Gen-2 from VRF to XARF)
Functionality	Vector type demotion with interleaving
Assembly format	VDemote_I<type> Vsrc1, Vsrc2, XACdst
Type and bit width	H: 2 x 16 x 24-bit → 32 x 16-bit, W: 2 x 8 x 48-bit → 16 x 32-bit
Predication	not available
Source options	src1: Single vector register in VRF src2: single vector register in VRF
Destination options	Single vector register in XARF
Additional options	

Intrinsics/operator	`void vdemote_i(vshortx src1, vshortx src2, xvcharx &dst);`
	`void vdemote_i(vintx  src1, vintx  src2, xvshortx &dst);`
	`// double vector pseudo intrinsics unavailable`

Additional details	Compress extended halfword to halfword or extended word to word. Example: VDemote_IH V0, V1, XAC2 with V0 = {0, 1, 2, ..., 15} and V1 = {16, 17, ..., 31} would copy V0 and V1 contents to V3 interleavingly, such that

Instruction name	VDEMOTE_I (Gen-2 from VRF to XARF)
	XAC2 = {0, 16, 1, 17, ..., 15, 31}.

9.8.4.27 VDEMOTE

Instruction name	VDEMOTE
Functionality	Vector type demotion
Assembly format	VDemote<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2, Vdst/Wdst
Type and bit width	HB: 2 x 16 x 24-bit → 32 x 12-bit, WH: 2 x 8 x 48-bit → 16 x 24-bit
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`void vdemote(vintx src1,  vintx src2, vshortx & dst);`
	`void vdemote(vshortx src1, vshortx src2, vcharx & dst);`
	`vcharx vdemote(dvshortx src);`
	`vshortx vdemote(dvintx src);`
	`// double vector pseudo intrinsics unavailable`

Additional details	Compress half-word to byte or word to half-word, with 2 single registers as source and without interleaving. For HB, lower 12 bits of the source lane is copied to the destination. For WH, lower 24 bits. Programmer should be aware of the possibility of overflow. Example: VDemoteWH V1, V2, V3 with V1 = {0, 1, 2, ..., 7} and V2 = {8, 9, ..., 15} would copy V1 and V2 contents to V3 sequentially, such that V3 = {0, 1, 2, ..., 15}

9.8.4.28 VBITINTRLV

Instruction name	VBITINTRLV
Functionality	Vector bit interleave
Assembly format	VBitIntrlv<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2, Vdst/Wdst
Type and bit width	W: 8 x 16-bit + 8 x 16-bit → 8 x 32-bit
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx vbit_interleave(vintx src1, vintx src2);`
	`// double vector pseudo intrinsics`
	`dvintx dvbit_interleave(dvintx src1, dvintx src2);`

Additional details	In each 48-bit W lane, bit-interleave src1[15:0] and src2[15:0] into dst dst[31] = src1[15], dst[30] = src2[15], dst[29] = src1[14], dst[28] = src2[14], and so on. dst[47:32] = 0.

Instruction name	VBITINTRLV21
Functionality	Vector bit interleave 2:1
Assembly format	VBitIntrlv21<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2, Vdst/Wdst
Type and bit width	W: 8 x 32-bit + 8 x 16-bit → 8 x 48-bit
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx vbit_interleave_21(vintx src1, vintx src2);`
	`// double vector pseudo intrinsics`
	`dvintx dvbit_interleave_21(dvintx src1, dvintx src2);`

Additional details	In each 48-bit W lane, bit-interleave src1[31:0] and src2[15:0] into dst in 2-bit, 1-bit pattern. dst[47:46] = src1[31:30], dst[45] = src2[15], dst[44:43] = src1[29:28], dst[42] = src2[14], and so on.

9.8.4.29 VAPPLYSIGN

Instruction name	VAPPLYSIGN
Functionality	Vector apply sign
Assembly format	VApplySign<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2, Vdst/Wdst
Type and bit width	B: 32 x 12-bit H: 16 x 24-bit W: 8 x 48-bit
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF
Destination options	dst: Single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vcharx vapply_sign(vcharx src1, vcharx src2);`
	`vshortx vapply_sign(vshortx src1, vshortx src2);`
	`vintx vapply_sign(vintx src1, vintx src2);`
	`// double vector pseudo intrinsics`
	`dvcharx dvapply_sign(dvcharx src1, dvcharx src2);`
	`dvshortx dvapply_sign(dvshortx src1, dvshortx src2);`
	`dvintx dvapply_sign(dvintx src1, dvintx src2);`

Additional details	Inverse operation of VSignMag (vector sign-magnitude). The intention is that in each vector lane (width depending on data type), we treat src1 as a Boolean carrying the sign bit that we want to apply on src2 that carries the magnitude. The hardware performs dst = src1 ? –src2 : src2 so that output gets negated src2 when src1 is true (non-zero), and src2 as-is when src1 is false (zero). Thus, in addition to applying a sign to a magnitude, it can be used to conditionally flip the sign of any number. Note that this instruction does not perform absolute value function on src2 before applying the sign.

9.8.4.30 VSelectLane

Instruction name	VSelectLane
Functionality	Vector select lane
Assembly format	VSelectLane<type> Vsrc1/Wsrc1, Rsrc2, Rdst
Type and bit width	Vector operand: W: 32-bit H: 24-bit sign-extend to 32-bit B: 12-bit sign-extend to 32-bit Scalar operand: W: 3 LSBs, H: 4 LSBs, B: 5 LSBs
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: scalar register
Destination options	dst: destination register
Additional options	

Intrinsics/operator	`int vselectlane(vintx  src1, int src2);`
	`int vselectlane(vshortx src1, int src2);`
	`int vselectlane(vcharx src1, int src2);`
	`float vselectlane(vfloatx src1, int src2);`
	`hfloat vselectlane(vhfloatx src1,int src2);`
	`// double vector pseudo intrinsics unavailable`

Additional details	Treat src1 as lane data and src2 as lane index. For W-type, 32 LSBs of the selected 48-bit lane is returned in the destination scalar register. Programmer should be aware of potential overflow in the outcome. For H/B-type, selected 12-bit/24-bit is sign-extend to 32-bit in the destination register. Only 3/4/5 LSBs are read as unsigned indices for W/H/B type, rest are ignored.

9.8.5 Vector OP22 Instructions

These are two-source, two-destination operation vector instructions.

9.8.5.1 Instruction Summary

Table 33. Vector OP22 instructions

Function	Assembly Format	Comments
Non-overwriting		
Vector 2-item sort	VSort2<W/H/B> Vsrc1, Vsrc2, Vdst1, Vdst2	Vdst1 = min(Vsrc1, Vsrc2); Vdst2 = max(Vsrc1, Vsrc2);
Vector add/subtract	VAddSub<W/H/B> Vsrc1, Vsrc2, Vdst1, Vdst2	Vdst1 = Vsrc1 + Vsrc2; Vdst2 = Vsrc1 – Vsrc2;
Vector complex add/sub	VCAddSubH Vsrc1, Vsrc2, Vdst1, Vdst2	Like VAddSub but swap even/odd lanes of Vsrc2 and add/subtract, see details
Vector min-LT-flag	VMinLT<W/H/B> Vsrc1, Vsrc2, Vdst1, Vdst2	Vdst1 = min(Vsrc1, Vsrc2); Vdst2 = Vsrc1 < Vsrc2;
Vector max-GT-flag	VMaxGT<W/H/B> Vsrc1, Vsrc2, Vdst1, Vdst2	Vdst1 = max(Vsrc1, Vsrc2); Vdst2 = Vsrc1 > Vsrc2;
Vector 2-item sort with payload	VSort2PL<W/H/B> Vsrc1, Vsrc2, Vdst1, Vdst2	Key and payload interleaved in each source and destination vector register
Vector split bits	VSplitBits Vsrc1, Vsrc2, Vdst1, Vdst2	Split src1 into two right-justified bit fields

9.8.5.2 VSORT2

Instruction name	VSORT2
Functionality	Vector 2-point sort
Assembly format	VSort2<type> Vsrc1, Vsrc2, Vdst1, Vdst2
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF
Destination options	dst1: single vector register in VRF dst2: single vector register in VRF
Additional options	

Intrinsics/operator	`void vsort2(vintx src1, vintx src2, vintx & dst1, vintx & dst2);`
	`void vsort2(vshortx src1, vshortx src2, vshortx & dst1, vshortx & dst2);`
	`void vsort2(vcharx src1, vcharx src2, vcharx & dst1, vcharx & dst2);`

Instruction name	VSORT2

	// double vector pseudo intrinsics
	void dvsort2(dvintx src1, dvintx src2, dvintx & dst1, dvintx & dst2);
	void dvsort2(dvshortx src1, dvshortx src2, dvshortx & dst1,dvshortx & dst2);
	void dvsort2(dvcharx src1, dvcharx src2, dvcharx & dst1, dvcharx & dst2);

Additional details	For each lane, dst1 = min(src1, src2), dst2 = max(src1, src2)

9.8.5.3 VADDSUB

Instruction name	VADDSUB
Functionality	Vector add-subtract
Assembly format	VAddSub<type> Vsrc1, Vsrc2, Vdst1, Vdst2
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF
Destination options	dst1: single vector register in VRF dst2: single vector register in VRF
Additional options	

Intrinsics/operator	void vaddsub(vintx src1, vintx src2, vintx & dst1, vintx & dst2);
	void vaddsub(vshortx src1, vshortx src2, vshortx & dst1, vshortx & dst2);
	void vaddsub(vcharx src1, vcharx src2, vcharx & dst1, vcharx & dst2);
	// double vector pseudo intrinsics
	void dvaddsub(dvintx src1, dvintx src2, dvintx & dst1, dvintx & dst2);
	void dvaddsub(dvshortx src1, dvshortx src2, dvshortx & dst1,dvshortx & dst2);
	void dvaddsub(dvcharx src1, dvcharx src2, dvcharx & dst1, dvcharx & dst2);

Additional details	For each lane, dst1 = src1 + src2, dst2 = src1 – src2.

9.8.5.4 VCADDSUB

Instruction name	VCADDSUB
Functionality	Vector add-subtract
Assembly format	VCAddSub<type> Vsrc1, Vsrc2, Vdst1, Vdst2
Type and bit width	H: 16 x 24-bit
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF
Destination options	dst1: single vector register in VRF dst2: single vector register in VRF
Additional options	
Intrinsics/operator	void vcaddsub(vshortx src1, vshortx src2, vshortx & dst1, vshortx & dst2);

Instruction name	VCADDSUB
	`// double vector pseudo intrinsics` `void dvcaddsub(dvshortx src1, dvshortx src2, dvshortx & dst1, dvshortx & dst2);`

Additional details	Even lanes, dst1[2*i] = src1[2*i] + src2[2*i+1]
	dst2[2*i] = src1[2*i] – src2[2*i+1]
	Odd lanes, , dst1[2*i+1] = src1[2*i+1] – src2[2*i]
	dst2[2*i+1] = src1[2*i+1] + src2[2*i]
	This is for 16-bit FFT acceleration, where real and imaginary components are interleaved, even lanes being real, odd lanes being imaginary.
	We are implementing rotating complex number src2 by +/- 90 degree and adding to src1:
	`dst1 = src1 – j*src2;`
	`dst2 = src1 + j*src2;`
	Thus,
	`dst1[2*i] (R) = src1[2*i] (R) + src2[2*i+1] (I)`
	`dst1[2*i+1] (I) = src1[2*i+1] (I) – src2[2*i] (R)`
	`dst2[2*i] (R) = src1[2*i] (R) – src2[2*i+1] (I)`
	`dst2[2*i+1] (I) = src1[2*i+1] (I) + src2[2*i] (R)`

9.8.5.5 VMINLT

Instruction name	VMINLT
Functionality	Vector min-less-than-flag
Assembly format	VMinLT<type> Vsrc1, Vsrc2, Vdst1, Vdst2
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF
Destination options	dst1: single vector register in VRF dst2: single vector register in VRF
Additional options	

Intrinsics/operator	`void vminLT(vintx src1, vintx src2, vintx & dst1, vintx & dst2);`
	`void vminLT(vshortx src1, vshortx src2, vshortx & dst1, vshortx & dst2);`
	`void vminLT(vcharx src1, vcharx src2, vcharx & dst1, vcharx & dst2);`
	`// double vector pseudo intrinsics`
	`void dvminLT(dvintx src1, dvintx src2, dvintx & dst1, dvintx & dst2);`
	`void dvminLT(dvshortx src1, dvshortx src2, dvshortx & dst1, dvshortx & dst2);`
	`void dvminLT(dvcharx src1, dvcharx src2, dvcharx & dst1, dvcharx & dst2);`

Additional details	For each lane, dst1 = min(src1, src2), dst2 = (src1 < src2), so that flag = 1 indicating src1 being the min, and 0 indicating src2 being the min.

9.8.5.6 VMAXGT

Instruction name	VMAXGT
Functionality	Vector max-greater-than-flag
Assembly format	VMaxGT<type> Vsrc1, Vsrc2, Vdst1, Vdst2
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF
Destination options	dst1: single vector register in VRF dst2: single vector register in VRF
Additional options	

Intrinsics/operator	`void vmaxGT(vintx src1, vintx src2, vintx & dst1, vintx & dst2);`
	`void vmaxGT(vshortx src1, vshortx src2, vshortx & dst1, vshortx & dst2);`
	`void vmaxGT(vcharx src1, vcharx src2, vcharx & dst1, vcharx & dst2);`
	`// double vector pseudo intrinsics`
	`void dvmaxGT(dvintx src1, dvintx src2, dvintx & dst1, dvintx & dst2);`
	`void dvmaxGT(dvshortx src1, dvshortx src2, dvshortx & dst1, dvshortx & dst2);`
	`void dvmaxGT(dvcharx src1, dvcharx src2, dvcharx & dst1, dvcharx & dst2);`

Additional details	For each lane, dst1 = max(src1, src2), dst2 = (src1 > src2), so that flag = 1 indicating src1 being the max, and 0 indicating src2 being the max.

9.8.5.7 VSORT2PL

Instruction name	VSORT2PL
Functionality	Vector 2-item sort with payload
Assembly format	VSort2PL<type> Vsrc1, Vsrc2, Vdst1, Vdst2
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF
Destination options	dst1: single vector register in VRF dst2: single vector register in VRF
Additional options	

Intrinsics/operator	`void vsort2pl(vintx src1, vintx src2, vintx & dst1, vintx & dst2);`
	`void vsort2pl(vshortx src1, vshortx src2, vshortx & dst1, vshortx & dst2);`
	`void vsort2pl(vcharx src1, vcharx src2, vcharx & dst1, vcharx & dst2);`
	`// double vector pseudo intrinsics`
	`void dvsort2pl(dvintx src1, dvintx src2, dvintx & dst1, dvintx & dst2);`
	`void dvsort2pl(dvshortx src1, dvshortx src2, dvshortx & dst1, dvshortx & dst2);`
	`void dvsort2pl(dvcharx src1, dvcharx src2, dvcharx & dst1, dvcharx & dst2);`

Additional details	Key and payload are lane-interleaved; even lanes carry key, odd lanes carry payload.

Instruction name	VSORT2PL

	For each pair of lanes 2*i and 2*i+1:
	`if (src1[2*i] <= src2]2*i]) {`
	`  dst1[2*i] = src1[2*i];`
	`  dst2[2*i] = src2[2*i];`
	`  dst1[2*i+1] = src1[2*i+1];`
	`  dst2[2*i+1] = src2[2*i+1];`
	`}`
	`else {`
	`  dst1[2*i] = src2[2*i];`
	`  dst2[2*i] = src1[2*i];`
	`  dst1[2*i+1] = src2[2*i+1];`
	`  dst2[2*i+1] = src1[2*i+1];`
	`}`

9.8.5.8 VSPLITBITS

Instruction name	VSPLITBITS
Functionality	Vector split bit fields and right-justify
Assembly format	VSplitBits<type> Vsrc1, Vsrc2, Vdst1, Vdst2
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF
Destination options	dst1: single vector register in VRF dst2: single vector register in VRF
Additional options	

Intrinsics/operator	`void vsplitbits(vintx src1, vintx src2, vintx & dst1, vintx & dst2);`
	`void vsplitbits(vshortx src1, vshortx src2, vshortx & dst1, vshortx & dst2);`
	`void vsplitbits(vcharx src1, vcharx src2, vcharx & dst1, vcharx & dst2);`
	`// double vector pseudo intrinsics`
	`void dvsplitbits(dvintx src1, dvintx src2, dvintx & dst1, dvintx & dst2);`
	`void dvsplitbits(dvshortx src1, dvshortx src2, dvshortx & dst1, dvshortx & dst2);`
	`void dvsplitbits(dvcharx src1, dvcharx src2, dvcharx & dst1, dvcharx & dst2);`

Additional details	Each lane of src2 is read as a signed number and saturated to [0, 48], [0, 24], [0, 12] to obtain the bit position T. Each lane of src1 is read as a signed number. dst1 (signed) gets right-justified upper bits of src1, from bit T and up. dst2 (unsigned) gets lower bits of src1, from bit T-1 down.
	Pseudo-code for the Halfword case:
	`T = (src2 < 0) ? 0 : ((src2 > 24) ? 24 : src2);`
	`mask = (1 << T) – 1;`
	`dst1 = src1 >> T;`
	`dst2 = src1 & mask;`

9.8.6 Vector OP31 Instructions

These are three-source, one-destination operation vector instructions.

9.8.6.1 Instruction Summary

The subset of three-source, one-destination instructions with "_CA" suffix support the "clear-accumulator" feature. They are optionally predicated but not predicated in the conventional sense of being executed or skipped. They are predicated to execute one of two different functionalities, and one being a subset of the other to clear the accumulators.

For example, [P2] VMin3W_CA V0, V1, V2 does

V2 = min(V0, V1, V2) when P2 != 0

V2 = min(V0, V1) otherwise

This is used to carry out cumulative minimum operation, with V2 being the accumulator. When the predicate is off, the minimum is carried out without V2, effectively clearing the accumulator.

The _CA suffix is also used in a few vector multiply-add, multiply-subtract instructions in the Vector Multiply-Add Instruction section.

The _CA predicated instructions are overwriting using the 3rd operand as both the 3rd source and the destination. This is so there's room in the encoding for the additional predication field.

The non-CA instructions in the Vector OP31 group are non-overwriting, with destination being a separate field. Compiler can opt to assign the same register as the 3rd source and destination, to accomplish overwriting.

Note that valid predicate registers are P2…P15 for predication. P0 and P1 are reserved for unpredicated execution of the full functionality (min of 3 items in case of VMin3, for example), and in assembly listing, the leading [P0] or [P1] would be omitted to indicate unpredicated execution.

Table 34 Vector OP31 instructions

Function	Assembly Format	Comments
Vector multiplexor	VMux<W/H/B> Vsrc1, Vsrc2, Vsrc3, Vdst VMux<W/H/B> Wsrc1, Vsrc2, Vsrc3, Vdst VMux<W/H/B> Vsrc1, Wsrc2, Vsrc3, Vdst VMux<W/H/B> Vsrc1, Vsrc2, Wsrc3, Vdst	Vdst = (src1 != 0) ? src2 : src3
Vector multiplexor with scalar src2	VMux<W/H/B> Vsrc1, Rsrc2, Vsrc3, Vdst	Vdst = (Vsrc1 != 0) ? Rsrc2 : Vsrc3
Double vector multiplexor	VMux<type> DVsrc1, DWsrc2, DVsrc3, DVdst1 VMux<type> DVsrc1, DVsrc2, DWsrc3, DVdst1 VMux<type> DVsrc1, Rsrc2, DVsrc3, DVdst1	Vdst = (src1 != 0) ? src2 : src3

Function	Assembly Format	Comments
Vector mid of 3	VMid3<W/H/B> Vsrc1, Vsrc2, Vsrc3, Vdst VMid3<W/H/B> Vsrc1, Vsrc2, Wsrc3, Vdst VMid3<W/H/B> Vsrc1, Wsrc2, Wsrc3, Vdst	Vdst = mid(src1, src2, src3)
Vector A+B-C	VAdd2Sub<W/H/B> Vsrc1, Vsrc2, Vsrc3/Rsrc3, Vdst	Vdst = Vsrc1 + Vsrc2 – Vsrc3
Vector shift-or	VShiftOr<W/H/B> Vsrc1, Vsrc2/Rsrc2, Vsrc3, Vdst	Vdst = VShift(Vsrc1, Vsrc2) | Vsrc3.
Vector shift-add	VShiftAdd<W/H/B> Vsrc1, Vsrc2/Rsrc2, Vsrc3, Vdst	Vdst = Vshift(Vsrc1, Vsrc2) + Vsrc3.
Vector extract bits	VExtrBits<W/H/B> Vsrc1, Vsrc2, Vsrc3/Rsrc3, Vdst	Extract low, high bits from src3, 8-bit each. Shift(Vsrc1, Vsrc2) then AND with bit mask between low and high bit positions.
Vector atan2 post-processing	VAtan2PPH Vsrc1, Vsrc2, Vsrc3, Vdst	Vsrc1 = Y, Vsrc2 = X, detect octant of (X, Y) vector then map Vsrc3 angle from first-octan arctan angle to the appropriate octant.
Vector min of 3	VMin3<W/H/B> Vsrc1, Vsrc2, Vsrc3, Vdst VMin3<W/H/B> Vsrc1, Vsrc2, Wsrc3, Vdst VMin3<W/H/B> Vsrc1, Wsrc2, Wsrc3, Vdst	Vdst = min(src1, src2, src3)
Vector max of 3	VMax3<W/H/B> Vsrc1, Vsrc2, Vsrc3, Vdst VMax3<W/H/B> Vsrc1, Vsrc2, Wsrc3, Vdst VMax3<W/H/B> Vsrc1, Wsrc2, Wsrc3, Vdst	Vdst = max(src1, src2, src3)
Vector add 3	VAdd3<W/H/B> Vsrc1, Vsrc2, Vsrc3, Vdst	Vdst = Vsrc1 + Vsrc2 + Vsrc3
Vector bitwise-and 3	VAnd3 Vsrc1, Vsrc2/, Vsrc3, Vdst	Vdst = Vsrc1 & Vsrc2 & Vsrc3
Vector bitwise-or 3	VOr3 Vsrc1, Vsrc2, Vsrc3, Vdst	Vdst = Vsrc1 | Vsrc2 | Vsrc3
Vector bitwise-xor 3	VXor3 Vsrc1, Vsrc2, Vsrc3, Vdst	Vdst = Vsrc1 ^ Vsrc2 ^ Vsrc3
Vector min of 3, predicated	<pred> VMin3<W/H/B>_CA Vsrc1, Vsrc2, ACsrc3dst	
Vector max of 3, predicated	<pred> VMax3<W/H/B>_CA Vsrc1, Vsrc2, ACsrc3dst	
Vector add 3, predicated	<pred> VAdd3<W/H/B>_CA Vsrc1, Vsrc2, ACsrc3dst	
Vector bitwise-and 3, predicated	<pred> VAnd3_CA Vsrc1, Vsrc2, Vsrc3dst	
Vector bitwise-or 3, predicated	<pred> VOr3_CA Vsrc1, Vsrc2, Vsrc3dst	
Vector bitwise-xor 3, predicated	<pred> VXor3_CA Vsrc1, Vsrc2, Vsrc3dst	
Vector sum of absolute differences	<pred> VSAD<W/H/B/BH/HW>_CA Vsrc1, Vsrc2, ACsrc3dst/DACsrc3dst	

Function	Assembly Format	Comments
Vector sum of Hamming distance	<pred> VSumHD<W/H/B>_CA Vsrc1, Vsrc2/Rsrc2, ACsrc3dst	
Vector compare LT and AndL	VCmpLT_AndL<W/H/B> Vsrc1, Vsrc2, Vsrc3, Vdst	
Vector compare LE and AndL	VCmpLE_AndL<W/H/B> Vsrc1, Vsrc2, Vsrc3, Vdst	
Vector compare EQ and AndL	VCmpEQ_AndL<W/H/B> Vsrc1, Vsrc2, Vsrc3, Vdst	
Vector compare NE and AndL	VCmpNE_AndL<W/H/B> Vsrc1, Vsrc2, Vsrc3, Vdst	
Vector compare LT and OrL	VCmpLT_OrL<W/H/B> Vsrc1, Vsrc2, Vsrc3, Vdst	
Vector compare LE and OrL	VCmpLE_OrL<W/H/B> Vsrc1, Vsrc2, Vsrc3, Vdst	
Vector compare EQ and OrL	VCmpEQ_OrL<W/H/B> Vsrc1, Vsrc2, Vsrc3, Vdst	
Vector compare NE and OrL	VCmpNE_OrL<W/H/B> Vsrc1, Vsrc2, Vsrc3, Vdst	
Vector cross-element shift right	VXShiftR <type> Vsrc1, Vsrc2, Rsrc3/Vsrc3, Vdst	
Vector cross-element shift left	VXShiftL <type> Vsrc1, Vsrc2, Rsrc3/Vsrc3, Vdst	

9.8.6.2 VMUX

Instruction name	VMUX
Functionality	Vector multiplexor
Assembly format	VMux<type> Vsrc1, Vsrc2, Vsrc3, Vdst1 VMux<type> Wsrc1, Vsrc2, Vsrc3, Vdst1 VMux<type> Vsrc1, Wsrc2, Vsrc3, Vdst1 VMux<type> Vsrc1, Vsrc2, Wsrc3, Vdst1
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit
Predication	not available
Source options	src1: single vector register in VRF or WRF src2: single vector register in VRF or WRF src3: single vector register in VRF or WRF
Destination options	dst1: single vector register in VRF
Additional options	

Intrinsics/operator	`vintx  vmux(vintx  src1, vintx  src2, vintx src3);`
	`vshortx vmux(vshortx src1, vshortx src2,vshortx src3);`
	`vcharx vmux(vcharx src1, vcharx src2, vcharx src3);`
	`vhfloatx vmux(vshortx src1,vhfloatx src2,vhfloatx src3);`
	`vfloatx vmux(vintx src1, vfloatx src2, vfloatx src3);`

Additional details	For each lane, dst1 = src1 ? src2 : src3.

9.8.6.3 VMUX (Rsrc2)

Instruction name	VMUX
Functionality	Vector multiplexor with scalar src2
Assembly format	VMux<type> Vsrc1, Rsrc2, Vsrc3, Vdst1
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF src2: scalar register src3: single vector register in VRF
Destination options	dst1: single vector register in VRF
Additional options	

Intrinsics/operator	`vintx  vmux(vintx  src1, int src2, vintx  src3);`
	`vshortx vmux(vshortx src1, int src2, vshortx src3);`
	`vcharx vmux(vcharx src1, int src2, vcharx src3);`
	`vhfloatx vmux(vshortx src1, hfloat src2,vhfloatx src3);`
	`vfloatx vmux(vintx  src1, float src2, vfloatx src3);`

Additional details	For each lane, dst1 = src1 ? src2 : src3.

Instruction name	VMUX
	src2 is from a scalar register, its lower 12/24/32-bit is broadcast to all vector lanes as the “true” data input.

9.8.6.4 VMUX (Double Vector)

Instruction name	VMUX (Gen-2 double vector/double throughput)
Functionality	Vector multiplexor
Assembly format	VMux<type> DVsrc1, DWsrc2, DVsrc3, DVdst1 VMux<type> DVsrc1, DVsrc2, DWsrc3, DVdst1 VMux<type> DVsrc1, Rsrc2, DVsrc3, DVdst1
Type and bit width	W: 2 x 8 x 48-bit, H: 2 x 16 x 24-bit, B: 2 x 32 x 12-bit
Predication	not available
Source options	src1: double vector register in VRF src2: double vector register in VRF or WRF, or scalar register src3: double vector register in VRF or WRF
Destination options	dst1: double vector register in VRF
Additional options	

Intrinsics/operator	`dvintx  dvmux(dvintx  src1, dvintx  src2, dvintx src3);`
	`dvshortx dvmux(dvshortx src1, dvshortx src2,dvshortx src3);`
	`dvcharx dvmux(dvcharx src1, dvcharx src2, dvcharx src3);`
	`dvhfloatx dvmux(dvshortx src1, dvhfloatx src2,dvhfloatx src3);`
	`dvfloatx dvmux(dvintx  src1, dvfloatx src2, dvfloatx src3);`
	`dvintx  dvmux(dvintx  src1, int src2, dvintx src3);`
	`dvshortx dvmux(dvshortx src1, int src2, dvshortx src3);`
	`dvcharx  dvmux(dvcharx src1, int src2, dvcharx src3);`
	`dvhfloatx dvmux(dvshortx src1, hfloat src2, dvhfloatx src3);`
	`dvfloatx dvmux(dvintx  src1, float src2, dvfloatx src3);`

Additional details	For each lane, dst1 = src1 ? src2 : src3.

9.8.6.5 VMID3

Instruction name	VMID3
Functionality	Vector median3
Assembly format	VMid3<type> Vsrc1, Vsrc2, Vsrc3, Vdst1 VMid3<type> Vsrc1, Vsrc2, Wsrc3, Vdst VMid3<type> Vsrc1, Wsrc2, Wsrc3, Vdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF or WRF src3: single vector register in VRF or WRF
Destination options	dst1: single vector register in VRF
Additional options	

Intrinsics/operator	`vintx vmid3(vintx src1, vintx src2, vintx src3);`
	`vshortx vmid3(vshortx src1, vshortx src2, vshortx src3);`
	`vcharx vmid3(vcharx src1, vcharx src2, vcharx src3);`
	`// double vector pseudo intrinsics`
	`dvintx dvmid3(dvintx src1, dvintx src2, dvintx src3);`
	`dvshortx dvmid3(dvshortx src1, dvshortx src2, dvshortx src3);`
	`dvcharx dvmid3(dvcharx src1, dvcharx src2, dvcharx src3);`

Additional details	For each lane, return median of 3 sources.

9.8.6.6 VADD2SUB

Instruction name	VADD2SUB (to change intrinsic to +/- operators)
Functionality	Vector add then subtract
Assembly format	VAdd2Sub<type> Vsrc1, Vsrc2, Vsrc3/Rsrc3, Vdst1
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF src3: single vector register in VRF or scalar register
Destination options	dst1: single vector register in VRF
Additional options	

Intrinsics/operator	`vintx vadd2sub(vintx src1, vintx src2, vintx sc3);`
	`vshortx vadd2sub(vshortx src1, vshortx src2, vshortx sc3);`
	`vcharx vadd2sub(vcharx src1, vcharx src2, vcharx sc3);`
	`vintx vadd2sub(vintx src1, vintx src2, int sc3);`
	`vshortx vadd2sub(vshortx src1, vshortx src2, int sc3);`
	`vcharx vadd2sub(vcharx src1, vcharx src2, int sc3);`

Instruction name	VADD2SUB (to change intrinsic to +/- operators)

	`// double vector pseudo intrinsics`
	`dvintx  dvadd2sub(dvintx  src1, dvintx  src2, dvintx  sc3);`
	`dvshortx dvadd2sub(dvshortx src1, dvshortx src2, dvshortx sc3);`
	`dvcharx dvadd2sub(dvcharx src1, dvcharx src2, dvcharx sc3);`
	`dvintx  dvadd2sub(dvintx  src1, dvintx  src2, int sc3);`
	`dvshortx dvadd2sub(dvshortx src1, dvshortx src2, int sc3);`
	`dvcharx dvadd2sub(dvcharx src1, dvcharx src2, int sc3);`

Additional details	For each lane, dst1 = src1 + src2 – src3.

9.8.6.7 VSHIFTOR

Instruction name	VSHIFTOR
Functionality	Vector shift-or
Assembly format	VShiftOr<type> Vsrc1, Vsrc2/Rsrc2, Vsrc3, Vdst1
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes, as signed.
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF or scalar register src3: single vector register in VRF
Destination options	dst1: single vector register in VRF
Additional options	

Intrinsics/operator	`vintx  vshiftor(vintx  src1, vintx  src2, vintx  src3);`
	`vshortx vshiftor(vshortx src1, vshortx src2, vshortx src3);`
	`vcharx vshiftor(vcharx src1, vcharx src2, vcharx src3);`
	`vintx  vshiftor(vintx  src1, int src2, vintx  src3);`
	`vshortx vshiftor(vshortx src1, int src2, vshortx src3);`
	`vcharx vshiftor(vcharx src1, int src2, vcharx src3);`
	`// double vector pseudo intrinsics`
	`dvintx  dvshiftor(dvintx  src1, dvintx  src2, dvintx  src3);`
	`dvshortx dvshiftor(dvshortx src1, dvshortx src2, dvshortx src3);`
	`dvcharx dvshiftor(dvcharx src1, dvcharx src2, dvcharx src3);`
	`dvintx  dvshiftor(dvintx  src1, int src2, dvintx  src3);`
	`dvshortx dvshiftor(dvshortx src1, int src2, dvshortx src3);`
	`dvcharx dvshiftor(dvcharx src1, int src2, dvcharx src3);`

Additional details	For each lane, dst1 = shift(src1, src2) | src3. Shift left or right based on sign of src2. src2 is read as a signed number and saturated at [-12, 12], [-24, 24], [-48, 48], before detecting sign and applying the shift. Positive bit count shifts left, and negative bit count shifts right.

9.8.6.8 VSHIFTADD

Instruction name	VSHIFTADD
Functionality	Vector shift-add
Assembly format	VShiftAdd<type> Vsrc1, Vsrc2/Rsrc2, Vsrc3, Vdst1
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit sign-extended to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes, as signed.
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF or scalar register src3: single vector register in VRF
Destination options	dst1: single vector register in VRF
Additional options	

Intrinsics/operator	`vintx  vshiftadd(vintx  src1, vintx  src2, vintx  src3);`
	`vshortx vshiftadd(vshortx src1, vshortx src2, vshortx src3);`
	`vcharx vshiftadd(vcharx src1, vcharx src2, vcharx src3);`
	`vintx  vshiftadd(vintx  src1, int src2, vintx  src3);`
	`vshortx vshiftadd(vshortx src1, int src2, vshortx src3);`
	`vcharx vshiftadd(vcharx src1, int src2, vcharx src3);`
	`// double vector pseudo intrinsics`
	`dvintx  dvshiftadd(dvintx  src1, dvintx  src2, dvintx  src3);`
	`dvshortx dvshiftadd(dvshortx src1, dvshortx src2, dvshortx src3);`
	`dvcharx dvshiftadd(dvcharx src1, dvcharx src2, dvcharx src3);`
	`dvintx  dvshiftadd(dvintx  src1, int src2, dvintx  src3);`
	`dvshortx dvshiftadd(dvshortx src1, int src2, dvshortx src3);`
	`dvcharx dvshiftadd(dvcharx src1, int src2, dvcharx src3);`

Additional details	For each lane, dst1 = shift(src1, src2) + src3. Shift left or right based on sign of src2. src2 is read as a signed number and saturated at [-12, 12], [-24, 24], [-48, 48], before detecting sign and applying the shift. Positive bit count shifts left, and negative bit count shifts right.

9.8.6.9 VEXTRBITS

Instruction name	VEXTRBITS
Functionality	Vector extract bits
Assembly format	VExtrBits<type> Vsrc1, Vsrc2, Vsrc3/Rsrc3, Vdst
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed Scalar operand: W: full 32-bit **zero-extended** to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes.
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF src3: single vector register in VRF or scalar register
Destination options	dst: Single vector register in VRF
Additional options	

Intrinsics/operator	`vintx  vextract_bits(vintx  src1, vintx  src2, vintx  src3);`
	`vshortx vextract_bits(vshortx src1, vshortx src2, vshortx src3);`
	`vcharx vextract_bits(vcharx src1, vcharx src2, vcharx src3);`
	`vintx  vextract_bits(vintx  src1, vintx  src2, int src3);`
	`vshortx vextract_bits(vshortx src1, vshortx src2, int src3);`
	`vcharx vextract_bits(vcharx src1, vcharx src2, int src3);`
	`// double vector pseudo intrinsics`
	`dvintx  dvextract_bits(dvintx  src1, dvintx  src2, dvintx  src3);`
	`dvshortx dvextract_bits(dvshortx src1, dvshortx src2, dvshortx src3);`
	`dvcharx dvextract_bits(dvcharx src1, dvcharx src2, dvcharx src3);`
	`dvintx  dvextract_bits(dvintx  src1, dvintx  src2, int src3);`
	`dvshortx dvextract_bits(dvshortx src1, dvshortx src2, int src3);`
	`dvcharx dvextract_bits(dvcharx src1, dvcharx src2, int src3);`

Additional details	Shift input then AND with bitmask between low and high bit positions.
	
	`low = src3[7:0];  // Unsigned bit position`
	`high = src3[15:8] // Unsigned bit position, Rsrc3 or Vsrc3`
	`          // in H/W types`
	`high = src3[11:8]; // Unsigned bit position, Vsrc3 in B type`
	`temp1 = shift(src1, src2); // up or down based on src2 sign`
	`temp2 = ~((1 << low)-1);`
	`temp3 = (1 << high+1) – 1;`
	`dst = temp1 & temp2 & temp3`
	
	If low > high or if low >= BITWIDTH, 0 is returned. Otherwise, high is saturated to top bit position of the lane.
	For example, with byte lane input src1 = 0x12, src2 = 4, low = 4, high = 7,
	temp1 = shift(0x12, 4) = 0x120
	temp2 = 0xFF0 (enable bits 4 and higher)
	temp3 = 0x0FF (enable bits 7 and lower)
	return 0x120 & 0xFF0 & 0x0FF = 0x20

9.8.6.10 VATAN2PP

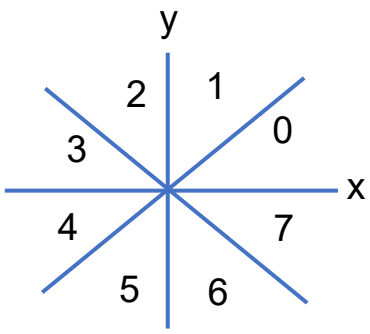
Instruction name	VATAN2PP
Functionality	Vector atan2 post-processing
Assembly format	VAtan2PP<type> Vsrc1, Vsrc2, Vsrc3, Vdst1
Type and bit width	H: 16 x 24-bit
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF src3: single vector register in VRF
Destination options	dst1: single vector register in VRF
Additional options	

Intrinsics/operator	`vshortx vatan2_postp(vshortx src1, vshortx src2, vshortx src3);`
	`// double vector pseudo intrinsics`
	`dvshortx dvatan2_postp(dvshortx src1, dvshortx src2, dvshortx src3);`

Additional details	Treat Vsrc1 as Y, Vsrc2 as X, detect octant of (X, Y) in 2D plane, 0 ~ 7 (see 9.8.4.25 VOctDetH). Treat Vsrc3 as first-octant outcome of atan, A, and return: Condition Octant and ang range Return angle X>=0, Y>=0, |Y|<=|X| 0: [0 ~ 0.25 pi] A & 0x7FFF X>=0, Y>=0, |Y|>|X| 1: (0.25 pi ~ 0.5 pi) (0x2000 – A) & 0x7FFF X<0, Y>=0, |Y|>|X| 2: (0.5 pi ~ 0.75 pi) (0x2000 + A) & 0x7FFF X<0, Y>=0, |Y|<=|X| 3: [0.75 pi ~ pi] (0x4000 – A) & 0x7FFF X<0, Y<0, |Y|<=|X| 4: (pi ~ 1.25 pi) (0x4000 + A) & 0x7FFF X<0, Y<0, |Y|>|X| 5: (1.25 pi ~ 1.5 pi) (0x6000 – A) & 0x7FFF X>=0,Y<0, |Y|>|X| 6: [1.5 pi ~ 1.75 pi] (0x6000 + A) & 0x7FFF X>=0,Y<0, |Y|<=|X| 7: [1.75 pi ~ 2 pi] (0x8000 – A) & 0x7FFF For example, in a particular lane, say we have src1 = X = 100, src2 = Y = -200, src3 = A = 0x972. It's in the 6th octant, as X is positive, Y is negative, and |Y| > |X|. Return value is 0x6000 + 0x972 = 0x6972.

The atan2(y, x) function is implemented with table lookup. In order to compress the table, we take the absolute value of y, x, and sort (|y|, |x|) so that |y| <= |x|. This folds the whole 2*pi range of output to 1/8 of the range, 0 ~ pi/4.

After doing lookup and post-lookup interpolation with the sorted (|y|, |x|), we use the VAtan2PPH with the first-octant angle and (y, x) as inputs to map the angle back to the full range, as shown in the following diagram:



oct	tan	ang
0	y/x	a
	x/y	1/4 – a
	-x/y	1/4 + a
	-y/x	1/2 – a
	y/x	1/2 + a
	x/y	3/4 – a
	-x/y	3/4 + a
	-y/x	1 – a

Note that the 2*pi full range is quantized to 15-bit, 0 ~ 0x7FFF. Thus, 90-degree is 0x2000, 180-degree 0x4000, and 270-degree 0x6000.

9.8.6.11 VMIN3

Instruction name	**VMIN3**
Functionality	Vector min3
Assembly format	VMin3<type> Vsrc1, Vsrc2, Vsrc3, Vdst1 VMin3<type> Vsrc1, Vsrc2, Wsrc3, Vdst VMin3<type> Vsrc1, Wsrc2, Wsrc3, Vdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF or WRF src3: single vector register in VRF or WRF
Destination options	dst1: single vector register in VRF
Additional options	

Intrinsics/operator	`vintx vmin3(vintx src1, vintx src2, vintx src3);`
	`vshortx vmin3(vshortx src1, vshortx src2, vshortx src3);`
	`vcharx vmin3(vcharx src1, vcharx src2, vcharx src3);`
	`// double vector pseudo intrinsics`
	`dvintx dvmin3(dvintx src1, dvintx src2, dvintx src3);`
	`dvshortx dvmin3(dvshortx src1, dvshortx src2, dvshortx src3);`
	`dvcharx dvmin3(dvcharx src1, dvcharx src2, dvcharx src3);`

Additional details	For each lane, return minimal of 3 sources.

9.8.6.12 VMIN3_CA

Instruction name	VMIN3_CA
Functionality	Vector min3
Assembly format	<pred> VMin3<type>_CA Vsrc1, Vsrc2, ACsrc3dst pred = none, [P2..P15] [P0] is omitted
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit
Predication	Available across lanes to clear accumulator
Source options	src1: single vector register in VRF src2: single vector register in VRF
Destination options	src3dst: single vector register in ARF
Additional options	

Intrinsics/operator	`vintx vmin3_ca(vintx src1, vintx src2, vintx src3, int pred);`
	`vshortx vmin3_ca(vshortx src1, vshortx src2, vshortx src3, int pred);`
	`vcharx vmin3_ca(vcharx src1, vcharx src2, vcharx src3, int pred);`
	`// double vector pseudo intrinsics`
	`dvintx dvmin3_ca(dvintx src1, dvintx src2, dvintx src3, int pred);`
	`dvshortx dvmin3_ca(dvshortx src1, dvshortx src2, dvshortx src3,int pred);`
	`dvcharx dvmin3_ca(dvcharx src1, dvcharx src2, dvcharx src3, int pred);`

Additional details	`Vsrc3dst = pred ? min(Vsrc1, Vsrc2, Vsrc3dst)`
	`: min(Vsrc1, Vsrc2);`
	When predicate is off, the operation becomes min of first 2 sources, allowing min accumulation to start fresh.

9.8.6.13 VMAX3

Instruction name	VMAX3
Functionality	Vector max3
Assembly format	VMax3<type> Vsrc1, Vsrc2, Vsrc3, Vdst1 VMax3<type> Vsrc1, Vsrc2, Wsrc3, Vdst VMax3<type> Vsrc1, Wsrc2, Wsrc3, Vdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF or WRF src3: single vector register in VRF or WRF
Destination options	dst1: single vector register
Additional options	
Intrinsics/operator	`vintx vmax3(vintx src1, vintx src2, vintx src3);` `vshortx vmax3(vshortx src1, vshortx src2, vshortx src3);`

Instruction name	VMAX3

	vcharx vmax3(vcharx src1, vcharx src2, vcharx src3);
	// double vector pseudo intrinsics
	dvintx dvmax3(dvintx src1, dvintx src2, dvintx src3);
	dvshortx dvmax3(dvshortx src1, dvshortx src2, dvshortx src3);
	dvcharx dvmax3(dvcharx src1, dvcharx src2, dvcharx src3);

Additional details	For each lane, return maximal of 3 sources.

9.8.6.14 VMAX3_CA

Instruction name	VMAX3_CA
Functionality	Vector max3
Assembly format	<pred> VMax3<type>_CA Vsrc1, Vsrc2, ACsrc3dst pred = none, [P2.. P15] [P0] is omitted
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit
Predication	Available across lanes to clear accumulator
Source options	src1: single vector register in VRF src2: single vector register in VRF
Destination options	src3dst: single vector register in ARF
Additional options	

Intrinsics/operator	vintx vmax3_ca(vintx src1, vintx src2, vintx src3, int pred);
	vshortx vmax3_ca(vshortx src1, vshortx src2, vshortx src3, int pred);
	vcharx vmax3_ca(vcharx src1, vcharx src2, vcharx src3, int pred);
	// double vector pseudo intrinsics
	dvintx dvmax3_ca(dvintx src1, dvintx src2, dvintx src3, int pred);
	dvshortx dvmax3_ca(dvshortx src1, dvshortx src2, dvshortx src3,int pred);
	dvcharx dvmax3_ca(dvcharx src1, dvcharx src2, dvcharx src3, int pred);

Additional details	Vsrc3dst = preg ? max(Vsrc1, Vsrc2, Vsrc3dst) : max(Vsrc1, Vsrc2); When predicate is off, the operation becomes max of first 2 sources, allowing max accumulation to start fresh.

9.8.6.15 VADD3

Instruction name	VADD3
Functionality	Vector add3
Assembly format	VAdd3<type> Vsrc1, Vsrc2, Vsrc3, Vdst1
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit
Predication	not available
Source options	src1: single vector register in VRF

	src2: single vector register in VRF src3: single vector register in VRF
Destination options	dst1: single vector register in VRF
Additional options	

Intrinsics/operator	`vintx  vadd3(vintx  src1, vintx  src2, vintx  src3);`
	`vshortx vadd3(vshortx src1, vshortx src2, vshortx src3);`
	`vcharx vadd3(vcharx src1, vcharx src2, vcharx src3);`
	`// double vector pseudo intrinsics`
	`dvintx  dvadd3(dvintx  src1, dvintx  src2, dvintx  src3);`
	`dvshortx dvadd3(dvshortx src1, dvshortx src2, dvshortx src3);`
	`dvcharx dvadd3(dvcharx src1, dvcharx src2, dvcharx src3);`

Additional details	

9.8.6.16 VADD3_CA

Instruction name	**VADD3_CA**
Functionality	Vector add3
Assembly format	<pred> VAdd3B/H/W_CA Vsrc1, Vsrc2, ACsrc3dst pred = none, [P2.. P15] [P0] is omitted
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit
Predication	Available across lanes to clear accumulator
Source options	src1: single vector register in VRF src2: single vector register in VRF
Destination options	src3dst: B/H/W: single vector register in ARF
Additional options	

Intrinsics/operator	`vintx  vadd3_ca(vintx  src1, vintx  src2, vintx  src3, int pred);`
	`vshortx vadd3_ca(vshortx src1, vshortx src2, vshortx src3, int pred);`
	`vcharx vadd3_ca(vcharx src1, vcharx src2, vcharx src3, int pred);`
	`// double vector pseudo intrinsics`
	`dvintx  dvadd3_ca(dvintx  src1, dvintx  src2, dvintx  src3, int pred);`
	`dvshortx dvadd3_ca(dvshortx src1, dvshortx src2, dvshortx src3, int pred);`
	`dvcharx dvadd3_ca(dvcharx src1, dvcharx src2, dvcharx src3, int pred);`

Additional details	`Vsrc3dst = preg ? (Vsrc1 + Vsrc2 + Vsrc3dst)`
	`: (Vsrc1 + Vsrc2);`
	When predicate is off, the operation becomes sum of first 2 sources, allowing accumulation to start fresh.

9.8.6.17 VAND3

Instruction name	VAND3
Functionality	Vector and 3
Assembly format	VAnd3 Vsrc1, Vsrc2, Vsrc3, Vdst1
Type and bit width	no type, full 384 bits
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF src3: single vector register in VRF
Destination options	dst1: single vector register in VRF
Additional options	

Intrinsics/operator	`vintx vand3(vintx src1, vintx src2, vintx src3);`
	`vshortx vand3(vshortx src1, vshortx src2, vshortx src3);`
	`vcharx vand3(vcharx src1, vcharx src2, vcharx src3);`
	`// double vector pseudo intrinsics`
	`dvintx dvand3(dvintx src1, dvintx src2, dvintx src3);`
	`dvshortx dvand3(dvshortx src1, dvshortx src2, dvshortx src3);`
	`dvcharx dvand3(dvcharx src1, dvcharx src2, dvcharx src3);`

Additional details	Bitwise AND in each vector lane

9.8.6.18 VAND3_CA

Instruction name	VAND3_CA
Functionality	Vector and 3
Assembly format	<pred> VAnd3_CA Vsrc1, Vsrc2, Vsrc3dst pred = none, [P2.. P15] [P0] is omitted
Type and bit width	no type, full 384 bits
Predication	Available across lanes to clear accumulator
Source options	src1: single vector register in VRF src2: single vector register in VRF
Destination options	src3dst: single vector register in VRF
Additional options	

Intrinsics/operator	`vintx vand3_ca(vintx src1, vintx src2, vintx src3, int pred);`
	`vshortx vand3_ca(vshortx src1, vshortx src2, vshortx src3, int pred);`
	`vcharx vand3_ca(vcharx src1, vcharx src2, vcharx src3, int pred);`
	`// double vector pseudo intrinsics`
	`dvintx dvand3_ca(dvintx src1, dvintx src2, dvintx src3, int pred);`
	`dvshortx dvand3_ca(dvshortx src1, dvshortx src2, dvshortx src3, int pred);`
	`dvcharx dvand3_ca(dvcharx src1, dvcharx src2, dvcharx src3, int pred);`

Instruction name	VAND3_CA

Additional details	Bitwise AND in each vector lane
	`Vsrc3dst = preg ? (Vsrc1 & Vsrc2 & Vsrc3dst)`
	`: (Vsrc1 & Vsrc2);`
	When predicate is off, the operation becomes AND of first 2 sources, allowing AND accumulation to start fresh.

9.8.6.19 VOR3

Instruction name	VOR3
Functionality	Vector or 3
Assembly format	VOr3 Vsrc1, Vsrc2, Vsrc3, Vdst1
Type and bit width	no type, full 384 bits
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF src3: single vector register in VRF
Destination options	dst1: single vector register in VRF
Additional options	

Intrinsics/operator	`vintx  vor3(vintx  src1, vintx  src2, vintx  src3);`
	`vshortx vor3(vshortx src1, vshortx src2, vshortx src3);`
	`vcharx vor3(vcharx src1, vcharx src2, vcharx src3);`
	`// double vector pseudo intrinsics`
	`dvintx  dvor3(dvintx  src1, dvintx  src2, dvintx  src3);`
	`dvshortx dvor3(dvshortx src1, dvshortx src2, dvshortx src3);`
	`dvcharx dvor3(dvcharx src1, dvcharx src2, dvcharx src3);`

Additional details	Bitwise OR in each vector lane

9.8.6.20 VOR3_CA

Instruction name	VOR3_CA
Functionality	Vector or 3
Assembly format	<pred> VOr3_CA Vsrc1, Vsrc2, Vsrc3dst pred = none, [P2.. P15] [P0] is omitted
Type and bit width	no type, full 384 bits
Predication	Available across lanes to clear accumulator
Source options	src1: single vector register in VRF src2: single vector register in VRF
Destination options	src3dst: single vector register in VRF

Instruction name	VOR3_CA
Additional options	

Intrinsics/operator	`vintx  vor3_ca(vintx  src1, vintx  src2, vintx  src3, int pred);`
	`vshortx vor3_ca(vshortx src1, vshortx src2, vshortx src3, int pred);`
	`vcharx vor3_ca(vcharx src1, vcharx src2, vcharx src3, int pred);`
	`// double vector pseudo intrinsics`
	`dvintx  dvor3_ca(dvintx  src1, dvintx  src2, dvintx  src3, int pred);`
	`dvshortx dvor3_ca(dvshortx src1, dvshortx src2, dvshortx src3, int pred);`
	`dvcharx dvor3_ca(dvcharx src1, dvcharx src2, dvcharx src3, int pred);`

Additional details	Bitwise OR in each vector lane
	`Vsrc3dst = preg ? (Vsrc1 | Vsrc2 | Vsrc3dst)`
	`  : (Vsrc1 | Vsrc2);`
	When predicate is off, the operation becomes OR of first 2 sources, allowing OR accumulation to start fresh.

9.8.6.21 VXOR3

Instruction name	VXOR3
Functionality	Vector exclusive-or 3
Assembly format	VXor3 Vsrc1, Vsrc2, Vsrc3, Vdst1
Type and bit width	no type, full 384 bits
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF src3: single vector register in VRF
Destination options	dst1: single vector register in VRF
Additional options	

Intrinsics/operator	`vintx  vxor3(vintx  src1, vintx  src2, vintx  src3);`
	`vshortx vxor3(vshortx src1, vshortx src2, vshortx src3);`
	`vcharx vxor3(vcharx src1, vcharx src2, vcharx src3);`
	`// double vector pseudo intrinsics`
	`dvintx  dvxor3(dvintx  src1, dvintx  src2, dvintx  src3);`
	`dvshortx dvxor3(dvshortx src1, dvshortx src2, dvshortx src3);`
	`dvcharx dvxor3(dvcharx src1, dvcharx src2, dvcharx src3);`

Additional details	Bitwise exclusive OR in each vector lane

9.8.6.22 VXOR3_CA

Instruction name	VXOR3_CA
Functionality	Vector exclusive-or 3
Assembly format	<pred> VXor3_CA Vsrc1, Vsrc2, Vsrc3dst pred = none, [P2.. P15] [P0] is omitted
Type and bit width	no type, full 384 bits
Predication	Available across lanes to clear accumulator
Source options	src1: single vector register in VRF src2: single vector register in VRF
Destination options	src3dst: single vector register in VRF
Additional options	

Intrinsics/operator	`vintx  vxor3_ca(vintx  src1, vintx  src2, vintx  src3, int pred);`
	`vshortx vxor3_ca(vshortx src1, vshortx src2, vshortx src3, int pred);`
	`vcharx vxor3_ca(vcharx src1, vcharx src2, vcharx src3, int pred);`
	`// double vector pseudo intrinsics`
	`dvintx  dvxor3_ca(dvintx  src1, dvintx  src2, dvintx  src3, int pred);`
	`dvshortx dvxor3_ca(dvshortx src1, dvshortx src2, dvshortx src3, int pred);`
	`dvcharx dvxor3_ca(dvcharx src1, dvcharx src2, dvcharx src3, int pred);`

Additional details	Bitwise exclusive-OR in each vector lane
	`Vsrc3dst = preg ? (Vsrc1 ^ Vsrc2 ^ Vsrc3dst)`
	`: (Vsrc1 ^ Vsrc2);`
	When predicate is off, the operation becomes XOR of first 2 sources, allowing XOR accumulation to start fresh.

9.8.6.23 VSAD_CA

Instruction name	VSAD_CA
Functionality	Vector sum of absolute differences
Assembly format	<pred> VSad<type>_CA Vsrc1, Vsrc2, ACsrc3dst/DACsrc3dst pred = none, [P2.. P15] [P0] is omitted
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit HW: 16 x (|24-bit – 24-bit| + 48-bit) BH: 32 x (|12-bit – 12-bit| + 24-bit)
Predication	Available across lanes to clear accumulator
Source options	src1: single vector register in VRF src2: single vector register in VRF
Destination options	src3dst: single vector register for W in ARF

Instruction name	VSAD_CA
	src3dst: double vector register for HW, BH in ARF
Additional options	

Intrinsics/operator	`vintx  vSAD_ca(vintx  src1, vintx  src2, vintx  src3, int pred);`
	`vshortx vSAD_ca(vshortx src1, vshortx src2, vshortx src3, int pred);`
	`vcharx vSAD_ca(vcharx src1, vcharx src2, vcharx src3, int pred);`
	`dvintx vSAD_ca(vshortx src1, vshortx src2, dvintx src3, int pred);`
	`dvshortx vSAD_ca(vcharx src1, vcharx src2, dvshortx src3, int pred);`
	`// double vector pseudo intrinsics`
	`dvintx  dvSAD_ca(dvintx  src1, dvintx  src2, dvintx  src3, int pred);`
	`dvshortx dvSAD_ca(dvshortx src1, dvshortx src2, dvshortx src3, int pred);`
	`dvcharx dvSAD_ca(dvcharx src1, dvcharx src2, dvcharx src3, int pred);`

Additional details	For each lane, src3dst += |src1 - src2| when predicate is on. Otherwise, src3dst = |src1 - src2|. For HW and BH types, destination is a double vector register. Lane 2*i from |src1 - src2| is added/stored to lane i of the lower register of the pair. Lane 2*i+1 from |src1 – src2| is added/stored to lane i of the upper register.

9.8.6.24 VSUMHD_CA

Instruction name	VSumHD_CA
Functionality	Vector sum of Hamming distance
Assembly format	<pred> VSumHD<type>_CA Vsrc1, Vsrc2/Rsrc2, ACsrc3dst pred = none, [P2.. P15] [P0] is omitted
Type and bit width	Vector operand: W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, **unsigned** Scalar operand: W: full 32-bit **zero-extended** to 48-bit, H: 24 LSBs, B: 12 LSBs broadcast to all vector lanes, as unsigned.
Predication	Available across lanes to clear accumulator
Source options	src1: single vector register in VRF src2: single vector register in VRF or scalar register
Destination options	src3dst: single vector register in ARF
Additional options	

Intrinsics/operator	`vintx  vSumHD_ca(vintx  src1, vintx  src2, vintx  src3, int pred);`
	`vshortx vSumHD_ca(vshortx src1, vshortx src2, vshortx src3, int pred);`
	`vcharx vSumHD_ca(vcharx src1, vcharx src2, vcharx src3, int pred);`
	`vintx  vSumHD_ca(vintx  src1, unsigned int src2, vintx  src3, int pred);`
	`vshortx vSumHD_ca(vshortx src1, unsigned int src2, vshortx src3, int pred);`
	`vcharx vSumHD_ca(vcharx src1, unsigned int src2, vcharx src3, int pred);`
	
	`// double vector pseudo intrinsics`
	`dvintx  dvSumHD_ca(dvintx  src1, dvintx  src2, dvintx  src3, int pred);`
	`dvshortx dvSumHD_ca(dvshortx src1, dvshortx src2, dvshortx src3, int pred);`

Instruction name	VSumHD_CA

	dvcharx dvSumHD_ca(dvcharx src1, dvcharx src2, dvcharx src3, int pred);
	dvintx dvSumHD_ca(dvintx src1, unsigned int src2, dvintx src3, int pred);
	dvshortx dvSumHD_ca(dvshortx src1, unsigned int src2, dvshortx src3, int pred);
	dvcharx dvSumHD_ca(dvcharx src1, unsigned int src2, dvcharx src3, int pred);

Additional details	For each lane, src3dst += bit_count(src1 ^ src2) when predicate is on, otherwise, src3dst = bit_count(src1 ^ src2). "^" is the bit-wise XOR operation.

