

Qualcomm[®] Hexagon[™] V75 HVX

Programmer's Reference Manual

80-N2040-58 Rev. AB

January 24, 2024

All Qualcomm products mentioned herein are products of Qualcomm Technologies, Inc. and/or its subsidiaries.

Qualcomm and Hexagon are trademarks or registered trademarks of Qualcomm Incorporated. Other product and brand names may be trademarks or registered trademarks of their respective owners.

This technical data may be subject to U.S. and international export, re-export, or transfer ("export") laws. Diversion contrary to U.S. and international law is strictly prohibited.

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

1 Introduction	8
1.1 SIMD coprocessor	8
1.2 HVX features.....	9
1.2.1 Vector length	9
1.2.2 Vector contexts	9
1.2.3 Memory access	10
1.2.4 Vector registers	11
1.2.5 Vector compute instructions	12
1.3 Technical assistance	12
 2 Registers	 13
2.1 Vector data registers	13
2.1.1 Unaligned vector pairs	13
2.1.2 VRF to GRF transfers	14
2.2 Vector predicate registers	14
 3 Memory.....	 15
3.1 Alignment.....	15
3.2 HVX local memory: VTCM	15
3.3 Scatter and gather	16
3.4 Memory type.....	17
3.5 Nontemporal	17
3.6 Permissions	17
3.7 Ordering	17
3.8 Atomicity	18
3.9 Maximizing performance of the vector memory system	18
3.9.1 Minimize VMEM access	18
3.9.2 Use aligned data	18
3.9.3 Avoid store to load stalls.....	19
3.9.4 L2FETCH	19
3.9.5 Access data contiguously	19
3.9.6 Use nontemporal for final data.....	19
3.9.7 Scalar processing of vector data	19
3.9.8 Avoid scatter/gather stalls	20

4 Vector instructions	21
4.1 VLIW packing rules	21
4.1.1 Double vector instructions	21
4.1.2 Vector instruction resource usage	22
4.1.3 Vector instruction	22
4.2 Vector load/store	23
4.3 Scatter and gather	24
4.4 Memory instruction slot combinations	25
4.5 Special instructions	25
4.5.1 Histogram	25
4.6 Qfloat	26
4.6.1 QFloat best practices	27
4.7 Instruction latency	27
4.8 Slot/resource/latency summary	28
5 HVX PMU events	29
6 HVX instruction set	31
6.1 HVX ALL compute resource	33
Histogram 33	
Weighted histogram	36
6.2 HVX ALU double resource	40
Predicate operations 40	
Combine	42
In-lane shuffle	43
Swap	45
Sign/zero extension	47
Arithmetic	50
6.3 HVX ALU resource	54
Predicate operations 54	
Byte-conditional vector assign	55
Min/max	56
Absolute value	59
Arithmetic	61
Arithmetic with carry bit	64
Logical operations	66
Copy	67
Temporary assignment	69
Average	70
Compare vectors	74
Conditional accumulate	82
Mux select	85

Saturation.....	87
In-lane shuffle	89
6.4 HVX debug.....	91
Extract vector element 91	
6.5 HVX gather double resource	93
Vector gather 93	
6.6 HVX gather	95
Vector gather 95	
6.7 HVX load.....	98
Load - aligned 98	
Load - immediate use	104
Load - temporary immediate use	107
Load - unaligned	110
6.8 HVX MPY double resource	112
3 × 3 multiply for 2 × 2 tile 112	
Arithmetic widening.....	125
Multiply with 2-wide reduction.....	128
Lookup table for piecewise from 64-bit scalar	133
Multiply with piecewise add/sub from 64-bit scalar.....	134
Multiply-add	135
Multiply - vector by scalar	140
Multiply - vector by vector	144
Multiply - half precision vector by vector	148
Integer multiply - vector by vector.....	151
Integer multiply (32×16).....	153
Integer multiply accumulate even/odd.....	155
Multiply - single precision vector by vector	157
Multiply (32 ×16)	158
Multiply bytes with 4-wide reduction - vector by scalar	161
Multiply by byte with accumulate and 4-wide reduction - vector by vector .	165
Multiply with 3-wide reduction.....	167
Sum of reduction of absolute differences halfwords.....	172
Sum of absolute differences byte	174
6.9 HVX MPY resource.....	177
Multiply by byte with 2-wide reduction 177	
Multiply by halfword with 2-wide reduction	179
Multiply - vector by scalar non-widening.....	181
Multiply - vector by vector - non-widening.....	183
Multiply half of the elements (16 ×16).....	184
Integer multiply by byte	185
Multiply half of the elements with scalar (16 ×16)	187
Multiply bytes with 4-wide reduction - vector by scalar	188
Multiply by byte with 4-wide reduction - vector by vector.....	190

Splat from scalar	192
Vector to predicate transfer	194
Predicate to vector transfer	195
Absolute value of difference	196
Insert element	198
6.10 HVX permute resource	199
Byte alignment 199	
General permute network	202
Shuffle - deal	207
Pack	210
Set predicate	213
Vector in-lane lookup table	214
6.11 HVX permute shift resource	220
Vector ASR overlay 220	
Vector shuffle and deal cross-lane	222
Vector in-lane lookup table	227
8-bit elements 227	
Unpack	235
6.12 HVX scatter double resource	237
Vector scatter 237	
6.13 HVX scatter	240
Vector scatter 240	
6.14 HVX shift resource	244
Narrowing shift 244	
Compute contiguous offsets for valid positions	248
Add - half precision vector by vector	250
Add - single precision vector by vector	252
Shift and add	254
Shift	257
Narrowing shift by vector	263
Convert IEEE floating point to IEEE integer	265
Convert IEEE integer to IEEE floating point	266
Convert Qfloat to IEEE floating point	267
Round to next smaller element size	268
Vector rotate right word	271
Subtract - half precision vector by vector	272
Subtract - single precision vector by vector	274
Bit counting	276
6.15 HVX store	278
Store - byte-enabled aligned 278	
Store - new	281
Store - aligned	284
Store - unaligned	291

Figures

Figure 1-1	Hexagon core with attached SIMD coprocessor	8
Figure 1-2	Registers using 128 byte with a vector length of 1024 bits	9
Figure 1-3	Four hardware threads (two HVX-enabled threads and two scalar-only threads) . .	10
Figure 1-4	1024-bit SIMD register	11
Figure 4-1	Qfloat format	26
Figure 6-1	vdelta example	202
Figure 6-2	vrdelta example	203
Figure 6-3	Vlut16 with even bytes used to look up a table value.	217
Figure 6-4	vshuff/vdeal (Vy,Vx, Rt) $N = 64/Rt$ $Rt = 2i$	222
Figure 6-5	vshuff(Vy, Vx, Rt) $Rt = -8 == 32 + 16 + 8$	223
Figure 6-6	vdeal(Vy,Vx, Rt) $Rt = -8$ or 56	224

Tables

Table 2-1	VRF to GRF transfer instructions	14
Table 3-1	Atomicity of types of memory accesses	18
Table 3-2	Peak scatter/gather performance for v75	20
Table 4-1	HVX execution resource usage	22
Table 4-2	HVX instruction to Hexagon slots mapping	22
Table 4-3	Sources for noncontiguous accesses: (Rt, Mu, Vv)	24
Table 4-4	Basic scatter and gather instructions	24
Table 4-5	Valid VMEM load/store and scatter/gather combinations.	25
Table 4-6	Differences between IEEE and Qfloat.	26
Table 4-7	HVX slot/resource/latency summary	28
Table 6-1	Instruction syntax symbols	31
Table 6-2	Instruction operand symbols.	31
Table 6-3	Instruction behavior symbols	32

1 Introduction

This document describes the Qualcomm® Hexagon™ Vector eXtensions (HVX) instruction set architecture. These extensions are implemented in an optional coprocessor. This document assumes that the reader is familiar with the Hexagon architecture. For a full description of the architecture, refer to the *Qualcomm Hexagon Programmer's Reference Manual*.

1.1 SIMD coprocessor

HVX instructions are primarily implemented in a single instruction multiple data (SIMD) coprocessor block that includes vector registers, vector compute elements, and dedicated memory. This extends the baseline Hexagon architecture to enable high-performance computer vision, image processing, or other workloads that can map to SIMD parallel processing.

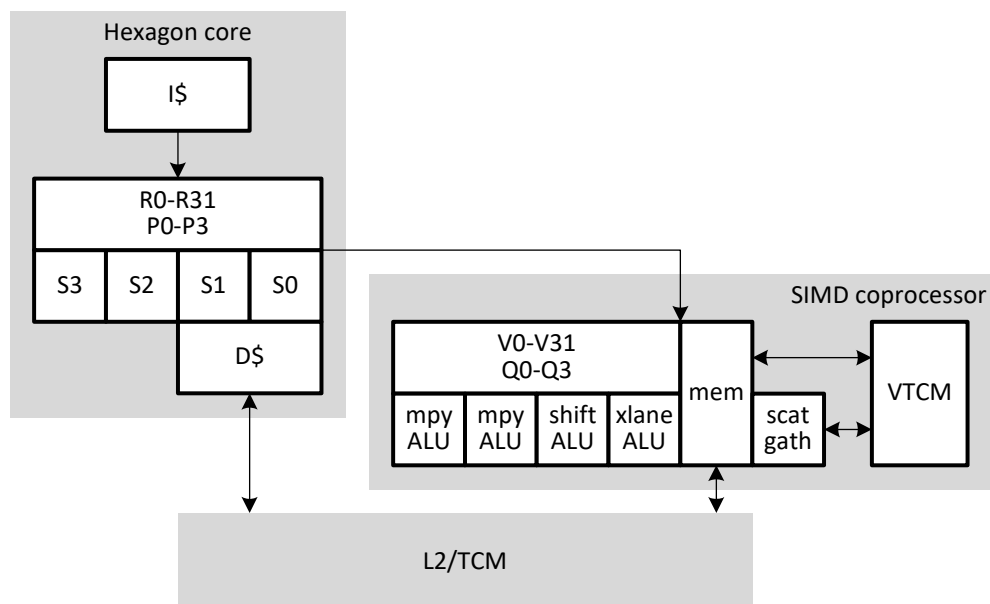


Figure 1-1 Hexagon core with attached SIMD coprocessor

The Hexagon instruction set architecture (ISA) is extended with HVX instructions. These instructions use HVX compute resources and can freely mix with normal Hexagon instructions in a very long instruction word (VLIW) packet. HVX instructions can also use scalar source operands from the core.

1.2 HVX features

HVX adds very wide SIMD capability to the Hexagon ISA. SIMD operations execute on vector registers (currently up to 1024 bits each), and multiple SIMD instructions can execute in parallel.

1.2.1 Vector length

HVX supports 1024-bit vectors (128 byte). To minimize porting effort, software should strive to treat vector length as an arbitrary constant power of two.

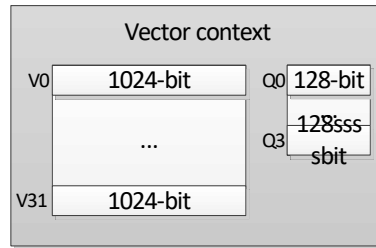


Figure 1-2 Registers using 128 byte with a vector length of 1024 bits

1.2.2 Vector contexts

A vector context consists of a vector register file, vector predicate file, and the ability to execute instructions using this state.

HVX hardware threads dynamically attach to a vector context; this enables the thread to execute HVX instructions. Multiple hardware threads can execute in parallel, each with a different vector context. The number of supported vector contexts is implementation-defined.

The HVX scalar core can contain any number of hardware threads greater or equal to the number of vector contexts. The scalar hardware thread is assignable to a vector context through per-thread SSR:XA programming, as follows:

- SSR:XA = 4: HVX instructions use vector context 0.
- SSR:XA = 5: HVX instructions use vector context 1, if it is available.
- SSR:XA = 6: HVX instructions use vector context 2, if it is available.
- SSR:XA = 7: HVX instructions use vector context 3, if it is available.

Figure 1-3 shows a vector context configuration with four hardware threads, but with two of the threads configured to use 128 byte vectors. In this configuration, two of the threads can execute 128 byte vector instructions, while the other two threads can execute scalar-only instructions.

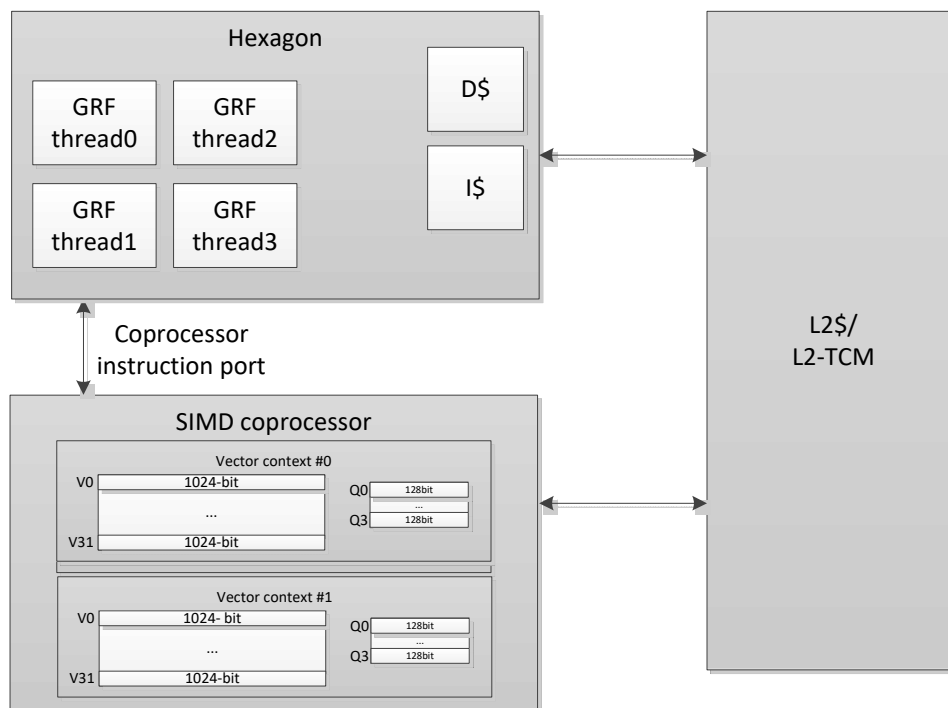


Figure 1-3 Four hardware threads (two HVX-enabled threads and two scalar-only threads)

1.2.3 Memory access

The HVX memory instructions (referred to as VMEM instructions) use the Hexagon general registers (R0 through R31) to form addresses that access memory. The memory access size of these instructions is the vector length or the size of a vector register.

VMEM loads and stores share a 32-bit virtual address space as normal scalar load/stores. VMEM load/stores are coherent with scalar load/stores and hardware maintains coherency.

1.2.4 Vector registers

HVX has two sets of registers:

- Data registers consist of 32 vector length registers. Certain operations can access a pair of registers to effectively double the vector length for the operand.
- Predicate registers consist of four registers, each with one bit per byte of vector length. These registers provide operands to compare, mux, and other special instructions.

The vector registers are partitioned into lanes that operate in SIMD fashion. For example, with 1024-bit (128 byte) vector length, each vector register can contain any of the following items:

- 32 words (32-bit elements)
- 64 halfwords (16-bit elements)
- 128 bytes (8-bit elements)

Element ordering is little-endian with the lowest byte in the least-significant position, as shown in [Figure 1-4](#).

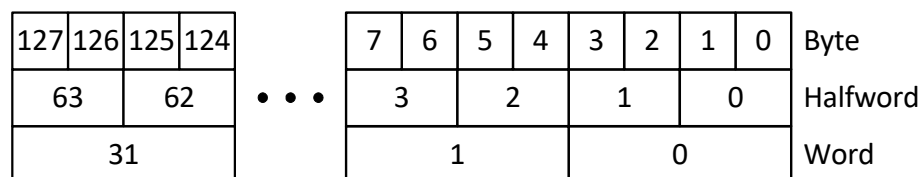


Figure 1-4 1024-bit SIMD register

1.2.5 Vector compute instructions

Vector instructions process vector register data in SIMD fashion. The operation is performed on each vector lane in parallel. For example, the following instruction performs a signed ADD operation over each halfword:

```
V2.h = VADD(V3.h, V4.h)
```

In this instruction, the halfwords in V3 are summed with the corresponding halfwords in V4, and the results stored in V2.

When vectors are specified in instructions, the element type is also usually specified:

- .b for signed byte
- .ub for unsigned byte
- .h for signed halfword
- .uh for unsigned halfword
- .w for signed word
- .uw for unsigned word
- .qf16 for qfloat16
- .qf32 for qfloat32
- .hf for half precision
- .sf for single precision

For example:

```
v0.b = vadd(v1.b, v2.b)           // Add vectors of bytes
v1:0.b = vadd(v3:2.b, v5:4.b)    // Add vector pairs of bytes
v1:0.h = vadd(v3:2.h, v5:4.h)    // Add vector pairs of halfwords
v5:4.w = vmpy(v0.h, v1.h)        // Widening vector 16 x 16 to 32
                                   // multiplies: halfword inputs,
                                   // word outputs
```

For operations with mixed element sizes, each operand with the smaller element size uses a single vector register and each operand with the larger element size (double the smaller) uses a vector register pair. One vector in a pair contains even elements and the other odd elements.

1.3 Technical assistance

For assistance or clarification on information in this document, open a technical support case at <https://support.qualcomm.com/>.

You will need to register for a Qualcomm ID account and your company must have support enabled to access our Case system.

Other systems and support resources are listed on <https://qualcomm.com/support>.

If you need further assistance, you can send an email to qualcomm.support@qti.qualcomm.com.

2 Registers

HVX is a load-store architecture where compute operands originate from registers and load/store instructions move data between memory and registers.

The vector registers are not for addressing or control information, but rather hold intermediate vector computation results. They are only accessible using HVX compute or load/store instructions.

The vector predicate registers contain the decision bits for each 8-bit quantity of the vector data registers.

2.1 Vector data registers

The HVX coprocessor contains 32 vector registers (named V0 through V31). These registers store operand data for the vector instructions.

For example:

```
V1 = vmem(R0)           // Load a vector of data
                          // from address R0

V4.w = vadd(V2.w, V3.w)  // Add each word in V2
                          // to corresponding word in V3
```

The vector data registers can be specified as register pairs representing a double-vector of data.

For example:

```
V5:4.w = vadd(V3:2.w, V1:0.w) // Add each word in V1:0 to
                               // corresponding word in V3:2
```

2.1.1 Unaligned vector pairs

Unaligned pairs are supported for vector pair register operands.

For example:

```
v6:7.b = vadd(v2:3.b, v4:5.b) // Add vector pairs of bytes
v0:1.h = vadd(v12:13.h, v5:4.h) // Add vector pairs of halfwords
```

2.1.2 VRF to GRF transfers

Table 2-1 lists the Hexagon instructions that transfer values between the vector register file (VRF) and the general register file (GRF).

A packet can contain up to two insert instructions or one extract instruction. The extract instruction incurs a long-latency stall and is primarily meant for debug purposes.

Table 2-1 VRF to GRF transfer instructions

Syntax	Behavior	Description
<code>Rd.w=extractw(Vu, Rs)</code>	<code>Rd = Vu.uw[Rs&0xF];</code>	Extract word from a vector into Rd with location specified by Rs. Primarily meant for debug.
<code>Vx.w=insertw(Rss)</code>	<code>Vx.uw[Rss.w[1]&0xF] = Rss.w[0];</code>	Insert word into vector at specified location. The low word in Rss specifies the data to insert, and the upper word specifies the location.

2.2 Vector predicate registers

Vector predicate registers hold the result of vector compare instructions, for example:

```
Q3 = vcmp.eq(V2.w, V5.w)
```

This example compares each 32-bit field of V2 and V5 and the corresponding 4-bit field is set in the corresponding predicate register Q3. For half-word operations, two bits are set per half-word. For byte operations, one bit is set per byte.

The vmux instruction frequently uses vector predicate instruction. This takes each bit in the predicate register and selects the first or second byte in each source, and places it in the corresponding destination output field.

```
V4 = vmux(Q2, V5, V6)
```

3 Memory

The Hexagon unified byte addressable memory has a single 32-bit virtual address space with little-endian format. All addresses, whether used by a scalar or vector operation go through the memory management unit (MMU) for address translation and protection.

3.1 Alignment

Unlike on the scalar processor, an unaligned pointer (a pointer that is not a multiple of the vector size) does not cause a memory fault or exception. When using a general VMEM load or store, the least-significant bits of the address are ignored.

```
VMEM(R0) = V1 // Store to R0 & ~(0x3F)
```

The intra-vector addressing bits are ignored.

Unaligned loads and stores are also explicitly supported through the VMEMU instruction.

```
V0 = VMEMU(R0) // Load a vector from R0 regardless of alignment
```

3.2 HVX local memory: VTCM

HVX supports a local memory called vector tightly coupled memory (VTCM) for scratch buffers and scatter/gather operations. The size of the memory is implementation-defined. The size is discoverable from the configuration table. VTCM needs normal virtual to physical translation just like other memory. This memory has higher performance and lower power.

Use VTCM for intermediate vector data, or as a temporary buffer. It serves as the input or output of the scatter/gather instructions. The following are advantages of using VTCM as the intermediate buffer:

- Guarantees no eviction (vs. L2 if the set is full)
- Faster than L2\$ (does not have the overhead of cache management, like association)
- Reduces L2\$ pressure
- Lower power than L2\$
- Supports continuous read and write for every packet without contention

In addition to HVX VMEM access, normal Hexagon memory access instructions can access this memory.

The following conditions are invalid for VTCM access:

- Using a page size larger than the VTCM size.
- Attempting to execute instructions from VTCM; including speculative access.
- Scalar VTCM access when the HVX fuse is blown (disabled).
- Load-locked or store-conditional to VTCM.
- A memw_phys instruction load from VTCM while more than one thread is active.
- Accessing VTCM while HVX is not fully powered up or VTCM banks are asleep.
- Unaligned access crossing between VTCM and non-VTCM pages.

3.3 Scatter and gather

Scatter and gather instructions allow for per-element random access of VTCM. Each element can specify an independent address to read (gather) or write (scatter). Gather for HVX is a vector copy from noncontiguous addresses to an aligned contiguous vector location. Gather operations use slot 0 + slot 1 on the scalar side, and HVX load + store resources.

Gather is formed by two instructions, one for reading from VTCM and one for storing to VTCM:

```
{ Vtmp.h = vgather(Rt,Mu,Vv.h)
  vmem(Rs+#1) = Vtmp.new
}
```

If the input data of gather is in DDR, it must first be copied to VTCM and gathered from there. Gather cannot be performed directly on DDR or L2\$ contents.

Vector gather (vgather) operations transfer elemental copies from a large region in VTCM to a smaller vector-sized region in VTCM. Each instruction can gather up to 64 elements. Gather supports halfword and word granularity. Emulate byte gather through vector predicate instructions using two packets.

Use gather for large lookup tables (up to VTCM size).

Except for scatters and following scatters, these instructions are ordered with the following operations. However, accesses from elements of the same scatter or gather instruction are not ordered. The primary ordered case is loading from a gather result or from a scatter region.

Operations via scatter or gather usually perform better via scatter.

The following conditions are invalid for scatter or gather access:

- The scatter (write) or gather (read) region covers more than one page, or the M source (length - 1) is negative. An exception is generated otherwise.
- Any of the accesses are not within VTCM. This includes the gather target addresses as well. An exception is generated otherwise.
- Both a gather region instruction and a scatter instruction in the same packet.

3.4 Memory type

HVX memory instructions (VMEM or scatter/gather) that target device-type memory raise a VMEM address error exception. It is also illegal to use HVX memory instructions while the MMU is off.

NOTE: HVX is designed to work with L2 cache, L2TCM, or VTCM. Mark memory as L2-cacheable for L2 cache data and uncached for data that resides in L2TCM or VTCM.

3.5 Nontemporal

A VMEM instruction can have an optional nontemporal attribute, specified in assembly with a `:nt` appendix. Marking an instruction nontemporal indicates to the microarchitecture that the data is no longer needed after the instruction. The cache memory system uses this information to inform replacement and allocation decisions.

3.6 Permissions

Unaligned VMEM instructions that are naturally aligned only require MMU permissions for the accessed line. The hardware suppresses generating an access to the unused portion.

The byte-enabled conditional VMEM store instruction requires MMU permissions regardless of whether bytes are performed or not. In other words, the state of the Q register is not considered when checking permissions.

3.7 Ordering

The HVX coprocessor follows the same sequentially consistent memory model as the scalar core for coprocessor packets. Coprocessor threads interleave their coprocessor memory operations with one another in an arbitrary but fair manner. This results in a consistent program order that is globally observable by threads in the same order.

The only exception to this rule is the scatter operations. Scatter operation memory updates are unordered with respect to each other. Their internal transactions are also unordered.

Direct memory access (DMA) through the external AXI slave port are also considered noncoherent with the coprocessor threads and require explicit memory synchronizations through the use of the store release or polling of the DMA descriptor performed by the scalar core.

3.8 Atomicity

Table 3-1 describes the size or alignment of decomposed atomic operations for different types of memory accesses. When an access is not fully atomic, an observer can see atomic components of the access.

Table 3-1 Atomicity of types of memory accesses

Access type	Atomic size
Scalar A mem-op is two accesses	Access size
Aligned vector	Base vector size
Unaligned vector	1 B
Scatter	1 B
Scatter-accumulate (read-modify-write)	1 B A larger read-modify-write can decompose into multiple equivalent smaller read-modify-writes.
Gather read	1 B
Gather write	1 B

Individual scatter and gather accesses are only guaranteed atomic with other scatter or gather accesses.

3.9 Maximizing performance of the vector memory system

The HVX vector processor is attached directly to the L2 cache. VMEM loads/stores move data to/from L2 and do not use L1 data cache. To ensure coherency with L1, VMEM stores check L1 and invalidate on hit.

3.9.1 Minimize VMEM access

Accessing data from the VRF is far cheaper in cycles and power than accessing data from memory. The simplest way to improve memory system performance is to reduce the number of VMEM instructions. Avoid moving data to/from memory when VRF can host it instead.

3.9.2 Use aligned data

VMEMU instruction access multiple L2 cache lines and are expensive in bandwidth and power. Where possible, align data structures to vector boundaries. Padding the image is often the most effective technique to provide aligned data.

3.9.3 Avoid store to load stalls

A VMEM load instruction that follows a VMEM store to the same address incurs a store-to-load penalty. The store must fully reach L2 before the load starts, thus the penalty can be quite large. To avoid store-to-load stalls, there should be approximately 15 packets of intervening work.

3.9.4 L2FETCH

Use the L2FETCH instruction to prepopulate the L2 cache with data prior to using VMEM loads.

L2FETCH is best performed in sizes less than 8 KB and issued at least several hundred cycles prior to using the data. If the L2FETCH instruction is issued too early, data can be evicted before use. In general, prefetching and processing on image rows or tiles works best.

Prefetch L2 cacheable data that VMEM uses, even if it is not used in the computation. Software pipelined loops often overload data unused data. Even though the pad data is not used in computation, the VMEM stalls if it has not been prefetched into L2.

3.9.5 Access data contiguously

Whenever possible, arrange data in memory so that it is accessed contiguously. For example, instead of repeatedly striding through memory, data might be first tiled, striped, or decimated to enable contiguous access

The following techniques achieve better spatial locality in memory to help avoid performance hazards:

- Bank conflicts - Lower address bits are typically used for parallel banks of memory. Accessing data contiguously achieves a good distribution of these address bits. If address bits [7:1] are unique across elements within a vector, the operation is conflict-free. Use a vector predicate to mask out “don’t care” values.
- Set aliasing. Caches hold some sets identified by lower address bits. Each set has a small number of methods (typically 4 to 8) to help manage aliasing and multi-threading.
- Micro-TLB misses. A limited number of pages are remembered for fast translation. Containing data to a smaller number of pages helps translation performance.

3.9.6 Use nontemporal for final data

On the last use of data, use the :nt attribute. The cache uses this hint to optimize the replacement algorithm.

3.9.7 Scalar processing of vector data

When a VMEM store instruction produces data, that data is placed into the L2 cache and L1 does not contain a valid copy. Thus, if scalar loads must access the data, it first must be fetched into L1.

Algorithms use the vector engine to produce results that must further process on the scalar core. The best practice is to use VMEM stores to get the data into L2, then use DCFETCH to get the data in L1, followed by scalar load instructions. Execute the DCFETCH anytime after the VMEM store, however, software should budget at least 30 cycles before issuing the scalar load instruction.

3.9.8 Avoid scatter/gather stalls

Scatter and gather operations compete for memory and can result in long latency, therefore take care to avoid stalls. The following techniques improve performance around scatter and gather:

- Distribute accesses across the intra-vector address range (lower address bits). Even distribution across the least significant inter-vector address bits can also be beneficial. Address bits [10:3] are important to avoid conflicts. Ideally this applies per vector instruction, but distributing these accesses out between vector instructions can help absorb conflicts within a vector instruction.
- Minimize the density of scatter and gather instructions. Spread out these instructions in a larger loop rather than concentrating them in a tight loop. The hardware can process a small number of these instructions in parallel. If it is difficult to spread these instructions out, limit bursts to four for a specified thread.
- Defer loading from a gather result or a scatter store release. If the in-flight scatters and gathers (including from other threads) avoid conflicts, generally a distance of 12 or more packets is sufficient. Double that distance if the addresses of in-flight accesses are not correlated.

Table 3-2 Peak scatter/gather performance for v75

Operation	Addressing	Vector bandwidth (per packet)	Latency (packets)
Scatter	Conflict-free	1/2	18
Gather	Conflict-free	1/2	24
Scatter	Random	1/6	30
Gather	Random	1/6	48

4 Vector instructions

This chapter provides an overview of the HVX load/store instructions, compute instructions, VLIW packet rules, dependency, and scheduling rules.

Section [4.8](#) summarizes Hexagon slot, HVX resource, and instruction latency for the instruction categories.

4.1 VLIW packing rules

HVX provides the following resources for vector instruction execution:

- load
- store
- shift
- permute
- two multiply

Each HVX instruction consumes some combination of these resources, as defined in Section [4.1.2](#). VLIW packets cannot oversubscribe resources.

An instruction packet can contain up to four instructions, plus an end loop. The instructions inside the packet must obey the packet grouping rules described in Section [4.1.3](#).

NOTE: The assembler should check and flag invalid packet combinations. When an invalid packet executes, the behavior is undefined.

4.1.1 Double vector instructions

Certain instructions consume a pair of resources, either both the shift and permute as a pair or both multiply resources as another pair. Such instructions are referred to as double vector instructions because they use two vector compute resources.

Halfword by halfword multiplies are double vector instructions, because they consume both the multiply resources.

4.1.2 Vector instruction resource usage

Table 4-1 summarizes the resources that an HVX instruction uses during execution. It also specifies the order in which the Hexagon assembler tries to build an instruction packet from the most to least stringent.

Table 4-1 HVX execution resource usage

Instruction	Used resources
Histogram	All
Unaligned memory access	Load, store, and permute
Double vector cross-lane permute	Permute and shift
Cross-lane permute	Permute
Shift	Shift
Double vector & halfword multiplies	Both multiply
Single vector	Either multiply
Double vector ALU operation	Either shift and permute or both multiply
Single vector ALU operation	Shift, permute, or multiply
Aligned memory	Shift, permute, or multiply and one of load or store
Aligned memory (.tmp/.new)	Load or store only
Scatter (single vector indexing)	Store and one of shift, permute, or multiply
Scatter (double vector indexing)	Store and either shift and permute or both multiply
Gather (single vector indexing)	Load and one of shift, permute, or multiply
Gather (double vector indexing)	Load and either shift and permute or both multiply

4.1.3 Vector instruction

Vector instructions map to certain Hexagon slots. A special subset of ALU instructions (including nclude lookup table, splat, insert, and addition/subtraction with Rt) that require either the full 32 bits of the scalar Rt register (or 64 bits of Rtt) map to slots 2 and 3.

Table 4-2 HVX instruction to Hexagon slots mapping

Instruction	Used Hexagon slots	Additional restrictions
Aligned memory load	0 or 1	
Aligned memory store	0	
Unaligned memory load/store	0	Slot 1 must be empty. Maximum of 3 instructions allowed in the packet.
Scatter	0	
Gather	1	.new store in slot 0
Vextract	-	Only instruction in packet
Histogram	0, 1, 2, or 3	.tmp load in same packet
Multiplies	2 or 3	
Using full 32-64 bit R	2 or 3	
Simple ALU, permute, shift	0, 1, 2, or 3	

4.2 Vector load/store

VMEM instructions move data between VRF and memory. VMEM instructions support the following addressing modes.

- Indirect
- Indirect with offset
- Indirect with auto-increment (immediate and register/modifier register)

For example:

```
V2 = vmem(R1+#4) // Address R1 + 4 * (vector-size) bytes
V2 = vmem(R1++M1) // Address R1, post-modify by the value of M1
```

The immediate increment and post increments values are vector counts. So the byte offset is in multiples of the vector length.

To facilitate unaligned memory access, unaligned load and stores are available. The VMEMU instructions generate multiple accesses to the L2 cache and use the permute network to align the data.