9.8.6.25 VCMPLT_ANDL

Instruction name	VCMPLT_ANDL
Functionality	Vector compare less than and logical AND
Assembly format	VCmpLT_AndL <type> Vsrc1, Vsrc2, Vsrc3, Vdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	src1, src2, src3: single vector register in VRF
Destination options	dst: Single vector register in VRF
Additional options	

Intrinsics/operator	vcharx vCmpLT_andL(vcharx src1, vcharx src2, vcharx src3);
	vshortx vCmpLT_andL(vshortx src1, vshortx src2, vshortx src3);
	vintx vCmpLT_andL(vintx src1, vintx src2, vintx src3);
	
	// compiler also instantiates from, e.g.,
	// vintx dst = (vintx src1 < vintx src2) && src3;
	// vshortx dst = (vshortx src1 < vshortx src2) && src3;
	// vcharx dst = (vcharx src1 < vcharx src2) && src3;
	
	// double vector pseudo intrinsics
	dvcharx dvCmpLT_andL(dvcharx src1, dvcharx src2, dvcharx src3);
	dvshortx dvCmpLT_andL(dvshortx src1, dvshortx src2, dvshortx src3);
	dvintx dvCmpLT_andL(dvintx src1, dvintx src2, dvintx src3);

Additional details	

9.8.6.26 VCMPLE_ANDL

Instruction name	VCMPLE_ANDL
Functionality	Vector compare less than or equal and logical AND
Assembly format	VCmpLE_AndL <type> Vsrc1, Vsrc2, Vsrc3, Vdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	src1, src2, src3: single vector register in VRF
Destination options	dst: Single vector register in VRF
Additional options	

Intrinsics/operator	`vcharx vCmpLE_andL(vcharx src1, vcharx src2, vcharx src3);`
	`vshortx vCmpLE_andL(vshortx src1, vshortx src2, vshortx src3);`
	`vintx vCmpLE_andL(vintx src1, vintx src2, vintx src3);`
	
	`// compiler also instantiates from, e.g.,`
	`// vintx  dst = (vintx  src1 <= vintx  src2) && src3;`
	`// vshortx dst = (vshortx src1 <= vshortx src2) && src3;`
	`// vcharx dst = (vcharx src1 <= vcharx src2) && src3;`
	
	`// double vector pseudo intrinsics`
	`dvcharx dvCmpLE_andL(dvcharx src1, dvcharx src2, dvcharx src3);`
	`dvshortx dvCmpLE_andL(dvshortx src1, dvshortx src2, dvshortx src3);`
	`dvintx dvCmpLE_andL(dvintx src1, dvintx src2, dvintx src3);`

Additional details	

9.8.6.27 VCMPEQ_ANDL

Instruction name	VCMPEQ_ANDL
Functionality	Vector compare equal and logical AND
Assembly format	VCmpEQ_AndL <type> Vsrc1, Vsrc2, Vsrc3, Vdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	src1, src2, src3: single vector register in VRF
Destination options	dst: Single vector register in VRF
Additional options	

Intrinsics/operator	`vcharx vCmpEQ_andL(vcharx src1, vcharx src2, vcharx src3);`
	`vshortx vCmpEQ_andL(vshortx src1, vshortx src2, vshortx src3);`
	`vintx vCmpEQ_andL(vintx src1, vintx src2, vintx src3);`
	
	`// compiler also instantiates from, e.g.,`
	`// vintx  dst = (vintx  src1 == vintx  src2) && src3;`
	`// vshortx dst = (vshortx src1 == vshortx src2) && src3;`
	`// vcharx dst = (vcharx src1 == vcharx src2) && src3;`

Instruction name	VCMPEQ_ANDL

	// double vector pseudo intrinsics
	dvcharx dvCmpEQ_andL(dvcharx src1, dvcharx src2, dvcharx src3);
	dvshortx dvCmpEQ_andL(dvshortx src1, dvshortx src2, dvshortx src3);
	dvintx dvCmpEQ_andL(dvintx src1, dvintx src2, dvintx src3);

Additional details	

9.8.6.28 VCMPNE_ANDL

Instruction name	VCMPNE_ANDL
Functionality	Vector compare not equal and logical AND
Assembly format	VCmpNE_AndL <type> Vsrc1, Vsrc2, Vsrc3, Vdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	src1, src2, src3: single vector register in VRF
Destination options	dst: Single vector register in VRF
Additional options	

Intrinsics/operator	vcharx vCmpNE_andL(vcharx src1, vcharx src2, vcharx src3);
	vshortx vCmpNE_andL(vshortx src1, vshortx src2, vshortx src3);
	vintx vCmpNE_andL(vintx src1, vintx src2, vintx src3);
	
	// compiler also instantiates from, e.g.,
	// vintx dst = (vintx src1 != vintx src2) && src3;
	// vshortx dst = (vshortx src1 != vshortx src2) && src3;
	// vcharx dst = (vcharx src1 != vcharx src2) && src3;
	
	// double vector pseudo intrinsics
	dvcharx dvCmpNE_andL(dvcharx src1, dvcharx src2, dvcharx src3);
	dvshortx dvCmpNE_andL(dvshortx src1, dvshortx src2, dvshortx src3);
	dvintx dvCmpNE_andL(dvintx src1, dvintx src2, dvintx src3);

Additional details	

9.8.6.29 VCMPLT_ORL

Instruction name	VCMPLT_ORL
Functionality	Vector compare less than and logical OR
Assembly format	VCmpLT_OrL <type> Vsrc1, Vsrc2, Vsrc3, Vdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	src1, src2, src3: single vector register in VRF
Destination options	dst: Single vector register in VRF
Additional options	

Instruction name	VCMPLT_ORL

Intrinsics/operator	`vcharx vCmpLT_orL(vcharx src1, vcharx src2, vcharx src3);`
	`vshortx vCmpLT_orL(vshortx src1, vshortx src2, vshortx src3);`
	`vintx vCmpLT_orL(vintx src1, vintx src2, vintx src3);`
	
	`// compiler also instantiates from, e.g.,`
	`// vintx  dst = (vintx  src1 < vintx  src2) || src3;`
	`// vshortx dst = (vshortx src1 < vshortx src2) || src3;`
	`// vcharx dst = (vcharx src1 < vcharx src2) || src3;`
	
	`// double vector pseudo intrinsics`
	`dvcharx dvCmpLT_orL(dvcharx src1, dvcharx src2, dvcharx src3);`
	`dvshortx dvCmpLT_orL(dvshortx src1, dvshortx src2, dvshortx src3);`
	`dvintx dvCmpLT_orL(dvintx src1, dvintx src2, dvintx src3);`

Additional details	

9.8.6.30 VCMPLE_ORL

Instruction name	VCMPLE_ORL
Functionality	Vector compare less than or equal and logical OR
Assembly format	VCmpLE_OrL <type> Vsrc1, Vsrc2, Vsrc3, Vdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	src1, src2, src3: single vector register in VRF
Destination options	dst: Single vector register in VRF
Additional options	

Intrinsics/operator	`vcharx vCmpLE_orL(vcharx src1, vcharx src2, vcharx src3);`
	`vshortx vCmpLE_orL(vshortx src1, vshortx src2, vshortx src3);`
	`vintx vCmpLE_orL(vintx src1, vintx src2, vintx src3);`
	
	`// compiler also instantiates from, e.g.,`
	`// vintx  dst = (vintx  src1 <= vintx  src2) || src3;`
	`// vshortx dst = (vshortx src1 <= vshortx src2) || src3;`
	`// vcharx dst = (vcharx src1 <= vcharx src2) || src3;`
	
	`// double vector pseudo intrinsics`
	`dvcharx dvCmpLE_orL(dvcharx src1, dvcharx src2, dvcharx src3);`
	`dvshortx dvCmpLE_orL(dvshortx src1, dvshortx src2, dvshortx src3);`
	`dvintx dvCmpLE_orL(dvintx src1, dvintx src2, dvintx src3);`

Additional details	

9.8.6.31 VCMPEQ_ORL

Instruction name	VCMPEQ_ORL
Functionality	Vector compare equal and logical OR
Assembly format	VCmpEQ_OrL <type> Vsrc1, Vsrc2, Vsrc3, Vdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	src1, src2, src3: single vector register in VRF
Destination options	dst: Single vector register in VRF
Additional options	

Intrinsics/operator	`vcharx vCmpEQ_orL(vcharx src1, vcharx src2, vcharx src3);`
	`vshortx vCmpEQ_orL(vshortx src1, vshortx src2, vshortx src3);`
	`vintx vCmpEQ_orL(vintx src1, vintx src2, vintx src3);`
	
	`// compiler also instantiates from, e.g.,`
	`// vintx  dst = (vintx  src1 == vintx  src2) || src3;`
	`// vshortx dst = (vshortx src1 == vshortx src2) || src3;`
	`// vcharx dst = (vcharx src1 == vcharx src2) || src3;`
	
	`// double vector pseudo intrinsics`
	`dvcharx dvCmpEQ_orL(dvcharx src1, dvcharx src2, dvcharx src3);`
	`dvshortx dvCmpEQ_orL(dvshortx src1, dvshortx src2, dvshortx src3);`
	`dvintx dvCmpEQ_orL(dvintx src1, dvintx src2, dvintx src3);`

Additional details	

9.8.6.32 VCMPNE_ORL

Instruction name	VCMPNE_ORL
Functionality	Vector compare not equal and logical OR
Assembly format	VCmpNE_OrL <type> Vsrc1, Vsrc2, Vsrc3, Vdst
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit, signed
Predication	not available
Source options	src1, src2, src3: single vector register in VRF
Destination options	dst: Single vector register in VRF
Additional options	

Intrinsics/operator	`vcharx vCmpNE_orL(vcharx src1, vcharx src2, vcharx src3);`
	`vshortx vCmpNE_orL(vshortx src1, vshortx src2, vshortx src3);`
	`vintx vCmpNE_orL(vintx src1, vintx src2, vintx src3);`
	
	`// compiler also instantiates from, e.g.,`
	`// vintx  dst = (vintx  src1 != vintx  src2) || src3;`
	`// vshortx dst = (vshortx src1 != vshortx src2) || src3;`
	`// vcharx dst = (vcharx src1 != vcharx src2) || src3;`

Instruction name	VCMPNE_ORL

	// double vector pseudo intrinsics
	dvcharx dvCmpNE_orL(dvcharx src1, dvcharx src2, dvcharx src3);
	dvshortx dvCmpNE_orL(dvshortx src1, dvshortx src2, dvshortx src3);
	dvintx dvCmpNE_orL(dvintx src1, dvintx src2, dvintx src3);

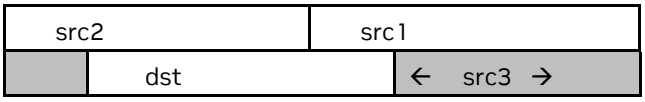
Additional details	

9.8.6.33 VXSHIFTR

Instruction name	VXShiftR
Functionality	Vector cross element shift right
Assembly format	VXShiftR <type> Vsrc1, Vsrc2, Rsrc3/Vsrc3, Vdst
Type and bit width	Vector operand: W: 8 x 32-bit, H: 16 x 16-bit, B: 32 x 8-bit, **unsigned** Scalar operand: W: full 32-bit, H: 16 LSBs, B: 8 LSBs broadcast to all vector lanes as **unsigned**.
Predication	not available
Source options	src1, src2: single vector register in VRF src3: single vector in VRF or scalar register
Destination options	dst: Single vector register in VRF
Additional options	

Intrinsics/operator	vcharx vxshiftr(vcharx src1, vcharx src2, vcharx src3);
	vshortx vxshiftr(vshortx src1, vshortx src2, vshortx src3);
	vintx vxshiftr(vintx src1, vintx src2, vintx src3);
	vcharx vxshiftr(vcharx src1, vcharx src2, unsigned int src3);
	vshortx vxshiftr(vshortx src1, vshortx src2, unsigned int src3);
	vintx vxshiftr(vintx src1, vintx src2, unsigned int src3);
	// double vector pseudo intrinsics
	dvcharx dvxshiftr(dvcharx src1, dvcharx src2, dvcharx src3);
	dvshortx dvxshiftr(dvshortx src1, dvshortx src2, dvshortx src3);
	dvintx dvxshiftr(dvintx src1, dvintx src2, dvintx src3);
	dvcharx dvxshiftr(dvcharx src1, dvcharx src2, unsigned int src3);
	dvshortx dvxshiftr(dvshortx src1, dvshortx src2, unsigned int src3);
	dvintx dvxshiftr(dvintx src1, dvintx src2, unsigned int src3);

Instruction name	VXShiftR

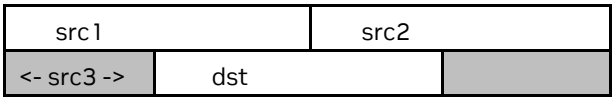
Additional details	Src1 carries current lane data. Src2 carries next lane data (from another load from memory). Src3[7:0] carries number of LSBs of src1 we want to shift right and throw out, and refill upper bits from LSBs of src2.
	Only lower 8/16/32 bits of src1 and src2 are used and treated as an unsigned number. Src3[7:0] is treated as an unsigned number and saturated to 8/16/32 before being used in the subsequent operations.
	In each lane we compute
	nbits = src3[7:0];
	nbits = (nbits > bitwidth) ? bitwidth : nbits;
	dst = ((src1 >> nbits) | (src2 << (bitwidth – nbits))) & mask;
	where bitwidth = 8/16/32 for B/H/W type, and mask = (1 << bitwidth) – 1.
	MSB LSB MSB LSB
	 

9.8.6.34 VXSHIFTL

Instruction name	VXShiftL
Functionality	Vector cross element shift left
Assembly format	VXShiftL <type> Vsrc1, Vsrc2, Rsrc3/Vsrc3, Vdst
Type and bit width	Vector operand: W: 8 x 32-bit, H: 16 x 16-bit, B: 32 x 8-bit, **unsigned** Scalar operand: W: full 32-bit, H: 16 LSBs, B: 8 LSBs broadcast to all vector lanes as **unsigned**.
Predication	not available
Source options	src1, src2: single vector register in VRF src3: single vector in VRF or scalar register
Destination options	dst: Single vector register in VRF
Additional options	

Intrinsics/operator	`vcharx vxshiftl(vcharx src1, vcharx src2, vcharx src3);`
	`vshortx vxshiftl(vshortx src1, vshortx src2, vshortx src3);`
	`vintx vxshiftl(vintx src1, vintx src2, vintx src3);`
	`vcharx vxshiftl(vcharx src1, vcharx src2, unsigned int src3);`
	`vshortx vxshiftl(vshortx src1, vshortx src2, unsigned int src3);`
	`vintx vxshiftl(vintx src1, vintx src2, unsigned int src3);`
	`// double vector pseudo intrinsics`
	`dvcharx dvxshiftl(dvcharx src1, dvcharx src2, dvcharx src3);`
	`dvshortx dvxshiftl(dvshortx src1, dvshortx src2, dvshortx src3);`
	`dvintx dvxshiftl(dvintx src1, dvintx src2, dvintx src3);`
	`dvcharx dvxshiftl(dvcharx src1, dvcharx src2, unsigned int src3);`
	`dvshortx dvxshiftl(dvshortx src1, dvshortx src2, unsigned int src3);`

Instruction name	VXShiftL
	`dvintx dvxshiftl(dvintx src1, dvintx src2, unsigned int src3);`

Additional details	Src1 carries current lane data. Src2 carries previous lane data (from another load from memory). Src3[7:0] carries number of MSBs of src1 we want to shift left and throw out, and refill lower bits from MSBs of src2.
	Only lower 8/16/32 bits of src1 and src2 are used and treated as an unsigned number. Src3[7:0] is treated as an unsigned number and saturated to 8/16/32 before being used in the subsequent operations.
	In each lane we compute
	nbits = src3[7:0];
	nbits = (nbits > bitwidth) ? bitwidth : nbits;
	dst = ((src1 << nbits) | (src2 >> (bitwidth – nbits))) & mask;
	where bitwidth = 8/16/32 for B/H/W type, and mask = (1 << bitwidth) – 1.
	MSB LSB MSB LSB
	 

9.8.7 Vector Multiply-Add Instructions

9.8.7.1 Types and Data Widths

Multiplication has higher area cost per bit, so instead of extended precision of B=12-bit, H=24-bit, W=48-bit, VPU supports B=9-bit, H=17-bit, W=33-bit of multiplication input. The 1 extra bits compared with standard bit width allows support of both signed and unsigned data of standard bit widths.

For src1 and src2, the B/H/W types correspond to 9/17/33 bits, as opposed to 12/24/48 bits for most other vector ALU operations.

For multiply-add/subtract and various dot-product/filtering operations, src3 is the operand to be added/subtracted from, and extended bit width of 12/24/48 bits of src3 are used. 12/24/48-bit results are calculated and written to the destination.

There is optional rounding/truncation after multiplication (and before add or subtract for VMAdd, VMSub). Rounding is not by arbitrary bit counts but with a few selected options.

There are two encoding schemes:

For Word type (33-bit by 33-bit) multiply/multiply-add/multiply-subtract resulting in 48-bit outcome, VPU supports:

- .R0/.T0/omitted: no rounding (encoded as 0)
- .T4: rounding down 4 bits (encoded as 5)
- .T8: truncate down 7 bits (encoded as 6)
- .T16: truncate down 16 bits (encoded as 7)

There is an instruction (VMulWWL) that carries out 33-bit x 33-bit multiply without rounding/truncation and produces 66-bit outcome in two register entries.

For all other type combinations (B, BBH, H, HHW, WHW):

- .R0/.T0/omitted: no rounding (encoded as 0)
- .R7: rounding down 7 bits (encoded as 1)
- .R15: rounding down 15 bits (encoded as 2)
- .R16: rounding down 16 bits (encoded as 3)
- .R4: rounding down 4 bits (encoded as 4)
- .T7: truncate down 7 bits (encoded as 5)
- .T15: truncate down 15 bits (encoded as 6)
- .T16: truncate down 16 bits (encoded as 7)

Note that in Gen-1 VPU MAC instructions, there is just one set of encoding for non-Word type:

- .R0/omitted: no rounding (encoded as 0)
- .R7: rounding down 7 bits (encoded as 1)
- .R15: rounding down 15 bits (encoded as 2)
- .R16: rounding down 16 bits (encoded as 3)
- .T0: truncating 0 bits, same as no rounding (encoded as 4)
- .T7: truncate down 7 bits (encoded as 5)
- .T15: truncate down 15 bits (encoded as 6)
- .T16: truncate down 16 bits (encoded as 7)

and for Word-type MAC, the only available rounding option is .T16, except for VMulWWL.

Note that not all options are supported in all VMul/VMAdd/VMSub instructions for all types. Please see individual MAC instruction for details.

VPU supports the following type combinations of multiply/multiply-add/multiply-subtract:

- B: 9-bit by 9-bit multiplication with 12-bit accumulator or product
- BBH: 9-bit by 9-bit multiplication with 24-bit accumulator or product
- BBW: 9-bit by 9-bit multiplication with 32-bit accumulator
- H: 17-bit by 17-bit multiplication with 24-bit accumulator or product
- HHW: 17-bit by 17-bit multiplication with 48-bit accumulator or product
- WHW: 33-bit by 17-bit multiplication with 48-bit accumulator or product, src2 is in half-word lanes, but only even-numbered lanes are used, lining up with W lanes
- W: 33-bit by 33-bit multiplication with truncation of 0/4/8/16 bits, keeping 48 LSBs of product.
- WWL: 33-bit by 33-bit multiplication with 66-bit product

B, H, WHW, W are lane-parallel operations, in the sense that we have src1[i] * src2[i] → dst[i]. The others, BBH, BBW, HHW, WWH, WWL, are lane-expansion operations, with

single vector as src1, single vector as src2, but double vector as src3/destination to receive the 2x wider output. In lane-expansion operations, outcome elements are **deinterleaved** to the two src3/destination registers to avoid long routes.

A few expanding operations (VDotP4x2BBH/HHW, VFilt4x2BBH/HHW, VFilt4x2x2BBH/BBW) produce two sets of products and accumulate into two sets of accumulators, requiring a quad vector for the accumulator operand.

For example, VMAddBBH.R15 V0, V1, V2:V3 would result in:

```
V2[0] += round(V0[0] * V1[0], 15);
V3[0] += round(V0[1] * V1[1], 15);
V2[1] += round(V0[2] * V1[2], 15);
V3[1] += round(V0[3] * V1[3], 15);
…
V2[15] += round(V0[30] * V1[30], 15);
V3[15] += round(V0[31] * V1[31], 15);
```

VMulWWL long multiplication (33-bit-by-33-bit multiplication producing 66-bit outcome) does not offer rounding option, and there's no multiply-add/subtract variations, just multiplication. The L in destination type indicates 66-bit width, and is split into a pair of destination registers, lower unsigned 32-bit stored in the low (even) register, and upper 33-bit signed extended in the high (odd) register.

For example, VMulWWL V0, V1, V2:V3 would result in:

```
V2[0] = low_32b(V0[0] * V1[0]);
V3[0] = high_33b(V0[0] * V1[0]);
V2[1] = low_32b(V0[1] * V1[1]);
V3[1] = high_33b(V0[1] * V1[1]);
…
V2[7] = low_32b(V0[7] * V1[7]);
V3[7] = high_33b(V0[7] * V1[7]);
```

VDotP2HHW carries out 2 taps x 16 outputs per instruction. We have two src1 (src1a and src1b) to supply data for first and second taps. We share src2 between each pair of output to supply the two taps per lane.

For example, VDotP2HHW V0, V1, V2, V4:V5 would treat V0, V1, V2 each as Half-word vectors (16H each), V4:V5 together as a Word vector (16W), and perform:

```
   V4[0] += V0[0]*V2[0] + V1[0]*V2[1];
   V5[0] += V0[1]*V2[0] + V1[1]*V2[1];
   V4[1] += V0[2]*V2[2] + V1[2]*V2[3];
   V5[1] += V0[3]*V2[2] + V1[3]*V2[3];
   …
   V4[7] += V0[14]*V2[14] + V1[14]*V2[15];
   V5[7] += V0[15]*V2[14] + V1[15]*V2[15];
```

In this case, conceptual data V4 and V5 are first **interleaved** into one stream to supply added to the multiply-add operation, then **deinterleaved** back to the two registers.

VBlend performs alpha blending between Vsrc1a and Vsrc1b, using Vsrc2 as the blending factor. There is no rounding option, as rounding is tied to the supported data type, 7 bits for Byte type, and 15 bits for Half-word type.

For example, in VBlendH V0, V1, V2, V3, we treat each vector as 16 Half-words, say V0 = A, V1 = B, V2 = alpha, V3 = Y, and perform

```
Y[0] = A[0] + round(B[0] * alpha[0] – A[0] * alpha[0], 15);
Y[1] = A[1] + round(B[1] * alpha[1] – A[1] * alpha[1], 15);
…
Y[15] = A[15] + round(B[15] * alpha[15] – A[15] * alpha[15], 15);
```

VCMulH and VCMulHHW complex multiplication is supported with rounding/truncation but no 3rd source input. Real and imaginary parts are interleaved in lanes.

For example, say we have VCMulH V0, V1, V2 instructions. V0, V1 each holds a vector of 8 complex numbers (16 real + imaginary components), and outcome V2 is another vector of 8 complex numbers. VPU calculates:

```
V2[0] = V0[0] * V1[0] – V0[1] * V1[1];          // C0.R = A0.R * B0.R – A0.I * B0.I
V2[1] = V0[0] * V1[1] + V0[1] * V1[0];    // C0.I  = A0.R * B0.I + A0.I * B0.R
…
V2[14] = V0[14] * V1[14] – V0[15] * V1[15]; // C7.R = A7.R * B7.R – A7.I * B7.I
V2[15] = V0[14] * V1[15] + V0[15] * V1[14]; // C7.I  = A7.R * B7.I + A7.I * B7.R
```

For VCMulHHW, outputs are **deinterleaved** between the two vector registers. For example, VCMulHHW V0, V1, V2:V3 would lead to:

```
V2[0] = V0[0] * V1[0] – V0[1] * V1[1];              // C0.R = A0.R * B0.R – A0.I * B0.I
V3[0] = V0[0] * V1[1] + V0[1] * V1[0];              // C0.I = A0.R * B0.I + A0.I * B0.R
   …
V2[7] = V0[14] * V1[14] – V0[15] * V1[15];  // C7.R = A7.R * B7.R – A7.I * B7.I
V3[7] = V0[14] * V1[15] + V0[15] * V1[14];  // C7.I = A7.R * B7.I + A7.I * B7.R
```

9.8.7.2 Instruction Summary

Table 35. Vector multiply-add instructions

Function	Assembly Format	Comments
Vector multiply	VMul<type> .R/T<bits> Vsrc1, Vsrc2/Rsrc2, Vdst/ACdst VMul<type> .R/T<bits> Vsrc1, Vsrc2/Rsrc2, DVdst/DACdst type = B, H, BBH, HHW, WHW, W W type does not allow Rsrc2. VMulWHW DVsrc1/DWsrc1, DVsrc2, DVdst/DWdst VMulWHW DVsrc1, DWsrc2, DVdst/DWdst VMulW.T<bits> DVsrc1/DWsrc1, DVsrc2, DVdst/DWdst VMulW.T<bits> DVsrc1, DWsrc2, DVdst/DWdst .R0/T4/T8/T16 for W	Vdst = round/trunc(Vsrc1 * Vsrc2, bits)
Vector multiply-add/sub	<pred> VMAdd<type>_CA . R/T<bits> Vsrc1, Vsrc2/Rsrc2, ACsrc3dst/DACsrc3dst VMAdd<type>.R/T<bits> Vsrc1, Vsrc2/Rsrc2, Vsrc3dst/DVsrc3dst pred> VMSub<type>_CA . R/T<bits> Vsrc1, Vsrc2/Rsrc2, ACsrc3dst/DACsrc3dst VMSub<type>.R/T<bits> Vsrc1, Vsrc2/Rsrc2, Vsrc3dst/DVsrc3dst type = B, H, BBH, HHW, WHW, W W type does not allow Rsrc2.	Vdst = -Vdst +/- round/trunc(Vsrc1 * Vsrc2, nbits)
DV multiply-add/sub	<pred> VMAdd<type>_CA.R/T<bits> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DACsrc3dst/QACsrc3dst <pred> VMSub<type>_CA.R/T<bits> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DACsrc3dst / QACsrc3dst type = {B, BBH, H, HHW, WHW, W} W type does not allow Rsrc2. .R0 only for B, BBH, H, HHW, WHW .R0/T4/T8/T16 for W	
Vector long multiply	VMulWWL Vsrc1, Vsrc2, DVdst/DACdst	Perform signed 33-bit x signed 33-bit multiplication producing 66-bit product, lower 32-bit unsigned in Vdst.lo and upper 34-bit sign-extended in Vdst.hi, no rounding

Function	Assembly Format	Comments
Vector complex multiply	VCMulHHW Vsrc1, Vsrc2, DACdst (Gen-1) VCMulHHW DVsrc1, DWsrc2, QACdst VCMulWHW DVsrc1, Wsrc2, DACdst	
Vector dot product, 2-term	<pred> VDotP2BBH/HHW_CA Vsrc1a, Vsrc1b, Vsrc2, DACsrc3dst <pred> VDotP2WHW_CA Vsrc1a, Vsrc1b, Vsrc2, ACsrc3dst <pred> VDotP2W_CA.T16 Vsrc1a, Vsrc1b, Vsrc2, ACsrc3dst	Perform 2-term dot product, Vdst += Vsrc1a * Vsrc2_even + Vsrc1b * Vsrc2_odd
Vector dot product 2-term with negation	<pred> VDotPN2<type>_CA Vsrc1a, Vsrc1b, Vsrc2, Vsrc3dst type = WHW	Perform a variation of 2-term dot product, Vdst += Vsrc1a * Vsrc2_even – Vsrc1b * Vsrc2_odd
Vector blend	VBlend<type> Vsrc1a, Vsrc1b, Vsrc2/Rsrc2, Vdst VBlend<type> Wsrc1a, Wsrc1b, Vsrc2/Rsrc2, Vdst type = B, H, W	Vsrc1a = X0, Vsrc1b = X1, Vsrc2 = alpha. Vdst = round(X1*alpha – X0*alpha, nbits) + X0
Vector blend horizontal	VHBlend_I<type> Vsrc1a, Vsrc1b, Vsrc2, Vdst type = B, H, W, BHB	
Vector double multiply	VMul2<type> . R/T<bits> DVsrc1, Vsrc2/Rsrc2, DVdst/DACdst type = B, H, WHW Rsrc2 option available for B/H types only WHW type requires .T16	Vsrc1 and Vdst are double vector. Double multiplication sharing Vsrc2
Vector 4-tap filter	<pred> VFilt4<type>_CA Vsrc1a, Vsrc1b, Wsrc2, DACsrc3dst type = BBH, HHW	Vsrc1a, Vsrc1b supplies overlapping data vector offset by 4 entries
Vector 4-tap x 2 filter	<pred> VFilt4x2<type>_CA Vsrc1a, Vsrc1b, DWsrc2, DACsrc3dst type = BBH, HHW	Vsrc1a, Vsrc1b supplies overlapping data vector offset by 4 entries
Vector 4x2-tap x 2 filter	<pred> VFilt4x2x2BBH_CA DVsrc1a, DVsrc1b, DWsrc2, QACsrc3dst <pred> VFilt4x2x2BBW_CA DVsrc1a, DVsrc1b, DWsrc2, QXACsrc3dst	For BBH/BBW, DVsrc1a, DVsrc1b each supplies overlapping data vector offset by 8 entries.
Vector XNor add 8x4x2	<pred> VXNorAdd8x4x2_CA DVsrc1a, DVsrc1b, DWsrc2, QXACsrc3dst	Convolution between binary data & coefficients, 8 horizontal x 4 vertical taps x 2 sets per byte lane
Vector 4-term dot product	<pred> VDotP4HHW_CA DVsrc1a, DVsrc1b, Wsrc2, DACsrc2dst <pred> VDotP4WHW_CA DVsrc1a, DVsrc1b, Wsrc2, ACsrc2dst <pred> VDotP4BBW_CA DVsrc1a, DVsrc1b, Wsrc2, DXACsrc3dst	DVsrc1a, DVsrc1b together supplies 4 independent data terms

Function	Assembly Format	Comments
Vector 4-term x 2 dot product	<pred> VDotP4x2BBH_CA DVsrc1a, DVsrc1b, DWsrc2, QACsrc3dst <pred> VDotP4x2HHW_CA DVsrc1a, DVsrc1b, DWsrc2, QACsrc3dst <pred> VDotP4x2BBW_CA DVsrc1a, DVsrc1b, DWsrc2, QXACsrc3dst	DVsrc1a, DVsrc1b together supplies 4 independent data terms
Vector 2-term x 2 dot product	<pred> VDotP2x2W_CA.T16 Vsrc1a, Vsrc1b, DWsrc2, DACsrc3dst	Vsrc1a, Vsrc1b together supplies 2 independent data terms
Vector sum of squares	VSumSq<type> Vsrc1, Vsrc2, Vdst/DVdst type = BBH, HHW, W.T16	dst = src1^2 + src2^2 Truncate each term by 16 bits for W type
Vector square of sum	VSqSum<type> Vsrc1, Vsrc2, DVdst type = BBH, HHW	dst = src1^2 + src2^2 + 2*src1*src2
Vector 2x2 matrix determinant	VDet2x2<type> DVsrc1, DVsrc2, Vdst/DVdst VDet2x2<type> DVsrc1, DWsrc2, Vdst/DVdst VDet2x2<type> DWsrc1, DVsrc2, Vdst/DVdst type = HHW, W.T16	dst = src1.lo * src2.hi – src1.hi * src2.lo

9.8.7.3 VMUL

Instruction name	VMUL
Functionality	Vector multiply
Assembly format	VMul<type>.R/T<bits> Vsrc1, Vsrc2/Rsrc2, Vdst/ACdst VMul<type>.R/T<bits> Vsrc1, Vsrc2/Rsrc2, DVdst/DACdst Rounding 0 bits (.R0) is omitted. For example, VMulH.R7 V2, V3, V4 VMulBBH V2, V3, V4:V5 VMulBBH V2, R3, V4:V5 VMulHHW V2, V3, AC0:AC1
Type and bit width	B: 32 x (9-bit src1/src2 → 12-bit dst) H: 16 x (17-bit src1/src2 → 24-bit dst) BBH: 32 x (9-bit src1/src2 → 24-bit dst) HHW: 16 x (17-bit src1/src2 → 48-bit dst) WHW: 8 x (33-bit src1, 17-bit src2 → 48-bit dst) W: 8 x (33-bit src1/src2 → 48-bit dst) For W type, only truncation options (R0/T4/T8/T16) are supported, and there is no support for Rsrc2. All other types support full set of rounding/truncation options and Rsrc2.
Predication	not available

Instruction name	VMUL
Source options	src1: single vector register in VRF src2: single vector register in VRF, scalar register (except W type)
Destination options	B/H/WHW/W: dst: single vector register in VRF or ARF BBH/HHW: dst: double vector register in VRF or ARF
Additional options	

Intrinsics/operator	`vcharx vmulb(vcharx src1, vcharx src2, u3imm rnd_opt);`
	`vshortx vmulh(vshortx src1, vshortx src2, u3imm rnd_opt);`
	`dvshortx vmulbh(vcharx src1, vcharx src2, u3imm rnd_opt);`
	`dvintx vmulhw(vshortx src1, vshortx src2, u3imm rnd_opt);`
	`vintx vmulwhw(vintx src1, vintx src2, u3imm rnd_opt);`
	`vintx vmulw(vintx src1, vintx src2, u3imm rnd_opt);`
	`vintx vmulw_t16(vintx src1, vintx src2); // Gen-1 legacy`
	
	`vcharx vmulb(vcharx src1, int src2, u3imm rnd_opt);`
	`vshortx vmulh(vshortx src1, int src2, u3imm rnd_opt);`
	`dvshortx vmulbh(vcharx src1, int src2, u3imm rnd_opt);`
	`dvintx vmulhw(vshortx src1, int src2, u3imm rnd_opt);`
	`vintx vmulwhw(vintx src1, int src2, u3imm rnd_opt);`
	
	`// Double vector pseudo intrinsics`
	`dvcharx dvmulb(dvcharx src1, dvcharx src2, u3imm rnd_opt);`
	`dvshortx dvmulh(dvshortx src1, dvshortx src2, u3imm rnd_opt);`
	`dvintx dvmulwhw(dvintx src1, dvintx src2, u3imm rnd_opt);`
	
	`dvcharx dvmulb(dvcharx src1, int src2, u3imm rnd_opt);`
	`dvshortx dvmulh(dvshortx src1, int src2, u3imm rnd_opt);`
	`dvintx dvmulwhw(dvintx src1, int src2, u3imm rnd_opt);`

Additional details	For each lane, dst = round(src1 * src2, rnd_opt), using the specified B/H/W lane, and taking lower 9/17/33-bit of operand. Exception is WHW; for source 2 we take lower 17-bit of each W lane. For BBH/HHW, destination double vector is **deinterleaved** between the two vector registers. See 6.2.3.6 for data ordering in single/double vector registers. See 9.8.7.1 for rounding/truncation options.

For example, VMulB.R7 V1, V2, V3 has the following data layout and behavior:

V1:	D[0]	D[1]	D[2]	D[3]	…	D[30]	D[31]
V2:	C[0]	C[1]	C[2]	C[3]	…	C[30]	C[31]
V3:	P[0]	P[1]	P[2]	P[3]	…	P[30]	P[31]

P[i] = round(D[i] * C[i], 7); // C[i], D[i] taken from 9 LSBs of each lane

While VMulHHW.T16 V1, V2, AC2:AC3 has the following data layout and behavior:

V1:	D[0]	D[1]	D[2]	D[3]	…	D[14]	D[15]
V2:	C[0]	C[1]	C[2]	C[3]	…	C[14]	C[15]
AC2:	P[0]		P[2]		…	P[14]	
AC3:	P[1]		P[3]			P[15]	

P[i] = truncate(D[i] * C[i], 16); // C[i], D[i] taken from 17 LSBs of each lane

The outcome from input lane 1 is deposited in AC3 lane 0, which is DAC1 (viewing AC2 and AC3 as a double vector) lane 8, outcome from input lane 3 is deposited in AC3 lane 1, which is DAC1 lane 9, and so on.

Instruction name	**VMUL (Gen-2 double throughput)**
Functionality	Vector multiply
Assembly format	VMulWHW DVsrc1, DVsrc2/DWsrc2/Rsrc2, DVdst/DWdst VMulWHW DWsrc1, DVsrc2/Rsrc2, DVdst/DWdst VMulW.T<bits> DVsrc1, DVsrc2/DWsrc2, DVdst/DWdst VMulW.T<bits> DWsrc1, DVsrc2, DVdst/DWdst For Word type, truncating by 0 bit is omitted.
Type and bit width	WHW: 2 x 8 x (33-bit src1, 17-bit src2 → 48-bit dst) W: 2 x 8 x (33-bit src1/src2 → 48-bit dst) For W type, only truncation options (R0/T4/T8/T16) are supported, and there is no support for Rsrc2. For WHW type, only no rounding/truncation (R0) is supported, and there is support for Rsrc2.
Predication	not available
Source options	src1: double vector register in VRF/WRF src2: double vector register in VRF/WRF or scalar register excluding both src1 and src2 from WRF
Destination options	Double vector register in VRF/WRF
Additional options	

Intrinsics/operator	`dvintx dvmulwhw(dvintx src1, dvintx src2);`
	`dvintx dvmulwhw(dvintx src1, int src2);`
	`dvintx dvmulw(dvintx src1, dvintx src2, u3imm rnd_opt);`
	`dvintx dvmulw_t16(dvintx src1, dvintx src2); // Gen-1 legacy`

Additional details	For each lane, dst = src1 * src2, or (src1 * src2) >> trunc_bits. No rounding is supported for VMulWHW. Truncation by 0/4/8/16 bits is supported for VMulW.

For example, VMulWHW V0:V1, V2:V3, V4:V5 has the following data layout and behavior:

V0:	D[0]	D[1]	D[2]	D[3]	…	D[7]
V2:	C[0]	C[1]	C[2]	C[3]	…	C[7]
V4:	P[0]	P[1]	P[2]	P[3]	…	P[7]

V1:	D[8]	D[9]	D[10]	D[11]	…	D[15]
V3:	C[8]	C[9]	C[10]	C[11]	…	C[15]
V5:	P[8]	P[9]	P[10]	P[11]	…	P[15]

```
P[i] = D[i] * C[i];        // D[i] taken from 33 LSBs of each lane,
                           // C[i] taken from 17 LSBs of each lane
```

There is nothing wrong with drawing the layout as a single row per operand, showing lane 0, 1, …, 15. The above style of drawing it as two rows matches with micro-architecture of the SIMD units inside the processor, and is more consistent across various MAC instructions.

9.8.7.4 VMADD_CA

Instruction name	**VMADD_CA**
Functionality	Vector multiply-add
Assembly format	<pred> VMAdd<type>_CA.R/T<bits> Vsrc1, Vsrc2/Rsrc2, ACsrc3dst/DACsrc3dst pred = none, [P2..P15] type = {B, H, BBH, HHW, WHW, W} .R0 omitted VMAdd<type>.R/T<bits> Vsrc1, Vsrc2/Rsrc2, Vsrc3dst/DVsrc3dst type = {B, BBH, H, HHW, WHW, W} .R0 omitted
Type and bit width	B: 32 x (9-bit src1/src2, 12-bit src3dst) H: 16 x (17-bit src1/src2, 24-bit src3dst) BBH: 32 x (9-bit src1/src2, 24-bit src3dst) HHW: 16 x (17-bit src1/src2, 48-bit src3dst) WHW: 8 x (33-bit src1, 17-bit src2, 48-bit src3dst) W: 8 x (33-bit src1/src2, 48-bit src3dst) For W type, only truncation options (R0/T4/T8/T16) are supported, and there is no support for Rsrc2. All other types support full set of rounding/truncation options and Rsrc2.
Predication	Available across lanes to clear accumulator

Instruction name	VMADD_CA
Source options	src1: single vector register in VRF src2: single vector register in VRF or scalar register (except W type)
Destination options	B/H/WHW/W: src3dst: single vector register in ARF or VRF BBH/HHW: src3dst: double vector register in ARF or VRF
Additional options	

Intrinsics/ operator	`// predicated`
	`vcharx vmaddb(vcharx src1, vcharx src2, vcharx src3, u3imm rnd_opt, int pred);`
	`vshortx vmaddh(vshortx src1, vshortx src2, vshortx src3, u3imm rnd_opt, int pred);`
	`dvshortx vmaddbh(vcharx src1, vcharx src2, dvshortx src3, u3imm rnd_opt,int pred);`
	`dvintx vmaddhw(vshortx src1, vshortx src2, dvintx src3, u3imm rnd_opt, int pred);`
	`vintx vmaddwhw(vintx src1, vintx src2, vintx src3, u3imm rnd_opt, int pred);`
	`vintx vmaddw(vintx src1, vintx src2, vintx src3, u3imm rnd_opt, int pred);`
	`vintx vmaddw_t16(vintx src1, vintx src2, vintx src3, int pred);`
	
	`vcharx vmaddb(vcharx src1, int src2, vcharx src3, u3imm rnd_opt, int pred);`
	`vshortx vmaddh(vshortx src1, int src2, vshortx src3, u3imm rnd_opt, int pred);`
	`dvshortx vmaddbh(vcharx src1, int src2, dvshortx src3, u3imm rnd_opt, int pred);`
	`dvintx vmaddhw(vshortx src1, int src2, dvintx src3, u3imm rnd_opt, int pred);`
	`vintx vmaddwhw(vintx src1, int src2, vintx src3, u3imm rnd_opt, int pred);`
	
	`// unpredicated`
	`vcharx vmaddb(vcharx src1, vcharx src2, vcharx src3, u3imm rnd_opt);`
	`vshortx vmaddh(vshortx src1, vshortx src2, vshortx src3, u3imm rnd_opt);`
	`dvshortx vmaddbh(vcharx src1, vcharx src2, dvshortx src3, u3imm rnd_opt);`
	`dvintx vmaddhw(vshortx src1, vshortx src2, dvintx src3, u3imm rnd_opt);`
	`vintx vmaddwhw(vintx src1, vintx src2, vintx src3, u3imm rnd_opt);`
	`vintx vmaddw(vintx src1, vintx src2, vintx src3, u3imm rnd_opt);`
	`vintx vmaddw_t16(vintx src1, vintx src2, vintx src3);`
	
	`vcharx vmaddb(vcharx src1, int src2, vcharx src3, u3imm rnd_opt);`
	`vshortx vmaddh(vshortx src1, int src2, vshortx src3, u3imm rnd_opt);`
	`dvshortx vmaddbh(vcharx src1, int src2, dvshortx src3, u3imm rnd_opt);`
	`dvintx vmaddhw(vshortx src1, int src2, dvintx src3, u3imm rnd_opt);`
	`vintx vmaddwhw(vintx src1, int src2, vintx src3, u3imm rnd_opt);`
	
	`// Double vector pseudo intrinsics, when (rnd_opt != 0)`
	`dvcharx dvmaddb(dvcharx src1, dvcharx src2, dvcharx src3, u3imm rnd_opt,int pred);`
	`dvshortx dvmaddh(dvshortx src1, dvshortx src2, dvshortx src3, u3imm rnd_opt,`
	`int pred);`
	`dvintx dvmaddwhw(dvintx src1, dvintx src2, dvintx src3, u3imm rnd_opt, int pred);`
	
	`void dvmaddbh(dvcharx src1, dvcharx src2, dvshortx src30, dvshortx src31, u3imm`
	`rnd_opt, int pred, dvshortx & dst0, dvshortx & dst1);`
	
	`void dvmaddhw(dvshortx src1, dvshortx src2, dvintx src30, dvintx src31, u3imm`
	`rnd_opt, int pred, dvintx & dst0, dvintx & dst1);`

Instruction name	VMADD_CA

	`dvcharx dvmaddb(dvcharx src1, int src2, dvcharx src3, u3imm rnd_opt, int pred);`
	`dvshortx dvmaddh(dvshortx src1, int src2, dvshortx src3, u3imm rnd_opt, int pred);`
	`dvintx dvmaddwhw(dvintx src1, int src2, dvintx src3, u3imm rnd_opt, int pred);`
	
	`void dvmaddbh(dvcharx src1, int src2, dvshortx src30, dvshortx src31, u3imm rnd_opt, int pred, dvshortx & dst0, dvshortx & dst1);`
	
	`void dvmaddhw(dvshortx src1, int src2, dvintx src30, dvintx src31, u3imm rnd_opt, int pred, dvintx & dst0, dvintx & dst1);`

Additional details	For each lane, src3dst **+=** round/trunc(src1 * src2, rnd_opt), using the specified B/H/W lane, and taking lower 9/17/33-bit of operand. Exception is WHW; for source 2 we take lower 17-bit of each W lane. When predicate is off, only multiply-round is performed, src3dst **=** round/trunc(src1 * src2, rnd_opt), effectively clearing the accumulator. For BBH/HHW, destination double vector registers are **deinterleaved** between the two vector registers. See 6.2.3.6 for data ordering in single/double vector registers. See 9.8.7.1 for rounding/truncating options. For W,.R0/T4/T8/T16 options are supported. Note that we do not support scalar source 2 when source 2 is of the Word type. This is because for Word type we would like to use 33 bits so we can support both signed 32-bit and unsigned 32-bit values. Scalar register is only 32-bit wide so cannot supply 33 bits, and we do not want to create variation of behavior between source 2 being from a vector or a scalar, nor do we want to have Signed/Unsigned designation in the instruction itself (like scalar having LMULSS/SU/UU), so we just don't support scalar source 2.

For example, VMAddB.R7 V1, V2, V3 has the following data layout and behavior:

V1:	D[0]	D[1]	D[2]	D[3]	…	D[30]	D[31]
V2:	C[0]	C[1]	C[2]	C[3]	…	C[30]	C[31]
V3:	A[0]	A[1]	A[2]	A[3]	…	A[30]	A[31]

A[i] = A[i] + round(D[i] * C[i], 7); // C[i], D[i] taken from 9 LSBs of each lane

While [P2] VMAddHHW.T16 V1, V2, AC2:AC3 has the following data layout and behavior:

V1:	D[0]	D[1]	D[2]	D[3]	…	D[14]	D[15]
V2:	C[0]	C[1]	C[2]	C[3]	…	C[14]	C[15]
AC2:	A[0]		A[2]		…	A[14]	
AC3:	A[1]		A[3]			A[15]	

A[i] = P2 ? (A[i] + truncate(D[i] * C[i], 16)) : truncate(D[i] * C[i], 16)

// C[i], D[i] taken from 17 LSBs of each lane

The accumulator for input lane 1 is mapped to AC3 lane 0, which is DAC1 (viewing AC2 and AC3 as a double vector) lane 8, accumulator for input lane 3 is mapped to AC3 lane 1, which is DAC1 lane 9, and so on.

Instruction name	**VMADD_CA (Gen-2 double vector/throughput)**
Functionality	Vector multiply-add
Assembly format	<pred> VMAdd<type>_CA.R/T<bits> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DACsrc3dst /QACsrc3dst pred = none, [P2..P15] type = {B, BBH, H, HHW, WHW, W} .R0 only for B, BBH, H, HHW, WHW .R0/T4/T8/T16 for W
Type and bit width	B: 2 x 32 x (9-bit src1/src2, 12-bit src3dst) BBH: 2 x 32 x (9-bit src1/src2, 24-bit src3dst) H: 2 x 16 x (17-bit src1/src2, 24-bit src3dst) HHW: 2 x 16 x (17-bit src1/src2, 48-bit src3dst) WHW: 2 x 8 x (33-bit src1, 17-bit src2, 48-bit src3dst) W: 2 x 8 x (33-bit src1/src2, 48-bit src3dst) For W type, only truncation options (R0/T4/T8/T16) are supported, and there is no support for Rsrc2. All other types support no rounding/truncation option (R0) and Rsrc2.
Predication	Available across lanes to clear accumulator
Source options	src1: double vector register in VRF src2: double vector register in VRF/WRF (all types) or scalar register (all except W type)
Destination options	B/H/WHW/W: src3dst: double vector register in ARF BBH/HHW: src3dst: quad vector register in ARF
Additional options	

Intrinsics/ operator	`// Note that some of the following intrinsic function names are the same as double vector`
	`// pseudo intrinsic functions in the non-double vector/throughput variations of VMAdd_CA.`
	`// For b, h, whw, bh, hw types, intrinsic functions are implemented to map to double`
	`// vector/throughput instructions when (rnd_opt == 0). Otherwise, each intrinsic function`
	`// maps to 2 instances of the single vector instructions. For w type, the double`
	`// vector intrinsic function always maps to a double vector/throughput instruction.`
	
	`dvcharx dvmaddb(dvcharx src1, dvcharx src2, dvcharx src3, u3imm rnd_opt, int pred);`
	`dvshortx dvmaddh(dvshortx src1,dvshortx src2,dvshortx src3, u3imm rnd_opt, int pred);`
	`dvintx dvmaddwhw(dvintx src1, dvintx src2, dvintx src3, u3imm rnd_opt, int pred);`
	`dvintx dvmaddw(dvintx src1, dvintx src2, dvintx src3, u3imm rnd_opt, int pred);`
	`dvintx dvmaddw_t16(dvintx src1, dvintx src2, dvintx src3, int pred);`
	
	`void dvmaddhw(dvshortx src1, dvshortx src2, dvintx src3_0, dvintx src3_1, u3imm`

Instruction name	VMADD_CA (Gen-2 double vector/throughput)

	rnd_opt, int pred, dvintx & dst_0, dvintx & dst_1);
	
	void dvmaddbh(dvcharx src1, dvcharx src2, dvshortx src3_0, dvshortx src3_1, u3imm rnd_opt, int pred, dvshortx & dst_0, dvshortx & dst_1);
	
	dvcharx dvmaddb(dvcharx src1, int src2, dvcharx src3, u3imm rnd_opt, int pred);
	dvshortx dvmaddh(dvshortx src1, int src2,dvshortx src3, u3imm rnd_opt, int pred);
	dvintx dvmaddwhw(dvintx src1, int src2, dvintx src3, u3imm rnd_opt, int pred);
	
	void dvmaddbh(dvcharx src1, int src2, dvshortx src3_0, dvshortx src3_1, u3imm rnd_opt, int pred, dvshortx & dst_0, dvshortx & dst_1);
	
	void dvmaddhw(dvshortx src1, int src2, dvintx src3_0, dvintx src3_1, u3imm rnd_opt, int pred, dvintx & dst_0, dvintx & dst_1);

Additional details	

For example, [P3] VMAddHHW V0:V1, V2:V3, AC0:AC3 has the following data layout and behavior:

V0:	D[0]	D[2]	D[4]	D[6]	…	D[28]	D[30]
V1:	D[1]	D[3]	D[5]	D[7]	…	D[29]	D[31]
V2:	C[0]	C[2]	C[4]	C[6]	…	C[28]	C[30]
V3:	C[1]	C[3]	C[5]	C[7]	…	C[29]	C[31]
AC0:	ACC[0]		ACC[4]		…	ACC[28]	
AC1:	ACC[2]		ACC[6]		…	ACC[30]	
AC2:	ACC[1]		ACC[5]		…	ACC[29]	
AC3:	ACC[3]		ACC[7]		…	ACC[31]	

ACC[i] = P3 ? (ACC[i] + D[i] * C[i]) : (D[i] * C[i]);

9.8.7.5 VMSUB_CA

Instruction name	VMSUB_CA
Functionality	Vector multiply-subtract
Assembly format	<pred> VMSub<type>_CA.R/T<bits> Vsrc1, Vsrc2/Rsrc2, ACsrc3dst/DACsrc3dst pred = none, [P2..P15] type = {B, H, BBH, HHW, WHW, W} .R0 omitted VMSub<type>.R/T<bits> Vsrc1, Vsrc2/Rsrc2, DVsrc3dst type = {B, BBH, H, HHW, WHW, W}

Instruction name	VMSUB_CA
	.R0 omitted
Type and bit width	B: 32 x (9-bit src1/src2, 12-bit src3dst) H: 16 x (17-bit src1/src2, 24-bit src3dst) BBH: 32 x (9-bit src1/src2, 24-bit src3dst) HHW: 16 x (17-bit src1/src2, 48-bit src3dst) WHW: 8 x (33-bit src1, 17-bit src2, 48-bit src3dst) W: 8 x (33-bit src1/src2, 48-bit src3dst) For W type, only truncation options (R0/T4/T8/T16) are supported, and there is no support for Rsrc2. All other types support full set of rounding/truncation options and Rsrc2.
Predication	Available across lanes to clear accumulator (except W type)
Source options	src1: single vector register in VRF src2: single vector register in VRF or scalar register
Destination options	B/H/WHW: src3dst: single vector register in ARF or VRF BBH/HHW: src3dst: double vector register in ARF or VRF
Additional options	

Intrinsics/ operator	`// predicated`
	`vcharx vmsubb(vcharx src1, vcharx src2, vcharx src3, u3imm rnd_opt, int pred);`
	`vshortx vmsubh(vshortx src1, vshortx src2, vshortx src3, u3imm rnd_opt, int pred);`
	`dvshortx vmsubbh(vcharx src1, vcharx src2, dvshortx src3, u3imm rnd_opt, int pred);`
	`dvintx vmsubhw(vshortx src1, vshortx src2, dvintx src3, u3imm rnd_opt, int pred);`
	`vintx vmsubwhw(vintx src1, vintx src2, vintx src3, u3imm rnd_opt, int pred);`
	`vintx vmsubw(vintx src1, vintx src2, vintx src3, u3imm rnd_opt, int pred);`
	`vintx vmsubw_t16(vintx src1, vintx src2, vintx src3, int pred);`
	
	`vcharx vmsubb(vcharx src1, int src2, vcharx src3, u3imm rnd_opt, int pred);`
	`vshortx vmsubh(vshortx src1, int src2, vshortx src3, u3imm rnd_opt, int pred);`
	`dvshortx vmsubbh(vcharx src1, int src2, dvshortx src3, u3imm rnd_opt, int pred);`
	`dvintx vmsubhw(vshortx src1, int src2, dvintx src3, u3imm rnd_opt, int pred);`
	`vintx vmsubwhw(vintx src1, int src2, vintx src3, u3imm rnd_opt, int pred);`
	
	`// unpredicated`
	`vcharx vmsubb(vcharx src1, vcharx src2, vcharx src3, u3imm rnd_opt);`
	`vshortx vmsubh(vshortx src1, vshortx src2, vshortx src3, u3imm rnd_opt);`
	`dvshortx vmsubbh(vcharx src1, vcharx src2, dvshortx src3, u3imm rnd_opt);`
	`dvintx vmsubhw(vshortx src1, vshortx src2, dvintx src3, u3imm rnd_opt);`
	`vintx vmsubwhw(vintx src1, vintx src2, vintx src3, u3imm rnd_opt);`
	`vintx vmsubw(vintx src1, vintx src2, vintx src3, u3imm rnd_opt);`
	`vintx vmsubw_t16(vintx src1, vintx src2, vintx src3);`
	
	`vcharx vmsubb(vcharx src1, int src2, vcharx src3, u3imm rnd_opt);`
	`vshortx vmsubh(vshortx src1, int src2, vshortx src3, u3imm rnd_opt);`
	`dvshortx vmsubbh(vcharx src1, int src2, dvshortx src3, u3imm rnd_opt);`
	`dvintx vmsubhw(vshortx src1, int src2, dvintx src3, u3imm rnd_opt);`

Instruction name	VMSUB_CA

	`vintx vmsubwhw(vintx src1, int src2, vintx src3, u3imm rnd_opt);`
	
	`// Double vector pseudo intrinsics, when (rnd_opt != 0)`
	`dvcharx dvmsubb(dvcharx src1, dvcharx src2, dvcharx src3, u3imm rnd_opt, int pred);`
	`dvshortx dvmsubh(dvshortx src1, dvshortx src2, dvshortx src3, u3imm rnd_opt, int pred);`
	`dvintx dvmsubwhw(dvintx src1, dvintx src2, dvintx src3, u3imm rnd_opt, int pred);`
	
	`void dvmsubbh(dvcharx src1, dvcharx src2, dvshortx src30, dvshortx src31, u3imm rnd_opt, int pred, dvshortx & dst0, dvshortx & dst1);`
	
	`void dvmsubhw(dvshortx src1, dvshortx src2, dvintx src30, dvintx src31, u3imm rnd_opt, int pred, dvintx & dst0, dvintx & dst1);`
	
	`dvcharx dvmsubb(dvcharx src1, int src2, dvcharx src3, u3imm rnd_opt, int pred);`
	`dvshortx dvmsubh(dvshortx src1, int src2, dvshortx src3, u3imm rnd_opt, int pred);`
	`dvintx dvmsubwhw(dvintx src1, int src2, dvintx src3, u3imm rnd_opt, int pred);`
	
	`void dvmsubbh(dvcharx src1, int src2, dvshortx src30, dvshortx src31, u3imm rnd_opt, int pred, dvshortx & dst0, dvshortx & dst1);`
	
	`void dvmsubhw(dvshortx src1, int src2, dvintx src30, dvintx src31, u3imm rnd_opt, int pred, dvintx & dst0, dvintx & dst1);`

Additional details	For each lane, src3dst **-=** round(src1 * src2, rnd_opt), using the specified B/H/W lane, and taking lower 9/17/33-bit of operand. Exception is WHW; for source 2 we take lower 17-bit of each W lane. When predicate is off, only multiply-round is performed, src3dst **=** round(src1 * src2, rnd_opt), effectively clearing the accumulator. For BBH/HHW, destination double vector registers are **deinterleaved** between the two vector registers. See Data Ordering in Single and Double Vector Registers for data ordering in single/double vector registers. See Types and Data Widths for rounding/truncating options. For W,.R0/T4/T8/T16 options are supported. Note that we do not support scalar source 2 when source 2 is of the Word type. This is because for Word type we would like to use 33 bits so we can support both signed 32-bit and unsigned 32-bit values. Scalar register is only 32-bit wide so cannot supply 33 bits, and we do not want to create variation of behavior between source 2 being from a vector or a scalar, nor do we want to have Signed/Unsigned designation in the instruction itself (like scalar having LMULSS/SU/UU), so we just don't support scalar source 2.

See VMAdd_CA for data layout and behavior examples.

Instruction name	VMSUB_CA (Gen-2 double vector/throughput)
Functionality	Vector multiply-subtract
Assembly format	<pred> VMSub<type>_CA.R/T<bits> DVsrc1, DVsrc2/DWsrc2/Rsrc2, DACsrc3dst /QACsrc3dst pred = none, [P2..P15]

Instruction name	VMSUB_CA (Gen-2 double vector/throughput)
	type = {B, BBH, H, HHW, WHW, W} .R0 only for BBH, HHW, WHW .R0/T4/T8/T16 for W
Type and bit width	B: 2 x 32 x (9-bit src1/src2, 12-bit src3dst) BBH: 2 x 32 x (9-bit src1/src2, 24-bit src3dst) H: 2 x 16 x (17-bit src1/src2, 24-bit src3dst) HHW: 2 x 16 x (17-bit src1/src2, 48-bit src3dst) WHW: 2 x 8 x (33-bit src1, 17-bit src2, 48-bit src3dst) W: 2 x 8 x (33-bit src1/src2, 48-bit src3dst) For W type, only truncation options (R0/T4/T8/T16) are supported, and there is no support for Rsrc2. All other types support no rounding/truncation option (R0) and Rsrc2.
Predication	Available across lanes to clear accumulator
Source options	src1: double vector register in VRF src2: double vector register in VRF/WRF (all types) or scalar register (all except W type)
Destination options	B/H/WHW/W: src3dst: double vector register in ARF BBH/HHW: src3dst: quad vector register in ARF
Additional options	

Intrinsics/ operator	`// Note that some of the following intrinsic function names are the same as double vector`
	`// pseudo intrinsic functions in the non-double vector/throughput variations of VMSub_CA.`
	`// For b, h, whw, bh, hw types, intrinsic functions are implemented to map to double`
	`// vector/throughput instructions when (rnd_opt == 0). Otherwise, each intrinsic function`
	`// maps to 2 instances of the single vector instructions. For w type, the double`
	`// vector intrinsic function always maps to a double vector/throughput instruction.`
	
	`dvcharx dvmsubb(dvcharx src1, dvcharx src2, dvcharx src3, u3imm rnd_opt, int pred);`
	`dvshortx dvmsubh(dvshortx src1,dvshortx src2,dvshortx src3, u3imm rnd_opt, int pred);`
	`dvintx dvmsubwhw(dvintx src1, dvintx src2, dvintx src3, u3imm rnd_opt, int pred);`
	`dvintx dvmsubw(dvintx src1, dvintx src2, dvintx src3, u3imm rnd_opt, int pred);`
	`dvintx dvmsubw_t16(dvintx src1, dvintx src2, dvintx src3, int pred);`
	
	`void dvmsubhw(dvshortx src1, dvshortx src2, dvintx src3_0, dvintx src3_1, u3imm rnd_opt, int pred, dvintx & dst_0, dvintx & dst_1);`
	
	`void dvmsubbh(dvcharx src1, dvcharx src2, dvshortx src3_0, dvshortx src3_1, u3imm rnd_opt, int pred, dvshortx & dst_0, dvshortx & dst_1);`
	
	`dvcharx dvmsubb(dvcharx src1, int src2, dvcharx src3, u3imm rnd_opt, int pred);`
	`dvshortx dvmsubh(dvshortx src1, int src2,dvshortx src3, u3imm rnd_opt, int pred);`
	`dvintx dvmsubwhw(dvintx src1, int src2, dvintx src3, u3imm rnd_opt, int pred);`
	
	`void dvmsubbh(dvcharx src1, int src2, dvshortx src3_0, dvshortx src3_1, u3imm`

Instruction name	VMSUB_CA (Gen-2 double vector/throughput)

	`rnd_opt, int pred, dvshortx & dst_0, dvshortx & dst_1);`
	
	`void dvmsubhw(dvshortx src1, int src2, dvintx src3_0, dvintx src3_1, u3imm rnd_opt, int pred, dvintx & dst_0, dvintx & dst_1);`

Additional details	

See VMAdd_CA for data layout and behavior examples.

9.8.7.6 VMULWWL

Instruction name	VMULWWL
Functionality	Vector long multiply
Assembly format	VMul<type> Vsrc1, Vsrc2, DVdst/DACdst
Type and bit width	WWL: 8 x (33-bit src1/src2 → 66-bit → 2 x 48-bit)
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF
Destination options	dst: double vector register in VRF or ARF, lower 32-bit zero-extended in the low register, upper 34-bit sign-extended in the high register
Additional options	
Intrinsics/operator	`dvintx vmulwl(vintx src1, vintx src2);`
Additional details	For each lane, dst = src1 * src2. Destination double vector registers are low/high **deinterleaved** between the two vector registers. See 6.2.3.6 for data ordering in single/double vector registers.