The load-temp and load-current forms allow immediate use of load data within the same packet. A load-temp instruction does not write the load data into the register file. A register must be specified, but it is not overwritten. Because the load-temp instruction does not write to the register file, it does not consume a vector ALU resource.

A load-temp destination register cannot be an accumulator register within the packet. The behavior is considered undefined.

```
{ V2.tmp = vmem(R1+#1)           // Data loaded to a tmp
  V5:4.ub = vadd(V3.ub, V2.ub) // Use loaded data as V2 source
  V7:6.uw = vrmpy(V5:4.ub, R5.ub, #0) }
```

Load-current is similar to load-temp, but consumes a vector ALU resource as the loaded data writes to the register file.

```
{ V2.cur = vmem(R1+#1)           // Data loaded into a V2
  V3 = valign(V1,V2, R4)        // Load data used immediately
  V7:6.ub = vrmpy(V5:4.ub, R5.ub, #0)
}
```

VMEM store instructions can store a newly generated value. They do not consume a vector ALU resource, as they do not read nor write the register file.

The register used for the new value VMEM store is encoded in a field in the store instruction. The store is always slot 0, and scalar instructions are skipped when counting the offset. Bit 0 corresponds to either the even (0) or odd (1) register of the pair.

```
vmem(R1+#1) = V20.new // Store V20 that was generated in the current
packet
```

An entire VMEM write can also be suppressed by a scalar predicate.

```
if P0 vmem(R1++M1) = V20 // Store V20 if P0 is true
```

A vector predicate register can issue and control a partial byte-enabled store.

```
if Q0 vmem(R1++M1) = V20 // Store bytes of V20 where Q0 is true
```

4.3 Scatter and gather

Unlike vector loads and stores that access contiguous vectors in memory, scatter and gather allow for noncontiguous memory access of vector data. With scatter and gather, each element can independently index into a region of memory. This allows for applications that otherwise do not map well to the SIMD parallelism that HVX provides.

A scatter transfers data from a contiguous vector to noncontiguous memory locations. Similarly, gather transfers data from noncontiguous memory locations to a contiguous vector. In HVX, scatter is a vector register to noncontiguous memory transfer and gather is a noncontiguous memory to contiguous memory transfer. Additionally, HVX supports scatter-accumulate instructions that atomically add.

To maximize performance and efficiency, the scatter and gather instructions define a bounded region that must contain noncontiguous accesses. This region must be within VTCM (scatter/gather capable) and be within one translatable page. A vector specifies offsets from the base of the region for each element access. [Table 4-3](#) lists the three sources that specify the noncontiguous accesses of a scatter or gather:

Table 4-3 Sources for noncontiguous accesses: (Rt, Mu, Vv)

Source	Meaning
Rt	Base address of the region
Mu	Byte offset of last valid byte of the region (for example, region size - 1)
Vv or Vvv	Vector of byte offsets for the accesses. Double-vector is used when the offset width is double the data width

To form an HVX gather (memory to memory), vgather is paired with a vector store to specify the destination address. A scatter is specified with a single instruction. Ignoring element sizes, [Table 4-4](#) describes the basic forms of scatter and gather instructions:

Table 4-4 Basic scatter and gather instructions

Instruction	Behavior
<code>vscatter (Rt, Mu, Vv) = Vw</code>	Write data in Vw to noncontiguous addresses specified by (Rt, Mu, Vv)
<code>vscatter (Rt, Mu, Vv) += Vw</code>	Atomically add data in Vw to noncontiguous addresses specified by (Rt, Mu, Vv)
<code>{ vtmp = vgather (Rt, Mu, Vv) ; vmem (Addr) = vtmp.new }</code>	Read data from noncontiguous addresses specified by (Rt, Mu, Vv) and write the data contiguously to the aligned address

4.4 Memory instruction slot combinations

VMEM load/store instructions and scatter/gather instructions can be grouped with normal scalar load/store instructions.

Table 4-5 provides the valid grouping combinations for HVX memory instructions. A combination that is not present in the table is invalid, and should be rejected by the assembler. The hardware generates an invalid packet error exception.

Table 4-5 Valid VMEM load/store and scatter/gather combinations

Slot 0 instruction	Slot 1 instruction
VMEM Ld	Nonmemory
VMEM St	Nonmemory
VMEM Ld	Scalar Ld
Scalar St	VMEM Ld
Scalar Ld	VMEM Ld
VMEM St	Scalar St
VMEM St	Scalar Ld
VMEM St	VMEM Ld
VMEMU Ld	Empty
VMEMU St	Empty
.new VMEM St	Gather
Scatter	Nonmemory
Scatter	Scalar St
Scatter	Scalar Ld
Scatter	VMEM Ld

4.5 Special instructions

4.5.1 Histogram

HVX contains a specialized histogram instruction. The vector register file divides into four histogram tables each of 256 entries (32 registers by 8 halfwords). A temporary VMEM load instruction fetches a line from memory. The top five bits of each byte provide a register select, and the bottom bits provide an element index. The value of the element in the register file is incremented. The programmer must clear the registers before use.

Example:

```
{  V31.tmp VMEM(R2) // Load a vector of data from memory
    VHIST();         // Perform histogram using counters in VRF and indexes
                      // from temp load
}
```

4.6 Qfloat

The Qfloat floating point format offers similar dynamic range and precision to that of IEEE 754. There are significant differences, as Qfloat is more hardware efficient.

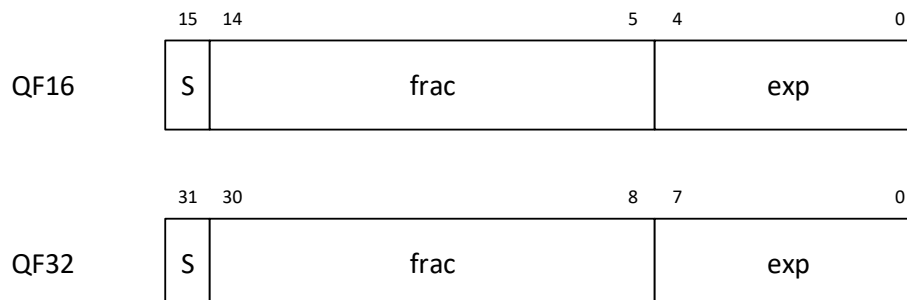


Figure 4-1 Qfloat format

The QFloat format has the following properties:

- The fractional field is two's complement fixed-point format.
- There is no implied MSB in the significand as there is in IEEE. The fractional field only encodes o.frac.
- Qfloat implements Von Neumann rounding, where the implied LSB of the fractional field is an implicit one.
- There is no concept of infinity or NaN. QFloat saturates to maximum exponent with maximum positive or minimum negative significand.
- The Qfloat format has one bit less of precision compared to IEEE for most algorithms.

Table 4-6 Differences between IEEE and Qfloat

Features	IEEE - 754	QFloat
Bits	1 + exp + mantissa	1 + mantissa + exp
Positive zero	Yes (sub-normal)	Rounded to +tiniest
Negative zero	Yes (sub-normal)	Rounded to -tiniest
Significant bits	Mantissa +1	Mantissa + 1 (limited by round)
Subnormal/zero	W/exp = min	Un-normals natural
Infinity/NaN	W/exp = max	Saturated
Lg(max/min)	$2^E - 2 + M$	$2^E + M$
Unique finite values	$(2^E - 1) * 2^M - 1$	$2^E * 2^M$
Rounding	Nearest even	Neatest odd or odd-even
Use with QFloat	Input and conversion/storage	Compute

QFloat instructions use the same shift and multiply resources as other HVX instructions.

4.6.1 QFloat best practices

Treat QFloat like an intermediate format where the input and output of an algorithm are in an IEEE format (single or half precision).

The QFloat instruction set supports IEEE float values as inputs on the vector operands. The intermediate computations of an algorithm are performed in native Qfloat. The final output is converted back to IEEE through explicit convert instructions before storing to memory.

Performing a normalization step prior to a multiply is beneficial when expecting massive cancellation in a prior addition or subtraction step.

4.7 Instruction latency

Latencies are implementation-defined and can change with future versions.

HVX packets execute over multiple clock cycles, but typically in a pipelined manner to issue and complete a packet on every context cycle. The contexts are time interleaved to share the hardware such that using all contexts might be required to reach peak compute bandwidth.

With a few exceptions (for example, histogram and extract), results of packets generate within a fixed time after execution starts. But, when the sources are required varies. Instructions that need more pipelining require early sources. Only HVX registers are early source registers. Early source operands include:

- Input to the multiplier. For example, `V3.h = vmpyh(V2.h, V4.h)`. V2 and V4 are multiplier inputs. For multiply instructions with accumulation, the accumulator is not considered an early source multiplier input.
- Input to shift/bit count instructions. Only the shifted or counted register is considered early source. Accumulators are not early sources.
- Input to permute instructions. Only permuted registers are considered early source (not an accumulator).
- Unaligned store data is an early source.

An early source register produced in the previous vector packet can incur an interlock stall. Software should strive to schedule an intervening packet between the producer and an early source consumer.

The following example shows interlock cases:

```
V8 = VADD(V0, V0)
V0 = VADD(V8, V9) // NO STALL
V1 = VMPY(V0, R0) // STALL due to V0
V2 = VSUB(V2, V1) // NO STALL on V1
V5:4 = VUNPACK(V2) // STALL due to V2
V2 = VADD(V0, V4) // NO STALL on V4
```

4.8 Slot/resource/latency summary

Table 4-7 summarizes the Hexagon slot, HVX resource, and latency requirements for HVX instruction types.

Table 4-7 HVX slot/resource/latency summary

Category	Variation	Core slots				Vector resources						Input latency
		3	2	1	0	ld	mpy	mpy	shift	xlane	st	
ALU	1 vector	any					any					1
	2 vectors	any					either pair					1
	Rt		either				either					1
Abs-diff	1 vector		either				either					2
	2 vector		either									2
Multiply	by 8 bits; 1 vec		either				either					2
	by 8 bits; 2 vec		either									2
	by 16 bits		either									2
Cross-lane	1 vector	any										2
	2 vectors	any										2
Shift or count	1 vector	any										2
load	aligned			either			any					-
	aligned; .tmp			either								-
	aligned; .cur			either			any					-
	unaligned											-
store	aligned						any					1
	aligned; .new											0
	unaligned											2
gather (needs .new store)	1 vector						any					1
	2 vector						either pair					1
scatter	1 vector						any					1
	2 vector						either pair					1
histogram (needs .tmp load)		any										2
extract												1

5 HVX PMU events

The Hexagon processor architecture defines a performance monitor unit (PMU) to provide on-target performance tracking.

The PMU allows for easy collection of aggregate performance data like cache performance and instructions per packet. This data is valuable for system planning and architecture purposes because it drives performance and power statistical models.

In V69 and later versions, the PMU event space is expanded to 1024 events. Events 0 to 255 describe core events. Coprocessors use events 384 and above; which are described in the *Qualcomm Hexagon V75 Programmer's Reference Manual* (80-N2040-53)

HVX events 256 to 299 are documented in [Table 5-1](#)

Table 5-1 HVX PMU events

Event	Name	Description
256	HVX_ACTIVE	VFIFO not empty.
257	HVX_REG_ORDER	Stall cycles due to interlocks.
258	HVX_ACC_ORDER	Stall cycles due to accumulator not produced in previous context cycle.
259	HVX_LD_L2_OUTSTANDING	Stall cycles due to load pending.
260	HVX_ST_L2_OUTSTANDING	Stall cycles due to store not yet allocated in L2.
261	HVX_VTCM_OUTSTANDING	Stall cycles due to VTCM transaction pending.
262	HVX_SCATGATH_FULL	Scatter/gather: network scoreboard not updated.
263	HVX_SCATGATH_IN_FULL	Scatter/gather input buffer full.
266	HVX_VOLTAGE_UNDER	Throttling: voltage model exceeds undershoot threshold.
267	HVX_POWER_OVER	Throttling: sustained power exceeds budget.
268	HVX_PKT_PARTIAL	Stall cycles due to multi-issue packet.
273	HVX_PKT	Packets with HVX instructions.
274	HVX_PKT_THREAD	Committed packets on a thread with the XE bit set, whether executed in Q6 or coprocessor.
275	HVX_CORE_VFIFO_FULL_STALL	Number of cycles a thread stalls due to VFIFO.
277	CYCLES_1_HVX_CONTEXTS_RUNNING	Cycles 1 HVX context running.
278	CYCLES_2_HVX_CONTEXTS_RUNNING	Cycles 2 HVX contexts running concurrently.
279	CYCLES_3_HVX_CONTEXTS_RUNNING	Cycles 3 HVX contexts running concurrently.

Table 5-1 HVX PMU events

Event	Name	Description
280	HVXLD_L2	L2 cacheable load access from HVX. Any load access from HVX that might cause a lookup in the L2 cache. Excludes cache ops, uncacheables, and scalars.
281	HVXLD_L2_TCM	TCM load access for HVX. HVX load from the L2 TCM space.
290	HVXST_VTCM_FULL	Write FIFO full.
291	HVXST_L2	Vector store to L2.
292	HVXST_L2_MISS	L2 cacheable miss from HVX store; the cases where the 128-byte-line address is not in the tag or a coalesce buffer.
295	HVXST_L2_SECODARY_MISS	L2 cacheable secondary miss from HVX store; the cases where the 128-byte-line address is not in the tag or a coalesce buffer.
296	HVXPIPE_ALU	Executed simple ALU instruction.
297	HVXPIPE_MPY	Executed multiply or abs-diff instruction.
300	CYCLES_4_HVX_CONTEXTS_RUNNING	Cycles four HVX contexts that run concurrently.

6 HVX instruction set

instructions are listed alphabetically within instruction categories. The following information is provided for each instruction:

- Instruction name
- A brief description of the instruction
- A high-level functional description (syntax and behavior) with all possible operand types
- Instruction class and slot information for grouping instructions in packets
- Notes on miscellaneous issues
- C intrinsic functions that provide access to the instruction
- Instruction encoding

[Table 6-1](#) lists the symbols used to specify the instruction syntax.

Table 6-1 Instruction syntax symbols

Symbol	Example	Meaning
=	R2 = R3;	Assignment of RHS to LHS
;	R2 = R3;	Marks the end of an instruction or group of instructions
{ ... }	{R2 = R3; R5 = R6;}	Indicates a group of parallel instructions.
#	#100	Immediate constant value
0x	R2 = #0x1fe;	Indicates hexadecimal number
:sat	R2 = add(r1,r2):sat	Perform optional saturation
:rnd	R2 = mpy(r1.h,r2.h):rnd	Perform optional rounding

[Table 6-2](#) lists the symbols used to specify instruction operands.

Table 6-2 Instruction operand symbols

Symbol	Example	Meaning
#uN	R2 = #u16	Unsigned N-bit immediate value
#sN	R2 = add(R3,#s16)	Signed N-bit immediate value
#mN	Rd = mpyi(Rs,#m9)	Signed N-bit immediate value
#uN:S	R2 = memh(#u16:1)	Unsigned N-bit immediate value representing integral multiples of 2S in the specified range
#sN:S	Rd = memw(Rs++#s4:2)	Signed N-bit immediate value representing integral multiples of 2S in the specified range
#rN:S	call #r22:2	Same as #sN:S, but value is offset from PC of current packet

When an instruction contains more than one immediate operand, the operand symbols are specified in upper and lower case (for example, #uN and #UN) to indicate where they appear in the instruction encodings.

The instruction behavior is specified using a superset of the C language. [Table 6-3](#) lists symbols not defined in C that are used to specify the instruction behavior.

Table 6-3 Instruction behavior symbols

Symbol	Example	Meaning
usat _N	usat ₁₆ (Rs)	Saturate a value to an unsigned N-bit
sat _N	sat ₁₆ (Rs)	Saturate a value to a signed N-bit number
sxt x->y	sxt32->64(Rs)	Sign-extend value from x to y bits
zxt x->y	zxt32->64(Rs)	Zero-extend value from x to y bits
>>>	Rss >>> offset	Logical right shift

6.1 HVX ALL compute resource

The HVX ALL compute resource instruction subclass includes ALU instructions that use a pair of HVX resources.

Histogram

The `vhist` instructions use all of the HVX core resources: the register file, V0 through V31, and all four instruction pipes. The instruction takes four execution packets to complete. The basic unit of the histogram instruction is a 128-bit wide slice - there can be four or eight slices, depending on the particular configuration. The 32 vector registers are configured as multiple 256-entry histograms, where each histogram bin has a width of 16 bits. This allows up to 65,535 8-bit elements of the same value to accumulate. Each histogram is 128 bits wide and 32 elements deep, for a total of 256 histogram bins. A vector is read from memory and stored in a temporary location, outside of the register file. The data read is divided equally between the histograms.

For example:

Bytes 0 to 15 are profiled into bits 0 to 127 of all 32 vector registers, histogram 0.

Bytes 16 to 31 are profiled into bits 128 to 255 of all 32 vector registers, histogram 1.

... and so on.

The bytes process over multiple cycles to update the histogram bins. For each of the histogram slices, the lower three bits of each byte element in the 128-bit slice is used to select the 16-bit position, while the upper five bits select the vector register. The register file entry is then incremented by one.

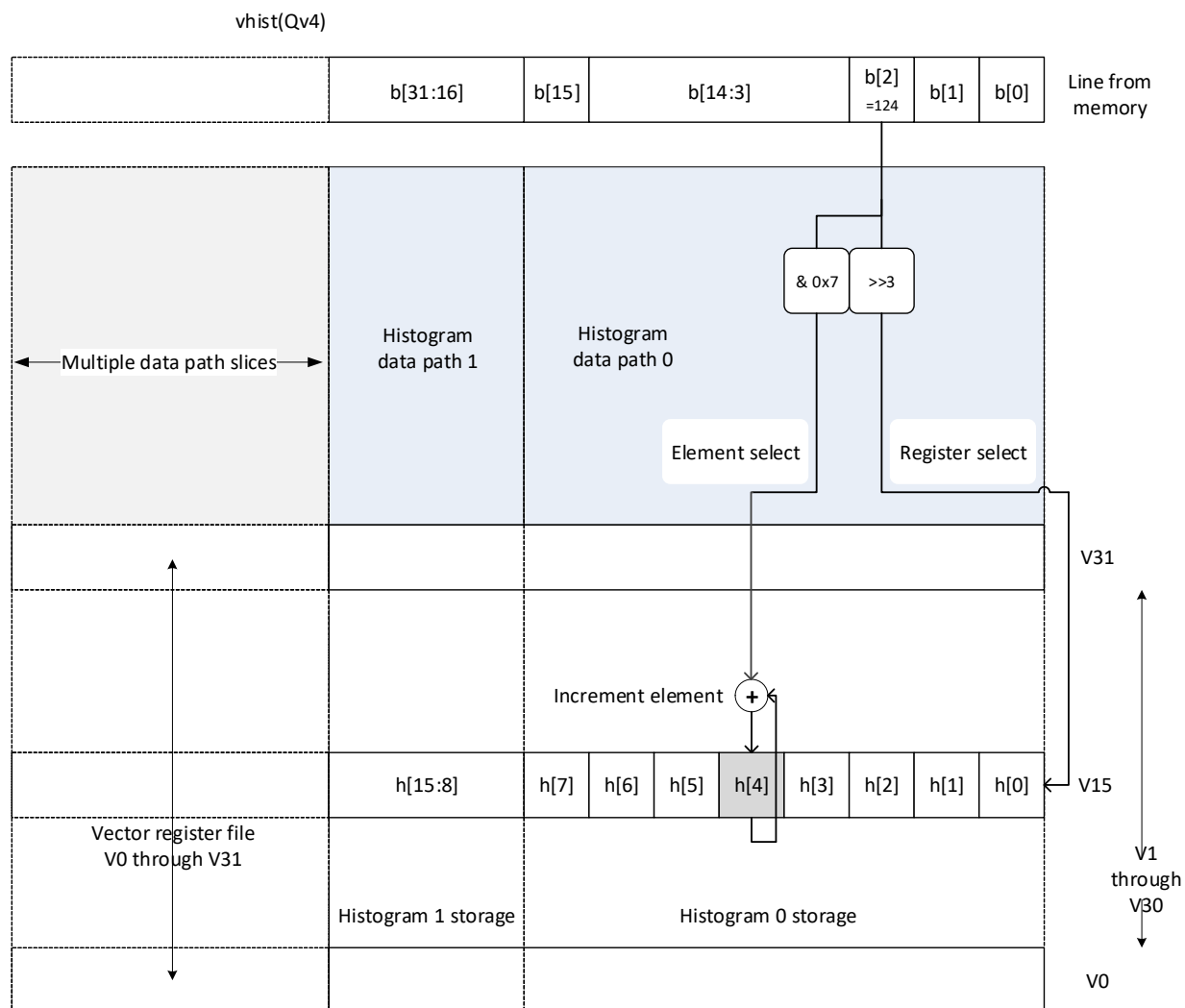
The `vhist` instruction is the only instruction that occupies all pipes and resources.

Before use, the vector register file must be cleared to begin a new histogram, otherwise the current state is added to the histograms of the next data.

The `vhist` instruction supports the same addressing modes as standard loads. In addition, a byte-enabled version is available that enables the selection of the elements used in the accumulation.

The following diagram shows a single 8-bit element in position two of the source data. The value is 124, the register number assigned to this is $124 \gg 3 = V15$, and the element number in the register is $124 \& 7 = 4$. The byte position in the example is two, which is in the first 16 bytes of the input line from memory, so the data affects the first 128-bit wide slice of the register file. The 16-bit histogram bin location is then incremented by 1. Each 64-bit input group of bytes affects the respective 128-bit histogram slice.

For a 64-byte vector size, there can be a peak total consumption of $64(\text{bytes per vector})/4(\text{packets per operation}) * 4(\text{threads}) = 64$ bytes per clock cycle per core, assuming all threads perform histogramming.

**Syntax**

vhist

Behavior

```

inputVec=Data from .tmp load;
for (lane = 0; lane < VELEM(128); lane++) {
    for (i=0; i<128/8; ++i) {
        unsigned char value = inputVec.ub[(128/8)* lane+i];
        unsigned char regno = value>>3;
        unsigned char element = value & 7;
        READ_EXT_VREG(regno,tmp,0);
        tmp.uh[(128/16)*lane+(element)]++;
        WRITE_EXT_VREG(regno,tmp,EXT_NEW);
    }
};

```

Syntax

vhist(Qv4)

Behavior

```

inputVec=Data from .tmp load;
for (lane = 0; lane < VELEM(128); lane++) {
    for (i=0; i<128/8; ++i) {
        unsigned char value = inputVec.ub[(128/8) * lane + i];
        unsigned char regno = value>>3;
        unsigned char element = value & 7;
        READ_EXT_VREG(regno,tmp,0);
        if (QvV[128/8*lane+i])
            tmp.uh[(128/16)*lane+(element)]++;
        WRITE_EXT_VREG(regno,tmp,EXT_NEW);
    }
};

```

Class: COPROC_VX (slots 0,1,2,3)**Encoding**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
ICLASS																Parse																		
0	0	0	1	1	1	1	0	-	-	0	-	-	0	0	0	P	P	1	-	0	0	0	-	1	0	0	-	-	-	-	-	vhist		
0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	0	P	P	1	-	-	0	0	-	1	0	0	-	-	-	-	-	vhist(Qv4)		

Field name**Description**

ICLASS

Instruction class

Parse

Packet loop parse bits

v2

Field to encode register v

Weighted histogram

The `vwhist` instructions use all of the HVX core resources: the register file, V0 through V31, and all four instruction pipes. The instruction takes four execution packets to complete. The basic unit of the histogram instruction is a 128-bit wide slice - there can be four or eight slices, depending on the particular configuration. The 32 vector registers are configured as multiple 256-entry histograms for `vwhist256`, where each histogram bin has a width of 16 bits. Each histogram is 128 bits wide and 32 elements deep, for a total of 256 histogram bins. For `vwhist256`, the 32 vector registers are configured as multiple 128-entry histograms where each histogram bin has a width of 32 bits. Each histogram is 128 bits wide and 16 elements deep, for a total of 128 histogram bins.

A vector is read from memory and stored in a temporary location, outside of the register file. The vector carries both the data that is used for the index into the histogram and the weight. The data occupies the even byte of each halfword and the weight the odd byte of each halfword. The data read is then divided equally between the histograms.

For example:

Even bytes 0 to 15 are profiled into bits 0 to 127 of all 32 vector registers, histogram 0.

Even bytes 16 to 31 are profiled into bits 128 to 255 of all 32 vector registers, histogram 1.

... and so on.

The bytes are processed over multiple cycles to update the histogram bins. For each of the histogram slices in `vwhist256`, the lower three bits of each even byte element in the 128-bit slice is used to select the 16-bit position, while the upper 5 bits select the vector register.

For each of the histogram slices in `vwhist128`, bits 2:1 of each even byte element in the 128-bit slice are used to select the 32-bit position, while the upper five bits select the vector register. The LSB of the byte is ignored.

The register file entry is then incremented by corresponding weight from the odd byte.

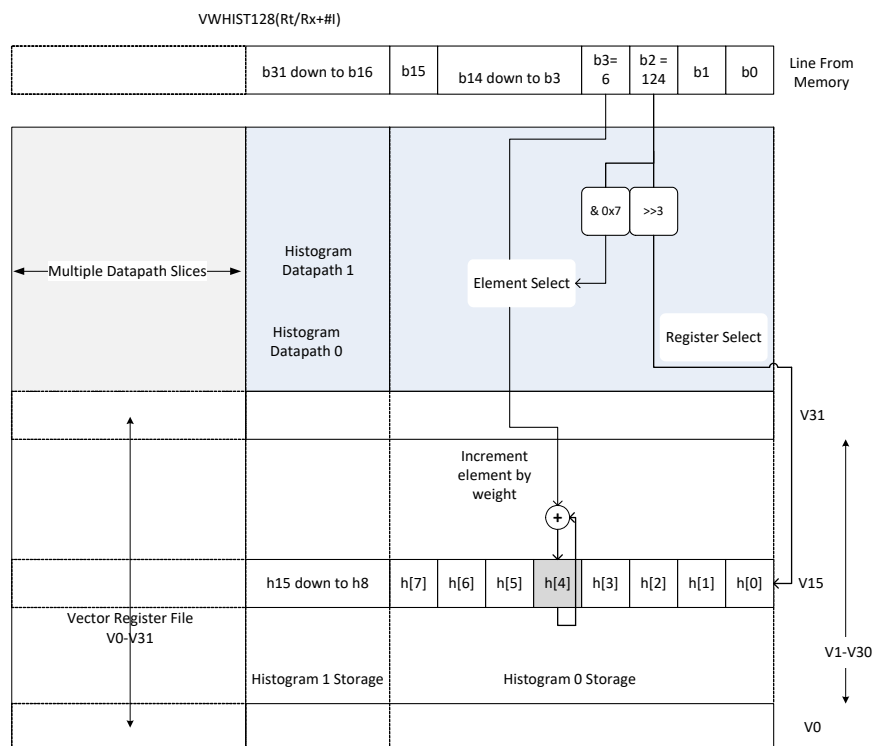
Like `vhist`, `vwhist` also occupies all pipes and resources.

Before use, the vector register file must be cleared to begin a new histogram, otherwise the current state is added to the histograms of the next data.

The `vwhist` instruction supports the same addressing modes as standard loads. In addition, a byte-enabled version is available that enables the selection of the elements used in the accumulation.

The following diagram shows a single 8-bit element in byte position 2 of the source data, with corresponding weight in byte position 3. The value is 124, the register number assigned to this is $124 \gg 3 = V15$, and the element number in the register is $124 \& 7 = 4$. The byte position in the example is 2, which is in the first 16 bytes of the input line from memory, so the data affects the first 128-bit wide slice of the register file. The 16-bit histogram bin location is then incremented by the weight from byte position 3. Each 64-bit input group of bytes affects the respective 128-bit histogram slice.

For a 64-byte vector size, there can be a peak total consumption of $64(\text{bytes per vector})/4(\text{packets per operation}) * 4(\text{threads}) = 64 \text{ bytes per clock cycle per core}$, assuming all threads perform histogramming.



Syntax

Behavior

vwhist128	<pre> input = Data from .tmp load; { for (i = 0; i < VELEM(16); i++) { bucket = input.h[i].ub[0]; weight = input.h[i].ub[1]; vindex = (bucket >> 3) & 0x1F; elindex = ((i>>1) & (~3)) ((bucket>>1) & 3); READ_EXT_VREG(vindex,tmp,0); tmp.uw[elindex] = (tmp.uw[elindex] + weight); WRITE_EXT_VREG(vindex,tmp,EXT_NEW); } </pre>
vwhist128(#u1)	<pre> input = Data from .tmp load; { for (i = 0; i < VELEM(16); i++) { bucket = input.h[i].ub[0]; weight = input.h[i].ub[1]; vindex = (bucket >> 3) & 0x1F; elindex = ((i>>1) & (~3)) ((bucket>>1) & 3); READ_EXT_VREG(vindex,tmp,0); if ((bucket & 1) == #u) tmp.uw[elindex] = (tmp.uw[elindex] + weight); WRITE_EXT_VREG(vindex,tmp, EXT_NEW);} </pre>

Syntax	Behavior
<code>vwhist128(Qv4)</code>	<pre> input = Data from .tmp load; { for (i = 0; i < VELEM(16); i++) { bucket = input.h[i].ub[0]; weight = input.h[i].ub[1]; vindex = (bucket >> 3) & 0x1F; elindex = ((i>>1) & (~3)) ((bucket>>1) & 3); READ_EXT_VREG(vindex,tmp,0); if (QvV[2*i]) tmp.uw[elindex] = (tmp.uw[elindex] + weight); WRITE_EXT_VREG(vindex,tmp, EXT_NEW); } </pre>
<code>vwhist128(Qv4,#u1)</code>	<pre> input = Data from .tmp load; { for (i = 0; i < VELEM(16); i++) { bucket = input.h[i].ub[0]; weight = input.h[i].ub[1]; vindex = (bucket >> 3) & 0x1F; elindex = ((i>>1) & (~3)) ((bucket>>1) & 3); READ_EXT_VREG(vindex,tmp,0); if (((bucket & 1) == #u) && QvV[2*i]) tmp.uw[elindex] = (tmp.uw[elindex] + weight); WRITE_EXT_VREG(vindex,tmp,EXT_NEW); } </pre>
<code>vwhist256</code>	<pre> input = Data from .tmp load; { for (i = 0; i < VELEM(16); i++) { bucket = input.h[i].ub[0]; weight = input.h[i].ub[1]; vindex = (bucket >> 3) & 0x1F; elindex = ((i>>0) & (~7)) ((bucket>>0) & 7); READ_EXT_VREG(vindex,tmp,0); tmp.uh[elindex] = (tmp.uh[elindex] + weight); WRITE_EXT_VREG(vindex,tmp,EXT_NEW); } </pre>
<code>vwhist256(Qv4)</code>	<pre> input = Data from .tmp load; { for (i = 0; i < VELEM(16); i++) { bucket = input.h[i].ub[0]; weight = input.h[i].ub[1]; vindex = (bucket >> 3) & 0x1F; elindex = ((i>>0) & (~7)) ((bucket>>0) & 7); READ_EXT_VREG(vindex,tmp,0); if (QvV[2*i]) tmp.uh[elindex] = (tmp.uh[elindex] + weight); WRITE_EXT_VREG(vindex,tmp,EXT_NEW); } </pre>

Syntax	Behavior
<code>vwhist256(Qv4):sat</code>	<pre> input = Data from .tmp load; { for (i = 0; i < VELEM(16); i++) { bucket = input.h[i].ub[0]; weight = input.h[i].ub[1]; vindex = (bucket >> 3) & 0x1F; elindex = ((i>>0) & (~7)) ((bucket >> 0) & 7); READ_EXT_VREG(vindex,tmp,0); if (QvV[2*i]) tmp.uh[elindex] = usat₁₆(tmp.uh[elindex] + weight); WRITE_EXT_VREG(vindex,tmp,EXT_NEW); } </pre>
<code>vwhist256:sat</code>	<pre> input = Data from .tmp load; { for (i = 0; i < VELEM(16); i++) { bucket = input.h[i].ub[0]; weight = input.h[i].ub[1]; vindex = (bucket >> 3) & 0x1F; elindex = ((i >> 0) & (~7)) ((bucket >> 0) & 7); READ_EXT_VREG(vindex,tmp,0); tmp.uh[elindex] = usat₁₆(tmp.uh[elindex] + weight); WRITE_EXT_VREG(vindex,tmp,EXT_NEW); } </pre>

Class: COPROC_VX (slots 0,1,2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse																
0	0	0	1	1	1	1	0	-	-	0	-	-	0	0	0	P	P	1	-	0	0	1	0	1	0	0	-	-	-	-	-	vwhist256
0	0	0	1	1	1	1	0	-	-	0	-	-	0	0	0	P	P	1	-	0	0	1	1	1	0	0	-	-	-	-	-	vwhist256:sat
0	0	0	1	1	1	1	0	-	-	0	-	-	0	0	0	P	P	1	-	0	1	0	-	1	0	0	-	-	-	-	-	vwhist128
0	0	0	1	1	1	1	0	-	-	0	-	-	0	0	0	P	P	1	-	0	1	1	i	1	0	0	-	-	-	-	-	vwhist128(#u1)
0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	0	P	P	1	-	-	0	1	0	1	0	0	-	-	-	-	-	vwhist256(Qv4)
0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	0	P	P	1	-	-	0	1	1	1	0	0	-	-	-	-	-	vwhist256(Qv4):sat
0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	0	P	P	1	-	-	1	0	-	1	0	0	-	-	-	-	-	vwhist128(Qv4)
0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	0	P	P	1	-	-	1	1	i	1	0	0	-	-	-	-	-	vwhist128(Qv4,#u1)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
v2	Field to encode register v

6.2 HVX ALU double resource

The HVX ALU double resource instruction subclass includes ALU instructions that use a pair of HVX resources.