While VMulWWL V1, V2, V4:V5 has the following data layout and behavior:

V1:	D[0]	D[1]	…	D[7]
V2:	C[0]	C[1]	…	C[7]
V4:	P[0].lo	P[1].lo	…	P[7].lo
V5:	P[0].hi	P[1].hi	…	P[7].hi

P[i] = D[i] * C[i]; // C[i], D[i] taken from 33 LSBs of each lane

P[i].lo = P[i] & ((1 << 32) – 1);

P[i].hi = P[i] >> 32;

9.8.7.7 VCMUL

Instruction name	VCMUL (Gen-1)
Functionality	Vector complex multiply

Instruction name	VCMUL (Gen-1)
Assembly format	VCMul<type>.R/T<bits> Vsrc1, Vsrc2, Vdst .R0 is omitted
Type and bit width	H: 8 x (complex 17-bit src1/src2 → 24-bit dst) HHW: 8 x (complex 17-bit src1/src2 → 48-bit dst)
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF
Destination options	H dst: single vector register in VRF HHW dst: double vector register in VRF
Additional options	

Intrinsics/operator	`vshortx vcmulh(vshortx src1, vshortx src2, u3imm rnd_opt);`
	`dvintx vcmulhw(vshortx src1, vshortx src2, u3imm rnd_opt);`
	`// double vector pseudo intrinsics`
	`dvshortx dvcmulh(dvshortx src1, dvshortx src2, u3imm rnd_opt);`

Additional details	Real/imaginary lane interleaved (even lane = real, odd lane = imaginary). Vdst.r = round(Vsrc1.r * Vsrc2.r, nbits) – round(Vsrc1.i * Vsrc2.i, nbits) Vdst.i = round(Vsrc1.r * Vsrc2.i, nbits) + round(Vsrc1.i * Vsrc2.r, nbits) or Vdst.r = trunc(Vsrc1.r * Vsrc2.r, nbits) – trunc(Vsrc1.i * Vsrc2.i, nbits) Vdst.i = trunc(Vsrc1.r * Vsrc2.i, nbits) + trunc(Vsrc1.i * Vsrc2.r, nbits)

For example, VCMulH.R7 V1, V2, V3 has the following data layout and behavior:

V1:	DR[0]	DI[0]	DR[1]	DI[1]	…	DR[7]	DI[7]
V2:	CR[0]	CI[0]	CR[1]	CI[1]	…	CR[7]	CI[7]
V3:	PR[0]	PI[0]	PR[1]	PI[1]	…	PR[7]	PI[7]

PR[i] = round(DR[i] * CR[i] , 7) – round(DI[i] * CI[i], 7);

PI[i] = round(DR[i] * CI[i] , 7) + round(DI[i] * CR[i], 7);

// C*[i], D*[i] taken from 17 LSBs of each lane

While VCMulHHW.T16 V1, V2, V4:V5 has the following data layout and behavior:

V1:	DR[0]	DI[0]	DR[1]	DI[1]	…	DR[7]	DI[7]
V2:	CR[0]	CI[0]	CR[1]	CI[1]	…	CR[7]	CI[7]
V4:	PR[0]		PR[1]		…	PR[7]	
V5:	PI[0]		PI[1]			PI[7]	

PR[i] = truncate(DR[i] * CR[i] , 16) – truncate (DI[i] * CI[i], 16);

```
PI[i] = truncate (DR[i] * CI[i] , 16) + truncate (DI[i] * CR[i], 16);
// C*[i], D*[i] taken from 17 LSBs of each lane
```

The following instruction is added in Gen-2 VPU to accelerate 16-bit x 16-bit and 32-bit x 16-bit FFT by 2x.

Instruction name	VCMUL (added in Gen-2)
Functionality	Vector complex multiply
Assembly format	VCMulHHW DVsrc1, DWsrc2, QACdst VCMulWHW DVsrc1, Wsrc2, DACdst VCMulW.T16 DVsrc1, DWsrc2, DACdst
Type and bit width	HHW: 16 x (complex 17-bit src1/src2 → 48-bit dst) WHW: 8 x (complex 33-bit src1, 17-bit src2 → 48-bit dst) W: 8 x (complex 33-bit src1/src2 → 48-bit dst)
Predication	not available
Source options	src1: double vector register real in .lo, imaginary in .hi, in VRF src2: HHW/W: double vector register real in .lo, imaginary in .hi, in WRF WHW: single vector register real/imaginary interleaved in WRF NOTE THAT storage format is different from single-vector VCMul instructions. In single-vector instructions real/imaginary are interleaved in a single vector, whereas in double-vector instructions real/imaginary are store in .lo and .hi single vector respectively.
Destination options	HHW: quad vector register with even real in .q0, odd real in .q1, even imaginary in .q2, odd imaginary in .q3, in ARF WHW/W: double vector register with real in .lo, imaginary in .hi, in ARF
Additional options	

Intrinsics/operator	`void dvcmulhw(dvshortx src1, dvshortx src2, dvintx &outr, dvintx &outi);`
	`dvintx dvcmulwhw(dvintx src1, vshortx src2);`
	`dvintx dvcmulw_t16(dvintx src1, dvintx src2);`

Additional details	For HHW/WHW: dst.r = src1.r * src2.r – src1.i * src2.i dst.i = src1.r * src2.i + src1.i * src2.r For W: dst.r = (src1.r * src2.r >> 16) – (src1.i * src2.i >> 16) dst.i = (src1.r * src2.i >> 16) + (src1.i * src2.r >> 16)

For example, VCMulHHW V0:V1, W2:W3, AC4:AC7 has the following data layout and behavior:

V0:	DR[0]	DR[1]	…	DR[14]	DR[15]
V1:	DI[0]	DI[1]	…	DI[14]	DI[15]
W2:	CR[0]	CR[1]	…	CR[14]	CR[15]
W3:	CI[0]	CI[1]	…	CI[14]	CI[15]
AC4:	PR[0]		…	PR[14]	
AC5:	PR[1]		…	PR[15]	
AC6:	PI[0]		…	PI[14]	
AC7:	PI[1]		…	PI[15]	

PR[i] = DR[i] * CR[i] – DI[i] * CI[i];

PI[i] = DR[i] * CI[i] + DI[i] * CR[i];

// C*[i], D*[i] taken from 17 LSBs of each lane

For example, VCMulWHW V0:V1, W2, AC4:AC5 has the following data layout and behavior:

V0:	DR[0]		DR[1]		…	DR[6]		DR[7]	
V1:	DI[0]		DI[1]		…	DI[6]		DI[7]	
W2:	CR[0]	CI[0]	CR[1]	CI[1]	…	CR[6]	CI[6]	CR[7]	CI[7]
AC4:	PR[0]		PR[1]		…	PR[6]		PR[7]	
AC5:	PI[0]		PI[1]		…	PI[6]		PI[7]	

PR[i] = DR[i] * CR[i] – DI[i] * CI[i]; // D*[i] taken from 33 LSBs of each lane,

PI[i] = DR[i] * CI[i] + DI[i] * CR[i]; // C*[i] taken from 17 LSBs of each lane

For example, VCMulW.T16 V0:V1, W2:W3, AC4:AC5 has the following data layout and behavior:

V0:	DR[0]	DR[1]	…	DR[7]
V1:	DI[0]	DI[1]	…	DI[7]
W2:	CR[0]	CR[1]	…	CR[7]
W3:	CI[0]	CI[1]	…	CI[7]
AC4:	PR[0]	PR[1]	…	PR[7]
AC5:	PI[0]	PI[1]	…	PI[7]

PR[i] = truncate(DR[i] * CR[i], 16) – truncate(DI[i] * CI[i] , 16); // C*[i], D*[i] taken from

PI[i] = truncate(DR[i] * CI[i] , 16) + truncate(DI[i] * CR[i] , 16); // 33 LSBs of each lane

9.8.7.8 VDOTP2_CA

Instruction name	VDOTP2_CA
Functionality	Vector 2-term dot product
Assembly format	<pred> VDotP2BBH/HHW_CA Vsrc1a, Vsrc1b, Vsrc2, DACsrc3dst <pred> VDotP2WHW_CA Vsrc1a, Vsrc1b, Vsrc2, ACsrc3dst <pred> VDotP2W_CA.T16 Vsrc1a, Vsrc1b, Vsrc2, ACsrc3dst pred = none, [P2..P15] .T16 is available only for W type and is always applied with W type.
Type and bit width	BBH: 32 x (9-bit src1a/src1b, 9-bit src2, 24-bit src3dst) HHW: 16 x (17-bit src1a/src1b, 17-bit src2, 48-bit src3dst) WHW: 8 x (33-bit src1a/src1b, 17-bit src2, 48-bit src3dst) W.T16: 8 x (33-bit src1a/src1b, 33-bit src2, 48-bit src3dst)
Predication	Available across lanes to clear accumulator
Source options	src1a: single vector register in VRF src1b: single vector register in VRF src2: single vector register in VRF
Destination options	src3dst: double vector register (BBH, HHW types) in ARF src3dst: single vector register (WHW, W type) in ARF
Additional options	

Intrinsics/operator	`dvshortx vdotp2_bbh(vcharx src1a, vcharx src1b, vcharx src2, dvshortx src3dst, int pred);`
	
	`dvintx vdotp2_hhw(vshortx src1a, vshortx src1b, vshortx src2, dvintx src3dst, int pred);`
	
	`vintx vdotp2_whw(vintx src1a, vintx src1b, vshortx src2, vintx src3dst, int pred);`
	
	`vintx vdotp2_w_t16(vintx src1a, vintx src1b, vintx src2, vintx src3dst, int pred);`

Additional details	When predicate is off, destination is replaced with the sum of 2 products, Vsrc3dst **=** Vsrc1a * Vsrc2_even + Vsrc1b * Vsrc2_odd, effectively clearing the accumulator. Otherwise, the sum of 2 products is added to the accumulator Vsrc3dst **+=** Vsrc1a * Vsrc2_even + Vsrc1b * Vsrc2_odd BBH: Treat coefficient vector as byte vector (32 x 9-bit), but share a pair of coefficients between a pair of accumulators. for i = 0..15: Vsrc3dst[2*i] += Vsrc1a[2*i] * Vsrc2[2*i] + Vsrc1b[2*i] * Vsrc2[2*i+1] Vsrc3dst[2*i+1] += Vsrc1a[2*i+1] * Vsrc2[2*i] + Vsrc1b[2*i+1] * Vsrc2[2*i+1]

Instruction name	VDOTP2_CA
	HHW: Treat coefficient vector as half-word vector (16 x 17-bit), but share a pair of coefficients between a pair of accumulators. for i = 0..7: Vsrc3dst[2*i] += Vsrc1a[2*i] * Vsrc2[2*i] + Vsrc1b[2*i] * Vsrc2[2*i+1] Vsrc3dst[2*i+1] += Vsrc1a[2*i+1] * Vsrc2[2*i] + Vsrc1b[2*i+1] * Vsrc2[2*i+1] WHW: Treat coefficient vector as half-word vector (16 x 17-bit). for i = 0..7: Vsrc3dst[i] += Vsrc1a[i] * Vsrc2[2*i] + Vsrc1b[i] * Vsrc2[2*i+1] W: Treat coefficient vector as word vector (8 x 48-bit) and each pair of W lanes share 2 coefficients. Each product is truncated by 16 bits. for i = 0..3: Vsrc3dst[2*i] += (Vsrc1a[2*i] * Vsrc2[2*i]>>16) + (Vsrc1b[2*i] * Vsrc2[2*i+1]>>16) Vsrc3dst[2*i+1] += (Vsrc1a[2*i+1] * Vsrc2[2*i]>>16) + (Vsrc1b[2*i+1] * Vsrc2[2*i+1]>>16) See 6.2.3.6 for data ordering in single/double vector registers.

For example, VDotP2BBH_CA V1, V2, V3, AC2:AC3 has the following data layout and behavior:

V1:	D[0]	D[1]	D[2]	D[3]	…	D[30]	D[31]
V2:	E[0]	E[1]	E[2]	E[3]	…	E[30]	E[31]
V3:	C[0][0]	C[1][0]	C[0][1]	C[1][1]	…	C[0][15]	C[1][15]
AC2:	A[0]		A[2]		…	A[30]	
AC3:	A[1]		A[3]		…	A[31]	

A[2*i] = A[2*i] + C[0][i] * D[2*i] + C[1][i] * E[2*i];

A[2*i + 1] = A[2*I + 1] + C[0][i] * D[2*i + 1] + C[1][i] * E[2*i + 1];

While VDotP2W.T16 V1, V2, V3, AC2 has the following data layout and behavior:

V1:	D[0]	D[1]	D[2]	D[3]	…	D[6]	D[7]
V2:	E[0]	E[1]	E[2]	E[3]	…	E[6]	E[7]
V3:	C[0][0]	C[1][0]	C[0][1]	C[1][1]	…	C[0][3]	C[1][3]
AC2:	A[0]	A[1]	A[2]	A[3]	…	A[6]	A[7]

A[2*i] = A[2*i] + truncate(C[0][i] * D[2*i], 16) + truncate(C[1][i] * E[2*i], 16);

A[2*i + 1] = A[2*I + 1] + truncate(C[0][i] * D[2*i + 1], 16) + truncate(C[1][i] * E[2*i + 1], 16);

9.8.7.9 VDOTPN2_CA

Instruction name	VDOTPN2_CA
Functionality	Vector 2-term dot product variation
Assembly format	<pred> VDotPN2<type>_CA Vsrc1a, Vsrc1b, Vsrc2, Vsrc3dst pred = none, [P2..P15]
Type and bit width	WHW: 8 x (33-bit src1a/src1b, 17-bit src2, 48-bit src3dst)
Predication	Available across lanes to clear accumulator
Source options	src1a: single vector register in VRF src1b: single vector register in VRF src2: single vector register in VRF
Destination options	src3dst: single vector register in ARF
Additional options	
Intrinsics/operator	`vintx vdotpn2_whw(vintx src1a, vintx src1b, vshortx src2, vintx src3dst, int pred);`
Additional details	Perform multiply add/sub, Vsrc3dst **+=** Vsrc1a * Vsrc2_even – Vsrc1b * Vsrc2_odd when predicate is on. When predicate is off, destination is replaced with the difference of products, Vsrc3dst **=** Vsrc1a * Vsrc2_even – Vsrc1b * Vsrc2_odd, effectively clearing the accumulator. Treat coefficient vector as half-word vector. for i = 0..7: Vsrc3dst[i] += Vsrc1a[i]*Vsrc2[2*i] – Vsrc1b[i]*Vsrc2[2*i+1]

See VDotP2_CA for data layout and behavior.

9.8.7.10 VBLEND

Instruction name	VBLEND
Functionality	Vector blend
Assembly format	VBlend<type> Vsrc1a, Vsrc1b, Vsrc2/Rsrc2, Vdst VBlend<type> Wsrc1a, Wsrc1b, Vsrc2/Rsrc2, Vdst
Type and bit width	B: 32 x (9-bit signed src1a/src1b, 8-bit unsigned src2 → 12-bit dst) H: 16 x (17-bit signed src1a/src1b, 16-bit unsigned src2 → 24-bit dst) W: 8 x (33-bit signed src1a/src1b, 32-bit unsigned src2 → 48-bit dst)
Predication	not available
Source options	src1a: single vector register in VRF or WRF src1b: single vector register in VRF or WRF src2: single vector register in VRF or scalar register
Destination options	dst: single vector register in VRF
Additional options	

Intrinsics/operator	`vcharx vblend(vcharx src1a, vcharx src1b, vcharx src2);`
	`vshortx vblend(vshortx src1a, vshortx src1b, vshortx src2);`
	`vintx vblendw_q15(vintx src1a, vintx src1b, vintx src2);`
	`vcharx vblend(vcharx src1a, vcharx src1b, unsigned int src2);`
	`vshortx vblend(vshortx src1a, vshortx src1b, unsigned int src2);`
	`vintx vblendw_q15(vintx src1a, vintx src1b, unsigned int src2);`
	
	`// double vector pseudo intrinsics`
	`dvcharx dvblend(dvcharx src1a, dvcharx src1b, dvcharx src2);`
	`dvshortx dvblend(dvshortx src1a, dvshortx src1b, dvshortx src2);`
	`dvintx dvblendw_q15(dvintx src1a, dvintx src1b, dvintx src2);`
	`dvcharx dvblend(dvcharx src1a, dvcharx src1b, unsigned int src2);`
	`dvshortx dvblend(dvshortx src1a, dvshortx src1b, unsigned int src2);`
	`dvintx dvblendw_q15(dvintx src1a, dvintx src1b, unsigned int src2);`

Additional details	Treat Vsrc1a lower 9/17/33 bits as X0, Vsrc1b lower 9/17/33 bits as X1, Vsrc2 lower 8/16/32 bits as unsigned alpha blending factor with Q7/Q15/Q31 fixed-point representation. B/H: Vdst = X0 + round(X1*alpha – X0*alpha, nbits); W: Vdst = (X0 << 15) + (X1*alpha >> 16) – (X0*alpha >> 16); nrbits = 7 for type B, 15 for type H (hard-wired, not as .R<nbits> option) Note that we do support scalar source 2 when source 2 is of the Word type, as opposed to VMAdd/VMSub not supporting scalar source 2. This is because this instruction supports only unsigned type for source 2, and indeed we can get unsigned 32-bit value from a scalar register.

9.8.7.11 VHBLEND_I

<table>
<tr><th>Instruction name</th><th>VHBLEND_I</th></tr>
<tr><td>Functionality</td><td>Vector blend horizontal interleaved</td></tr>
<tr><td>Assembly format</td><td>VHBlend_I<type> Vsrc1a, Vsrc1b, Vsrc2, Vdst</td></tr>
<tr><td>Type and bit width</td><td>B: 32 x (9-bit signed src1a/src1b, 8-bit unsigned src2 → 12-bit dst)
H: 16 x (17-bit signed src1a/src1b, 16-bit unsigned src2 → 24-bit dst)
W: 8 x (33-bit signed src1a/src1b, 32-bit unsigned src2 → 48-bit dst)
BHB: like B but with each lane pair sharing blending factor in Halfword lanes</td></tr>
<tr><td>Predication</td><td>not available</td></tr>
<tr><td>Source options</td><td>src1a: single vector register in VRF
src1b: single vector register in VRF
src2: single vector register in VRF</td></tr>
<tr><td>Destination options</td><td>dst: single vector register in VRF</td></tr>
<tr><td>Additional options</td><td></td></tr>
<tr><td>Intrinsics/operator</td><td><pre>
vcharx vhblend_i(vcharx src1a, vcharx src1b, vcharx src2);
vshortx vhblend_i(vshortx src1a, vshortx src1b, vshortx src2);
vintx vhblend_iw_q15(vintx src1a, vintx src1b, vintx src2);
vcharx vhblend_i(vcharx src1a, vcharx src1b, vshortx src2); // BHB

// double vector pseudo intrinsics
dvcharx dvhblend_i(dvcharx src1a, dvcharx src1b, dvcharx src2);
dvshortx dvhblend_i(dvshortx src1a, dvshortx src1b, dvshortx src2);
dvintx dvhblend_iw_q15(dvintx src1a, dvintx src1b, dvintx src2);
</pre></td></tr>
<tr><td>Additional details</td><td>Perform blending within each pair of lanes in src1a, src1b and interleave outcome.
In each even/odd pair of extended Byte/Halfword/Word lanes, extract 9/17/33 LSBs as signed src1a/src1b for X0/X1, extract 8/16/32 LSBs of src2 for as unsigned blending factor, according to this pattern for B/H/W types
<table>
<tr><td>src1a</td><td>A[0]</td><td>B[0]</td></tr>
<tr><td>src1b</td><td>A[1]</td><td>B[1]</td></tr>
<tr><td>src2</td><td>alpha[0]</td><td>alpha[1]</td></tr>
<tr><td>dst</td><td>Y[0]</td><td>Y[1]</td></tr>
</table>
For BHB type, both lanes share the same blending factor:
<table>
<tr><td>src1a</td><td>A[0]</td><td>B[0]</td></tr>
<tr><td>src1b</td><td>A[1]</td><td>B[1]</td></tr>
<tr><td>src2</td><td colspan="2">alpha[0] = alpha[1]</td></tr>
<tr><td>dst</td><td>Y[0]</td><td>Y[1]</td></tr>
</table>
The datapath carries out:
B/H/BHB: Y[i] = A[i] + round(B[i] * alpha[i] – A[i] *alpha[i], nbits);
i = {0, 1}, nrbits = 7 for type B/BHB, 15 for type H
W: Y[i] = (A[i] << 15) + (B[i]*alpha[i] >> 16) – (A[i]*alpha[i] >> 16);</td></tr>
</table>

Instruction name	VHBLEND_I
	Note that the BHB variation still support Byte-type blending factor (unsigned 8 bits in Q7 fixed-point), just that the lane position of blending factors are in even Byte lanes, as if it's a Halfword type vector.

VHBlend_I is intended to use with DVLUT_2x2pt to achieve bilinear interpolation with maximal throughput bottlenecked by the lookup.

As DVLUT_2x2pt for B/H/W type fetch up to 8/8/4 sets of 2x2 table entries, after interpolation the yield is half a single vector worth of outcome. To maximize throughput, at a minimum we would bundle up two DVLUT_2x2pt instructions and subsequent 3 blending instructions. Then we will need to unroll the loop to compensate for load-to-use latency.

The intended code loop is as follows for halfword (short) type:

```
for (...) {
  idx = dvshort_load_di(...);   // .lo: 0, 2, ...    .hi: 1, 3, ...
  x_frac = dvshort_load_perm(...); // .lo: 0, 0, 2, 2, ... .hi: 1, 1, 3, 3, ...
  y_frac = vshort_load(...);
  entries1 = dvlut_2x2pt_8h(table, idx.lo);
  entries2 = dvlut_2x2pt_8h(table, idx.hi);
  y_intrp1 = vhblend_i(entries1.lo, entries1.hi, x_frac.lo); // horz interpolation
  y_intrp2 = vhblend_i(entries2.lo, entries2.hi, x_frac.hi); // horz interpolation
  out = vhblend_i(y_intrp1, y_intrp2, y_frac);        // vert interpolation
  vstore(out);
}
```

Similarly for word (int) type:

```
for (...) {
  idx = dvint_load_di(...);   // .lo: 0, 2, ...    .hi: 1, 3, ...
  x_frac = dvint_load_perm(...); // .lo: 0, 0, 2, 2, ... .hi: 1, 1, 3, 3, ...
  y_frac = vint_load(...);
  entries1 = dvlut_2x2pt_4w(table, idx.lo);
  entries2 = dvlut_2x2pt_4w(table, idx.hi);
  y_intrp1 = vhblend_i(entries1.lo, entries1.hi, x_frac.lo); // horz interpolation
  y_intrp2 = vhblend_i(entries2.lo, entries2.hi, x_frac.hi); // horz interpolation
  out = vhblend_i(y_intrp1, y_intrp2, y_frac);        // vert interpolation
  vstore(out);
}
```

For byte (char) type, there is no load-permute feature, so we will have to use byte-to-halfword promoting load and the BHB type variation of VHBlend:

```
for (...) {
  idx = dvchar_load_di(...);            // .lo: 0, 2, ...       .hi: 1, 3, ...
  x_frac = vchar_dvshortx_load_di(...); // .lo: 0, -, 2, -, ... .hi: 1, -, 3, -, ...
  y_frac = vchar_load(...);
  entries1 = dvlut_2x2pt_8b(table, idx.lo);
```

```
  entries2 = dvlut_2x2pt_8b(table, idx.hi);
  y_intrp1 = vhblend_i(entries1.lo, entries1.hi, x_frac.lo); // horz interpolation
  y_intrp2 = vhblend_i(entries2.lo, entries2.hi, x_frac.hi); // horz interpolation
  out = vhblend_i(y_intrp1, y_intrp2, y_frac);               // vert interpolation
  vstore(out);
}
```

9.8.7.12 VMUL2

Instruction name	VMUL2
Functionality	Vector double multiply
Assembly format	VMul2<type>.R/T<bits> DVsrc1, Vsrc2/Rsrc2, DVdst/DACdst Rsrc2 option is available for B and H types only. WHW type requires .T16 .R0 is omitted
Type and bit width	B: 32 x (9-bit src1/src2 → 12-bit dst) H: 16 x (17-bit src1/src2 → 24-bit dst) WHW: 8 x (33-bit src1 x 17-bit src2 → 48-bit dst) only with .T16
Predication	not available
Source options	src1: double vector register in VRF src2: single vector register in VRF or single scalar register (B and H types)
Destination options	dst: double vector register in VRF or ARF
Additional options	

Intrinsics/operator	`dvcharx dvmulb(dvcharx src1, vcharx src2, u3imm rnd_opt);`
	`dvshortx dvmulh(dvshortx src1, vshortx src2, u3imm rnd_opt);`
	`dvintx  dvmulwhw_t16(dvintx  src1, vintx  src2);`
	`dvcharx dvmulb(dvcharx src1, int src2,   u3imm rnd_opt);`
	`dvshortx dvmulh(dvshortx src1, int src2,   u3imm rnd_opt);`

Additional details	Perform 2 sets of multiplication, sharing src2 dst.lo = round(src1.lo * src2, rnd_opt) dst.hi = round(src1.hi * src2, rnd_opt) See 9.8.7.1 for rounding/truncating options.

For example, VMu2lH.R7 V0:V1, V2, AC4:AC5 has the following data layout and behavior:

V0:	D[0]	D[1]	…	D[15]
V1:	E[0]	E[1]	…	E[15]
V2:	C[0]	C[1]	…	C[15]
AC4:	P[0]	P[1]	…	P[15]
AC5:	Q[0]	Q[1]	…	Q[15]

P[i] = round(C[i] * D[i], 7); // C[i], D[i], E[i] taken from 17 LSBs of each lane

Q[i] = round(C[i] * E[i], 7);

9.8.7.13 VFILT4_CA

This is an instruction added in Gen-2 VPU to accelerate filtering and CNN applications by 2x.

<table>
<tr><th>Instruction name</th><th>VFILT4_CA</th></tr>
<tr><td>Functionality</td><td>Vector 4-term filter</td></tr>
<tr><td>Assembly format</td><td><pred> VFilt4<type>_CA Vsrc1a, Vsrc1b, Wsrc2, DACsrc3dst

pred = none, [P2..P15]</td></tr>
<tr><td>Type and bit width</td><td>BBH: 32 x (9-bit src1a/src1b, 9-bit src2, 24-bit src3dst)
HHW: 16 x (17-bit src1a/src1b, 17-bit src2, 48-bit src3dst)</td></tr>
<tr><td>Predication</td><td>Available across lanes to clear accumulator</td></tr>
<tr><td>Source options</td><td>src1a: single vector register in VRF
src1b: single vector register in VRF
src2: single vector register in WRF</td></tr>
<tr><td>Destination options</td><td>src3dst: double vector register in ARF</td></tr>
<tr><td>Additional options</td><td></td></tr>
<tr><td>Intrinsics/operator</td><td><pre>dvshortx vfilt4_bbh(vcharx src1a, vcharx src1b, vcharx src2, dvshortx src3dst, int pred);

dvintx vfilt4_hhw(vshortx src1a, vshortx src1b, vshortx src2, dvintx src3dst, int pred);</pre></td></tr>
<tr><td>Additional details</td><td>When predicate is off, destination is replaced with the sum of 4 products,
effectively clearing the accumulator. Otherwise, the sum of 4 products is added to the accumulator.
Data entries for the products are formed with 4-tap filtering pattern, treating src1a and src1b as two data vectors offset by 4 elements. Coefficient entries are shared among 4 outputs. Accumulators are double vector registers to accommodate type promotion.
BBH data, coefficient, accumulator layout per 48-bit and
HHW data, coefficient, accumulator layout per 96-bit:

<table>
<tr><td>src1a</td><td>D[0]</td><td>D[1]</td><td>D[2]</td><td>D[3]</td></tr>
<tr><td>src1b</td><td>D[4]</td><td>D[5]</td><td>D[6]</td><td>D[7]</td></tr>
<tr><td>src2</td><td>C[0]</td><td>C[1]</td><td>C[2]</td><td>C[3]</td></tr>
<tr><td>src3dst.lo</td><td colspan="2">ACC[0]</td><td colspan="2">ACC[2]</td></tr>
<tr><td>src3dst.hi</td><td colspan="2">ACC[1]</td><td colspan="2">ACC[3]</td></tr>
</table>

ACC[0] += D[0] * C[0] + D[1] * C[1] + D[2] * C[2] + D[3] * C[3];
ACC[1] += D[1] * C[0] + D[2] * C[1] + D[3] * C[2] + D[4] * C[3];
ACC[2] += D[2] * C[0] + D[3] * C[1] + D[4] * C[2] + D[5] * C[3];
ACC[3] += D[3] * C[0] + D[4] * C[1] + D[5] * C[2] + D[6] * C[3];</td></tr>
</table>

Instruction name	VFILT4_CA
	See Data Ordering in Single and Double Vector Registers for data ordering in single/double vector registers.

9.8.7.14 VFILT4x2_CA

This is an instruction added in Gen-2 VPU to accelerate 8-bit/16-bit filtering and CNN applications by 4x. It's doing twice the amount of work compared to VFilt4_CA by accepting two sets of coefficients (src2) and accumulating onto two sets of accumulators.

Instruction name	VFILT4x2_CA
Functionality	Vector 4-term filter
Assembly format	<pred> VFilt4x2<type>_CA Vsrc1a, Vsrc1b, DWsrc2, QACsrc3dst pred = none, [P2..P15]
Type and bit width	BBH: 32 x (9-bit src1a/src1b, 9-bit src2, 24-bit src3dst) HHW: 16 x (17-bit src1a/src1b, 17-bit src2, 48-bit src3dst)
Predication	Available across lanes to clear accumulator
Source options	src1a: single vector register in VRF src1b: single vector register in VRF src2: double vector register in WRF
Destination options	src3dst: quad vector register in ARF
Additional options	

Intrinsics/operator	`void vfilt4x2_bbh(vcharx src1a, vcharx src1b, dvcharx src2, dvshortx src3 dvshortx src3_1, int pred, dvshortx & dst_0, dvshortx & dst_1);`
	
	`void vfilt4x2_hhw(vshortx src1a, vshortx src1b, dvshortx src2, dvintx src3_0, dvintx src3_1, int pred, dvintx & dst_0, dvintx & dst_1);`

Additional details	When predicate is off, destination is replaced with the sum of 4 products, effectively clearing the accumulator. Otherwise, the sum of 4 products is added to the accumulator. Data entries for the products are formed with horizontal 4-tap filtering pattern, treating src1a and src1b as two data vectors offset by 4 elements. Coefficient entries are shared among 4 outputs. There are two sets of coefficients and two sets of accumulators. Accumulators are quad vector registers to accommodate type promotion. BBH data, coefficient, accumulator layout per 48-bit and HHW data, coefficient, accumulator layout per 96-bit:

src1a	D[0]	D[1]	D[2]	D[3]
src1b	D[4]	D[5]	D[6]	D[7]
src2.lo	C[0][0]	C[0][1]	C[0][2]	C[0][3]

Instruction name	VFILT4x2_CA

src2.hi	C[1][0]	C[1][1]	C[1][2]	C[1][3]
src3dst.q0	ACC[0][0]		ACC[0][2]	
src3dst.q1	ACC[0][1]		ACC[0][3]	
src3dst.q2	ACC[1][0]		ACC[1][2]	
src3dst.q3	ACC[1][1]		ACC[1][3]	

```
ACC[0][0] += D[0] * C[0][0] + D[1] * C[0][1] + D[2] * C[0][2] + D[3] * C[0][3];
ACC[0][1] += D[1] * C[0][0] + D[2] * C[0][1] + D[3] * C[0][2] + D[4] * C[0][3];
ACC[0][2] += D[2] * C[0][0] + D[3] * C[0][1] + D[4] * C[0][2] + D[5] * C[0][3];
ACC[0][3] += D[3] * C[0][0] + D[4] * C[0][1] + D[5] * C[0][2] + D[6] * C[0][3];

ACC[1][0] += D[0] * C[1][0] + D[1] * C[1][1] + D[2] * C[1][2] + D[3] * C[1][3];
ACC[1][1] += D[1] * C[1][0] + D[2] * C[1][1] + D[3] * C[1][2] + D[4] * C[1][3];
ACC[1][2] += D[2] * C[1][0] + D[3] * C[1][1] + D[4] * C[1][2] + D[5] * C[1][3];
ACC[1][3] += D[3] * C[1][0] + D[4] * C[1][1] + D[5] * C[1][2] + D[6] * C[1][3];
```

9.8.7.15 VFILT4x2x2_CA

This is an instruction added in Gen-2 VPU to further accelerate 8-bit CNN applications by 2x (compared to VFilt4x2). Source 1a and 1b are double vectors each, and accumulator bit width is extended from 24-bit to 32-bit in VFilt4x2x2BBW_CA. This instruction delivers 4 horizontal taps x 2 deep/vertical taps x 2 sets of accumulators x 32 lanes = 512 INT8 MACs per instruction. Per VPU we have 1K INT8 MACs, and per PVA we have 2K INT8 MACs. This is 8X of Gen-1 PVA INT8 MAC performance.

Instruction name	VFILT4x2x2_CA
Functionality	Vector 4x2-term filter x 2 sets
Assembly format	<pred> VFilt4x2x2BBH_CA DVsrc1a, DVsrc1b, DWsrc2, QACsrc3dst <pred> VFilt4x2x2BBW_CA DVsrc1a, DVsrc1b, DWsrc2, QXACsrc3dst pred = none, [P2..P15]
Type and bit width	BBH: 32 x (9-bit src1a/src1b, 9-bit src2, 24-bit src3dst) BBW: 32 x (9-bit src1a/src1b, 9-bit src2, 32-bit src3dst)
Predication	Available across lanes to clear accumulator
Source options	src1a: double vector register in VRF src1b: double vector register in VRF src2: double vector register in WRF
Destination options	src3dst: BBH: quad vector register in ARF BBW: quad vector register in XARF

Instruction name	VFILT4x2x2_CA
Additional options	

Intrinsics/operator	`void vfilt4x2x2_bbh(dvcharx src1a, dvcharx src1b, dvcharx src2, dvshortx src3_0, dvshortx src3_1, int pred, dvshortx & dst_0, dvshortx & dst_1);`
	
	`void vfilt4x2x2_bbw(dvcharx src1a, dvcharx src1b, dvcharx src2, dxvshortx src3_0, dxvshortx src3_1, int pred, dxvshortx & dst_0, dxvshortx & dst_1);`

Additional details	When predicate is off, destination is replaced with the sum of 2x4 products, effectively clearing the accumulator. Otherwise, the sum of 2x4 products is added to the accumulator. Data entries for the products are formed with 4 (horizontal) x 2 (vertical or deep) tap filtering pattern, treating src1a and src1b as two sets of two data vectors offset by 8 elements. Coefficient entries are shared among 8 outputs in a slice. There are two sets of coefficients and two sets of accumulators. For BBH, each accumulator is 24-bit wide and mapped to quad vector in ARF. For BBW, each accumulator is 32-bit wide and mapped to quad vector in ARF as well as quad vector in XRF. Lower 24-bit comes from ARF and upper 8-bit comes from XRF.

Data layout per group of 8 byte lanes for VFilt4x2x2BBH & VFilt4x2x2BBW:

src1a.lo	D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
src1a.hi	D[8]	D[9]	D[10]	D[11]	D[12]	D[13]	D[14]	D[15]
src1b.lo	E[0]	E[1]	E[2]	E[3]	E[4]	E[5]	E[6]	E[7]
src1b.hi	E[8]	E[9]	E[10]	E[11]	E[12]	E[13]	E[14]	E[15]
src2.lo	C[0][0][0]	C[0][0][1]	C[0][0][2]	C[0][0][3]	C[0][1][0]	C[0][1][1]	C[0][1][2]	C[0][1][3]
src2.hi	C[1][0][0]	C[1][0][1]	C[1][0][2]	C[1][0][3]	C[1][1][0]	C[1][1][1]	C[1][1][2]	C[1][1][3]
src3dst.q0	ACC[0][0]		ACC[0][2]		ACC[0][4]		ACC[0][6]	
src3dst.q1	ACC[0][1]		ACC[0][3]		ACC[0][5]		ACC[0][7]	
src3dst.q2	ACC[1][0]		ACC[1][2]		ACC[1][4]		ACC[1][6]	
src3dst.q3	ACC[1][1]		ACC[1][3]		ACC[1][5]		ACC[1][7]	

For VFilt4x2x2BBW, XRF provides 8 upper bits for each accumulator:

src3dst.x0	ACC[0][0]	ACC[0][2]	ACC[0][4]	ACC[0][6]
src3dst.x1	ACC[0][1]	ACC[0][3]	ACC[0][5]	ACC[0][7]
src3dst.x2	ACC[1][0]	ACC[1][2]	ACC[1][4]	ACC[1][6]
src3dst.x3	ACC[1][1]	ACC[1][3]	ACC[1][1]	ACC[1][7]

ACC[0][0] += D[0] * C[0][0][0] + D[1] * C[0][0][1] + D[2] * C[0][0][2] + D[3] * C[0][0][3]
+ E[0] * C[0][1][0] + E[1] * C[0][1][1] + E[2] * C[0][1][2] + E[3] * C[0][1][3];
ACC[0][1] += D[1] * C[0][0][0] + D[2] * C[0][0][1] + D[3] * C[0][0][2] + D[4] * C[0][0][3]
+ E[1] * C[0][1][0] + E[2] * C[0][1][1] + E[3] * C[0][1][2] + E[4] * C[0][1][3];
ACC[0][2] += D[2] * C[0][0][0] + D[3] * C[0][0][1] + D[4] * C[0][0][2] + D[5] * C[0][0][3]

```
        + E[2] * C[0][1][0] + E[3] * C[0][1][1] + E[4] * C[0][1][2] + E[5] * C[0][1][3];
...
ACC[0][7] += D[7] * C[0][0][0] + D[8] * C[0][0][1] + D[9] * C[0][0][2] + D[10] * C[0][0][3]
        + E[7] * C[0][1][0] + E[8] * C[0][1][1] + E[9] * C[0][1][2] + E[10] * C[0][1][3];

ACC[1][0] += D[0] * C[1][0][0] + D[1] * C[1][0][1] + D[2] * C[1][0][2] + D[3] * C[1][0][3]
        + E[0] * C[1][1][0] + E[1] * C[1][1][1] + E[2] * C[1][1][2] + E[3] * C[1][1][3];

ACC[1][1] += D[1] * C[1][0][0] + D[2] * C[1][0][1] + D[3] * C[1][0][2] + D[4] * C[1][0][3]
        + E[1] * C[1][1][0] + E[2] * C[1][1][1] + E[3] * C[1][1][2] + E[4] * C[1][1][3];
ACC[1][2] += D[2] * C[1][0][0] + D[3] * C[1][0][1] + D[4] * C[1][0][2] + D[5] * C[1][0][3]
        + E[2] * C[1][1][0] + E[3] * C[1][1][1] + E[4] * C[1][1][2] + E[5] * C[1][1][3];
...
ACC[1][7] += D[7] * C[1][0][0] + D[8] * C[1][0][1] + D[9] * C[1][0][2] + D[10] * C[1][0][3]
        + E[7] * C[1][1][0] + E[8] * C[1][1][1] + E[9] * C[1][1][2] + E[10] * C[1][1][3];
```

9.8.7.16 VXNORADD8x4x2_CA

This is an instruction added in Gen-2 VPU to accelerate binary CNN convolution layers, by operating on 1-bit data/coefficients, and computing 8 horizontal taps x 4 deep taps x 2 sets of accumulators x 64 lanes = 4K XNor-accumulate per instruction. This is equivalent to 4K binary MACs (one XNor-Add translating to 1 binary Multiply-Accumulate). Per VPU we have 8K binary MACs, and per PVA we have 16K binary MACs. This is 4X of INT8 MAC performance.

Instruction name	**VXNorAdd8x4x2_CA**
Functionality	Vector exclusive NOR 8x4 filter x 2 sets
Assembly format	<pred> VXNorAdd8x4x2_CA DVsrc1a, DVsrc1b, Wsrc2, QXACsrc3dst pred = none, [P2..P15]
Type and bit width	Binary data/coefficients, extended charx (16-bit) accumulators
Predication	Available across lanes to clear accumulator
Source options	src1a: double vector register in VRF src1b: double vector register in VRF src2: single vector register in WRF Implicit PL scalar register
Destination options	src3dst:quad vector ARF + quad vector XRF
Additional options	
Intrinsics/operator	`void vxnor_add8x4x2(dvcharx src1a, dvcharx src1b, vcharx src2, dxvcharx src3_0, dxvcharx src3_1, int pred, unsigned int mask, dxvcharx & dst_0, dxvcharx & dst_1);`
Additional details	This instruction accelerates binary CNN 3D convolution. Per group of 8 byte lanes, this instruction delivers 1024 XNor-accumulate throughput per instruction via 8 horizontal taps (S) x 4 deep taps (C) x 2 sets x 16 lanes of accumulator of XNOR-add throughput. Each instruction delivers 8 x 4 x 2 x 64 = 4096 binary XNOR-accumulate throughput.

Instruction name	VXNorAdd8x4x2_CA
	Data is read from src1a and src1b (together 4 single registers supplying 4 rows of data). In each group of 8 byte lanes, each single vector source supplies 16 + 8 – 1 = 23 bits from the first 3 extended byte lanes. Coefficients are read from src2. For 2 sets of 8x4 binary coefficients, we need 2*8*4 = 64 bits, and they are read from 8 extended byte lanes of src2. Accumulators are read from and written back to src3dst, which is a quad extended ARF (XARF) register. In each slice, we need 2 sets x 16 horizontal lanes x 16-bit accumulator = 512 bits of accumulators, provided by 4 registers x 8 lanes x (12 + 4) bits from twice extended byte type = 512 bits of src3dst. A 32-bit mask is read from scalar register PL to enable/disable each XNor contribution to the accumulation. This is needed to trim the horizontal 8 taps and/or vertical 4 taps as needed to implement arbitrary weight tensor dimension through looping. For example, when S = 13 and C = 3, we would accumulate throughput looping, first 8 x 3 then 5 x 3 weight data, and would need to feed PL with correct mask values for these 2 sets of weight data. In a non-binary CNN, weights can be zero-padded as needed to trim the weight set. However, in binary CNN, we are matching activation binary with weight binary, and there is no room in the weight binary to encode a neutral weight value needed to trim down the weight tensor dimension. It's technically possible to use 2 bits per tap of weight to encode "don't care", but this would double the weight storage and traffic so is less efficient. The weight mask provides a mechanism to trim the weight dimension. Horizontally overlapped 8x4 data bits are XNORed with 2 sets of horizontally shared 8x4 coefficient bits, then ANDed with the 8x4 mask bits. When predicate is off, the destination is replaced with the masked sum of XNOR terms, effectively clearing the accumulator. Otherwise, the masked sum of XNOR terms is added to the accumulator.

Data layout per 96-bit:

Lowest byte lane						Highest byte lane		
src1a.lo	D[0][0..7]	D[0][8..15]	D[0][16..23]	D.C.	D.C.	D.C.	D.C.	D.C.
src1a.hi	D[1][0..7]	D[1][8..15]	D[2][16..23]	D.C.	D.C.	D.C.	D.C.	D.C.
src1b.lo	D[2][0..7]	D[2][8..15]	D[2][16..23]	D.C.	D.C.	D.C.	D.C.	D.C.
src1b.hi	D[3][0..7]	D[3][8..15]	D[3][16..23]	D.C.	D.C.	D.C.	D.C.	D.C.
src2	C[0][0][0..7]	C[0][1][0..7]	C[0][2][0..7]	C[0][3][0..7]	C[1][0][0..7]	C[1][1][0..7]	C[1][2][0..7]	C[1][3][0..7]
src3dst.q0	ACC[0][0]	ACC[0][2]	ACC[0][4]	ACC[0][6]	ACC[0][8]	ACC[0][10]	ACC[0][12]	ACC[0][14]
src3dst.q1	ACC[0][1]	ACC[0][3]	ACC[0][5]	ACC[0][7]	ACC[0][9]	ACC[0][11]	ACC[0][13]	ACC[0][15]
src3dst.q2	ACC[1][0]	ACC[1][2]	ACC[1][4]	ACC[1][6]	ACC[1][8]	ACC[1][10]	ACC[1][12]	ACC[1][14]
src3dst.q3	ACC[1][1]	ACC[1][3]	ACC[1][5]	ACC[1][7]	ACC[1][9]	ACC[1][11]	ACC[1][13]	ACC[1][15]

XRF and ARF together provides the 16-bit accumulator as src3 and destination; XRF provides 4 upper bits and ARF provides the lower 12 bits for each accumulator:

src3dst.x0	ACC[0][0]	ACC[0][2]	ACC[0][4]	ACC[0][6]	ACC[0][8]	ACC[0][10]	ACC[0][12]	ACC[0][14]
src3dst.x1	ACC[0][1]	ACC[0][3]	ACC[0][5]	ACC[0][7]	ACC[0][9]	ACC[0][11]	ACC[0][13]	ACC[0][15]
src3dst.x2	ACC[1][0]	ACC[1][2]	ACC[1][4]	ACC[1][6]	ACC[1][8]	ACC[1][10]	ACC[1][12]	ACC[1][14]
src3dst.x3	ACC[1][1]	ACC[1][3]	ACC[1][5]	ACC[1][7]	ACC[1][9]	ACC[1][11]	ACC[1][13]	ACC[1][15]

In the data layout diagram, each column represents a 12-bit extended byte lane. Each "D.C." entry represents a 12-bit don't care value. Each entry in the src1a/src1b/src2 rows having 8 bits of activation/weight data also includes 4 upper don't care bits.

Activation inputs involved are indexed as D[C][W], C being input depth and W being horizontal index. Coefficients (or weights) are indexed as C[K][C][S], K being output depth, C being input depth, and S being kernel horizontal index. Accumulators are indexed as ACC[K][Q], K being output depth and Q being output horizontal index. Mask bits are indexed as mask[C][S], C being input depth and S being kernel horizontal index.

The instructions carry out this nested for loop in each group of 8 byte lanes of vector math to **add to** the accumulators when predicate is true:

```
for (k = 0..1) // output depth (K)
  for (q=0..15) // output horizontal (Q)
   for (s=0..7) // kernel horizontal (S)
     ACC[k][q] += (mask[0][s] & ~(C[k][0][s] ^ D[0][q+s]))
          + (mask[1][s] & ~(C[k][1][s] ^ D[1][q+s]))
          + (mask[2][s] & ~(C[k][2][s] ^ D[2][q+s]))
          + (mask[3][s] & ~(C[k][3][s] ^ D[3][q+s]));
```

Otherwise (when predicate is false), we **write** bit counts of XNor between binary activation and weights to the accumulators, resulting in this behavior:

```
for (k = 0..1) // output depth (K)
  for (q=0..15) // output horizontal (Q)
   ACC[k][q] = 0;
   for (s=0..7) // kernel horizontal (S)
     ACC[k][q] += (mask[0][s] & ~(C[k][0][s] ^ D[0][q+s]))
          + (mask[1][s] & ~(C[k][1][s] ^ D[1][q+s]))
          + (mask[2][s] & ~(C[k][2][s] ^ D[2][q+s]))
          + (mask[3][s] & ~(C[k][3][s] ^ D[3][q+s]));
```

For intended binary CNN mapping, the 4 slices are supplied with activation data with 16 bits of offset between slices. It's 16 bits because each slice produces 2 planes x 16 horizontal outputs. We intend to use VLDPermHBU_P with permute indices {0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4} to permute bit-packed activation data in memory as Halfwords, then take the 16 permuted halfwords and zero-extend each 8-bit into 12-bit extended byte lane in each single vector register in src1a/src1b.

The same 2 * 8 * 4 = 64 bits of weight data is replicated among slices, so we can use VLDPermHBU_P with permute indices {0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3} to permute bit-packed weight data in memory as Halfwords, then take the 16 permuted

halfwords and zero-extend each 8-bit into 12-bit extended byte lane in the single vector register src2.

9.8.7.17 VDOTP4_CA

This is an instruction added in Gen-2 VPU to accelerate matrix multiplication by 2x.

One use case is bilinear interpolation. Another use case is fully connected convolution layer.

Instruction name	**VDotP4_CA**
Functionality	Vector 4-term dot product
Assembly format	<pred> VDotP4HHW/BBH_CA DVsrc1a, DVsrc1b, Wsrc2, DACsrc3dst <pred> VDotP4WHW_CA DVsrc1a, DVsrc1b, Wsrc2, ACsrc3dst <pred> VDotP4BBW_CA DVsrc1a, DVsrc1b, Wsrc2, DXACsrc3dst pred = none, [P2..P15]
Type and bit width	BBH: 32 x (9-bit src1a/src1b, 9-bit src2, 24-bit src3dst) HHW: 16 x (17-bit src1a/src1b, 17-bit src2, 48-bit src3dst) WHW: 8 x (33-bit src1a/src1b, 17-bit src2, 48-bit src3dst) BBW: 32 x (9-bit src1a/src1b, 9-bit src2, 32-bit src3dst)
Predication	Available across lanes to clear accumulator
Source options	src1a: double vector register in VRF src1b: double vector register in VRF src2:: single vector register in WRF
Destination options	src3dst: BBH/HHW: double vector register in ARF WHW: single vector register in ARF BBW: double vector register in XARF
Additional options	

Intrinsics/operator	`dvshortx vdotp4_bbh(dvcharx src1a, dvcharx src1b, vcharx src2, dvshortx src3dst, int pred);`
	
	`dvintx vdotp4_hhw(dvshortx src1a, dvshortx src1b, vshortx src2, dvintx src3dst, int pred);`
	
	`vintx vdotp4_whw(dvintx src1a, dvintx src1b, vshortx src2, vintx src3dst, int pred);`
	
	`dxvshortx vdotp4_bbw(dvcharx src1a, dvcharx src1b, vcharx src2, dxvshortx src3dst, int pred);`

Additional details	When predicate is off, destination is replaced with the sum of 4 products, effectively clearing the accumulator. Otherwise, the sum of 4 products is added to the accumulator. There are 4 independent data vectors. Coefficients are shared, within each group of 4 byte lanes for BBW, and within each group of 4 halfword lanes for HHW.

<table>
<tr><th>Instruction name</th><th>VDotP4_CA</th></tr>
<tr><td></td><td>
Accumulators for HHW are in a double vector register to accommodate type promotion. Accumulators for W are in a single vetor register.
BBH data, coefficient, accumulator layout per group of 4 byte lanes:
HHW data, coefficient, accumulator layout per group of 4 halfword lanes:

<table>
<tr><td>src1a.lo</td><td>D[0][0]</td><td>D[0][1]</td><td>D[0][2]</td><td>D[0][3]</td></tr>
<tr><td>src1a.hi</td><td>D[1][0]</td><td>D[1][1]</td><td>D[1][2]</td><td>D[1][3]</td></tr>
<tr><td>src1b.lo</td><td>D[2][0]</td><td>D[2][1]</td><td>D[2][2]</td><td>D[2][3]</td></tr>
<tr><td>src1b.hi</td><td>D[3][0]</td><td>D[3][1]</td><td>D[3][2]</td><td>D[3][3]</td></tr>
<tr><td>src2</td><td>C[0]</td><td>C[1]</td><td>C[2]</td><td>C[3]</td></tr>
<tr><td>src3dst.lo</td><td colspan="2">ACC[0]</td><td colspan="2">ACC[2]</td></tr>
<tr><td>src3dst.hi</td><td colspan="2">ACC[1]</td><td colspan="2">ACC[3]</td></tr>
</table>

ACC[0] += D[0][0] * C[0] + D[1][0] * C[1] + D[2][0] * C[2] + D[3][0] * C[3];
ACC[1] += D[0][1] * C[0] + D[1][1] * C[1] + D[2][1] * C[2] + D[3][1] * C[3];
ACC[2] += D[0][2] * C[0] + D[1][2] * C[1] + D[2][2] * C[2] + D[3][2] * C[3];
ACC[3] += D[0][3] * C[0] + D[1][3] * C[1] + D[2][3] * C[2] + D[3][3] * C[3];

For VDotP4BBW, XRF provides 8 upper bits for each accumulator:
<table>
<tr><td>src3dst.x0</td><td>ACC[0]</td><td>ACC[2]</td></tr>
<tr><td>src3dst.x1</td><td>ACC[1]</td><td>ACC[3]</td></tr>
</table>

WHW data, coefficient, accumulator layout per 96-bit:
<table>
<tr><td>src1a.lo</td><td colspan="2">D[0][0]</td><td colspan="2">D[0][1]</td></tr>
<tr><td>src1a.hi</td><td colspan="2">D[1][0]</td><td colspan="2">D[1][1]</td></tr>
<tr><td>src1b.lo</td><td colspan="2">D[2][0]</td><td colspan="2">D[2][1]</td></tr>
<tr><td>src1b.hi</td><td colspan="2">D[3][0]</td><td colspan="2">D[3][1]</td></tr>
<tr><td>src2</td><td>C[0]</td><td>C[1]</td><td>C[2]</td><td>C[3]</td></tr>
<tr><td>src3dst</td><td colspan="2">ACC[0]</td><td colspan="2">ACC[1]</td></tr>
</table>

ACC[0] += D[0][0] * C[0] + D[1][0] * C[1] + D[2][0] * C[2] + D[3][0] * C[3];
ACC[1] += D[0][1] * C[0] + D[1][1] * C[1] + D[2][1] * C[2] + D[3][1] * C[3];

See 6.2.3.6 for data ordering in single/double vector registers.
</td></tr>
</table>

9.8.7.18 VDOTP4x2_CA

This is an instruction added in Gen-2 VPU to accelerate 16-bit matrix multiplication by 4x. It's doing twice the amount of work compared to VDotP4_CA by accepting two sets of coefficients (src2) and accumulating onto two sets of accumulators.

<table>
<tr><th>Instruction name</th><th>VDotP4x2_CA</th></tr>
<tr><td>Functionality</td><td>Vector 4-term dot product</td></tr>
<tr><td>Assembly format</td><td><pred> VDotP4x2BBH/HHW_CA DVsrc1a, DVsrc1b, DWsrc2, QACsrc3dst
<pred> VDotP4x2BBW_CA DVsrc1a, DVsrc1b, DWsrc2, QXACsrc3dst

pred = none, [P2..P15]</td></tr>
<tr><td>Type and bit width</td><td>BBH: 32 x (9-bit src1a/src1b, 9-bit src2, 24-bit src3dst)
HHW: 16 x (17-bit src1a/src1b, 17-bit src2, 48-bit src3dst)
BBW: 32 x (9-bit src1a/src1b, 9-bit src2, 32-bit src3dst)</td></tr>
<tr><td>Predication</td><td>Available across lanes to clear accumulator</td></tr>
<tr><td>Source options</td><td>src1a: double vector register in VRF
src1b: double vector register in VRF
src2: double vector register in WRF</td></tr>
<tr><td>Destination options</td><td>src3dst: quad vector register in ARF (BBH/HHW)
quad vector register in XARF (BBW)</td></tr>
<tr><td>Additional options</td><td></td></tr>
<tr><td>Intrinsics/operator</td><td><pre>void vdotp4x2_bbh(dvcharx src1a, dvcharx src1b, dvcharx src2, dvshortx src3_0, dvshortx src3_1, int pred, dvshortx & dst_0, dvshortx & dst_1);

void vdotp4x2_hhw(dvshortx src1a, dvshortx src1b, dvshortx src2, dvintx src3_0, dvintx src3_1, int pred, dvintx & dst_0, dvintx & dst_1);

void vdotp4x2_bbw(dvcharx src1a, dvcharx src1b, dvcharx src2, dxvshortx src3_0, dxvshortx src3_1, int pred, dxvshortx &dst_0, dxvshortx &dst_1);</pre></td></tr>
<tr><td>Additional details</td><td>When predicate is off, destination is replaced with the sum of 4 products, effectively clearing the accumulator. Otherwise, the sum of 4 products is added to the accumulator.
There are 4 independent data vectors. Coefficients are shared, within each group of 4 byte lanes for BBW, and within each group of 4 halfword lanes for HHW. Accumulators are in a quad vector register to accommodate type promotion.
BBH data, coefficient, accumulator layout per 48-bit:
Also, HHW data, coefficient, accumulator layout per 96-bit:
<table>
<tr><td>src1a.lo</td><td>D[0][0]</td><td>D[0][1]</td><td>D[0][2]</td><td>D[0][3]</td></tr>
<tr><td>src1a.hi</td><td>D[1][0]</td><td>D[1][1]</td><td>D[1][2]</td><td>D[1][3]</td></tr>
<tr><td>src1b.lo</td><td>D[2][0]</td><td>D[2][1]</td><td>D[2][2]</td><td>D[2][3]</td></tr>
<tr><td>src1b.hi</td><td>D[3][0]</td><td>D[3][1]</td><td>D[3][2]</td><td>D[3][3]</td></tr>
<tr><td>src2.lo</td><td>C[0][0]</td><td>C[0][1]</td><td>C[0][2]</td><td>C[0][3]</td></tr>
<tr><td>src2.hi</td><td>C[1][0]</td><td>C[1][1]</td><td>C[1][2]</td><td>C[1][3]</td></tr>
</table></td></tr>
</table>

Instruction name	VDotP4x2_CA			
	src3dst.q0	ACC[0][0]	ACC[0][2]	
	src3dst.q1	ACC[0][1]	ACC[0][3]	
	src3dst.q2	ACC[1][0]	ACC[1][2]	
	src3dst.q3	ACC[1][1]	ACC[1][3]	

ACC[0][0] += D[0][0] * C[0][0] + D[1][0] * C[0][1] + D[2][0] * C[0][2] + D[3][0] * C[0][3];
ACC[0][1] += D[0][1] * C[0][0] + D[1][1] * C[0][1] + D[2][1] * C[0][2] + D[3][1] * C[0][3];
ACC[0][2] += D[0][2] * C[0][0] + D[1][2] * C[0][1] + D[2][2] * C[0][2] + D[3][2] * C[0][3];
ACC[0][3] += D[0][3] * C[0][0] + D[1][3] * C[0][1] + D[2][3] * C[0][2] + D[3][3] * C[0][3];

ACC[1][0] += D[0][0] * C[1][0] + D[1][0] * C[1][1] + D[2][0] * C[1][2] + D[3][0] * C[1][3];
ACC[1][1] += D[0][1] * C[1][0] + D[1][1] * C[1][1] + D[2][1] * C[1][2] + D[3][1] * C[1][3];
ACC[1][2] += D[0][2] * C[1][0] + D[1][2] * C[1][1] + D[2][2] * C[1][2] + D[3][2] * C[1][3];
ACC[1][3] += D[0][3] * C[1][0] + D[1][3] * C[1][1] + D[2][3] * C[1][2] + D[3][3] * C[1][3];

BBW data, coefficient layout per 48-bit is the same as that of BBH.

BBW accumulator is similar, with layout of lower 24-bit of each accumulator same as that of BBH, and upper 8-bit of each accumulator in the extension part of XARF:

src3dst.x0	ACC[0][0]	ACC[0][2]	ACC[0][4]	ACC[0][6]
src3dst.x1	ACC[0][1]	ACC[0][3]	ACC[0][5]	ACC[0][7]
src3dst.x2	ACC[1][0]	ACC[1][2]	ACC[1][4]	ACC[1][6]
src3dst.x3	ACC[1][1]	ACC[1][3]	ACC[1][5]	ACC[1][7]

9.8.7.19 VDOTP2x2_CA

Instruction name	VDOTP2x2_CA
Functionality	Vector 2-term dot product
Assembly format	<pred> VDotP2x2W_CA.T16 Vsrc1a, Vsrc1b, DWsrc2, DACsrc3dst pred = none, [P2..P15] .T16 is always applied with W type.
Type and bit width	W.T16: 8 x (33-bit src1a/src1b, 33-bit src2, 48-bit src3dst)
Predication	Available across lanes to clear accumulator
Source options	src1a: single vector register in VRF src1b: single vector register in VRF src2: double vector register in WRF
Destination options	src3dst: single vector register in ARF
Additional options	

<table>
<tr><th>Instruction name</th><th>VDOTP2x2_CA</th></tr>
<tr><td>Intrinsics/operator</td><td><code>dvintx vdotp2x2_w_t16(vintx src1a, vintx src1b, dvintx src2, dvintx src3d
int pred);</code></td></tr>
<tr><td>Additional details</td><td>W: Treat src1a and src1b as data vector, each being 8 x 31-bit in a 8 x 48-bit container. Treat src2 as coefficient vector, 2 x 2 x 31-bit in a 2 x 2 x 48-bit container replicated per group of 2 word lanes. Each product is truncated by 16 bits before being summed.
Data, coefficient, accumulator layout per 96-bit:
<table>
<tr><td>src1a</td><td>D[0][0]</td><td>D[0][1]</td></tr>
<tr><td>src1b</td><td>D[1][0]</td><td>D[1][1]</td></tr>
<tr><td>src2.lo</td><td>C[0][0]</td><td>C[0][1]</td></tr>
<tr><td>src2.hi</td><td>C[1][0]</td><td>C[1][1]</td></tr>
<tr><td>src3dst.lo</td><td>ACC[0][0]</td><td>ACC[0][1]</td></tr>
<tr><td>src3dst.hi</td><td>ACC[1][0]</td><td>ACC[1][1]</td></tr>
</table>

When predicate is true, do
ACC[0][0] += (D[0][0] * C[0][0] >> 16) + (D[1][0] * C[0][1] >> 16);
ACC[0][1] += (D[0][1] * C[0][0] >> 16) + (D[1][1] * C[0][1] >> 16);
ACC[1][0] += (D[0][0] * C[1][0] >> 16) + (D[1][0] * C[1][1] >> 16);
ACC[1][1] += (D[0][1] * C[1][0] >> 16) + (D[1][1] * C[1][1] >> 16);
Otherwise
ACC[0][0] = (D[0][0] * C[0][0] >> 16) + (D[1][0] * C[0][1] >> 16);
ACC[0][1] = (D[0][1] * C[0][0] >> 16) + (D[1][1] * C[0][1] >> 16);
ACC[1][0] = (D[0][0] * C[1][0] >> 16) + (D[1][0] * C[1][1] >> 16);
ACC[1][1] = (D[0][1] * C[1][0] >> 16) + (D[1][1] * C[1][1] >> 16);</td></tr>
</table>

9.8.7.20 VSUMSQ

Instruction name	VSUMSQ
Functionality	Vector sum of squares
Assembly format	VSumSq<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2, DVdst/DWdst type = BBH, HHW VSumSq<type> Vsrc1/Wsrc1, Vsrc2/Wsrc2, Vdst/Wdst type = W.T16
Type and bit width	BBH: 32 x (9-bit src1/src2 → 24-bit dst) HHW: 16 x (17-bit src1/src2 → 48-bit dst) W.T16: 8 x (33-bit src1/src2 → 48-bit dst)
Predication	not available
Source options	src1: single vector register in VRF/WRF src2: single vector register in VRF/WRF
Destination options	dst: double vector register in VRF/WRF for BBH/HHW single vector register in VRF/WRF for W.T16
Additional options	

Intrinsics/operator	`dvshortx vsumsq(vcharx src1, vcharx src2);`
	`dvintx  vsumsq(vshortx src1, vshortx src2);`
	`vintx  vsumsq_t16(vintx src1, vintx src2);`
	
	`// double vector pseudo intrinsics`
	`dvintx  dvsumsq_t16(dvintx src1, dvintx src2);`

Additional details	Perform sum of squares operation in each lane,
	dst = src1 * src1 + src2 * src2 // BBH/HHW
	dst = ((src1 * src1)>>16) + ((src2 * src2)>>16) // W.T16
	9/17/33 LSBs of src1 and src2 are used and interpreted as signed numbers.
	See 6.2.3.6 for data ordering in single/double vector registers.
	VSumSQ can be used in calculation Euclidean distance, sqrt(x^2 + y^2), or magnitude of a complex number, sqrt(real^2 + imaginary^2).