Predicate operations

Perform bitwise logical operations between two vector predicate registers Qs and Qt, and place the result in Qd. The operations are element-size agnostic.

The following combinations are implemented: Qs & Qt, Qs & !Qt, Qs | Qt, Qs | !Qt, Qs ^ Qt. Interleave predicate bits from two vectors to match a shuffling operation such as vsat or vround. Forms that match word-to-halfword and halfword-to-byte shuffling are available.

Syntax	Behavior
<code>Qd4.b=vshuffe(Qs4.h,Qt4.h)</code>	<pre>for (i = 0; i < VELEM(8); i++) { QdV[i]=(i & 1) ? QsV[i-1] : QtV[i]; }</pre>
<code>Qd4.h=vshuffe(Qs4.w,Qt4.w)</code>	<pre>for (i = 0; i < VELEM(8); i++) { QdV[i]=(i & 2) ? QsV[i-2] : QtV[i]; }</pre>
<code>Qd4=and(Qs4,!Qt4)</code>	<pre>for (i = 0; i < VELEM(8); i++) { QdV[i]=QsV[i] && !QtV[i]; }</pre>
<code>Qd4=or(Qs4,!Qt4)</code>	<pre>for (i = 0; i < VELEM(8); i++) { QdV[i]=QsV[i] !QtV[i]; }</pre>
<code>Qd4=xor(Qs4,Qt4)</code>	<pre>for (i = 0; i < VELEM(8); i++) { QdV[i]=QsV[i] ^ QtV[i]; }</pre>

Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction uses any pair of the HVX resources (both multiply or shift/permute).

Intrinsics

<code>Qd4.b=vshuffe(Qs4.h,Qt4.h)</code>	HVX_VectorPred Q6_Qb_vshuffe_QhQh(HVX_VectorPred Qs, HVX_VectorPred Qt)
<code>Qd4.h=vshuffe(Qs4.w,Qt4.w)</code>	HVX_VectorPred Q6_Qh_vshuffe_QwQw(HVX_VectorPred Qs, HVX_VectorPred Qt)
<code>Qd4=and(Qs4,!Qt4)</code>	HVX_VectorPred Q6_Q_and_QQn(HVX_VectorPred Qs, HVX_VectorPred Qt)
<code>Qd4=and(Qs4,Qt4)</code>	HVX_VectorPred Q6_Q_and_QQ(HVX_VectorPred Qs, HVX_VectorPred Qt)
<code>Qd4=or(Qs4,!Qt4)</code>	HVX_VectorPred Q6_Q_or_QQn(HVX_VectorPred Qs, HVX_VectorPred Qt)
<code>Qd4=or(Qs4,Qt4)</code>	HVX_VectorPred Q6_Q_or_QQ(HVX_VectorPred Qs, HVX_VectorPred Qt)
<code>Qd4=xor(Qs4,Qt4)</code>	HVX_VectorPred Q6_Q_xor_QQ(HVX_VectorPred Qs, HVX_VectorPred Qt)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
ICLASS								t2								Parse								s2								d2		
0	0	0	1	1	1	1	0	t	t	0	-	-	-	1	1	P	P	0	-	-	-	s	s	0	0	0	0	0	0	0	d	d	Qd4=and(Qs4,Qt4)	
0	0	0	1	1	1	1	0	t	t	0	-	-	-	1	1	P	P	0	-	-	-	s	s	0	0	0	0	0	0	1	d	d	Qd4=or(Qs4,Qt4)	
0	0	0	1	1	1	1	0	t	t	0	-	-	-	1	1	P	P	0	-	-	-	s	s	0	0	0	0	1	1	d	d	Qd4=xor(Qs4,Qt4)		
0	0	0	1	1	1	1	0	t	t	0	-	-	-	1	1	P	P	0	-	-	-	s	s	0	0	0	1	0	0	d	d	Qd4=or(Qs4,!Qt4)		
0	0	0	1	1	1	1	0	t	t	0	-	-	-	1	1	P	P	0	-	-	-	s	s	0	0	0	1	0	1	d	d	Qd4=and(Qs4,!Qt4)		
0	0	0	1	1	1	1	0	t	t	0	-	-	-	1	1	P	P	0	-	-	-	s	s	0	0	0	1	1	0	d	d	Qd4.b=vshuffe(Qs4.h,Qt4.h)		
0	0	0	1	1	1	1	0	t	t	0	-	-	-	1	1	P	P	0	-	-	-	s	s	0	0	0	1	1	1	d	d	Qd4.h=vshuffe(Qs4.w,Qt4.w)		

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d2	Field to encode register d
s2	Field to encode register s
t2	Field to encode register t

Combine

Combine two input vector registers into a single destination vector register pair.

Using a scalar predicate, conditionally copy a single vector register to a destination vector register, or conditionally combine two input vectors into a destination vector register pair. A scalar predicate guards the entire operation. If the scalar predicate is true, the operation is performed. Otherwise the instruction is treated as a NOP.

Syntax	Behavior
<code>Vdd=vcombine (Vu,Vv)</code>	<pre>for (i = 0; i < VELEM(8); i++) { Vdd.v[0].ub[i] = Vv.ub[i]; Vdd.v[1].ub[i] = Vu.ub[i] ; }</pre>
<code>if (!Ps) Vdd=vcombine (Vu,Vv)</code>	<pre>if (!Ps[0]) { for (i = 0; i < VELEM(8); i++) { Vdd.v[0].ub[i] = Vv.ub[i]; Vdd.v[1].ub[i] = Vu.ub[i]; } } else { NOP;</pre>

Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction uses any pair of the HVX resources (both multiply or shift/permute).

Intrinsics

`Vdd=vcombine (Vu,Vv)` `HVX_VectorPair Q6_W_vcombine_VV(HVX_Vector Vu, HVX_Vector Vv)`

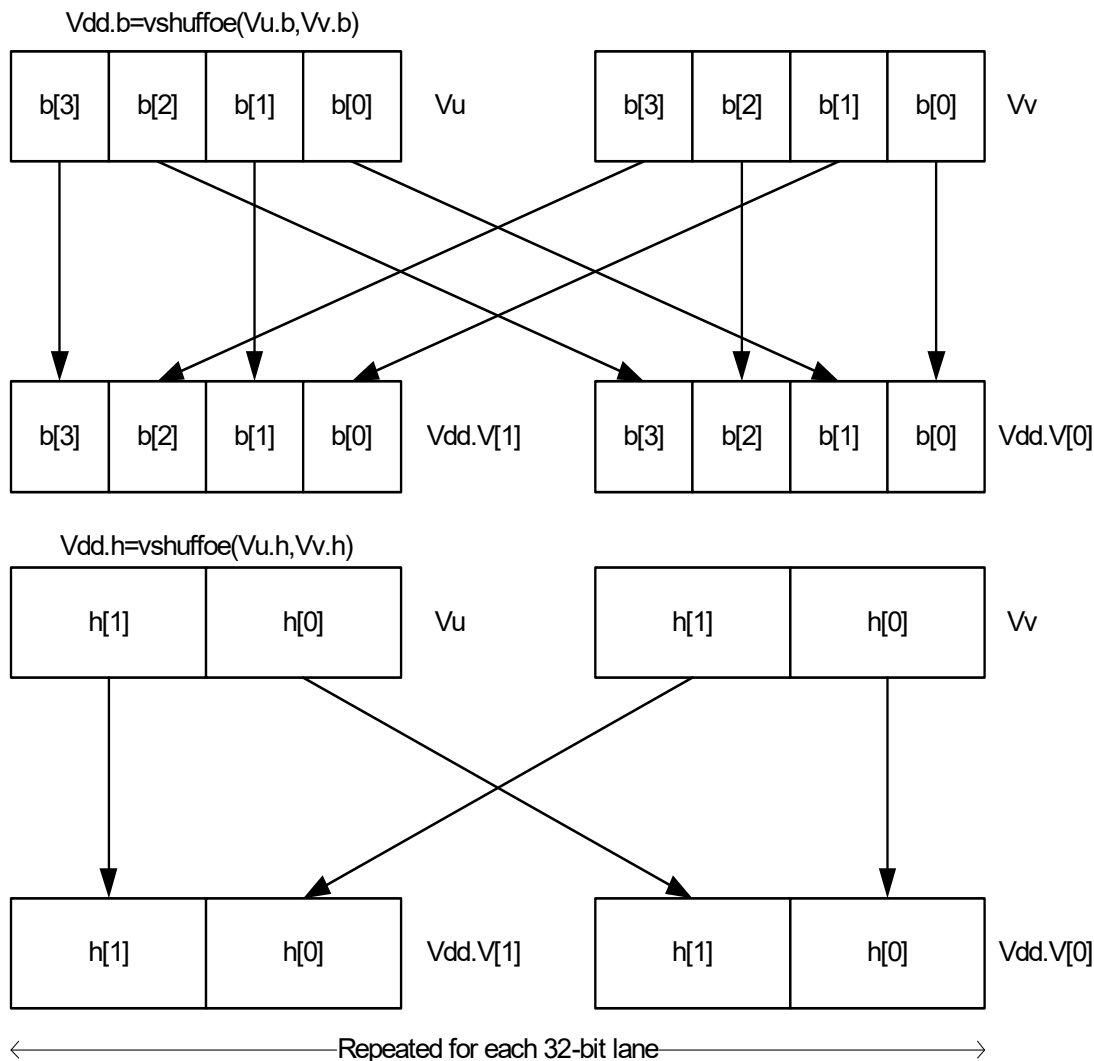
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
ICLASS																Parse				u5							s2		d5					
0	0	0	1	1	0	1	0	0	1	0	v	v	v	v	v	P	P	-	u	u	u	u	u	-	s	s	d	d	d	d	d	if (!Ps) Vdd=vcombine(Vu,Vv)		
0	0	0	1	1	0	1	0	0	1	1	v	v	v	v	v	P	P	-	u	u	u	u	u	-	s	s	d	d	d	d	d	if (Ps) Vdd=vcombine(Vu,Vv)		
ICLASS																Parse				u5									d5					
0	0	0	1	1	1	1	1	0	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vdd=vcombine(Vu,Vv)		

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
s2	Field to encode register s
u5	Field to encode register u
v5	Field to encode register v

In-lane shuffle

The `vshuffoe` instruction performs both the `vshuffo` and `vshuffe` operation at the same time, with even elements placed into the even vector register of `Vdd`, and odd elements placed in the odd vector register of the destination vector pair.



This group of shuffles is limited to bytes and halfwords.

Syntax

```
Vdd.b=vshuffoe(Vu.b,Vv.b)
```

Behavior

```
for (i = 0; i < VELEM(16); i++) {
    Vdd.v[0].uh[i].b[0]=Vv.uh[i].ub[0];
    Vdd.v[0].uh[i].b[1]=Vu.uh[i].ub[0];
    Vdd.v[1].uh[i].b[0]=Vv.uh[i].ub[1];
    Vdd.v[1].uh[i].b[1]=Vu.uh[i].ub[1] ;
}
```

Syntax

```
Vdd.h=vshuffoe(Vu.h,Vv.h)
```

Behavior

```
for (i = 0; i < VELEM(32); i++) {
    Vdd.v[0].uw[i].h[0]=Vv.uw[i].uh[0];
    Vdd.v[0].uw[i].h[1]=Vu.uw[i].uh[0];
    Vdd.v[1].uw[i].h[0]=Vv.uw[i].uh[1];
    Vdd.v[1].uw[i].h[1]=Vu.uw[i].uh[1] ;
}
```

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction uses any pair of the HVX resources (both multiply or shift/permute).

Intrinsics

```
Vdd.b=vshuffoe(Vu.b,Vv.b)
```

```
HVX_VectorPair
Q6_Wb_vshuffoe_VbVb(HVX_Vector Vu,
HVX_Vector Vv)
```

```
Vdd.h=vshuffoe(Vu.h,Vv.h)
```

```
HVX_VectorPair
Q6_Wh_vshuffoe_VhVh(HVX_Vector Vu,
HVX_Vector Vv)
```

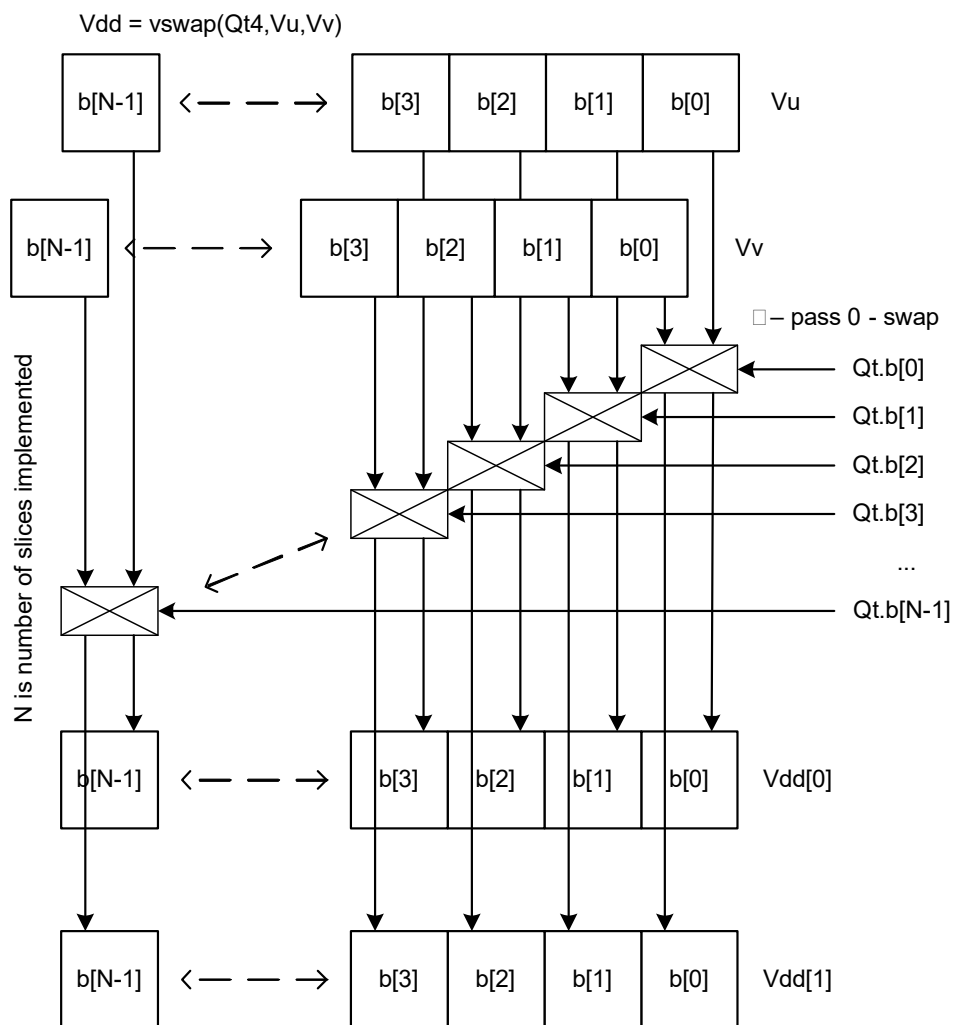
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
ICLASS																Parse		u5										d5					
0	0	0	1	1	1	1	1	0	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vdd.h=vshuffoe(Vu.h,Vv.h)	
0	0	0	1	1	1	1	1	0	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vdd.b=vshuffoe(Vu.b,Vv.b)	

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Swap

Based on a predicate bit in a vector predicate register, if the bit is set the corresponding byte from vector register Vu is placed in the even destination vector register of Vdd, and the byte from Vv is placed in the even destination vector register of Vdd. Otherwise, the corresponding byte from Vv is written to the even register, and Vu to the odd register. The operation works on bytes so it can handle all data sizes. It is similar to the vmux operation, but places the opposite case output into the odd vector register of the destination vector register pair.



Syntax

```
Vdd=vswap (Qt4, Vu, Vv)
```

Behavior

```
for (i = 0; i < VELEM(8); i++) {
    Vdd.v[0].ub[i] = QtV[i] ? Vu.ub[i] : Vv.ub[i];
    Vdd.v[1].ub[i] = !QtV[i] ? Vu.ub[i] : Vv.ub[i] ;
}
```

Syntax

```
Vdd=vswap(Qt8,Vu,Vv)
```

Behavior

```
int i;
for (i = 0; i < 1024/8; i++) {
    Vdd.V8u[i] = ( QtV[i] ? Vu.V8u[i] : Vv.V8u[i]);
    Vdd.V8u[i+128] = (!QtV[i] ? Vu.V8u[i] : Vv.V8u[i]);
}
```

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction uses any pair of the HVX resources (both multiply or shift/permute).

Intrinsics

```
Vdd=vswap(Qt8,Vu,Vv)
```

```
VectorPair W_vswap_QVV(VectorPred Qt, Vector Vu,
Vector Vv)
```

```
Vdd=vswap(Qt4,Vu,Vv)
```

```
HVX_VectorPair Q6_W_vswap_QVV(HVX_VectorPred Qt,
HVX_Vector Vu, HVX_Vector Vv)
```

Encoding

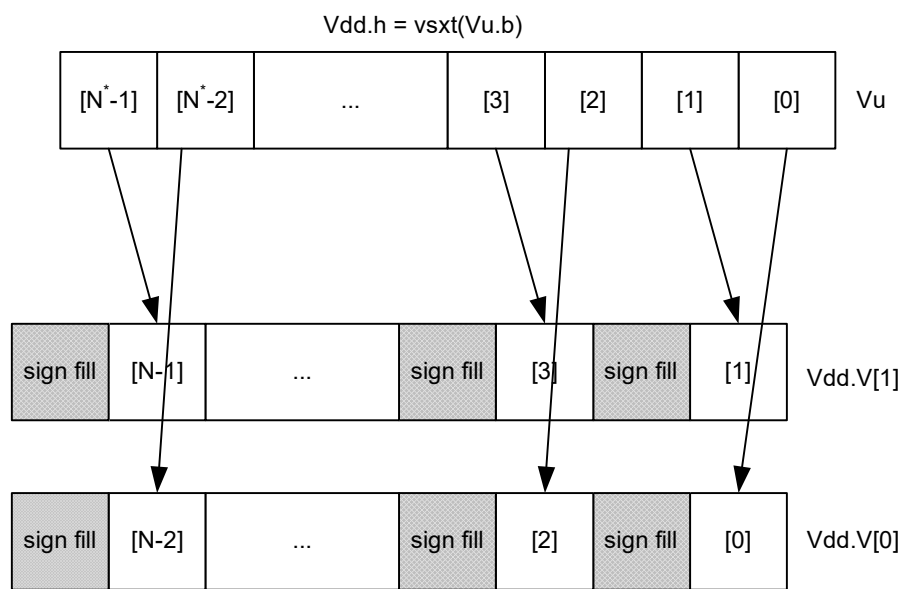
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse		u5							t2		d5					
0	0	0	1	1	1	1	0	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	-	t	t	d	d	d	d	d	Vdd=vswap(Qt4,Vu,Vv)
ICLASS																u5		Parse							d5					t3		
0	0	0	1	1	1	1	1	1	0	1	u	u	u	u	u	P	P	0	v	v	v	v	v	d	d	d	d	d	t	t	t	Vdd=vswap(Qt8,Vu,Vv)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t2	Field to encode register t
t3	Field to encode register t
u5	Field to encode register u
v5	Field to encode register v

Sign/zero extension

Perform sign extension on each even element in V_u , and place it in the even destination vector register $V_{dd}[0]$. Odd elements are sign-extended and placed in the odd destination vector register $V_{dd}[1]$. Bytes are converted to halfwords, and halfwords are converted to words.

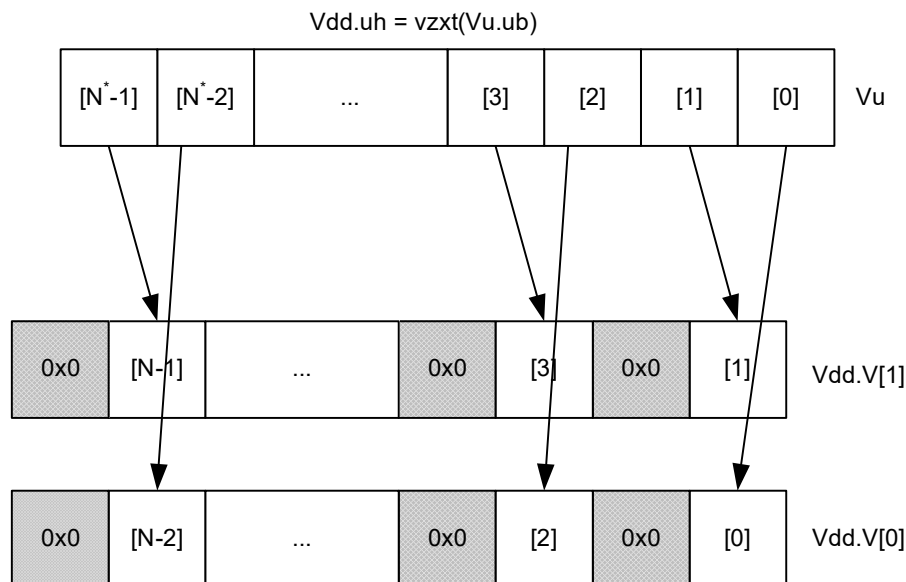
Sign extension of words is a cross-lane operation, and can only execute on the permute slot.



*N is number of operations in vector

Perform zero extension on each even element in V_u , and place it in the even destination vector register $V_{dd}[0]$. Odd elements are zero-extended and placed in the odd destination vector register $V_{dd}[1]$. Bytes are converted to halfwords, and halfwords are converted to words.

Zero extension of words is a cross-lane operation, and can only execute on the permute slot.



*N is number of operations in vector

Syntax	Behavior
$Vdd.h = vsxt(Vu.b)$	<pre>for (i = 0; i < VELEM(16); i++) { Vdd.v[0].h[i] = Vu.h[i].b[0]; Vdd.v[1].h[i] = Vu.h[i].b[1]; }</pre>
$Vdd.uh = vzxt(Vu.ub)$	<pre>for (i = 0; i < VELEM(16); i++) { Vdd.v[0].uh[i] = Vu.uh[i].ub[0]; Vdd.v[1].uh[i] = Vu.uh[i].ub[1]; }</pre>
$Vdd.uw = vzxt(Vu.uh)$	<pre>for (i = 0; i < VELEM(32); i++) { Vdd.v[0].uw[i] = Vu.uw[i].uh[0]; Vdd.v[1].uw[i] = Vu.uw[i].uh[1]; }</pre>
$Vdd.w = vsxt(Vu.h)$	<pre>for (i = 0; i < VELEM(32); i++) { Vdd.v[0].w[i] = Vu.w[i].h[0]; Vdd.v[1].w[i] = Vu.w[i].h[1]; }</pre>

Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction uses any pair of the HVX resources (both multiply or shift/permute).

Intrinsics

Vdd.h=vsxt (Vu.b)	HVX_VectorPair Q6_Wh_vsxt_Vb (HVX_Vector Vu)
Vdd.uh=vzxt (Vu.ub)	HVX_VectorPair Q6_Wuh_vzxt_Vub (HVX_Vector Vu)
Vdd.uw=vzxt (Vu.uh)	HVX_VectorPair Q6_Wuw_vzxt_Vuh (HVX_Vector Vu)
Vdd.w=vsxt (Vu.h)	HVX_VectorPair Q6_Ww_vsxt_Vh (HVX_Vector Vu)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
ICLASS																Parse		u5										d5					
0	0	0	1	1	1	1	0	-	-	0	-	-	-	1	0	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vdd.uh=vzxt(Vu.ub)	
0	0	0	1	1	1	1	0	-	-	0	-	-	-	1	0	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vdd.uw=vzxt(Vu.uh)	
0	0	0	1	1	1	1	0	-	-	0	-	-	-	1	0	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vdd.h=vsxt(Vu.b)	
0	0	0	1	1	1	1	0	-	-	0	-	-	-	1	0	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vdd.w=vsxt(Vu.h)	

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u

Arithmetic

Perform simple arithmetic operations, add and subtract, between the elements of the two vectors Vu and Vv. Supports word, halfword (signed and unsigned), and byte (signed and unsigned).

Optionally saturate for word and halfword. Always saturate for unsigned types.

Syntax	Behavior
<code>Vdd.b=vadd(Vuu.b,Vvv.b)[:sat]</code>	<pre>for (i = 0; i < VELEM(8); i++) { Vdd.v[0].b[i] = [sat₈](Vuu.v[0].b[i] + Vvv.v[0].b[i]); Vdd.v[1].b[i] = [sat₈](Vuu.v[1].b[i] + Vvv.v[1].b[i]); }</pre>
<code>Vdd.b=vsub(Vuu.b,Vvv.b)[:sat]</code>	<pre>for (i = 0; i < VELEM(8); i++) { Vdd.v[0].b[i] = [sat₈](Vuu.v[0].b[i]- Vvv.v[0].b[i]); Vdd.v[1].b[i] = [sat₈](Vuu.v[1].b[i]- Vvv.v[1].b[i]); }</pre>
<code>Vdd.h=vadd(Vuu.h,Vvv.h)[:sat]</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vdd.v[0].h[i] = [sat₁₆](Vuu.v[0].h[i] + Vvv.v[0].h[i]); Vdd.v[1].h[i] = [sat₁₆](Vuu.v[1].h[i] +Vvv.v[1].h[i]); }</pre>
<code>Vdd.h=vsub(Vuu.h,Vvv.h)[:sat]</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vdd.v[0].h[i] = [sat₁₆](Vuu.v[0].h[i]- Vvv.v[0].h[i]); Vdd.v[1].h[i] = [sat₁₆](Vuu.v[1].h[i]- Vvv.v[1].h[i]); }</pre>
<code>Vdd.ub=vadd(Vuu.ub,Vvv.ub):sat</code>	<pre>for (i = 0; i < VELEM(8); i++) { Vdd.v[0].ub[i] = usat₈(Vuu.v[0].ub[i] +Vvv.v[0].ub[i]); Vdd.v[1].ub[i] = usat₈(Vuu.v[1].ub[i] + Vvv.v[1].ub[i]); }</pre>
<code>Vdd.ub=vsub(Vuu.ub,Vvv.ub):sat</code>	<pre>for (i = 0; i < VELEM(8); i++) { Vdd.v[0].ub[i] = usat₈(Vuu.v[0].ub[i]- Vvv.v[0].ub[i]); Vdd.v[1].ub[i] = usat₈(Vuu.v[1].ub[i]- Vvv.v[1].ub[i]); }</pre>
<code>Vdd.uh=vadd(Vuu.uh,Vvv.uh):sat</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vdd.v[0].uh[i] = usat₁₆(Vuu.v[0].uh[i] + Vvv.v[0].uh[i]); Vdd.v[1].uh[i] = usat₁₆(Vuu.v[1].uh[i] + Vvv.v[1].uh[i]); }</pre>

Syntax	Behavior
<code>Vdd.uh=vsub(Vuu.uh,Vvv.uh):sat</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vdd.v[0].uh[i] = usat₁₆(Vuu.v[0].uh[i]- Vvv.v[0].uh[i]); Vdd.v[1].uh[i] = usat₁₆(Vuu.v[1].uh[i]- Vvv.v[1].uh[i]); }</pre>
<code>Vdd.uw=vadd(Vuu.uw,Vvv.uw):sat</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vdd.v[0].uw[i] = usat₃₂(Vuu.v[0].uw[i] +Vvv.v[0].uw[i]); Vdd.v[1].uw[i] = ussat₃₂(Vuu.v[1].uw[i] +Vvv.v[1].uw[i]); }</pre>
<code>Vdd.uw=vsub(Vuu.uw,Vvv.uw):sat</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vdd.v[0].uw[i] = usat₃₂(Vuu.v[0].uw[i]- Vvv.v[0].uw[i]); Vdd.v[1].uw[i] = usat₃₂(Vuu.v[1].uw[i]- Vvv.v[1].uw[i]); }</pre>
<code>Vdd.w=vadd(Vuu.w,Vvv.w)[:sat]</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vdd.v[0].w[i] = [sat₃₂](Vuu.v[0].w[i] +Vvv.v[0].w[i]); Vdd.v[1].w[i] = [sat₃₂](Vuu.v[1].w[i] +Vvv.v[1].w[i]); }</pre>
<code>Vdd.w=vsub(Vuu.w,Vvv.w)[:sat]</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vdd.v[0].w[i] = [sat₃₂](Vuu.v[0].w[i]- Vvv.v[0].w[i]); Vdd.v[1].w[i] = [sat₃₂](Vuu.v[1].w[i]- Vvv.v[1].w[i]); }</pre>

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction uses any pair of the HVX resources (both multiply or shift/permute).

Intrinsics

<code>Vdd.b=vadd(Vuu.b,Vvv.b)</code>	<code>HVX_VectorPair Q6_Wb_vadd_WbWb(HVX_VectorPair Vuu, HVX_VectorPair Vvv)</code>
<code>Vdd.b=vadd(Vuu.b,Vvv.b):sat</code>	<code>HVX_VectorPair Q6_Wb_vadd_WbWb_sat(HVX_VectorPair Vuu, HVX_VectorPair Vvv)</code>
<code>Vdd.b=vsub(Vuu.b,Vvv.b)</code>	<code>HVX_VectorPair Q6_Wb_vsub_WbWb(HVX_VectorPair Vuu, HVX_VectorPair Vvv)</code>

Vdd.b=vsub(Vuu.b,Vvv.b):sat	HVX_VectorPair Q6_Wb_vsub_WbWb_sat(HVX_VectorPair Vuu, HVX_VectorPair Vvv)
Vdd.h=vadd(Vuu.h,Vvv.h)	HVX_VectorPair Q6_Wh_vadd_WhWh(HVX_VectorPair Vuu, HVX_VectorPair Vvv)
Vdd.h=vadd(Vuu.h,Vvv.h):sat	HVX_VectorPair Q6_Wh_vadd_WhWh_sat(HVX_VectorPair Vuu, HVX_VectorPair Vvv)
Vdd.h=vsub(Vuu.h,Vvv.h)	HVX_VectorPair Q6_Wh_vsub_WhWh(HVX_VectorPair Vuu, HVX_VectorPair Vvv)
Vdd.h=vsub(Vuu.h,Vvv.h):sat	HVX_VectorPair Q6_Wh_vsub_WhWh_sat(HVX_VectorPair Vuu, HVX_VectorPair Vvv)
Vdd.ub=vadd(Vuu.ub,Vvv.ub):sat	HVX_VectorPair Q6_Wub_vadd_WubWub_sat(HVX_VectorPair Vuu, HVX_VectorPair Vvv)
Vdd.ub=vsub(Vuu.ub,Vvv.ub):sat	HVX_VectorPair Q6_Wub_vsub_WubWub_sat(HVX_VectorPair Vuu, HVX_VectorPair Vvv)
Vdd.uh=vadd(Vuu.uh,Vvv.uh):sat	HVX_VectorPair Q6_Wuh_vadd_WuhWuh_sat(HVX_VectorPair Vuu, HVX_VectorPair Vvv)
Vdd.uh=vsub(Vuu.uh,Vvv.uh):sat	HVX_VectorPair Q6_Wuh_vsub_WuhWuh_sat(HVX_VectorPair Vuu, HVX_VectorPair Vvv)
Vdd.uw=vadd(Vuu.uw,Vvv.uw):sat	HVX_VectorPair Q6_Wuw_vadd_WuwWuw_sat(HVX_VectorPair Vuu, HVX_VectorPair Vvv)
Vdd.uw=vsub(Vuu.uw,Vvv.uw):sat	HVX_VectorPair Q6_Wuw_vsub_WuwWuw_sat(HVX_VectorPair Vuu, HVX_VectorPair Vvv)
Vdd.w=vadd(Vuu.w,Vvv.w)	HVX_VectorPair Q6_Ww_vadd_WwWw(HVX_VectorPair Vuu, HVX_VectorPair Vvv)
Vdd.w=vadd(Vuu.w,Vvv.w):sat	HVX_VectorPair Q6_Ww_vadd_WwWw_sat(HVX_VectorPair Vuu, HVX_VectorPair Vvv)
Vdd.w=vsub(Vuu.w,Vvv.w)	HVX_VectorPair Q6_Ww_vsub_WwWw(HVX_VectorPair Vuu, HVX_VectorPair Vvv)
Vdd.w=vsub(Vuu.w,Vvv.w):sat	HVX_VectorPair Q6_Ww_vsub_WwWw_sat(HVX_VectorPair Vuu, HVX_VectorPair Vvv)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS															Parse		u5										d5					
0	0	0	1	1	1	0	0	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vdd.b=vadd(Vuu.b,Vvv.b)
0	0	0	1	1	1	0	0	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vdd.h=vadd(Vuu.h,Vvv.h)
0	0	0	1	1	1	0	0	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vdd.w=vadd(Vuu.w,Vvv.w)
0	0	0	1	1	1	0	0	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vdd.ub=vadd(Vuu.ub,Vvv.ub):sat
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vdd.uh=vadd(Vuu.uh,Vvv.uh):sat
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vdd.h=vadd(Vuu.h,Vvv.h):sat
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vdd.w=vadd(Vuu.w,Vvv.w):sat
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vdd.b=vsub(Vuu.b,Vvv.b)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vdd.h=vsub(Vuu.h,Vvv.h)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vdd.w=vsub(Vuu.w,Vvv.w)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vdd.ub=vsub(Vuu.ub,Vvv.ub):sat
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vdd.uh=vsub(Vuu.uh,Vvv.uh):sat
0	0	0	1	1	1	0	0	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vdd.h=vsub(Vuu.h,Vvv.h):sat
0	0	0	1	1	1	0	0	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vdd.w=vsub(Vuu.w,Vvv.w):sat
0	0	0	1	1	1	1	0	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vdd.b=vadd(Vuu.b,Vvv.b):sat
0	0	0	1	1	1	1	0	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vdd.b=vsub(Vuu.b,Vvv.b):sat
0	0	0	1	1	1	1	0	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vdd.uw=vadd(Vuu.uw,Vvv.uw):sat
0	0	0	1	1	1	1	0	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vdd.uw=vsub(Vuu.uw,Vvv.uw):sat

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

6.3 HVX ALU resource

The HVX ALU resource instruction subclass includes ALU instructions that use a single HVX resource.