For example, VSumSqHHW V1, W2, V4:V5 has the following data layout and behavior:

V1:	X[0]	X[1]	X[2]	X[3]	…	X[14]	X[15]
W2:	Y[0]	Y[1]	Y[2]	Y[3]	…	Y[14]	Y[15]
V4:	S[0]		S[2]		…	S[14]	
V5:	S[1]		S[3]		…	S[15]	

S[i] = X[i] * X[i] + Y[i] * Y[i]; // X[i], Y[i] taken from 17 LSBs from each lane

9.8.7.21 VSQSUM

Instruction name	VSQSUM
Functionality	Vector square of sum
Assembly format	VSqSum<type> Vsrc1, Vsrc2, Vdst/DVdst
Type and bit width	BBH: 32 x (9-bit src1/src2 → 24-bit dst) HHW: 16 x (17-bit src1/src2 → 48-bit dst)
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF
Destination options	dst: BBH/HHW: double vector register in VRF
Additional options	
Intrinsics/operator	`dvshortx vsqsum(vcharx src1, vcharx src2);` `dvintx  vsqsum(vshortx src1, vshortx src2);`
Additional details	Perform square of sum operation in each lane. dst = (src1 + src2)^2 = src1^2 + src2^2 + 2 * src1 * src2 9/17 LSBs of src1 and src2 are used and interpreted as signed numbers. See 6.2.3.6 for data ordering in single/double vector registers. VSqSum can be used to calculate trace-square of a 2x2 matrix, trace being the sum of two diagonal terms. Trace-sqare is used in the Harris Corner feature point detection algorithm.

Note that VSqSumW.T16 was considered but deferred. For timing we would implement it as a^2 + b^2 + 2*a*b, but we would need to add another 32-bit multiplier per W lane to implement it.

For example, VSqSumBBH V1, V2, V4:V5 has the following data layout and behavior:

V1:	X[0]	X[1]	X[2]	X[3]	…	X[30]	X[31]
V2:	Y[0]	Y[1]	Y[2]	Y[3]	…	Y[30]	Y[31]
V4:	S[0]		S[2]		…	S[30]	
V5:	S[1]		S[3]		…	S[31]	

S[i] = (X[i] + Y[i]) * (X[i] + Y[i]); // X[i], Y[i] taken from 9 LSBs from each lane

9.8.7.22 VDET2x2

Instruction name	VDET2x2
Functionality	Vector 2x2 matrix determinant
Assembly format	VDet2x2<type> DVsrc1, DVsrc2, Vdst/DVdst VDet2x2<type> DVsrc1, DWsrc2, Vdst/DVdst VDet2x2<type> DWsrc1, DVsrc2, Vdst/DVdst
Type and bit width	HHW: 16 x (17-bit src1/src2 → 48-bit dst) W.T16: 8 x (33-bit src1/src2 → 48-bit dst)
Predication	Not available
Source options	src1: double vector register in VRF or WRF src2: double vector register in VRF or WRF
Destination options	dst: HHW: double vector register in VRF W.T16: single vector register in VRF
Additional options	
Intrinsics/operator	`dvintx vdet2x2_hhw(dvshortx src1, dvshortx src2);` `vintx vdet2x2_w_t16(dvintx src1, dvintx src2);`
Additional details	Treat two double vector sources as 4 entries in a 2x2 matrix. Src1.lo contains A00, src1.hi contains A01, src2.lo contains A10, src2.hi contains A11, in each lane. For HHW return A00*A11 – A01*A10 in each lane, extending precision into a double vector. For W.T16 return ((A00*A11)>>16) – ((A01*A10)>>16) in each lane, keeping word precision in a single vector. See 6.2.3.6 for data ordering in single/double vector registers.

For example, VDet2x2HHW V0:V1, V2:V3, V4:V5 has the following data layout and behavior:

V0:	X[0]	X[1]	X[2]	X[3]	…	X[14]	X[15]
V1:	Y[0]	Y[1]	Y[2]	Y[3]	…	Y[14]	Y[15]
V2:	Z[0]	Z[1]	Z[2]	Z[3]		Z[14]	Z[15]
V3:	W[0]	W[1]	W[2]	W[3]		W[14]	W[15]
V4:	S[0]		S[2]		…	S[14]	
V5:	S[1]		S[3]		…	S[15]	

$$S[i] = \det\left(\begin{bmatrix} X[i] & Y[i] \\ Z[i] & W[i] \end{bmatrix}\right) = X[i] * W[i] - Y[i] * Z[i]$$

9.8.8 Vector Floating-point Instructions

9.8.8.1 Instruction Summary

Floating-point add, subtract, multiply, multiply-add, and float-to-int, int-to-float conversion instructions are available in the vector math V0 and V1 instruction slots. The main vector register file VRF and working register file WRF supply the sources and destination of FP instructions.

FP multiply-add is implemented with a fused multiply-add datapath that preserves full product precision and has higher precision than separate FP multiply and FP add operations.

Invalid outcome is captured in the sticky invalid status bit, INV, as described in section 9.4.5.

Handling of NaN and various corner cases in vector FP math follows that of scalar FP math. FP comparison behavior of vector FP math follows that of scalar FP math. See FP Math Corner Cases, FP Comparison Corner Cases, and FP Conversion Corner Cases for corner case details.

Table 36. Vector floating-point instructions

Function	Assembly Format	Comments
Vector FP add	VAddF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	
Vector FP subtract	VSubF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	
Vector FP multiply	VMulF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	
Vector FP multiply-add	VMAddF Vsrc1, Wsrc2/Rsrc2, Vsrc3, Vdst <pred> VMAddF_CA Vsrc1, Vsrc2/Wsrc2/Rsrc2, Vsrc3dst	
Vector FP multiply-subtract	VMSubF Vsrc1, Wsrc2/Rsrc2, Vsrc3, Vdst	
Vector FP16 add	VAddHF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	
Vector FP16 subtract	VSubHF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	
Vector FP16 multiply	VMulHF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	
Vector FP16 multiply-add	VMAddHF Vsrc1, Wsrc2/Rsrc2, Vsrc3, Vdst <pred> VMAddHF_CA Vsrc1, Vsrc2/Wsrc2/Rsrc2, Vsrc3dst	
Vector FP16 multiply-subtract	VMSubHF Vsrc1, Wsrc2/Rsrc2, Vsrc3, Vdst	
Vector INT to FP conversion	VINT_FP Vsrc1, Vdst	

Function	Assembly Format	Comments
Vector FP to INT conversion with truncation	VFP_INT_Trunc Vsrc, Vdst	
Vector FP to INT conversion with rounding	VFP_INT_Round Vsrc, Vdst	
Vector INTX to FP conversion	VINTX_FP Vsrc1, Vdst	
Vector FP to INTX conversion with truncation	VFP_INTX_Trunc Vsrc, Vdst	
Vector FP to INTX conversion with rounding	VFP_INTX_Round Vsrc, Vdst	
Vector INT to FP16 conversion	VINT_FP16 DVsrc1, Rsrc2, Vdst	Rsrc2 conveys qbit for fixed-point representation.
Vector FP16 to INT conversion wth truncation	VFP16_INT_Trunc Vsrc1, Rsrc2, DVdst	Rsrc2 conveys qbit for fixed-point representation.
Vector FP16 to INT conversion wth rounding	VFP16_INT_Round Vsrc1, Rsrc2, DVdst	Rsrc2 conveys qbit for fixed-point representation.
Vector INT24 to FP16 conversion	VINT24_FP16 Vsrc1, Rsrc2, Vdst	Rsrc2 conveys qbit for fixed-point representation.
Vector FP16 to INT24 conversion wth truncation	VFP16_INT24_Trunc Vsrc1, Rsrc2, Vdst	Rsrc2 conveys qbit for fixed-point representation.
Vector FP16 to INT24 conversion wth rounding	VFP16_INT24_Round Vsrc1, Rsrc2, Vdst	Rsrc2 conveys qbit for fixed-point representation.
Vector FP16 to FP32 conversion	VFP16_FP Vsrc, DVdst	
Vector FP32 to FP16 conversion	VFP_FP16 DVsrc, Vdst	
Vector FP compare LT	VCmpLTF Vsrc1/Wsrc1,Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	
Vector FP compare LE	VCmpLEF Vsrc1/Wsrc1,Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	
Vector FP compare GT	VCmpGTF Vsrc1/Wsrc1,Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	
Vector FP compare GE	VCmpGEF Vsrc1/Wsrc1,Vsrc2/Wsrc2/Rsrc2,Vdst/Wdst	

Function	Assembly Format	Comments
Vector FP compare EQ	VCmpEQF Vsrc1/Wsrc1,Vsrc2/Wsrc2/Rsrc2,Vdst/Wdst	
Vector FP compare NE	VCmpNEF Vsrc1/Wsrc1,Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	
Vector FP16 compare LT	VCmpLTHF Vsrc1/Wsrc1,Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	
Vector FP16 compare LE	VCmpLEHF Vsrc1/Wsrc1,Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	
Vector FP16 compare GT	VCmpGTHF Vsrc1/Wsrc1,Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	
Vector FP16 compare GE	VCmpGEHF Vsrc1/Wsrc1,Vsrc2/Wsrc2/Rsrc2,Vdst/Wdst	
Vector FP16 compare EQ	VCmpEQHF Vsrc1/Wsrc1,Vsrc2/Wsrc2/Rsrc2,Vdst/Wdst	
Vector FP16 compare NE	VCmpNEHF Vsrc1/Wsrc1,Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst	
Vector FP reciprocal	VRCPF Vsrc/Wsrc, Vdst/Wdst	
Vector FP square root	VSQRTF Vsrc/Wsrc, Vdst/Wdst	
Vector FP reciprocal square root	VRSQF Vsrc/Wsrc, Vdst/Wdst	
Vector FP exponential base-2	VEXP2F Vsrc/Wsrc, Vdst/Wdst	
Vector FP logarithm base-2	VLOG2F Vsrc/Wsrc, Vdst/Wdst	
Vector FP sine	VSINF Vsrc/Wsrc, Vdst/Wdst	
Vector FP cosine	VCOSF Vsrc/Wsrc, Vdst/Wdst	
Vector FP hyperbolic tangent	VTANHF Vsrc/Wsrc, Vdst/Wdst	

9.8.8.2 VAddF

Instruction name	VAddF
Functionality	Floating-point add
Assembly format	VAddF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Scalar input: 32-bit scalar float broadcast to each lane Output: 8 x 48-bit (sign-extend FP32 to 48-bit)
Predication	not available
Source options	src1: vector register in VRF or WRF src2: vector register in VRF, WRF or scalar register
Destination options	vector register in VRF or WRF

Instruction name	VAddF
Additional options	

Intrinsics/operator	`vfloatx operator+( vfloatx src1, vfloatx src2);`
	`vfloatx operator+( vfloatx src1, float src2);`
	`vfloatx vaddf(vfloatx src1, vfloatx src2);`
	`vfloatx vaddf(vfloatx src1, float  src2);`
	
	`// Double vector pseudo intrinsics`
	`dvfloatx operator+( dvfloatx src1, dvfloatx src2);`
	`dvfloatx operator+( dvfloatx src1, float src2);`
	`dvfloatx dvaddf(dvfloatx src1, dvfloatx src2);`
	`dvfloatx dvaddf(dvfloatx src1, float  src2);`

Additional details	IEEE compliant floating-point add. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN. Outcome sign extended to fill bits 47 ~ 32.

9.8.8.3 VSubF

Instruction name	VSubF
Functionality	Floating-point subtract
Assembly format	VSubF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Scalar input: 32-bit scalar float broadcast to each lane Output: 8 x 48-bit (sign-extend FP32 to 48-bit
Predication	not available
Source options	src1: vector register in VRF or WRF src2: vector register in VRF, WRF or scalar register
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vfloatx operator-( vfloatx src1, vfloatx src2);`
	`vfloatx operator-( vfloatx src1, float src2);`
	`vfloatx vsubf(vfloatx src1, vfloatx src2);`
	`vfloatx vsubf(vfloatx src1, float  src2);`
	
	`// Double vector pseudo intrinsics`
	`dvfloatx operator-( dvfloatx src1, dvfloatx src2);`
	`dvfloatx operator-( dvfloatx src1, float src2);`
	`dvfloatx dvsubf(dvfloatx src1, dvfloatx src2);`
	`dvfloatx dvsubf(dvfloatx src1, float  src2);`

Additional details	IEEE compliant floating-point subtract. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even.

Instruction name	VSubF
	Set the invalid status flag when any input or output is NaN. Outcome sign extended to fill bits 47 ~ 32.

9.8.8.4 VMulF

Instruction name	VMulF
Functionality	Floating-point multiply
Assembly format	VMulF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Scalar input: 32-bit scalar float broadcast to each lane Output: 8 x 48-bit (sign-extend FP32 to 48-bit)
Predication	not available
Source options	src1: vector register in VRF or WRF src2: vector register in VRF, WRF or scalar register
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vfloatx operator*( vfloatx src1, vfloatx src2);`
	`vfloatx operator*( vfloatx src1, float src2);`
	`vfloatx vmulf(vfloatx src1, vfloatx src2);`
	`vfloatx vmulf(vfloatx src1, float  src2);`
	
	`// Double vector pseudo intrinsics`
	`dvfloatx operator*( dvfloatx src1, dvfloatx src2);`
	`dvfloatx operator*( dvfloatx src1, float src2);`
	`dvfloatx dvmulf(dvfloatx src1, dvfloatx src2);`
	`dvfloatx dvmulf(dvfloatx src1, float  src2);`

Additional details	IEEE compliant floating-point multiply. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN. Outcome sign extended to fill bits 47 ~ 32.

9.8.8.5 VMAddF

Instruction name	VMAddF
Functionality	Floating-point multiply-add
Assembly format	VMAddF Vsrc1, Wsrc2/Rsrc2, Vsrc3, Vdst <pred> VMAddF_CA Vsrc1, Vsrc2/Wsrc2/Rsrc2, Vsrc3dst pred = none, [P2..P15]
Type and bit width	Vector input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Scalar input: 32-bit scalar float broadcast to each lane Output: 8 x 48-bit (sign-extend FP32 to 48-bit)
Predication	Available across lanes to clear accumulator (CA variation)
Source options	unpredicated: src1/src3: vector register in VRF src2: vector register WRF or scalar register predicated (_CA): src1/src3: vector register in VRF src2: vector register VRF/WRF or scalar register
Destination options	vector register in VRF
Additional options	

Intrinsics/operator	`vfloatx vmaddf(vfloatx src1, vfloatx src2, vfloatx src3);`
	`vfloatx vmaddf(vfloatx src1, float src2, vfloatx src3);`
	`vfloatx vmaddf(vfloatx src1, vfloatx src2, vfloatx src3, int pred);`
	`vfloatx vmaddf(vfloatx src1, float src2, vfloatx src3, int pred);`
	
	`// Double vector pseudo intrinsics`
	`dvfloatx dvmaddf(dvfloatx src1, dvfloatx src2, dvfloatx src3);`
	`dvfloatx dvmaddf(dvfloatx src1, float src2, dvfloatx src3);`
	`dvfloatx dvmaddf(dvfloatx src1, dvfloatx src2, dvfloatx src3, int pred);`
	`dvfloatx dvmaddf(dvfloatx src1, float src2, dvfloatx src3, int pred);`

Additional details	Performing multiply-add with IEEE compliant floating-point multiply and add. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN. Outcome sign extended to fill bits 47 ~ 32. When predicate is true, perform multiply-add src1 * src2 + src3. Otherwise perform src1 * src2 + positive_zero, ignoring the src3 input. Note that for corner cases around positive/negative zeros, src1 * src2 and src1 * src2 + positive_zero produce different outcomes. Example: VMAddF V1, V2, V3, V4 would perform V4 = V1 * V2 + V3 in each Word lane.

9.8.8.6 VMSubF

Instruction name	VMSubF
Functionality	Floating-point multiply-subtract
Assembly format	VMSubF Vsrc1, Wsrc2/Rsrc2, Vsrc3, Vdst
Type and bit width	Vector input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Scalar input: 32-bit scalar float broadcast to each lane Output: 8 x 48-bit (sign-extend FP32 to 48-bit)
Predication	Not available
Source options	src1/src3: vector register in VRF src2: vector register in WRF or scalar register
Destination options	vector register in VRF
Additional options	

Intrinsics/operator	`vfloatx vmsubf(vfloatx src1, vfloatx src2, vfloatx src3);`
	`vfloatx vmsubf(vfloatx src1, float src2, vfloatx src3);`
	
	`// Double vector pseudo intrinsics`
	`dvfloatx dvmsubf(dvfloatx src1, dvfloatx src2, dvfloatx src3);`
	`dvfloatx dvmsubf(dvfloatx src1, float src2, dvfloatx src3);`

Additional details	Performing IEEE compliant floating-point multiply-sub, src3 – src1 * src2. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN. Outcome sign extended to fill bits 47 ~ 32. Example: VMSubF V1, W2, V3, V4 would perform V4 = V3 – V1 * W2 in each Word lane.

9.8.8.7 VAddHF

Instruction name	VAddHF
Functionality	FP16 add
Assembly format	VAddHF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector input: 16 x 16-bit float (16 LSBs of each 24-bit lane) Scalar input: 16-bit scalar float (16 LSBs) broadcast to each lane Output: 16 x 24-bit float (sign-extend FP16 to 24-bit)
Predication	not available
Source options	src1: vector register in VRF/WRF src2: vector register VRF/WRF or scalar register
Destination options	vector register in VRF/WRF
Additional options	
Intrinsics/operator	`vhfloatx operator+( vhfloatx src1, vhfloatx src2);`

Instruction name	VAddHF

	`vhfloatx operator+( vhfloatx src1, hfloat src2);`
	`vhfloatx vaddhf(vhfloatx src1, vhfloatx src2);`
	`vhfloatx vaddhf(vhfloatx src1, hfloat  src2);`
	
	`// Double vector pseudo intrinsics`
	`dvhfloatx operator+( dvhfloatx src1, dvhfloatx src2);`
	`dvhfloatx operator+( dvhfloatx src1, hfloat src2);`
	`dvhfloatx dvaddhf(dvhfloatx src1, dvhfloatx src2);`
	`dvhfloatx dvaddhf(dvhfloatx src1, hfloat  src2);`

Additional details	Least significant 16 bits of each Halfword lane in each source are read as FP16 numbers, FP16 addition performed, and FP16 outcome is sign-extended to 24-bit in each Halfword lane of the destination register. IEEE compliant half-precision floating-point add. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN.

9.8.8.8 VSubHF

Instruction name	VSubHF
Functionality	FP16 subtract
Assembly format	VSubHF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector input: 16 x 16-bit float (16 LSBs of each 24-bit lane) Scalar input: 16-bit scalar float (16 LSBs) broadcast to each lane Output: 16 x 24-bit float (sign-extend FP16 to 24-bit)
Predication	not available
Source options	src1: vector register in VRF/WRF src2: vector register VRF/WRF or scalar register
Destination options	vector register in VRF/WRF
Additional options	

Intrinsics/operator	`vhfloatx operator-( vhfloatx src1, vhfloatx src2);`
	`vhfloatx operator-( vhfloatx src1, hfloat src2);`
	`vhfloatx vsubhf(vhfloatx src1, vhfloatx src2);`
	`vhfloatx vsubhf(vhfloatx src1, hfloat  src2);`
	
	`// Double vector pseudo intrinsics`
	`dvhfloatx operator-( dvhfloatx src1, dvhfloatx src2);`
	`dvhfloatx operator-( dvhfloatx src1, hfloat src2);`
	`dvhfloatx dvsubhf(dvhfloatx src1, dvhfloatx src2);`
	`dvhfloatx dvsubhf(dvhfloatx src1, hfloat  src2);`

Additional details	Least significant 16 bits of each Halfword lane in each source are read as FP16 numbers, FP16 subtraction performed, and FP16 outcome is sign-extended to 24-bit in each Halfword lane of the destination register.

Instruction name	VSubHF
	IEEE compliant half-precision floating-point add. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN.

9.8.8.9 VMulHF

Instruction name	VMulHF
Functionality	FP16 multiply
Assembly format	VMulHF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector input: 16 x 16-bit float (16 LSBs of each 24-bit lane) Scalar input: 16-bit scalar float (16 LSBs) broadcast to each lane Output: 16 x 24-bit float (sign-extend FP16 to 24-bit)
Predication	not available
Source options	src1: vector register in VRF/WRF src2: vector register VRF/WRF or scalar register
Destination options	vector register in VRF/WRF
Additional options	

Intrinsics/operator	`vhfloatx operator*( vhfloatx src1, vhfloatx src2);`
	`vhfloatx operator*( vhfloatx src1, hfloat src2);`
	`vhfloatx vmulhf(vhfloatx src1, vhfloatx src2);`
	`vhfloatx vmulhf(vhfloatx src1, hfloat  src2);`
	
	`// Double vector pseudo intrinsics`
	`dvhfloatx operator*( dvhfloatx src1, dvhfloatx src2);`
	`dvhfloatx operator*( dvhfloatx src1, hfloat src2);`
	`dvhfloatx dvmulhf(dvhfloatx src1, dvhfloatx src2);`
	`dvhfloatx dvmulhf(dvhfloatx src1, hfloat  src2);`

Additional details	Least significant 16 bits of each Halfword lane in each source are read as FP16 numbers, FP16 multiplication performed, and FP16 outcome is sign-extended to 24-bit in each Halfword lane of the destination register. IEEE compliant half-precision floating-point multiply. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN.

9.8.8.10 VMAddHF

Instruction name	VMAddHF
Functionality	FP16 multiply-add
Assembly format	VMAddHF Vsrc1, Wsrc2/Rsrc2, Vsrc3, Vdst <pred> VMAddHF_CA Vsrc1, Vsrc2/Wsrc2/Rsrc2, Vsrc3dst pred = none, [P2..P15]
Type and bit width	Vector input: 16 x 16-bit float (16 LSBs of each 24-bit lane) Scalar input: 16-bit scalar float (16 LSBs) broadcast to each lane Output: 16 x 24-bit float (sign-extend FP16 to 24-bit)
Predication	Instruction level predication
Source options	unpredicated: src1/src3: vector register in VRF src2: vector register WRF or scalar register predicated (_CA): src1/src3: vector register in VRF src2: vector register VRF/WRF or scalar register
Destination options	vector register in VRF
Additional options	

Intrinsics/operator	`vhfloatx vmaddhf(vhfloatx src1, vhfloatx src2, vhfloatx src3);`
	`vhfloatx vmaddhf(vhfloatx src1, hfloat  src2, vhfloatx src3);`
	`vhfloatx vmaddhf(vhfloatx src1, vhfloatx src2, vhfloatx src3, int pred);`
	`vhfloatx vmaddhf(vhfloatx src1, hfloat src2, vhfloatx src3, int pred);`
	
	`// Double vector pseudo intrinsics`
	`dvhfloatx dvmaddhf(dvhfloatx src1, dvhfloatx src2, dvhfloatx src3);`
	`dvhfloatx dvmaddhf(dvhfloatx src1, hfloat  src2, dvhfloatx src3);`
	`dvhfloatx dvmaddhf(dvhfloatx src1, dvhfloatx src2, dvhfloatx src3, int pred);`
	`dvhfloatx dvmaddhf(dvhfloatx src1, hfloat src2, dvhfloatx src3, int pred);`

Additional details	Least significant 16 bits of each Halfword lane in each source are read as FP16 numbers, FP16 multiply-add performed, and FP16 outcome is sign-extended to 24-bit in each Halfword lane of the destination register. When predicate is true, perform multiply-add src1 * src2 + src3. Otherwise perform src1 * src2 + positive_zero, ignoring the src3 input. Fused multiply-add is performed, preserving intermediate precision as much as possible. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN.

9.8.8.11 VMSubHF

Instruction name	VMSubHF
Functionality	FP16 multiply-subtract
Assembly format	VMSubHF Vsrc1, Wsrc2/Rsrc2, Vsrc3, Vdst
Type and bit width	Vector input: 16 x 16-bit float (16 LSBs of each 24-bit lane) Scalar input: 16-bit scalar float (16 LSBs) broadcast to each lane Output: 16 x 24-bit float (sign-extend FP16 to 24-bit)
Predication	Not available
Source options	src1/src3: vector register in VRF src2: vector register in WRF or scalar register
Destination options	vector register in VRF
Additional options	

Intrinsics/operator	`vhfloatx vmsubhf(vhfloatx src1, vhfloatx src2, vhfloatx src3);`
	`vhfloatx vmsubhf(vhfloatx src1, hfloat src2, vhfloatx src3);`
	
	`// Double vector pseudo intrinsics`
	`dvhfloatx dvmsubhf(dvhfloatx src1, dvhfloatx src2, dvhfloatx src3);`
	`dvhfloatx dvmsubhf(dvhfloatx src1, hfloat  src2, dvhfloatx src3);`

Additional details	Least significant 16 bits of each Halfword lane in each source are read as FP16 numbers, FP16 multiply-subtract performed, and FP16 outcome is sign-extended to 24-bit in each Halfword lane of the destination register. Fused multiply-subtract is performed, preserving intermediate precision as much as possible. Handles denormal, zero, infinity, NaN. Generates quiet NaN. Only rounding mode supported is round to nearest, ties to even. Set the invalid status flag when any input or output is NaN.

9.8.8.12 VINT_FP

Instruction name	VINT_FP
Functionality	Integer to floating-point conversion
Assembly format	VINT_FP Vsrc, Vdst
Type and bit width	Input: 8 x 48-bit integer Output: 8 x 48-bit (sign-extend FP32 to 48-bit)
Predication	not available
Source options	vector register in VRF
Destination options	vector register in VRF
Additional options	

Intrinsics/operator	`vfloatx vint_vfp(vintx src);`
	
	`// Double vector pseudo intrinsics`
	`dvfloatx dvint_dvfp(dvintx src);`

Instruction name	VINT_FP
Additional details	Each Word-lane 48-bit integer input is first saturated to 32-bit integer range, [-2^31, 2^31-1], before converting to 32-bit floating-point. Each 32-bit floating-point outcome is sign-extended back into a 48-bit Word lane. The 32-bit integer to 32-bit floating-point conversion process is the same as in INT_FP scalar instruction. Note that rounding is included in this instruction's functionality. Only rounding mode supported is round to nearest, ties to even.

9.8.8.13 VFP_INT_Trunc

Instruction name	VFP_INT_Trunc
Functionality	Floating-point to integer conversion
Assembly format	VFP_INT_Trunc Vsrc, Vdst
Type and bit width	Input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Output: 8 x 48-bit integer
Predication	not available
Source options	vector register in VRF
Destination options	vector register in VRF
Additional options	

Intrinsics/operator	`vintx vfp_vint_trunc(vfloatx src);`
	
	`// Double vector pseudo intrinsics`
	`dvintx dvfp_dvint_trunc(dvfloatx src);`

Additional details	FP32 to integer conversion with truncation. For example, if input is 0x3FC0_0000 (1.5 in FP32), output is trunc(1.5) = 1 Note that - truncation is used during the conversion, consistent with C float-to-int type casting. - Both zero and minus zero maps to zero. - Infinity maps to maximal 32-bit int value (0x7FFF_FFFF). - Minus infinity maps to minimal 32-bit int value (0x8000_0000). - When output value exceeds 32-bit int representation range, output is saturated between 0x8000_0000 and x7FFF_FFFF. - NaN maps to either 0x8000_0000 or 0x7FFF_FFFF, preserving the sign. - The invalid status flag is NOT set when input is NaN.

9.8.8.14 VFP_INT_Round

Instruction name	VFP_INT_Round
Functionality	Floating-point to integer conversion
Assembly format	VFP_INT_Round Vsrc, Vdst
Type and bit width	Input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Output: 8 x 48-bit integer
Predication	not available
Source options	vector register in VRF
Destination options	vector register in VRF
Additional options	

Intrinsics/operator	`vintx vfp_vint_round(vfloatx src);`
	
	`// Double vector pseudo intrinsics`
	`dvintx dvfp_dvint_round(dvfloatx src);`

Additional details	FP32 to integer conversion with rounding. For example, if input is 0x3FC0_0000 (1.5 in FP32), output is round(1.5) = 2, as 1.5 is tied between 1 and 2, so we round to 2 (even). Note that - Rounding is used during the conversion. The only rounding mode supported is round to nearest, ties to even. - Both zero and minus zero maps to zero. - Infinity maps to maximal 32-bit int value (0x7FFF_FFFF). - Minus infinity maps to minimal 32-bit int value (0x8000_0000). - When output value exceeds 32-bit int representation range, output is saturated between 0x8000_0000 and x7FFF_FFFF. - NaN maps to either 0x8000_0000 or 0x7FFF_FFFF, preserving the sign. - The invalid status flag is NOT set when input is NaN. Gen-1 legacy intrinsic function f32_to_i32() is supported. As it implements rounding implicitly, programmers are strongly encouraged to switch to Gen-2 intrinsic function fp_int_round() to avoid confusion.

9.8.8.15 VINTX_FP

Instruction name	VINTX_FP
Functionality	Extended integer to floating-point conversion
Assembly format	VINTX_FP Vsrc, Vdst
Type and bit width	Input: 8 x 48-bit integer Output: 8 x 48-bit (sign-extend FP32 to 48-bit)
Predication	not available
Source options	vector register in VRF
Destination options	vector register in VRF
Additional options	

Intrinsics/operator	`vfloatx vintx_vfp(vintx src);`
	
	`// Double vector pseudo intrinsics`
	`dvfloatx dvintx_dvfp(dvintx src);`

Additional details	Each Word-lane 48-bit integer input is converted to 32-bit floating-point. Each 32-bit floating-point outcome is sign-extended back into a 48-bit Word lane. Note that rounding is included in this instruction's functionality. Only rounding mode supported is round to nearest, ties to even.

9.8.8.16 VFP_INTX_Trunc

Instruction name	VFP_INTX_Trunc
Functionality	Floating-point to extended integer conversion with truncation
Assembly format	VFP_INTX_Trunc Vsrc, Vdst
Type and bit width	Input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Output: 8 x 48-bit integer
Predication	not available
Source options	vector register in VRF
Destination options	vector register in VRF
Additional options	

Intrinsics/operator	`vintx vfp_vintx_trunc(vfloatx src);`
	
	`// Double vector pseudo intrinsics`
	`dvintx dvfp_dvintx_trunc(dvfloatx src);`

Additional details	FP32 to INT48 conversion with truncation. Note that - truncation is used during the conversion, consistent with C float-to-int type casting. - Both zero and minus zero maps to zero. - Infinity maps to maximal 48-bit int value (0x7FFF_FFFF_FFFF). - Minus infinity maps to minimal 48-bit int value (0x8000_0000_0000).

Instruction name	VFP_INTX_Trunc
	- When output value exceeds 48-bit int representation range, output is saturated between 0x8000_0000_0000 and x7FFF_FFFF_FFFF. - NaN maps to either 0x8000_0000_0000 or 0x7FFF_FFFF_FFFF, preserving the sign. - The invalid status flag is NOT set when input is NaN.

9.8.8.17 VFP_INTX_Round

Instruction name	VFP_INTX_Round
Functionality	Floating-point to extended integer conversion with rounding
Assembly format	VFP_INTX_Round Vsrc, Vdst
Type and bit width	Input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Output: 8 x 48-bit integer
Predication	not available
Source options	vector register in VRF
Destination options	vector register in VRF
Additional options	

Intrinsics/operator	`vintx vfp_vintx_round(vfloatx src);`
	
	`// Double vector pseudo intrinsics`
	`dvintx dvfp_dvintx_round(dvfloatx src);`

Additional details	FP32 to INT48 conversion with rounding. Note that - Rounding is used during the conversion. The only rounding mode supported is round to nearest, ties to even. - Both zero and minus zero maps to zero. - Infinity maps to maximal 48-bit int value (0x7FFF_FFFF_FFFF). - Minus infinity maps to minimal 48-bit int value (0x8000_0000_0000). - When output value exceeds 48-bit int representation range, output is saturated between 0x8000_0000_0000 and x7FFF_FFFF_FFFF. - NaN maps to either 0x8000_0000_0000 or 0x7FFF_FFFF_FFFF, preserving the sign. - The invalid status flag is NOT set when input is NaN.

9.8.8.18 VINT_FP16

Instruction name	VINT_FP16
Functionality	Integer to 16-bit floating-point conversion
Assembly format	VINT_FP16 DVsrc1, Rsrc2, Vdst
Type and bit width	Input: 2 x 8 x 48-bit integer Output: 16 x 24-bit (FP16 sign-extend to 24-bit)
Predication	not available
Source options	src1: double vector register in VRF src2: scalar register
Destination options	vector register in VRF
Additional options	
Intrinsics/operator	`vhfloatx dvint_vfp16(dvintx src1, int src2);`
Additional details	src2 (read as sign number and saturated to [0, 31]) conveys qbit in source fixed-point representation. dst = src1 / 2^src2. Each Word-lane 48-bit integer input is first saturated to 32-bit integer range, [-2^31, 2^31-1], before converting to 16-bit floating-point along with the qbit information. Each 16-bit floating-point outcome is sign-extended back into a 24-bit Halfword lane. The 32-bit integer to 16-bit floating-point conversion process is the same as in INT_FP16 scalar instruction. Note that rounding is included in this instruction's functionality. Only rounding mode supported is round to nearest, ties to even. Where output absolute value falls below normal FP16 range, denormal FP16 output is generated. Conversion inputs come interleaved from a double vector register.

9.8.8.19 VFP16_INT_Trunc

Instruction name	VFP16_INT_Trunc
Functionality	Floating-point to integer conversion with truncation
Assembly format	VFP16_INT_Round Vsrc1, Rsrc2, DVdst
Type and bit width	Input: 16 x 16-bit (16 LSBs of each 24-bit lane) Output: 2 x 8 x 48-bit integer
Predication	not available
Source options	src1: vector register in VRF src2: scalar register
Destination options	double vector register in VRF
Additional options	
Intrinsics/operator	`dvintx vfp16_dvint_trunc(vhfloatx src1, int src2);`
Additional details	src2 (read as sign number and saturated to [0, 31]) conveys qbit in destination fixed-point representation. dst = trunc(src1 * 2^src2). Each 16-bit floating-point input is read from 16 LSBs of a Halfword lane (24-bit container). Note that - truncation is used during the conversion. - Both zero and minus zero maps to zero. - Infinity maps to maximal 32-bit int value (0x7FFF_FFFF). - Minus infinity maps to minimal 32-bit int value (0x8000_0000). - When output value trunc(src1 * 2^src2) exceeds 32-bit int representation range, output is saturated between 0x8000_0000 and x7FFF_FFFF. - NaN maps to either 0x8000_0000 or 0x7FFF_FFFF, preserving the sign. - The invalid status flag is NOT set when input is NaN. - Denormal FP16 input value is supported. - Conversion outputs are deinterleaved into a double vector register.

9.8.8.20 VFP16_INT_Round

Instruction name	VFP16_INT_Round
Functionality	Floating-point to integer conversion with rounding
Assembly format	VFP16_INT_Round Vsrc1, Rsrc2, DVdst
Type and bit width	Input: 16 x 16-bit (16 LSBs of each 24-bit lane) Output: 2 x 8 x 48-bit integer
Predication	not available
Source options	src1: vector register in VRF src2: scalar register
Destination options	double vector register in VRF
Additional options	
Intrinsics/operator	`dvintx vfp16_dvint_round(vhfloatx src1, int src2);`
Additional details	src2 (read as sign number and saturated to [0, 31]) conveys qbit in destination fixed-point representation. dst = round(src1 * 2^src2). Each 16-bit floating-point input is read from 16 LSBs of a Halfword lane (24-bit container). Note that - Rounding is used during the conversion. The only rounding mode supported is round to nearest, ties to even. - Both zero and minus zero maps to zero. - Infinity maps to maximal 32-bit int value (0x7FFF_FFFF). - Minus infinity maps to minimal 32-bit int value (0x8000_0000). - When output value round(src1 * 2^src2) exceeds 32-bit int representation range, output is saturated between 0x8000_0000 and x7FFF_FFFF. - NaN maps to either 0x8000_0000 or 0x7FFF_FFFF, preserving the sign. - The invalid status flag is NOT set when input is NaN. - Denormal FP16 input value is supported. - Conversion outputs are deinterleaved into a double vector register.

9.8.8.21 VINT24_FP16

Instruction name	VINT24_FP16
Functionality	24-bit integer to 16-bit floating-point conversion
Assembly format	VINT24_FP16 Vsrc1, Rsrc2, Vdst
Type and bit width	Input: 16 x 24-bit integer Output: 16 x 24-bit (FP16 sign-extend to 24-bit)
Predication	not available
Source options	src1: vector register in VRF src2: scalar register
Destination options	vector register in VRF
Additional options	

Intrinsics/operator	`vhfloatx vint24_vfp16(vshortx src1, int src2);`
	
	`// Double vector pseudo intrinsics`
	`dvhfloatx dvint24_dvfp16(dvshortx src1, int src2);`

Additional details	src2 (read as sign number and saturated to [0, 23]) conveys qbit in source fixed-point representation. dst = src1 / 2^src2. Note that rounding is included in this instruction's functionality. Only rounding mode supported is round to nearest, ties to even. Each 16-bit floating-point output is sign-extended into a Halfword lane (24-bit container). Where output absolute value falls below normal FP16 range, denormal FP16 output is generated.

9.8.8.22 VFP16_INT24_Trunc

Instruction name	VFP16_INT24_Trunc
Functionality	Floating-point to integer conversion with truncation
Assembly format	VFP16_INT24_Trunc Vsrc1, Rsrc2, Vdst
Type and bit width	Input: 16 x 16-bit float (16 LSBs of each 24-bit lane) Output: 16 x 24-bit integer
Predication	not available
Source options	src1: vector register in VRF src2: scalar register
Destination options	vector register in VRF
Additional options	

Intrinsics/operator	`vshortx vfp16_vint24_trunc(vhfloatx src1, int src2);`
	
	`// Double vector pseudo intrinsics`
	`dvshortx dvfp16_dvint24_trunc(dvhfloatx src1, int src2);`

Additional details	src2 (read as sign number and saturated to [0, 23]) conveys qbit in destination fixed-point representation. dst = trunc(src1 * 2^src2). Each 16-bit floating-point input is read from 16 LSBs of a Halfword lane (24-bit container). Note that - truncation is used during the conversion. - Both zero and minus zero maps to zero. - Infinity maps to maximal 24-bit int value (0x7F_FFFF). - Minus infinity maps to minimal 24-bit int value (0x80_0000). - When output value trunc(src1 * 2^src2) exceeds 24-bit int representation range, output is saturated between 0x80_0000 and x7F_FFFF. - NaN maps to either 0x80_0000 or 0x7F_FFFF, preserving the sign. - Denormal FP16 input value is supported. - The invalid status flag is NOT set when input is NaN.

9.8.8.23 VFP16_INT24_Round

Instruction name	VFP16_INT24_Round
Functionality	Floating-point to integer conversion with rounding
Assembly format	VFP16_INT24_Round Vsrc1, Rsrc2, Vdst
Type and bit width	Input: 16 x 16-bit float (16 LSBs of each 24-bit lane) Output: 16 x 24-bit integer
Predication	not available
Source options	src1: vector register in VRF src2: scalar register
Destination options	vector register in VRF
Additional options	

Intrinsics/operator	`vshortx vfp16_vint24_round(vhfloatx src1, int src2);`
	
	`// Double vector pseudo intrinsics`
	`dvshortx dvfp16_dvint24_round(dvhfloatx src1, int src2);`

Additional details	src2 (read as sign number and saturated to [0, 23]) conveys qbit in destination fixed-point representation. dst = round(src1 * 2^src2). Each 16-bit floating-point input is read from 16 LSBs of a Halfword lane (24-bit container). Note that - Rounding is used during the conversion. The only rounding mode supported is round to nearest, ties to even. - Both zero and minus zero maps to zero. - Infinity maps to maximal 24-bit int value (0x7F_FFFF). - Minus infinity maps to minimal 24-bit int value (0x80_0000). - When output value round(src1 * 2^src2) exceeds 24-bit int representation range, output is saturated between 0x80_0000 and x7F_FFFF. - NaN maps to either 0x80_0000 or 0x7F_FFFF, preserving the sign. - The invalid status flag is NOT set when input is NaN. - Denormal FP16 input value is supported.

9.8.8.24 VFP16_FP

Instruction name	VFP16_FP
Functionality	Vector floating-point FP16 to floating-point FP32 conversion
Assembly format	VFP16_FP Vsrc, DVdst
Type and bit width	Input: 16 x 16-bit float (16 LSBs of each 24-bit lane) Output: 2 x 8 x 48-bit (FP32 sign-extend to 48-bit)
Predication	not available
Source options	vector register in VRF
Destination options	double vector register in VRF
Additional options	
Intrinsics/operator	`dvfloatx vfp16_dvfp(vhfloatx src);`
Additional details	FP16 floating-point input is read from 16 LSBs of each Halfword lane in the source, converted to FP32 floating-point outcome, sign-extended, and written to 48-bit Word lane in the destination. Note that the invalid status flag is NOT set when input is NaN. Conversion outputs are deinterleaved into a double vector register.

9.8.8.25 VFP_FP16

Instruction name	VFP_FP16
Functionality	Vector floating-point FP32 to floating-point FP16 conversion
Assembly format	VFP_FP16 DVsrc, Vdst
Type and bit width	Input: 2 x 8 x 32-bit (32 LSBs of each 48-bit lane) Output: 16 x 24-bit (FP16 sign-extend to 24-bit)
Predication	not available
Source options	double vector register in VRF
Destination options	vector register in VRF
Additional options	
Intrinsics/operator	`vhfloatx dvfp_vfp16(dvfloatx src);`
Additional details	FP32 floating-point input is read from 32 LSBs of each Word lane in the source, converted to FP16 floating-point outcome, sign-extended, and written to 24-bit Halfword lane in the destination. Note that the invalid status flag is NOT set when input is NaN. Conversion inputs come interleaved from a double vector register.

9.8.8.26 VCmpLTF

Instruction name	VCmpLTF
Functionality	Floating-point compare less than
Assembly format	VCmpLTF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst

Instruction name	VCmpLTF
Type and bit width	Vector input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Scalar input: 32-bit scalar float broadcast to each lane Output: 8 x 48-bit integer
Predication	not available
Source options	src1: vector register in VRF or WRF src2: vector register in VRF, WRF or scalar register
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx operator<( vfloatx src1, vfloatx src2);`
	`vintx operator<( vfloatx src1, float src2);`
	
	`// Double vector pseudo intrinsics`
	`dvintx operator<( dvfloatx src1, dvfloatx src2);`
	`dvintx operator<( dvfloatx src1, float src2);`

Additional details	Always return 0 or 1 and never set invalid status flag. See for FP comparison corner cases.

9.8.8.27 VCmpLEF

Instruction name	VCmpLEF
Functionality	Floating-point compare less than or equal to
Assembly format	VCmpLEF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Scalar input: 32-bit scalar float broadcast to each lane Output: 8 x 48-bit integer
Predication	not available
Source options	src1: vector register in VRF or WRF src2: vector register in VRF, WRF or scalar register
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx operator<=( vfloatx src1, vfloatx src2);`
	`vintx operator<=( vfloatx src1, float src2);`
	
	`// Double vector pseudo intrinsics`
	`dvintx operator<=( dvfloatx src1, dvfloatx src2);`
	`dvintx operator<=( dvfloatx src1, float src2);`

Additional details	Always return 0 or 1 and never set invalid status flag. See FP Comparison Corner Cases for FP comparison corner cases.

9.8.8.28 VCmpGTF

Instruction name	VCmpGTF
Functionality	Floating-point compare greater than
Assembly format	VCmpGTF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Scalar input: 32-bit scalar float broadcast to each lane Output: 8 x 48-bit integer
Predication	not available
Source options	src1: vector register in VRF or WRF src2: vector register in VRF, WRF or scalar register
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx operator>( vfloatx src1, vfloatx src2);`
	`vintx operator>( vfloatx src1, float src2);`
	
	`// Double vector pseudo intrinsics`
	`dvintx operator>( dvfloatx src1, dvfloatx src2);`
	`dvintx operator>( dvfloatx src1, float src2);`

Additional details	Always return 0 or 1 and never set invalid status flag. See FP Comparison Corner Cases for FP comparison corner cases.

9.8.8.29 VCmpGEF

Instruction name	VCmpGEF
Functionality	Floating-point compare greater than or equal to
Assembly format	VCmpGEF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Scalar input: 32-bit scalar float broadcast to each lane Output: 8 x 48-bit integer
Predication	not available
Source options	src1: vector register in VRF or WRF src2: vector register in VRF, WRF or scalar register
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx operator>=( vfloatx src1, vfloatx src2);`
	`vintx operator>=( vfloatx src1, float src2);`
	
	`// Double vector pseudo intrinsics`
	`dvintx operator>=( dvfloatx src1, dvfloatx src2);`
	`dvintx operator>=( dvfloatx src1, float src2);`

Additional details	Always return 0 or 1 and never set invalid status flag.

Instruction name	VCmpGEF
	See FP Comparison Corner Cases for FP comparison corner cases.

9.8.8.30 VCmpEQF

Instruction name	VCmpEQF
Functionality	Floating-point compare equal
Assembly format	VCmpEQF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Scalar input: 32-bit scalar float broadcast to each lane Output: 8 x 48-bit integer
Predication	not available
Source options	src1: vector register in VRF or WRF src2: vector register in VRF, WRF or scalar register
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vintx operator==( vfloatx src1, vfloatx src2);`
	`vintx operator==( vfloatx src1, float src2);`
	
	`// Double vector pseudo intrinsics`
	`dvintx operator==( dvfloatx src1, dvfloatx src2);`
	`dvintx operator==( dvfloatx src1, float src2);`

Additional details	Always return 0 or 1 and never set invalid status flag. See FP Comparison Corner Cases for FP comparison corner cases.

9.8.8.31 VCmpNEF

Instruction name	VCmpNEF
Functionality	Floating-point compare not equal
Assembly format	VCmpNEF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Scalar input: 32-bit scalar float broadcast to each lane Output: 8 x 48-bit integer
Predication	not available
Source options	src1: vector register in VRF or WRF src2: vector register in VRF, WRF or scalar register
Destination options	vector register in VRF or WRF
Additional options	
Intrinsics/operator	`vintx operator!=( vfloatx src1, vfloatx src2);` `vintx operator!=( vfloatx src1, float src2);`

Instruction name	VCmpNEF

	`// Double vector pseudo intrinsics`
	`dvintx operator!=( dvfloatx src1, dvfloatx src2);`
	`dvintx operator!=( dvfloatx src1, float src2);`

Additional details	Always return 0 or 1 and never set invalid status flag. See FP Comparison Corner Cases for FP comparison corner cases.

9.8.8.32 VCmpLTHF

Instruction name	VCmpLTHF
Functionality	FP16 compare less than
Assembly format	VCmpLTHF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector input: 16 x 16-bit float (16 LSBs of each 24-bit lane) Scalar input: 16-bit scalar float (16 LSBs) broadcast to each lane Output: 16 x 24-bit integer
Predication	not available
Source options	src1: vector register in VRF or WRF src2: vector register in VRF, WRF or scalar register
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vshortx operator<( vhfloatx src1, vhfloatx src2);`
	`vshortx operator<( vhfloatx src1, hfloat src2);`
	
	`// Double vector pseudo intrinsics`
	`dvshortx operator<( dvhfloatx src1, dvhfloatx src2);`
	`dvshortx operator<( dvhfloatx src1, hfloat src2);`

Additional details	Always return 0 or 1 and never set invalid status flag. See FP Comparison Corner Cases for FP comparison corner cases.

9.8.8.33 VCmpLEHF

Instruction name	VCmpLEHF
Functionality	FP16 compare less than or equal
Assembly format	VCmpLEHF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector input: 16 x 16-bit float (16 LSBs of each 24-bit lane) Scalar input: 16-bit scalar float (16 LSBs) broadcast to each lane Output: 16 x 24-bit integer
Predication	not available
Source options	src1: vector register in VRF or WRF src2: vector register in VRF, WRF or scalar register
Destination options	vector register in VRF or WRF

Instruction name	VCmpLEHF
Additional options	

Intrinsics/operator	`vshortx operator<=( vhfloatx src1, vhfloatx src2);`
	`vshortx operator<=( vhfloatx src1, hfloat src2);`
	
	`// Double vector pseudo intrinsics`
	`dvshortx operator<=( dvhfloatx src1, dvhfloatx src2);`
	`dvshortx operator<=( dvhfloatx src1, hfloat src2);`

Additional details	Always return 0 or 1 and never set invalid status flag. See FP Comparison Corner Cases for FP comparison corner cases.

9.8.8.34 VCmpGTHF

Instruction name	VCmpGTHF
Functionality	FP16 compare greater than
Assembly format	VCmpGTHF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector input: 16 x 16-bit float (16 LSBs of each 24-bit lane) Scalar input: 16-bit scalar float (16 LSBs) broadcast to each lane Output: 16 x 24-bit integer
Predication	not available
Source options	src1: vector register in VRF or WRF src2: vector register in VRF, WRF or scalar register
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vshortx operator>( vhfloatx src1, vhfloatx src2);`
	`vshortx operator>( vhfloatx src1, hfloat src2);`
	
	`// Double vector pseudo intrinsics`
	`dvshortx operator>( dvhfloatx src1, dvhfloatx src2);`
	`dvshortx operator>( dvhfloatx src1, hfloat src2);`

Additional details	Always return 0 or 1 and never set invalid status flag. See FP Comparison Corner Cases for FP comparison corner cases.

9.8.8.35 VCmpGEHF

Instruction name	VCmpGEHF
Functionality	FP16 compare greater than or equal
Assembly format	VCmpGEHF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector input: 16 x 16-bit float (16 LSBs of each 24-bit lane) Scalar input: 16-bit scalar float (16 LSBs) broadcast to each lane Output: 16 x 24-bit integer

Instruction name	VCmpGEHF
Predication	not available
Source options	src1: vector register in VRF or WRF src2: vector register in VRF, WRF or scalar register
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vshortx operator>=( vhfloatx src1, vhfloatx src2);`
	`vshortx operator>=( vhfloatx src1, hfloat src2);`
	
	`// Double vector pseudo intrinsics`
	`dvshortx operator>=( dvhfloatx src1, dvhfloatx src2);`
	`dvshortx operator>=( dvhfloatx src1, hfloat src2);`

Additional details	Always return 0 or 1 and never set invalid status flag. See FP Comparison Corner Cases for FP comparison corner cases.

9.8.8.36 VCmpEQHF

Instruction name	VCmpEQHF
Functionality	FP16 compare equal
Assembly format	VCmpEQHF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector input: 16 x 16-bit float (16 LSBs of each 24-bit lane) Scalar input: 16-bit scalar float (16 LSBs) broadcast to each lane Output: 16 x 24-bit integer
Predication	not available
Source options	src1: vector register in VRF or WRF src2: vector register in VRF, WRF or scalar register
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vshortx operator==( vhfloatx src1, vhfloatx src2);`
	`vshortx operator==( vhfloatx src1, hfloat src2);`
	
	`// Double vector pseudo intrinsics`
	`dvshortx operator==( dvhfloatx src1, dvhfloatx src2);`
	`dvshortx operator==( dvhfloatx src1, hfloat src2);`

Additional details	Always return 0 or 1 and never set invalid status flag. See FP Comparison Corner Cases for FP comparison corner cases.

9.8.8.37 VCmpNEHF

Instruction name	VCmpNEHF
Functionality	FP16 compare not equal
Assembly format	VCmpNEHF Vsrc1/Wsrc1, Vsrc2/Wsrc2/Rsrc2, Vdst/Wdst
Type and bit width	Vector input: 16 x 16-bit float (16 LSBs of each 24-bit lane) Scalar input: 16-bit scalar float (16 LSBs) broadcast to each lane Output: 16 x 24-bit integer
Predication	not available
Source options	src1: vector register in VRF or WRF src2: vector register in VRF, WRF or scalar register
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vshortx operator!=( vhfloatx src1, vhfloatx src2);`
	`vshortx operator!=( vhfloatx src1, hfloat src2);`
	
	`// Double vector pseudo intrinsics`
	`dvshortx operator!=( dvhfloatx src1, dvhfloatx src2);`
	`dvshortx operator!=( dvhfloatx src1, hfloat src2);`

Additional details	Always return 0 or 1 and never set invalid status flag. See FP Comparison Corner Cases for FP comparison corner cases.

9.8.8.38 VRCPF

Instruction name	VRCPF
Functionality	Floating-point reciprocal
Assembly format	VRCPF Vsrc/Wsrc, Vdst/Wdst
Type and bit width	Input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Output: 8 x 48-bit (sign-extend FP32 to 48-bit)
Predication	not available
Source options	vector register in VRF or WRF
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vfloatx vfrcp(vfloatx src);`
	
	`// Double vector pseudo intrinsics`
	`dvfloatx dvfrcp(dvfloatx src);`

Additional details	Performing FP32-input, FP32-output reciprocal. Set invalid status flag when output is NaN. Corner cases: RCP(+denorm) gives +Inf RCP(-denorm) gives -Inf

Instruction name	VRCPF
	RCP(+0.0) gives +Inf RCP(-0.0) gives -Inf RCP(+1.0) gives +1.0 RCP(-1.0) gives -1.0 RCP(+Inf) gives +0.0 RCP(-Inf) gives -0.0 RCP(NaN) gives NaN Max relative error is 2^-23 over entire normal floating-point range.

9.8.8.39 VSQRTF

Instruction name	VSQRTF
Functionality	Floating-point square root
Assembly format	VSQRTF Vsrc/Wsrc, Vdst/Wdst
Type and bit width	Input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Output: 8 x 48-bit (sign-extend FP32 to 48-bit)
Predication	not available
Source options	vector register in VRF or WRF
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vfloatx vfsqrt(vfloatx src);`
	
	`// Double vector pseudo intrinsics`
	`dvfloatx dvfsqrt(dvfloatx src);`

Additional details	Performing FP32-input, FP32-output square root. Set invalid status flag when output is NaN. Corner cases: SQRT(+denorm) gives +0.0 SQRT(-denorm) gives -0.0 SQRT(+0.0) gives +0.0 SQRT(-0.0) gives -0.0 SQRT(+1.0) gives +1.0 SQRT(-1.0) gives NaN SQRT(+Inf) gives +Inf SQRT(-Inf) gives NaN SQRT(NaN) gives NaN SQRT(negative) gives NaN (other than for -denorm or -0) Max relative error is 2^-23 over entire normal floating-point range.

9.8.8.40 VRSQF

Instruction name	VRSQF
Functionality	Floating-point reciprocal square root
Assembly format	VRSQF Vsrc/Wsrc, Vdst/Wdst
Type and bit width	Input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Output: 8 x 48-bit (sign-extend FP32 to 48-bit)
Predication	not available
Source options	vector register in VRF or WRF
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vfloatx vfrsq(vfloatx src);`
	
	`// Double vector pseudo intrinsics`
	`dvfloatx dvfrsq(dvfloatx src);`

Additional details	Performing FP32-input, FP32-output reciprocal square root. Set invalid status flag when output is NaN. Corner cases: RSQ(+denorm) gives +Inf RSQ(-denorm) gives -Inf RSQ(+0.0) gives +Inf RSQ(-0.0) gives -Inf RSQ(+1.0) gives +1.0 RSQ(-1.0) gives NaN RSQ(+Inf) gives +0.0 RSQ(-Inf) gives NaN RSQ(NaN) gives NaN RSQ(negative) gives NaN (other than for -denorm or -0) Max relative error is 2^-22.4 over entire normal floating-point range.

9.8.8.41 VEXP2F

Instruction name	VEXP2F
Functionality	Floating-point exponential base-2
Assembly format	VEXP2F Vsrc/Wsrc, Vdst/Wdst
Type and bit width	Input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Output: 8 x 48-bit (sign-extend FP32 to 48-bit)
Predication	not available
Source options	vector register in VRF or WRF
Destination options	vector register in VRF or WRF

Instruction name	VEXP2F
Additional options	

Intrinsics/operator	`vfloatx vfexp2(vfloatx src);`
	
	`// Double vector pseudo intrinsics`
	`dvfloatx dvfexp2(dvfloatx src);`

Additional details	Performing FP32-input, FP32-output exponential base-2 function. Set invalid status flag when output is NaN. Corner cases: EXP2(+denorm) gives +1.0 EXP2(-denorm) gives +1.0 EXP2(+0.0) gives +1.0 EXP2(-0.0) gives +1.0 EXP2(+Inf) gives +Inf EXP2(-Inf) gives +0.0 EXP2(NaN) gives NaN Max relative error is 2^-22.5 over entire normal floating-point range.

9.8.8.42 VLOG2F

Instruction name	VLOG2F
Functionality	Floating-point logarithm base-2
Assembly format	VLOG2F Vsrc/Wsrc, Vdst/Wdst
Type and bit width	Input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Output: 8 x 48-bit (sign-extend FP32 to 48-bit)
Predication	not available
Source options	vector register in VRF or WRF
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vfloatx vflog2(vfloatx src);`
	
	`// Double vector pseudo intrinsics`
	`dvfloatx dvflog2(dvfloatx src);`

Additional details	Performing FP32-input, FP32-output logarithm base-2 function. Set invalid status flag when output is NaN. Corner cases: LOG2(+denorm) gives -Inf LOG2(-denorm) gives -Inf LOG2(+0.0) gives -Inf LOG2(-0.0) gives -Inf LOG2(+Inf) gives +Inf

Instruction name	VLOG2F
	LOG2(-Inf) gives NaN LOG2(NaN) gives NaN LOG2(negative) gives NaN (other than for -denorm or -0) Max absolute error is 2^-22 in range (0.5, 2.0). Max relative error can be as large as 0.9 in range (0.5, 2.0). Max relative error is 2^-22.5 in range [0, 0.5] and [2.0, +Inf].

9.8.8.43 VSINF

Instruction name	VSINF
Functionality	Floating-point sine
Assembly format	VSINF Vsrc/Wsrc, Vdst/Wdst
Type and bit width	Input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Output: 8 x 48-bit (sign-extend FP32 to 48-bit)
Predication	not available
Source options	vector register in VRF or WRF
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vfloatx vfsin(vfloatx src);`
	
	`// Double vector pseudo intrinsics`
	`dvfloatx dvfsin(dvfloatx src);`

Additional details	Performing FP32-input, FP32-output sine function. Input in radians should be pre-normalized by multiplying 1.0/(2*pi). Input in degrees should be pre-normalized by multiplying 1.0/360. Set invalid status flag when output is NaN. Corner cases: SIN(+denorm) gives +0.0 SIN(-denorm) gives -0.0 SIN(+0.0) gives +0.0 SIN(-0.0) gives -0.0 SIN(+Inf) gives NaN SIN(-Inf) gives NaN SIN(NaN) gives NaN SIN(normal) is always in the range [-1, +1] Max absolute error is 2^-20.5 in range -2*pi ~ 2*pi. Max absolute error is 2^-14.7 in range -100*pi ~ 100*pi. The max error includes cumulative error of performing the required pre-normalization.

Instruction name	VSINF
	Outside of range -100*pi ~ 100*pi, only best effort is provided; there are no defined error guarantees.

9.8.8.44 VCOSF

Instruction name	VCOSF
Functionality	Floating-point cosine
Assembly format	VCOSF Vsrc/Wsrc, Vdst/Wdst
Type and bit width	Input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Output: 8 x 48-bit (sign-extend FP32 to 48-bit)
Predication	not available
Source options	vector register in VRF or WRF
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vfloatx vfcos(vfloatx src);`
	
	`// Double vector pseudo intrinsics`
	`dvfloatx dvfcos(dvfloatx src);`

Additional details	Performing FP32-input, FP32-output cosine function. Input in radians should be pre-normalized by multiplying 1.0/(2*pi). Input in degrees should be pre-normalized by multiplying 1.0/360. Set invalid status flag when output is NaN. Corner cases: COS(+denorm) gives +1.0 COS(-denorm) gives +1.0 COS(+0.0) gives +1.0 COS(-0.0) gives +1.0 COS(+Inf) gives NaN COS(-Inf) gives NaN COS(NaN) gives NaN COS(normal) is always in the range [-1, +1] Max absolute error is 2^-20.9 in range -2*pi ~ 2*pi. Max absolute error is 2^-15.3 in range -100*pi ~ 100*pi. The max error includes cumulative error of performing the required pre-normalization. Outside of range -100*pi ~ 100*pi, only best effort is provided; there are no defined error guarantees.

9.8.8.45 VTANHF

Instruction name	VTANHF
Functionality	Vector floating-point hyperbolic tangent
Assembly format	VTANHF Vsrc/Wsrc, Vdst/Wdst
Type and bit width	Input: 8 x 32-bit float (32 LSBs of each 48-bit lane) Output: 8 x 48-bit (sign-extend FP32 to 48-bit)
Predication	not available
Source options	vector register in VRF or WRF
Destination options	vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vfloatx vftanh(vfloatx src);`
	
	`// Double vector pseudo intrinsics`
	`dvfloatx dvftanh(dvfloatx src);`

Additional details	Performing FP32-input, FP32-output hyperbolic function. Set invalid status flag when output is NaN. Corner cases: TANH(-denorm) gives -0.0 TANH(-0.0) gives -0.0 TANH(+0.0) gives +0.0 TANH(+denorm) gives +0.0 TANH(-Inf) gives -1.0 TANH(+Inf) gives 1.0 TANH(NaN) gives NaN TANH(normal) is always in the range [-1.0 .. +1.0] Max relative error is 2^-11 over the entire normal floating-point range. Max absolute error is 2^-12 over the entire normal floating-point range.

9.8.9 Vector Misc Instructions

9.8.9.1 Instruction Summary

Table 37. Vector miscellaneous instructions

Function	Assembly Format	Comments
Vector min path cost	VMinPathCost<H/B> Vsrc1, Vsrc2, Vsrc3, Vdst	For SGM, semi-global matching algorithm
Vector Boolean map	VBMap31 Rsrc, Vsrc1, Vsrc2, Vsrc3, Vdst	Arbitrary 3-input-1-output Boolean operation, use Rsrc1 to encode the function
Vector 4x2 add/sub	VAddSub4x2_op<type> Vsrc1, Vsrc2, Vsrc3, Vsrc4, Vdst1, Vdst2	4-input-2-output, various add/sub operations
Vector configurable 4x2 add/sub	VCfgAddSub4x2<type> DVsrc1, DVsrc2, Rsrc3, DVdst VCfgAddSub4x2<type> DVsrc1, DWsrc2, Rsrc3, DVdst VCfgAddSub4x2<type> DWsrc1, DVsrc2, Rsrc3, DVdst	4-input-2-output, configurable add/sub operations
Vector normalize and extract index and fraction	VNormIdxFrac<type> Vsrc1, Vsrc2, Vdst1, Vdst2	Normalize src1 into index and fraction fields
Vector horizontal min4 accumulate	<pred> VHMin4<type>_CA DVsrc1, Wsrc2, ACsrc3dst	Min across (up to) 4 data terms and accumulator
Vector horizontal max4 accumulate	<pred> VHMax4<type>_CA DVsrc1, Wsrc2, ACsrc3dst	Max across (up to) 4 data terms and accumulator

9.8.9.2 VMIN_PATH_COST

Instruction name	VMIN_PATH_COST
Functionality	Vector min path cost
Assembly format	VMinPathCost1<type> Vsrc1, Vsrc2, Vsrc3, Vdst VMinPathCost2<type> Vsrc1, Vsrc2, Vsrc3, Vdst
Type and bit width	B: 32 x 12-bit signed Vsrc1/Vsrc2/Vsrc3, two 12-bit unsigned scalars packed in PL or PH H: 16 x 24-bit signed Vsrc1/Vsrc2/Vsrc3, 15-bit and 17-bit unsigned scalars packed in PL or PH
Predication	not available
Source options	src1, src2, src3: singe vector register Implicit source PL (R12, VMinPathCost1) or PH (R13, VMinPathCost2)
Destination options	dst: single vector register
Additional options	

Intrinsics/operator	`vshortx vminpathcost1(vshortx src1, vshortx src2, vshortx src3, int src4); // src4 in PL`
	
	`vshortx vminpathcost2(vshortx src1, vshortx src2, vshortx src3, int src4); // src4 in PH`
	
	`vcharx vminpathcost1(vcharx src1, vcharx src2, vcharx src3, int src4); // src4 in PL`
	
	`vcharx vminpathcost2(vcharx src1, vcharx src2, vcharx src3, int src4); // src4 in PH`
	
	`vshortx vminpathcost(vshortx src1, vshortx src2, vshortx src3, int src4); // same functionality as vminpathcost1`
	
	`vcharx vminpathcost(vcharx src1, vcharx src2, vcharx src3, int src4); // same functionality as vminpathcost1`

Additional details	Perform SGM min path cost calculation, which involves neighboring lanes. Each lane i of output involves itself, previous (i-1) and next (i+1) lanes: dst[i] = min(cost[i], cost[i-1]+p, cost[i+1]+p, q). Implicit (in assembly, not in intrinsic calls) scalar register PL = R12 or PH = R13 supplies p and q. VMinPathCost1 uses PL, and VMinPathCost2 uses PH. These 2 variants are mapped to the “1” and “2” variants in the intrinsic functions. For Byte type, p = src4[27:16] (unsigned 12-bit) and q = src4[11:0] (unsigned 12-bit). For Half-word type, p = src4[31:17] (unsigned 15-bit) and q = src4[16:0] (unsigned 17-bit). Treat Vsrc1, Vsrc2, Vsrc3 as 3 neighboring sections of a cost array, Vsrc2 supplying the current section, Vsrc1 the previous section, and Vsrc3 the next section. For Byte type, a single vector register contains 32 12-bit lanes. For lanes 1..30 of output, previous/current/next lanes are all available in Vsrc2. Lane 0 output shall use Vsrc1[31] to supply the previous lane, and lane 31 output shall use Vsrc3[0] to supply the next lane. For Half-word type, a single vector register contains 16 24-bit lanes. For lanes 1..14 of output, previous/current/next lanes are all available in Vsrc2. Lane 0 output shall use Vsrc1[15] to supply the previous lane, and lane 15 output shall use Vsrc3[0] to supply the next lane.

9.8.9.3 VBMap31

Instruction name	VBMap31
Functionality	Vector Boolean map
Assembly format	VBMap31 Rsrc, Vsrc1, Vsrc2, Vsrc3, Vdst
Type and bit width	none (bit-wise)
Predication	not available
Source options	Rsrc: scalar register Vsrc1, Vsrc2, Vsrc3: single vector register
Destination options	Vdst: single vector register
Additional options	

Intrinsics/operator	`vintx  vbmap31(int src, vintx src1, vintx src2, vintx src3);`
	`vshortx vbmap31(int src, vshortx src1, vshortx src2, vshortx src3);`
	`vcharx vbmap31(int src, vcharx src1, vcharx src2, vcharx src3);`
	
	`// Double vector pseudo intrinsics`
	`dvintx  dvbmap31(int src, dvintx src1, dvintx src2, dvintx src3);`
	`dvshortx dvbmap31(int src, dvshortx src1, dvshortx src2, dvshortx src3);`
	`dvcharx dvbmap31(int src, dvcharx src1, dvcharx src2, dvcharx src3);`

Additional details	Perform an arbitrary 3-input-1-output Boolean function using bits 7..0 of the scalar register source. These 8 bits are read as a truth table, indicating 0/1 outcome for the 8 combinations of 3 inputs, Vsrc1 contributing to bit 2 of the bit position, Vsrc2 contributing to bit 1, Vsrc3 contributing to bit 0. This is a bitwise operation across all 384 bits. For example, to implement the following Boolean function, Vsrc1 Vsrc2 Vsrc3 Output 0 0 0 1 0 0 1 1 0 1 0 0 0 1 1 0 1 0 0 1 1 0 1 0 1 1 0 0 1 1 1 1 bits 7..0 of Rsrc should contain 0x93.

9.8.9.4 VAddSub4x2

Instruction name	VAddSub4x2
Functionality	Vector 4x2 add/sub
Assembly format	VAddSub4x2_op<type> Vsrc1, Vsrc2, Vsrc3, Vsrc4, Vdst1, Vdst2
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit, B: 32 x 12-bit
Predication	not available
Source options	Vsrc1, Vsrc2, Vsrc3, Vsrc4: single vector register in VRF Only selected combination of VRF entries are allowed.
Destination options	Vdst1, Vdst2: single vector register in VRF
Additional options	op = 0 or 1, implementing one of two patterns

Intrinsics/operator	`void vaddsub4x2_0(vintx src1, vintx src2, vintx src3, vintx src4, vintx &dst1, vintx &dst2);`
	
	`void vaddsub4x2_1(vintx src1, vintx src2, vintx src3, vintx src4, vintx &dst1, vintx &dst2);`
	
	`void vaddsub4x2_0(vshortx src1, vshortx src2, vshortx src3, vshortx src4, vshortx &dst1, vshortx &dst2);`
	
	`void vaddsub4x2_1(vshortx src1, vshortx src2, vshortx src3, vshortx src4, vshortx &dst1, vshortx &dst2);`
	
	`void vaddsub4x2_0(vcharx src1, vcharx src2, vcharx src3, vcharx src4, vcharx &dst1, vcharx &dst2);`
	
	`void vaddsub4x2_1(vcharx src1, vcharx src2, vcharx src3, vcharx src4, vcharx &dst1, vcharx &dst2);`

Additional details	When op = 0, perform dst1 = src1 + src2 + src3 + src4 dst2 = src1 – src2 + src3 – src4 When op = 1, perform dst1 = src1 + src2 – src3 – src4 dst2 = src1 – src2 – src3 + src4

The VAddSub4x2 instruction is architected to accelerate FFT as well as Hadamard transform. Number of input/output operands makes it infeasible to allow arbitrary combination of operands, so the instruction is encoded so that only specific combinations of VRF entries are allowed:

For radix-4 DIF (decimation in frequency) FFT, the add/sub network carries out

z0.r = x0.r + x1.r + x2.r + x3.r
z2.r = x0.r - x1.r + x2.r - x3.r } op = 0

z0.i = x0.i + x1.i + x2.i + x3.i
z2.i = x0.i - x1.i + x2.i - x3.i } op = 0

z1.r = x0.r + x1.i - x2.r - x3.i } op = 1

z3.r = x0.r - x1.i - x2.r + x3.i
z1.i = x0.i - x1.r - x2.i + x3.r } op = 1
z3.i = x0.i + x1.r - x2.i - x3.r

Multiple sets of register assignment are supported to allow loop unrolling:

	x0.r	x0.i	x1.r	x1.i	x2.r	x2.i	x3.r	x3.i
RA 0	V0	V1	V2	V3	V4	V5	V6	V7
RA 1	V8	V9	V10	V11	V12	V13	V14	V15
RA 2	V16	V17	V18	V19	V20	V21	V22	V23
RA 3	V24	V25	V26	V27	V28	V29	V30	V31

There are 16 combinations of VRF input operands needed:

Combo	for	op	src1	src2	src3	src4
0	RA 0 z0.r, z2.r	0	V0	V2	V4	V6
1	RA 0 z0.i, z2.i	0	V1	V3	V5	V7
2	RA 0 z1.r, z3.r	1	V0	V3	V4	V7
3	RA 0 z3.i, z1.i	1	V1	V2	V5	V6
4	RA 1 z0.r, z2.r	0	V8	V10	V12	V14
5	RA 1 z0.i, z2.i	0	V9	V11	V13	V15
6	RA 1 z1.r, z3.r	1	V8	V11	V12	V15
7	RA 1 z3.i, z1.i	1	V9	V10	V13	V14
8	RA 2 z0.r, z2.r	0	V16	V18	V20	V22
9	RA 2 z0.i, z2.i	0	V17	V19	V21	V23
10	RA 2 z1.r, z3.r	1	V16	V19	V20	V23
11	RA 2 z3.i, z1.i	1	V17	V18	V21	V22
12	RA 3 z0.r, z2.r	0	V24	V26	V28	V30
13	RA 3 z0.i, z2.i	0	V25	V27	V29	V31
14	RA 3 z1.r, z3.r	1	V24	V27	V28	V31
15	RA 3 z3.i, z1.i	1	V25	V26	V29	V30

9.8.9.5 VCfgAddSub4x2

Instruction name	VCfgAddSub4x2
Functionality	Vector configurable 4x2 add/sub
Assembly format	VCfgAddSub4x2<type> DVsrc1, DVsrc2, Rsrc3, DVdst VCfgAddSub4x2<type> DVsrc1, DWsrc2, Rsrc3, DVdst VCfgAddSub4x2<type> DWsrc1, DVsrc2, Rsrc3, DVdst
Type and bit width	B: 32 x 12-bit, H: 16 x 24-bit, W: 8 x 48-bit
Predication	not available
Source options	DVsrc1, DVsrc2: double vector register in VRF or WRF Rsrc3: scalar register
Destination options	DVdst: double vector register in VRF
Additional options	

Intrinsics/operator	`dvcharx vcfg_addsub4x2(dvcharx src1, dvcharx src2, int src3);`
	`dvshortx vcfg_addsub4x2(dvshortx src1, dvshortx src2, int src3);`
	`dvintx vcfg_addsub4x2(dvintx src1, dvintx src2, int src3);`

Additional details	Decode configuration from scalar Rsrc3 by extracting 8 2-bit parameters: m11 = Rsrc3[1:0], m12 = Rsrc3[3:2], m13 = Rsrc3[5:4], m14 = Rsrc3[7:6], m21 = Rsrc3[9:8], m22 = Rsrc3[11:10], m23 = Rsrc3[13:12], m24 = Rsrc3[15:14]. Each parameter is interpreted as “00”: 0 “01”: 1 “10”: -1 “11”: -1 Compute dst.lo = m11*src1.lo + m12*src1.hi + m13*src2.lo + m14*src2.hi dst.hi = m21*src1.lo + m22*src1.hi + m23*src2.lo + m24*src2.hi Note that .lo and .hi components are derived from double vector operands as described in 6.2.3.6, according to the interleaved format.

9.8.9.6 VNormIdxFrac

Instruction name	VNormIdxFrac
Functionality	Vector normalize and extract index/fraction
Assembly format	VNormIdxFrac<type> Vsrc1, Vsrc2, Vdst1, Vdst2
Type and bit width	W: 8 x 48-bit, H: 16 x 24-bit (B type is omitted, as including it would increase bitwidth of shared shifter required to implement this feature)
Predication	not available
Source options	src1: single vector register in VRF src2: single vector register in VRF

Instruction name	VNormIdxFrac
	src3 (implicit) PL scalar register
Destination options	dst1: single vector register in VRF dst2: single vector register in VRF
Additional options	

Intrinsics/operator	`void vnorm_idx_frac(vintx src1, vintx src2, int src3, vintx & dst1, vintx & dst2);`
	
	`void vnorm_idx_frac(vshortx src1, vshortx src2, int src3, vshortx & dst1, vshortx & dst2);`
	
	`// Double vector pseudo intrinsic`
	`void dvnorm_idx_frac(dvintx src1, dvintx src2, int src3, dvintx & dst1, dvintx & dst2);`
	
	`void dvnorm_idx_frac(dvshortx src1, dvshortx src2, int src3, dvshortx & dst1, dvshortx & dst2);`

Additional details	src1 carries the input data, src2[7:0] carries the MSB position previously detected via VMSBD on src1. src3[3:0] (implicit in PL scalar register) carries index_nbits, number of index bits in a subsequent table lookup. Dst1 returns the index, and dst2 returns the fraction.
	src2[7:0] is read as a signed 8-bit number to accommodate VMSBD return value in [-1, 23] for Halfword and [-1, 47] for Word.
	src3[3:0] conveys index_nbits, and has valid range of 6 ~ 9. In case src3[3:0] is below 6 or above 9, both dst1 and dst2 return 0.
	The lookup table should contain 2^index_nbits + 1 entries, so index_nbits being 6 ~ 9 corresponds to 65 ~ 1025 entries, which is a reasonable table size of lookup table for a log table to get reasonable accuracy through linearly interpolated lookup.
	Index output is for a subsequent table lookup, so is extracted from src1 bits from msb_pos-1 downto msb_pos – index_nbits and right justified.
	Fraction output is for post-lookup linear interpolation, so is extracted from src1 bits from msb_pos – index_nbits – 1 (following index bits) downto 0 and left justified.
	Pseudo-code for Halfword type, in lane i:
	`norm_pos = 15;`
	`frac_mask = (1 << norm_pos) – 1;`
	`index_nbits = PL[3:0]; // read as unsigned int4`
	`idx_mask = (1 << index_nbits) – 1;`
	`input = src1[i];`
	`msb_pos = src2[i][7:0]; // read as signed int8`
	
	`if (index_nbits < 6 || index_nbits > 9)`
	`dst[i] = 0;`
	`else {`
	`shiftVal = norm_pos – msb_pos + index_nbits;`
	`shiftVal = (shiftVal < -24) ? -24 :`
	`((shiftVal > 24) ? 24 : ShiftVal);`

Instruction name	VNormIdxFrac

	idx_frac = shift(input, shiftVal);
	// shift left for positive shiftVal
	// shift right for negative shiftVal
	frac = idx_frac & frac_mask;
	idx = (idx_frac >> norm_pos) & idx_mask;
	}
	
	For Word type, norm_pos = 31, and shiftVal is saturated to [-48, 48] instead.

9.8.9.7 VHMin4_CA

Instruction name	VHMin4_CA
Functionality	Vector horizontal min-4 accumulate
Assembly format	<pred> VHMin4<type>_CA DVsrc1, Wsrc2, ACsrc3dst pred = none, [P2..P15]
Type and bit width	B: 32 x 12-bit, H: 16 x 24-bit
Predication	Available across lanes to clear accumulator
Source options	src1: double vector register in VRF src2: single vector register in WRF
Destination options	src3dst: single vector register in ARF
Additional options	

Intrinsics/operator	vcharx vhmin4_ca(dvcharx src1, vcharx src2, vcharx src3, int pred);
	
	vshortx vhmin4_ca(dvshortx src1, vshortx src2, vshortx src3,int pred);

Additional details	Src1 .lo and .hi carry overlapping data elements offset by 4 elements. Src2 carries control parameter to include/exclude input in bit 0 of each lane. Src3dst is the accumulator. The instruction carries out min operation among horizontally overlapping 4 data terms and the accumulator when the predicate is true. When the predicate is false, the accumulator input is ignored, effectively clearing the accumulator. Layout of data for each 4 lane group:

src1.lo	D[0]	D[1]	D[2]	D[3]
src1.hi	D[4]	D[5]	D[6]	D[7]
src2	C[0]	C[1]	C[2]	C[3]
src3dst	ACC[0]	ACC[1]	ACC[2]	ACC[3]

m0 = C[0][0] (bit 0 of C[0]), m1 = C[1][0], m2 = C[2][0], m3 = C[3][0]
ACC[0] = min(mask(m0, D[0]), mask(m1, D[1]), mask(m2, D[2]), mask(m3, D[3]), ACC[0]);
ACC[1] = min(mask(m0, D[1]), mask(m1, D[2]), mask(m2, D[3]), mask(m3, D[4]),

	ACC[1]); ACC[2] = min(mask(m0, D[2]), mask(m1, D[3]), mask(m2, D[4]), mask(m3, D[5]), ACC[2]); ACC[3] = min(mask(m0, D[3]), mask(m1, D[4]), mask(m2, D[5]), mask(m3, D[6]), ACC[3]); mask(m, d) = (m == 0) ? INT_MAX : d INT_MAX is the maximal integer value for the type. Basically, when the control parameter is 0, the data term is replaced with INT_MAX and thus excluded from the min operation.

9.8.9.8 VHMax4_CA

Instruction name	**VHMax4_CA**
Functionality	Vector horizontal max-4 accumulate
Assembly format	<pred> VHMax4<type>_CA DVsrc1, Wsrc2, ACsrc3dst pred = none, [P2..P15]
Type and bit width	B: 32 x 12-bit, H: 16 x 24-bit
Predication	Available across lanes to clear accumulator
Source options	src1: double vector register in VRF src2: single vector register in WRF
Destination options	src3dst: single vector register in ARF
Additional options	

Intrinsics/operator	`vcharx vhmax4_ca(dvcharx src1, vcharx src2, vcharx src3, int pred);`
	
	`vshortx vhmax4_ca(dvshortx src1, vshortx src2, vshortx src3,int pred);`

Additional details	Src1 .lo and .hi carry overlapping data elements offset by 4 elements. Src2 carries control parameter to include/exclude input in bit 0 of each lane. Src3dst is the accumulator. The instruction carries out max operation among horizontally overlapping 4 data terms and the accumulator when the predicate is true. When the predicate is false, the accumulator input is ignored, effectively clearing the accumulator. Layout of data for each 4 lane group:

src1.lo	D[0]	D[1]	D[2]	D[3]
src1.hi	D[4]	D[5]	D[6]	D[7]
src2	C[0]	C[1]	C[2]	C[3]
src3dst	ACC[0]	ACC[1]	ACC[2]	ACC[3]

m0 = C[0][0] (bit 0 of C[0]), m1 = C[1][0], m2 = C[2][0], m3 = C[3][0]
ACC[0] = max(mask(m0, D[0]), mask(m1, D[1]), mask(m2, D[2]), mask(m3, D[3]), ACC[0]);

Instruction name	VHMax4_CA
	ACC[1] = max(mask(m0, D[1]), mask(m1, D[2]), mask(m2, D[3]), mask(m3, D[4]), ACC[1]); ACC[2] = max(mask(m0, D[2]), mask(m1, D[3]), mask(m2, D[4]), mask(m3, D[5]), ACC[2]); ACC[3] = max(mask(m0, D[3]), mask(m1, D[4]), mask(m2, D[5]), mask(m3, D[6]), ACC[3]); mask(m, d) = (m == 0) ? INT_MIN : d INT_MIN is the minimal integer value for the type. Basically, when the control parameter is 0, the data term is replaced with INT_MIN and thus excluded from the max operation.

9.9 Load/Store Instructions

9.9.1 Scalar Load/Store

9.9.1.1 Instruction Summary

Scalar load/store are supported with signed/unsigned word/half-word/byte data types and the following address modes:

- Indexed, with register base and 12-bit (singed) immediate offset
- Post increment, with register base and register modifier value

Table 38. Scalar load/store instructions

Function	Assembly Format	Comments
Load base-offset	LD<type> *(Rbase+imm12), Rdst type = B, BU, H, HU, W	Use Rbase + (signed) imm12 as byte address. Data is sign-extended or zero-padded.
Load post-modify	<pred> LD<type> *Rbase+=Rmod, Rdst pred = none, [P2.. P15] type = B, BU, H, HU, W	Use Rbase as byte address, Rbase is post-modified. Data is sign-extended or zero-padded.
Store base-offset	ST<type> Rsrc, *(Rbase+imm12) type = B, H, W	Use Rbase + (signed) imm12 as byte address.
Store post-modify	<pred> ST<type> Rsrc, *Rbase+=Rmod pred = none, [P2.. P15] type = B, H, W	Use Rbase as byte address, Rbase is post-modified.

9.9.1.2 LD Base-Offset

Instruction name	LD
Functionality	Load
Assembly format	LD<type> *(Rbase+imm12), Rdst
Type and bit width	B/BU: 8-bit (char, unsigned char) H/HU: 16-bit (short, unsigned short, hfloat) W: 32-bit (int, unsigned int, float)
Predication	Not available
Source options	Rbase: scalar register
Destination options	Rdst: scalar register
Additional options	

Intrinsics/operator	`not needed`
	`// Instantiated to read from array or local frame, e.g.,`
	`// a = array[10];`

Additional details	For example, LDW *(R1+12), R4 Use Rbase + (signed) imm12 as byte address, Rbase is not modified. Data is sign-extended or zero-padded, based on specified type being signed or unsigned.

9.9.1.3 LD Post-Modify

Instruction name	LD
Functionality	Load post-modify
Assembly format	<pred> LD<type> *Rbase+=Rmod, Rdst pred = none, [P2.. P15]
Type and bit width	B/BU: 8-bit (char, unsigned char) H/HU: 16-bit (short, unsigned short, hfloat) W: 32-bit (int, unsigned int, float)
Predication	Instruction-level predication
Source options	Rbase: scalar register Rmod: scalar register
Destination options	Rdst: scalar register Rbase: scalar register
Additional options	

Intrinsics/operator	`not needed`
	`// Instantiated to read from array with pointer increment, e.g.,`
	`// a = *ptr++;`

Additional details	Use Rbase as byte address, Rbase is post-modified to Rbase+(signed) Rmod. Data is sign-extended or zero-padded, based on specified type being signed or unsigned.

Instruction name	LD
	Predication: Execute (memory read into Rdst and Rbase post-modify) only if the referenced predicate register != 0.

9.9.1.4 ST Base-Offset

Instruction name	ST
Functionality	Store
Assembly format	ST<type> Rsrc, *(Rbase+imm12)
Type and bit width	B: 8-bit (char) H: 16-bit (short, hfloat) W: 32-bit (int, float)
Predication	not available
Source options	Rbase: scalar register Rsrc: scalar register
Destination options	
Additional options	

Intrinsics/operator	`not needed`
	`// Instantiated to write into array or local frame, e.g.,`
	`// array[10] = b;`

Additional details	For example, STW R4, *(R1+12) STH R5, *(R1+16) Use Rbase + (signed) imm12 as byte address, Rbase is not modified.

9.9.1.5 ST Post-Modify

Instruction name	ST
Functionality	Store post-modify
Assembly format	<pred> ST<type> Rsrc, *Rbase+=Rmod pred = none, [P2.. P15]
Type and bit width	B: 8-bit (char) H: 16-bit (short, hfloat) W: 32-bit (int, float)
Predication	Instruction-level predication
Source options	Rbase: scalar register Rmod: scalar register Rsrc: scalar register
Destination options	Rbase: scalar register
Additional options	
Intrinsics/operator	`not needed`

Instruction name	ST
	`// Instantiated to write into array with pointer increment, e.g.,` `// *ptr++ = b;`
Additional details	Use Rbase as byte address, Rbase is post-modified to Rbase+(signed) Rmod. Predication: Execute (memory write and Rbase modify) only if the referenced predicate register != 0.

9.9.2 Scalar-Based Vector Load/Store

9.9.2.1 Instruction Summary

Table 39. Scalar-based vector load/store instructions

Function	Assembly Format	Comments
Vector load base plus offset	VLD<type>_P *(Rbase+Imm), Vdst type = B, BU, H, HU, W, WU, WX VLDWX_P *(Rbase+Imm), Wdst	Use Rbase + (4*imm10) as byte address. Data is sign-extended or zero-padded.
Vector load post-modify	VLD<type>_P *Rbase+=Rmod, Vdst type = B, BU, H, HU, W, WU, WX	Use Rbase as byte address, Rbase is post-modified to Rbase+Rmod. Data is sign-extended or zero-padded.
Double vector load post-modify	DVLD<type>_P *Rbase+=Rmod, Vdst type = B, BU, H, HU, W, WU	Use Rbase as byte address, Rbase is post-modified to Rbase+Rmod. Data is sign-extended or zero-padded.
Vector store base plus offset	VST<type>_P Vsrc, *(Rbase+Imm) type = B, H, W, WX VSTWX_P Wsrc, *(Rbase+Imm)	Use Rbase + (4*imm10) as byte address.
Vector store post-modify	VST<type>_P Vsrc, *Rbase+=Rmod type B, H, W, WX, BH, HW	Use Rbase as byte address, Rbase is post-modified to Rbase+Rmod.
Double vector store post-modify	DVST<type>_P Vsrc, *Rbase+=Rmod type B, H, W	Use Rbase as byte address, Rbase is post-modified to Rbase+Rmod.

9.9.2.2 Base-Offset

Instruction name	VLD base-offset
Functionality	Vector load base plus offset
Assembly format	VLD<type>_P *(Rbase+Imm), Vdst VLDWX_P *(Rbase+Imm), Vdst/Wdst
Type and bit width	B/BU: 32 x 8-bit → 32 x 12-bit (vchar/vuchar -> vcharx) H/HU: 16 x 16-bit → 16 x 24-bit (vshort/vushort -> vshortx, vhfloat -> vhfloatx) W/WU for VRF: 8 x 32-bit → 8 x 48-bit (vint/vuint -> vintx, vfloat -> vfloatx) WX: 8 x 48-bit → 8 x 48-bit (vcharx, vshortx, vintx)
Predication	Not available
Source options	Rbase: scalar register
Destination options	Single vector register in VRF, WRF
Additional options	

Intrinsics/operator	`vcharx sign_extend(vchar src);`
	`vshortx sign_extend(vshort src);`
	`vintx sign_extend(vint src);`
	`vfloatx sign_extend(vfloat src);`
	`vhfloatx sign_extend(vhfloat src);`
	
	`vcharx zero_extend(vuchar src);`
	`vshortx zero_extend(vushort src);`
	`vintx zero_extend(vuint src);`
	
	`// Instantiated with memory read with sign/zero extension, e.g.,`
	`// vcharx v1 = sign_extend(vchar_array[3]);`
	`// vcharx v1 = sign_extend(*((vchar *)(char_array + 10)));`
	
	`// WX load does not require intrinsic function. Instantiated with,`
	`// e.g.,`
	`// vcharx v2 = vcharx_array[10];`

Additional details	10-bit immediate field is scaled by 4 and added to Rbase as the byte address. Rbase is not modified. Data is sign-extended or zero-padded, based on specified type being signed or unsigned.

9.9.2.3 VLD Post-Modify

Instruction name	VLD post-modify
Functionality	Vector load post-modify
Assembly format	VLD<type>_P *Rbase+=Rmod, Vdst
Type and bit width	B/BU: 32 x 8-bit → 32 x 12-bit (vchar/vuchar -> vcharx) H/HU: 16 x 16-bit → 16 x 24-bit (vshort/vushort -> vshortx, vhfloat -> vhfloatx) W/WU for VRF: 8 x 32-bit → 8 x 48-bit (vint/vuint -> vintx, vfloat -> vfloatx) WX: 8 x 48-bit → 8 x 48-bit (vcharx, vshortx, vintx)
Predication	Not available
Source options	Rbase: scalar register
Destination options	Vdst: single vector register
Additional options	

Intrinsics/operator	`vcharx sign_extend(vchar src);`
	`vshortx sign_extend(vshort src);`
	`vintx sign_extend(vint src);`
	`vfloatx sign_extend(vfloat src);`
	`vhfloatx sign_extend(vhfloat src);`
	
	`vcharx zero_extend(vuchar src);`
	`vshortx zero_extend(vushort src);`
	`vintx zero_extend(vuint src);`
	
	`// Instantiated with post-increment memory read with`
	`// sign/zero extension, e.g.,`
	`// vcharx v1 = sign_extend(*vchar_ptr++);`
	
	`// WX load does not require intrinsic function. Instantiated with,`
	`// e.g.,`
	`// vcharx v2 = *vcharx_ptr++;`

Additional details	Use Rbase as byte address, Rbase is post-modified to Rbase+Rmod. Data is sign-extended or zero-padded, based on specified type being signed or unsigned.

9.9.2.4 DVLD Post-Modify

Instruction name	DVLD post-modify
Functionality	Double vector load post-modify
Assembly format	DVLD<type>_P *Rbase+=Rmod, DVdst
Type and bit width	B/BU: 64 x 8-bit → 2 x 32 x 12-bit (dvchar/dvuchar -> dvcharx) H/HU: 32 x 16-bit → 2 x 16 x 24-bit (dvshort/dvushort -> dvshortx, dvhfloat -> dvhfloatx) W/WU: 16 x 32-bit → 2 x 8 x 48-bit (dvint/dvuint -> dvintx, dvfloat -> dvfloatx)
Predication	Not available
Source options	Rbase: scalar register
Destination options	Vdst: double vector register
Additional options	

Intrinsics/operator	`dvcharx sign_extend(dvchar src);`
	`dvshortx sign_extend(dvshort src);`
	`dvintx sign_extend(dvint src);`
	`dvfloatx sign_extend(dvfloat src);`
	`dvhfloatx sign_extend(dvhfloat src);`
	
	`dvcharx zero_extend(dvuchar src);`
	`dvshortx zero_extend(dvushort src);`
	`dvintx zero_extend(dvuint src);`
	
	`// Instantiated with post-increment memory read with sign/zero`
	`// extension, e.g.,`
	`// dvcharx v1 = sign_extend(*dvchar_ptr++);`

Additional details	Use Rbase as byte address, Rbase is post-modified to Rbase+Rmod. Data is sign-extended or zero-padded, based on specified type being signed or unsigned.

9.9.2.5 VST Base-Offset

Instruction name	VST base-offset
Functionality	Vector store base plus offset
Assembly format	VST<type>_P Vsrc, *(Rbase+Imm) VSTWX_P Vsrc/Wsrc, *(Rbase+Imm)
Type and bit width	B: 32 x 12-bit → 32 x 8-bit (vcharx -> vchar/vuchar) H: 16 x 24-bit → 16 x 16-bit (vshortx -> vshort/vushort, vhfloatx -> vhfloat) W for VRF: 8 x 48-bit → 8 x 32-bit (vintx -> vint/vuint, vfloatx -> vfloat) WX: 8 x 48-bit → 8 x 48-bit (vcharx, vshortx, vintx)
Predication	Not available
Source options	Rbase: scalar register Single vector register in VRF, WRF
Destination options	
Additional options	

Intrinsics/operator	`vchar extract(vcharx src);`
	`vshort extract(vshortx src);`
	`vint extract(vintx src);`
	`vfloat extract(vfloatx src);`
	`vhfloat extract(vhfloatx src);`
	
	`// Instantiated with memory write with sign/zero extension,`
	`// e.g.,`
	`// vchar_array[3] = extract(vcharx_var);`
	`// *((vchar *)(char_array + 10)) = extract(vcharx_var);`
	
	`// WX does not require intrinsic function. Instantiated`
	`// with, e.g.,`
	`// vcharx_array[3] = vcharx_var;`

Additional details	10-bit immediate field is scaled by 4 and added to Rbase as the byte address. Rbase is not modified.

9.9.2.6 VST Post-Modify

Instruction name	VST post-modify
Functionality	Vector store post-modify
Assembly format	VST<type>_P Vsrc, *Rbase+=Rmod
Type and bit width	B: 32 x 12-bit → 32 x 8-bit (vcharx -> vchar/vuchar) H: 16 x 24-bit → 16 x 16-bit (vshortx -> vshort/vushort, vhfloatx -> vhfloat) W for VRF: 8 x 48-bit → 8 x 32-bit (vintx -> vint/vuint, vfloatx -> vfloat) WX: 8 x 48-bit → 8 x 48-bit (vcharx, vshortx, vintx) BH: 32 x 12-bit → 32 x 16-bit (vcharx -> dvshort) HW: 16 x 24-bit → 16 x 32-bit (vshortx -> dvint)
Predication	Not available
Source options	Rbase: scalar register Vsrc: single vector register
Destination options	Rbase: scalar register
Additional options	

Intrinsics/operator	`vchar extract(vcharx src);`
	`vshort extract(vshortx src);`
	`vint extract(vintx src);`
	`vfloat extract(vfloatx src);`
	`vhfloat extract(vhfloatx src);`
	
	`// Instantiated with post-increment memory write with`
	`// sign/zero extension, e.g.,`
	`// *vchar_ptr++ = extract(vcharx_var);`
	
	`// WX does not require intrinsic function. Instantiated`
	`// with, e.g.,`
	`// *vcharx_ptr++ = vcharx_var;`

Additional details	Use Rbase as byte address, Rbase is post-modified to Rbase+Rmod.

9.9.2.7 DVST Post-Modify

Instruction name	DVST post-modify
Functionality	Double vector store post-modify
Assembly format	DVST<type>_P DVsrc, *Rbase+=Rmod
Type and bit width	B: 2 x 32 x 12-bit → 64 x 8-bit (dvcharx -> dvchar/dvuchar) H: 2 x 16 x 24-bit → 32 x 16-bit (dvshortx -> dvshort/dvushort) W: 2 x 8 x 48-bit → 16 x 32-bit (dvintx -> dvint/dvuint)
Predication	Not available
Source options	Rbase: scalar register Vsrc: double vector register
Destination options	Rbase: scalar register
Additional options	

Intrinsics/operator	`dvchar extract(dvcharx src);`
	`dvshort extract(dvshortx src);`
	`dvint extract(dvintx src);`
	`dvfloat extract(dvfloatx src);`
	`dvhfloat extract(dvhfloatx src);`
	
	`// Instantiated with post-increment memory write with`
	`// sign/zero extension, e.g.,`
	`// *dvchar_ptr++ = extract(dvcharx_var);`

Additional details	Use Rbase as byte address, Rbase is post-modified to Rbase+Rmod.

9.9.3 Agen Configuration

9.9.3.1 Instruction Summary

In scalar slots we allow the following instructions to configure the agen.

Table 40. Agen config instructions

Function	Assembly Format	Comments
Initialize agen	InitAgen Rsrc, A<id>.Base id = 0..7	Set base address and initialize all other parameters to default values, including resetting loop variables I1..I6 to 0
Configure agen base	<pred> CfgAgen Rsrc, A<id>.Base id = 0..7	Set base address (predicated)
Configure agen num Iterations	CfgAgen Rsrc, A<id>.N<level> id = 0..7, level = 1..6	Only lower 16 bits are used. Default = 1.
Configure agen address modifier	CfgAgen Rsrc, A<id>.Mod<level> id = 0..7, level = 1..6	Address modifiers are signed.

Function	Assembly Format	Comments
		32 bits are stored, but only lower 17 bits are used in address calculation. Default = 0
Configure rounding	CfgAgen Rsrc, A<id>.Round	Rounding applies to store only and is ignored for WX type store. Bit 7 specifies round (0) or truncate (1). Bits 6:0 specifies number of bits to round/truncate. Default = 0 (no rounding)
Configure saturation option	CfgAgen Rsrc, A<id>.SatOpt	Only 2 LSBs of Rsrc are used. 0 : no saturation (default) 1 : no saturation 2 : treat 32-bit comparison values as signed 3 : treat 32-bit comparison values as unsigned Saturation option is ignored for WX type store.
Configure lane offset	CfgAgen Rsrc, A<id>.LaneOfst	Lane offsets are unsigned. Default = 0
Configure saturation	CfgAgen Rsrc, A<id>.SatLimLo CfgAgen Rsrc, A<id>.SatLimHi CfgAgen Rsrc, A<id>.SatValLo CfgAgen Rsrc, A<id>.SatValHi	Saturation applies to store only and is ignored for WX type store. Default = 0
Configure circular buffer	CfgAgen Rsrc, A<id>.CBStart CfgAgen Rsrc, A<id>.CBSize	Configure starting address and size of circular buffer, Rsrc is read as byte address, and is right-shifted 6 bits before writing to the CBStart and CBSize fields to force 64-byte alignment. CBSize = 0 indicates circular buffer is disabled. Default = 0.
Agen Config Move	MovAgen A<src_id>, A<dst_id>	Copy all agen parameters and loop variables (608-bit).
Save agen config	AgenCfgST A<id>, *Rptr += Rmod AgenCfgST_p2 A<id>, *Rptr += Rmod	Save first/second 512-bit of agen data structure.
Restore agen config	AgenCfgLD *Rptr += Rmod, A<id> AgenCfgLD_p2 *Rptr += Rmod, A<id>	Restore first/second 512-bit of agen data structure.
Agen update	AgenUpd A<id>++	Update agen without memory transaction
Move agen base	MovAgen A<id>.Base, Rdst	Copy agen base address to scalar
Advance agen base	AdvAgen A<id>.Base, Rsrc2	Perform circular buffer wrap-around if configured

Function	Assembly Format	Comments
Configure min/max option	CfgAgen Rsrc, A<id>.MinMaxOpt	Configure minmax_opt, initialize min/max values accordingly
Move agen min/max	MovAgen A<id>.MinVal, Rdst MovAgen A<id>.MaxVal, Rdst	Copy agen collected min/max value to scalar

9.9.3.2 InitAgen

Instruction name	InitAgen
Functionality	Initialize agen with base address
Assembly format	InitAgen Rsrc, A<id>.Base id = 0..7
Type and bit width	Base: 32-bit unsigned
Predication	Not available
Source options	Rsrc: scalar register
Destination options	Agen config (all parameters including base)
Additional options	
Intrinsics/operator	`agen init(vint * arr1);`
Additional details	Set base address and initialize all other parameters to default values, including resetting loop variables I1..I6 to 0

9.9.3.3 CfgAgen Base

Instruction name	CfgAgen base
Functionality	Configure agen base address
Assembly format	<pred> CfgAgen Rsrc, A<id>.Base id = 0..7 pred = none, [P2.. P15]
Type and bit width	32-bit unsigned
Predication	Instruction-level predication
Source options	Rsrc: scalar register
Destination options	Agen config base
Additional options	

Intrinsics/operator	`// Not needed, just assign to agen member a`
	`// For example,`
	`agen1.a = (vint *) array1;`

Additional details	Set base address

9.9.3.4 CfgAgen NIter

Instruction name	CfgAgen Niter
Functionality	Configure number of iterations
Assembly format	CfgAgen Rsrc, A<id>.N<level> id = 0..7, level = 1..6
Type and bit width	16-bit
Predication	Not available
Source options	Rsrc: scalar register
Destination options	Agen[id].N[level]
Additional options	

Intrinsics/operator	`// Not needed, just assign to agen member n1..n6`
	`// For example,`
	`agen1.n1 = niter1;`

Additional details	Only lower 16 bits of Rsrc are used. Default = 1. Programming it to 0 would exhibit the same looping behavior as programming it to 1.

9.9.3.5 CfgAgen Mod

Instruction name	CfgAgen Mod
Functionality	Configure agen address modifier
Assembly format	CfgAgen Rsrc, A<id>.Mod<level> id = 0..7, level = 1..6
Type and bit width	Of the 32-bit value in Rsrc, only 18 LSBs stored in the designated Mod register.
Predication	Not available
Source options	Rsrc: scalar register
Destination options	Agen[id].Mod[level]
Additional options	

Intrinsics/operator	`// Not needed, just assign to agen member mod1..mod6`
	`// For example,`
	`agen1.mod1 = vector_width * sizeof(data);`

Additional details	Default = 0. Note that address modiers are signed.

9.9.3.6 CfgAgen Round

Instruction name	CfgAgen Round
Functionality	Configure agen rounding parameter
Assembly format	CfgAgen Rsrc, A<id>.Round

Instruction name	CfgAgen Round
	id = 0..7
Type and bit width	8-bit unsigned
Predication	Not available
Source options	Rsrc: scalar register
Destination options	Agen[id].Round
Additional options	

Intrinsics/operator	`// Not needed, just assign to agen member round`
	`// For example,`
	`agen1.round = qbits;`

Additional details	Rounding applies to store only, and is ignored for WX-type store. Only bit 7 and bits 6:0 of Rsrc are used. Bits 7 specifies round (0), truncate (1) Bits 6:0 specifies number of bits to round/truncate. When number of bits exceeds source lane width (B=12, H=24, W=48), rounding leads to zero for all inputs, and truncation leads to zero for zero/positive inputs, and to -1 for negative inputs. Default = 0 (no rounding)

9.9.3.7 CfgAgen SatOpt

Instruction name	CfgAgen SatOpt
Functionality	Configure agen saturation option
Assembly format	CfgAgen Rsrc, A<id>.SatOpt id = 0..7
Type and bit width	8-bit unsigned (only 2 LSBs are used)
Predication	Not available
Source options	Rsrc: scalar register
Destination options	Agen[id].SatOpt
Additional options	

Intrinsics/operator	`// Not needed, just assign to agen member round`
	`// For example,`
	`agen1.sat_opt = 0;`

Additional details	Only 2 LSBs of Rsrc are used. 0 : no saturation (default) 1 : no saturation 2 : treat 32-bit comparison values as signed 3 : treat 32-bit comparison values as unsigned This is ignored for WX-type store.

9.9.3.8 CfgAgen LaneOfst

Instruction name	CfgAgen LaneOfst
Functionality	Configure agen lane offset
Assembly format	CfgAgen Rsrc, A<id>.LaneOfst id = 0..7
Type and bit width	12-bit unsigned
Predication	Not available
Source options	Rsrc: scalar register
Destination options	Agen[id].LaneOfst
Additional options	

Intrinsics/operator	`// Not needed, just assign to agen member lane_ofst`
	`// For example,`
	`agen1.lane_ofst = num_columns/16;`

Additional details	Only 12 LSBs of Rsrc are stored into the lane offset field. Default = 0. Note that lane offsets are unsigned. Used for transposing vector load/store. See 6.3.7 for details.

9.9.3.9 CfgAgen Sat

Instruction name	CfgAgen Sat
Functionality	Configure agen saturation
Assembly format	CfgAgen Rsrc, A<id>.SatLimLo CfgAgen Rsrc, A<id>.SatLimHi CfgAgen Rsrc, A<id>.SatValLo CfgAgen Rsrc, A<id>.SatValHi id = 0..7
Type and bit width	32-bit
Predication	Not available
Source options	Rsrc: scalar register
Destination options	Agen[id].SatLimLo/SatLimHi/SatValLo/SatValHi
Additional options	

Intrinsics/operator	`// Not needed; just assign to corresponding agen struct members`
	`// For example,`
	`agen1.sat_lim_lo = low_bound;`
	`agen1.sat_val_lo = low_bound;`
	`agen1.sat_lim_hi = high_bound;`

Additional details	Saturation applies to store only and is ignored for WX-type store. Default = 0

9.9.3.10 CfgAgen CB

Instruction name	CfgAgen CB
Functionality	Configure agen circular buffer
Assembly format	CfgAgen Rsrc, A<id>.CBStart CfgAgen Rsrc, A<id>.CBSize id = 0..7
Type and bit width	16-bit (from 32-bit scalar register source, 6 LSBs are dropped, bits 21:6 are stored into agen cb_start or cb_size fields)
Predication	Not available
Source options	Rsrc: scalar register
Destination options	Agen[id].CBStart/CBSize
Additional options	

Intrinsics/operator	`// Recommended syntax, works in ISS and Native`
	`short chess_storage(DMh%64: chess_segment(C)) cb_buf1[CB1_SIZE];`
	`agen1 = update_agen_cb_start(agen1, (short *) cb_buf1);`
	`agen2 = update_agen_cb_size(agen2, CB1_SIZE * sizeof(short));`
	`// Legacy syntax, works in ISS but not in Native`
	`agen agen1.cb_start = (short *) cb_buf1;`
	`agen agen2.cb_size = CB1_SIZE * sizeof(short);`

Additional details	Configure starting address and size of circular buffer, Rsrc is read as byte address or size in bytes, and is right-shifted 6 bits before writing to the CBStart and CBSize fields to force 64-byte alignment. CBSize = 0 indicates circular buffer is disabled. Default = 0. Note that CBStart and CBSize are both unsigned.

9.9.3.11 MovAgen

Instruction name	MovAgen
Functionality	Agen config move
Assembly format	MovAgen A<src_id>, A<dst_id> src_id/dst_id = 0..7
Type and bit width	608-bit
Predication	Not available
Source options	Agen[src_id]
Destination options	Agen[dst_id]
Additional options	

Intrinsics/operator	`// Not needed, just assign an agen to another agen`
	`// For example,`
	`agen2 = agen1;`

Additional details	Copy all agen parameters and loop variables (608-bit).

9.9.3.12 AgenCfgST

Instruction name	AgenCfgST
Functionality	Save agen config
Assembly format	AgenCfgST A<id>, *Rptr += Rmod AgenCfgST_p2 A<id>, *Rptr += Rmod id = 0..7
Type and bit width	512-bit or 192-bit
Predication	Not available
Source options	Agen[id] Rptr/Rmod: scalar register
Destination options	Rmod: scalar register
Additional options	

Intrinsics/operator	`AgenCFG agen1.get_cfg(); // Legacy syntax also supported`
	`AgenCFG_p2 extract_agen_cfg_p2(agen src);`
	`// For example,`
	`AgenCFG * ptr = &cfg_arr[0];`
	`AgenCFG_p2 ptr2 = &cfg2_arr[0];`
	`*ptr++ = extract_agen_cfg(agen1);`
	`*ptr2++ = extract_agen_cfg_p2(agen1);`

Additional details	AgenCfgST saves the first 512-bit of agen data structure AgenCfgST_p2 saves the remaining 192-bit of agen data structure Address should be 32-bit aligned. For readability each MOD1..MOD6 register is sign-extended to 32-bit in stored memory locations. Available only in the M0 slot.

Instruction name	AgenCfgST (base-offset)
Functionality	Save agen config
Assembly format	AgenCfgST A<id>, *(Rbase + Imm12) AgenCfgST_p2 A<id>, *(Rbase + Imm12) id = 0..7
Type and bit width	512-bit or 192-bit
Predication	Not available
Source options	Agen[id] Rbase: scalar register Imm12: 12-bit immediate byte address offset
Destination options	n/a
Additional options	

Intrinsics/operator	`AgenCFG extract_agen_cfg(agen src); // Recommended`
	`AgenCFG agen1.get_cfg(); // Legacy syntax also supported`
	`AgenCFG_p2 extract_agen_cfg_p2(agen src);`
	`// For example,`

Instruction name	AgenCfgST (base-offset)

	`AgenCFG cfg_arr[3];`
	`AgenCFG cfg2_arr[3];`
	`cfg_arr[0] = extract_agen_cfg(agen1);`
	`cfg2_arr[0] = extract_agen_cfg_p2(agen1);`

Additional details	AgenCfgST saves the first 512-bit of agen data structure AgenCfgST_p2 saves the remaining 192-bit of agen data structure Address should be 32-bit aligned. For readability each MOD1..MOD6 register is sign-extended to 32-bit in stored memory locations. Available only in the M0 slot.

9.9.3.13 AgenCfgLD

Instruction name	AgenCfgLD
Functionality	Restore agen config
Assembly format	AgenCfgLD *Rptr += Rmod, A<id> AgenCfgLD_p2 *Rptr += Rmod, A<id> id = 0..7
Type and bit width	512-bit or 192-bit
Predication	Not available
Source options	Rptr/Rmod: scalar register
Destination options	Agen[id] Rmod: scalar register
Additional options	

Intrinsics/operator	`agen init_agen_from_cfg(AgenCFG src); // Recommended`
	`agen agen1.expand_cfg(AgenCFG src); // Legacy syntax also supported`
	`agen update_agen_p2(agen a1, AgenCFG_p2 data_p2);`
	`// For example,`
	`AgenCFG * ptr1 = &cfg_arr[0];`
	`AgenCFG_p2 * ptr2 = &cfg2_arr[0];`
	`agen a1 = init_agen_from_cfg(*ptr1++);`
	`a1 = update_agen_p2(a1, *ptr2++);`

Additional details	AgenCfgLD restores the first 512-bit of agen data structure, and set the rest to “sensible” initial state ready to execute dependent agen-based load/store instructions All loop variables to 0 auto_pred_off to 0 MinVal/MaxVal to 0, INT32 MAX/MIN, UINT32 MAX/MIN, according to MinMaxOpt AgenCfgLD_p2 restores the remaining 192-bit of agen data structure from memory. Address in Rptr should be 32-bit aligned.

Instruction name	AgenCfgLD
	Each of MOD1..MOD6 register will only take 18 LSBs in corresponding 32-bit memory locations. Available only in the M0 slot.

Instruction name	AgenCfgLD (base-offset)
Functionality	Restore agen config
Assembly format	AgenCfgLD *(Rbase + Imm12), A<id> AgenCfgLD_p2 *(Rbase + Imm12), A<id> id = 0..7
Type and bit width	512-bit or 192-bit
Predication	Not available
Source options	Rbase: scalar register Imm12: 12-bit immediate byte address offset
Destination options	Agen[id]
Additional options	

Intrinsics/operator	`agen init_agen_from_cfg(AgenCFG src); // Recommended`
	`agen agen1.expand_cfg(AgenCFG src); // Legacy syntax also supported`
	`agen update_agen_p2(agen a1, AgenCFG_p2 data_p2);`
	`// For example,`
	`AgenCFG cfg_arr[4];`
	`AgenCFG_p2 cfg2_arr[4];`
	`agen a1 = init_agen_from_cfg(cfg_arr[0]);`
	`a1 = update_agen_p2(a1, cfg2_arr[0]);`

Additional details	AgenCfgLD restores the first 512-bit of agen data structure, and set the rest to “sensible” initial state ready to execute dependent agen-based load/store instructions All loop variables to 0 auto_pred_off to 0 MinVal/MaxVal to 0, INT32 MAX/MIN, UINT32 MAX/MIN, according to MinMaxOpt AgenCfgLD_p2 restores the remaining 192-bit of agen data structure from memory. Address in Rptr should be 32-bit aligned. Each of MOD1..MOD6 register will only take 18 LSBs in corresponding 32-bit memory locations. Available only in the M0 slot.

9.9.3.14 AgenUpd

Instruction name	AgenUpd
Functionality	Update agen loop variables and address without memory transaction
Assembly format	AgenUpd A<id>++
Type and bit width	
Predication	Not available
Source options	
Destination options	
Additional options	
Intrinsics/operator	`agen update_agen(agen a);`
Additional details	Perform agen loop variables and address update as configured by agen parameters, without performing any memory load/store transaction. Note that this instruction is available in memory slots, as opposed to scalar slots for the other non-load/store agen configuration instructions.

9.9.3.15 Move Agen Base

Instruction name	MovAgen Base
Functionality	Copy agen address to scalar
Assembly format	MovAgen A<id>.Base, Rdst id = 0..7
Type and bit width	32-bit unsigned
Predication	Not available
Source options	Agen address
Destination options	scalar register
Additional options	
Intrinsics/operator	`// not needed, just access agen member a` `int * ptr = (int *) agen.a;`
Additional details	Move the current agen address (updated with each execution of agen-based load/store, rather than the starting address) to scalar register, mostly to facilitate debug. Note that this instruction is available in memory slots, as opposed to scalar slots for the other non-load/store agen configuration instructions.

9.9.3.16 Advance Agen Base

Instruction name	AdvAgenBase
Functionality	Advance agen base address by offset
Assembly format	AdvAgen A<id>.Base, Rsrc2 id = 0..7
Type and bit width	32-bit unsigned base + 18-bit signed offset
Predication	Not available
Source options	Agen address, scalar register supplying address offset
Destination options	Agen address
Additional options	
Intrinsics/operator	`void adv_agen_base(agen& srcdst, int ofst);`
Additional details	When circular buffer is configured (cb_size > 0), the base address is advanced by the offset (which can be positive or negative) with circular buffer wrap-around. In this case, magnitude of offset must not exceed circular buffer size, otherwise, the circular buffer addressing logic may not correctly wrap the modified address back into the circular buffer. See 6.4.6 Circular Buffer Addressing for details. When circular buffer is not configured (cb_size = 0), the base address is simply advanced (positively or negatively) by the offset, i.e., base += offset. Only 18 LSBs of Rsrc2 providing the offset is used in the address calculation, so this feature should not be used to move the base address between one superbank to another superbank. FINE PRINT: Technically it's possible, but leveraging the address wrapping behavior (see Memory Address Range Constraints) to place the base address at the edge of one superbank's primary-or-alias address space, and to advance it by as little as one byte to fall into another superbank's primary-or-alias address space. However, the address wrapping behavior is not backward or forward compatible, so this practice is very dangerous. This instruction is available in memory slots, as opposed to scalar slots for the other non-load/store agen configuration instructions.

9.9.3.17 CfgAgen MinMaxOpt

Instruction name	CfgAgen MinMaxOpt
Functionality	Configure min/max option
Assembly format	CfgAgen Rsrc, A<id>.MinMaxOpt id = 0..7
Type and bit width	8-bit unsigned (only 2 LSBs are used)
Predication	Not available
Source options	Scalar register
Destination options	Agen[id].MinMaxOpt
Additional options	

Instruction name	CfgAgen MinMaxOpt
Intrinsics/operator	`// Not needed; just assign to agen member minmax_opt` `agen1.minmax_opt = value;`
Additional details	Min/max option: 0: disable (default) 1: disable 2: enable for signed min/max 3: enable for unsigned min/max Upon configuring the min/max option to 2 (enabled for signed min/max), the min value is initialized to MAX_INT32 = 0x7FFF_FFFF. The max value is initialized to MIN_INT32 = 0x8000_0000. Upon configurating the min/max option to 3 (enabled for unsigned min/max), the min value is initialized to MAX_UINT32 = 0xFFFF_FFFF. The max value is initialized to MIN_UINT32 = 0. Upon configurating the min/max option to 0 or 1, the min/max values are reset to 0. Resetting min/max values as a consequence of configurating min/max option happens not just by this instruction, but also by InitAgen (setting min/max option to default 0 and min/max values to 0) and AgenCfgLD (restoring min/max option to whatever value saved in memory, and initializing min/max values according to the option).

9.9.3.18 Move Agen Min/Max

Instruction name	MovAgen Min/Max
Functionality	Copy agen collected min or max value to scalar
Assembly format	MovAgen A<id>.MinVal, Rdst MovAgen A<id>.MaxVal, Rdst id = 0..7
Type and bit width	32-bit signed
Predication	Not available
Source options	Agen min or max value
Destination options	scalar register
Additional options	

Intrinsics/operator	`// Not needed; just access agen member min_val or max_vol`
	`int dst1 = agen.min_val;`
	`int dst2 = agen.max_val;`

Additional details	

9.9.4 Agen-Based Vector Load/Store

Agen-based load/store offers more flexibility, in expanding and contracting between memory and vector register, and with data distribution options.

When double vector registers are used, it must be a consecutive V[2*i]:V[2*i+1] pair.

A parallel quad vector register store is also offered and only in M0 slot, to store 4 vector registers with demotion into 512-bit memory space. This is needed for filtering with 16-bit-by-16-bit multiply and 48-bit accumulator to achieve peak performance.