Predicate operations

Perform bitwise logical operation on a vector predicate register Qs, and place the result in Qd. This operation works on vectors with any element size.

The following combination is implemented: !Qs.

Syntax	Behavior
<code>Qd4=not(Qs4)</code>	<pre>for (i = 0; i < VELEM(8); i++) { QdV[i] =!QsV[i] ; }</pre>

Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction can use any HVX resource.

Intrinsics

<code>Qd4=not(Qs4)</code>	<code>HVX_VectorPred Q6_Q_not_Q(HVX_VectorPred Qs)</code>
---------------------------	---

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse						s2						d2				
0	0	0	1	1	1	1	0	-	-	0	-	-	-	1	1	P	P	0	-	-	-	s	s	0	0	0	0	1	0	d	d	Qd4=not(Qs4)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d2	Field to encode register d
s2	Field to encode register s

Byte-conditional vector assign

If the bit in Qv is set, copy the byte. Otherwise, set the byte in the destination to zero.

Syntax

```
Vd=vand(!Qv4,Vu)
```

Behavior

```
for (i = 0; i < VELEM(8); i++) {
    Vd.b[i] = !QvV[i] ? Vu.b[i] : 0 ;
}
```

Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction can use any HVX resource.

Intrinsics

```
Vd=vand(!Qv4,Vu)
```

```
HVX_Vector Q6_V_vand_QnV(HVX_VectorPred Qv,
HVX_Vector Vu)
```

```
Vd=vand(Qv4,Vu)
```

```
HVX_Vector Q6_V_vand_QV(HVX_VectorPred Qv,
HVX_Vector Vu)
```

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
ICLASS																Parse		u5										d5					
0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	1	P	P	1	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd=vand(Qv4,Vu)	
0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	1	P	P	1	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd=vand(!Qv4,Vu)	

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v2	Field to encode register v

Min/max

Compare the respective elements of Vu and Vv, and return the maximum or minimum. The result is placed in the same position as the inputs.

Supports unsigned byte, signed and unsigned halfword, and signed word.

Syntax	Behavior
Vd.b=vmax(Vu.b,Vv.b)	<pre>for (i = 0; i < VELEM(8); i++) { Vd.b[i] = (Vu.b[i] > Vv.b[i]) ? Vu.b[i] : Vv.b[i]; }</pre>
Vd.b=vmin(Vu.b,Vv.b)	<pre>for (i = 0; i < VELEM(8); i++) { Vd.b[i] = (Vu.b[i] < Vv.b[i]) ? Vu.b[i] : Vv.b[i]; }</pre>
Vd.h=vmax(Vu.h,Vv.h)	<pre>for (i = 0; i < VELEM(16); i++) { Vd.h[i] = (Vu.h[i] > Vv.h[i]) ? Vu.h[i] : Vv.h[i]; }</pre>
Vd.h=vmin(Vu.h,Vv.h)	<pre>for (i = 0; i < VELEM(16); i++) { Vd.h[i] = (Vu.h[i] < Vv.h[i]) ? Vu.h[i] : Vv.h[i]; }</pre>
Vd.hf=vmax(Vu.hf,Vv.hf)	<pre>for (i = 0; i < VELEM(16); i++) { Vd.hf[i] = max(Vu.hf[i],Vv.hf[i]); }</pre>
Vd.hf=vmin(Vu.hf,Vv.hf)	<pre>for (i = 0; i < VELEM(16); i++) { Vd.hf[i] = min(Vu.hf[i],Vv.hf[i]); }</pre>
Vd.sf=vmax(Vu.sf,Vv.sf)	<pre>for (i = 0; i < VELEM(32); i++) { Vd.sf[i] = max(Vu.sf[i],Vv.sf[i]); }</pre>
Vd.sf=vmin(Vu.sf,Vv.sf)	<pre>for (i = 0; i < VELEM(32); i++) { Vd.sf[i] = min(Vu.sf[i],Vv.sf[i]); }</pre>
Vd.ub=vmax(Vu.ub,Vv.ub)	<pre>for (i = 0; i < VELEM(8); i++) { Vd.ub[i] = (Vu.ub[i] > Vv.ub[i]) ? Vu.ub[i] : Vv.ub[i]; }</pre>
Vd.ub=vmin(Vu.ub,Vv.ub)	<pre>for (i = 0; i < VELEM(8); i++) { Vd.ub[i] = (Vu.ub[i] < Vv.ub[i]) ? Vu.ub[i] : Vv.ub[i]; }</pre>
Vd.uh=vmax(Vu.uh,Vv.uh)	<pre>for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = (Vu.uh[i] > Vv.uh[i]) ? Vu.uh[i] : Vv.uh[i]; }</pre>
Vd.uh=vmin(Vu.uh,Vv.uh)	<pre>for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = (Vu.uh[i] < Vv.uh[i]) ? Vu.uh[i] : Vv.uh[i]; }</pre>

Syntax	Behavior
<code>Vd.w=vmax(Vu.w,Vv.w)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i] = (Vu.w[i] > Vv.w[i]) ? Vu.w[i] : Vv.w[i]; }</pre>
<code>Vd.w=vmin(Vu.w,Vv.w)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i] = (Vu.w[i] < Vv.w[i]) ? Vu.w[i] : Vv.w[i]; }</pre>

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction can use any HVX resource.

Intrinsics

<code>Vd.b=vmax(Vu.b,Vv.b)</code>	<code>HVX_Vector Q6_Vb_vmax_VbVb(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.b=vmin(Vu.b,Vv.b)</code>	<code>HVX_Vector Q6_Vb_vmin_VbVb(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.h=vmax(Vu.h,Vv.h)</code>	<code>HVX_Vector Q6_Vh_vmax_VhVh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.h=vmin(Vu.h,Vv.h)</code>	<code>HVX_Vector Q6_Vh_vmin_VhVh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.hf=vmax(Vu.hf,Vv.hf)</code>	<code>HVX_Vector Q6_Vhf_vmax_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.hf=vmin(Vu.hf,Vv.hf)</code>	<code>HVX_Vector Q6_Vhf_vmin_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.sf=vmax(Vu.sf,Vv.sf)</code>	<code>HVX_Vector Q6_Vsf_vmax_VsfVsf(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.sf=vmin(Vu.sf,Vv.sf)</code>	<code>HVX_Vector Q6_Vsf_vmin_VsfVsf(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.ub=vmax(Vu.ub,Vv.ub)</code>	<code>HVX_Vector Q6_Vub_vmax_VubVub(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.ub=vmin(Vu.ub,Vv.ub)</code>	<code>HVX_Vector Q6_Vub_vmin_VubVub(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.uh=vmax(Vu.uh,Vv.uh)</code>	<code>HVX_Vector Q6_Vuh_vmax_VuhVuh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.uh=vmin(Vu.uh,Vv.uh)</code>	<code>HVX_Vector Q6_Vuh_vmin_VuhVuh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.w=vmax(Vu.w,Vv.w)</code>	<code>HVX_Vector Q6_Vw_vmax_VwVw(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.w=vmin(Vu.w,Vv.w)</code>	<code>HVX_Vector Q6_Vw_vmin_VwVw(HVX_Vector Vu, HVX_Vector Vv)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse				u5							d5					
0	0	0	1	1	1	1	1	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.ub=vmin(Vu.ub,Vv.ub)
0	0	0	1	1	1	1	1	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.uh=vmin(Vu.uh,Vv.uh)
0	0	0	1	1	1	1	1	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.h=vmin(Vu.h,Vv.h)
0	0	0	1	1	1	1	1	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.w=vmin(Vu.w,Vv.w)
0	0	0	1	1	1	1	1	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.ub=vmax(Vu.ub,Vv.ub)
0	0	0	1	1	1	1	1	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.uh=vmax(Vu.uh,Vv.uh)
0	0	0	1	1	1	1	1	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.h=vmax(Vu.h,Vv.h)
0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.w=vmax(Vu.w,Vv.w)
0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.b=vmin(Vu.b,Vv.b)
0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.b=vmax(Vu.b,Vv.b)
0	0	0	1	1	1	1	1	1	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.sf=vmax(Vu.sf,Vv.sf)
0	0	0	1	1	1	1	1	1	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.sf=vmin(Vu.sf,Vv.sf)
0	0	0	1	1	1	1	1	1	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.hf=vmax(Vu.hf,Vv.hf)
0	0	0	1	1	1	1	1	1	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.hf=vmin(Vu.hf,Vv.hf)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Absolute value

Take the absolute value of the vector register elements. Supports signed halfword and word. Optionally saturate to deal with the max negative value overflow case.

Syntax	Behavior
<code>Vd.b=vabs(Vu.b)[:sat]</code>	<pre>for (i = 0; i < VELEM(8); i++) { Vd.b[i] = [sat₈](ABS(Vu.b[i])); }</pre>
<code>Vd.h=vabs(Vu.h)[:sat]</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.h[i] = [sat₁₆](ABS(Vu.h[i])); }</pre>
<code>Vd.ub=vabs(Vu.b)</code>	Assembler mapped to: <code>"Vd.b=vabs(Vu.b)"</code>
<code>Vd.uh=vabs(Vu.h)</code>	Assembler mapped to: <code>"Vd.h=vabs(Vu.h)"</code>
<code>Vd.uw=vabs(Vu.w)</code>	Assembler mapped to: <code>"Vd.w=vabs(Vu.w)"</code>
<code>Vd.w=vabs(Vu.w)[:sat]</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i] = [sat₃₂](ABS(Vu.w[i])); }</pre>

Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction can use any HVX resource.

Intrinsics

<code>Vd.b=vabs(Vu.b)</code>	<code>HVX_Vector Q6_Vb_vabs_Vb(HVX_Vector Vu)</code>
<code>Vd.b=vabs(Vu.b):sat</code>	<code>HVX_Vector Q6_Vb_vabs_Vb_sat(HVX_Vector Vu)</code>
<code>Vd.h=vabs(Vu.h)</code>	<code>HVX_Vector Q6_Vh_vabs_Vh(HVX_Vector Vu)</code>
<code>Vd.h=vabs(Vu.h):sat</code>	<code>HVX_Vector Q6_Vh_vabs_Vh_sat(HVX_Vector Vu)</code>
<code>Vd.w=vabs(Vu.w)</code>	<code>HVX_Vector Q6_Vw_vabs_Vw(HVX_Vector Vu)</code>
<code>Vd.w=vabs(Vu.w):sat</code>	<code>HVX_Vector Q6_Vw_vabs_Vw_sat(HVX_Vector Vu)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
ICLASS																Parse						u5								d5					
0	0	0	1	1	1	1	0	-	-	0	-	-	-	0	0	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.h=vabs(Vu.h)			
0	0	0	1	1	1	1	0	-	-	0	-	-	-	0	0	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.h=vabs(Vu.h):sat			
0	0	0	1	1	1	1	0	-	-	0	-	-	-	0	0	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.w=vabs(Vu.w)			
0	0	0	1	1	1	1	0	-	-	0	-	-	-	0	0	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.w=vabs(Vu.w):sat			
0	0	0	1	1	1	1	0	-	-	0	-	-	-	0	1	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.b=vabs(Vu.b)			
0	0	0	1	1	1	1	0	-	-	0	-	-	-	0	1	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.b=vabs(Vu.b):sat			

Field name		Description
ICLASS		Instruction class
Parse		Packet loop parse bits
d5		Field to encode register d
u5		Field to encode register u

Arithmetic

Perform simple arithmetic operations, add and subtract, between the elements of the two vectors Vu and Vv. Supports unsigned and signed byte and halfword.

Optionally saturate for word and signed halfword. Always saturate for unsigned types except byte.

Syntax	Behavior
<code>Vd.b=vadd(Vu.b,Vv.b)[:sat]</code>	for (i = 0; i < VELEM(8); i++) { Vd.b[i] = [sat ₈](Vu.b[i]+Vv.b[i]); }
<code>Vd.b=vsub(Vu.b,Vv.b)[:sat]</code>	for (i = 0; i < VELEM(8); i++) { Vd.b[i] = [sat ₈](Vu.b[i]-Vv.b[i]); }
<code>Vd.h=vadd(Vu.h,Vv.h)[:sat]</code>	for (i = 0; i < VELEM(16); i++) { Vd.h[i] = [sat ₁₆](Vu.h[i]+Vv.h[i]); }
<code>Vd.h=vsub(Vu.h,Vv.h)[:sat]</code>	for (i = 0; i < VELEM(16); i++) { Vd.h[i] = [sat ₁₆](Vu.h[i]-Vv.h[i]); }
<code>Vd.ub=vadd(Vu.ub,Vv.b):sat</code>	for (i = 0; i < VELEM(8); i++) { Vd.ub[i] = usat ₈ (Vu.ub[i] + Vv.b[i]); }
<code>Vd.ub=vadd(Vu.ub,Vv.ub):sat</code>	for (i = 0; i < VELEM(8); i++) { Vd.ub[i] = usat ₈ (Vu.ub[i]+Vv.ub[i]); }
<code>Vd.ub=vsub(Vu.ub,Vv.b):sat</code>	for (i = 0; i < VELEM(8); i++) { Vd.ub[i] = usat ₈ (Vu.ub[i] - Vv.b[i]); }
<code>Vd.ub=vsub(Vu.ub,Vv.ub):sat</code>	for (i = 0; i < VELEM(8); i++) { Vd.ub[i] = usat ₈ (Vu.ub[i]-Vv.ub[i]); }
<code>Vd.uh=vadd(Vu.uh,Vv.uh):sat</code>	for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = usat ₁₆ (Vu.uh[i]+Vv.uh[i]); }
<code>Vd.uh=vsub(Vu.uh,Vv.uh):sat</code>	for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = usat ₁₆ (Vu.uh[i]-Vv.uh[i]); }
<code>Vd.uw=vadd(Vu.uw,Vv.uw):sat</code>	for (i = 0; i < VELEM(32); i++) { Vd.uw[i] = usat ₃₂ (Vu.uw[i]+Vv.uw[i]); }
<code>Vd.uw=vsub(Vu.uw,Vv.uw):sat</code>	for (i = 0; i < VELEM(32); i++) { Vd.uw[i] = usat ₃₂ (Vu.uw[i]-Vv.uw[i]); }
<code>Vd.w=vadd(Vu.w,Vv.w)[:sat]</code>	for (i = 0; i < VELEM(32); i++) { Vd.w[i] = [sat ₃₂](Vu.w[i]+Vv.w[i]); }

Syntax	Behavior
<code>Vd.w=vsub(Vu.w,Vv.w)[:sat]</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vd.w[i] = [sat₃₂](Vu.w[i]-Vv.w[i]); } </pre>

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction can use any HVX resource.

Intrinsics

<code>Vd.b=vadd(Vu.b,Vv.b)</code>	<code>HVX_Vector Q6_Vb_vadd_VbVb(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.b=vadd(Vu.b,Vv.b):sat</code>	<code>HVX_Vector Q6_Vb_vadd_VbVb_sat(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.b=vsub(Vu.b,Vv.b)</code>	<code>HVX_Vector Q6_Vb_vsub_VbVb(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.b=vsub(Vu.b,Vv.b):sat</code>	<code>HVX_Vector Q6_Vb_vsub_VbVb_sat(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.h=vadd(Vu.h,Vv.h)</code>	<code>HVX_Vector Q6_Vh_vadd_VhVh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.h=vadd(Vu.h,Vv.h):sat</code>	<code>HVX_Vector Q6_Vh_vadd_VhVh_sat(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.h=vsub(Vu.h,Vv.h)</code>	<code>HVX_Vector Q6_Vh_vsub_VhVh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.h=vsub(Vu.h,Vv.h):sat</code>	<code>HVX_Vector Q6_Vh_vsub_VhVh_sat(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.ub=vadd(Vu.ub,Vv.b):sat</code>	<code>HVX_Vector Q6_Vub_vadd_VubVb_sat(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.ub=vadd(Vu.ub,Vv.ub):sat</code>	<code>HVX_Vector Q6_Vub_vadd_VubVub_sat(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.ub=vsub(Vu.ub,Vv.b):sat</code>	<code>HVX_Vector Q6_Vub_vsub_VubVb_sat(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.ub=vsub(Vu.ub,Vv.ub):sat</code>	<code>HVX_Vector Q6_Vub_vsub_VubVub_sat(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.uh=vadd(Vu.uh,Vv.uh):sat</code>	<code>HVX_Vector Q6_Vuh_vadd_VuhVuh_sat(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.uh=vsub(Vu.uh,Vv.uh):sat</code>	<code>HVX_Vector Q6_Vuh_vsub_VuhVuh_sat(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.uw=vadd(Vu.uw,Vv.uw):sat</code>	<code>HVX_Vector Q6_Vuw_vadd_VuwVuw_sat(HVX_Vector Vu, HVX_Vector Vv)</code>

Vd.uw=vsub(Vu.uw,Vv.uw):sat	HVX_Vector Q6_Vuw_vsub_VuwVu_w_sat(HVX_Vector Vu, HVX_Vector Vv)
Vd.w=vadd(Vu.w,Vv.w)	HVX_Vector Q6_Vw_vadd_VwVw(HVX_Vector Vu, HVX_Vector Vv)
Vd.w=vadd(Vu.w,Vv.w):sat	HVX_Vector Q6_Vw_vadd_VwVw_sat(HVX_Vector Vu, HVX_Vector Vv)
Vd.w=vsub(Vu.w,Vv.w)	HVX_Vector Q6_Vw_vsub_VwVw(HVX_Vector Vu, HVX_Vector Vv)
Vd.w=vsub(Vu.w,Vv.w):sat	HVX_Vector Q6_Vw_vsub_VwVw_sat(HVX_Vector Vu, HVX_Vector Vv)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse		u5								d5						
0	0	0	1	1	1	0	0	0	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.w=vadd(Vu.w,Vv.w)
0	0	0	1	1	1	0	0	0	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.ub=vadd(Vu.ub,Vv.ub):sat
0	0	0	1	1	1	0	0	0	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.uh=vadd(Vu.uh,Vv.uh):sat
0	0	0	1	1	1	0	0	0	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.h=vadd(Vu.h,Vv.h):sat
0	0	0	1	1	1	0	0	0	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.w=vadd(Vu.w,Vv.w):sat
0	0	0	1	1	1	0	0	0	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.b=vsub(Vu.b,Vv.b)
0	0	0	1	1	1	0	0	0	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.h=vsub(Vu.h,Vv.h)
0	0	0	1	1	1	0	0	0	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.w=vsub(Vu.w,Vv.w)
0	0	0	1	1	1	0	0	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.ub=vsub(Vu.ub,Vv.ub):sat
0	0	0	1	1	1	0	0	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.uh=vsub(Vu.uh,Vv.uh):sat
0	0	0	1	1	1	0	0	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.h=vsub(Vu.h,Vv.h):sat
0	0	0	1	1	1	0	0	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.w=vsub(Vu.w,Vv.w):sat
0	0	0	1	1	1	1	0	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.ub=vadd(Vu.ub,Vv.b):sat
0	0	0	1	1	1	1	0	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.ub=vsub(Vu.ub,Vv.b):sat
0	0	0	1	1	1	1	1	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.b=vadd(Vu.b,Vv.b):sat
0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.b=vsub(Vu.b,Vv.b):sat
0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.uw=vadd(Vu.uw,Vv.uw):sat
0	0	0	1	1	1	1	1	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.b=vadd(Vu.b,Vv.b)
0	0	0	1	1	1	1	1	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.h=vadd(Vu.h,Vv.h)
0	0	0	1	1	1	1	1	1	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.uw=vsub(Vu.uw,Vv.uw):sat

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Arithmetic with carry bit

Perform simple arithmetic operations, add and subtract, between the word elements of the two vectors Vu and Vv and a carry-out bit.

Optionally saturate for word.

Syntax	Behavior
<code>Vd.w, Qe4=vadd(Vu.w, Vv.w) :carry</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i] = Vu.w[i]+Vv.w[i]; QeV[4*i+4-1:4*i] = -carry_from(Vu.w[i], Vv.w[i], 0); }</pre>
<code>Vd.w, Qe4=vsub(Vu.w, Vv.w) :carry</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i] = Vu.w[i]+~Vv.w[i]+1; QeV[4*i+4-1:4*i] = -carry_from(Vu.w[i], ~Vv.w[i], 1); }</pre>
<code>Vd.w=vadd(Vu.w, Vv.w, Qs4) :carry :sat</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i] = sat₃₂(Vu.w[i]+Vv.w[i]+QsV[i*4]); }</pre>
<code>Vd.w=vadd(Vu.w, Vv.w, Qx4) :carry</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i] = Vu.w[i]+Vv.w[i]+QxV[i*4]; QxV[4*i+4-1:4*i] = -carry_from(Vu.w[i], Vv.w[i], QxV[i*4]); }</pre>
<code>Vd.w=vsub(Vu.w, Vv.w, Qx4) :carry</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i] = Vu.w[i]+~Vv.w[i]+QxV[i*4]; QxV[4*i+4-1:4*i] = -carry_from(Vu.w[i], ~Vv.w[i], QxV[i*4]); }</pre>

Class: XTYPE (slots 2,3)

Notes

- This instruction can use any HVX resource.

Intrinsics

<code>Vd.w=vadd(Vu.w, Vv.w, Qs4) :carry:sat</code>	<pre>HVX_Vector Q6_Vw_vadd_VwVwQ_carry_sat(HVX_Vector Vu, HVX_Vector Vv, HVX_VectorPred Qs)</pre>
<code>Vd.w=vadd(Vu.w, Vv.w, Qx4) :carry</code>	<pre>HVX_Vector Q6_Vw_vadd_VwVwQ_carry(HVX_Vector Vu, HVX_Vector Vv, HVX_VectorPred* Qp)</pre>
<code>Vd.w=vsub(Vu.w, Vv.w, Qx4) :carry</code>	<pre>HVX_Vector Q6_Vw_vsub_VwVwQ_carry(HVX_Vector Vu, HVX_Vector Vv, HVX_VectorPred* Qp)</pre>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse		u5							x2		d5					
0	0	0	1	1	1	0	0	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	x	x	d	d	d	d	d	Vd.w=vadd(Vu.w,Vv.w,Qx4):carry
0	0	0	1	1	1	0	0	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	x	x	d	d	d	d	d	Vd.w=vsub(Vu.w,Vv.w,Qx4):carry
ICLASS																Parse		u5							s2		d5					
0	0	0	1	1	1	0	1	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	s	s	d	d	d	d	d	Vd.w=vadd(Vu.w,Vv.w,Qs4):carry:sat
ICLASS																Parse		u5							e2		d5					
0	0	0	1	1	1	0	1	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	e	e	d	d	d	d	d	Vd.w,Qe4=vadd(Vu.w,Vv.w):carry
0	0	0	1	1	1	0	1	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	e	e	d	d	d	d	d	Vd.w,Qe4=vsub(Vu.w,Vv.w):carry

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
e2	Field to encode register e
s2	Field to encode register s
u5	Field to encode register u
x2	Field to encode register x
Maj	Major opcode
Min	Minor opcode
RegType	Register type

Logical operations

Perform bitwise logical operations (AND, OR, XOR) between the two vector registers. In the case of VNOT, invert the input register.

Syntax	Behavior
<code>Vd=vand(Vu,Vv)</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = Vu.uh[i] & Vv.h[i]; }</pre>
<code>Vd=vnot(Vu)</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = ~Vu.uh[i]; }</pre>
<code>Vd=vor(Vu,Vv)</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = Vu.uh[i] Vv.h[i]; }</pre>
<code>Vd=vxor(Vu,Vv)</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = Vu.uh[i] ^ Vv.h[i]; }</pre>

Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction can use any HVX resource.

Intrinsics

<code>Vd=vand(Vu,Vv)</code>	<code>HVX_Vector Q6_V_vand_VV(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd=vnot(Vu)</code>	<code>HVX_Vector Q6_V_vnot_V(HVX_Vector Vu)</code>
<code>Vd=vor(Vu,Vv)</code>	<code>HVX_Vector Q6_V_vor_VV(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd=vxor(Vu,Vv)</code>	<code>HVX_Vector Q6_V_vxor_VV(HVX_Vector Vu, HVX_Vector Vv)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse				u5							d5					
0	0	0	1	1	1	0	0	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd=vand(Vu,Vv)
0	0	0	1	1	1	0	0	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd=vor(Vu,Vv)
0	0	0	1	1	1	0	0	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd=vxor(Vu,Vv)
0	0	0	1	1	1	1	0	-	-	0	-	-	-	0	0	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd=vnot(Vu)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u

Copy

Copy a single input vector register to a new output vector register.

Using a scalar predicate, conditionally copy a single vector register to a destination vector register, or conditionally combine two input vectors into a destination vector register pair. A scalar predicate guards the entire operation. If the scalar predicate is true, the operation is performed. Otherwise the instruction is treated as a NOP.

Syntax	Behavior
Vd=Vu	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i]=Vu.w[i] ; }</pre>
if (!Ps) Vd=Vu	<pre>if (!Ps[0]) { for (i = 0; i < VELEM(8); i++) { Vd.ub[i] = Vu.ub[i]; } } else { NOP; }</pre>
if (Ps) Vd=Vu	<pre>if (Ps[0]) { for (i = 0; i < VELEM(8); i++) { Vd.ub[i] = Vu.ub[i]; } } else { NOP; }</pre>

Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction can use any HVX resource.

Intrinsics

Vd=Vu

HVX_Vector Q6_V_equals_V(HVX_Vector Vu)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse		u5							s2		d5					
0	0	0	1	1	0	1	0	0	0	0	-	-	-	-	-	P	P	-	u	u	u	u	u	-	s	s	d	d	d	d	d	if (Ps) Vd=Vu
0	0	0	1	1	0	1	0	0	0	1	-	-	-	-	-	P	P	-	u	u	u	u	u	-	s	s	d	d	d	d	d	if (IPs) Vd=Vu
ICLASS																Parse		u5									d5					
0	0	0	1	1	1	1	0	-	-	0	-	-	0	1	1	P	P	1	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd=Vu

Field name		Description
ICLASS		Instruction class
Parse		Packet loop parse bits
d5		Field to encode register d
s2		Field to encode register s
u5		Field to encode register u

Temporary assignment

Copy an input vector register(s) to a temporary vector register (pair) that is immediately used within the current packet.

Syntax	Behavior
<code>Vd.tmp=Vu</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i]=Vu.w[i]; }</pre>
<code>Vdd.tmp=vcombine(Vu,Vv)</code>	<pre>for (i = 0; i < VELEM(8); i++) { Vdd.v[0].ub[i] = Vv.ub[i]; Vdd.v[1].ub[i] = Vu.ub[i]; }</pre>

Class: COPROC_VX (slots 0,1,2,3)

Encoding

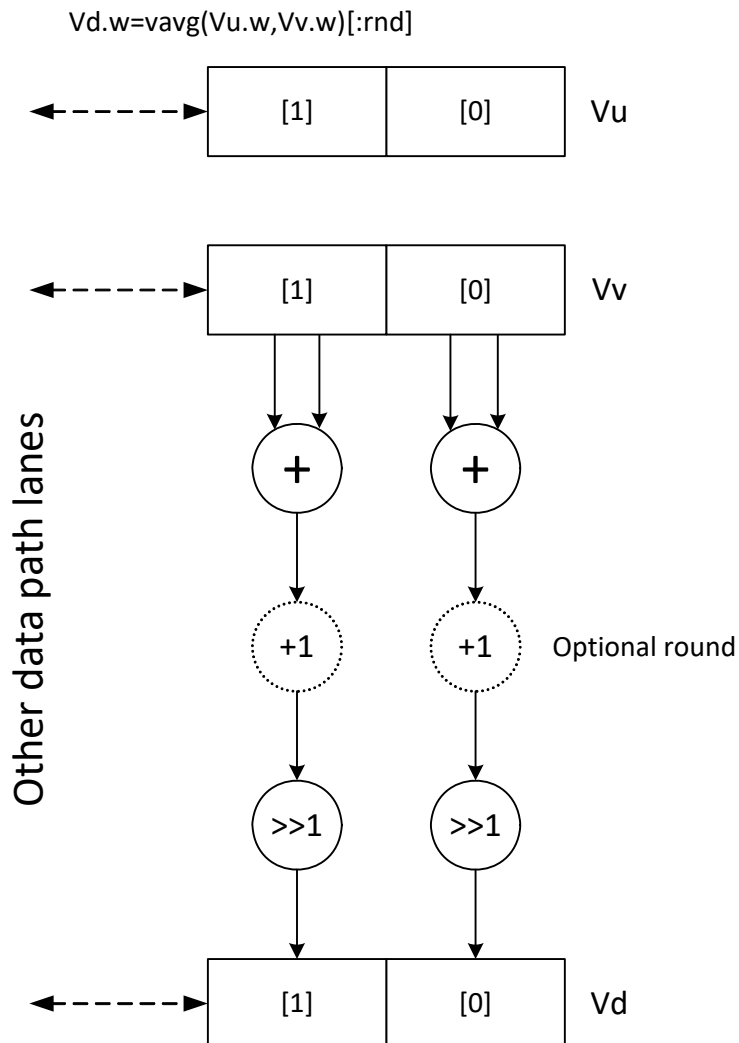
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse		u5								d5						
0	0	0	1	1	1	1	0	-	-	0	-	-	-	0	1	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.tmp=Vu
0	0	0	1	1	1	1	0	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vdd.tmp=vcombine(Vu,Vv)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Average

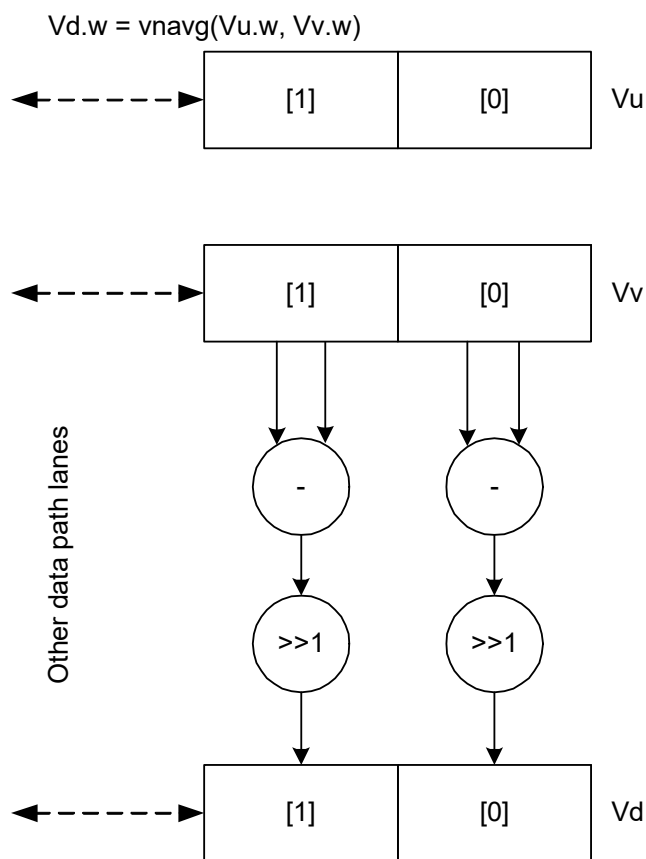
Add the elements of V_u to the respective elements of V_v , and shift the results right by 1 bit. The intermediate precision of the sum is larger than the input data precision. Optionally, a rounding constant 0x1 is added before shifting.