9.9.4.1 Instruction Summary

Vector load/store instructions:

Table 41. Agen-based vector load/store instructions

Function	Assembly Format	Comments
Vector load agen-based	<pred> VLD<type>_distr *A<id>++, Vdst/Wdst	
Double vector load agen-based	<pred> DVLD<type>_distr *A<id>++, DVdst/DWdst	
Vector store agen-based	<pred> VST<type>_distr Vsrc/ACsrc/XACsrc, *A<id>++	
Double vector store agen-based	<pred> DVST<type>_distr DVsrc/DACsrc/DXACsrc, *A<id>++	
Quad vector store agen-based	<pred> QVST<type>_distr DVsrc1, DVsrc2, *A<id>++ <pred> QVST<type>_distr DACsrc1, DACsrc2, *A<id>++	
Vector load + permute agen-based	VLDPerm<type>_<distr> *A<id>++, Vsrc/Wsrc, Vdst/Wdst	
Double vector load + permute agen-based	DVLDPerm<type>_<distr> *A<id>++, Vsrc/Wsrc, DVdst/DWdst	
Vector store with per-land rounding	DVST <type>_PLRound_distr Vsrc1/Wsrc1, DVsrc2/DACsrc2, *A<id>++	

9.9.4.2 VLD Agen

Instruction name	VLD agen
Functionality	Vector load agen-based
Assembly format	<pred> VLD<type>_<distr> *A<id>++, Vdst <pred> VLD<type>_<distr> *A<id>++, Wdst pred = none, [P2.. P15]
Type and bit width	Type/distribution supported: B_P, H_P, W_P, BU_P, HU_P, WU_P, B_T, H_T, W_T, BU_T, HU_T, WU_T, B_S, H_S, W_S, BU_S, HU_S, WU_S, B_C2, H_C2, W_C2, BU_C2, HU_C2, WU_C2, WX_P For example: VLDB_P *A0++, V0 [P1] VLDH_T, *A1++, V2
Predication	Instruction-level predication
Source options	
Destination options	Vdst: single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`// unpredicated`
	`vcharx vchar_load( agen& a1); // B_P`
	`vshortx vshort_load( agen& a1); // H_P`
	`vintx  vint_load(  agen& a1); // W_P`
	`vcharx vuchar_load( agen& a1); // BU_P`
	`vshortx vushort_load(agen& a1); // HU_P`
	`vintx  vuint_load( agen& a1); // WU_P`
	`vcharx vchar_load_transp( agen& a1); // B_T`
	`vshortx vshort_load_transp( agen& a1); // H_T`
	`vintx  vint_load_transp(  agen& a1); // W_T`
	`vcharx vuchar_load_transp( agen& a1); // BU_T`
	`vshortx vushort_load_transp(agen& a1); // HU_T`
	`vintx  vuint_load_transp( agen& a1); // WU_T`
	`vcharx vchar_load_bs( agen& a1); // B_S`
	`vshortx vshort_load_hs( agen& a1); // H_S`
	`vintx  vint_load_ws(  agen& a1); // W_S`
	`vcharx vuchar_load_bs( agen& a1); // BU_S`
	`vshortx vushort_load_hs(agen& a1); // HU_S`
	`vintx  vuint_load_ws( agen& a1); // WU_S`
	`vcharx vchar_load_c2( agen& a1); // B_C2`
	`vshortx vshort_load_c2( agen& a1); // H_C2`
	`vintx  vint_load_c2(  agen& a1); // W_C2`
	`vcharx vuchar_load_c2( agen& a1); // BU_C2`
	`vshortx vushort_load_c2(agen& a1); // HU_C2`
	`vintx  vuint_load_c2( agen& a1); // WU_C2`

Instruction name	VLD agen

	vcharx vcharx_load(agen& a1); // WX
	vshortx vshortx_load(agen& a1); // WX
	vintx vintx_load(agen& a1); // WX
	
	// predicated
	void vchar_load(vcharx& dst, agen& a1, bool pred); // B_P
	void vshort_load(vshortx& dst, agen& a1, bool pred); // H_P
	void vint_load(vintx& dst, agen& a1, bool pred); // W_P
	void vuchar_load(vcharx& dst, agen& a1, bool pred); // BU_P
	void vushort_load(vshortx& dst, agen& a1, bool pred); // HU_P
	void vuint_load(vintx& dst, agen& a1, bool pred); // WU_P
	void vchar_load_transp(vcharx& dst, agen& a1,bool pred);//B_T
	void vshort_load_transp(vshortx& dst, agen& a1,bool pred);//H_T
	void vint_load_transp(vintx& dst, agen& a1,bool pred);//W_T
	void vuchar_load_transp(vcharx& dst, agen& a1,bool pred);//BU_T
	void vushort_load_transp(vshortx& dst, agen& a1,bool pred);//HU_T
	void vuint_load_transp(vintx& dst, agen& a1,bool pred);//WU_T
	void vchar_load_bs(vcharx& dst, agen& a1, bool pred); //B_S
	void vshort_load_hs(vshortx& dst, agen& a1, bool pred); //H_S
	void vint_load_ws(vintx& dst, agen& a1, bool pred); //W_S
	void vuchar_load_bs(vcharx& dst, agen& a1, bool pred); //BU_S
	void vushort_load_hs(vshortx& dst, agen& a1, bool pred); //HU_S
	void vuint_load_ws(vintx& dst, agen& a1, bool pred); //WU_S
	void vchar_load_c2(vcharx& dst, agen& a1, bool pred); //B_C2
	void vshort_load_c2(vshortx& dst, agen& a1, bool pred); //H_C2
	void vint_load_c2(vintx& dst, agen& a1, bool pred); //W_C2
	void vuchar_load_c2(vcharx& dst, agen& a1, bool pred); //BU_C2
	void vushort_load_c2(vshortx& dst, agen& a1, bool pred); //HU_C2
	void vuint_load_c2(vintx& dst, agen& a1, bool pred); //WU_C2
	void vcharx_load(vcharx& dst, agen& a1, bool pred); //WX
	void vshortx_load(vshortx& dst, agen& a1, bool pred); //WX
	void vintx_load(vintx& dst, agen& a1, bool pred); //WX
	
	// Float
	vfloatx vfloat_load(agen& a); // W_P
	vfloatx vfloat_load_transp(agen& a); // W_T
	vfloatx vfloat_load_ws(agen& a); // W_S
	vfloatx vfloat_load_c2(agen& a); // W_C2
	void vfloat_load(vfloatx& dst, agen& a, bool pred);// W_P
	void vfloat_load_transp(vfloatx& dst, agen& a, bool pred);// W_T
	void vfloat_load_ws(vfloatx& dst, agen& a, bool pred);// W_S
	void vfloat_load_c2(vfloatx& dst, agen& a, bool pred);// W_C2
	vhfloatx vhfloat_load(agen& a); // H_P
	vhfloatx vhfloat_load_transp(agen& a); // H_T
	vhfloatx vhfloat_load_hs(agen& a); // H_S
	vhfloatx vhfloat_load_c2(agen& a); // H_C2
	void vhfloat_load(vhfloatx& dst, agen& a, bool pred);//H_P
	void vhfloat_load_transp(vhfloatx& dst, agen& a, bool pred);//H_T

Instruction name	VLD agen

	`void vhfloat_load_hs(  vhfloatx& dst, agen& a, bool pred);//H_S`
	`void vhfloat_load_c2(  vhfloatx& dst, agen& a, bool pred);//H_C2`

Additional details	Use Agen to supply address; address is post-modified according to multi-dimensional (up to 6D) address modifier scheme. When predication is off, writing to Vdst is skipped. See Transposing Load/Store for address calculation and pattern for transpose distribution.

9.9.4.3 DVLD Agen

Instruction name	DVLD agen
Functionality	Double vector load agen-based
Assembly format	<pred> DVLD<type>_<distr> *A<id>++, DVdst <pred> DVLD<type>_<distr> *A<id>++, DWdst pred = none, [P2.. P15]
Type and bit width	Type/distribution supported: B_P, H_P, W_P, BU_P, HU_P, WU_P, H_T, W_T, HU_T, WU_T, B_PDI, H_PDI, W_PDI, BU_PDI, HU_PDI, WU_PDI, H_TDI, W_TDI, HU_TDI, WU_TDI, BH_P, BW_P, HW_P, BHU_P, BWU_P, HWU_P, BH_T, BW_T, HW_T, BHU_T, BWU_T, HWU_T BH_PDI, BW_PDI, HW_PDI, BHU_PDI, BWU_PDI, HWU_PDI, H_T2DI, HU_T2DI, W_T2DI, WU_T2DI, B_T32, BU_T32, H_T2, HU_T2, H_T4, HU_T4, H_T8, HU_T8, H_T16, HU_T16, W_T8, WU_T8 For example: DVLDB_P *A0++, V0:V1 [P1] DVLDH_T, *A1++, V2:V3
Predication	Instruction-level predication
Source options	
Destination options	Vdst: double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`// unpredicated`
	`dvcharx dvchar_load( agen& a1); // B_P`
	`dvshortx dvshort_load( agen& a1); // H_P`
	`dvintx  dvint_load(  agen& a1); // W_P`
	`dvcharx dvuchar_load( agen& a1); // BU_P`
	`dvshortx dvushort_load(agen& a1); // HU_P`

Instruction name	DVLD agen

	dvintx dvuint_load(agen& a1); // WU_P
	dvshortx dvshort_load_transp(agen& a1); // H_T
	dvintx dvint_load_transp(agen& a1); // W_T
	dvshortx dvushort_load_transp(agen& a1); // HU_T
	dvintx dvuint_load_transp(agen& a1); // WU_T
	dvcharx dvchar_load_di(agen& a1); // B_PDI
	dvshortx dvshort_load_di(agen& a1); // H_PDI
	dvintx dvint_load_di(agen& a1); // W_PDI
	dvcharx dvuchar_load_di(agen& a1); // BU_PDI
	dvshortx dvushort_load_di(agen& a1); // HU_PDI
	dvintx dvuint_load_di(agen& a1); // WU_PDI
	dvshortx dvshort_load_transp_di(agen& a1); // H_TDI
	dvintx dvint_load_transp_di(agen& a1); // W_TDI
	dvshortx dvushort_load_transp_di(agen& a1); // HU_TDI
	dvintx dvuint_load_transp_di(agen& a1); // WU_TDI
	
	dvshortx vchar_dvshortx_load(agen& a1); // BH_P
	dvintx vchar_dvintx_load(agen& a1); // BW_P
	dvintx vshort_dvintx_load(agen& a1); // HW_P
	dvshortx vuchar_dvshortx_load(agen& a1); // BHU_P
	dvintx vuchar_dvintx_load(agen& a1); // BWU_P
	dvintx vushort_dvintx_load(agen& a1); // HWU_P
	dvshortx vchar_dvshortx_load_transp(agen& a1); // BH_T
	dvintx vchar_dvintx_load_transp(agen& a1); // BW_T
	dvintx vshort_dvintx_load_transp(agen& a1); // HW_T
	dvshortx vuchar_dvshortx_load_transp(agen& a1); // BHU_T
	dvintx vuchar_dvintx_load_transp(agen& a1); // BWU_T
	dvintx vushort_dvintx_load_transp(agen& a1); // HWU_T
	dvshortx vchar_dvshortx_load_di(agen& a1); // BH_PDI
	dvintx vchar_dvintx_load_di (agen& a1); // BW_PDI
	dvintx vshort_dvintx_load_di (agen& a1); // HW_PDI
	dvshortx vuchar_dvshortx_load_di (agen& a1); // BHU_PDI
	dvintx vuchar_dvintx_load_di (agen& a1); // BWU_PDI
	dvintx vushort_dvintx_load_di (agen& a1); // HWU_PDI
	
	dvshortx dvshort_load_transp2_di(agen& a1); // H_T2DI
	dvshortx dvushort_load_transp2_di(agen& a1); // HU_T2DI
	dvintx dvint_load_transp2_di(agen& a1); // W_T2DI
	dvintx dvuint_load_transp2_di(agen& a1); // WU_T2DI
	dvcharx dvchar_load_transp32(agen& a1); // B_T32
	dvshortx dvshort_load_transp2(agen& a1); // H_T2
	dvshortx dvshort_load_transp4(agen& a1); // H_T4
	dvshortx dvshort_load_transp8(agen& a1); // H_T8
	dvshortx dvshort_load_transp16(agen& a1); // H_T16
	dvintx dvint_load_transp8(agen& a1); // W_T8
	dvcharx dvuchar_load_transp32(agen& a1); // BU_T32
	dvshortx dvushort_load_transp2(agen& a1); // HU_T2
	dvshortx dvushort_load_transp4(agen& a1); // HU_T4

Instruction name	DVLD agen

	dvshortx dvushort_load_transp8(agen& a1); // HU_T8
	dvshortx dvushort_load_transp16(agen& a1); // HU_T16
	dvintx dvuint_load_transp8(agen& a1); // WU_T8
	
	// predicated
	void dvchar_load(dvcharx& dst, agen& a1, bool pred); // B_P
	void dvshort_load(dvshortx& dst, agen& a1, bool pred); // H_P
	void dvint_load(dvintx& dst, agen& a1, bool pred); // W_P
	void dvuchar_load(dvcharx& dst, agen& a1, bool pred); // BU_P
	void dvushort_load(dvshortx& dst, agen& a1, bool pred); // HU_P
	void dvuint_load(dvintx& dst, agen& a1, bool pred); // WU_P
	void dvshort_load_transp(dvshortx& dst, agen& a1, bool pred); // H_T
	void dvint_load_transp(dvintx& dst, agen& a1, bool pred); // W_T
	void dvushort_load_transp(dvshortx& dst, agen& a1, bool pred); // HU_T
	void dvuint_load_transp(dvintx& dst, agen& a1, bool pred); // WU_T
	void dvchar_load_di(dvcharx& dst, agen& a1, bool pred); // B_PDI
	void dvshort_load_di(dvshortx& dst, agen& a1, bool pred); // H_PDI
	void dvint_load_di(dvintx& dst, agen& a1, bool pred); // W_PDI
	void dvuchar_load_di(dvcharx& dst, agen& a1, bool pred); // BU_PDI
	void dvushort_load_di(dvshortx& dst, agen& a1, bool pred); // HU_PDI
	void dvuint_load_di(dvintx& dst, agen& a1, bool pred); // WU_PDI
	void dvshort_load_transp_di(dvshortx& dst, agen& a1,bool pred); // H_TDI
	void dvint_load_transp_di(dvintx& dst, agen& a1,bool pred); // W_TDI
	void dvushort_load_transp_di(dvshortx& dst, agen& a1,bool pred); // HU_TDI
	void dvuint_load_transp_di(dvintx& dst, agen& a1,bool pred); // WU_TDI
	
	void vchar_dvshortx_load(dvshortx& dst, agen& a1, bool pred); // BH_P
	void vchar_dvintx_load(dvintx& dst, agen& a1, bool pred); // BW_P
	void vshort_dvintx_load(dvintx& dst, agen& a1, bool pred); // HW_P
	void vuchar_dvshortx_load(dvshortx& dst, agen& a1, bool pred); // BHU_P
	void vuchar_dvintx_load(dvintx& dst, agen& a1, bool pred); // BWU_P
	void vushort_dvintx_load(dvintx& dst, agen& a1, bool pred); // HWU_P
	void vchar_dvshortx_load_transp(dvshortx& dst, agen& a, bool p); //BH_T
	void vchar_dvintx_load_transp(dvintx& dst, agen& a, bool p); //BW_T
	void vshort_dvintx_load_transp(dvintx& dst, agen& a, bool p); //HW_T
	void vuchar_dvshortx_load_transp(dvshortx& dst, agen& a, bool p); //BHU_T
	void vuchar_dvintx_load_transp(dvintx& dst, agen& a, bool p); //BWU_T
	void vushort_dvintx_load_transp(dvintx& dst, agen& a, bool p); //HWU_T
	void vchar_dvshortx_load_di(dvshortx& dst,agen& a1,bool pred); //BH_PDI
	void vchar_dvintx_load_di(dvintx& dst,agen& a1,bool pred); //BW_PDI
	void vshort_dvintx_load_di(dvintx& dst,agen& a1,bool pred); //HW_PDI
	void vuchar_dvshortx_load_di(dvshortx& dst,agen& a1,bool pred); //BHU_PDI
	void vuchar_dvintx_load_di(dvintx& dst,agen& a1,bool pred); //BWU_PDI
	void vushort_dvintx_load_di(dvintx& dst,agen& a1,bool pred); //HWU_PDI
	
	void dvshort_load_transp2_di(dvshortx& dst, agen& a, bool p); // H_T2DI
	void dvushort_load_transp2_di(dvshortx& dst, agen& a, bool p); // HU_T2DI
	void dvint_load_transp2_di(dvintx& dst, agen& a, bool p); // W_T2DI

Instruction name	DVLD agen

	void dvuint_load_transp2_di(dvintx& dst, agen& a, bool p); // WU_T2DI
	void dvchar_load_transp32(dvcharx& dst, agen& a1, bool pred); //B_T32
	void dvshort_load_transp2(dvshortx& dst, agen& a1, bool pred); //H_T2
	void dvshort_load_transp4(dvshortx& dst, agen& a1, bool pred); //H_T4
	void dvshort_load_transp8(dvshortx& dst, agen& a1, bool pred); //H_T8
	void dvshort_load_transp16(dvshortx& dst, agen& a1, bool pred); //H_T16
	void dvint_load_transp8(dvintx& dst, agen& a1, bool pred); //W_T8
	void dvuchar_load_transp32(dvcharx& dst, agen& a1, bool pred); //BU_T32
	void dvushort_load_transp2(dvshortx& dst, agen& a1, bool pred); //HU_T2
	void dvushort_load_transp4(dvshortx& dst, agen& a1, bool pred); //HU_T4
	void dvushort_load_transp8(dvshortx& dst, agen& a1, bool pred); //HU_T8
	void dvushort_load_transp16(dvshortx& dst, agen& a1, bool pred); //HU_T16
	void dvuint_load_transp8(dvintx& dst, agen& a1, bool pred); //WU_T8
	
	// Float & Hfloat
	dvfloatx dvfloat_load(agen& a1); // W_P
	dvfloatx dvfloat_load_transp(agen& a1); // W_T
	dvfloatx dvfloat_load_di(agen& a1); // W_PDI
	dvfloatx dvfloat_load_transp_di(agen& a1); // W_TDI
	dvfloatx dvfloat_load_transp2_di(agen& a1); // W_T2DI
	dvfloatx dvfloat_load_transp8(agen& a1); // W_T8
	
	void dvfloat_load(dvfloatx& dst, agen& a, bool p); // W_P
	void dvfloat_load_transp(dvfloatx& dst, agen& a, bool p); // W_T
	void dvfloat_load_di(dvfloatx& dst, agen& a, bool p); // W_PDI
	void dvfloat_load_transp_di(dvfloatx& dst, agen& a, bool p); // W_TDI
	void dvfloat_load_transp2_di(dvfloatx& dst, agen& a, bool p); // W_T2DI
	void dvfloat_load_transp8(dvfloatx& dst, agen& a, bool p); // W_T8
	
	dvhfloatx dvhfloat_load(agen& a1); // H_P
	dvhfloatx dvhfloat_load_transp(agen& a1); // H_T
	dvhfloatx dvhfloat_load_di(agen& a1); // H_PDI
	dvhfloatx dvhfloat_load_transp_di(agen& a1); // H_TDI
	dvhfloatx dvhfloat_load_transp2(agen& a1); // H_T2
	dvhfloatx dvhfloat_load_transp2_di(agen& a1); // H_T2DI
	dvhfloatx dvhfloat_load_transp4(agen& a1); // H_T4
	dvhfloatx dvhfloat_load_transp8(agen& a1); // H_T8
	dvhfloatx dvhfloat_load_transp16(agen& a1); // H_T16
	
	void dvhfloat_load(dvhfloatx& dst, agen& a, bool p); // H_P
	void dvhfloat_load_transp(dvhfloatx& dst, agen& a, bool p); // H_T
	void dvhfloat_load_di(dvhfloatx& dst, agen& a, bool p); // H_PDI
	void dvhfloat_load_transp_di(dvhfloatx& dst, agen& a, bool p); // H_TDI
	void dvhfloat_load_transp2(dvhfloatx& dst, agen& a, bool p); // H_T2
	void dvhfloat_load_transp2_di(dvhfloatx& dst, agen& a, bool p); // H_T2DI
	void dvhfloat_load_transp4(dvhfloatx& dst, agen& a, bool p); // H_T4
	void dvhfloat_load_transp8(dvhfloatx& dst, agen& a, bool p); // H_T8
	void dvhfloat_load_transp16(dvhfloatx& dst, agen& a, bool p); // H_T16

Instruction name	DVLD agen
Additional details	Use Agen to supply address; address is post-modified according to multi-dimensional (up to 6D) address modifier scheme. When predication is off, writing to Vdst is skipped. Please see Transposing Load/Store for address calculation and pattern for transpose distributions. For Byte type loads, the starting address is aligned to 16-bit in order to access 64 bytes of data with 32 memory banks.

9.9.4.4 VST Agen

Instruction name	VST agen
Functionality	Vector store agen-based
Assembly format	<pred> VST<type>_<distr> Vsrc, *A<id>++ pred = none, [P2.. P15], [V0..V15] <pred> VST<type>_<distr> ACsrc, *A<id>++ <pred> VST<type>_<distr> XACsrc, *A<id>++ pred = none, [P2..P15]
Type and bit width	For VRF source, predicate register and VRF predication are supported for these type/distributions: B_P, H_P, W_P, B_T, H_T, W_T, BH_P, HW_P, BH_T, HW_T, WX_P For VRF source, only predicate register predication is supported for these type/distributions: B_S, H_S, W_S For ARF source, predication through predicate register is supported for these type/distributions: B_P, H_P, W_P, B_T, H_T, W_T, B_S, H_S, W_S, BH_P, HW_P, BH_T, HW_T, WX_P For XAC source, predication through predicate register is supported and only with W_P type/distribution. In addition, rounding/saturation operations are bypassed. Note that WX_P distribution is predicated as 8 lanes x 48-bit (versus 16 lanes x 24-bit or 32 lanes x 12-bit). For example: VSTB_P V0, *A0++ [P1] VSTH_T V2, *A1++
Predication	Per-lane predication
Source options	Vsrc: single vector register in VRF, ARF, or XARF
Destination options	
Additional options	
Intrinsics/operator	`// unpredicated`

Instruction name	VST agen

	void vstore(vcharx vec1, agen& a1); // B_P
	void vstore(vshortx vec1, agen& a1); // H_P
	void vstore(vintx vec1, agen& a1); // W_P
	void vstore(xvshortx vec1, agen& a1); // W_P for XAC
	void vstore_bh(vcharx vec1, agen& a1); // BH_P
	void vstore_hw(vshortx vec1, agen& a1); // HW_P
	void vstore_transp(vcharx vec1, agen& a1); // B_T
	void vstore_transp(vshortx vec1, agen& a1); // H_T
	void vstore_transp(vintx vec1, agen& a1); // W_T
	void vstore_transp_bh(vcharx vec1, agen& a1); // BH_T
	void vstore_transp_hw(vshortx vec1, agen& a1); // HW_T
	void vstore_bs(vcharx vec1, agen& a1); // B_S
	void vstore_hs(vshortx vec1, agen& a1); // H_S
	void vstore_ws(vintx vec1, agen& a1); // W_S
	void vstore_ext(vcharx vec1, agen& a1); // WX_P
	void vstore_ext(vshortx vec1, agen& a1); // WX_P
	void vstore_ext(vintx vec1, agen& a1); // WX_P
	
	// predicate register per-lane predicated
	void vstore(vcharx vec1, agen& a1, int pred); // B_P
	void vstore(vshortx vec1, agen& a1, int pred); // H_P
	void vstore(vintx vec1, agen& a1, int pred); // W_P
	void vstore(xvshortx vec1, agen& a1, int pred); // W_P for XAC
	void vstore_bh(vcharx vec1, agen& a1, int pred); // BH_P
	void vstore_hw(vshortx vec1, agen& a1, int pred); // HW_P
	void vstore_transp(vcharx vec1, agen& a1, int pred); // B_T
	void vstore_transp(vshortx vec1, agen& a1, int pred); // H_T
	void vstore_transp(vintx vec1, agen& a1, int pred); // W_T
	void vstore_transp_bh(vcharx vec1, agen& a1, int pred); //BH_T
	void vstore_transp_hw(vshortx vec1, agen& a1, int pred); //HW_T
	void vstore_bs(vcharx vec1, agen& a1, int pred); // B_S
	void vstore_hs(vshortx vec1, agen& a1, int pred); // H_S
	void vstore_ws(vintx vec1, agen& a1, int pred); // W_S
	
	void vstore_ext(vcharx vec1, agen& a1, int pred); // WX_P
	void vstore_ext(vshortx vec1, agen& a1, int pred); // WX_P
	void vstore_ext(vintx vec1, agen& a1, int pred); // WX_P
	
	// Note that vstore_ext() for vcharx and vshortx are predicated
	// as 8 x 48-bit lanes, like for vintx, as opposed to 32 x 12-
	// bit lanes or 16 x 24-bit lanes. vstore_ext() for the 3
	// vector types are mapped to the same instruction.
	
	// VRF per-lane predicated
	void vstore(vcharx vec1, agen& a1, vcharx pred); // B_P
	void vstore(vshortx vec1, agen& a1, vshortx pred); // H_P
	void vstore(vintx vec1, agen& a1, vintx pred); // W_P
	void vstore_bh(vcharx vec1, agen& a1, vcharx pred); // BH_P

Instruction name	VST agen

	`void vstore_hw(vshortx vec1, agen& a1, vshortx pred); //HW_P`
	`void vstore_transp(  vcharx v1, agen& a1, vcharx p);//B_T`
	`void vstore_transp(  vshortx v1, agen& a1, vshortx p);//H_T`
	`void vstore_transp(  vintx  v1, agen& a1, vintx  p);//W_T`
	`void vstore_transp_bh(vcharx v1, agen& a1, vcharx p);//BH_T`
	`void vstore_transp_hw(vshortx v1, agen& a1, vshortx p);//HW_T`
	`void vstore_ext(   vintx  v1, agen& a1, vintx  p);//WX_P`
	
	`// vstore_ext(vcharx v1, agen& a1, vcharx p) and`
	`// vstore_ext(vshortx v1, agen& a1, vshortx p) are not`
	`// supported as we cannot support appropriate predicate`
	`// datatype for per-lane predication.`
	
	`// Float`
	`void vstore(    vfloatx vec1, agen& a1); // W_P`
	`void vstore_transp(vfloatx vec1, agen& a1); // W_T`
	`void vstore_ws(  vfloatx vec1, agen& a1); // W_S`
	`void vstore(    vfloatx vec1, agen& a1, int  pred); // W_P`
	`void vstore_transp(vfloatx vec1, agen& a1, int  pred); // W_T`
	`void vstore_ws(  vfloatx vec1, agen& a1, int  pred); // W_S`
	`void vstore(    vfloatx vec1, agen& a1, vintx pred); // W_P`
	`void vstore_transp(vfloatx vec1, agen& a1, vintx pred); // W_T`
	
	`void vstore(    vhfloatx vec1, agen& a1); // H_P`
	`void vstore_transp(vhfloatx vec1, agen& a1); // H_T`
	`void vstore_hs(  vhfloatx vec1, agen& a1); // H_S`
	`void vstore(    vhfloatx vec1, agen& a1, int   pred);//H_P`
	`void vstore_transp(vhfloatx vec1, agen& a1, int   pred);//H_T`
	`void vstore_hs(  vhfloatx vec1, agen& a1, int   pred);//H_S`
	`void vstore(    vhfloatx vec1, agen& a1, vshortx pred);//H_P`
	`void vstore_transp(vhfloatx vec1, agen& a1, vshortx pred);//H_T`

Additional details	Use Agen to supply address; address is post-modified according to multi-dimensional (up to 6D) address modifier scheme. Per-lane predicated. When predication is off, writing to specific memory object is skipped. Address updates are always carried out. Consumes lower K bits of Preg or a single VRF for K-lane predication. See 9.5.3.4 for details. Please see Transposing Load/Store for address calculation and pattern for transpose distribution. Per-lane predication via vector register is only available in the M0 slot, and is NOT supported for scalar distribution.

9.9.4.5 DVST Agen

Instruction name	DVST agen
Functionality	Double vector store agen-based
Assembly format	<pred> DVST<type>_<distr> DVsrc, *A<id>++ pred = none, [P2.. P15], [V0..V15] <pred> DVST<type>_<distr> DACsrc, *A<id>++ <pred> DVST<type>_<distr> DXACsrc, *A<id>++ pred = none, [P2.. P15]
Type and bit width	For double VRF source, predicate register and VRF predication are supported with type/distribution: B_P, H_P, W_P, H_T, W_T, B_PI, H_PI, W_PI, H_TI, W_TI, HB_P, WH_P, HB_T, WH_T, HB_PI, WH_PI, HB_TI, WH_TI, W_T2, W_T2I, In addition, for double VRF source, only predicate register is supported with type/distribution: B_S, H_S, W_S, B_T32, H_T2, H_T2I, H_T4, H_T8, H_T16, W_T8 For double ARF source, predication through predicate register is supported with type/distribution: B_P, H_P, W_P, H_T, W_T, B_S, H_S, W_S, B_PI, H_PI, W_PI, H_TI, W_TI, HB_P, WH_P, HB_T, WH_T, HB_PI, WH_PI, HB_TI, WH_TI, B_T32, H_T2, H_T2I, H_T4, H_T8, H_T16, W_T2, W_T2I, W_T8 For double XAC source, predication through predicate register is supported and only with WH_PI type/distribution. In addition, rounding/saturation operations are bypassed, and it's available in the M0 slot. For example: DVSTB_P V0:V1, *A0++ [P1] DVSTH_T V2:V3, *A1++
Predication	Per-lane predication
Source options	DVsrc: double vector register in VRF, ARF, or ARF + XRF (together 32-bit per Halfword lane)
Destination options	
Additional options	

Intrinsics/operator	`// unpredicated`
	`void vstore(  dvcharx vec1, agen& a1); // B_P`
	`void vstore(  dvshortx vec1, agen& a1); // H_P`
	`void vstore(  dvintx  vec1, agen& a1); // W_P`

Instruction name	DVST agen

	void vstore_hb(dvshortx vec1, agen& a1); // HB_P
	void vstore_wh(dvintx vec1, agen& a1); // WH_P
	void vstore_transp(dvshortx vec1, agen& a1); // H_T
	void vstore_transp(dvintx vec1, agen& a1); // W_T
	void vstore_transp_hb(dvshortx vec1, agen& a1); // HB_T
	void vstore_transp_wh(dvintx vec1, agen& a1); // WH_T
	
	void vstore_i(dvcharx vec1, agen& a1); // B_PI
	void vstore_i(dvshortx vec1, agen& a1); // H_PI
	void vstore_i(dvintx vec1, agen& a1); // W_PI
	void vstore_i_hb(dvshortx vec1, agen& a1); // HB_PI
	void vstore_i_wh(dvintx vec1, agen& a1); // WH_PI
	void vstore_i(dxvshortx vec1, agen& a1); // WH_PI DXAC
	void vstore_transp_i(dvshortx vec1, agen& a1); // H_TI
	void vstore_transp_i(dvintx vec1, agen& a1); // W_TI
	void vstore_transp_i_hb(dvshortx vec1, agen& a1); // HB_TI
	void vstore_transp_i_wh(dvintx vec1, agen& a1); // WH_TI
	void vstore_bs(dvcharx vec1, agen& a1); // B_S
	void vstore_hs(dvshortx vec1, agen& a1); // H_S
	void vstore_ws(dvintx vec1, agen& a1); // W_S
	
	void vstore_transp32(dvcharx vec1, agen& a1); // B_T32
	void vstore_transp2(dvshortx vec1, agen& a1); // H_T2
	void vstore_transp2_i(dvshortx vec1, agen& a1); // H_T2I
	void vstore_transp4(dvshortx vec1, agen& a1); // H_T4
	void vstore_transp8(dvshortx vec1, agen& a1); // H_T8
	void vstore_transp16(dvshortx vec1, agen& a1); // H_T16
	void vstore_transp2(dvintx vec1, agen& a1); // W_T2
	void vstore_transp2_i(dvintx vec1, agen& a1); // W_T2I
	void vstore_transp8(dvintx vec1, agen& a1); // W_T8
	
	// per-lane predicated via predicate register
	void vstore(dvcharx vec1, agen& a1, dpred pred); // B_P
	void vstore(dvshortx vec1, agen& a1, int pred); // H_P
	void vstore(dvintx vec1, agen& a1, int pred); // W_P
	void vstore_hb(dvshortx vec1, agen& a1, int pred); // HB_P
	void vstore_wh(dvintx vec1, agen& a1, int pred); // WH_P
	void vstore_transp(dvshortx vec1, agen& a1, int pred);//H_T
	void vstore_transp(dvintx vec1, agen& a1, int pred);//W_T
	void vstore_transp_hb(dvshortx vec1, agen& a1, int pred);//HB_T
	void vstore_transp_wh(dvintx vec1, agen& a1, int pred);//WH_T
	
	void vstore_i(dvcharx vec1, agen& a1, dpred pred); // B_PI
	void vstore_i(dvshortx vec1, agen& a1, int pred); // H_PI
	void vstore_i(dvintx vec1, agen& a1, int pred); // W_PI
	void vstore_i_hb(dvshortx vec1, agen& a1, int pred); //HB_PI
	void vstore_i_wh(dvintx vec1, agen& a1, int pred); //WH_PI
	void vstore_i(dxvshortx vec1, agen& a1, int pred); //WH_PI DXAC

Instruction name	DVST agen

	void vstore_transp_i(dvshortx vec1, agen& a1, int pred); //H_TI
	void vstore_transp_i(dvintx vec1, agen& a1, int pred); //W_TI
	void vstore_transp_i_hb(dvshortx vec1, agen& a1, int pred); //HB_TI
	void vstore_transp_i_wh(dvintx vec1, agen& a1, int pred); //WH_TI
	void vstore_bs(dvcharx vec1, agen& a1, int pred); // B_S
	void vstore_hs(dvshortx vec1, agen& a1, int pred); // H_S
	void vstore_ws(dvintx vec1, agen& a1, int pred); // W_S
	void vstore_transp32(dvcharx vec1, agen& a1, dpred pred); //B_T32
	void vstore_transp2(dvshortx vec1, agen& a1, int pred); //H_T2
	void vstore_transp2_i(dvshortx vec1, agen& a1, int pred); //H_T2I
	void vstore_transp4(dvshortx vec1, agen& a1, int pred); //H_T4
	void vstore_transp8(dvshortx vec1, agen& a1, int pred); //H_T8
	void vstore_transp16(dvshortx vec1, agen& a1, int pred); //H_T16
	void vstore_transp2(dvintx vec1, agen& a1, int pred); //W_T2
	void vstore_transp2_i(dvintx vec1, agen& a1, int pred); //W_T2I
	void vstore_transp8(dvintx vec1, agen& a1, int pred); //W_T8
	
	// per-lane predicated via VRF
	void vstore(dvcharx vec1, agen& a1, vcharx pred); // B_P
	void vstore(dvshortx vec1, agen& a1, vcharx pred); // H_P
	void vstore(dvintx vec1, agen& a1, vshortx pred); // W_P
	void vstore_hb(dvshortx vec1, agen& a1, vcharx pred); // HB_P
	void vstore_wh(dvintx vec1, agen& a1, vshortx pred); // WH_P
	void vstore_transp(dvshortx vec1, agen& a1, vcharx pred);//H_T
	void vstore_transp(dvintx vec1, agen& a1, vshortx pred);//W_T
	void vstore_transp_hb(dvshortx vec1, agen& a1, vcharx pred);//HB_T
	void vstore_transp_wh(dvintx vec1, agen& a1, vshortx pred);//WH_T
	void vstore_i(dvcharx vec1, agen& a1, vcharx pred); // B_PI
	void vstore_i(dvshortx vec1, agen& a1, vcharx pred); // H_PI
	void vstore_i(dvintx vec1, agen& a1, vshortx pred); // W_PI
	void vstore_i_hb(dvshortx vec1, agen& a1, vcharx pred); // HB_PI
	void vstore_i_wh(dvintx vec1, agen& a1, vshortx pred); // WH_PI
	void vstore_transp_i(dvshortx v1, agen& a1, vcharx p); // H_TI
	void vstore_transp_i(dvintx v1, agen& a1, vshortx p); // W_TI
	void vstore_transp_i_hb(dvshortx v1, agen& a1, vcharx p); // HB_TI
	void vstore_transp_i_wh(dvintx v1, agen& a1, vshortx p); // WH_TI
	void vstore_transp2 (dvintx v1, agen& a1, vshortx p); //W_T2
	void vstore_transp2_i(dvintx v1, agen& a1, vshortx p); //W_T2I
	
	// Float (basically leveraging H and W type stores)
	void vstore(dvhfloatx vec1, agen& a1); // H_P
	void vstore_transp(dvhfloatx vec1, agen& a1); // H_T
	void vstore_i(dvhfloatx vec1, agen& a1); // H_PI
	void vstore_transp_i(dvhfloatx vec1, agen& a1); // H_TI
	void vstore_hs(dvhfloatx vec1, agen& a1); // H_S
	void vstore_transp2(dvhfloatx vec1, agen& a1); // H_T2
	void vstore_transp2_i(dvhfloatx vec1, agen& a1); // H_T2I
	void vstore_transp4(dvhfloatx vec1, agen& a1); // H_T4

Instruction name	DVST agen

	void vstore_transp8(dvhfloatx vec1, agen& a1); // H_T8
	void vstore_transp16(dvhfloatx vec1, agen& a1); // H_T16
	void vstore(dvhfloatx vec1, agen& a1, int pred); // H_P
	void vstore_transp(dvhfloatx vec1, agen& a1, int pred); // H_T
	void vstore_i(dvhfloatx vec1, agen& a1, int pred); // H_PI
	void vstore_transp_i(dvhfloatx vec1, agen& a1, int pred); // H_TI
	void vstore_hs(dvhfloatx vec1, agen& a1, int pred); // H_S
	void vstore_transp2(dvhfloatx vec1, agen& a1, int pred); // H_T2
	void vstore_transp2_i(dvhfloatx vec1, agen& a1, int pred); // H_T2I
	void vstore_transp4(dvhfloatx vec1, agen& a1, int pred); // H_T4
	void vstore_transp8(dvhfloatx vec1, agen& a1, int pred); // H_T8
	void vstore_transp16(dvhfloatx vec1, agen& a1, int pred); // H_T16
	void vstore(dvhfloatx vec1, agen& a1, vcharx p); // H_P
	void vstore_transp(dvhfloatx vec1, agen& a1, vcharx p); // H_T
	void vstore_i(dvhfloatx vec1, agen& a1, vcharx p); // H_PI
	void vstore_transp_i(dvhfloatx vec1, agen& a1, vcharx p); // H_TI
	
	void vstore(dvfloatx vec1, agen& a1); // W_P
	void vstore_transp(dvfloatx vec1, agen& a1); // W_T
	void vstore_i(dvfloatx vec1, agen& a1); // W_PI
	void vstore_transp_i(dvfloatx vec1, agen& a1); // W_TI
	void vstore_ws(dvfloatx vec1, agen& a1); // W_S
	void vstore_transp2(dvfloatx vec1, agen& a1);// W_T2
	void vstore_transp2_i(dvfloatx vec1, agen& a1);// W_T2I
	void vstore_transp8(dvfloatx vec1, agen& a1);// W_T8
	void vstore(dvfloatx vec1, agen& a1, int pred); // W_P
	void vstore_transp(dvfloatx vec1, agen& a1, int pred); // W_T
	void vstore_i(dvfloatx vec1, agen& a1, int pred); // W_PI
	void vstore_transp_i(dvfloatx vec1, agen& a1, int pred); // W_TI
	void vstore_ws(dvfloatx vec1, agen& a1, int pred); // W_S
	void vstore_transp2(dvfloatx vec1, agen& a1, int pred); // W_T2
	void vstore_transp2_i(dvfloatx vec1, agen& a1, int pred); // W_T2I
	void vstore_transp8(dvfloatx vec1, agen& a1, int pred); // W_T8
	void vstore(dvfloatx vec1, agen& a1, vshortx p); // W_P
	void vstore_transp(dvfloatx vec1, agen& a1, vshortx p); // W_T
	void vstore_i(dvfloatx vec1, agen& a1, vshortx p); // W_PI
	void vstore_transp_i(dvfloatx vec1, agen& a1, vshortx p); // W_TI
	void vstore_transp2 (dvfloatx vec1, agen& a1, vshortx p); // W_T2
	void vstore_transp2_i(dvfloatx vec1, agen& a1, vshortx p); // W_T2I

Additional details	Use Agen to supply address; address is post-modified according to multi-dimensional (up to 6D) address modifier scheme. Per-lane predicated. When predication is off, writing to specific memory object is skipped. Address updates are always carried out. Consumes lower K bits of Preg or a single VRF for K-lane predication. For transposition distribution, each element is separately predicated, so that DVSTW_T2 requires 16 predication bits, just like DVSTW_P and DVSTW_T. See Lane Predication for Agen-Based Vector Store for additional details.

Instruction name	DVST agen
	See Transposing Load/Store for address calculation and pattern for transpose distributions. Per-lane predication via vector register is only available in the M0 slot, and is NOT supported for scalar distribution.

9.9.4.6 QVST Agen

Instruction name	QVST agen
Functionality	Quad vector store agen-based
Assembly format	<pred> QVST<type>_distr DVsrc1, DVsrc2, *A<id>++ <pred> QVST<type>_distr DACsrc1, DACsrc2, *A<id>++ pred = none, [P2.. P15]
Type and bit width	Type/distribution supported for quad vector VRF and ARF source: HB_P, HB_PI, HB_PI2 WH_P, WH_PI, WH_PI2, WH_T, WH_TI
Predication	Per-lane predication
Source options	two double vector registers all in VRF or ARF
Destination options	
Additional options	

Intrinsics/operator	`// unpredicated`
	`void vstore(dvshortx v1, dvshortx v2, agen& a); // HB_P`
	`void vstore(dvintx v1, dvintx v2, agen& a); // WH_P`
	`void vstore_i(dvshortx v1, dvshortx v2, agen& a); // HB_PI`
	`void vstore_i(dvintx v1, dvintx v2, agen& a); // WH_PI`
	`void vstore_i2(dvshortx v1, dvshortx v2, agen& a); // HB_PI2`
	`void vstore_i2(dvintx v1, dvintx v2, agen& a); // WH_PI2`
	`void vstore_transp(dvintx v1, dvintx v2, agen& a); // WH_T`
	`void vstore_transp_i(dvintx v1, dvintx v2, agen& a); // WH_TI`
	
	`// per-lane predicated`
	`void vstore(dvshortx v1, dvshortx v2, agen& a, dpred p); //HB_P`
	`void vstore(dvintx v1, dvintx v2, agen& a, int p); //WH_P`
	`void vstore_i(dvshortx v1, dvshortx v2, agen& a, dpred p); //HB_PI`
	`void vstore_i(dvintx v1, dvintx v2, agen& a, int p); //WH_PI`
	`void vstore_i2(dvshortx v1, dvshortx v2, agen& a, dpred p);//HB_PI2`
	`void vstore_i2(dvintx v1, dvintx v2, agen& a, int p); //WH_PI2`
	`void vstore_transp( dvintx v1, dvintx v2, agen& a, int p); //WH_T`
	`void vstore_transp_i(dvintx v1, dvintx v2, agen& a, int p); //WH_TI`

Additional details	Use Agen to supply address; address is post-modified according to multi-dimensional (up to 6D) address modifier scheme. Per-lane predicated. When predication is off, writing to specific memory object is skipped. Consumes lower K bits of Preg for K-lane predication. Address update is always carried out.

Instruction name	QVST agen
	See Transposing Load/Store for address calculation and pattern for T transpose distribution.

9.9.4.7 VLDPerm Agen

Instruction name	VLDPerm agen
Functionality	Vector load + permute agen-based
Assembly format	VLDPerm<type>_<distr> *A<id>++, Vsrc/Wsrc, Vdst/Wdst
Type and bit width	Type/distribution available: HB_P, HBU_P, For example: VLDPermHB_P *A0++, V2, V1
Predication	Not available
Source options	Vsrc: single vector register in VRF or WRF specifying permutation pattern
Destination options	Vdst/Wdst: single vector register in VRF or WRF
Additional options	

Intrinsics/operator	`vcharx vchar_load_perm( agen& agen1, vshortx src);  // HB_P`
	`vcharx vuchar_load_perm(agen& agen1, vshortx src);  // HBU_P`

Additional details	Use Agen to supply address; address is post-modified according to multi-dimensional (up to 6D) address modifier scheme. HB_P/HBU_P case: 512-bit is read from memory and treated like a 32 entries x 16-bit table. From the single Halfword vector source, 5 LSBs of each of 16 Halfword lanes, bits 4:0, are used to index the table to return 16 x 16-bit permutation outcome. **Next higher bit, bit 5, is used to conditionally replace outcome with zero when the bit is set.** Then, the 16 x 16-bit data is repartitioned as 32 x 8-bit and expanded every 8-bit into 12-bit of space in the destination vector register, with sign-extension performed for HB_P case, and zero-extension performed for HBU_P case. For example, in case of VLDPermHBU_P (unsigned version), say memory location pointed by the agen address contains these halfwords: 0x0123, 0x4567, 0x89AB, 0xCDEF, … and Vsrc read as Halfword lanes contains: 0x1, 0x0, 0x2, 0x3, … First the memory word would be permuted into 0x4567, 0x0123, 0x89AB, 0xCDEF, … then byte by byte extended into 0x067, 0x045, 0x023, 0x001, 0x0AB, 0x089, 0x0EF, 0x0CD … in the destination vector register. The same memory and Vsrc contents with VLDPermHB_P (signed) would return 0x067, 0x045, 0x023, 0x001, 0xFAB, 0xF89, 0xFEF, 0xFCD … VLDPerm is supported in all 3 memory slots.

9.9.4.8 DVLDPerm Agen

Instruction name	DVLDPerm agen
Functionality	Double vector load + permute agen-based
Assembly format	DVLDPerm<type>_<distr> *A<id>++, Vsrc/Wsrc, DVdst/DWdst
Type and bit width	Type/distribution available: H_P, W_P, HU_P, WU_P, H_T, W_T, HU_T, WU_T, HB_P, HBU_P W_T2, WU_T2 For example: DVLDPermH_P *A0++, V2, V0:V1
Predication	Not available
Source options	Vsrc: single vector register in VRF or WRF specifying permutation pattern
Destination options	DVdst/DWdst: double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`dvshortx dvshort_load_perm( agen& agen1, vcharx src);  // H_P`
	`dvintx  dvint_load_perm(  agen& agen1, vshortx src);  // W_P`
	`dvshortx dvushort_load_perm(agen& agen1, vcharx src);  // HU_P`
	`dvintx  dvuint_load_perm( agen& agen1, vshortx src);  // WU_P`
	
	`dvshortx dvshort_load_perm_transp( agen& agen1, vcharx src);//H_T`
	`dvintx  dvint_load_perm_transp(  agen& agen1, vshortx src);//W_T`
	`dvshortx dvushort_load_perm_transp(agen& agen1, vcharx src);//HU_T`
	`dvintx  dvuint_load_perm_transp( agen& agen1, vshortx src);//WU_T`
	
	`dvcharx dvchar_load_perm( agen& agen1, vcharx src);  // HB_P`
	`dvcharx dvuchar_load_perm(agen& agen1, vcharx src);  // HBU_P`
	
	`dvintx  dvint_load_perm_transp2( agen& agen1, vshortx src);//W_T2`
	`dvintx  dvuint_load_perm_transp2(agen& agen1, vshortx src);//WU_T2`
	
	`//Float`
	`dvfloatx dvfloat_load_perm(  agen& agen1, vshortx src);`
	`dvfloatx dvfloat_load_perm_transp( agen& agen1, vshortx src);`
	`dvfloatx dvfloat_load_perm_transp2(agen& agen1, vshortx src);`
	`dvhfloatx dvhfloat_load_perm(  agen& agen1, vcharx src);`
	`dvhfloatx dvhfloat_load_perm_transp(agen& agen1, vcharx src);`

Additional details	Use Agen to supply address; address is post-modified according to multi-dimensional (up to 6D) address modifier scheme. Since we are returning double vector destination using single vector lane selection source, we are using a smaller data type as lane selection than the destination.

Instruction name	DVLDPerm agen
	Word destination (W_P/WU_P/W_T/WU_T/W_T2/WU_T2): 512-bit is read from memory (consecutively for P distribution, transposed from 16 x 32-bit for T/T2 distribution) and treated like a 16 entries x 32-bit table. From the single vector source, 4 LSBs of each of 16 **Halfword** lanes, bits 3:0, are used to index the table to return 16 x 32-bit permutation outcome. **Next higher bit, bit 4, is used to conditionally replace outcome with zero when the bit is set.** The permutation outcome, 16 x 32-bit, is zero or sign extended to the **Word** type double vector destination according to signed (W_P/W_T) or unsigned (WU_P/WU_T) designation. **Halfword** destination (H_P/HU_P/H_T/HU_T): 512-bit is read from memory (consecutively for P distribution, transposed from 32 x 16-bit for T distribution) and treated like a 32 entries x 16-bit table. From the single vector source, 5 LSBs of each of 32 **Byte** lanes, bits 4:0, are used to index the table to return 32 x 16-bit permutation outcome. **Next higher bit, bit 5, is used to conditionally replace outcome with zero when the bit is set.** The permutation outcome, 32 x 16-bit, is zero or sign extended to the **Halfword** type double vector destination according to signed (H_P/H_T) or unsigned (HU_P/HU_T) designation. Permuted as **Halfword** into **Byte** destination (HB_P/HBU_P): 512-bit is read from memory consecutively (as only P distribution is supported) and treated like a 32 entries x 16-bit table. From the single vector source, 5 LSBs of each of 32 **Byte** lanes, bits 4:0, are used to index the table to return 32 x 16-bit permutation outcome. **Next higher bit, bit 5, is used to conditionally replace outcome with zero when the bit is set.** The permutation outcome, 32 x 16-bit, is **repartitioned** as **Byte** lanes, 64 x 8-bit, and then zero or sign extended into the Byte type double vector destination according to signed (HB_P) or unsigned (HBU_P) designation. Ordering of lanes for destinations shall be .lo components first then .hi components. In other words, DVLDPermH_P with {0, 1, 2, ..., 31} selection data shall behave like "_P" distributed double vector load, DVLDH_P. DVLDPerm is supported in all 3 memory slots. See Transposing Load/Store for address calculation and pattern for transpose distributions.

9.9.4.9 DVST_PLROUND Agen

Instruction name	DVST_PLROUND agen
Functionality	Double vector store agen-based with per-lane rounding
Assembly format	DVST <type>_PLRound_distr Vsrc1/Wsrc1, DVsrc2/DACsrc2, *A<id>++
Type and bit width	Type/distribution available: H_P, W_P, HB_P, WH_P, WB_P, H_PI, W_PI, HB_PI, WH_PI, WB_PI

Instruction name	DVST_PLROUND agen
	For example: DVSTH_PLRound_P W0, V0:V1, *A0++
Predication	Not supported
Source options	src1: single vector register in VRF/WRF to carry rounding information src2: double vector register in VRF/ARF to carry data
Destination options	
Additional options	

Intrinsics/operator	`void vstore_plround(  vcharx s1, dvshortx s2, agen& a1); // H_P`
	`void vstore_plround(  vshortx s1, dvintx  s2, agen& a1); // W_P`
	`void vstore_hb_plround(vcharx s1, dvshortx s2, agen& a1); // HB_P`
	`void vstore_wh_plround(vshortx s1, dvintx  s2, agen& a1); // WH_P`
	`void vstore_wb_plround(vshortx s1, dvintx  s2, agen& a1); // WB_P`
	`void vstore_i_plround( vcharx s1, dvshortx s2, agen& a1); // H_PI`
	`void vstore_i_plround( vshortx s1, dvintx  s2, agen& a1); // W_PI`
	`void vstore_hbi_plround(vcharx s1, dvshortx s2, agen& a1); // HB_PI`
	`void vstore_whi_plround(vshortx s1, dvintx  s2, agen& a1); // WH_PI`
	`void vstore_wbi_plround(vshortx s1, dvintx  s2, agen& a1); // WB_PI`

Additional details	Use Agen to supply address; address is post-modified according to multi-dimensional (up to 6D) address modifier scheme. Because we use a single vector to provide rounding parameters for storing of a double vector, in the intrinsic functions, source 1 single vector data type is half the size of the source 2 double vector data type, so we can match number of lanes. Use 8 LSBs of source 1 to supply rounding/truncation parameters, overriding the rounding/truncation configuration from Agen. Bit 7 indicates rounding (0) vs truncation (1). Bits 6:0 specifies number of bits to round/truncate. When number of bits to round/truncate exceeds the data source (src2) lane bit width, outcome is 0 for rounding any value, truncating any non-negative value, and -1 for truncating any negative value. Note that rounding/truncation information in the single vector source 1 is ordered sequentially as stored data in memory. For example, for Word type, with the P distribution option, pairing of two sources are: src1[0] – src2.lo[0], src1[1] – src2.lo[1], ..., src1[7] – src2.lo[7], src1[8] – src2.hi[0], src1[9] – src2.hi[1], ..., src1[15] – src2.hi[7]. With the PI distribution option, pairing of two sources are: src1[0] – src2.lo[0], src1[1] – src2.hi[0], src1[2] – src2.lo[1], src1[3] – src2.hi[1], ..., src1[14] – src2.lo[7], src1[15] – src2.hi[7]. Per-lane rounding vector store is only available in the M0 slot.

9.9.5 Agen-Based Scalar Load/Store

9.9.5.1 Instruction Summary

Scalar load/store instructions:

Table 42. Agen-based scalar load/store instructions

Function	Assembly Format	Comments
Scalar load agen-based	LD<type> *A<id>++, Rdst Type available: B, BU, H, HU, W	
Dual scalar load agen-based	DLD<type> *A<id>++, Rdst1, Rdst2 Type available: B, BU, H, HU, W	
Scalar store agen-based	<pred> ST<type> Rsrc, *A<id>++ pred = none, [P2.. P15], instruction level predication Type available: B, H, W	
Dual scalar store agen-based	<pred> DST<type> Rsrc1, Rsrc2, *A<id>++ pred = none, [P2.. P15], instruction level predication Type available: B, H, W	

Agen features supported and not supported for scalar load/store:

- Distribution: not supported; dual register accesses consecutive items in memory
- Type promotion/demotion: not supported; only single data type
- Multi-dimensional addressing: supported
- Circular buffer addressing: supported
- Lane offset/transposition: not supported
- Rounding: not supported
- Saturation: not supported

9.9.5.2 LD Agen

Instruction name	LD agen
Functionality	Scalar load agen-based
Assembly format	LD<type> *A<id>++, Rdst
Type and bit width	Type available: B, BU, H, HU, W
Predication	not available

Instruction name	LD agen
Source options	
Destination options	Single scalar register
Additional options	

Intrinsics/operator	`int int_load(agen& agen1);`
	`unsigned int uint_load(agen& agen1);`
	`short short_load(agen& agen1);`
	`unsigned short ushort_load(agen& agen1);`
	`char char_load(agen& agen1);`
	`unsigned char uchar_load(agen& agen1);`
	`float float_load(agen& agen1);`
	`hfloat hfloat_load(agen& agen1);`

Additional details	Use Agen to supply address; address is post-modified according to multi-dimensional (up to 6D) address modifier scheme.

9.9.5.3 DLD Agen

Instruction name	DLD agen
Functionality	Dual scalar load agen-based
Assembly format	DLD<type> *A<id>++, Rdst1, Rdst2
Type and bit width	Type available: B, BU, H, HU, W
Predication	not available
Source options	
Destination options	Two scalar registers
Additional options	

Intrinsics/operator	`void int_load(agen& agen1, int &dst1, int &dst2);`
	`void uint_load(agen& agen1, uint &dst1, uint &dst2);`
	`void short_load(agen& agen1, short &dst1, short &dst2);`
	`void ushort_load(agen& agen1, unsigned short &dst1, unsigned short &dst2`
	`void char_load(agen& agen1, char &dst1, char &dst2);`
	`void uchar_load(agen& agen1, unsigned char &dst1, unsigned char &dst2);`
	`void float_load(agen& agen1, float &dst1, float &dst2);`
	`void hfloat_load(agen& agen1, hfloat &dst1, hfloat &dst2);`

Additional details	Use Agen to supply address; address is post-modified according to multi-dimensional (up to 6D) address modifier scheme. Two successive items in memory pointed by the Agen are loaded, the first item into Rdst1, the second item into Rdst2.

9.9.5.4 ST Agen

Instruction name	ST agen
Functionality	Scalar store agen-based
Assembly format	<pred> ST<type> Rsrc, *A<id>++
Type and bit width	Type available: B, H, W
Predication	Instruction level predication
Source options	Single scalar register
Destination options	
Additional options	

Intrinsics/operator	`// unpredicated`
	`void int_store(int src, agen& agen1);`
	`void short_store(short src, agen& agen1);`
	`void char_store(char src, agen& agen1);`
	`void uint_store(unsigned int src, agen& agen1);`
	`void ushort_store(unsigned short src, agen& agen1);`
	`void uchar_store(unsigned char src, agen& agen1);`
	`void float_store(float src, agen& agen1);`
	`void hfloat_store(hfloat src, agen& agen1);`
	
	`// predicated`
	`void int_store(int src, agen& agen1, bool pred);`
	`void short_store(short src, agen& agen1, bool pred);`
	`void char_store(char src, agen& agen1, bool pred);`
	`void uint_store(unsigned int src, agen& agen1, bool pred);`
	`void ushort_store(unsigned short src, agen& agen1, bool pred);`
	`void uchar_store(unsigned char src, agen& agen1, bool pred);`
	`void float_store(float src, agen& agen1, bool pred);`
	`void hfloat_store(hfloat src, agen& agen1, bool pred);`

Additional details	Use Agen to supply address; address is post-modified according to multi-dimensional (up to 6D) address modifier scheme. Note that when compile-time-constant 0 is used on the predicate argument, the intrinsic would be compiled into update_agen(), which is equivalent in functionality.

9.9.5.5 DST Agen

Instruction name	DST agen
Functionality	Dual scalar store agen-based
Assembly format	<pred> ST<type> Rsrc1, Rsrc2, *A<id>++
Type and bit width	Type available: B, H, W
Predication	Instruction level predication

Instruction name	DST agen
Source options	Two scalar registers
Destination options	
Additional options	

Intrinsics/operator	`// unpredicated`
	`void int_store(int src1, int src2, agen& agen1);`
	`void short_store(short src1, short src2, agen& agen1);`
	`void char_store(char src1, char src2, agen& agen1);`
	`void uint_store(unsigned int src1, unsigned int src2, agen& agen1);`
	`void ushort_store(unsigned short src1, unsigned short src2, agen& agen1);`
	`void uchar_store(unsigned char src1, unsigned char src2, agen& agen1);`
	`void float_store(float src1, float src2, agen& a1);`
	`void hfloat_store(hfloat src1, hfloat src2, agen& a1);`
	
	`// predicated`
	`void int_store(int src1, int src2, agen& agen1, bool pred);`
	`void short_store(short src1, short src2, agen& agen1, bool pred);`
	`void char_store(char src1, char src2, agen& agen1, bool pred);`
	`void uint_store(unsigned int src1, unsigned int src2, agen& agen1, bool pred);`
	`void ushort_store(unsigned short src1, unsigned short src2, agen& agen1, bool pred);`
	`void uchar_store(unsigned char src1, unsigned char src2, agen& agen1, bool pred);`
	`void float_store(float src1, float src2, agen& a1, bool pred);`
	`void hfloat_store(hfloat src1, hfloat src2, agen& a1, bool pred);`

Additional details	Use Agen to supply address; address is post-modified according to multi-dimensional (up to 6D) address modifier scheme. Two successive items are store to memory pointed by the Agen, the first item from Rsrc1, the second item from Rsrc2. Note that when compile-time-constant 0 is used on the predicate argument, the intrinsic would be compiled into update_agen(), which is equivalent in functionality.

9.9.6 Parallel Lookup, Histogram, Vector Addressed Store

9.9.6.1 Instruction Summary

Instructions for lookup, histogram and vector addressed store are shown as follows.

Table 43 Parallel lookup, histogram, vector addressed store instructions

Function	Assembly Format	Comments
Parallel lookup	DVLUT_<type-parallelism> *(Rbase+DVsrc/DWsrc), DVdst/DWdst type-parallelism = {32H, 32HU, 16W, 16WU}	Rbase should be 64-byte aligned, bits 5..0 are ignored. Use DVsrc as indices.
	VLUT_<type-parallelism> *(Rbase+Vsrc), Vdst type-parallelism = {32/16/8/4/2/1 B/BU, 16/8/4/2/1 H/HU, 8/4/2/1 W/WU} VLUT_<type-parallelism> *(Rbase+DVsrc), Vdst type-parallelism = {32HB, 32HBU} VLUT_<type-parallelism> *(Rbase+Vsrc), Vdst type-parallelism = {16/8/4/2/1HB, 16/8/4/2/1HBU}	Rbase should be 64-byte aligned, bits 5..0 are ignored. Use Vsrc as indices.
Parallel 2-point lookup	DVLUT_2pt_<type-parallelism> *(Rbase+Vsrc), DVdst type-parallelism = {16/8/4/2/1 B/BU, 16/8/4/2/1 H/HU, 8/4/2/1 W/WU, 16/8/4/2/1 HB/HBU}	Lookup table[index] and table[index+1] and return a double vector
Parallel 2x2-pt lookup	DVLUT_2x2pt_<type-parallelism> *(Rbase+Vsrc), DVdst/DWdst type-parallelism = {8/4/2/1 B/BU, 8/4/2/1 H/HU, 4/2/1 W/WU, 8/4/2/1 HB/HBU }	Lookup table[index], table[index+1], table[line_pitch + index], table[line_pitch + index + 1] in a double vector
Parallel histogram	DVHist_<type-parallelism> *(Rbase+DVsrc1), DVsrc2, DVdst DVHist_<type-parallelism> *(Rbase+DVsrc1), DVsrc2 type-parallelism = {32H, 16W}	Rbase should be 64-byte aligned, bits 5..0 are ignored. Use DVsrc1 as indices, DVsrc2 as update (additive) values. Optionally return bin value before the update in DVdst
	VHist_<type-parallelism> *(Rbase+Vsrc1), Vsrc2, Vdst VHist_<type-parallelism> *(Rbase+Vsrc1), Vsrc2 type-parallelism = {16/8/4/2/1 H, 8/4/2/1 W}	Use Vsrc1 as indices, Vsrc2 as update (additive) values. Optionally return bin value before the update in Vdst
Parallel OR histogram	DVHist_OR_<type-parallelism> *(Rbase+DVsrc1), DVsrc2, DVdst DVHist_OR_<type-parallelism> *(Rbase+DVsrc1), DVsrc2 type-parallelism = {32H, 16W}	Perform bitwise OR operation instead of addition

Function	Assembly Format	Comments
	VHist_OR_<type-parallelism> *(Rbase+Vsrc1), Vsrc2, Vdst VHist_OR_<type-parallelism> *(Rbase+Vsrc1), Vsrc2 type-parallelism = {16/8/4/2/1H, 8/4/2/1W}	Perform bitwise OR operation instead of addition
Vector addressed store	DVAST_<type-parallelism> DVsrc, *(Rbase+DVidx) type-parallelism = {32H, 16W}	Rbase should be 64-byte aligned, bits 5..0 are ignored. Use DVidx as indices, DVsrc as data to write.

9.9.6.2 DVLUT

Instruction name	DVLUT
Functionality	Double vector lookup
Assembly format	DVLUT_<type-parallelism> *(Rbase+DVsrc/DWsrc), DVdst/DWdst
Type and bit width	type-parallelism = {32H, 32HU, 16W, 16WU} Same type applies to indices and table entries, but indices are always signed even when unsigned type is used. Table entries are signed or unsigned indicated in the type. For example: DVLUT_16W *(R4 + V0:V1), V2:V3
Predication	Not available
Source options	Base address: scalar register Index: double vector register in VRF or WRF
Destination options	Double vector register in VRF or WRF
Additional options	

Intrinsics/operator	`dvshortx vlookup_32h( const short* tbl,  dvshortx idx);`
	`dvshortx vlookup_32hu(cont unsigned short* tbl, dvshortx idx);`
	`dvhfloatx vlookup_32hf(const hfloat* tbl,  dvshortx idx);`
	`dvintx  vlookup_16w( const int*  tbl,  dvintx idx);`
	`dvintx  vlookup_16wu(const unsigned int* tbl, dvintx idx);`
	`dvfloatx vlookup_16f( const float* tbl,  dvintx idx);`

Additional details	Use double vector to supply indices to lookup parallel tables. Rbase is forced to be 64-byte aligned by ignoring its bits 5:0. Refer to Table Lookup for index bit width used in address calculation.

9.9.6.3 VLUT

Instruction name	VLUT
Functionality	Single vector lookup

Instruction name	VLUT
Assembly format	VLUT_<type-parallelism> *(Rbase+Vsrc), Vdst
Type and bit width	type-parallelism = {32/16/8/4/2/1B, 32/16/8/4/2/1BU, 16/8/4/2/1H, 16/8/4/2/1HU, 8/4/2/1W, 8/4/2/1WU} Same type applies to indices and table entries, but indices are always signed even when unsigned type is used. Table entries are signed or unsigned indicated in the type. For example: VLUT_4W *(R4 + V0), V2
Predication	Not available
Source options	Rbase: scalar register Vsrc: single vector register
Destination options	Vdst: single vector register
Additional options	

Intrinsics/operator	`vcharx vlookup_32b( const char* tbl, vcharx idx);`
	`vcharx vlookup_32bu(const unsigned char* tbl, vcharx idx);`
	`vcharx vlookup_16b( const char* tbl, vcharx idx);`
	`vcharx vlookup_16bu(const unsigned char* tbl, vcharx idx);`
	`vcharx vlookup_8b( const char* tbl, vcharx idx);`
	`vcharx vlookup_8bu( const unsigned char* tbl, vcharx idx);`
	`vcharx vlookup_4b( const char* tbl, vcharx idx);`
	`vcharx vlookup_4bu( const unsigned char* tbl, vcharx idx);`
	`vcharx vlookup_2b( const char* tbl, vcharx idx);`
	`vcharx vlookup_2bu( const unsigned char* tbl, vcharx idx);`
	`vcharx vlookup_1b( const char* tbl, vcharx idx);`
	`vcharx vlookup_1bu( const unsigned char* tbl, vcharx idx);`
	
	`vshortx vlookup_16h(const short* tbl, vshortx idx);`
	`vshortx vlookup_16hu(const unsigned short* tbl,vshortx idx);`
	`vshortx vlookup_8h(const short* tbl, vshortx idx);`
	`vshortx vlookup_8hu(const unsigned short* tbl, vshortx idx);`
	`vshortx vlookup_4h(const short* tbl, vshortx idx);`
	`vshortx vlookup_4hu(const unsigned short* tbl, vshortx idx);`
	`vshortx vlookup_2h(const short* tbl, vshortx idx);`
	`vshortx vlookup_2hu(const unsigned short* tbl, vshortx idx);`
	`vshortx vlookup_1h(const short* tbl, vshortx idx);`
	`vshortx vlookup_1hu(const unsigned short* tbl, vshortx idx);`
	
	`vintx vlookup_8w( const int* tbl, vintx idx);`
	`vintx vlookup_8wu(const unsigned int* tbl, vintx idx);`
	`vintx vlookup_4w( const int* tbl, vintx idx);`
	`vintx vlookup_4wu(const unsigned int* tbl, vintx idx);`
	`vintx vlookup_2w( const int* tbl, vintx idx);`
	`vintx vlookup_2wu(const unsigned int* tbl, vintx idx);`
	`vintx vlookup_1w( const int* tbl, vintx idx);`

Instruction name	VLUT

	`vintx vlookup_1wu(const unsigned int* tbl, vintx idx);`
	
	`vfloatx vlookup_8f(const float* tbl, vintx idx);`
	`vfloatx vlookup_4f(const float* tbl, vintx idx);`
	`vfloatx vlookup_2f(const float* tbl, vintx idx);`
	`vfloatx vlookup_1f(const float* tbl, vintx idx);`
	
	`vhfloatx vlookup_16hf(const hfloat* tbl, vshortx idx);`
	`vhfloatx vlookup_8hf( const hfloat* tbl, vshortx idx);`
	`vhfloatx vlookup_4hf( const hfloat* tbl, vshortx idx);`
	`vhfloatx vlookup_2hf( const hfloat* tbl, vshortx idx);`
	`vhfloatx vlookup_1hf( const hfloat* tbl, vshortx idx);`

Additional details	Use first K lanes of a single vector to supply K indices to lookup K parallel tables. Rbase is forced to be 64-byte aligned by ignoring its bits 5:0. Returned table entries are placed on the first K lanes of the destination vector register. Remaining lanes, if any, are returned 0. Refer to Table Lookup for index bit width used in address calculation.

Instruction name	VLUT (looking up bytes with halfword indices)
Functionality	Single vector lookup
Assembly format	VLUT_<type-parallelism> *(Rbase+Vsrc), Vdst VLUT_<type-parallelism> *(Rbase+DVsrc), Vdst
Type and bit width	type-parallelism = {32/16/8/4/2/1HB, 32/16/8/4/2/1HBU} The first type letter indicates type of indices; indices are always signed. The second type letter indicates type of table entries including signed/unsigned In case of 32-way parallel lookup, 32 short indices require a double vector source. For other parallelism, a single vector source is used. For example: VLUT_4HB *(R4 + V0), V2 VLUT_32HB *(R4 + V0:V1), V2
Predication	Not available
Source options	Rbase: scalar register Vsrc: single vector register (1 ~ 16-way) DVsrc: double vector register (32-way)
Destination options	Vdst: single vector register
Additional options	

Intrinsics/operator	`vcharx vlookup_32hb( const char* tbl,dvshortx idx);`
	`vcharx vlookup_32hbu(const unsigned char* tbl,dvshortx idx);`
	`vcharx vlookup_16hb( const char* tbl, vshortx idx);`
	`vcharx vlookup_16hbu(const unsigned char* tbl, vshortx idx);`
	`vcharx vlookup_8hb( const char* tbl, vshortx idx);`
	`vcharx vlookup_8hbu( const unsigned char* tbl, vshortx idx);`
	`vcharx vlookup_4hb( const char* tbl, vshortx idx);`
	`vcharx vlookup_4hbu( const unsigned char* tbl, vshortx idx);`
	`vcharx vlookup_2hb( const char* tbl, vshortx idx);`

Instruction name	VLUT (looking up bytes with halfword indices)

	`vcharx vlookup_2hbu( const unsigned char* tbl, vshortx idx);`
	`vcharx vlookup_1hb( const char* tbl, vshortx idx);`
	`vcharx vlookup_1hbu( const unsigned char* tbl, vshortx idx);`

Additional details	Use first K lanes of a single vector to supply K indices to lookup K parallel tables. Rbase is forced to be 64-byte aligned by ignoring its bits 5:0. Returned table entries are placed on the first K lanes of the destination vector register. Remaining lanes, if any, are returned 0. In the case of 32-way parallel byte lookup with double short vector indices, the .lo component of double vector supplies the first 16 indices, the .hi component of double vector supplies the last 16 indices. Refer to Table Lookup for index bit width used in address calculation.

9.9.6.4 DVLUT_2PT

Instruction name	DVLUT_2PT
Functionality	Double vector two-point lookup
Assembly format	DVLUT_2pt_<type-parallelism> *(Rbase+Vsrc), DVdst
Type and bit width	type-parallelism = {16/8/4/2/1 B/BU, 16/8/4/2/1 H/HU, 8/4/2/1 W/WU} Same type applies to indices and table entries, but indices are always signed even when unsigned type is used. Table entries are signed or unsigned indicated in the type. For example: DVLUT_2pt_16HU *(R4 + V0), V2:V3
Predication	Not available
Source options	Rbase: scalar register Vsrc: single vector register
Destination options	DVdst: double vector register
Additional options	

Intrinsics/operator	`dvcharx vlookup_2pt_16b( const char* tbl, vcharx idx);`
	`dvcharx vlookup_2pt_16bu(const unsigned char* tbl, vcharx idx);`
	`dvcharx vlookup_2pt_8b( const char* tbl, vcharx idx);`
	`dvcharx vlookup_2pt_8bu( const unsigned char* tbl, vcharx idx);`
	`dvcharx vlookup_2pt_4b( const char* tbl, vcharx idx);`
	`dvcharx vlookup_2pt_4bu( const unsigned char* tbl, vcharx idx);`
	`dvcharx vlookup_2pt_2b( const char* tbl, vcharx idx);`
	`dvcharx vlookup_2pt_2bu( const unsigned char* tbl, vcharx idx);`
	`dvcharx vlookup_2pt_1b( const char* tbl, vcharx idx);`
	`dvcharx vlookup_2pt_1bu( const unsigned char* tbl, vcharx idx);`
	
	`dvshortx vlookup_2pt_16h( const short* tbl, vshortx idx);`
	`dvshortx vlookup_2pt_16hu(const unsigned short* tbl, vshortx idx);`
	`dvshortx vlookup_2pt_8h( const short* tbl, vshortx idx);`
	`dvshortx vlookup_2pt_8hu( const unsigned short* tbl, vshortx idx);`
	`dvshortx vlookup_2pt_4h( const short* tbl, vshortx idx);`

Instruction name	DVLUT_2PT

	dvshortx vlookup_2pt_4hu(const unsigned short* tbl, vshortx idx);
	dvshortx vlookup_2pt_2h(const short* tbl, vshortx idx);
	dvshortx vlookup_2pt_2hu(const unsigned short* tbl, vshortx idx);
	dvshortx vlookup_2pt_1h(const short* tbl, vshortx idx);
	dvshortx vlookup_2pt_1hu(const unsigned short* tbl, vshortx idx);
	
	dvintx vlookup_2pt_8w(const int* tbl, vintx idx);
	dvintx vlookup_2pt_8wu(const unsigned int* tbl, vintx idx);
	dvintx vlookup_2pt_4w(const int* tbl, vintx idx);
	dvintx vlookup_2pt_4wu(const unsigned int* tbl, vintx idx);
	dvintx vlookup_2pt_2w(const int* tbl, vintx idx);
	dvintx vlookup_2pt_2wu(const unsigned int* tbl, vintx idx);
	dvintx vlookup_2pt_1w(const int* tbl, vintx idx);
	dvintx vlookup_2pt_1wu(const unsigned int* tbl, vintx idx);

Additional details	Use first P lanes of a single vector to supply indices to look up P parallel tables (P = parallelism). Rbase is forced to be 64-byte aligned by ignoring its bits 5:0. Table[index] and the next entry in the table per parallel table are returned in the low and high registers, respectively, in the first P lanes. Remaining lanes are returned as zero. Note that the parallelism indicates number of parallel sub-tables we have. Number of data points returned is twice as many, as we look up 2 data points from each subtable. Refer to Table Lookup for index bit width used in address calculation.

For example, DVLUT_2pt_8W returns 2 data points from each of 8 subtables. Layout of an 8-way-parallel word-type table and picking up data points via index vector {0, 1, 2, 3, 4, 5, 4, 3}:

T0[0]	T0[1]	T1[0]	T1[1]	T2[0]	T2[1]	T3[0]	T3[1]	T4[0]	T4[1]	T5[0]	T5[1]	T6[0]	T6[1]	T7[0]	T7[1]
T0[2]	T0[3]	T1[2]	T1[3]	T2[2]	T2[3]	T3[2]	T3[3]	T4[2]	T4[3]	T5[2]	T5[3]	T6[2]	T6[3]	T7[2]	T7[3]
T0[4]	T0[5]	T1[4]	T1[5]	T2[4]	T2[5]	T3[4]	T3[5]	T4[4]	T4[5]	T5[4]	T5[5]	T6[4]	T6[5]	T7[4]	T7[5]
T0[6]	T0[7]	T1[6]	T1[7]	T2[6]	T2[7]	T3[6]	T3[7]	T4[6]	T4[7]	T5[6]	T5[7]	T6[6]	T6[7]	T7[6]	T7[7]

Instruction name	DVLUT_2PT (looking up bytes with halfword indices)
Functionality	Double vector two-point lookup
Assembly format	DVLUT_2pt_<type-parallelism> *(Rbase+Vsrc), DVdst
Type and bit width	type-parallelism = {16/8/4/2/1 HB/HBU} The first type letter indicates type of indices; indices are always signed. The second type letter indicates type of table entries including signed/unsigned For example: DVLUT_2pt_4HB *(R4 + V0), V2:V3
Predication	Not available
Source options	Rbase: scalar register Vsrc: single vector register
Destination options	DVdst: double vector register

Instruction name	DVLUT_2PT (looking up bytes with halfword indices)
Additional options	

Intrinsics/operator	`dvcharx vlookup_2pt_16hb( const char* tbl, vshortx idx);`
	`dvcharx vlookup_2pt_16hbu( const unsigned char* tbl, vshortx idx);`
	`dvcharx vlookup_2pt_8hb( const char* tbl, vshortx idx);`
	`dvcharx vlookup_2pt_8hbu( const unsigned char* tbl, vshortx idx);`
	`dvcharx vlookup_2pt_4hb( const char* tbl, vshortx idx);`
	`dvcharx vlookup_2pt_4hbu( const unsigned char* tbl, vshortx idx);`
	`dvcharx vlookup_2pt_2hb( const char* tbl, vshortx idx);`
	`dvcharx vlookup_2pt_2hbu( const unsigned char* tbl, vshortx idx);`
	`dvcharx vlookup_2pt_1hb( const char* tbl, vshortx idx);`
	`dvcharx vlookup_2pt_1hbu( const unsigned char* tbl, vshortx idx);`

Additional details	Use single vector to supply indices to lookup parallel tables in the first P lanes (P = parallelism). Rbase is forced to be 64-byte aligned by ignoring its bits 5:0. Table[index] and the next entry in the table per parallel table are returned in the first P lanes respectively in the low and high parts of destination double register. Remaining lanes are returned as zero. Note that the parallelism indicates number of parallel sub-tables we have. Number of data points returned is twice as many, as we look up 2 data points from each subtable. Refer to Table Lookup for index bit width used in address calculation.

9.9.6.5 DVLUT_2X2PT

Instruction name	DVLUT_2X2PT
Functionality	Double vector two-by-two-point lookup
Assembly format	DVLUT_2x2pt_<type-parallelism> *(Rbase+Vsrc), DVdst/DWdst
Type and bit width	type-parallelism = {8/4/2/1B/BU, 8/4/2/1H/HU, 4/2/1W/WU} Same type applies to indices and table entries, but indices are always signed even when unsigned type is used. Table entries are signed or unsigned indicated in the type. For example: DVLUT_2x2pt_2W *(R4 + V0), V2:V3
Predication	Not available
Source options	Rbase: scalar register Vsrc: single vector register Implicit PL scalar register to derive line pitch
Destination options	DVdst: double vector register
Additional options	

Intrinsics/operator	`dvcharx vlookup_2x2pt_8b( const char* tbl, vcharx idx, int k);`
	`dvcharx vlookup_2x2pt_8bu(const unsigned char* tbl, vcharx idx, int k);`
	`dvcharx vlookup_2x2pt_4b( const char* tbl, vcharx idx, int k);`
	`dvcharx vlookup_2x2pt_4bu(const unsigned char* tbl, vcharx idx, int k);`
	`dvcharx vlookup_2x2pt_2b( const char* tbl, vcharx idx, int k);`

Instruction name	DVLUT_2X2PT

	dvcharx vlookup_2x2pt_2bu(const unsigned char* tbl, vcharx idx, int k);
	dvcharx vlookup_2x2pt_1b(const char* tbl, vcharx idx, int k);
	dvcharx vlookup_2x2pt_1bu(const unsigned char* tbl, vcharx idx, int k);
	
	dvshortx vlookup_2x2pt_8h(const short* tbl, vshortx idx, int k);
	dvshortx vlookup_2x2pt_8hu(const unsigned short* tbl, vshortx idx, int k);
	dvshortx vlookup_2x2pt_4h(const short* tbl, vshortx idx, int k);
	dvshortx vlookup_2x2pt_4hu(const unsigned short* tbl, vshortx idx, int k);
	dvshortx vlookup_2x2pt_2h(const short* tbl, vshortx idx, int k);
	dvshortx vlookup_2x2pt_2hu(const unsigned short* tbl, vshortx idx, int k);
	dvshortx vlookup_2x2pt_1h(const short* tbl, vshortx idx, int k);
	dvshortx vlookup_2x2pt_1hu(const unsigned short* tbl, vshortx idx, int k);
	
	dvintx vlookup_2x2pt_4w(const int* tbl, vintx idx, int k);
	dvintx vlookup_2x2pt_4wu(const unsigned int* tbl, vintx idx, int k);
	dvintx vlookup_2x2pt_2w(const int* tbl, vintx idx, int k);
	dvintx vlookup_2x2pt_2wu(const unsigned int* tbl, vintx idx, int k);
	dvintx vlookup_2x2pt_1w(const int* tbl, vintx idx, int k);
	dvintx vlookup_2x2pt_1wu(const unsigned int* tbl, vintx idx, int k);

Additional details	Use a single vector to supply indices to lookup parallel tables in the first P lanes (P = parallelism). Rbase is forced to be 64-byte aligned by ignoring its bits 5:0. Table[index], table[index+1], table[index + line_pitch], and table[index + line_pitch + 1] are returned for each parallel table. Table[index] and table[index + 1], are interleaved in the first 2*P lanes of the low part of destination double register. Table[index+ line_pitch] and table[index + line_pitch + 1], are interleaved in the first 2*P lanes of the high part of destination double register. Remaining lanes are returned as zero. Line_pitch is restricted to (64/P)*k+4 for Byte-type table, (32/P)*k+2 for Halfword-type table, and (16/P)*k+2 for Word-type table, k being an integer >= 0 and P = parallelism. The restriction ensures that the 2x2 points being read in each subtable do not collide in memory banks (16-bit per bank). The integer k is conveyed in the 8 LSBs of implicit scalar register PL, as an unsigned number. Note that the parallelism indicates number of parallel sub-tables we have. Number of data points returned is 4 times as many, as we look up 2 x 2 = 4 data points from each subtable. It IS allowed to have k = PL = 0. In this case, for H and W types, the lookup behaves like looking up 4 consecutive items from the indexed item. For B type, we would be fetching table[index], table[index+1], table[index+4], table[index+5] in each subtable. The access pattern is such that it's not obvious how it might be used. Refer to Table Lookup for index bit width used in address calculation.

For example, DVLUT_2x2pt_2W returns 4 data points from each of 2 subtables. Assume PL = 1, line pitch = 16/2*1 + 2 = 10. The following diagram shows the layout of an 2-way-parallel word-type table with line pitch of 10 elements, and where data points are picked up from index vector {1, 13}.

A[0][0]	A[0][1]	A[0][2]	A[0][3]	A[0][4]	A[0][5]	A[0][6]	A[0][7]	B[0][0]	B[0][1]	B[0][2]	B[0][3]	B[0][4]	B[0][5]	B[0][6]	B[0][7]
A[0][8]	A[0][9]	A[1][0]	A[1][1]	A[1][2]	A[1][3]	A[1][4]	A[1][5]	B[0][8]	B[0][9]	B[1][0]	B[1][1]	B[1][2]	B[1][3]	B[1][4]	B[1][5]
A[1][6]	A[1][7]	A[1][8]	A[1][9]	A[2][0]	A[2][1]	A[2][2]	A[2][3]	B[1][6]	B[1][7]	B[1][8]	B[1][9]	B[2][0]	B[2][1]	B[2][2]	B[2][3]
A[2][4]	A[2][5]	A[2][6]	A[2][7]	A[2][8]	A[2][9]	A[3][0]	A[3][1]	B[2][4]	B[2][5]	B[2][6]	B[2][7]	B[2][8]	B[2][9]	B[3][0]	B[3][1]

Returned destination low part = {A[0][1], A[0][2], B[1][3], B[1][4], 0, 0, 0, 0},
and high part = {A[1][1], A[1][2], B[2][3], B[2][4], 0, 0, 0, 0}.

Instruction name	**DVLUT_2X2PT (HB/HBU)**
Functionality	Double vector two-by-two-point lookup
Assembly format	DVLUT_2x2pt_<type-parallelism> *(Rbase+Vsrc), DVdst
Type and bit width	type-parallelism = {8/4/2/1 HB/HBU } The first type letter indicates type of indices; indices are always signed. The second type letter indicates type of table entries including signed/unsigned For example: DVLUT_2x2pt_2HB *(R4 + V0), V2:V3
Predication	Not available
Source options	Rbase: scalar register Vsrc: single vector register
Destination options	DVdst: double vector register
Additional options	

Intrinsics/operator	`dvcharx vlookup_2x2pt_8hb( const char* tbl, vshortx idx, int k);`
	`dvcharx vlookup_2x2pt_8hbu(const unsigned char* tbl,vshortx idx,int k);`
	`dvcharx vlookup_2x2pt_4hb( const char* tbl, vshortx idx, int k);`
	`dvcharx vlookup_2x2pt_4hbu(const unsigned char* tbl,vshortx idx,int k);`
	`dvcharx vlookup_2x2pt_2hb( const char* tbl, vshortx idx, int k);`
	`dvcharx vlookup_2x2pt_2hbu(const unsigned char* tbl,vshortx idx,int k);`
	`dvcharx vlookup_2x2pt_1hb( const char* tbl, vshortx idx, int k);`
	`dvcharx vlookup_2x2pt_1hbu(const unsigned char* tbl,vshortx idx,int k);`

Additional details	Use single vector to supply indices to lookup parallel tables in the first P lanes (P = parallelism). Rbase is forced to be 64-byte aligned by ignoring its bits 5:0. Table[index] and table[index + 1], are interleaved in the first 2*P lanes of the low part of destination double register. Table[index + line_pitch] and table[index + line_pitch + 1], are interleaved in the first 2*P lanes of the high part of destination double register. Remaining lanes are returned as zero. Line_pitch is restricted to (64/P)*k+4 for Byte-type table, k being an integer > 0 and P = parallelism. The restriction ensures that the 2x2 points being read in each subtable do not collide in memory banks (16-bit per bank). The integer k is conveyed in the 8 LSBs of implicit scalar register PL, as an unsigned number. Note that with 8-bit unsigned number we can represent line pitch more than 16,000 8-bit data points, 8,000 16-bit data points, 4,000 32-bit data points, which are more than sufficient for normal applications. Note that the parallelism indicates number of sub-tables we have. Number of data points returned is 4 times as many, as we look up 2 x 2 = 4 data points from each subtable.

Instruction name	DVLUT_2X2PT (HB/HBU)
	Refer to Table Lookup for index bit width used in address calculation.

9.9.6.6 DVHist

Instruction name	DVHist
Functionality	Double vector histogram
Assembly format	DVHist_<type-parallelism> *(Rbase+DVsrc1), DVsrc2, DVdst DVHist_<type-parallelism> *(Rbase+DVsrc1), DVsrc2
Type and bit width	type-parallelism = {32H, 16W} Same type applies to indices and table entries. Both indices and table entries are signed. Note that it is possible to maintain unsigned histogram, as histogram update operation (addition) behaves the same way for signed or unsigned data. Just that the pre-update bin read back data are always sign-extended in the destination registers. For example: DVHist_16W *(R4 + V0:V1), V2:V3, V4:V5 DVHist_16W *(R4 + V0:V1), V2:V3
Predication	Not available
Source options	Rbase: scalar register DVsrc1: double vector register DVsrc2: double vector register
Destination options	DVdst: double vector register or none
Additional options	

Intrinsics/operator	`dvshortx vhist_32h( short* hist, dvshortx idx, dvshortx upd);`
	`dvintx vhist_16w( int* hist, dvintx idx, dvintx upd);`
	
	`void vhist_simple_32h( short* hist, dvshortx idx, dvshortx upd);`
	`void vhist_simple_16w( int* hist, dvintx idx, dvintx upd);`

Additional details	Use DVsrc1 as indices and DVsrc2 as weights for weighted histogram. The indexed entries are updated by adding the corresponding weights. Pre-update entries are optionally returned in DVdst. Rbase is forced to be 64-byte aligned by ignoring its bits 5:0.

9.9.6.7 VHist

Instruction name	VHist
Functionality	Single vector histogram
Assembly format	VHist_<type-parallelism> *(Rbase+Vsrc1), Vsrc2, Vdst VHist_<type-parallelism> *(Rbase+Vsrc1), Vsrc2

Instruction name	VHist
Type and bit width	type-parallelism = {16/8/4/2/1H, 8/4/2/1W} Same type applies to indices and histogram entries. Both indices and histogram entries are signed. Note that it is possible to maintain unsigned histogram, as histogram update operation (addition) behaves the same way for signed vs unsigned data. Just that the pre-update bin read back data are always sign-extended in the destination register. For example: VHist_4W *(R4 + V0), V1, V2 VHist_4W *(R4 + V0), V1
Predication	Not available
Source options	Rbase: scalar register Vsrc1: single vector register Vsrc2: single vector register
Destination options	Vdst: single vector register or none
Additional options	

Intrinsics/operator	`vshortx vhist_16h(short* hist, vshortx idx, vshortx upd);`
	`vshortx vhist_8h( short* hist, vshortx idx, vshortx upd);`
	`vshortx vhist_4h( short* hist, vshortx idx, vshortx upd);`
	`vshortx vhist_2h( short* hist, vshortx idx, vshortx upd);`
	`vshortx vhist_1h( short* hist, vshortx idx, vshortx upd);`
	
	`vintx  vhist_8w( int*  hist, vintx  idx, vintx  upd);`
	`vintx  vhist_4w( int*  hist, vintx  idx, vintx  upd);`
	`vintx  vhist_2w( int*  hist, vintx  idx, vintx  upd);`
	`vintx  vhist_1w( int*  hist, vintx  idx, vintx  upd);`
	
	`void vhist_simple_16h(short* hist, vshortx idx, vshortx upd);`
	`void vhist_simple_8h( short* hist, vshortx idx, vshortx upd);`
	`void vhist_simple_4h( short* hist, vshortx idx, vshortx upd);`
	`void vhist_simple_2h( short* hist, vshortx idx, vshortx upd);`
	`void vhist_simple_1h( short* hist, vshortx idx, vshortx upd);`
	
	`void vhist_simple_8w( int*  hist, vintx  idx, vintx  upd);`
	`void vhist_simple_4w( int*  hist, vintx  idx, vintx  upd);`
	`void vhist_simple_2w( int*  hist, vintx  idx, vintx  upd);`
	`void vhist_simple_1w( int*  hist, vintx  idx, vintx  upd);`

Additional details	Use Vsrc1 as indices and Vsrc2 as weights for weighted histogram. The indexed entries are updated by adding the corresponding weights. First K lanes of Vsrc1 and Vsrc2 are used for K-way histogram. Rbase is forced to be 64-byte aligned by ignoring its bits 5:0. Pre-update entries are optionally returned in the first K lanes of Vdst. The remaining lanes, if any, are returned 0.

9.9.6.8 DVHist_OR

Instruction name	DVHist_OR
Functionality	Double vector histogram with bitwise OR operation
Assembly format	DVHist_OR_<type-parallelism> *(Rbase+DVsrc1), DVsrc2, DVdst DVHist_OR_<type-parallelism> *(Rbase+DVsrc1), DVsrc2
Type and bit width	type-parallelism = {32H, 16W} Same type applies to indices and histogram entries. Both indices and histogram entries are signed. Note that it is possible to maintain unsigned histogram, as histogram update operation (addition) behaves the same way for signed vs unsigned data. Just that the pre-update bin read back data are always sign-extended in the destination registers. For example: DVHist_OR_16W *(R4 + V0:V1), V2:V3, V4:V5
Predication	Not available
Source options	Rbase: scalar register DVsrc1: double vector register DVsrc2: double vector register
Destination options	DVdst: double vector register or none (no-return/simple version)
Additional options	

Intrinsics/operator	`dvshortx vhist_or_32h( short* hist, dvshortx idx, dvshortx upd);`
	`dvintx vhist_or_16w( int* hist, dvintx idx, dvintx upd);`
	
	`void vhist_or_simple_32h( short* hist, dvshortx idx, dvshortx upd);`
	`void vhist_or_simple_16w( int* hist, dvintx idx, dvintx upd);`

Additional details	Use DVsrc1 as indices and DVsrc2 as updates. The indexed entries are updated by bitwise-ORing the corresponding updates. Pre-update entries are optionally returned in DVdst. Rbase is forced to be 64-byte aligned by ignoring its bits 5:0.

9.9.6.9 VHist_OR

Instruction name	VHist_OR
Functionality	Single vector histogram with bitwise OR operation
Assembly format	VHist_OR_<type-parallelism> *(Rbase+Vsrc1), Vsrc2, Vdst VHist_OR_<type-parallelism> *(Rbase+Vsrc1), Vsrc2
Type and bit width	type-parallelism = {16/8/4/2/1H, 8/4/2/1W} Same type applies to indices and histogram entries. Both indices and histogram entries are signed. Note that it is possible to maintain unsigned histogram, as histogram update operation (bitwise OR) behaves the same way for signed vs unsigned data. Just

Instruction name	VHist_OR
	that the pre-update bin read back data are always sign-extended in the destination register. For example: VHist_OR_4W *(R4 + V0), V1, V2
Predication	Not available
Source options	Rbase: scalar register Vsrc1: single vector register Vsrc2: single vector register
Destination options	Vdst: single vector register or none (no-return/simple version)
Additional options	

Intrinsics/operator	`vshortx vhist_or_16h(short* hist, vshortx idx, vshortx upd);`
	`vshortx vhist_or_8h( short* hist, vshortx idx, vshortx upd);`
	`vshortx vhist_or_4h( short* hist, vshortx idx, vshortx upd);`
	`vshortx vhist_or_2h( short* hist, vshortx idx, vshortx upd);`
	`vshortx vhist_or_1h( short* hist, vshortx idx, vshortx upd);`
	
	`vintx  vhist_or_8w( int*  hist, vintx  idx, vintx  upd);`
	`vintx  vhist_or_4w( int*  hist, vintx  idx, vintx  upd);`
	`vintx  vhist_or_2w( int*  hist, vintx  idx, vintx  upd);`
	`vintx  vhist_or_1w( int*  hist, vintx  idx, vintx  upd);`
	
	`void vhist_or_simple_16h(short* hist, vshortx idx, vshortx upd);`
	`void vhist_or_simple_8h( short* hist, vshortx idx, vshortx upd);`
	`void vhist_or_simple_4h( short* hist, vshortx idx, vshortx upd);`
	`void vhist_or_simple_2h( short* hist, vshortx idx, vshortx upd);`
	`void vhist_or_simple_1h( short* hist, vshortx idx, vshortx upd);`
	
	`void vhist_or_simple_8w( int*  hist, vintx  idx, vintx  upd);`
	`void vhist_or_simple_4w( int*  hist, vintx  idx, vintx  upd);`
	`void vhist_or_simple_2w( int*  hist, vintx  idx, vintx  upd);`
	`void vhist_or_simple_1w( int*  hist, vintx  idx, vintx  upd);`

Additional details	Use Vsrc1 as indices and Vsrc2 as update. The indexed entries are updated by biwise-ORing the corresponding updates. First K lanes of Vsrc1 and Vsrc2 are used for K-way histogram. Rbase is forced to be 64-byte aligned by ignoring its bits 5:0. Pre-update entries are optionally returned in the first K lanes of Vdst. Remaining lanes, if any, are returned 0.

9.9.6.10 DVAST

Instruction name	DVAST
Functionality	Double vector addressed store
Assembly format	<pred> DVAST_<type-parallelism> DVsrc1, *(Rbase+DVsrc2) pred = none, [P2.. P15]
Type and bit width	type-parallelism = {32H, 16W} Same type applies to indices and entries in memory. The indices are signed, whereas the entries in memory can be signed or unsigned, as memory store behaves the same way for signed vs unsigned data. For example: [P3] DVAST_16W V2:V3, *(R4 + V0:V1)
Predication	Per-lane predication
Source options	Rbase: scalar register DVsrc1: double vector register (as data) DVsrc2: double vector register (as indices)
Destination options	
Additional options	

Intrinsics/operator	`void vast_32h(short* arr, dvshortx idx, dvshortx data,`
	`  int pred);`
	`void vast_32hf(hfloatx* arr, dvshortx idx, dvhfloatx data, int pred);`
	`void vast_16w(int*  arr, dvintx  idx, dvintx  data, int pred);`
	`void vast_16f(float* arr, dvintx  idx, dvfloatx data, int pred);`

Additional details	Use DVsrc1 as data and DVsrc2 as indices, write each lane of DVsrc1 into memory object indexed by a corresponding lane of DVsrc2. Lowest K bits of pred argument is used to predicate K lanes. Rbase is forced to be 64-byte aligned by ignoring its bits 5:0.

9.9.7 Misc Register Store

9.9.7.1 Instruction Summary

These instructions support debug functionality by storing out otherwise inaccessible architecture registers to memory, so that debug controller can read the contents from memory.

Table 44 Miscellaneous register store instructions

Function	Assembly Format	Comments
Store hardware loop register	STW HWLP.<reg>, *(Rbase+imm12) reg = LF, LS[0/1], LE[0/1], LC[0/1]	Use Rbase + (signed) imm12 as byte address. Data zero-padded in case of LF (2 bits)
Store Agen loop variable	STH A<id>.I<level>, *(Rbase+imm12)	Each variable is 16-bit

9.9.7.2 STW HWLP

Instruction name	STW HWLP
Functionality	Store hardware loop register
Assembly format	STW HWLP.<reg>, *(Rbase+imm12) reg = LF, LS[0/1], LE[0/1], LC[0/1]
Type and bit width	LF: 2-bit, zero-padded into 32-bit LS/LE/LC: 32-bit
Predication	Not available
Source options	Specific HWLP register Rbase: scalar register
Destination options	
Additional options	
Intrinsics/operator	`not available`
Additional details	This instruction is intended to be used in Debug State, injected through debug client to query hardware loop registers through VMEM. In normal (non-debug) programming, placement of STW HWLP in the following packets lead to indefinite outcome: • In two packets before RPT • In the same packet as RPT • In two delay slots following RPT • In first 3 packets of loop body • In last 3 packets of loop body • In first 2 packets after the loop Note that debug-injection of STW HWLP is not hindered, as pipeline is flushed before and existing debug state.

9.9.7.3 AgenLpv ST

Instruction name	AgenLpv ST
Functionality	Store Agen loop variable
Assembly format	STH A<id>.I<level>, *(Rbase+imm12) id = 0..7, level = 1..6
Type and bit width	16-bit
Predication	Not available
Source options	Specific Agen loop variable register Rbase: scalar register
Destination options	
Additional options	
Intrinsics/operator	`not available`
Additional details	

Chapter 10. Decoupled Lookup Unit (DLUT) Reference

10.1 Index and Output Data Format

To provide some degree of flexibility in data formatting without sacrificing area/performance/power efficiency, DLUT supports a subset of address calculation capability via a reduced set of agen, address generator, parameters. There is one set of agen parameters for index read, and another set of agen parameters for output write. Agen operation is similar to the agen in VPU instruction set, but some of the configuration parameters are derived from other DLUT parameters:

- Indices are always unsigned.
- Table entries can be treated as signed or unsigned in post-lookup interpolation. There is no bit width expansion from calculated output to memory, so the store itself is agnostic to signed/unsigned data types.
- Number of inner-loop iterations N1 and size of partial memory transactions are derived from task length N1 and size of index/output as configured:

```
INDEX_AGEN_N1 = ((TASK_LEN_N1 * idx_dim) + (64/sizeof_idx – 1))/ (64/sizeof_idx);
INDEX_REMAINDER = (TASK_LEN_N1 * idx_dim) % (64/sizeof_idx);
OUTPUT_AGEN_N1 = (TASK_LEN_N1 + (64/sizeof_entry – 1)) / (64/sizeof_entry);
OUTPUT_REMAINDER = TASK_LEN_N1 % (64/sizeof_entry);
```

  idx_dim = 1 for 1D lookup/interpolation, and 2 for 2D lookup/interpolation. Note that zero remainder means all read/write transactions are full, 64-byte, transactions.
- Number of outer-loop iterations N2 are simply copied from task length N2:

```
INDEX_AGEN_N2 = OUTPUT_AGEN_N2 = TASK_LEN_N2;
```

- Two-level address calculation capability is supported, so there are AMOD1 and AMOD2 parameters for each agen.
- Transpose option is included to select among P (no transpose), T1, T2, T4, T8, T16, T32, along with lane_ofst parameter to specify transpose line pitch.
- In case of 2D lookup (with or without interpolation), X/Y indices can be element-wise interleaved, or 32-byte interleaved (with 32 bytes of X then 32 bytes of Y). This is analogous of VPU double vector load having deinterleaving option, or VPU double vector store having interleaving option.
- No rounding/saturation agen fields are supported.

Note that auto-indexing mode uses the index stream to load parameters that drive DLUT index generation, so it has a different data layout in the index stream. See 2D Lookup with Interpolation with Auto Index Generation for details.

When INDEX_REMAINDER is non-zero, hardware handles partial index read memory transactions correctly. For example, if INDEX_REMAINDER = 10 while full memory read involves 16 indices (say idx_dim = 2 and index type = Halfword), the last 6 index lanes of the last inner-loop iteration will be discarded.

When OUTPUT_REMAINDER is non-zero, hardware handles partial output write memory transactions correctly. For example, if OUTPUT_REMAINDER = 5 while full memory write involves 32 indies (say entry type = Halfword), the last 27 output lanes of last inner-loop iterations are predicated off and not write any garbage values.

With element-wise interleaved X/Y format, the expected data layout for the first transaction is:

```
X[0] Y[0] X[1] Y[1] X[2] Y[2] … X[15] Y[15]
```

The last transaction is:

```
X[80] Y[80] X[81] Y[81] X[82] Y[82] X[83] Y[83] DC0 DC1 … DC23 (DC = don't care)
```

With 32-byte interleaved X/Y format, the expected data layout for the first transaction is:

```
X[0] X[1] X[2] … X[15] Y[0] Y[1] Y[2] … Y[15]
```

The last 32 indices is:

```
X[80] X[81] X[82] X[83] DC0 … DC11 Y[80] Y[81] X[82] X[83] DC12 … DC23
```

Since index read and output write agen N1/N2 are derived from task length N1/N2, there cannot be inconsistency in index data stream, between production and consumption, and in output data stream, between production and consumption.

With N1/N2 derived from task length N1/N2, additional parameters for index read agen are:

- index_addr
- index_amod1
- index_amod2
- index_transp_mode
- index_lane_ofst
- index_interleave_format

Additional parameters for output write agen are:

- output_addr
- output_amod1
- output_amod2
- output_transp_mode
- output_lane_ofst

Transposing mode can be P (no transpose), T1 (halfword or word only), T2, T4, T8, T16 (byte or halfword only), or T32 (byte only). The “T” number (for example of 2 in T2) indicates number consecutive elements before applying line pitch to go down to the next row.

Disallowed type and transpose combinations shall be detected as incorrect configuration. See Incorrect Task Configuration for handling.

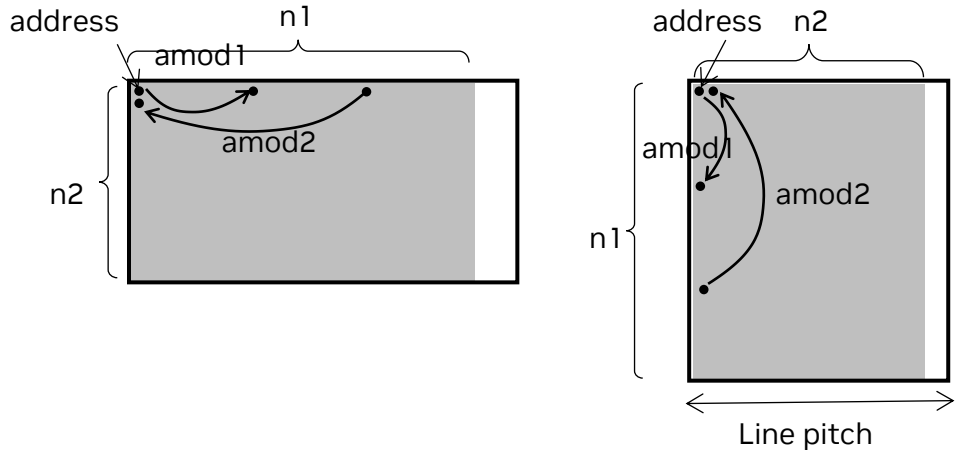
Like VPU transposed load/store, the line pitch (in data elements) must comply with the following constraint, with index/output_lane_ofst supplying the integer “k” in the line pitch constraint.

Table 45 Index and output line pitch and transpose modes

LINE PITCH	Transpose mode					
Entry type	T1	T2	T4	T8	T16	T32
Byte	n/a	64k + 2	64k + 4	64k + 8	64k + 16	64k + 32
Halfword	32k + 1	32k + 2	32k + 4	32k + 8	32k + 16	n/a
Word	16k + 1	16k + 2	16k + 4	16k + 8	n/a	n/a

With flexibility in the agen (even one trimmed down to 2 levels) together with various transpose modes, it is quite difficult to visualize all possible data layout for index and output. The following diagrams show two example layouts in P (no transpose) mode and T1 mode.

Figure 14. DLUT index/output data layout



Index/output layout non-transposed **Index/output layout T1 transposed**

In the un-transposed example layout above, parameter n1 specifies the width of the rectangle data region (grey box), and parameter n2 specifies the height. In the transposed example layout, parameter n1 specifies the height of width of the rectangle data region (grey box), and parameter n2 specifies the width. In both cases, parameters amod1/amod2 specify offset between address pointer updates (address pointers

expressed as dots in the diagram). Amod1 specifies the address offset between read/write memory transactions, and amod2 specifies the address offset at end of the inner loop (after index/output agen N1 transactions).

Index and output addressing shall have consistent address alignment requirements as VPU load/store instructions. Byte/Halfword index/output shall be 16-bit aligned. Here we consider byte data to be access 64 bytes per transaction so Agen address pointers should be 16-bit aligned instead of being 8-bit aligned. Word index/output shall be 32-bit aligned.

Address alignment is enforced by ignoring 1 or 2 LSBs of the agen address driving index reads and output writes. That is, address = base and address += AMOD1/AMOD2 steps are calculated without AND masking. AND masking is applied as address goes to VMEM for read/write, ignoring 1 LSB (aligned to halfword) for Byte/Halfword types and ignoring 2 LSBs (aligned to word) for Word type. This is consistent with VPU agen addressing behavior when reading/writing Byte/Halfword/Word type double vector.

For example, say INDEX_ADDR is configured as 0x1001 and INDEX_AGEN_AMOD1 is configured as 0x41 for a LOOKUP_2D task with Halfword index type. The first few iterations of index agen base address and address used to read indices are as follows:

Iteration	Agen base	Read address (aligned to Halfword)
0	0x1001	0x1000
1	0x1042	0x1042
2	0x1083	0x1082
3	0x10C4	0x10C4
4	0x1105	0x1104

Index and output agen address calculation shall behave the same as in VPU agen address calculation (see 6.4.1) in that the selected AMOD is read as signed 18-bit number does not encode large enough jump to go from one superbank primary region into another superbank's primary region. However, it is possible to walk through an aliased region into another superbank, though it is strongly discouraged to address an aliased region, as it may break software compatibility in the future.

Agen address update process can be expressed as:

```
lpend1 = (i1 == (agen_n1 - 1)) || (agen_n1 == 0);
lpend2 = (i2 == (agen_n2 - 1)) || (agen_n2 == 0);

if (lpend1 && lpend2) {
 amod = 0; // stay at last data point
} else if (lpend1) {
 i1 = 0;
 i2 = i2+1;
 amod = amod2;
} else {
 i1 += 1;
 amod = amod1;
}
```

```
addr += amod;
```

10.2 Table Data Format

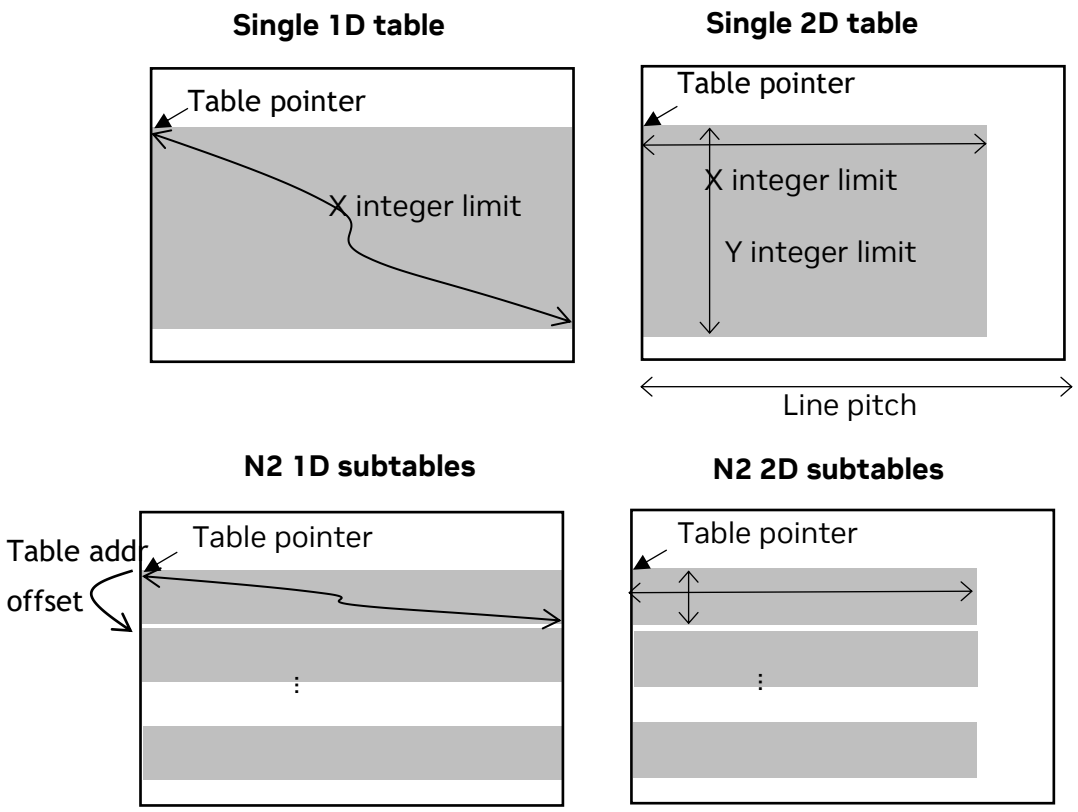
Table data, the target of table lookups, should be a single data type, Byte, Halfword, or Word, specified in the task configuration.

Table data format is specified via these task configuration parameters:

- Task mode: 1D or 2D table
- Entry type can be signed/unsigned Byte/Halfword/Word
- X integer limit: linear (for 1D table) or X dimension limit (for 2D table)
- Y integer limit: Y dimension limit (for 2D table)
- Line pitch: for 2D table
- Table pointer: starting address of the table, 64 bytes aligned
- Table address offset: address offset per outer iteration; there are N1 inner iterations and N2 outer iterations, 64 bytes aligned

The following diagrams show table organization with various parameters.

Figure 15. DLUT table data layout



The table pointer can vary among task_N2 rounds of lookup, by adding the table offset after each round of task_N1 lookups.

The table data address shall be 512-bit or 64-byte aligned, to be consistent with VPU lookup instructions. The alignment constraint applies to table base address and table address offset.

Table size in any DLUT task is limited to one superbank, for the index calculation process includes steps to map lookup accesses into the superbank where the table data address resides. Note that this is different from index read and output write agen address update, where it is possible to walk from one superbank into another superbank.

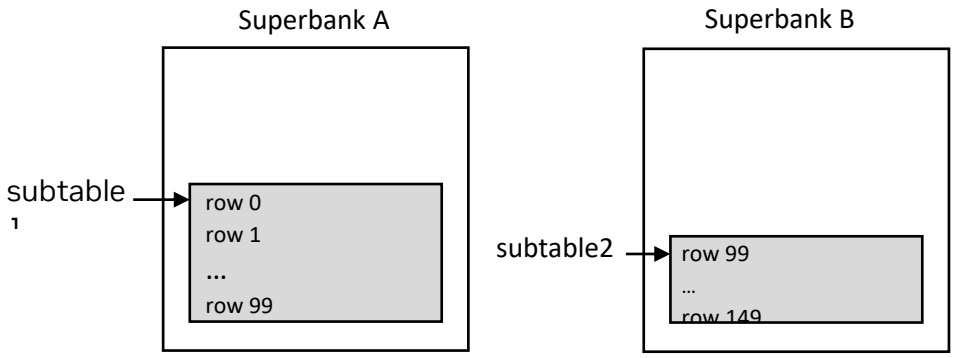
With the predicating off output write out-of-range handling option, the programmer can construct multiple DLUT tasks to implement table being allocated in 2 or even 3 superbanks, by proper configuration of out-of-range option, X/Y offset and X/Y integer limit.

For example, for a 2D lookup with interpolation with 150 rows x 400 columns of lookup table, say only 100 table rows of would fit one superbank. We would place the last 51 table rows in the other superbank (to have 1 row of overlap between 2 subtables). We would construct DLUT tasks as:

```
task1_param.out_of_range_hndl = OOR_EN_SENTINEL;
task1_param.X_offset = 0;
task1_param.Y_offset = 0;
task1_param.X_int_limit = 399;
task1_param.Y_int_limit = 99;
task1_param.table_addr = subtable1;
task1_param.next_task = &task2_param;
task2_param.out_of_range_hndl = OOR_EN_PRED_OFF;
task2_param.X_offset = 0;
task2_param.Y_offset = -99;   // maps original row 99 to subtable2 row 0
task2_param.X_int_limit = 399;
task2_param.Y_int_limit = 50;  // since original row 149 is the last valid row
task2_param.table_addr = subtable2;
task2_param.next_task = NULL;
```

Note that the same index buffer and output buffer are provided to both tasks, so there is no preprocessing or postprocessing needed to separate indices or combine outputs.

Figure 16. Example to leverage out-of-range handling to split a large table as two sub-table lookups



With out-of-range handling configured as predicating off, potentially we can have every lane of a write transaction being predicated off, particularly at the last write transactions of the inner (task_length_N1) loop that can be partial transactions. The current implementation does not optimize out such NULL write transactions. The occurrence of such transactions is data dependent and should be infrequent if the use case is well optimized.

10.3 Index Calculation

Index and lookup address calculation for the first 4 modes are described in the following subsections.

Input and parameter bit width common to these modes are as follows:

- index[i]: U16 or U32, use worst case U32
- x_offset, y_offset: S32
- frac_bits: 0 ~ 16
- frac_mask = (1 << frac_bits) -1; // U16
- round_add = (frac_bits == 0) ? 0 : (1 << (frac_bits – 1)); // U16
- tbl_addr: U20
- line_pitch: U16

10.3.1 1D Lookup

In 1D lookup (without interpolation), DLUT shall perform for each data point iterated by task_i2 in the outer loop and task_i1 in the inner loop:

```
x = index[task_i1] + x_offset;                          // U32 + S32 = S34
x_int = (round_trunc_mode == 0) ? (x >> frac_bits)
   : ((x + round_add) >> frac_bits);       // S34
x_in_range = (x_int >= 0) && (x_int <= x_int_limit); // Boolean
out_of_range = out_of_range_enable && !x_in_range;  // Boolean
```

```
lu_idx = x_int;                                  // S32
```

When out-of-range detection is enabled and index is detected to be out of range, lookup for that specific output is not performed, and either configured sentinel value is returned instead, or writing of that output is predicated off.

When the task outer loop parameter task_length_N2 is greater than one, table pointer advances with each round of task_length_N1 outcomes. The lookup index is scaled by entry size and added to the table pointer as well. Note that we never switch superbank (from the one configured in the task parameter) with the table pointer advancing or entry indexing, as they only affect the lower 17 bits of the byte address (covering 128KB inside a superbank).

```
entry_addr = (tbl_ptr & 0xC0000)
+ ((tbl_ptr + task_i2 * tbl_addr_offset
       + lu_idx * sizeof_entry) & 0x1FFFF); // U20
```

The entry address is decomposed into superbank ID, row address, bank ID, and byte ID (only for Byte-type entries):

```
entry_superbank = entry_addr[19:18]; // 0 = superbank A, 1 = B, 2/3 = C
entry_row_addr = entry_addr[16:6];  // 11 bits covering 2K rows
entry_bank_id  = entry_addr[5:1];     // 5 bits covering 32 banks
entry_byte_id  = entry_addr[0];   // 1 bit covering 2 bytes
```

Although index data is unsigned, x_offset is signed, so index calculation involves signed arithmetic. When out-of-range detection is disabled, the lookup index lu_idx can be negative or can exceed VMEM superbank address range. With the way entry address is calculated, the lookup would wrap address back into the same superbank as configured in the task parameter. This is consistent with VPU table lookup address wrapping described in 0.

10.3.2 1D Lookup with interpolation

Index calculation for 1D lookup with interpolation mode is the same as that of 1D lookup mode, except:

- We need to calculate the fraction part of the index to perform interpolation, so the integer component is always calculated with truncation.
- Out-of-range detection takes into account index value right on the last valid data point with zero fraction.

```
x = index[task_i1] + x_offset;                      // U32 + S32 = S34
x_int = x >> frac_bits;                                    // S34
x_frac = x & frac_mask;                                    // U16
x_in_range = (x_int >= 0) && ((x_int < x_int_limit) ||
       ((x_int == x_int_limit) && (x_frac == 0))); // Boolean
out_of_range = out_of_range_enable && !x_in_range;   // Boolean
lu_idx = x_int;                                  // S32
```

When out-of-range detection is enabled and index is detected to be out of range, the two lookups for that specific output is not performed, and either configured sentinel value is returned instead or, writing of that output is predicated off.

Like the 1D lookup mode, we never switch superbank (from the one configured in the task parameter) with the table pointer advancing or entry indexing:

```
entry_addr = (tbl_ptr & 0xC0000)
+ ((tbl_ptr + task_i2 * tbl_addr_offset
       + lu_idx * sizeof_entry) & 0x1FFFF); // U20
```

In addition, the entry address for the next table entry is calculated in order to look up two items for interpolation. The next table entry shall be also in the same superbank:

```
entry_addr2 = (tbl_ptr & 0xC0000)
+ (entry_addr + sizeof_entry) & 0x1FFFF; // U20
```

How each entry address is decomposed into superbank ID, row address, bank ID, and optionally byte ID is similar to the 1D lookup mode. See 1D Lookup for details.

Again, although index data is unsigned, x_offset is signed, so index calculation involves signed arithmetic. When out-of-range detection is disabled, the lookup index lu_idx can be negative or can exceed VMEM superbank address range. With the way entry address is calculated, the lookup would wrap address back into the same superbank as configured in the task parameter. This is consistent with VPU table lookup address wrapping described in 0.

10.3.3 2D Lookup

In 2D lookup (without interpolation), DLUT shall perform for each data point:

```
x = index[2*task_i1] + x_offset;                    // U32 + S32 = S34
y = index[2*task_i1+1] + y_offset;                  // U32 + S32 = S34
x_int = (round_trunc_mode == 0) ? (x >> frac_bits) // S34
   : ((x + round_add) >> frac_bits);
y_int = (round_trunc_mode == 0) ? (y >> frac_bits) // S34
   : ((y + round_add) >> frac_bits);
x_in_range = (x_int >= 0) && (x_int <= x_int_limit); // Boolean
y_in_range = (y_int >= 0) && (y_int <= y_int_limit); // Boolean
out_of_range = out_of_range_enable && (!x_in_range || !y_in_range); //Boolean
lu_idx = y_int * line_pitch + x_int;                // S32
```

When out-of-range is detected, lookup for that specific output is not performed, and either configured sentinel value is returned instead, or writing of that output is predicated off.

Similar to the 1D lookup mode, we never switch superbank (from the one configured in the task parameter) with the table pointer advancing or entry indexing:

```
entry_addr = (tbl_ptr & 0xC0000)
+ ((tbl_ptr + task_i2 * tbl_addr_offset
       + lu_idx * sizeof_entry) & 0x1FFFF); // U20
```

How each entry address is decomposed into superbank ID, row address, bank ID, and optionally byte ID is similar to the 1D lookup mode. See 1D Lookup for details.

Again, although index data is unsigned, x_offset and y_offset are signed, so index calculation involves signed arithmetic. When out-of-range detection is disabled, the lookup index lu_idx can be negative or can exceed VMEM superbank address range. With the way entry address is calculated, the lookup would wrap address back into the same superbank as configured in the task parameter. This is consistent with VPU table lookup address wrapping described in 0.

10.3.4 2D Lookup with Interpolation

Index calculation for 2D lookup with interpolation mode is the same as that of 2D lookup mode, except:

- We need to calculate the X and Y fraction parts of the index to perform interpolation, so the integer component is always calculated with truncation.
- Out-of-range detection takes into account index value right on the last row or last column of valid data region with zero fraction.

```
x = index[2*task_i1] + x_offset;                    // U32 + S32 = S34
y = index[2*task_i1+1] + y_offset;                  // U32 + S32 = S34
x_int = x >> frac_bits;                                    // S34
y_int = y >> frac_bits;                                    // S34
x_frac = x & frac_mask;                                    // U16
y_frac = y & frac_mask;                                    // U16
x_in_range = (x_int >= 0) && ((x_int < x_int_limit) ||
       ((x_int == x_int_limit) && (x_frac == 0))); // Boolean
y_in_range = (y_int >= 0) && ((y_int < y_int_limit) ||
       ((y_int == y_int_limit) && (y_frac == 0))); // Boolean
out_of_range = out_of_range_enable && (!x_in_range || !y_in_range); //Boolean
lu_idx = y_int * line_pitch + x_int;                // S32
```

When out-of-range detection is enabled and index is detected to be out of range, the 4 lookups for that specific output is not performed, and either configured sentinel value is returned instead, or writing of that output is predicated off.

Similar to the 1D lookup mode, we never switch superbank (from the one configured in the task parameter) with the table pointer advancing or entry indexing:

```
entry_addr = (tbl_ptr & 0xC0000)
+ ((tbl_ptr + task_i2 * tbl_addr_offset
       + lu_idx * sizeof_entry) & 0x1FFFF);    // U20
```

In addition, address for 3 additional table entries is calculated to look up 2x2 = 4 items for interpolation. These additional table entries shall be also in the same superbank:

```
entry_addr2 = (tbl_ptr & 0xC0000)
  + (entry_addr + sizeof_entry) & 0x1FFFF;       // U20
entry_addr3 = (tbl_ptr & 0xC0000)
  + (entry_addr + line_pitch * sizeof_entry) & 0x1FFFF; // U20
entry_addr4 = (tbl_ptr & 0xC0000)
  + (entry_addr + line_pitch * sizeof_entry + sizeof_entry) & 0x1FFFF;//U20
```

How each entry address is decomposed into superbank ID, row address, bank ID, and optionally byte ID is similar to the 1D lookup mode. See 1D Lookup for details.

Again, although index data is unsigned, x_offset and y_offset are signed, so index calculation involves signed arithmetic. When out-of-range detection is disabled, the lookup index lu_idx can be negative or can exceed VMEM superbank address range. With the way entry address is calculated, the lookup would wrap address back into the same superbank as configured in the task parameter. This is consistent with VPU table lookup address wrapping described in 0.

10.3.5 2D Lookup with Interpolation with Auto Index Generation

The 2D lookup with interpolation with automatic index generation mode involves these additional parameters:

- AUTO_IDX_MODE: specifies whether it's translation-only mode (index stream loads x0/y0, 2 parameters per round of lookup/interpolation) or translation-scaling mode (index stream loads x0/y0/step_x/step_y, 4 parameters per round of lookup/interpolation) per round of lookup/interpolation.
- AUTO_IDX_TRAVERSAL_DIR: specifies that index traversal going horizontally first (when it's 0) or vertically first (when it's 1).
- AUTO_IDX_PATCH_WIDTH (U8): specifies patch width
- AUTO_IDX_PATCH_HEIGHT (U8): specifies patch height

Basically, DLUT in this mode instead of reading Task_len_N2 x Task_len_N1 pairs of indices from VMEM, would read just Task_len_N2 sets of (2 or 4) parameters and generate indices on the fly to drive lookup and interpolation.

DLUT hardware shall carry out the following process to generate indices for task_len_N1 outputs in the inner loop. task_len_N1 must match PATCH_WIDTH * PATCH_HEIGHT. Also, we need PATCH_WIDTH >= 8 when traversing horizontally first, and PATCH_HEIGHT >= 8 when traversing vertically first, to simplify index generation.

This process is for group size of 8, which applies for IDX_W index type and U16/S16 table entry type that we are supporting for auto-indexing mode.