Supports unsigned byte, signed and unsigned halfword, and signed word. The operation is replicated to fill the implemented data path width.



Subtract the elements of V_u from the respective elements of V_v , and shift the results right by 1 bit. The intermediate precision of the sum is larger than the input data precision. Saturate the data to the required precision.

Supports unsigned byte, halfword, and word. The operation is replicated to fill the implemented data path width.



Syntax	Behavior
$Vd.b = vavg(Vu.b, Vv.b) [:rnd]$	<pre>for (i = 0; i < VELEM(8); i++) { Vd.b[i] = (Vu.b[i] + Vv.b[i] + 1) / 2; }</pre>
$Vd.b = vnavg(Vu.b, Vv.b)$	<pre>for (i = 0; i < VELEM(8); i++) { Vd.b[i] = (Vu.b[i] - Vv.b[i]) / 2 ; }</pre>
$Vd.b = vnavg(Vu.ub, Vv.ub)$	<pre>for (i = 0; i < VELEM(8); i++) { Vd.b[i] = (Vu.ub[i] - Vv.ub[i]) / 2 ; }</pre>
$Vd.h = vavg(Vu.h, Vv.h) [:rnd]$	<pre>for (i = 0; i < VELEM(16); i++) { Vd.h[i] = (Vu.h[i] + Vv.h[i] + 1) / 2 ; }</pre>
$Vd.h = vnavg(Vu.h, Vv.h)$	<pre>for (i = 0; i < VELEM(16); i++) { Vd.h[i] = (Vu.h[i] - Vv.h[i]) / 2 ; }</pre>

Syntax	Behavior
<code>Vd.ub=vavg(Vu.ub,Vv.ub)[:rnd]</code>	<pre>for (i = 0; i < VELEM(8); i++) { Vd.ub[i] = (Vu.ub[i]+Vv.ub[i]+1)/2 ; }</pre>
<code>Vd.uh=vavg(Vu.uh,Vv.uh)[:rnd]</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = (Vu.uh[i]+Vv.uh[i]+1)/2 ; }</pre>
<code>Vd.uw=vavg(Vu.uw,Vv.uw)[:rnd]</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.uw[i] = (Vu.uw[i]+Vv.uw[i]+1)/2 ; }</pre>
<code>Vd.w=vavg(Vu.w,Vv.w)[:rnd]</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i] = (Vu.w[i]+Vv.w[i]+1)/2 ; }</pre>
<code>Vd.w=vnavg(Vu.w,Vv.w)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i] = (Vu.w[i]-Vv.w[i])/2 ; }</pre>

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction can use any HVX resource.

Intrinsics

<code>Vd.b=vavg(Vu.b,Vv.b)</code>	<code>HVX_Vector Q6_Vb_vavg_VbVb(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.b=vavg(Vu.b,Vv.b):rnd</code>	<code>HVX_Vector Q6_Vb_vavg_VbVb_rnd(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.b=vnavg(Vu.b,Vv.b)</code>	<code>HVX_Vector Q6_Vb_vnavg_VbVb(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.b=vnavg(Vu.ub,Vv.ub)</code>	<code>HVX_Vector Q6_Vb_vnavg_VubVub(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.h=vavg(Vu.h,Vv.h)</code>	<code>HVX_Vector Q6_Vh_vavg_VhVh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.h=vavg(Vu.h,Vv.h):rnd</code>	<code>HVX_Vector Q6_Vh_vavg_VhVh_rnd(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.h=vnavg(Vu.h,Vv.h)</code>	<code>HVX_Vector Q6_Vh_vnavg_VhVh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.ub=vavg(Vu.ub,Vv.ub)</code>	<code>HVX_Vector Q6_Vub_vavg_VubVub(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.ub=vavg(Vu.ub,Vv.ub):rnd</code>	<code>HVX_Vector Q6_Vub_vavg_VubVub_rnd(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.uh=vavg(Vu.uh,Vv.uh)</code>	<code>HVX_Vector Q6_Vuh_vavg_VuhVuh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.uh=vavg(Vu.uh,Vv.uh):rnd</code>	<code>HVX_Vector Q6_Vuh_vavg_VuhVuh_rnd(HVX_Vector Vu, HVX_Vector Vv)</code>

Vd.uw=vavg(Vu.uw,Vv.uw)	HVX_Vector Q6_Vuw_vavg_VuwVuW(HVX_Vector Vu, HVX_Vector Vv)
Vd.uw=vavg(Vu.uw,Vv.uw):rnd	HVX_Vector Q6_Vuw_vavg_VuwVuW_rnd(HVX_Vector Vu, HVX_Vector Vv)
Vd.w=vavg(Vu.w,Vv.w)	HVX_Vector Q6_Vw_vavg_VwVw(HVX_Vector Vu, HVX_Vector Vv)
Vd.w=vavg(Vu.w,Vv.w):rnd	HVX_Vector Q6_Vw_vavg_VwVw_rnd(HVX_Vector Vu, HVX_Vector Vv)
Vd.w=vnavg(Vu.w,Vv.w)	HVX_Vector Q6_Vw_vnavg_VwVw(HVX_Vector Vu, HVX_Vector Vv)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
ICLASS																Parse				u5								d5					
0	0	0	1	1	1	0	0	1	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.ub=vavg(Vu.ub,Vv.ub)	
0	0	0	1	1	1	0	0	1	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.uh=vavg(Vu.uh,Vv.uh)	
0	0	0	1	1	1	0	0	1	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.h=vavg(Vu.h,Vv.h)	
0	0	0	1	1	1	0	0	1	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.w=vavg(Vu.w,Vv.w)	
0	0	0	1	1	1	0	0	1	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.b=vnavg(Vu.ub,Vv.ub)	
0	0	0	1	1	1	0	0	1	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.h=vnavg(Vu.h,Vv.h)	
0	0	0	1	1	1	0	0	1	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.w=vnavg(Vu.w,Vv.w)	
0	0	0	1	1	1	0	0	1	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.ub=vavg(Vu.ub,Vv.ub):rnd	
0	0	0	1	1	1	0	0	1	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.uh=vavg(Vu.uh,Vv.uh):rnd	
0	0	0	1	1	1	0	0	1	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.h=vavg(Vu.h,Vv.h):rnd	
0	0	0	1	1	1	0	0	1	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.w=vavg(Vu.w,Vv.w):rnd	
0	0	0	1	1	1	1	1	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.uw=vavg(Vu.uw,Vv.uw)	
0	0	0	1	1	1	1	1	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.uw=vavg(Vu.uw,Vv.uw):rnd	
0	0	0	1	1	1	1	1	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.b=vavg(Vu.b,Vv.b)	
0	0	0	1	1	1	1	1	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.b=vavg(Vu.b,Vv.b):rnd	
0	0	0	1	1	1	1	1	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.b=vnavg(Vu.b,Vv.b)	

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Compare vectors

Perform compares between the two vector register inputs Vu and Vv. Depending on the element size, an appropriate number of bits are written into the vector predicate register Qd for each pair of elements.

Two types of compare are supported: equal (.eq) and greater than (.gt)

Supports comparison of word, signed and unsigned halfword, signed and unsigned byte.

For each element comparison, the respective number of bits in the destination register are: bytes one bit, halfwords two bits, and words four bits.

Optionally supports XOR(^) with the destination, AND(&) with the destination, and OR(|) with the destination.

Syntax	Behavior
<code>Qd4=vcmp.eq(Vu.b,Vv.b)</code>	<pre>for(i = 0; i < VWIDTH; i += 1) { QdV[i+1-1:i] = ((Vu.b[i/1] == Vv.b[i/1]) ? 0x1 : 0); }</pre>
<code>Qd4=vcmp.eq(Vu.h,Vv.h)</code>	<pre>for(i = 0; i < VWIDTH; i += 2) { QdV[i+2-1:i] = ((Vu.h[i/2] == Vv.h[i/2]) ? 0x3 : 0); }</pre>
<code>Qd4=vcmp.eq(Vu.ub,Vv.ub)</code>	Assembler mapped to: <code>"Qd4=vcmp.eq(Vu." "b" ",Vv." "b" ")"</code>
<code>Qd4=vcmp.eq(Vu.uh,Vv.uh)</code>	Assembler mapped to: <code>"Qd4=vcmp.eq(Vu." "h" ",Vv." "h" ")"</code>
<code>Qd4=vcmp.eq(Vu.uw,Vv.uw)</code>	Assembler mapped to: <code>"Qd4=vcmp.eq(Vu." "w" ",Vv." "w" ")"</code>
<code>Qd4=vcmp.eq(Vu.w,Vv.w)</code>	<pre>for(i = 0; i < VWIDTH; i += 4) { QdV[i+4-1:i] = ((Vu.w[i/4] == Vv.w[i/4]) ? 0xF : 0); }</pre>
<code>Qd4=vcmp.gt(Vu.b,Vv.b)</code>	<pre>for(i = 0; i < VWIDTH; i += 1) { QdV[i+1-1:i] = ((Vu.b[i/1] > Vv.b[i/1]) ? 0x1 : 0); }</pre>
<code>Qd4=vcmp.gt(Vu.bf,Vv.bf)</code>	<pre>; for(i = 0; i < VWIDTH; i += 2) { VAL = (Vu.bf[i/2] > Vv.bf[i/2]) ? 0x3 : 0 ; QdV[i+2-1:i] = VAL; }</pre>
<code>Qd4=vcmp.gt(Vu.h,Vv.h)</code>	<pre>for(i = 0; i < VWIDTH; i += 2) { QdV[i+2-1:i] = ((Vu.h[i/2] > Vv.h[i/2]) ? 0x3 : 0); }</pre>

Syntax	Behavior
<code>Qd4=vcmp.gt(Vu.hf,Vv.hf)</code>	<pre>for(i = 0; i < VWIDTH; i += 2) { VAL = (Vu.hf[i/2] > Vv.hf[i/2]) ? 0x3 : 0 ; QdV[i+2-1:i] = VAL; }</pre>
<code>Qd4=vcmp.gt(Vu.sf,Vv.sf)</code>	<pre>for(i = 0; i < VWIDTH; i += 4) { VAL = (Vu.sf[i/4] > Vv.sf[i/4]) ? 0xF : 0 ; QdV[i+4-1:i] = VAL; }</pre>
<code>Qd4=vcmp.gt(Vu.ub,Vv.ub)</code>	<pre>for(i = 0; i < VWIDTH; i += 1) { QdV[i+1-1:i] = ((Vu.ub[i/1] > Vv.ub[i/1]) ? 0x1 : 0); }</pre>
<code>Qd4=vcmp.gt(Vu.uh,Vv.uh)</code>	<pre>for(i = 0; i < VWIDTH; i += 2) { QdV[i+2-1:i] = ((Vu.uh[i/2] > Vv.uh[i/2]) ? 0x3 : 0); }</pre>
<code>Qd4=vcmp.gt(Vu.uw,Vv.uw)</code>	<pre>for(i = 0; i < VWIDTH; i += 4) { QdV[i+4-1:i] = ((Vu.uw[i/4] > Vv.uw[i/4]) ? 0xF : 0); }</pre>
<code>Qd4=vcmp.gt(Vu.w,Vv.w)</code>	<pre>for(i = 0; i < VWIDTH; i += 4) { QdV[i+4-1:i] = ((Vu.w[i/4] > Vv.w[i/4]) ? 0xF : 0); }</pre>
<code>Qx4[& =vcmp.eq(Vu.b,Vv.b)</code>	<pre>for(i = 0; i < VWIDTH; i += 1) { QxV[i+1-1:i] = QxV[i+1-1:i] [&] ((Vu.b[i/1] == Vv.b[i/1]) ? 0x1 : 0); }</pre>
<code>Qx4[& =vcmp.eq(Vu.h,Vv.h)</code>	<pre>for(i = 0; i < VWIDTH; i += 2) { QxV[i+2-1:i] = QxV[i+2-1:i] [&] ((Vu.h[i/2] == Vv.h[i/2]) ? 0x3 : 0); }</pre>
<code>Qx4[& =vcmp.eq(Vu.ub,Vv.ub)</code>	Assembler mapped to: <code>"Qx4[=vcmp.eq(Vu." "b" ",Vv." "b" ")"</code>
<code>Qx4[& =vcmp.eq(Vu.uh,Vv.uh)</code>	Assembler mapped to: <code>"Qx4[=vcmp.eq(Vu." "h" ",Vv." "h" ")"</code>
<code>Qx4[& =vcmp.eq(Vu.uw,Vv.uw)</code>	Assembler mapped to: <code>"Qx4[=vcmp.eq(Vu." "w" ",Vv." "w" ")"</code>
<code>Qx4[& =vcmp.eq(Vu.w,Vv.w)</code>	<pre>for(i = 0; i < VWIDTH; i += 4) { QxV[i+4-1:i] = QxV[i+4-1:i] [&] ((Vu.w[i/4] == Vv.w[i/4]) ? 0xF : 0); }</pre>
<code>Qx4[& =vcmp.gt(Vu.b,Vv.b)</code>	<pre>for(i = 0; i < VWIDTH; i += 1) { QxV[i+1-1:i] = QxV[i+1-1:i] [&] ((Vu.b[i/1] > Vv.b[i/1]) ? 0x1 : 0); }</pre>

Syntax	Behavior
<code>Qx4[&]=vcmp.gt(Vu.bf,Vv.bf)</code>	<pre> ; for(i = 0; i < VWIDTH; i += 2) { VAL = (Vu.bf[i/2] > Vv.bf[i/2]) ? 0x3 : 0 ; QxV[i+2-1:i] = QxV[i+2-1:i] [&] VAL; } </pre>
<code>Qx4[&]=vcmp.gt(Vu.h,Vv.h)</code>	<pre> for(i = 0; i < VWIDTH; i += 2) { QxV[i+2-1:i] = QxV[i+2-1:i] [&] ((Vu.h[i/2] > Vv.h[i/2]) ? 0x3 : 0); } </pre>
<code>Qx4[&]=vcmp.gt(Vu.hf,Vv.hf)</code>	<pre> for(i = 0; i < VWIDTH; i += 2) { VAL = (Vu.hf[i/2] > Vv.hf[i/2]) ? 0x3 : 0 ; QxV[i+2-1:i] = QxV[i+2-1:i] [&] VAL; } </pre>
<code>Qx4[&]=vcmp.gt(Vu.sf,Vv.sf)</code>	<pre> for(i = 0; i < VWIDTH; i += 4) { VAL = (Vu.sf[i/4] > Vv.sf[i/4]) ? 0xF : 0 ; QxV[i+4-1:i] = QxV[i+4-1:i] [&] VAL; } </pre>
<code>Qx4[&]=vcmp.gt(Vu.ub,Vv.ub)</code>	<pre> for(i = 0; i < VWIDTH; i += 1) { QxV[i+1-1:i] = QxV[i+1-1:i] [&] ((Vu.ub[i/1] > Vv.ub[i/1]) ? 0x1 : 0); } </pre>
<code>Qx4[&]=vcmp.gt(Vu.uh,Vv.uh)</code>	<pre> for(i = 0; i < VWIDTH; i += 2) { QxV[i+2-1:i] = QxV[i+2-1:i] [&] ((Vu.uh[i/2] > Vv.uh[i/2]) ? 0x3 : 0); } </pre>
<code>Qx4[&]=vcmp.gt(Vu.uw,Vv.uw)</code>	<pre> for(i = 0; i < VWIDTH; i += 4) { QxV[i+4-1:i] = QxV[i+4-1:i] [&] ((Vu.uw[i/4] > Vv.uw[i/4]) ? 0xF : 0); } </pre>
<code>Qx4[&]=vcmp.gt(Vu.w,Vv.w)</code>	<pre> for(i = 0; i < VWIDTH; i += 4) { QxV[i+4-1:i] = QxV[i+4-1:i] [&] ((Vu.w[i/4] > Vv.w[i/4]) ? 0xF : 0); } </pre>
<code>Qx4^=vcmp.eq(Vu.b,Vv.b)</code>	<pre> for(i = 0; i < VWIDTH; i += 1) { QxV[i+1-1:i] = QxV[i+1-1:i] ^ ((Vu.b[i/1] == Vv.b[i/1]) ? 0x1 : 0); } </pre>
<code>Qx4^=vcmp.eq(Vu.h,Vv.h)</code>	<pre> for(i = 0; i < VWIDTH; i += 2) { QxV[i+2-1:i] = QxV[i+2-1:i] ^ ((Vu.h[i/2] == Vv.h[i/2]) ? 0x3 : 0); } </pre>
<code>Qx4^=vcmp.eq(Vu.ub,Vv.ub)</code>	Assembler mapped to: <code>"Qx4^=vcmp.eq(Vu." "b" ",Vv." "b" ")"</code>
<code>Qx4^=vcmp.eq(Vu.uh,Vv.uh)</code>	Assembler mapped to: <code>"Qx4^=vcmp.eq(Vu." "h" ",Vv." "h" ")"</code>
<code>Qx4^=vcmp.eq(Vu.uw,Vv.uw)</code>	Assembler mapped to: <code>"Qx4^=vcmp.eq(Vu." "w" ",Vv." "w" ")"</code>

Syntax	Behavior
<code>Qx4^=vcmp.eq(Vu.w,Vv.w)</code>	<pre>for(i = 0; i < VWIDTH; i += 4) { QxV[i+4-1:i] = QxV[i+4-1:i] ^ ((Vu.w[i/4] == Vv.w[i/4]) ? 0xF : 0); }</pre>
<code>Qx4^=vcmp.gt(Vu.b,Vv.b)</code>	<pre>for(i = 0; i < VWIDTH; i += 1) { QxV[i+1-1:i] = QxV[i+1-1:i] ^ ((Vu.b[i/1] > Vv.b[i/1]) ? 0x1 : 0); }</pre>
<code>Qx4^=vcmp.gt(Vu.bf,Vv.bf)</code>	<pre>; for(i = 0; i < VWIDTH; i += 2) { VAL = (Vu.bf[i/2] > Vv.bf[i/2]) ? 0x3 : 0 ; QxV[i+2-1:i] = QxV[i+2-1:i] ^ VAL; }</pre>
<code>Qx4^=vcmp.gt(Vu.h,Vv.h)</code>	<pre>for(i = 0; i < VWIDTH; i += 2) { QxV[i+2-1:i] = QxV[i+2-1:i] ^ ((Vu.h[i/2] > Vv.h[i/2]) ? 0x3 : 0); }</pre>
<code>Qx4^=vcmp.gt(Vu.hf,Vv.hf)</code>	<pre>for(i = 0; i < VWIDTH; i += 2) { VAL = (Vu.hf[i/2] > Vv.hf[i/2]) ? 0x3 : 0 ; QxV[i+2-1:i] = QxV[i+2-1:i] ^ VAL; }</pre>
<code>Qx4^=vcmp.gt(Vu.sf,Vv.sf)</code>	<pre>for(i = 0; i < VWIDTH; i += 4) { VAL = (Vu.sf[i/4] > Vv.sf[i/4]) ? 0xF : 0 ; QxV[i+4-1:i] = QxV[i+4-1:i] ^ VAL; }</pre>
<code>Qx4^=vcmp.gt(Vu.ub,Vv.ub)</code>	<pre>for(i = 0; i < VWIDTH; i += 1) { QxV[i+1-1:i] = QxV[i+1-1:i] ^ ((Vu.ub[i/1] > Vv.ub[i/1]) ? 0x1 : 0); }</pre>
<code>Qx4^=vcmp.gt(Vu.uh,Vv.uh)</code>	<pre>for(i = 0; i < VWIDTH; i += 2) { QxV[i+2-1:i] = QxV[i+2-1:i] ^ ((Vu.uh[i/2] > Vv.uh[i/2]) ? 0x3 : 0); }</pre>
<code>Qx4^=vcmp.gt(Vu.uw,Vv.uw)</code>	<pre>for(i = 0; i < VWIDTH; i += 4) { QxV[i+4-1:i] = QxV[i+4-1:i] ^ ((Vu.uw[i/4] > Vv.uw[i/4]) ? 0xF : 0); }</pre>
<code>Qx4^=vcmp.gt(Vu.w,Vv.w)</code>	<pre>for(i = 0; i < VWIDTH; i += 4) { QxV[i+4-1:i] = QxV[i+4-1:i] ^ ((Vu.w[i/4] > Vv.w[i/4]) ? 0xF : 0); }</pre>

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction can use any HVX resource.

Intrinsics

<code>Qd4=vcmp.eq(Vu.b,Vv.b)</code>	<code>HVX_VectorPred Q6_Q_vcmp_eq_VbVb(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qd4=vcmp.eq(Vu.h,Vv.h)</code>	<code>HVX_VectorPred Q6_Q_vcmp_eq_VhVh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qd4=vcmp.eq(Vu.w,Vv.w)</code>	<code>HVX_VectorPred Q6_Q_vcmp_eq_VwVw(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qd4=vcmp.gt(Vu.b,Vv.b)</code>	<code>HVX_VectorPred Q6_Q_vcmp_gt_VbVb(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qd4=vcmp.gt(Vu.bf,Vv.bf)</code>	<code>HVX_VectorPred Q6_Q_vcmp_gt_VbfVbf(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qd4=vcmp.gt(Vu.h,Vv.h)</code>	<code>HVX_VectorPred Q6_Q_vcmp_gt_VhVh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qd4=vcmp.gt(Vu.hf,Vv.hf)</code>	<code>HVX_VectorPred Q6_Q_vcmp_gt_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qd4=vcmp.gt(Vu.sf,Vv.sf)</code>	<code>HVX_VectorPred Q6_Q_vcmp_gt_VsfVsf(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qd4=vcmp.gt(Vu.ub,Vv.ub)</code>	<code>HVX_VectorPred Q6_Q_vcmp_gt_VubVub(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qd4=vcmp.gt(Vu.uh,Vv.uh)</code>	<code>HVX_VectorPred Q6_Q_vcmp_gt_VuhVuh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qd4=vcmp.gt(Vu.uw,Vv.uw)</code>	<code>HVX_VectorPred Q6_Q_vcmp_gt_VuwVuw(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qd4=vcmp.gt(Vu.w,Vv.w)</code>	<code>HVX_VectorPred Q6_Q_vcmp_gt_VwVw(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4&=vcmp.eq(Vu.b,Vv.b)</code>	<code>HVX_VectorPred Q6_Q_vcmp_eqand_QVbVb(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4&=vcmp.eq(Vu.h,Vv.h)</code>	<code>HVX_VectorPred Q6_Q_vcmp_eqand_QVhVh(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4&=vcmp.eq(Vu.w,Vv.w)</code>	<code>HVX_VectorPred Q6_Q_vcmp_eqand_QVwVw(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4&=vcmp.gt(Vu.b,Vv.b)</code>	<code>HVX_VectorPred Q6_Q_vcmp_gtand_QVbVb(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4&=vcmp.gt(Vu.bf,Vv.bf)</code>	<code>HVX_VectorPred Q6_Q_vcmp_gtand_QVbfVbf(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4&=vcmp.gt(Vu.h,Vv.h)</code>	<code>HVX_VectorPred Q6_Q_vcmp_gtand_QVhVh(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4&=vcmp.gt(Vu.hf,Vv.hf)</code>	<code>HVX_VectorPred Q6_Q_vcmp_gtand_QVhfVhf(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4&=vcmp.gt(Vu.sf,Vv.sf)</code>	<code>HVX_VectorPred Q6_Q_vcmp_gtand_QVsfVsf(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>

<code>Qx4&=vcmp.gt (Vu.ub, Vv.ub)</code>	HVX_VectorPred <code>Q6_Q_vcmp_gtand_QVubVub (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4&=vcmp.gt (Vu.uh, Vv.uh)</code>	HVX_VectorPred <code>Q6_Q_vcmp_gtand_QVuhVuh (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4&=vcmp.gt (Vu.uw, Vv.uw)</code>	HVX_VectorPred <code>Q6_Q_vcmp_gtand_QVuwVuw (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4&=vcmp.gt (Vu.w, Vv.w)</code>	HVX_VectorPred <code>Q6_Q_vcmp_gtand_QVwVw (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4^=vcmp.eq (Vu.b, Vv.b)</code>	HVX_VectorPred <code>Q6_Q_vcmp_eqxacc_QVbVb (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4^=vcmp.eq (Vu.h, Vv.h)</code>	HVX_VectorPred <code>Q6_Q_vcmp_eqxacc_QVhVh (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4^=vcmp.eq (Vu.w, Vv.w)</code>	HVX_VectorPred <code>Q6_Q_vcmp_eqxacc_QVwVw (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4^=vcmp.gt (Vu.b, Vv.b)</code>	HVX_VectorPred <code>Q6_Q_vcmp_gtxacc_QVbVb (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4^=vcmp.gt (Vu.bf, Vv.bf)</code>	HVX_VectorPred <code>Q6_Q_vcmp_gtxacc_QVbfVbf (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4^=vcmp.gt (Vu.h, Vv.h)</code>	HVX_VectorPred <code>Q6_Q_vcmp_gtxacc_QVhVh (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4^=vcmp.gt (Vu.hf, Vv.hf)</code>	HVX_VectorPred <code>Q6_Q_vcmp_gtxacc_QVhfVhf (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4^=vcmp.gt (Vu.sf, Vv.sf)</code>	HVX_VectorPred <code>Q6_Q_vcmp_gtxacc_QVsfVsf (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4^=vcmp.gt (Vu.ub, Vv.ub)</code>	HVX_VectorPred <code>Q6_Q_vcmp_gtxacc_QVubVub (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4^=vcmp.gt (Vu.uh, Vv.uh)</code>	HVX_VectorPred <code>Q6_Q_vcmp_gtxacc_QVuhVuh (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4^=vcmp.gt (Vu.uw, Vv.uw)</code>	HVX_VectorPred <code>Q6_Q_vcmp_gtxacc_QVuwVuw (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4^=vcmp.gt (Vu.w, Vv.w)</code>	HVX_VectorPred <code>Q6_Q_vcmp_gtxacc_QVwVw (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4 =vcmp.eq (Vu.b, Vv.b)</code>	HVX_VectorPred <code>Q6_Q_vcmp_eqor_QVbVb (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4 =vcmp.eq (Vu.h, Vv.h)</code>	HVX_VectorPred <code>Q6_Q_vcmp_eqor_QVhVh (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Qx4 =vcmp.eq (Vu.w, Vv.w)</code>	HVX_VectorPred <code>Q6_Q_vcmp_eqor_QVwVw (HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)</code>

$Qx4 = \text{vcmp.gt}(Vu.b, Vv.b)$	$\text{HVX_VectorPred } Q6_Q_vcmp_gtor_QVbVb(\text{HVX_VectorPred } Qx, \text{HVX_Vector } Vu, \text{HVX_Vector } Vv)$
$Qx4 = \text{vcmp.gt}(Vu.bf, Vv.bf)$	$\text{HVX_VectorPred } Q6_Q_vcmp_gtor_QVbfVbf(\text{HVX_VectorPred } Qx, \text{HVX_Vector } Vu, \text{HVX_Vector } Vv)$
$Qx4 = \text{vcmp.gt}(Vu.h, Vv.h)$	$\text{HVX_VectorPred } Q6_Q_vcmp_gtor_QVhVh(\text{HVX_VectorPred } Qx, \text{HVX_Vector } Vu, \text{HVX_Vector } Vv)$
$Qx4 = \text{vcmp.gt}(Vu.hf, Vv.hf)$	$\text{HVX_VectorPred } Q6_Q_vcmp_gtor_QVhfVhf(\text{HVX_VectorPred } Qx, \text{HVX_Vector } Vu, \text{HVX_Vector } Vv)$
$Qx4 = \text{vcmp.gt}(Vu.sf, Vv.sf)$	$\text{HVX_VectorPred } Q6_Q_vcmp_gtor_QVsfVsf(\text{HVX_VectorPred } Qx, \text{HVX_Vector } Vu, \text{HVX_Vector } Vv)$
$Qx4 = \text{vcmp.gt}(Vu.ub, Vv.ub)$	$\text{HVX_VectorPred } Q6_Q_vcmp_gtor_QVubVub(\text{HVX_VectorPred } Qx, \text{HVX_Vector } Vu, \text{HVX_Vector } Vv)$
$Qx4 = \text{vcmp.gt}(Vu.uh, Vv.uh)$	$\text{HVX_VectorPred } Q6_Q_vcmp_gtor_QVuhVuh(\text{HVX_VectorPred } Qx, \text{HVX_Vector } Vu, \text{HVX_Vector } Vv)$
$Qx4 = \text{vcmp.gt}(Vu.uw, Vv.uw)$	$\text{HVX_VectorPred } Q6_Q_vcmp_gtor_QVuwVuw(\text{HVX_VectorPred } Qx, \text{HVX_Vector } Vu, \text{HVX_Vector } Vv)$
$Qx4 = \text{vcmp.gt}(Vu.w, Vv.w)$	$\text{HVX_VectorPred } Q6_Q_vcmp_gtor_QVwVw(\text{HVX_VectorPred } Qx, \text{HVX_Vector } Vu, \text{HVX_Vector } Vv)$

Encoding

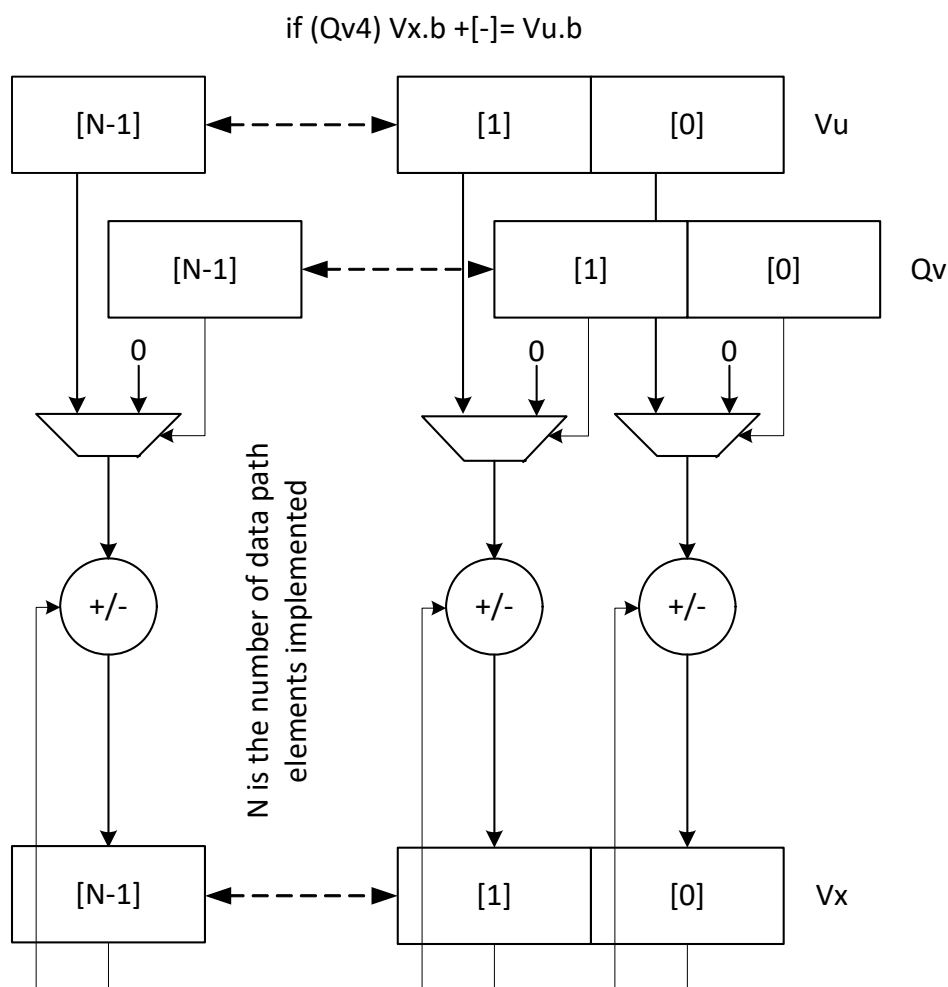
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse				u5										x2		
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	0	0	0	x	x	Qx4&=vcmp.eq(Vu.b,Vv.b)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	0	0	1	x	x	Qx4&=vcmp.eq(Vu.h,Vv.h)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	0	1	0	x	x	Qx4&=vcmp.eq(Vu.w,Vv.w)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	1	0	0	x	x	Qx4&=vcmp.gt(Vu.b,Vv.b)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	1	0	1	x	x	Qx4&=vcmp.gt(Vu.h,Vv.h)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	1	1	0	x	x	Qx4&=vcmp.gt(Vu.w,Vv.w)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	0	0	0	x	x	Qx4&=vcmp.gt(Vu.ub,Vv.ub)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	0	0	1	x	x	Qx4&=vcmp.gt(Vu.uh,Vv.uh)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	0	1	0	x	x	Qx4&=vcmp.gt(Vu.uw,Vv.uw)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	1	0	0	x	x	Qx4 =vcmp.gt(Vu.sf,Vv.sf)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	1	0	1	x	x	Qx4 =vcmp.gt(Vu.hf,Vv.hf)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	1	1	0	x	x	Qx4 =vcmp.gt(Vu.bf,Vv.bf)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	0	0	0	0	x	x	Qx4 =vcmp.eq(Vu.b,Vv.b)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	0	0	0	1	x	x	Qx4 =vcmp.eq(Vu.h,Vv.h)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	0	0	1	0	x	x	Qx4 =vcmp.eq(Vu.w,Vv.w)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	0	1	0	0	x	x	Qx4 =vcmp.gt(Vu.b,Vv.b)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	0	1	0	1	x	x	Qx4 =vcmp.gt(Vu.h,Vv.h)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	0	1	1	0	x	x	Qx4 =vcmp.gt(Vu.w,Vv.w)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	0	0	0	x	x	Qx4 =vcmp.gt(Vu.ub,Vv.ub)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	0	1	0	x	x	Qx4 =vcmp.gt(Vu.uh,Vv.uh)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	0	1	0	x	x	Qx4 =vcmp.gt(Vu.uw,Vv.uw)
ICLASS																Parse				u5										d2		

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	1	0	0	d	d	Qd4=vcmp.gt(Vu.sf,Vv.sf)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	1	0	1	d	d	Qd4=vcmp.gt(Vu.hf,Vv.hf)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	1	1	0	d	d	Qd4=vcmp.gt(Vu.bf,Vv.bf)
ICLASS																Parse		u5										x2				
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	0	0	0	x	x	Qx4^=vcmp.eq(Vu.b,Vv.b)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	0	0	1	x	x	Qx4^=vcmp.eq(Vu.h,Vv.h)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	0	1	0	x	x	Qx4^=vcmp.eq(Vu.w,Vv.w)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	1	0	0	x	x	Qx4^=vcmp.gt(Vu.b,Vv.b)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	1	0	1	x	x	Qx4^=vcmp.gt(Vu.h,Vv.h)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	1	1	0	x	x	Qx4^=vcmp.gt(Vu.w,Vv.w)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	1	0	0	0	x	x	Qx4^=vcmp.gt(Vu.ub,Vv.ub)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	1	0	0	1	x	x	Qx4^=vcmp.gt(Vu.uh,Vv.uh)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	1	0	1	0	x	x	Qx4^=vcmp.gt(Vu.uw,Vv.uw)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	0	0	1	0	x	x	Qx4&=vcmp.gt(Vu.sf,Vv.sf)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	0	0	1	1	x	x	Qx4&=vcmp.gt(Vu.hf,Vv.hf)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	0	1	0	0	x	x	Qx4&=vcmp.gt(Vu.bf,Vv.bf)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	0	1	0	x	x	Qx4^=vcmp.gt(Vu.sf,Vv.sf)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	0	1	1	x	x	Qx4^=vcmp.gt(Vu.hf,Vv.hf)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	1	0	0	x	x	Qx4^=vcmp.gt(Vu.bf,Vv.bf)
ICLASS																Parse		u5										d2				
0	0	0	1	1	1	1	1	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	0	0	0	d	d	Qd4=vcmp.eq(Vu.b,Vv.b)
0	0	0	1	1	1	1	1	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	0	0	1	d	d	Qd4=vcmp.eq(Vu.h,Vv.h)
0	0	0	1	1	1	1	1	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	0	1	0	d	d	Qd4=vcmp.eq(Vu.w,Vv.w)
0	0	0	1	1	1	1	1	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	1	0	0	d	d	Qd4=vcmp.gt(Vu.b,Vv.b)
0	0	0	1	1	1	1	1	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	1	0	1	d	d	Qd4=vcmp.gt(Vu.h,Vv.h)
0	0	0	1	1	1	1	1	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	1	1	0	d	d	Qd4=vcmp.gt(Vu.w,Vv.w)
0	0	0	1	1	1	1	1	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	0	0	0	d	d	Qd4=vcmp.gt(Vu.ub,Vv.ub)
0	0	0	1	1	1	1	1	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	0	0	1	d	d	Qd4=vcmp.gt(Vu.uh,Vv.uh)
0	0	0	1	1	1	1	1	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	0	1	0	d	d	Qd4=vcmp.gt(Vu.uw,Vv.uw)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d2	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v
x2	Field to encode register x

Conditional accumulate

Conditionally add or subtract a value to the destination register. If the corresponding bits are set in the vector predicate register, the elements in Vu are added to or subtracted from the corresponding elements in Vx. Supports byte, halfword, and word. No saturation is performed on the result.



Syntax

```
if (!Qv4) Vx.b[+-]=Vu.b
```

```
if (!Qv4) Vx.h[+-]=Vu.h
```

```
if (!Qv4) Vx.w[+-]=Vu.w
```

Behavior

```
for (i = 0; i < VELEM(8); i[+-][+-]) {
    Vx.ub[i]=QvV.i ? Vx.ub[i] : Vx.ub[i][+-]Vu.ub[i] ;
}
```

```
for (i = 0; i < VELEM(16); i[+-][+-]) {
    Vx.h[i]=select_bytes(QvV,i,Vx.h[i],Vx.h[i][+-]
    Vu.h[i]);
}
```

```
for (i = 0; i < VELEM(32); i[+-][+-]) {
    Vx.w[i]=select_bytes(QvV,i,Vx.w[i],Vx.w[i][+-]
    Vu.w[i]);
}
```

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction can use any HVX resource.

Intrinsics

if (!Qv4) Vx.b+=Vu.b	HVX_Vector Q6_Vb_condacc_QnVbVb(HVX_VectorPred Qv, HVX_Vector Vx, HVX_Vector Vu)
if (!Qv4) Vx.b-=Vu.b	HVX_Vector Q6_Vb_condnac_QnVbVb(HVX_VectorPred Qv, HVX_Vector Vx, HVX_Vector Vu)
if (!Qv4) Vx.h+=Vu.h	HVX_Vector Q6_Vh_condacc_QnVhVh(HVX_VectorPred Qv, HVX_Vector Vx, HVX_Vector Vu)
if (!Qv4) Vx.h-=Vu.h	HVX_Vector Q6_Vh_condnac_QnVhVh(HVX_VectorPred Qv, HVX_Vector Vx, HVX_Vector Vu)
if (!Qv4) Vx.w+=Vu.w	HVX_Vector Q6_Vw_condacc_QnVwVw(HVX_VectorPred Qv, HVX_Vector Vx, HVX_Vector Vu)
if (!Qv4) Vx.w-=Vu.w	HVX_Vector Q6_Vw_condnac_QnVwVw(HVX_VectorPred Qv, HVX_Vector Vx, HVX_Vector Vu)
if (Qv4) Vx.b+=Vu.b	HVX_Vector Q6_Vb_condacc_QVbVb(HVX_VectorPred Qv, HVX_Vector Vx, HVX_Vector Vu)
if (Qv4) Vx.b-=Vu.b	HVX_Vector Q6_Vb_condnac_QVbVb(HVX_VectorPred Qv, HVX_Vector Vx, HVX_Vector Vu)
if (Qv4) Vx.h+=Vu.h	HVX_Vector Q6_Vh_condacc_QVhVh(HVX_VectorPred Qv, HVX_Vector Vx, HVX_Vector Vu)
if (Qv4) Vx.h-=Vu.h	HVX_Vector Q6_Vh_condnac_QVhVh(HVX_VectorPred Qv, HVX_Vector Vx, HVX_Vector Vu)
if (Qv4) Vx.w+=Vu.w	HVX_Vector Q6_Vw_condacc_QVwVw(HVX_VectorPred Qv, HVX_Vector Vx, HVX_Vector Vu)
if (Qv4) Vx.w-=Vu.w	HVX_Vector Q6_Vw_condnac_QVwVw(HVX_VectorPred Qv, HVX_Vector Vx, HVX_Vector Vu)

Encoding

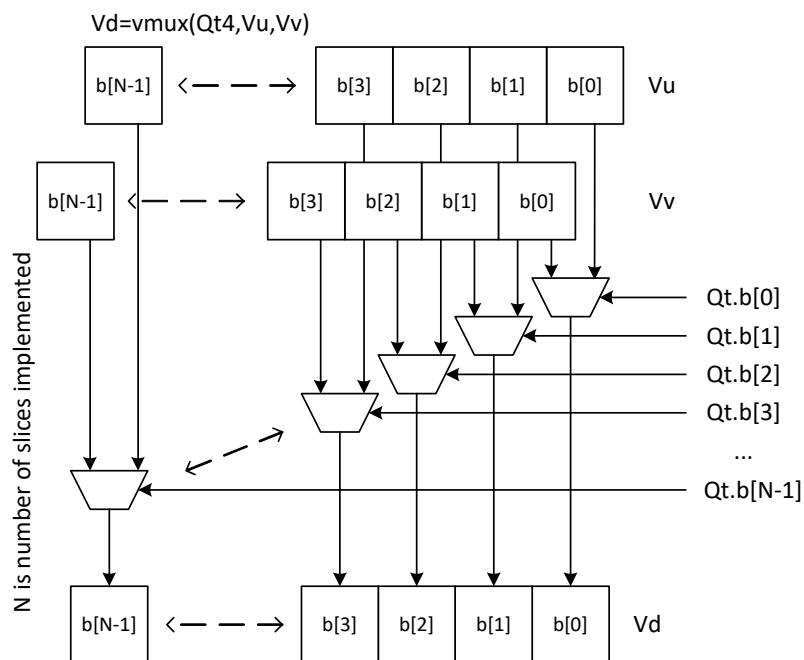
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse				u5								x5				
0	0	0	1	1	1	1	0	v	v	0	-	-	0	0	1	P	P	1	u	u	u	u	u	0	0	0	x	x	x	x	x	if (Qv4) Vx.b+=Vu.b
0	0	0	1	1	1	1	0	v	v	0	-	-	0	0	1	P	P	1	u	u	u	u	u	0	0	1	x	x	x	x	x	if (Qv4) Vx.h+=Vu.h
0	0	0	1	1	1	1	0	v	v	0	-	-	0	0	1	P	P	1	u	u	u	u	u	0	1	0	x	x	x	x	x	if (Qv4) Vx.w+=Vu.w
0	0	0	1	1	1	1	0	v	v	0	-	-	0	0	1	P	P	1	u	u	u	u	u	0	1	1	x	x	x	x	x	if (!Qv4) Vx.b+=Vu.b
0	0	0	1	1	1	1	0	v	v	0	-	-	0	0	1	P	P	1	u	u	u	u	u	1	0	0	x	x	x	x	x	if (!Qv4) Vx.h+=Vu.h
0	0	0	1	1	1	1	0	v	v	0	-	-	0	0	1	P	P	1	u	u	u	u	u	1	0	1	x	x	x	x	x	if (!Qv4) Vx.w+=Vu.w
0	0	0	1	1	1	1	0	v	v	0	-	-	0	0	1	P	P	1	u	u	u	u	u	1	1	0	x	x	x	x	x	if (Qv4) Vx.b-=Vu.b

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	1	1	1	1	0	v	v	0	-	-	0	0	1	P	P	1	u	u	u	u	u	1	1	1	x	x	x	x	x	if (Qv4) Vx.h=Vu.h
0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	0	P	P	1	u	u	u	u	u	0	0	0	x	x	x	x	x	if (Qv4) Vx.w=Vu.w
0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	0	P	P	1	u	u	u	u	u	0	0	1	x	x	x	x	x	if (!Qv4) Vx.b=Vu.b
0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	0	P	P	1	u	u	u	u	u	0	1	0	x	x	x	x	x	if (!Qv4) Vx.h=Vu.h
0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	0	P	P	1	u	u	u	u	u	0	1	1	x	x	x	x	x	if (!Qv4) Vx.w=Vu.w

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
u5	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x

Mux select

Perform a parallel if-then-else operation. Based on a predicate bit in a vector predicate register, if the bit is set, the corresponding byte from vector register Vu is placed in the destination vector register Vd. Otherwise, the corresponding byte from Vv is written. The operation works on bytes so it can handle all data sizes.



Syntax

$Vd = vmux(Qt4, Vu, Vv)$

Behavior

```
for (i = 0; i < VELEM(8); i++) {
    Vd.ub[i] = QtV[i] ? Vu.ub[i] : Vv.ub[i];
}
```

Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction can use any HVX resource.

Intrinsics

$Vd = vmux(Qt4, Vu, Vv)$ `HVX_Vector Q6_V_vmux_QVV(HVX_VectorPred Qt, HVX_Vector Vu, HVX_Vector Vv)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse		u5						t2		d5						
0	0	0	1	1	1	1	0	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	-	t	t	d	d	d	d	d	Vd=vmux(Qt4,Vu,Vv)

Field name		Description
ICLASS		Instruction class
Parse		Packet loop parse bits
d5		Field to encode register d
t2		Field to encode register t
u5		Field to encode register u

Saturation

Perform simple arithmetic operations, add and subtract, between the elements of the two vectors Vu and Vv. Supports word, halfword (signed and unsigned), and byte (signed and unsigned).

Optionally saturate for word and halfword. Always saturate for unsigned types.

Syntax	Behavior
<code>Vd.h=vsat(Vu.w,Vv.w)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i].h[0]=sat₁₆(Vv.w[i]); Vd.w[i].h[1]=sat₁₆(Vu.w[i]); }</pre>
<code>Vd.ub=vsat(Vu.h,Vv.h)</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.uh[i].b[0]=usat₈(Vv.h[i]); Vd.uh[i].b[1]=usat₈(Vu.h[i]); }</pre>
<code>Vd.uh=vsat(Vu.uw,Vv.uw)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i].h[0]=usat₁₆(Vv.uw[i]); Vd.w[i].h[1]=usat₁₆(Vu.uw[i]); }</pre>
<code>Vd.w=vsatdw(Vu.w,Vv.w)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i] = usat₃₂(Vu.w[i]:Vv.w[i]); }</pre>

Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction can use any HVX resource.

Intrinsics

<code>Vd.h=vsat(Vu.w,Vv.w)</code>	<code>HVX_Vector Q6_Vh_vsatsat_VwVw(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.ub=vsat(Vu.h,Vv.h)</code>	<code>HVX_Vector Q6_Vub_vsatsat_VhVh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.uh=vsat(Vu.uw,Vv.uw)</code>	<code>HVX_Vector Q6_Vuh_vsatsat_VuwVuw(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.w=vsatdw(Vu.w,Vv.w)</code>	<code>HVX_Vector Q6_Vw_vsatsatdw_VwVw(HVX_Vector Vu, HVX_Vector Vv)</code>

Encoding

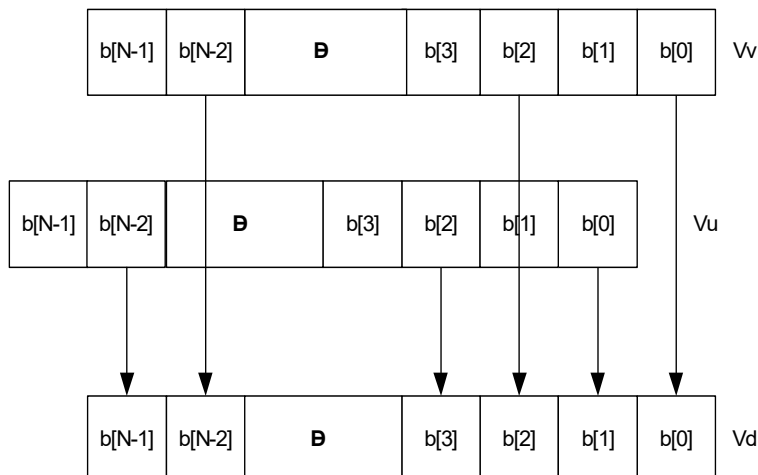
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse		u5						d5								
0	0	0	1	1	1	0	1	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.w=vsatdw(Vu.w,Vv.w)
0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.uh=vsat(Vu.uw,Vv.uw)
0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.ub=vsat(Vu.h,Vv.h)
0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.h=vsat(Vu.w,Vv.w)

Field name		Description
ICLASS		Instruction class
Parse		Packet loop parse bits
d5		Field to encode register d
u5		Field to encode register u

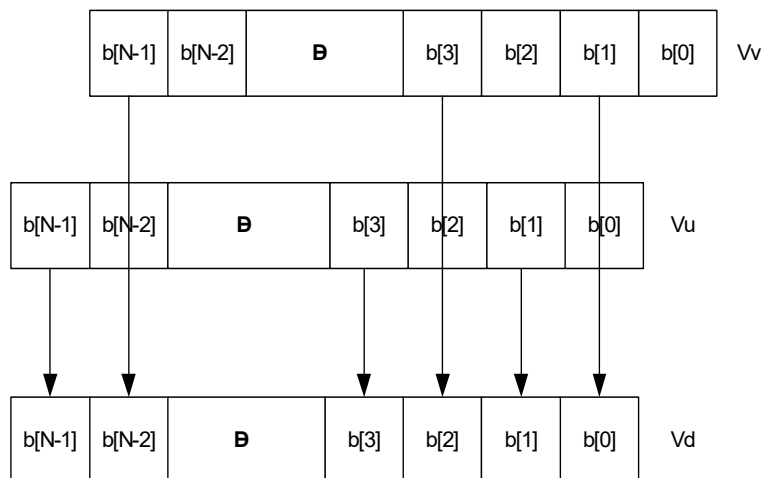
In-lane shuffle

Shuffle the even or odd elements respectively from two vector registers into one destination vector register. Supports bytes and halfwords.

$Vd.b = vshuffe(Vu.b, Vv.b)$



$Vd.b = vshuffo(Vu.b, Vv.b)$



This group of shuffles is limited to bytes and halfwords.

Syntax

`Vd.b=vshuffe(Vu.b,Vv.b)`

`Vd.b=vshuffo(Vu.b,Vv.b)`

Behavior

```
for (i = 0; i < VELEM(16); i++) {
    Vd.uh[i].b[0]=Vv.uh[i].ub[0];
    Vd.uh[i].b[1]=Vu.uh[i].ub[0] ;
}
```

```
for (i = 0; i < VELEM(16); i++) {
    Vd.uh[i].b[0]=Vv.uh[i].ub[1];
    Vd.uh[i].b[1]=Vu.uh[i].ub[1] ;
}
```

Syntax	Behavior
<code>Vd.h=vshuffe(Vu.h,Vv.h)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.uw[i].h[0]=Vv.uw[i].uh[0]; Vd.uw[i].h[1]=Vu.uw[i].uh[0] ; }</pre>
<code>Vd.h=vshuffo(Vu.h,Vv.h)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.uw[i].h[0]=Vv.uw[i].uh[1]; Vd.uw[i].h[1]=Vu.uw[i].uh[1] ; }</pre>

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction can use any HVX resource.

Intrinsics

<code>Vd.b=vshuffe(Vu.b,Vv.b)</code>	<code>HVX_Vector Q6_Vb_vshuffe_VbVb(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.b=vshuffo(Vu.b,Vv.b)</code>	<code>HVX_Vector Q6_Vb_vshuffo_VbVb(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.h=vshuffe(Vu.h,Vv.h)</code>	<code>HVX_Vector Q6_Vh_vshuffe_VhVh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.h=vshuffo(Vu.h,Vv.h)</code>	<code>HVX_Vector Q6_Vh_vshuffo_VhVh(HVX_Vector Vu, HVX_Vector Vv)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
ICLASS																Parse		u5										d5					
0	0	0	1	1	1	1	1	0	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.b=vshuffe(Vu.b,Vv.b)	
0	0	0	1	1	1	1	1	0	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.b=vshuffo(Vu.b,Vv.b)	
0	0	0	1	1	1	1	1	0	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.h=vshuffe(Vu.h,Vv.h)	
0	0	0	1	1	1	1	1	0	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.h=vshuffo(Vu.h,Vv.h)	

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

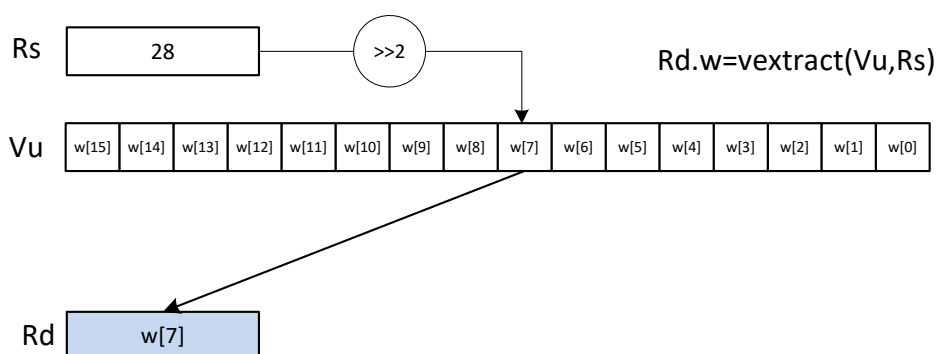
6.4 HVX debug

The HVX debug instruction subclass includes debugging instructions.

Extract vector element

Extract a word from the vector register Vu using bits 5:2 of Rs as the word index. The result is placed in the scalar register Rd. A memory address can be used as the control selection Rs after data has been read from memory using a vector load.

This is a very high latency instruction and should only be used in debug. A memory to memory transfer is more efficient.



Syntax

```
Rd.w=vextract (Vu, Rs)
```

```
Rd=vextract (Vu, Rs)
```

Behavior

Assembler mapped to: "Rd=vextract (Vu, Rs) "

```
Rd = Vu.uw[ (Rs & (VWIDTH-1)) >> 2];
```

Class: LD (slots 0)

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Intrinsics

```
Rd=vextract (Vu, Rs)
```

```
Word32 Q6_R_vextract_VR(HVX_Vector Vu,
Word32 Rs)
```

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode				Type				U	s5				Parse				u5				d5							
1	0	0	1	0	0	1	0	0	0	0	s	s	s	s	s	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Rd=vextract(Vu,Rs)

Field name		Description
ICLASS	Instruction class	
Amode	Amode	
Type	Type	
UN	Unsigned	
Parse	Packet loop parse bits	
d5	Field to encode register d	
s5	Field to encode register s	
u5	Field to encode register u	

6.5 HVX gather double resource

The HVX gather double resource instruction subclass includes instructions that perform gather operations into the vector TCM.

Vector gather

The following instructions perform gather operations in the vector TCM. Gather operations are effectively element copies from a large region in VTCM to a smaller vector-sized region. The larger region of memory is specified by two scalar registers: Rt32 is the base and Mu2 specified the length-1 of the region in bytes. This region must reside in VTCM and cannot cross a page boundary. A vector register, Vv32, specifies byte offsets to this region. Elements of halfword granularity are copied from the address pointed to by Rt + Vv32 for each element in the vector to the corresponding element in the linear element pointed to by the accompanying store.

The offset vector, Vv32, can contain byte offsets specified in either halfword or word sizes. The final element addresses do not have to be byte aligned. If an offset crosses the end of the gather region, it is dropped. Offsets must be positive, otherwise they are dropped. A vector predicate register can also be specified. If a the predicate is false, that byte is not copied. This can be used to emulate a byte gather.

The gather instruction must be paired with a VMEM .new store that uses a tmp register source.

Example: {VMEM(R0+#0) = Vtmp.new; Vtmp.h = vgather(R1,M0, V1:0.w);} gathers halfwords with halfword addresses and saves the results to the address pointed to by R0 of the VMEM instruction. If a `vgather` is not accompanied with a store, it is dropped.

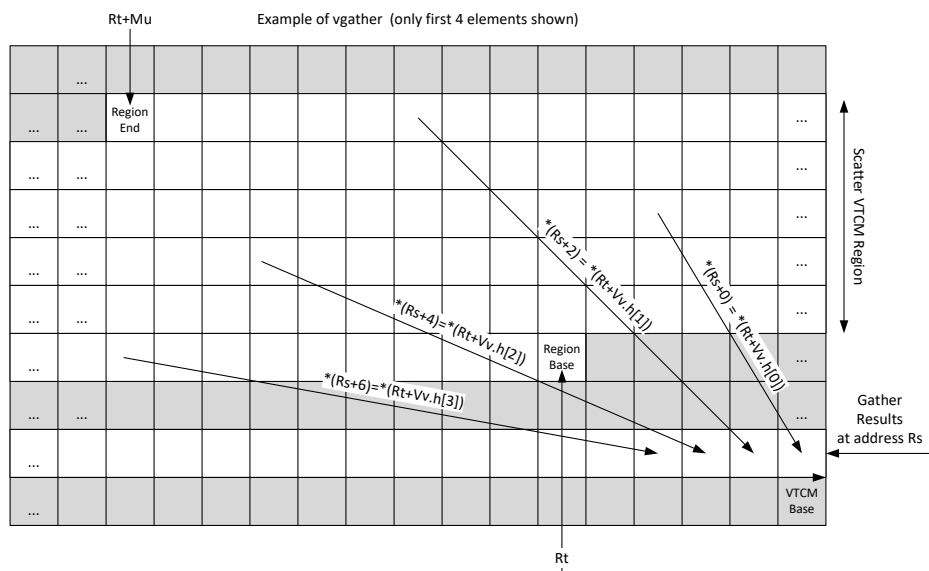
```
{ vmem(Rs+#l)=Vtmp.h; vtmp.h = vgather(Rt,Mu,Vv.h) }
```

Rs – Address of gathered values in VTCM

Rt – Scalar Indicating base address in VTCM

Mu – Scalar indicating length-1 of Region

Vv – Vector with byte offsets from base



Syntax	Behavior
<pre>if (Qs4) vtmp.h=vgather(Rt,Mu,Vvv.w).h</pre>	<pre>MuV = MuV (element_size-1); Rt = Rt & ~(element_size-1); for (i = 0; i < VELEM(32); i++) { for(j = 0; j < 2; j++) { EA = Rt+Vvv.v[j].uw[i]; if ((Rt <= EA <= Rt + MuV) & QsV) TEMP.uw[i].uh[j] = *EA; } }</pre>
<pre>vtmp.h=vgather(Rt,Mu,Vvv.w).h</pre>	<pre>MuV = MuV (element_size-1); Rt = Rt & ~(element_size-1); for (i = 0; i < VELEM(32); i++) { for(j = 0; j < 2; j++) { EA = Rt+Vvv.v[j].uw[i]; if (Rt <= EA <= Rt + MuV) TEMP.uw[i].uh[j] = *EA; } }</pre>

Class: COPROC_VMEM (slots 0,1)**Notes**

- This instruction can use any HVX resource.

Intrinsics

```
if (Qs4) vtmp.h =
vgather(Rt,Mu,Vvv.w).h
```

```
void Q6_vgather_AQRMWw(HVX_Vector* A,
HVX_VectorPred Qs, HVX_Vector* Rb, Word32 Mu,
HVX_VectorPair Vvv)
```

```
vtmp.h=vgather(Rt,Mu,Vvv.w).h
```

```
void Q6_vgather_ARMWw(HVX_Vector* A,
HVX_Vector* Rb, Word32 Mu, HVX_VectorPair Vvv)
```

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
ICLASS									NT		t5					Parse		u1																
0	0	1	0	1	1	1	1	0	0	0	t	t	t	t	t	P	P	u	-	-	0	1	0	-	-	-	v	v	v	v	v	vtmp.h=vgather(Rt,Mu,Vvv.w).h		
ICLASS									NT		t5					Parse		u1						s2										
0	0	1	0	1	1	1	1	0	0	0	t	t	t	t	t	P	P	u	-	-	1	1	0	-	s	s	v	v	v	v	v	if (Qs4) vtmp.h=vgather(Rt,Mu,Vvv.w).h		

Field name	Description
ICLASS	Instruction class
NT	Nontemporal
Parse	Packet loop parse bits
s2	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
v5	Field to encode register v

6.6 HVX gather

The HVX gather instruction subclass includes instructions that perform gather operations.

Vector gather

The following instructions perform gather operations in the vector TCM. Gather operations are effectively element copies from a large region in VTCM to a smaller vector-sized region. The larger region of memory is specified by two scalar registers: Rt32 is the base and Mu2 specified the length-1 of the region in bytes. This region must reside in VTCM and cannot cross a page boundary. A vector register, Vv32, specifies byte offsets to this region. Elements of either halfword or word granularity are copied from the address pointed to by Rt + Vv32 for each element in the vector to the corresponding element in the linear element pointed to by the accompanying store.

The offset vector, Vv32, can contain byte offsets specified in either halfword or word sizes. The final element addresses do not have to be byte aligned. If an offset crosses the end of a gather region, it is dropped. Offsets must be positive, otherwise they are dropped. A vector predicate register can also be specified. If a the predicate is false, that byte is not copied. This can be used to emulate a byte gather.

The gather instruction must be paired with a VMEM .new store that uses a tmp register source.

For example: `{VMEM(R0+#0) = Vtmp.new; Vtmp.h = vgather(R1,M0, V0.h);}` gathers halfwords with halfword addresses and saves the results to the address pointed to by R0 of the VMEM instruction. If a vgather is not accompanied with a store, it is dropped.

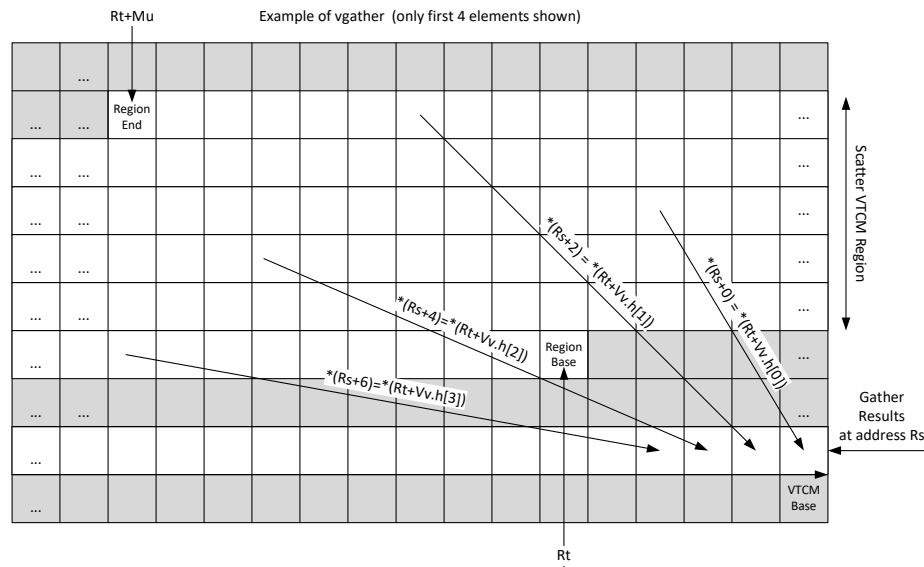
```
{ vmem(Rs+#I)=Vtmp.h; vtmp.h = vgather(Rt,Mu,Vv.h) }
```

Rs – Address of gathered values in VTCM

Rt – Scalar Indicating base address in VTCM

Mu – Scalar indicating length-1 of Region

Vv – Vector with byte offsets from base



Syntax	Behavior
<pre>if (Qs4) vtmp.h=vgather(Rt,Mu,Vv.h).h</pre>	<pre>MuV = MuV (element_size-1); Rt = Rt & ~(element_size-1); for (i = 0; i < VELEM(16); i++) { EA = Rt+Vv.uh[i]; if ((Rt <= EA <= Rt + MuV) & QsV) TEMP.uh[i] = *EA; }</pre>
<pre>if (Qs4) vtmp.w=vgather(Rt,Mu,Vv.w).w</pre>	<pre>MuV = MuV (element_size-1); Rt = Rt & ~(element_size-1); for (i = 0; i < VELEM(32); i++) { EA = Rt+Vv.uw[i]; if ((Rt <= EA <= Rt + MuV) & QsV) TEMP.uw[i] = *EA; }</pre>
<pre>vtmp.h=vgather(Rt,Mu,Vv.h).h</pre>	<pre>MuV = MuV (element_size-1); Rt = Rt & ~(element_size-1); for (i = 0; i < VELEM(16); i++) { EA = Rt+Vv.uh[i]; if (Rt <= EA <= Rt + MuV) TEMP.uh[i] = *EA; }</pre>
<pre>vtmp.w=vgather(Rt,Mu,Vv.w).w</pre>	<pre>MuV = MuV (element_size-1); Rt = Rt & ~(element_size-1); for (i = 0; i < VELEM(32); i++) { EA = Rt+Vv.uw[i]; if (Rt <= EA <= Rt + MuV) TEMP.uw[i] = *EA; }</pre>

Class: COPROC_VMEM (slots 0,1)

Notes

- This instruction can use any HVX resource.