In the normal (horizontally first) mode, hardware follows this behavior:

```
int xi[] = {0, 1, 2, 3, 4, 5, 6, 7};
int yi[] = {0, 0, 0, 0, 0, 0, 0, 0};
y = replicate(y0);    // all lanes initialized to y0
x = x0 + xi * step_x; // lane i = x0 + i*step_x

for (i1 = 0, i1 = 0; i1 < N1; i1 += group_size) {

// proceed with address calculation with coordinate (x, y)

// update x, y for next group
  adv_mask = (xi+8) >= PATCH_WIDTH; // boolean vector
 yi += adv_mask;          // add 0 or 1
  y += adv_mask * step_y;     // add 0 or step_y
```

```
 valid_mask = yi < PATCH_HEIGHT;  // mask out lanes in last group
 xi += adv_mask ? (8 - PATCH_WIDTH) : 8;
  x += adv_mask ? ((8 – PATCH_WIDTH) * step_x) : (8 * step_x);
}
```

Otherwise (in the vertically first mode), hardware follows this behavior:

```
int xi[] = {0, 0, 0, 0, 0, 0, 0, 0};
int yi[] = {0, 1, 2, 3, 4, 5, 6, 7};
x = replicate(x0);    // all lanes initialized to x0
y = y0 + yi * step_y; // lane i = y0 + i*step_y

for (i1 = 0, i1 = 0; i1 < N1; i1 += group_size) {

// proceed with address calculation with coordinate (x, y)

// update x, y for next group
 adv_mask = (yi+8) >= PATCH_HEIGHT; // boolean vector
 xi += adv_mask;          // add 0 or 1
  x += adv_mask * step_x;      // add 0 or step_x
 valid_mask = xi < PATCH_WIDTH;   // mask out lanes in last group
 yi += adv_mask ? (8 - PATCH_HEIGHT) : 8;
  y += adv_mask ? ((8 – PATCH_HEIGTH) * step_y) : (8 * step_y);
}
```

When AUTO_IDX_MODE specifies translation-only mode, only x0/y0 are loaded per round of task_len_N1 outputs, hardware would derive step X/Y from frac_bits:

step_x = step_y = 1 << frac_bits

X/Y update is expressed for index traversal going horizontally first. The vertically first option can be implemented by swapping the X/Y feeding rest of the address calculation.

Rest of the address calculation process, duplicate detection, conflict resolution, post lookup interpolation, index/output agen (other than index agen N1/N2 derivation) all operate the same way as in the common table lookup with and without interpolation modes.

Index agen parameters should be derived from Task_len_N1, Task_len_N2, differently than the normal lookup/interpolation modes:

```
INDEX_AGEN_N1 = (TASK_LEN_N2 * [2 or 4] + 64/sizeof_idx – 1)/ (64/sizeof_idx);
INDEX_REMAINDER = (TASK_LEN_N2 * [2 or 4]) % (64/sizeof_idx);
INDEX_AGEN_N2 = 1;
```

Note that x0/y0 or x0/y0/step_x/step_y data in memory shall be interpreted consistently with INDEX_DATA_TYPE (constrained to IDX_W for the auto-indexing mode), as unsigned 32-bit words.

Other relevant parameters:

- INDEX_ADDR specifies starting address of these parameters.
- INDEX_AGEN_TRANSP_MODE should be 0 (no transposition).
- INDEX_AGEN_LANE_OFST is not used.
- INDEX_AGEN_AMOD2 is not used.

10.4 Duplicate Detection and Consolidation

Before sending read/read-modify-write/write requests to VMEM, DLUT shall first detect duplicate requests. Duplicate requests are consolidated for performance and power, and the same return values are broadcast to the multiple return-value lanes as needed.

Note that the hardware has a certain window where duplicate detection works; not all duplicates within a task are caught. The duplicate detection feature is for performance and has no effect on the final outcome.

Duplicate detection logic does consume some power in operation. There is an enable bit in the task parameter block to enable/disable duplicate detection/consolidation on the task. Programmer should disable duplicate detection/consolidation only for DLUT tasks that are expected to have very few duplicates.

10.5 Conflict Resolution and Lookup

After duplicate requests are detected and consolidated by voiding redundant requests, the bank address of valid requests are compared, conflict detected, and DLUT hardware issues read requests to complete the lookup as needed to the table in VMEM.

Not all individual lookups are performed due to out-of-range detection and duplicate detection. For the sake of functionality description, we can say that hardware performs 1, 2, or 4 lookups as prescribed using the entry address(es) calculated for each output not deemed out-of-range:

```
entry = * entry_addr;  // all modes, the anchor entry
entry2 = * entry_addr2; // 1D or 2D interpolation, to the right of anchor
entry3 = * entry_addr3; // 2D interpolation only, down from anchor
entry4 = * entry_addr4; // 2D interpolation only, down-and-right from anchor
```

10.6 Post Lookup Interpolation

For 1D lookup with linear interpolation, DLUT performs for each output not deemed out-of-range:

```
y_out = entry + round((entry2 – entry) * x_frac, frac_bits);
```

For 2D lookup with bilinear interpolation, DLUT performs for each output not deemed out-of-range:

```
y0 = entry + round((entry2 – entry) * x_frac, frac_bits);
y1 = entry3 + round((entry4 – entry3) * x_frac, frac_bits);
y_out = y0 + round((y1 – y0) * y_frac, frac_bits);
```

Note that x_frac and y_frac are extracted from frac_bits LSBs of x and y, so for linear/bilinear interpolation to work correctly, round/trunc mode shall be set to truncation.

In all lookup modes, when X or Y are out of range and out-of-range is enabled, either the configured sentinel value is returned instead of interpolated value, or output write is predicated off. Note that the out-of-range detection is performed in case of interpolated lookup such that when any dependent entry (out of 2 or 2x2 entries) is out-of-range, it's detected as out-of-range.

It's the programmer's responsibility to set round/truncate mode and X/Y limit correctly for the mode of operation performed. DLUT hardware shall just carry out lookup and calculation with configuration parameters provided.

10.7 2D Conflict-free Lookup with Interpolation

The 2D conflict-free bilinear interpolation mode allows DLUT to offload 2D-to-linear index calculation, lookup, and post-lookup interpolation from VPU, reducing energy and improving performance as well.

The following parameters are constrained for this 2D conflict-free lookup with interpolation:

- Task mode: 0x04 (2D conflict-free lookup with interpolation)
- Entry data type: must be signed or unsigned Halfword
- Index data type: must be unsigned Halfword or unsigned Word
- Line pitch: must be 4k + 2 (k being any integer)

In this mode, table data is organized as 8-way parallel subtables with Halfword entries. It is the same table organization as for VPU DVLUT_8H and DVLUT_2x2pt_8H instructions, and is shown as follows with linear indexing for each subtable:

Figure 17. Table layout for VPU lookup instructions

T0[0..3]	T1[0..3]	...	T7[0..3]
T0[4..7]	T1[4..7]	...	T7[4..7]
T0[8..11]	T1[8..11]	...	T7[8..11]

...

For example, if each subtable has width of 9 and height of 4, we would pick line pitch = 4*2+2 = 10, and have the following table layout (p = pad, or don't care value):

Figure 18. Table layout for DLUT 2D conflict-free lookup w/ interpolation

T0[0][0]	T0[0][1]	T0[0][2]	T0[0][3]	T1[0][0]	T1[0][1]	…	T7[0][3]
T0[0][4]	T0[0][5]	T0[0][6]	T0[0][7]	T1[0][4]	T1[0][5]		T7[0][7]
T0[0][8]	p	**T0[1][0]**	T0[1][1]	T1[0][8]	p		T7[1][1]
T0[1][2]	T0[1][3]	T0[1][4]	T0[1][5]	T1[1][2]	T1[1][3]		T7[1][5]
T0[1][6]	T0[1][7]	T0[1][8]	p	T1[1][6]	T1[1][7]		p
T0[2][0]	T0[2][1]	T0[2][2]	T0[2][3]	T1[2][0]	T1[2][1]		T7[2][3]
T0[2][4]	T0[2][5]	T0[2][6]	T0[2][7]	T1[2][4]	T1[2][5]		T7[2][7]
T0[2][8]	p	**T0[3][0]**	T0[3][1]	T1[2][8]	p		T7[3][1]

…

Note that the line pitch is inside each subtable, which is 4 elements wide, instead of line pitch in the full-width VMEM in the context of storing a 2D array in VMEM. In this case, each subtable contains a 2D table, and the line pitch is needed to translate 2D indices (x_int, y_int) into linear indices (lu_idx*) to perform lookups in each subtable.

With such table organization and such line pitch, each set of 2x2 lookups go to its own set of 4 memory banks, and each of the 2x2 lookups goes to its own memory bank, so there is no conflict, and no replication either.

Example:

- line_pitch = 10. fraction_bits = 2.
- At lane 0 of certain group, we get X index = 7, Y index = 5.
- x_int = 1, x_frac = 3, y_int = 1, y_frac = 1.
- lu_idx = 1 * 10 + 1 = 11.
- lu_idx2 = 11 + 1 = 12.
- lu_idx3 = 11 + 10 = 21.
- lu_idx4 = 11 + 10 + 1 = 22.

We use 8-way parallel, halfword variation of the parallel table address calculation (from 0):

```
byte_offset[i] = ((index modulo K) + i*K) * M + floor(index / K)*64, for i = 0..N-1
```

where N = parallelism, K = stride, M = entry size in bytes.

It's clearer in this context to translate the above to a halfword offset:

```
 halfword_offset = (index modulo K) + i*K + floor(index / K)*32
```

In this case, parallelism N=8, data size M=2, stride K=4, and lane i = 0, so for the first of the 2x2 lookups we have

```
halfword_offset = ((lu_idx modulo 4) + 0*4 + floor(lu_idx / 4) * 32
= (11 modulo 4) + floor(11/4)*32 = 3 + 2*32 = 67
```

For rest of the 2x2 lookups we have:

```
   halfword_offset2 = (12 modulo 4) + floor(12/4)*32 = 0 + 3*32 = 96
halfword_offset3 = (21 modulo 4) + floor(21/4)*32 = 1 + 5*32 = 161
halfword_offset4 = (22 modulo 4) + floor(22/4)*32 = 2 + 5*32 = 162
```

We can see that the 2x2 lookups go to memory banks 3, 0, 1, 2 of the first subtable, due to the 4k + 2 line pitch.

Index read agen should be configured appropriately to supply X/Y indices to perform address calculation and post-lookup interpolation. Since index read transactions are configured to read 64 bytes at a time, for Halfword indices, 16 pairs of X/Y indices are read at a time to feed 2 cycles of lookup. For Word indices, 8 pairs of X/Y indices are read at a time to feed one cycle of lookup.

Output write agen should be configured appropriately to write outputs. Since output write transactions are configured to write 64 bytes or 32 halfwords at a time, the hardware accumulates 4 cycles of lookup/interpolation to issue each output write.

Features supported in this mode are:

- Out-of-range detection: supported.
- Round mode: not relevant, as it only applies to lookup without interpolation.
- Duplicate handling: not relevant, as subtables are separate so no duplicate lookups are possible.
- Task length N1/N2 and table address stepping: supported.
- Index agen transposition modes: supported.
- Index X/Y interleaving modes: per element and per 32B both supported.
- Output agen transposition modes: supported.

Parameters involved and not involved for this mode of operation are summarized after the task parameters detailed.

10.8 Table Reformatting

Table reformatting feature allows DLUT to offload the reorganizing of data from VPU, reducing energy and sometimes improving performance as well.

As the reformatting is simply data movement without involving any arithmetic processing on data, it is signed/unsigned and byte/halfword/word type agnostic. To simplify hardware verification, only Halfword type is supported. Index type must be unsigned Halfword, and entry type can be signed or unsigned Halfword. This is so the derivation from task_len_N1/N2 to index/output_agen_N1/N2 can be consistent.

The following is the definition of a table reformatting task.

- Input: accept N * P subtables, each is of length L and is stored in consecutive memory region, with each subtable being LP (line pitch) entries apart. P = 2/4/8/16/32, referred to as the parallelism in VPU lookup instructions and DLUT task definition. N is any integer, and basically an optional outer-loop number of iterations and number of rows of parallel subtables.
- DLUT is to reformat the table data and produce output: N blocks of P parallel subtables across 512-bit (or 32 halfowrds) VMEM superbank memory width.

Note that subtable length L does NOT have to be a multiple of (32/P). 32/P can be regarded as “stride” in parallel subtable organization, number of entries each subtable has in halfword-aligned 64 consecutive bytes. As hardware reads and writes P strides at a time, or P*L halfwords. L not being a multiple of 32/P means P*L halfwords not being 32 halfwords = 64 byes aligned. Since partial index read and output write transactions are supported in the common table lookup/interpolation modes, table reformatting mode takes advantage of this to allow flexibility that may lead to memory footprint saving.

The parallel subtable format is consistent with that of VPU 2/4/8/16/32-way parallel lookup/histogram instructions, as well as DLUT conflict-free lookup mode. Please consult Data Organization in Memory for VPU lookup/histogram data format.

Common table reformatting input/output data format is as follows:

Figure 19. Table reformatting input/output layout scheme

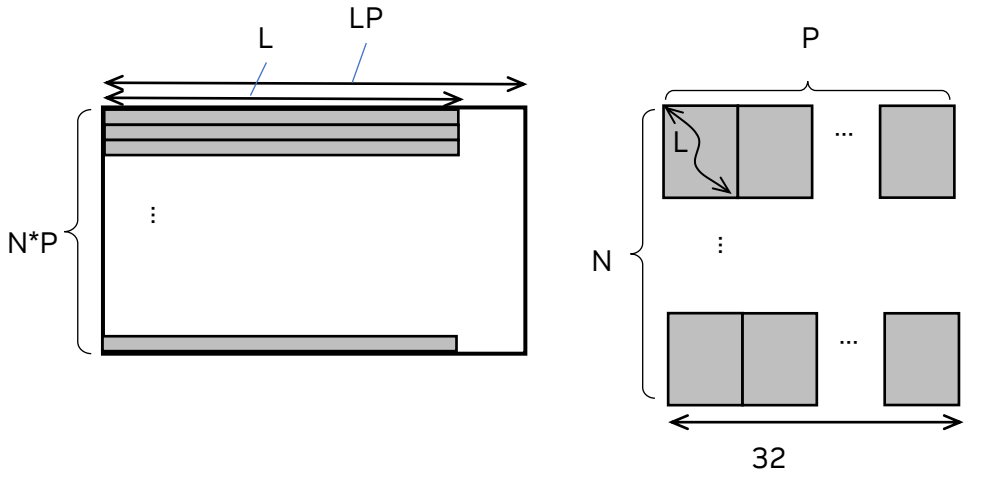


Table reformatting is basically accomplished through various transpose mode configured in the index agen with appropriate N1, N2, AMOD1, AMOD2 programming in index and output agens.

Line pitch (LP in the diagram) is constrained to 32*k + 32/P, with k configured as the transpose lane offset parameter.

The following are the parameters involved in a table reformatting task:

- task_mode = table reformatting
- index_type = unsigned halfword
- entry_type = signed or unsigned halfword
- task_len_N1 = P*L (index_agen_N1 = output_agen_N1 = P*L/32)
- task_len_N2 = N (index_agen_N2 = output_agen_N2 = N)
- index_addr = starting address of input (data to be reformatted)

- index_agen_amod1 = 64/P, advancing by one stride
- index_agen_amod2 = P*LP*2 – (L/(32/P) – 1) *(32/P)*2 = P*LP*2 – L*2 + 64/P
- index_agen_transp_mode = T<32/P>
- index_agen_lane_ofst = (LP – 32/P) / 32, the “k” in 32*k + 32/P
- index_interleave_format = don’t care
- out_addr = starting address of output (reformatted data)
- out_agen_amod1 = 64
- out_agen_amod2 = 64
- out_agen_transp_mode = None
- out_agen_lane_ofst = don’t care
- next_task: points to parameters of next task, 0 to terminate the task sequence

Note that these are recommended values to accomplish the table reformatting task as stated. Hardware just carries out index/output agen update as configured and passes data from the load stream to the store stream without verifying various parameter values.

For example, a table reformatting task with N = 2, P = 4, L = 32, line pitch = 32*k + 32/P = 40 (thus k = 1) shall have the following input and output organization:

Figure 20. Table reformatting input/output layout example

Input:

T0[0..31]	8 entry skipped
T1[0..31]	8 entry skipped
T2[0..31]	8 entry skipped
T3[0..31]	8 entry skipped
T4[0..31]	8 entry skipped
T5[0..31]	8 entry skipped
T6[0..31]	8 entry skipped
T7[0..31]	8 entry skipped

Output:

T0[0..7]	T1[0..7]	T2[0..7]	T3[0..7]
T0[8..15]	T1[8..15]	T2[8..15]	T3[8..15]
T0[16..23]	T1[16..23]	T2[16..23]	T3[16..23]
T0[24..31]	T1[24..31]	T2[24..31]	T3[24..31]
T4[0..7]	T5[0..7]	T6[0..7]	T7[0..7]
T4[8..15]	T5[8..15]	T6[8..15]	T7[8..15]
T4[16..23]	T5[16..23]	T6[16..23]	T7[16..23]
T4[24..31]	T5[24..31]	T6[24..31]	T7[24..31]

10.9 VPU/DLUT Interface

VPU/DLUT interface consists of a set of coprocessor control/status registers similar to how R5 would launch a VPU task. In VPU task launch, R5 software programs VPU starting PC, programs DMA to supply/consume input/output data, then commands VPU and DMA to go. In DLUT task launch, VPU software programs DLUT task parameter pointer (for first parameter block, which links to the next parameter blocks and so on) in the Coprocessor address space, allocates DLUT input/output regions in VMEM, then command DLUT to go by asserting a GPO signal.

The following subsections describe task control/status registers in the coprocessor address space, task parameter block data structure, and GPIO signaling.

10.9.1 Task Parameters

Task configuration parameters are stored in VMEM and have the following data structure per task. Note that any unused encoding option (for example, task mode 6 ~ 15) are reserved, and unused bit fields are ignored.

Table 46. DLUT task parameter data structure

Word/Field	Byte offset	Bits	Description
TASK_INFO	0x00		Task basic information
MODE		30:28	0x00: LOOKUP_1D, 1D lookup (one common table) 0x01: LOOKUP_2D, 2D lookup (one common table) 0x02: INTERP_1D, 1D lookup with linear interpolation (one common table) 0x03: INTERP_2D, 2D lookup with bilinear interpolation (one common table) 0x04: CONFLICT_FREE_2D_INTERP, conflict free 2D lookup with bilinear interpolation, (Halfword entry type only, 8 parallel tables) 0x05: TABLE_REFORMAT, table reformatting 0x06: INTERP_2D_AUTO_IDX, 2D lookup with interpolation by using automatically generated index data
INDEX_DATA_TYPE		25:24	0x1: IDX_H, Halfword 0x2: IDX_W, Word
ENTRY_DATA_TYPE		22:20	0x0: S8, signed Byte 0x1: S16, signed Halfword 0x2: S32, signed Word 0x4: U8, unsigned Byte 0x5: U16, unsigned Halfword

Word/Field	Byte offset	Bits	Description
			0x6: U32, unsigned Word
OUT_OF_RANGE_ HANDLING		19:18	0: disable 1: enable, return sentinel value for OOR lookup 2: enable, predicate off writing output for an OOR lookup
ROUND_MODE_NO_INTRP		17	0: truncate 1: round Only applied to modes 0 & 1 (without interpolation)
DUPLICATE_HANDLING		16	Duplicate detection and consolidation 0: disabled 1: enabled
FRACTION_BITS		4:0	U5, number of fraction bits to round/truncate, 0 ~ 16.
X_INT_LIMIT	0x04	17:0	U18, upper limit of X integer, or linear index for 1D lookup. For example, if there are 480 valid columns in a 2D table, or 480 valid entries in a 1D table, valid X integer range is 0 ~ 479, and this parameter should be configured as 479
Y_INT_LIMIT	0x08	17:0	U18, upper limit of Y integer. For example, if there are 240 valid rows in a 2D table, valid Y integer range is 0 ~ 239, and this parameter should be configured as 239.
X_OFFSET	0x0C	31:0	S32, number to add to X indices, with same number of fraction bits as input indices. This is to translate between global coordinates to local/tile coordinates.
Y_OFFSET	0x10	31:0	S32, number to add to Y indices, with same number of fraction bits as input indices. This is to translate between global coordinates to local/tile coordinates.
TASK_LEN	0x14		
N2		31:16	U16, number of rounds
N1		15:0	U16, number of elements to output per round of lookup
OOR_SENTINEL	0x18	31:0	Return value for out-of-range indices, use 8 LSBs (S8 or U8) for Byte entry type, 16 LSBs (S16 or U16) for Halfword entry type, all 32 bits (S32 or U32) for Word entry type.
TABLE_ADDR	0x1C	19:6	U20, pointer to table, 64 bytes aligned
TABLE_ADDR_OFFSET	0x20	17:6	S18, address update between rounds of lookup, 64 bytes aligned
TABLE_LINE_PITCH	0x24	15:0	U16. Note that this is in terms of number of table entries, and it's the line pitch itself, instead of providing k and line pitch being 32*k+n. For mode 4 (8 parallel table conflict-free lookup), line pitch must be 4k + 2, k being any integer.
AUTO_IDX_CFG	0x28		Auto-indexing configuration

Word/Field	Byte offset	Bits	Description
MODE		20	0: Index stream loads starting X/Y per round of task_len_N1 outputs 1: Index stream loads starting X/Y and step scale X/Y per round of task_len_N1 outputs
TRAVERSAL_DIR		16	0: Index traverses horizontally first, the raster-scan order, 1: Index traverses vertically first.
PATCH_WIDTH		15:8	U8, patch width
PATCH_HEIGHT		7:0	U8, patch height
INDEX_AGEN_CFG	0x2C		
TRANSP_MODE		30:28	0: None, no transposition 1: T1 2: T2 3: T4 4: T8 5: T16 6: T32
INTERLEAVE_FORMAT		24	0 : element-wise interleaved 1: 32B interleaved
LANE_OFST		11:0	U12, for transposed access, specify “k” in line pitch constraint 32k + t, t depending on transp_mode
TRANSP_MODE		30:28	0: None, no transposition 1: T1 2: T2 3: T4 4: T8 5: T16 6: T32
INTERLEAVE_FORMAT		24	0 : element-wise interleaved 1: 32B interleaved
LANE_OFST		11:0	U12, for transposed access, specify “k” in line pitch constraint 32k + t, t depending on transp_mode
OUTPUT_AGEN_CFG	0x30		
TRANSP_MODE		30:28	0: no transposition 1: T1 2: T2 3: T4 4: T8 5: T16 6: T32

Word/Field	Byte offset	Bits	Description
LANE_OFST		11:0	U12, for transposed access, specify “k” in line pitch constraint 32k + t, t depending on transp_mode
Reserved_1	0x34		
Reserved_2	0x38		
Reserved_3	0x3C		
INDEX_ADDR	0x40	19:0	U20, pointer to index array
INDEX_AGEN_AMOD1	0x44	17:0	S18, index address modifier for inner iterations
INDEX_AGEN_AMOD2	0x48	17:0	S18, index address modifier for outer iterations
OUTPUT_ADDR	0x4C	19:0	U20, pointer to output
OUTPUT_AGEN_AMOD1	0x50	17:0	S18, output address modifier for inner iterations
OUTPUT_AGEN_AMOD2	0x54	17:0	S18, output address modifier for outer iterations
Reserved_4	0x58		
NEXT_TASK	0x5C	19:2	U20, pointer to next task configuration data, zero for last task, 4 bytes aligned

Parameters relevant to various task modes are tabulated as follows. Blank entries are not used, and values are “don’t care.” Constrained values are also shown.

Table 47. DLUT parameter usage and constraints

Parameter	LU_1D	LU_2D	INTRP_1D	INTRP_2D	CF_INTRP_2D	TBL_RFMT	INTERP_2D_AUTO_IDX
MODE	0	1	2	3	4	5	6
INDEX_DATA_TYPE	Yes	Yes	Yes	Yes	Yes	IDX_H or IDX_W	IDX_W
ENTRY_DATA_TYPE	Yes	Yes	Yes	Yes	S16/U16	S16/U16	S16/U16
OUT_OF_RANGE_ HANDLING	Yes	Yes	Yes	Yes	Yes	disable	Yes
ROUND_MODE_NO_INTRP	Yes	Yes					
DUPLICATE_HANDLING	Yes	Yes	Yes	Yes			Yes
FRACTION_BITS	Yes	Yes	Yes	Yes	Yes	0	Yes
X_INT_LIMIT	Yes	Yes	Yes	Yes	Yes		Yes
Y_INT_LIMIT		Yes		Yes	Yes		Yes
X_OFFSET	Yes	Yes	Yes	Yes	Yes	0	Yes
Y_OFFSET		Yes		Yes	Yes	0	Yes
TASK_LEN N2	Yes	Yes	Yes	Yes	Yes	Yes	Yes
TASK_LEN N1	Yes	Yes	Yes	Yes	Yes	Yes	must match PW*PH
OOR_SENTINEL	Yes	Yes	Yes	Yes	Yes		Yes

Parameter	LU_1D	LU_2D	INTRP_1D	INTRP_2D	CF_INTRP_2D	TBL_RFMT	INTERP_2D_AUTO_IDX
TABLE_ADDR	Yes	Yes	Yes	Yes	Yes		Yes
TABLE_ADDR_OFFSET	Yes	Yes	Yes	Yes	Yes		Yes
TABLE_LINE_PITCH		Yes		Yes	Yes (4k+2)		Yes
INDEX_ADDR	Yes	Yes	Yes	Yes	Yes	Yes	Yes
INDEX_AGEN_TRANSP_MODE	Yes	Yes	Yes	Yes	Yes	Yes	0
INDEX_INTERLEAVE_FORMAT		Yes		Yes	Yes	0	0
INDEX_AGEN_LANE_OFST	Yes	Yes	Yes	Yes	Yes	Yes	
INDEX_AGEN_AMOD1	Yes	Yes	Yes	Yes	Yes	Yes	Yes
INDEX_AGEN_AMOD2	Yes	Yes	Yes	Yes	Yes	Yes	
OUTPUT_ADDR	Yes	Yes	Yes	Yes	Yes	Yes	Yes
OUTPUT_AGEN_TRANSP_MODE	Yes	Yes	Yes	Yes	Yes	Yes	Yes
OUTPUT_AGEN LANE_OFST	Yes	Yes	Yes	Yes	Yes	Yes	Yes
OUTPUT_AGEN_AMOD1	Yes	Yes	Yes	Yes	Yes	Yes	Yes
OUTPUT_AGEN_AMOD2	Yes	Yes	Yes	Yes	Yes	Yes	Yes
NEXT_TASK	Yes	Yes	Yes	Yes	Yes	Yes	Yes
AUTO_IDX_CFG							Yes

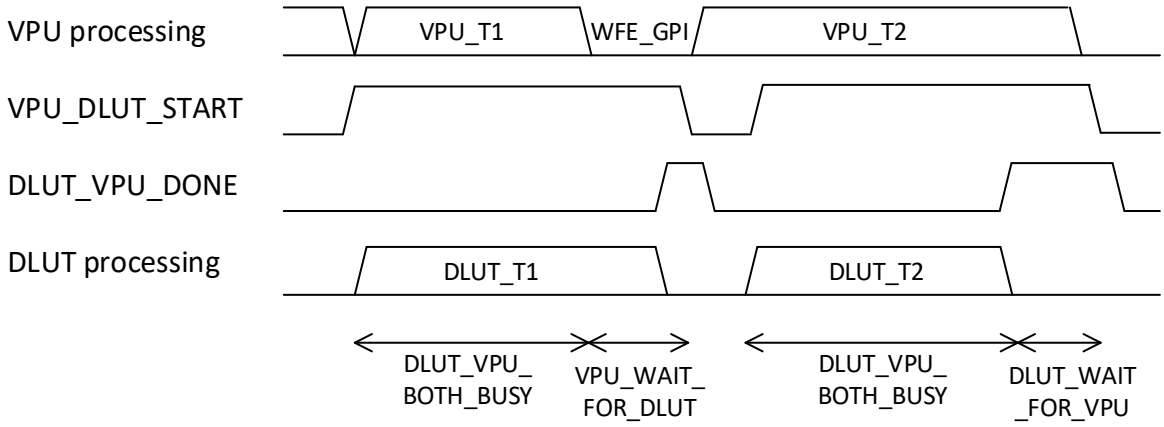
10.9.2 Interaction Sequence

VPU software is expected to interact with DLUT with the following sequence:

- Prepare task parameters in VMEM.
- Make sure previous task interaction is done, so that both VPU_DLUT_START (GPO[10]) and DLUT_VPU_DONE (GPI[10]) signals are low.
- Write starting address of task parameters in VPU_DLUT_TASK register.
- Assert VPU_DLUT_START
- Go on to execute other tasks, and when DLUT outcome is needed (no more independent tasks to run), executes WFE_GPI to wait for DLUT task completion in low power mode
- DLUT executes requested lookup tasks; there can be multiple tasks per interaction.
- DLUT asserts DLUT_VPU_DONE.
- VPU, upon sensing assertion of DLUT_VPU_DONE, resumes operations, de-asserts VPU_DLUT_START before go on to execute the next task.
- DLUT, upon sensing de-assertion of VPU_DLUT_START, de-asserts DLUT_VPU_DONE to complete the current round of interaction.

Two instances of typical VPU/DLUT interaction is shown in the following diagram:

Figure 21. VPU/DLUT interaction timing diagram



VPU is the master in the interaction. Before VPU_DLUT_START is asserted, software should ensure that:

- Configuration parameters for the requested DLUT tasks are ready in VMEM for DLUT to consume.
- Input index and table data for these tasks are either already in VMEM for DLUT to consume, or they will be ready when DLUT gets to the task that consumes the data by nature of task sequencing.
- Space needed for DLUT to write lookup/interpolation outcome for these tasks, are either all available in VMEM for DLUT to write, or they will be available when DLUT gets to the task that writes the outcome by nature of task sequencing.

DLUT executes requested tasks sequentially, so it is possible to have data and/or space dependency among DLUT tasks in the same sequence. Task i output can be safely consumed as task i+1 input, and task i input, if not dependent upon by subsequent tasks, can be overwritten by a subsequent task.

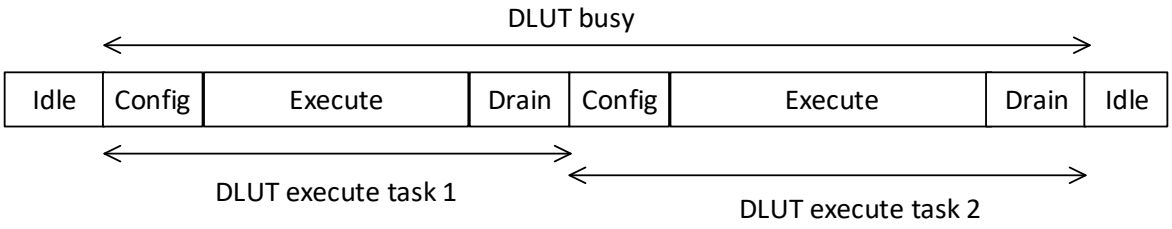
For example, we can have one task performing table reformatting, and the very next lookup task using the reformatted table.

If it also allowed to have space dependency within one task. For example, if the index and outcome group pitches are programmed appropriately, we can overwrite lookup/interpolation outcome onto the index array, if the index data is only consumed once and never needed again.

In case VPU software does not follow the recommended protocol, and de-asserts VPU_DLUT_START before DLUT completed the task(s) and asserts DLUT_VPU_DONE, DLUT detects the issue and error-halt, as described in 10.9.4

DLUT internally has multiple stages of processing. Processing of a sequence of two tasks is shown in the following diagram:

Figure 22. DLUT processing stages



10.9.3 Incorrect Task Configuration

DLUT incorrect task configuration is defined as

- Having a parameter value outside valid range.
  - For example, task mode is defined as a 3-bit field, with values 0 ~ 5 mapped to valid modes, and values 6 ~ 7 being reserved. A task configured with task mode 6 ~ 7 is deemed as having incorrect configuration.
  - As another example, a number of fraction bits is supposed to be 0 ~ 16 in the 5-bit parameter, so values 17 ~ 31 are invalid. A task configured with number of fraction bits being 17 ~ 31 is deemed having incorrect configuration.
- Having a parameter value not allowed for the operation mode.
  - In a 2D conflict-free lookup with interpolation task, index type not being unsigned Halfword or unsigned Word, or entry type not being signed or unsigned Halfword.
  - In a 2D conflict-free lookup with interpolation task, line pitch not being 4k + 2, k being any integer.
  - In a table reformatting task, index type not being unsigned Halfword, or entry type can be signed or unsigned Halfword.
  - In a table reformatting task, out-of-range handling not being disabled, fraction_bits, X offset, Y offset not being zero.
  - In a table reformatting task, index interleave format not being element-wise interleaved.
  - In auto-indexing mode, index type not being IDX_W, index TRANSP_MODE not being None, index interleave format not being element-wise interleaved.
- Having disallowed index/entry type and transpose mode combination in agen:
  - Byte with T1 transpose is not allowed.
  - Halfword with T32 transpose is not allowed.
  - Word with T16 and T32 transpose is not allowed.
- Having inconsistent/conflicting parameters:
  - In auto-indexing mode, task_len_N1 not matching PATCH_WIDTH * PATCH_HEIGHT.
  - In auto-indexing mode, when traversing horizontally first, PATCH_WIDTH < 8.
  - In auto-indexing mode, when traversing vertically first, PATCH_HEIGHT < 8.

Handling of incorrect configuration is described in the next subsection.

The following cases of parameter configuration seem “strange”, but are NOT considered incorrect configuration, meaning hardware would carry out the task as configured, mostly because it would be cumbersome to detect:

- Having nonzero value in an unused bit location. For example, parameter word 0 bit 3 is not used (between task mode and index type), so a task configured with nonzero value there does not cause incorrect behavior, nor trigger configuration error.
- Having nonzero value in an unused bit field for that operating mode. For example, Y-related fields are not used for 1D lookup (with or without interpolation) modes. Such fields are simply ignored, so nonzero values there does not cause incorrect behavior, nor trigger configuration error.
- Starting address for index/table/output or next task parameter block not aligned to required address alignment. Address alignment is forced by hardware ignoring 1, 2, or 6 LSBs of the byte address from agen base. This is consistent with address alignment handling in VPU load/store instructions (see Memory Address Alignment).
- In a table reformatting task, normally we use None (no transposition) in one end and a transposition mode (T1/T2 etc) in the other end. If we have None-to-None or Transpose-to-Transpose combinations, the agen programming is likely to be incorrect. However, hardware in table reformatting will just perform input read and output write as configured by the agen parameters, not checking for transpose combinations between read and write.
- Infinite task sequence by task parameter blocks forming a cycle. It is most likely incorrectly programmed, but the hardware does not have an easy way to detect such a condition. In this case, DLUT would not terminate, and VPU software would wait forever in the WFE_GPI state. The R5 processor runs an RTOS and is able to detect and handle such error conditions.

10.9.4 DLUT Execution States, Error Handling, Halt and Debug

10.9.4.1 Normal Execution Behavior and Conceptual State Diagram

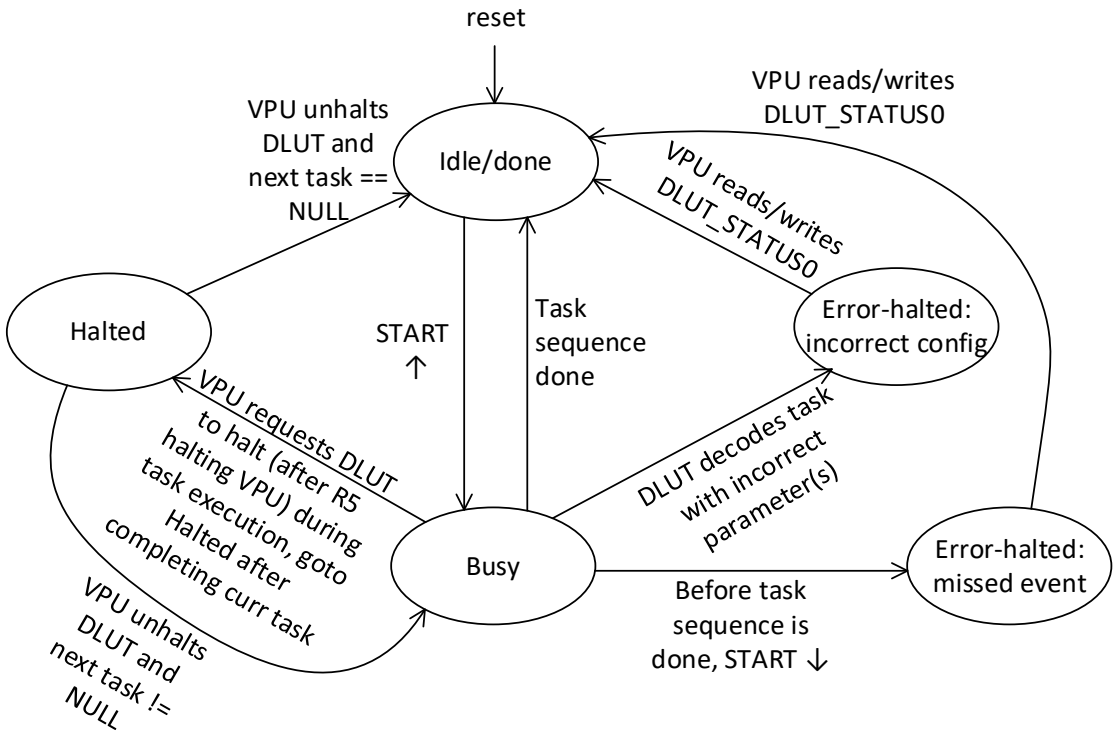
Upon reset, DLUT execution state becomes idle.

After VPU asserts VPU_DLUT_START to start DLUT operation, DLUT execution state becomes busy.

After DLUT completes configured task sequence, DLUT asserts DLUT_VPU_DONE and execution state returns to being idle. DLUT_VPU_DONE is asserted until VPU_DLUT_START is deasserted.

DLUT execution state transition behavior can be described by the following conceptual state diagram. Transitions other than normal execution, between idle/done and busy, shall be explained in the following subsections.

Figure 23. DLUT execution state conceptual state diagram



10.9.4.2 Error Handling

In case of an incorrectly configured task, DLUT would execute (correctly configured) proceeding tasks to completion, change execution state from “1: busy” to “2: halted due to incorrect configuration”, show number of tasks completed successfully, say K, raise DLUT_VPU_DONE, and show pointer to task parameter for the incorrectly configured task, which is task K+1.

In case of a missed event, defined as VPU_DLUT_START being asserted to start a DLUT task sequence, then deasserted prematurely, before DLUT asserts DLUT_VPU_DONE to convey that the sequence is completed. This is in violation of the protocol and can lead to race conditions in the VPU/DLUT interaction. DLUT records the issue, and after the currently executing task is completed, raises DLUT_VPU_DONE and goes to error-halted state, showing execution state being “3: halted due to missed event”. The number of tasks completed, and current task parameter pointer shown in status registers, shall be dependent on when the missed event was detected with respect to the tasks being executed. If number of tasks completed is shown as K, it’s possible for task parameter pointer to point to task K, in case missed event was detected when task K is being executed, and next task not yet parsed, of it can point to task K+1, in case missed event

was detected when task K is just finished, task K+1 parameters are parsed but task K+1 execution not yet started.

In both error cases, DLUT stays in the appropriate error-halted state until VPU software acknowledges the error by reading DLUT_STATUS0.EX_STATE and writing the same value (2 or 3) back to the same register address. Such write would clear DLUT execution state to (0: idle or done) and get DLUT ready for next task launch. Until such a write, DLUT status registers showing execution state, number of tasks completed and pointer to task parameter, as well as DLUT_VPU_DONE (having been raised to high) all remain unchanged.

For an incorrectly configured task error, there is precise definition of which is the first task being incorrectly configured, so number of tasks completed and pointer to parameter of currently executing task shall be kept consistent. For example, if task 1 (being starting task of the sequence) and 2 are fine and task 3 is incorrectly configured, DLUT shall complete the first 2 tasks, signals DLUT_VPU_DONE, show 2 tasks completed and points to parameter block of task 3.

For missed event errors, hardware detects the error but runs current task to completion, so if it's detected during execution of task 3, task 3 is completed, DLUT signals DLUT_VPU_DONE, and status registers show 3 tasks completed and points to parameter block of task 3, thereby keeping these two status registers consistent. However, there is no precise way to predict when hardware detects the error relative to task sequence execution, so it is not feasible to predict exactly how many tasks will be completed if a missed event occurs after a set time after kicking of a task sequence, even if the task sequence is fully known (parameters and index/table data). This is because DLUT can be stalled by VPU and DMA upon VMEM superbank contentions.

DLUT incorrect configuration and DLUT missed event are among VPS error clauses, and in VPS error handling, each clause can be configured to error-halt both VPU and DLUT, or to continue with VPU execution.

Both kinds of DLUT errors are recoverable through normal VPU software interaction, unlike VPU error-halt, which can only be recovered by VPS-level reset that resets both VPU and DLUT. Thus, if deemed appropriate by the programmer, VPU software can carry out quicker recovery from DLUT errors, as opposed to PVA-level reset which will take longer.

10.9.4.3 R5 Halt/Unhalt

We provide a mechanism for R5 processors to halt/unhalt VPU. Halting means suspension of operation, and unhalting means resume of operation. Halting and subsequent unhalting should not alter the eventual outcome. Both VPU and DLUT are NOT expected to suspend operation immediately, but to do so when it's convenient to do so. VPU has a processor pipeline, so it would suspend after its pipeline is drained. DLUT is a decoupled engine with multiple stages pipelined together and configured to process a task at a time, each task producing task_len_N1 x task_len_N2 outputs, and it's only convenient to suspend after the current task is done.

In case VPU is halted from R5 writing VPS Config register to halt VPU, VPU, after it has halted, forwards the halt request to DLUT. DLUT handling is as follows:

- If DLUT is idle, it remains idle.
- If DLUT has already error-halted from a previous task, it remains error-halted.
- Otherwise (DLUT is processing a task), DLUT would attempt to run that task to completion.
  - After the task is completed without error, DLUT goes to Halted execution state.
  - In case an error occurs, DLUT goes to Error-Halted execution state

When VPU is subsequently unhalted by R5, VPU requests DLUT to unhalt:

- If DLUT is idle, it remains idle.
- If DLUT is error-halted, it remains error-halted.
- Otherwise (DLUT is in Halted state), it moves on to process the next task.
  - If there is no next task it becomes idle
  - Otherwise, it becomes busy processing the next task.

The VPS config register VPS_STATUS.EXE_STATE showing halted when VPU has been halted, and it's possible that DLUT would remain active/busy for a while, until current task is completed or terminated with error.

10.9.4.4 Debug Mode

In case VPU enters debug mode, DLUT continues to execute until configured task sequence is completed and raises DLUT_VPU_DONE. This is consistent with DMA's handling of VPU entering debug mode.

10.9.5 Other Control/Status Registers

DLUT contributes one VPS error cause, DLUT incorrect task configuration. See 13.2 for details.

DLUT contributes a few counters to the VPS performance monitor feature. See PVA VPS IAS VPS Register Spec for details.

DLUT is part of VPS, and its clock gating behavior shall be consistent with VPS clock gating behavior. See PVA VPS IAS VPS Register Spec for details.

Chapter 11. Register Spec

11.1 VPS Coprocessor Registers

The VPS Coprocessor registers are, in general, accessible by VPU coprocessor read/write instruction.

The Revision ID register is described in the following subsection. The DLUT control/status registers are described in DLUT Task Control/State Registers.

11.1.1 Revision ID Register

The revision ID register is read-only. The purpose of the revision ID is to allow VPU software to read and differentiate among versions. revisions and instances of VPU module when needed.

Tie-in at PVA module boundary drives PVA_ID. Tie-in at VPS module boundary drives the VPU_ID. PVA revision and release IDs are identical copies from PVA-top level PVA_CFG registers.

Table 48. VPU revision ID register

Register/Field	RW	Bits	Reset	Description
REVISION_ID 0x200				Revision ID register
PVA_INST_ID	R	31:28	0/1	PVA instance ID 0: PVA0 1: PVA1
VPU_INST_ID	R	27:24	0/1	VPU instance ID 0: VPU0 1: VPU1
REV_ID	R	23:20	2	PVA revision ID, matching PVA_CFG_PVA_ID.PVAREVID 1: T19x PVA 1.0 2: T23x PVA 2.0
REL_ID	R	15:0	0x36	PVA release ID, matching PVA_CFG_PVAREL_ID.PVAREL_REV

11.1.2 DLUT Task Control/Status Registers

There exists configuration and status registers on the VPU's coprocessor space. VPU software can read/write these registers via the CP_LD and CP_ST instructions.

Table 49. VPU DLUT task control/status registers

Register/Field	RW	Bits	Reset	Description
VPU_DLUT_TASK 0x800				VPU-DLUT task configuration
CFG_ADDR	RW	19:2	0	Configuration byte address in VMEM for DLUT to execute for the next task. A byte address is to be written to the 32-bit register. 2 LSBs are non-writable and thus dropped, enforcing the 32-bit alignment. 12 MSBs are non-writable, since VMEM has an address range of 1 MB (20-bit byte address).
DLUT_STATUS0 0x804				DLUT execution status 0
EX_STATE	RW	2:0	0	Execution state 0: idle or done 1: busy 2: error-halted due to incorrect configuration 3: error-halted due to missed event 4: halted (by R5 halt, to resume later) Upon DLUT being in error-halted state (2 or 3), before the next DLUT task can be started, VPU software should read this register and write the contents back to this register to return the status to idle/done.
DLUT_STATUS1 0x808				DLUT execution status 1
NTASKS_CMPLTD	R	7:0	0	Number of tasks completed in current task launch (writing of CFG_ADDR). This would clear upon task launch and increment by 1 at a time as DLUT completes tasks. This would stay at number of tasks configured when EX_STATE = done.

Register/Field	RW	Bits	Reset	Description
				In case number of tasks configured exceeds 255, instead of rolling over to 0, the count would saturate to 255.
DLUT_CURR_TASK 0x80C				
PARAM_ADDR	R	19:2	0	VMEM address of task parameter block that DLUT is executing

Chapter 12. General Purpose Input/Output

12.1 VPU/DMA Control Interface

VPU/DMA control interfaces include:

- VPU-DMA start event signaling
- DMA-VPU done event signaling

VPU has 32-bit General Purpose Input (GPI) for DMA-VPU event signaling, and 32-bit General Purpose Output (GPO) for VPU-DMA event signaling. VPU/DMA event signaling is described in detail in the PVA DMA IAS, and is summarized here.

The VPU signals DMA to start DMA transfer for **read** (reading from system memory to VMEM), **store** (writing from VMEM to system memory), and **config** (reading address/data pairs in VMEM to configure registers and descriptors), and DMA signals back when the transfer is completed.

Table 50. VPU/DMA control signal list

Signal	Driver	GPIO	Notes
vpu_dma_read_start[6:0]	VPU	GPO[22:16]	Start DMA read from external mem into VMEM, action upon positive edge. VPU software asserts to send the request, and deasserts upon detecting the corresponding DONE signals.
vpu_dma_store_start[6:0]	VPU	GPO[29:23]	Start DMA store to external mem from VMEM, action upon positive edge. VPU software asserts to send the request, and deasserts upon detecting the corresponding DONE signals.
vpu_dma_config_start	VPU	GPO[4]	Start DMA write config space from addr/data pairs in VMEM, action upon positive edge. VPU software asserts to send the request, and deasserts upon detecting the corresponding DONE signals.

Signal	Driver	GPIO	Notes
dma_vpu_read_done[6:0]	DMA	GPI[22:16]	DMA read done, level, cleared upon corresponding read_start being deasserted
dma_vpu_store_done[6:0]	DMA	GPI[29:23]	DMA store done, level, cleared upon corresponding store_start being deasserted
dma_vpu_config_done	DMA	GPI[4]	DMA config done, level, cleared upon vpu_dma_config_start being deasserted
dma_hwseqstart_vpu	DMA	GPI[15]	DMA HWSeq start, DMA telling VPU to start processing a tile
	DMA	GPI[14]	
vpu_hwseqdone_dma	VPU	GPO[15]	VPU HWSeq done, VPU telling DMA that processing is done for a tile
	VPU	GPO[14]	

12.2 Summary of GPI/GPO Signals

A few additional GPI/GPO signals are used for debug and performance monitoring. Full GPI/GPO allocation is as follows.

Table 51. VPU GPI/GPO signal list

GPIO	Signal	Driver	Receiver	Value after reset	Notes
GPI[29:23]	dma_vpu_store_done[6:0]	DMA	VPU	0	
GPI[22:16]	dma_vpu_read_done[6:0]	DMA	VPU	0	
GPI[15:14]	dma_vpu_hwseqs	DMA	VPU	0	
GPI[10]	dlut_vpu_done	DLUT	VPU	0	DLUT telling VPU it's done
GPI[9]	icache2vpu_config _invalidate_rdy	I-cache	VPU	1	
GPI[8]	vps_sw_event	SEC	VPU	0	
GPI[7]	icache2vpu_gpio_invalidate_ all_rdy	I-cache	VPU	1	
GPI[6]	icache_vpu_prefetch_done	I-cache	VPU	1	
GPI[5]	icache_vpu_prefetch_rdy	I-cache	VPU	1	
GPI[4]	dma_vpu_config_done	DMA	VPU	0	
GPI[0]	vpu_cntlin_debug	VPU GPO[30]	VPU	0	Debug control in, loop back from vpu_cntlout_debug

GPIO	Signal	Driver	Receiver	Value after reset	Notes
GPO[31]	vpu_testfail_debug	VPU	n/a	0	Test done pass/fail signaling for simulation & debug, 0 = pass, 1 = fail, connected to testbench
GPO[30]	vpu_cntlout_debug	VPU	VPU GPI[0]	0	Debug control out
GPO[29:23]	vpu_dma_store_start[6:0]	VPU	DMA	0	
GPO[22:16]	vpu_dma_read_start[6:0]	VPU	DMA	0	
GPO[15:14]	vpu_dma_hwseq	VPU	DMA	0	
GPO[10]	vpu_dlut_start	VPU	DLUT	0	VPU telling DLUT to start
GPO[4]	vpu_dma_config_start	VPU	DMA	0	
GPO[3]	vpu_perf_monitor_en	VPU	VPS config	0	VPU software drives this pin to 1: enable or 0: disable performance monitor counters for optional kernel/loop level control
GPO[2]	vpu_start_r5	VPU	SEC	0	
GPO[1]	vpu2icache_gpio_invalidate_all	VPU	I-cache	0	
GPO[0]	vpu_stimwd_debug	VPU	n/a	0	STIM window, for power test case simulation & debug, connected to testbench

Note that GPI reset values are driven by various driver modules outside VPU, so the reset values are applied when the corresponding module (DMA or SEC) is reset. I-cache and DLUT are reset with VPU. Unused GPIs are tied to 0.

Chapter 13. Design for Test and Safety

13.1 Debug Features

The VPU has a CoreSight/APB-based debug interface that is hooked up to system-level JTAG interface and is accessible through JTAG or through CPU software. The VPU debug features are:

- Enter/exit debug state.
- Read program memory.
- Invalidate I-cache. Debug writing program memory is implemented by writing to system memory then invalidate I-cache, which drives I-cache to refetch from system memory.
- Read/write VMEM.
- Directly feed the instruction word to be executed.
- Read/write processor registers, including PC, scalar/vector register file, HW loop control registers, predicate register file, and agen config register file (through injection of instruction sequence to store the relevant register into VMEM then reading VMEM).
- Read/write GPO, read GPI (through injection of instruction sequence).
- Read/write PC (through injection of instruction sequence).
- Read/write SES (shadow execution state) register. The VPU execution state (active, WFE_R5, WFE_GPI, error-halted, or halted) before transitioning into debug state is saved in this register. It is read/write accessible by debug software via OCD_LD/OCD_ST instructions and can be changed to drive VPU to a different state after exiting the debug state.
- Read/write DLUT configuration/status registers via CPLD/CPST.
- Single step execution.
- 24 watch or break points (combined, for example we can configure them into 18 watch points and 6 break points.
  - Hardware break points: when PC matches one of the configured break point PC values, VPU enters debug state.
  - Single watch points: when VMEM read or write (need to specify which direction) address from designated load/store slot matches with one of the configured addresses, VPU enters debug state. (Only starting address, not an address range

in case of vector load/store, so only matches starting/base address in case of transposition and histogram; table lookup is read-only access so is not covered)

- Range watch points: Two single watch points can be configured as lower/upper bounds of an address range. Reading or writing with any base address in the range would trigger the request to enter VPU into debug state.
- Each watch point is specified as a full 32-bit byte address, with designation of load/store and which memory slot (M0/M1/M2). A single watch point on IDE does not specify memory slot and costs 3 watch point resources. A range watch point on IDE costs 6 watch point resources.
- Note that watch point does not capture aliased accesses. For example, any binary address xxxx_xxxx_xxxx_00xx_0100_0000_0000_0000 maps to address 0x4_0000, so a watch point on 0x4_0000 does not capture accesses to 0x14_0000, which also maps to the same physical address.

> SW break points (unlimited), program contents substituted with SWBRK instruction, upon execution of which, VPU is instructed to enter debug state.
> SWBRK being executed when DBGEN = 1 will transition VPU to the debug state.
> SWBRK being executed when DBGEN = 0 constitutes an illegal debug error and is captured in error logging (see Soft Error Cases and Handling), with the option to either continue execution (treating SWBRK as NOP) or error-halt.
> Cross-trigger input/output, to optionally allow other processors enter/exit debug to cause VPU to enter/exit debug, and vice versa.
> First 64 bytes of VMEM is reserved for debug software as staging area to query and save/restore registers or VMEM data.

13.2 Soft Error Cases and Handling

VPU has the following features to detect various “soft” errors, so named because most likely they occur during software development, so these features can be regarded as design-for-test and debug features:

> Illegal instruction detection
> Scalar divide by zero error
> Floating-point invalid outcome
> Illegal debug
> Illegal instruction from alignment stage
> DLUT task incorrect configuration
> Coprocessor load/store access error
> DLUT missed event

Each error case is configurable whether to error-halt in the ERR_HNDL_CFG register. When an error occurs, an interrupt is sent to the SEC block (safety and event control) in PVA top level, where the interrupts are optionally forwarded to VIC (vectored interrupt

controller) then to the R5. R5 and/or SEC can optionally forward error events to system-level error collator.

Error handling and context capture for each soft error follows.

Table 52. VPU soft error cases and handling

Error case	**Illegal instruction**
Where/when it is detected	VPU instruction decode stages. The decode stage detects illegal instruction in each 32-bit instruction for scalar, vector, math units.
Error handling	The erroneous instruction is sent down the pipeline. VPU enters Error-halted state right after the erroneous instruction when configured so in ERR_HNDL_CFG. SEC, when properly configured, detects VPU execution transition into Error-Halted state and sends an interrupt to R5. A dedicated interrupt, invalid_instruction_error, is always sent independently of the ERR_HNDL_CFG setting.
Context captured	PC and timestamp.
SEC signal	vpu_sec_illinstr_uncorrerr
Additional details	

Error case	**Scalar divide by zero**
Where/when it is detected	Scalar divider unit, when zero divisor is supplied for a divide operation.
Error handling	Max unsigned int value (0xFFFF_FFFF) is returned as the quotient. VPU enters Error-halted state when it's configured so in ERR_HNDL_CFG. SEC, when properly configured, detects VPU execution transition into Error-Halted state and sends an interrupt to R5.
Context captured	PC and timestamp.
SEC signal	n/a
Additional details	

Error case	**Floating-point invalid**
Where/when it is detected	Scalar and vector FP unit, when invalid (NaN) outcome is generated.
Error handling	VPU enters Error-halted state when it's configured so in ERR_HNDL_CFG. SEC, when properly configured, detects VPU execution transition into Error-Halted state and sends an interrupt to R5.
Context captured	PC and timestamp.
SEC signal	n/a
Additional details	

Error case	Illegal debug
Where/when it is detected	VPU executing a SWBRK (software break point) instruction when debug is disabled on the debug interface.
Error handling	VPU enters Error-halted state when it's configured so in ERR_HNDL_CFG. Otherwise, VPU ignores the SWBRK instruction, treating it as an NOP instruction. SEC, when properly configured, detects VPU execution transition into Error-Halted state and sends an interrupt to R5.
Context captured	PC and timestamp.
SEC signal	n/a
Additional details	

Error case	Illegal instruction from alignment stage
Where/when it is detected	VPU alignment stage detects illegal instruction.
Error handling	VPU enters Error-halted state when it's configured so in ERR_HNDL_CFG. SEC, when properly configured, detects VPU execution transition into Error-Halted state and sends an interrupt to R5.
Context captured	PC and timestamp.
SEC signal	n/a
Additional details	

Error case	DLUT task incorrect configuration
Where/when it is detected	DLUT executes a task that is incorrectly configured.
Error handling	VPU enters Error-halted state when it's configured so in ERR_HNDL_CFG. DLUT terminates task sequence upon detection by asserting DLUT_VPU_DONE, and shows execution state as error-halted to incorrect configuration. SEC, when properly configured, detects VPU execution transition into Error-Halted state and sends an interrupt to R5.
Context captured	Timestamp.
SEC signal	n/a
Additional details	

Error case	Coprocessor access error
Where/when it is detected	Coprocessor load/store instruction reading from a invalid/reserved address, or writing to a read-only or invalid/reserved address.
Error handling	VPU enters Error-halted state when it's configured so in ERR_HNDL_CFG. SEC, when properly configured, detects VPU execution transition into Error-Halted state and sends an interrupt to R5.
Context captured	PC and timestamp.
SEC signal	n/a
Additional details	

Error case	DLUT missed event
Where/when it is detected	VPU_DLUT_START is de-asserted before DLUT_VPU_DONE (GPI) is asserted.
Error handling	VPU enters Error-halted state when it's configured so in ERR_HNDL_CFG. DLUT terminates task sequence when current task is completed, by asserting DLUT_VPU_DONE, and shows execution state as error-halted to missed event. SEC, when properly configured, detects VPU execution transition into Error-Halted state and sends an interrupt to R5.
Context captured	PC and timestamp.
SEC signal	n/a
Additional details	

13.3 Safety Features

VPS has the following error handling as safety features. These errors most likely originated from some permanent or transient hardware fault:

- Illegal instruction
- I-cache ECC single-bit error (correctable)
- I-cache ECC double-bit error (uncorrectable)
- VMEM per-byte parity error

Note that divide-by-0 and floating-point invalid detection are not safety features, but design-for-test/debug features.

Illegal instruction error source is configurable whether to error-halt in the ERR_HNDL_CFG register.

When an error occurs, an interrupt is sent to the SEC block (safety and event control) in PVA top level, where the interrupts are optionally forwarded to VIC (vectored interrupt controller) then to the R5. R5 and/or SEC can optionally forward error events to system-level error collator.

Safety-related error handling and context capture for each error follows.

Table 53. VPU safety error cases and handling

Error case	I-cache ECC single-bit error
Where/when it is detected	I-cache reading an entry upon request from VPU instruction fetch
Error handling	Erroneous instruction word is corrected on the fly before returning it to VPU instruction fetch/align unit. An error interrupt is sent by I-cache to SEC. R5 software can choose to either A) Respond to the interrupt, invalidate the cache line (hopefully before a 2nd error occurs), or

Error case	I-cache ECC single-bit error
	B) Ignore the interrupt. Note that VPU continues execution in this case.
Context captured	None
SEC signal	icache_memr_sec_correrr
Additional details	ECC is applied on 32 bytes basis. The corrected data will arrive at the VPU fetch/align unit on cycle later. The VPU fetch/align unit would invalidate the erroneous instruction data and send one cycle of bubble down the pipeline.

Error case	I-cache ECC double-bit error
Where/when it is detected	I-cache reading an entry upon request from VPU instruction fetch
Error handling	An error interrupt is sent by I-cache to SEC. R5 software can choose to either A) Halt/reset VPU immediately, or B) Give VPU a chance (until watchdog timer expires) to run to task completion. VPU continues, decoding/executing returned instruction data from I-cache.
Context captured	None
SEC signal	icache_memr_sec_correrr
Additional details	ECC is applied on 32 bytes basis

Error case	VMEM parity error
Where/when it is detected	VMEM reading an entry upon request from VPU or external host (DMA, R5 or outside-PVA host)
Error handling	Erroneous data is retuned to VPU or external host. VPU continues execution. An error interrupt is sent to SEC. R5 software can choose to respond to the interrupt or ignore it.
Context captured	None
SEC signal	vmem_memr_sec_dperr
Additional details	Parity is applied per byte

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation (“NVIDIA”) makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA, the NVIDIA logo, CUDA, CUPVA, Orin, Thor are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

VESA DisplayPort

DisplayPort and DisplayPort Compliance Logo, DisplayPort Compliance Logo for Dual-mode Sources, and DisplayPort Compliance Logo for Active Cables are trademarks owned by the Video Electronics Standards Association in the United States and other countries.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

Arm

Arm, AMBA, and ARM Powered are registered trademarks of Arm Limited. Cortex, MPCore, and Mali are trademarks of Arm Limited. All other brands or product names are the property of their respective holders. ”Arm” is used to represent ARM Holdings plc; its operating company Arm Limited; and the regional subsidiaries Arm Inc.; Arm KK; Arm Korea Limited.; Arm Taiwan Limited; Arm France SAS; Arm Consulting (Shanghai) Co. Ltd.; Arm Germany GmbH; Arm Embedded Technologies Pvt. Ltd.; Arm Norway, AS, and Arm Sweden AB.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Copyright

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.NVIDIA.com