Intrinsics

<pre>if (Qs4) vtmp.h=vgather(Rt,Mu,Vv.h).h</pre>	<pre>void Q6_vgather_AQRMVh(HVX_Vector* A, HVX_VectorPred Qs, HVX_Vector* Rb, Word32 Mu, HVX_Vector Vv)</pre>
<pre>if (Qs4) vtmp.w=vgather(Rt,Mu,Vv.w).w</pre>	<pre>void Q6_vgather_AQRMVw(HVX_Vector* A, HVX_VectorPred Qs, HVX_Vector* Rb, Word32 Mu, HVX_Vector Vv)</pre>
<pre>vtmp.h=vgather(Rt,Mu,Vv.h).h</pre>	<pre>void Q6_vgather_ARMVh(HVX_Vector* A, HVX_Vector* Rb, Word32 Mu, HVX_Vector Vv)</pre>
<pre>vtmp.w=vgather(Rt,Mu,Vv.w).w</pre>	<pre>void Q6_vgather_ARMVw(HVX_Vector* A, HVX_Vector* Rb, Word32 Mu, HVX_Vector Vv)</pre>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
ICLASS									NT		t5					Parse		u1																
0	0	1	0	1	1	1	1	0	0	0	t	t	t	t	t	P	P	u	-	-	0	0	0	-	-	-	v	v	v	v	v	vtmp.w=vgather(Rt,Mu,Vv.w).w		
0	0	1	0	1	1	1	1	0	0	0	t	t	t	t	t	P	P	u	-	-	0	0	1	-	-	-	v	v	v	v	v	vtmp.h=vgather(Rt,Mu,Vv.h).h		
ICLASS									NT		t5					Parse		u1								s2								
0	0	1	0	1	1	1	1	0	0	0	t	t	t	t	t	P	P	u	-	-	1	0	0	-	s	s	v	v	v	v	v	if (Qs4) vtmp.w=vgather(Rt,Mu,Vv.w).w		
0	0	1	0	1	1	1	1	0	0	0	t	t	t	t	t	P	P	u	-	-	1	0	1	-	s	s	v	v	v	v	v	if (Qs4) vtmp.h=vgather(Rt,Mu,Vv.h).h		

Field name	Description
ICLASS	Instruction class
NT	Nontemporal
Parse	Packet loop parse bits
s2	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
v5	Field to encode register v

6.7 HVX load

The HVX load instruction subclass includes memory load instructions.

Load - aligned

Read a full vector register Vd from memory, using a vector-size-aligned address. The operation has three ways to generate the memory pointer address: Rt with a constant 4-bit signed offset, Rx with a signed post-increment, and Rx with a modifier register Mu post-increment. For the immediate forms, the value specifies the number of vectors worth of data. Mu contains the actual byte offset.

If the pointer presented to the instruction is not aligned, the instruction ignores the lower bits, yielding an aligned address.

If a scalar predicate register Pv evaluates true, load a full vector register Vs from memory, using a vector-size-aligned address. Otherwise, the operation becomes a NOP.

Syntax	Behavior
Vd=vmem(Rt)	Assembler mapped to: "Vd=vmem(Rt+#0)"
Vd=vmem(Rt)	Assembler mapped to: "Vd=vmem(Rt+#0)"
Vd=vmem(Rt):nt	Assembler mapped to: "Vd=vmem(Rt+#0):nt"
Vd=vmem(Rt+#s10:6)	EA=Rt+#s; Vd = L2S[EA&~(128-1)];
Vd=vmem(Rt+#s4)	EA=Rt+#s*VBYTES; Vd = *(EA&~(ALIGNMENT-1));
Vd=vmem(Rt+#s4):nt	EA=Rt+#s*VBYTES; Vd = *(EA&~(ALIGNMENT-1));
Vd=vmem(Rt+Rs)	EA=Rt+Rs; Vd = L2S[EA&~(128-1)];
Vd=vmem(Rx++#s10:6)	EA=Rx; Vd = L2S[EA&~(128-1)]; Rx=Rx+#s;
Vd=vmem(Rx++#s3)	EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+#s*VBYTES;
Vd=vmem(Rx++#s3):nt	EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+#s*VBYTES;
Vd=vmem(Rx++I:circ(Mu))	EA=Rx; Vd = L2S[EA&~(128-1)]; Rx=Rx=circ_add(Rx,I,MuV);

Syntax	Behavior
<code>Vd = vmem(Rx++I:circ(Mu), Rt)</code>	<pre> EA=Rx; int aligned_load =0; if (aligned_load) { Vd = L2S[EA&~(128-1)]; } else { 0}; TMP0 = L2S[EA&~(128-1)]TMP1 = L2S[PA+128]if ((Rt == 0x80000400)) { I Q_SWAP(TMP0,V8u,128); TMP1=SWAP_IQ(TMP1); } else if ((Rt == 0x80000600)) { TMP0=SWAP_IQ(TMP0); TMP1=SWAP_IQ(TMP1); } if(Rt>>31) sh = ((EA & 0x7f)); for (k=0;k<128-sh;k++) VTMP.V8u[k] = TMP0.V8u[sh+k]; for (k2=0;k2<sh;k2++) VTMP.V8u[k++] = TMP1.V8u[k2]; Vd = VTMP; if ((Rt == 0x80000001)) { Vd = UNPACK TO PCFL 10_4 (VTMP); } else if ((Rt == 0x80000002)) { Vd = UNPACK TO PCFL 12_4 (VTMP); } else if ((Rt == 0x80000003)) { Vd = UNPACK TO PCFL 16_4 (VTMP); } else if ((Rt == 0x80000004)) { Vd = UNPACK TO PCFL 20_4 (VTMP); } else if ((Rt == 0x80000005)) { Vd = UNPACK TO PCFL 24_4 (VTMP); } else if ((Rt == 0x8000000b)) { Vd = UNPACK TO PCFL 16_2 (VTMP); } else if ((Rt == 0x80000010)) { Vd = UNPACK TO UFL 8_5 (VTMP); } else if ((Rt == 0x80000011)) { Vd = UNPACK TO UFL 10_5 (VTMP); } else if ((Rt == 0x80000012)) { Vd = UNPACK TO UFL 12_5 (VTMP); } else if ((Rt == 0x80000013)) { Vd = UNPACK TO UFL 16_5 (VTMP); } else if ((Rt == 0x8000001b)) { Vd = UNPACK TO SFL 16_5 (VTMP); } else if (((Rt & ~(7 <<12)) == 0x8000002c)) { Vd = UNPACK TO PCFL 20_H (VTMP); } else if (((Rt & ~(3 <<12)) == 0x8000002d)) { Vd = UNPACK TO PCFL 24_H (VTMP); } else if ((Rt == 0x80000030)) { Vd = UNPACK TO LLR6 (VTMP); };} Rx=Rx=circ_add(Rx,I,MuV); </pre>
<code>Vd = vmem(Rx ++ Mu)</code>	<pre> EA=Rx; Vd = L2S[EA&~(128-1)]; Rx=Rx+MuV; </pre>

Syntax	Behavior
<code>Vd = vmem(Rx ++ Mu)</code>	<pre>EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+MuV;</pre>
<code>Vd = vmem(Rx ++ Mu):nt</code>	<pre>EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+MuV;</pre>
<code>Vd = vmem(Rx++Mu,Rt)</code>	<pre>EA=Rx; int aligned_load = 0; if (aligned_load) { Vd = L2S[EA&~(128-1)]; } else {0}; TMP0 = L2S[EA&~(128-1)]TMP1 = L2S[PA+128]if ((Rt == 0x80000400)) { I Q_SWAP(TMP0,V8u,128); TMP1=SWAP_IQ(TMP1); } else if ((Rt == 0x80000600)) { TMP0=SWAP_IQ(TMP0); TMP1=SWAP_IQ(TMP1);} if(Rt>>31) sh = ((EA & 0x7f)); for (k=0;k<128-sh;k++) VTMP.V8u[k] = TMP0.V8u[sh+k]; for (k2=0;k2<sh;k2++) VTMP.V8u[k++] = TMP1.V8u[k2]; Vd = VTMP; if ((Rt == 0x80000001)) { Vd = UNPACK TO PCFL 10_4(VTMP); } else if ((Rt == 0x80000002)) { Vd = UNPACK TO PCFL 12_4(VTMP); } else if ((Rt == 0x80000003)) { Vd = UNPACK TO PCFL 16_4(VTMP); } else if ((Rt == 0x80000004)) { Vd = UNPACK TO PCFL 20_4(VTMP); } else if ((Rt == 0x80000005)) { Vd = UNPACK TO PCFL 24_4(VTMP); } else if ((Rt == 0x8000000b)) { Vd = UNPACK TO PCFL 16_2(VTMP); } else if ((Rt == 0x80000010)) { Vd = UNPACK TO UFL 8_5(VTMP); } else if ((Rt == 0x80000011)) { Vd = UNPACK TO UFL 10_5(VTMP); } else if ((Rt == 0x80000012)) { Vd = UNPACK TO UFL 12_5(VTMP); } else if ((Rt == 0x80000013)) { Vd = UNPACK TO UFL 16_5(VTMP); } else if ((Rt == 0x8000001b)) { Vd = UNPACK TO SFL 16_5(VTMP); } else if (((Rt & ~(7 <<12)) == 0x8000002c)) { Vd = UNPACK TO PCFL 20_H(VTMP); } else if (((Rt & ~(3 <<12)) == 0x8000002d)) { Vd = UNPACK TO PCFL 24_H(VTMP); } else if ((Rt == 0x80000030)) { Vd = UNPACK TO LLR6(VTMP);};} Rx=Rx+MuV;</pre>

Syntax	Behavior
<code>if (!Pv) Vd=vmem(Rt)</code>	Assembler mapped to: <code>"if (!Pv) Vd=vmem(Rt+#0)"</code>
<code>if (!Pv) Vd=vmem(Rt):nt</code>	Assembler mapped to: <code>"if (!Pv) Vd=vmem(Rt+#0):nt"</code>
<code>if (!Pv) Vd=vmem(Rt+#s4)</code>	<pre> if (!Pv[0]) { EA=Rt+#s*VBYTES; Vd = *(EA&~(ALIGNMENT-1)); } else { NOP; } </pre>
<code>if (!Pv) Vd=vmem(Rt+#s4):nt</code>	<pre> if (!Pv[0]) { EA=Rt+#s*VBYTES; Vd = *(EA&~(ALIGNMENT-1)); } else { NOP; } </pre>
<code>if (!Pv) Vd=vmem(Rx++#s3)</code>	<pre> if (!Pv[0]) { EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+#s*VBYTES; } else { NOP; } </pre>
<code>if (!Pv) Vd=vmem(Rx++#s3):nt</code>	<pre> if (!Pv[0]) { EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+#s*VBYTES; } else { NOP; } </pre>
<code>if (!Pv) Vd=vmem(Rx++Mu)</code>	<pre> if (!Pv[0]) { EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+MuV; } else { NOP; } </pre>
<code>if (!Pv) Vd=vmem(Rx++Mu):nt</code>	<pre> if (!Pv[0]) { EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+MuV; } else { NOP; } </pre>

Class: COPROC_VMEM (slots 0,1)**Notes**

- This instruction can use any HVX resource.

- An optional non-temporal hint to the microarchitecture can be specified to indicate the data has no reuse.
- Immediates used in address computation are specified in multiples of vector length.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
ICLASS									NT		t5					Parse													d5					
0	0	1	0	1	0	0	0	0	0	0	t	t	t	t	t	P	P	i	0	0	i	i	i	0	0	0	d	d	d	d	d	Vd=vmem(Rt+#s4)		
0	0	1	0	1	0	0	0	0	1	0	t	t	t	t	t	P	P	i	0	0	i	i	i	0	0	0	d	d	d	d	d	Vd=vmem(Rt+#s4):nt		
0	0	1	0	1	0	0	0	1	0	0	t	t	t	t	t	P	P	i	v	v	i	i	i	0	1	0	d	d	d	d	d	if (Pv) Vd=vmem(Rt+#s4)		
0	0	1	0	1	0	0	0	1	0	0	t	t	t	t	t	P	P	i	v	v	i	i	i	0	1	1	d	d	d	d	d	if (IPv) Vd=vmem(Rt+#s4)		
ICLASS									NT		t5					Parse			s5										d5					
0	0	1	0	1	0	0	0	1	0	1	t	t	t	t	t	P	P	0	s	s	s	s	s	d	d	d	d	d	0	0	0	Vd=vmem(Rt+Rs)		
ICLASS									NT		t5					Parse													d5					
0	0	1	0	1	0	0	0	1	1	0	t	t	t	t	t	P	P	i	v	v	i	i	i	0	1	0	d	d	d	d	d	if (Pv) Vd=vmem(Rt+#s4):nt		
0	0	1	0	1	0	0	0	1	1	0	t	t	t	t	t	P	P	i	v	v	i	i	i	0	1	1	d	d	d	d	d	if (IPv) Vd=vmem(Rt+#s4):nt		
ICLASS									NT		x5					Parse													d5					
0	0	1	0	1	0	0	1	0	0	0	x	x	x	x	x	P	P	-	0	0	i	i	i	0	0	0	d	d	d	d	d	Vd=vmem(Rx++#s3)		
0	0	1	0	1	0	0	1	0	1	0	x	x	x	x	x	P	P	-	0	0	i	i	i	0	0	0	d	d	d	d	d	Vd=vmem(Rx++#s3):nt		
0	0	1	0	1	0	0	1	1	0	0	x	x	x	x	x	P	P	-	v	v	i	i	i	0	1	0	d	d	d	d	d	if (Pv) Vd=vmem(Rx++#s3)		
0	0	1	0	1	0	0	1	1	0	0	x	x	x	x	x	P	P	-	v	v	i	i	i	0	1	1	d	d	d	d	d	if (IPv) Vd=vmem(Rx++#s3)		
0	0	1	0	1	0	0	1	1	1	0	x	x	x	x	x	P	P	-	v	v	i	i	i	0	1	0	d	d	d	d	d	if (Pv) Vd=vmem(Rx++#s3):nt		
0	0	1	0	1	0	0	1	1	1	0	x	x	x	x	x	P	P	-	v	v	i	i	i	0	1	1	d	d	d	d	d	if (IPv) Vd=vmem(Rx++#s3):nt		
ICLASS									NT		x5					Parse			u1	d5														
0	0	1	0	1	0	1	0	0	0	0	x	x	x	x	x	P	P	u	-	-	-	-	-	d	d	d	d	d	0	-	0	Vd=vmem(Rx++!circ(Mu))		
ICLASS									NT		x5					Parse			u1	t5					d5									
0	0	1	0	1	0	1	0	0	0	1	x	x	x	x	x	P	P	u	t	t	t	t	t	d	d	d	d	d	-	-	-	Vd=vmem(Rx++!circ(Mu), Rt)		
ICLASS									NT		x5					Parse			u1	d5														
0	0	1	0	1	0	1	1	0	0	0	x	x	x	x	x	P	P	u	-	-	-	-	-	d	d	d	d	d	0	-	0	Vd=vmem(Rx++Mu)		
ICLASS									NT		x5					Parse			u1	d5														
0	0	1	0	1	0	1	1	0	0	0	x	x	x	x	x	P	P	u	0	0	-	-	-	0	0	0	d	d	d	d	d	Vd=vmem(Rx++Mu)		
ICLASS									NT		x5					Parse			u1	t5					d5									
0	0	1	0	1	0	1	1	0	0	1	x	x	x	x	x	P	P	u	t	t	t	t	t	d	d	d	d	d	-	-	-	Vd=vmem(Rx++Mu, Rt)		
ICLASS									NT		x5					Parse			u1	d5														
0	0	1	0	1	0	1	1	0	1	0	x	x	x	x	x	P	P	u	0	0	-	-	-	0	0	0	d	d	d	d	d	Vd=vmem(Rx++Mu):nt		
0	0	1	0	1	0	1	1	1	0	0	x	x	x	x	x	P	P	u	v	v	-	-	-	0	1	0	d	d	d	d	d	if (Pv) Vd=vmem(Rx++Mu)		
0	0	1	0	1	0	1	1	1	0	0	x	x	x	x	x	P	P	u	v	v	-	-	-	0	1	1	d	d	d	d	d	if (IPv) Vd=vmem(Rx++Mu)		
0	0	1	0	1	0	1	1	1	1	0	x	x	x	x	x	P	P	u	v	v	-	-	-	0	1	0	d	d	d	d	d	if (Pv) Vd=vmem(Rx++Mu):nt		
0	0	1	0	1	0	1	1	1	1	0	x	x	x	x	x	P	P	u	v	v	-	-	-	0	1	1	d	d	d	d	d	if (IPv) Vd=vmem(Rx++Mu):nt		
ICLASS									NT		x5					Parse			d5															
0	0	1	0	1	1	0	0	0	0	i	x	x	x	x	x	P	P	i	i	i	i	i	i	d	d	d	d	d	i	i	i	Vd=vmem(Rx++#s10:6)		
ICLASS									NT		t5					Parse			d5															
0	0	1	0	1	1	1	1	0	1	i	t	t	t	t	t	P	P	i	i	i	i	i	i	d	d	d	d	d	i	i	i	Vd=vmem(Rt+#s10:6)		

Field name		Description
ICLASS	Instruction class	
NT	Nontemporal	
Parse	Packet loop parse bits	
d5	Field to encode register d	
s5	Field to encode register s	
t5	Field to encode register t	
u1	Field to encode register u	
v2	Field to encode register v	
x5	Field to encode register x	

Load - immediate use

Read a full vector register Vd (and/or temporary vector register) from memory, using a vector-size-aligned address. The operation has three ways to generate the memory pointer address: Rt with a constant 4-bit signed offset, Rx with a signed post-increment, and Rx with a modifier register Mu post-increment. For the immediate forms, the value indicates the number of vectors worth of data. Mu contains the actual byte offset.

If the pointer presented to the instruction is not aligned, the instruction ignores the lower bits, yielding an aligned address. The value is used immediately in the packet as a source operand of any instruction.

The Vd.cur instruction writes the load value to a vector register in addition to consuming it within the packet.

The Vd.tmp instruction does not write the incoming data to the vector register file. The data is only used as a source in the current packet, and then immediately discarded. This form does not consume vector resources, allowing it to be placed in parallel with some instructions that a normal align load cannot.

If a scalar predicate register Pv evaluates true, load a full vector register Vs from memory, using a vector-size-aligned address. Otherwise, the operation becomes a NOP.

Syntax	Behavior
Vd.cur=vmem(Rt+#s4)	EA=Rt+#s*VBYTES; Vd = *(EA&~(ALIGNMENT-1));
Vd.cur=vmem(Rt+#s4):nt	EA=Rt+#s*VBYTES; Vd = *(EA&~(ALIGNMENT-1));
Vd.cur=vmem(Rx++#s3)	EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+#s*VBYTES;
Vd.cur=vmem(Rx++#s3):nt	EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+#s*VBYTES;
Vd.cur=vmem(Rx++Mu)	EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+MuV;
Vd.cur=vmem(Rx++Mu):nt	EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+MuV;
if (!Pv) Vd.cur=vmem(Rt)	Assembler mapped to: "if (!Pv) Vd.cur=vmem(Rt+#0)"
if (!Pv) Vd.cur=vmem(Rt):nt	Assembler mapped to: "if (!Pv) Vd.cur=vmem(Rt+#0):nt"
if (!Pv) Vd.cur=vmem(Rt+#s4)	if (!Pv[0]) { EA=Rt+#s*VBYTES; Vd = *(EA&~(ALIGNMENT-1)); } else { NOP; }

Syntax	Behavior
<pre>if (!Pv) Vd.cur=vmem(Rt+#s4):nt</pre>	<pre>if (!Pv[0]) { EA=Rt+#s*VBYTES; Vd = *(EA&~(ALIGNMENT-1)); } else { NOP; }</pre>
<pre>if (!Pv) Vd.cur=vmem(Rx++#s3)</pre>	<pre>if (!Pv[0]) { EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+#s*VBYTES; } else { NOP; }</pre>
<pre>if (!Pv) Vd.cur=vmem(Rx++#s3):nt</pre>	<pre>if (!Pv[0]) { EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+#s*VBYTES; } else { NOP; }</pre>
<pre>if (!Pv) Vd.cur=vmem(Rx++Mu)</pre>	<pre>if (!Pv[0]) { EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+MuV; } else { NOP; }</pre>
<pre>if (!Pv) Vd.cur=vmem(Rx++Mu):nt</pre>	<pre>if (!Pv[0]) { EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+MuV; } else { NOP; }</pre>

Class: COPROC_VMEM (slots 0,1)**Notes**

- This instruction can use any HVX resource.
- An optional non-temporal hint to the microarchitecture can be specified to indicate the data has no reuse.
- Immediates used in address computation are specified in multiples of vector length.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
ICLASS											NT		t5					Parse												d5					
0	0	1	0	1	0	0	0	0	0	0	t	t	t	t	t	P	P	i	0	0	i	i	i	0	0	1	d	d	d	d	d	Vd.cur=vmem(Rt+#s4)			
0	0	1	0	1	0	0	0	0	1	0	t	t	t	t	t	P	P	i	0	0	i	i	i	0	0	1	d	d	d	d	d	Vd.cur=vmem(Rt+#s4):nt			
0	0	1	0	1	0	0	0	1	0	0	t	t	t	t	t	P	P	i	v	v	i	i	i	1	0	0	d	d	d	d	d	if (Pv) Vd.cur=vmem(Rt+#s4)			
0	0	1	0	1	0	0	0	1	0	0	t	t	t	t	t	P	P	i	v	v	i	i	i	1	0	1	d	d	d	d	d	if (!Pv) Vd.cur=vmem(Rt+#s4)			
0	0	1	0	1	0	0	0	1	1	0	t	t	t	t	t	P	P	i	v	v	i	i	i	1	0	0	d	d	d	d	d	if (Pv) Vd.cur=vmem(Rt+#s4):nt			
0	0	1	0	1	0	0	0	1	1	0	t	t	t	t	t	P	P	i	v	v	i	i	i	1	0	1	d	d	d	d	d	if (!Pv) Vd.cur=vmem(Rt+#s4):nt			
ICLASS											NT		x5					Parse												d5					
0	0	1	0	1	0	0	1	0	0	0	x	x	x	x	x	P	P	-	0	0	i	i	i	0	0	1	d	d	d	d	d	Vd.cur=vmem(Rx++#s3)			
0	0	1	0	1	0	0	1	0	1	0	x	x	x	x	x	P	P	-	0	0	i	i	i	0	0	1	d	d	d	d	d	Vd.cur=vmem(Rx++#s3):nt			
0	0	1	0	1	0	0	1	1	0	0	x	x	x	x	x	P	P	-	v	v	i	i	i	1	0	0	d	d	d	d	d	if (Pv) Vd.cur=vmem(Rx++#s3)			
0	0	1	0	1	0	0	1	1	0	0	x	x	x	x	x	P	P	-	v	v	i	i	i	1	0	1	d	d	d	d	d	if (!Pv) Vd.cur=vmem(Rx++#s3)			
0	0	1	0	1	0	0	1	1	1	0	x	x	x	x	x	P	P	-	v	v	i	i	i	1	0	0	d	d	d	d	d	if (Pv) Vd.cur=vmem(Rx++#s3):nt			
0	0	1	0	1	0	0	1	1	1	0	x	x	x	x	x	P	P	-	v	v	i	i	i	1	0	1	d	d	d	d	d	if (!Pv) Vd.cur=vmem(Rx++#s3):nt			
ICLASS											NT		x5					Parse		u1										d5					
0	0	1	0	1	0	1	1	1	0	0	x	x	x	x	x	P	P	u	0	0	-	-	-	0	0	1	d	d	d	d	d	Vd.cur=vmem(Rx++Mu)			
0	0	1	0	1	0	1	1	0	1	0	x	x	x	x	x	P	P	u	0	0	-	-	-	0	0	1	d	d	d	d	d	Vd.cur=vmem(Rx++Mu):nt			
0	0	1	0	1	0	1	1	1	0	0	x	x	x	x	x	P	P	u	v	v	-	-	-	1	0	0	d	d	d	d	d	if (Pv) Vd.cur=vmem(Rx++Mu)			
0	0	1	0	1	0	1	1	1	0	0	x	x	x	x	x	P	P	u	v	v	-	-	-	1	0	1	d	d	d	d	d	if (!Pv) Vd.cur=vmem(Rx++Mu)			
0	0	1	0	1	0	1	1	1	1	0	x	x	x	x	x	P	P	u	v	v	-	-	-	1	0	0	d	d	d	d	d	if (Pv) Vd.cur=vmem(Rx++Mu):nt			
0	0	1	0	1	0	1	1	1	1	0	x	x	x	x	x	P	P	u	v	v	-	-	-	1	0	1	d	d	d	d	d	if (!Pv) Vd.cur=vmem(Rx++Mu):nt			

Field name	Description
ICLASS	Instruction class
NT	Nontemporal
Parse	Packet loop parse bits
d5	Field to encode register d
t5	Field to encode register t
u1	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x

Load - temporary immediate use

Read a full vector register Vd (and/or temporary vector register) from memory, using a vector-size-aligned address. The operation has three ways to generate the memory pointer address: Rt with a constant 4-bit signed offset, Rx with a signed post-increment, and Rx with a modifier register Mu post-increment. For the immediate forms, the value indicates the number of vectors worth of data. Mu contains the actual byte offset.

If the pointer presented to the instruction is not aligned, the instruction ignores the lower bits, yielding an aligned address. The value is used immediately in the packet as a source operand of any instruction.

The Vd.tmp instruction does not write the incoming data to the vector register file. The data is only used as a source in the current packet, and then immediately discarded. This form does not consume vector resources, allowing it to be placed in parallel with some instructions that a normal align load cannot.

If a scalar predicate register Pv evaluates true, load a full vector register Vs from memory, using a vector-size-aligned address. Otherwise, the operation becomes a NOP.

Syntax	Behavior
Vd.tmp=vmem(Rt+#s4)	EA=Rt+#s*VBYTES; Vd = *(EA&~(ALIGNMENT-1));
Vd.tmp=vmem(Rt+#s4):nt	EA=Rt+#s*VBYTES; Vd = *(EA&~(ALIGNMENT-1));
Vd.tmp=vmem(Rx++#s3)	EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+#s*VBYTES;
Vd.tmp=vmem(Rx++#s3):nt	EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+#s*VBYTES;
Vd.tmp=vmem(Rx++Mu)	EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+MuV;
Vd.tmp=vmem(Rx++Mu):nt	EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+MuV;
if (!Pv) Vd.tmp=vmem(Rt)	Assembler mapped to: "if (!Pv) Vd.tmp=vmem(Rt+#0)"
if (!Pv) Vd.tmp=vmem(Rt):nt	Assembler mapped to: "if (!Pv) Vd.tmp=vmem(Rt+#0):nt"
if (!Pv) Vd.tmp=vmem(Rt+#s4)	if (!Pv[0]) { EA=Rt+#s*VBYTES; Vd = *(EA&~(ALIGNMENT-1)); } else { NOP; }

Syntax	Behavior
<pre>if (!Pv) Vd.tmp=vmem(Rt+#s4):nt</pre>	<pre>if (!Pv[0]) { EA=Rt+#s*VBYTES; Vd = *(EA&~(ALIGNMENT-1)); } else { NOP; }</pre>
<pre>if (!Pv) Vd.tmp=vmem(Rx++#s3)</pre>	<pre>if (!Pv[0]) { EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+#s*VBYTES; } else { NOP; }</pre>
<pre>if (!Pv) Vd.tmp=vmem(Rx++#s3):nt</pre>	<pre>if (!Pv[0]) { EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+#s*VBYTES; } else { NOP; }</pre>
<pre>if (!Pv) Vd.tmp=vmem(Rx++Mu)</pre>	<pre>if (!Pv[0]) { EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+MuV; } else { NOP; }</pre>
<pre>if (!Pv) Vd.tmp=vmem(Rx++Mu):nt</pre>	<pre>if (!Pv[0]) { EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+MuV; } else { NOP; }</pre>

Class: COPROC_VMEM (slots 0,1)**Notes**

- This instruction can use any HVX resource.
- An optional non-temporal hint to the microarchitecture can be specified to indicate the data has no reuse.
- The tmp load instruction destination register cannot be an accumulator register.
- Immediates used in address computation are specified in multiples of vector length.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
ICLASS									NT			t5					Parse										d5							
0	0	1	0	1	0	0	0	0	0	0	t	t	t	t	t	P	P	i	0	0	i	i	i	0	1	0	d	d	d	d	d	Vd.tmp=vmem(Rt+#s4)		
0	0	1	0	1	0	0	0	0	1	0	t	t	t	t	t	P	P	i	0	0	i	i	i	0	1	0	d	d	d	d	d	Vd.tmp=vmem(Rt+#s4):nt		
0	0	1	0	1	0	0	0	1	0	0	t	t	t	t	t	P	P	i	v	v	i	i	i	1	1	0	d	d	d	d	d	if (Pv) Vd.tmp=vmem(Rt+#s4)		
0	0	1	0	1	0	0	0	1	0	0	t	t	t	t	t	P	P	i	v	v	i	i	i	1	1	1	d	d	d	d	d	if (IPv) Vd.tmp=vmem(Rt+#s4)		
0	0	1	0	1	0	0	0	1	1	0	t	t	t	t	t	P	P	i	v	v	i	i	i	1	1	0	d	d	d	d	d	if (Pv) Vd.tmp=vmem(Rt+#s4):nt		
0	0	1	0	1	0	0	0	1	1	0	t	t	t	t	t	P	P	i	v	v	i	i	i	1	1	1	d	d	d	d	d	if (IPv) Vd.tmp=vmem(Rt+#s4):nt		
ICLASS									NT			x5					Parse										d5							
0	0	1	0	1	0	0	1	0	0	0	x	x	x	x	x	P	P	-	0	0	i	i	i	0	1	0	d	d	d	d	d	Vd.tmp=vmem(Rx++#s3)		
0	0	1	0	1	0	0	1	0	1	0	x	x	x	x	x	P	P	-	0	0	i	i	i	0	1	0	d	d	d	d	d	Vd.tmp=vmem(Rx++#s3):nt		
0	0	1	0	1	0	0	1	1	0	0	x	x	x	x	x	P	P	-	v	v	i	i	i	1	1	0	d	d	d	d	d	if (Pv) Vd.tmp=vmem(Rx++#s3)		
0	0	1	0	1	0	0	1	1	0	0	x	x	x	x	x	P	P	-	v	v	i	i	i	1	1	1	d	d	d	d	d	if (IPv) Vd.tmp=vmem(Rx++#s3)		
0	0	1	0	1	0	0	1	1	1	0	x	x	x	x	x	P	P	-	v	v	i	i	i	1	1	0	d	d	d	d	d	if (Pv) Vd.tmp=vmem(Rx++#s3):nt		
0	0	1	0	1	0	0	1	1	1	0	x	x	x	x	x	P	P	-	v	v	i	i	i	1	1	1	d	d	d	d	d	if (IPv) Vd.tmp=vmem(Rx++#s3):nt		
ICLASS									NT			x5					Parse										u1		d5					
0	0	1	0	1	0	1	1	0	0	0	x	x	x	x	x	P	P	u	0	0	-	-	-	0	1	0	d	d	d	d	d	Vd.tmp=vmem(Rx++Mu)		
0	0	1	0	1	0	1	1	0	1	0	x	x	x	x	x	P	P	u	0	0	-	-	-	0	1	0	d	d	d	d	d	Vd.tmp=vmem(Rx++Mu):nt		
0	0	1	0	1	0	1	1	1	0	0	x	x	x	x	x	P	P	u	v	v	-	-	-	1	1	0	d	d	d	d	d	if (Pv) Vd.tmp=vmem(Rx++Mu)		
0	0	1	0	1	0	1	1	1	0	0	x	x	x	x	x	P	P	u	v	v	-	-	-	1	1	1	d	d	d	d	d	if (IPv) Vd.tmp=vmem(Rx++Mu)		
0	0	1	0	1	0	1	1	1	1	0	x	x	x	x	x	P	P	u	v	v	-	-	-	1	1	0	d	d	d	d	d	if (Pv) Vd.tmp=vmem(Rx++Mu):nt		
0	0	1	0	1	0	1	1	1	1	0	x	x	x	x	x	P	P	u	v	v	-	-	-	1	1	1	d	d	d	d	d	if (IPv) Vd.tmp=vmem(Rx++Mu):nt		

Field name	Description
ICLASS	Instruction class
NT	Nontemporal
Parse	Packet loop parse bits
d5	Field to encode register d
t5	Field to encode register t
u1	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x

Load - unaligned

Read a full vector register Vd from memory, using an arbitrary byte-aligned address. The operation has three ways to generate the memory pointer address: Rt with a constant 4-bit signed offset, Rx with a 3-bit signed post-increment, and Rx with a modifier register Mu post-increment. For the immediate forms, the value indicates the number of vectors worth of data. Mu contains the actual byte offset. Unaligned memory operations require two accesses to the memory system, and thus incur increased power and bandwidth over aligned accesses. However, they require fewer instructions.

It is more efficient to use aligned memory operations when possible, and sometimes multiple aligned memory accesses and the `valign` operation, to synthesize a non-aligned access.

This instruction uses both slot 0 and slot 1, allowing at most three instructions to execute in a packet with `vmemu` in it.

Syntax	Behavior
Vd=vmemu (Rt)	Assembler mapped to: "Vd=vmemu (Rt+#0) "
Vd=vmemu (Rt+#s4)	EA=Rt+#s*VBYTES; Vd = *EA;
Vd=vmemu (Rx++#s3)	EA=Rx; Vd = *EA; Rx=Rx+#s*VBYTES;
Vd=vmemu (Rx++Mu)	EA=Rx; Vd = *EA; Rx=Rx+MuV;

Class: COPROC_VMEM (slots 0)

Notes

- This instruction uses the HVX permute resource.
- Immediates used in address computation are specified in multiples of vector length.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS									NT		t5					Parse									d5							
0	0	1	0	1	0	0	0	0	0	0	t	t	t	t	t	P	P	i	0	0	i	i	i	1	1	1	d	d	d	d	d	Vd=vmemu(Rt+#s4)
ICLASS									NT		x5					Parse									d5							
0	0	1	0	1	0	0	1	0	0	0	x	x	x	x	x	P	P	-	0	0	i	i	i	1	1	1	d	d	d	d	d	Vd=vmemu(Rx++#s3)
ICLASS									NT		x5					Parse									d5							
0	0	1	0	1	0	1	1	0	0	0	x	x	x	x	x	P	P	u	0	0	-	-	-	1	1	1	d	d	d	d	d	Vd=vmemu(Rx++Mu)

Field name	Description
ICLASS	Instruction class
NT	Nontemporal
Parse	Packet loop parse bits

Field name		Description
d5	Field to encode register d	
t5	Field to encode register t	
u1	Field to encode register u	
x5	Field to encode register x	

6.8 HVX MPY double resource

The HVX MPY double resource instruction subclass includes memory load instructions.

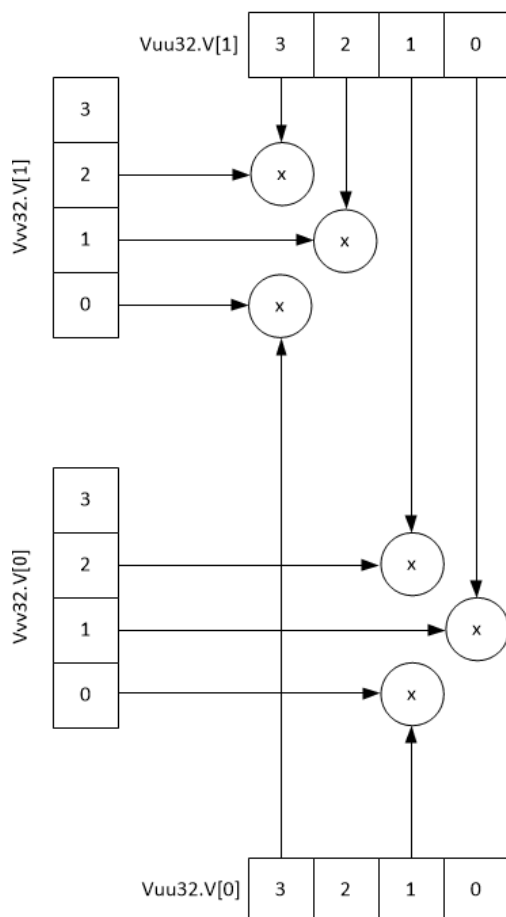
3×3 multiply for 2×2 tile

Multiply optimized for 3×3 filtering for a 2×2 tile format of two horizontal by two vertical bytes with three 10-bit coefficients. Byte 4 of the inlane word in the coefficient vector specifies the upper two bits for each coefficient.

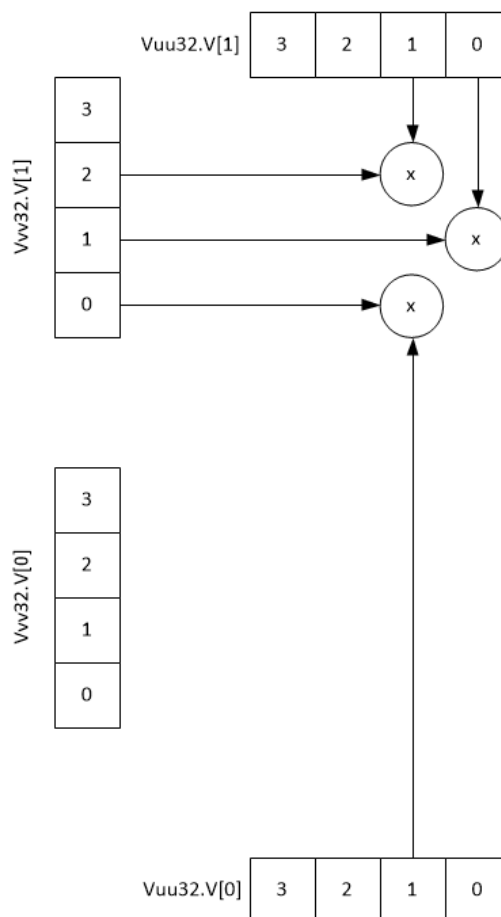
To simplify the diagrams, accumulation is not shown, but all multiplies are assumed to reduce and accumulate with the corresponding output.

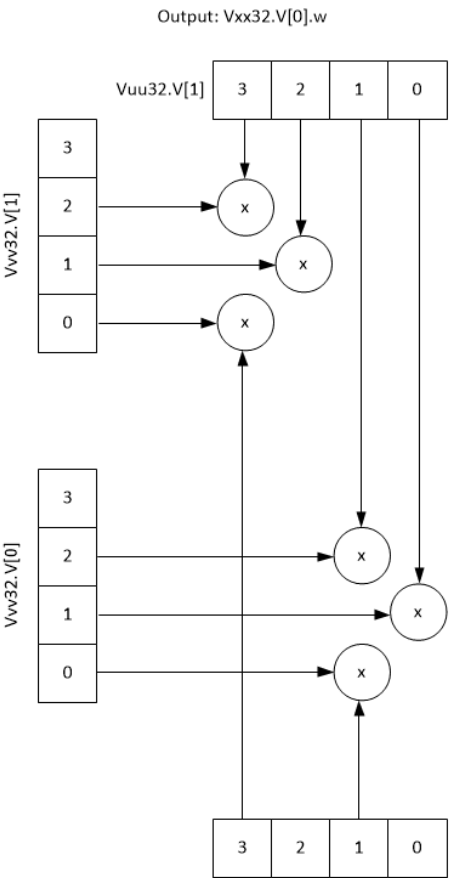
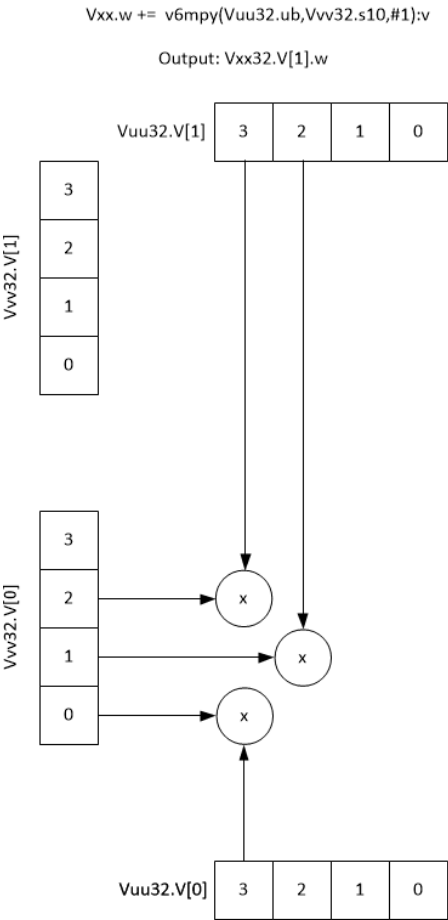
`Vxx.w += v6mpy(Vuu32.ub,Vvv32.s10,#0):v`

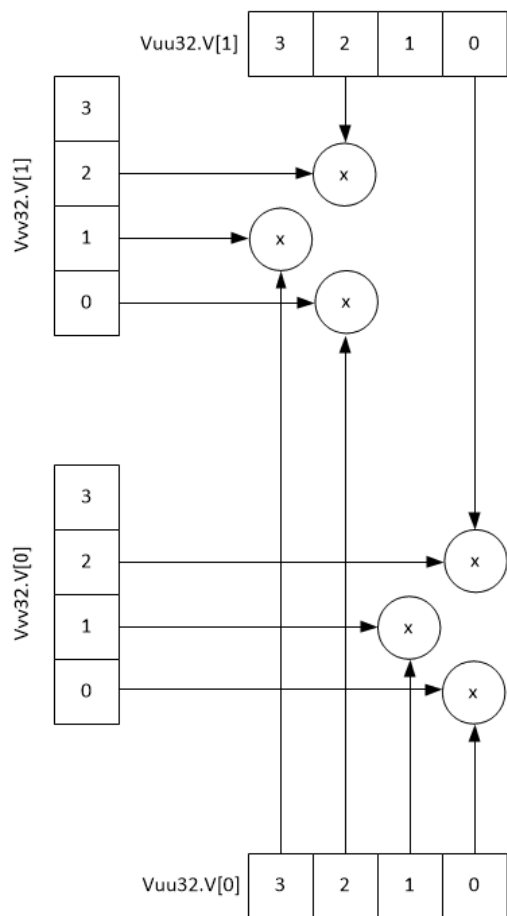
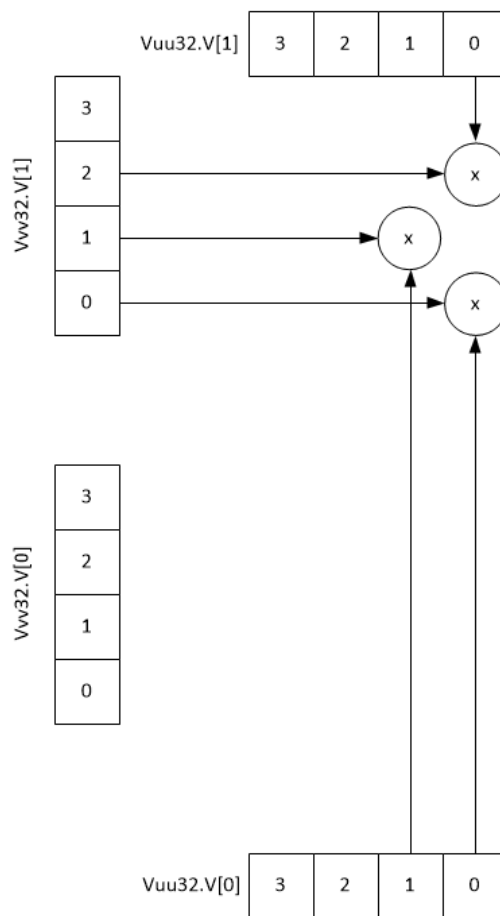
Output: Vxx32.V[1].w

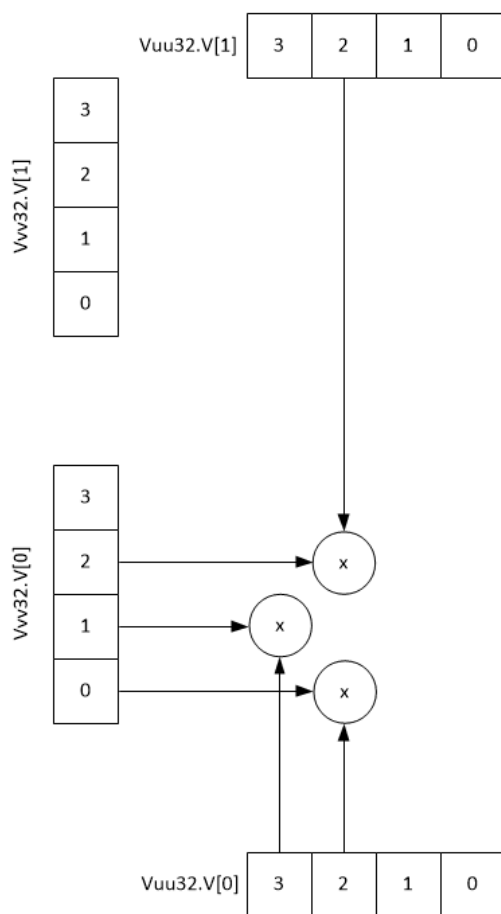
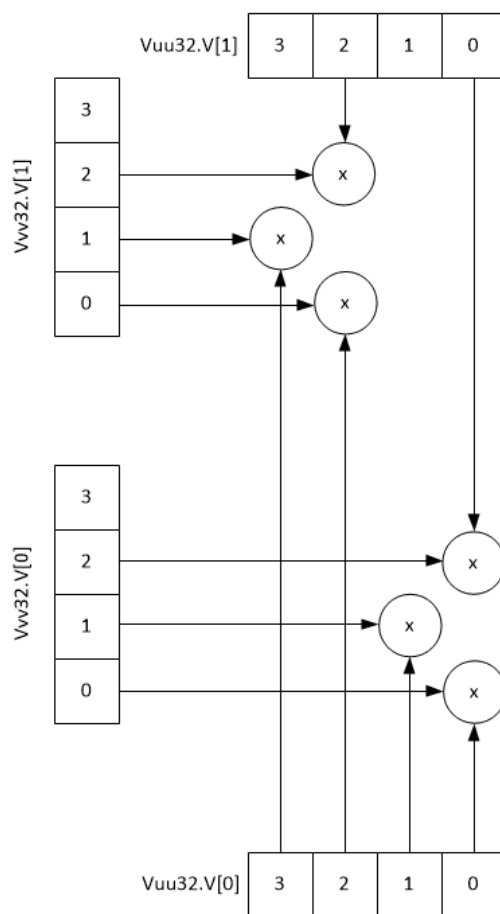


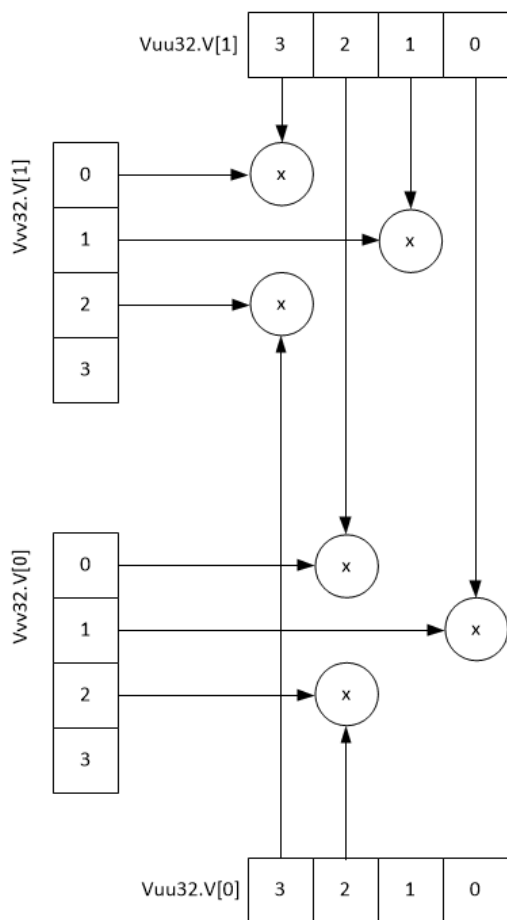
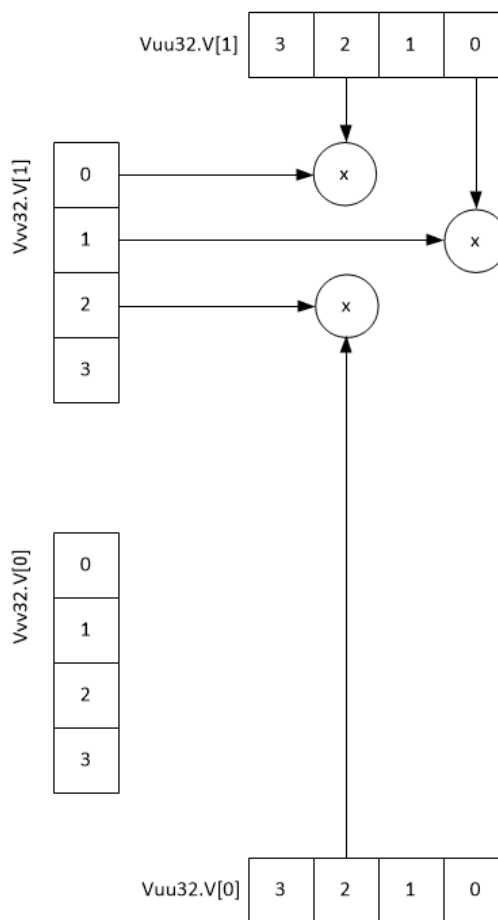
Output: Vxx32.V[0].w

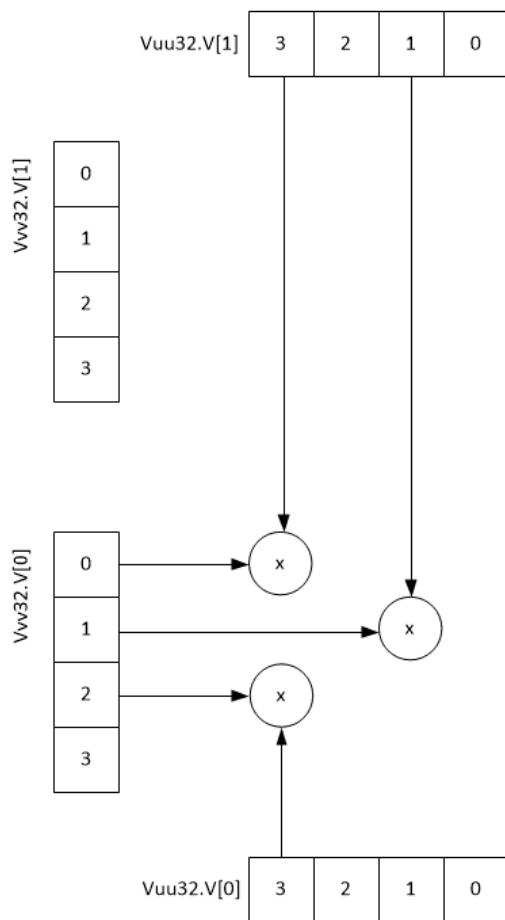
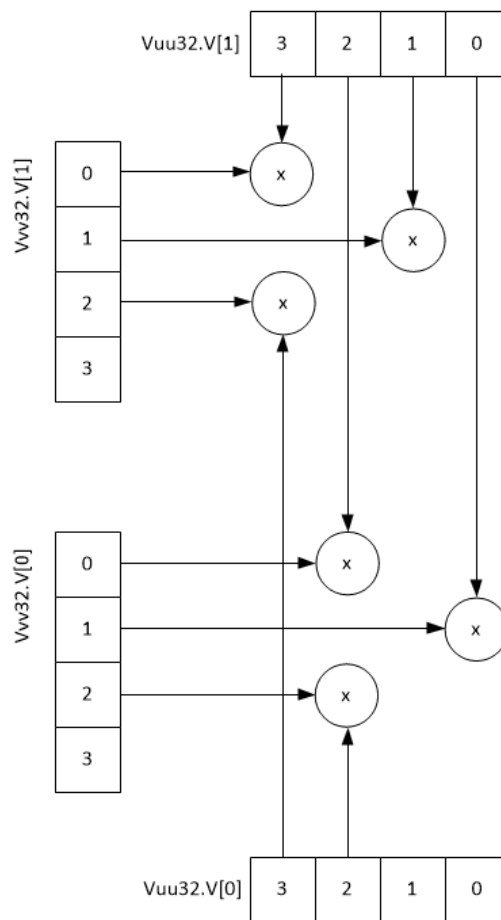


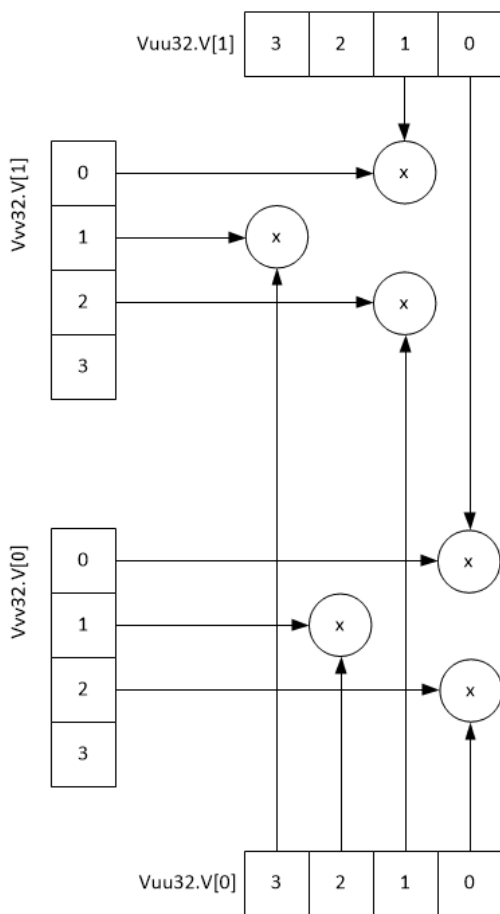
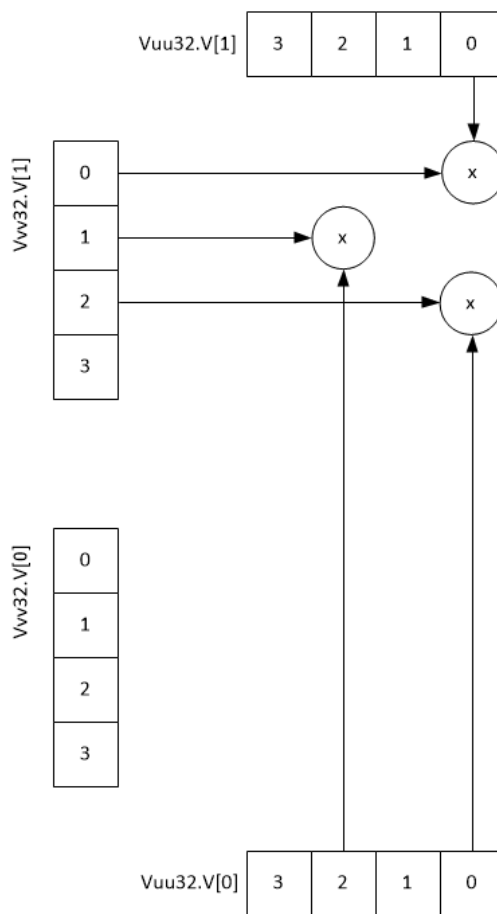


$$V_{xx}.w += v6mpy(V_{uu}32.ub, V_{vv}32.s10, \#2):v$$
Output: $V_{xx}32.V[1].w$ Output: $V_{xx}32.V[0].w$ 

$$V_{xx}.w += v6mpy(V_{uu}32.ub, V_{vv}32.s10, \#3); v$$
Output: $V_{xx}32.V[1].w$ Output: $V_{xx}32.V[0].w$ 

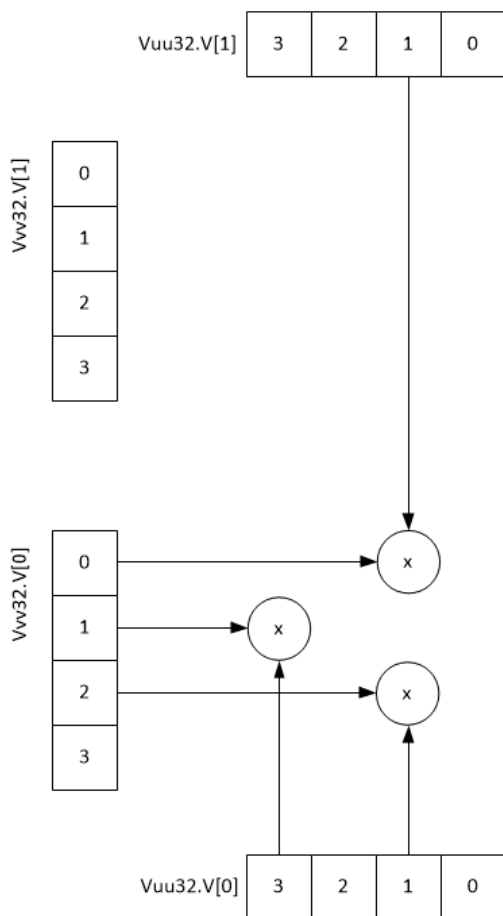
$$V_{xx}.w += v6mpy(V_{uu}32.ub, V_{vv}32.s10, \#0):h$$
Output: $V_{xx}32.V[1].w$ Output: $V_{xx}32.V[0].w$ 

$$Vxx.w += v6mpy(Vuu32.ub, Vvv32.s10, \#1):h$$
Output: $Vxx32.V[1].w$ Output: $Vxx32.V[0].w$ 

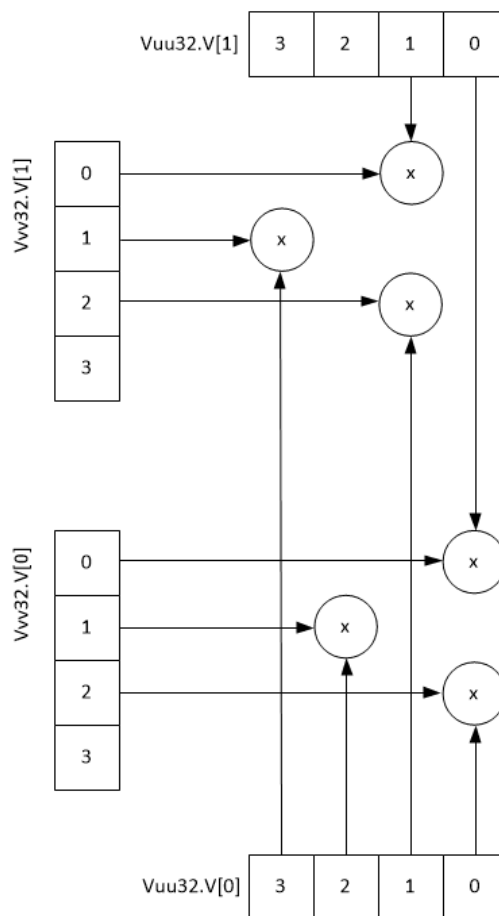
$$V_{xx}.w += v6mpy(V_{uu32}.ub, V_{vv32}.s10, \#2):h$$
Output: $V_{xx32}.V[1].w$ Output: $V_{xx32}.V[0].w$ 

Vxx.w += v6mpy(Vuu32.ub,Vvv32.s10,#3):h

Output: Vxx32.V[1].w



Output: Vxx32.V[0].w



Syntax

Vdd.w=v6mpy(Vuu.ub,Vvv.b,#u2):h

Behavior

```
for (i = 0; i < VELEM(32); i++) {
    c00=(((Vvv.v[0].uw[i].ub[3] >> (2 * 0)) & 3)
    << 8) | Vvv.v[0].uw[i].ub[0]) << 6) >> 6;
    c01=(((Vvv.v[0].uw[i].ub[3] >> (2 * 1)) & 3)
    << 8) | Vvv.v[0].uw[i].ub[1]) << 6) >> 6;
    c02=(((Vvv.v[0].uw[i].ub[3] >> (2 * 2)) & 3) <<
    8) | Vvv.v[0].uw[i].ub[2])<< 6) >> 6;
    c10=(((Vvv.v[1].uw[i].ub[3] >> (2 * 0)) & 3)
    << 8) | Vvv.v[1].uw[i].ub[0]) << 6) >> 6;
    c11=(((Vvv.v[1].uw[i].ub[3] >> (2 * 1)) & 3)
    << 8) | Vvv.v[1].uw[i].ub[1]) << 6) >> 6;
    c12=(((Vvv.v[1].uw[i].ub[3] >> (2 * 2)) & 3)
    << 8) | Vvv.v[1].uw[i].ub[2]) << 6) >> 6;
    if (#u == 0) {
        Vdd.v[1].w[i] = (Vuu.v[1].uw[i].ub[3] * c10);
        Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[1] * c11);
        Vdd.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c12);
        Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[2] * c00);
        Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[0] * c01);
    }
}
```

Syntax

```
Vdd.w=v6mpy(Vuu.ub,Vvv.b,#u2):h
(cont)
```

Behavior

```
Vdd.v[1].w[i] += (Vuu.v[0].uw[i].ub[2] * c02);
Vdd.v[0].w[i] = (Vuu.v[1].uw[i].ub[2] * c10);
Vdd.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] * c11);
Vdd.v[0].w[i] += (Vuu.v[0].uw[i].ub[2] * c12);
} else if (#u == 1) {
    Vdd.v[1].w[i] = (Vuu.v[1].uw[i].ub[3] * c00);
    Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[1] * c01);
    Vdd.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c02);
    Vdd.v[0].w[i] = (Vuu.v[1].uw[i].ub[3] * c10);
    Vdd.v[0].w[i] += (Vuu.v[1].uw[i].ub[1] * c11);
    Vdd.v[0].w[i] += (Vuu.v[0].uw[i].ub[3] * c12);
    Vdd.v[0].w[i] += (Vuu.v[1].uw[i].ub[2] * c00);
    Vdd.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] * c01);
    Vdd.v[0].w[i] += (Vuu.v[0].uw[i].ub[2] * c02);
} else if (#u == 2) {
    Vdd.v[1].w[i] = (Vuu.v[1].uw[i].ub[1] * c10);
    Vdd.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c11);
    Vdd.v[1].w[i] += (Vuu.v[0].uw[i].ub[1] * c12);
    Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[0] * c00);
    Vdd.v[1].w[i] += (Vuu.v[0].uw[i].ub[2] * c01);
    Vdd.v[1].w[i] += (Vuu.v[0].uw[i].ub[0] * c02);
    Vdd.v[0].w[i] = (Vuu.v[1].uw[i].ub[0] * c10);
    Vdd.v[0].w[i] += (Vuu.v[0].uw[i].ub[2] * c11);
    Vdd.v[0].w[i] += (Vuu.v[0].uw[i].ub[0] * c12);
} else if (#u == 3) {
    Vdd.v[1].w[i] = (Vuu.v[1].uw[i].ub[1] *
c00);
Vdd.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c01);
    Vdd.v[1].w[i] += (Vuu.v[0].uw[i].ub[1] * c02);
    Vdd.v[0].w[i] = (Vuu.v[1].uw[i].ub[1] * c10);
    Vdd.v[0].w[i] += (Vuu.v[0].uw[i].ub[3] * c11);
Vdd.v[0].w[i] += (Vuu.v[0].uw[i].ub[1] * c12);
    Vdd.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] *
c00);
    Vdd.v[0].w[i] += (Vuu.v[0].uw[i].ub[2] *
c01);
    Vdd.v[0].w[i] += (Vuu.v[0].uw[i].ub[0] *
c02);
};
}
```

```
Vdd.w=v6mpy(Vuu.ub,Vvv.b,#u2):v
```

```
for (i = 0; i < VELEM(32); i++) {
    c00=((((Vvv.v[0].uw[i].ub[3] >> (2 * 0)) & 3)
<< 8) | Vvv.v[0].uw[i].ub[0]) << 6) >> 6;
    c01=((((Vvv.v[0].uw[i].ub[3] >> (2 * 1)) & 3)
<< 8) | Vvv.v[0].uw[i].ub[1]) << 6) >> 6;
    c02=((((Vvv.v[0].uw[i].ub[3] >> (2 * 2)) & 3)
<< 8) | Vvv.v[0].uw[i].ub[2]) << 6) >> 6;
    c10=((((Vvv.v[1].uw[i].ub[3] >> (2 * 0)) & 3)
<< 8) | Vvv.v[1].uw[i].ub[0]) << 6) >> 6;
    c11=((((Vvv.v[1].uw[i].ub[3] >> (2 * 1)) & 3)
<< 8) | Vvv.v[1].uw[i].ub[1]) << 6) >> 6;
    c12=((((Vvv.v[1].uw[i].ub[3] >> (2 * 2)) & 3)
<< 8) | Vvv.v[1].uw[i].ub[2]) << 6) >> 6;
```

Syntax

```
Vdd.w=v6mpy(Vuu.ub,Vvv.b,#u2):v
(cont.)
```

Behavior

```
if (#u == 0) {
    Vdd.v[1].w[i] = (Vuu.v[0].uw[i].ub[3] * c10);
    Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[2] * c11);
    Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[3] * c12);
    Vdd.v[1].w[i] += (Vuu.v[0].uw[i].ub[1] * c00);
    Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[0] * c01);
    Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[1] * c02);
    Vdd.v[0].w[i] = (Vuu.v[0].uw[i].ub[1] * c10);
    Vdd.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] * c11);
    Vdd.v[0].w[i] += (Vuu.v[1].uw[i].ub[1] * c12);
} else if (#u == 1) {
    Vdd.v[1].w[i] = (Vuu.v[0].uw[i].ub[3] * c00);
    Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[2] * c01);
    Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[3] * c02);
    Vdd.v[0].w[i] = (Vuu.v[0].uw[i].ub[3] * c10);
    Vdd.v[0].w[i] += (Vuu.v[1].uw[i].ub[2] * c11);
    Vdd.v[0].w[i] += (Vuu.v[1].uw[i].ub[3] * c12);
    Vdd.v[0].w[i] += (Vuu.v[0].uw[i].ub[1] * c00);
    Vdd.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] * c01);
    Vdd.v[0].w[i] += (Vuu.v[1].uw[i].ub[1] * c02);
} else if (#u == 2) {
    Vdd.v[1].w[i] = (Vuu.v[0].uw[i].ub[2] * c10);
    Vdd.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c11);
    Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[2] * c12);
    Vdd.v[1].w[i] += (Vuu.v[0].uw[i].ub[0] * c00);
    Vdd.v[1].w[i] += (Vuu.v[0].uw[i].ub[1] * c01);
    Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[0] * c02);
    Vdd.v[0].w[i] = (Vuu.v[0].uw[i].ub[0] * c10);
    Vdd.v[0].w[i] += (Vuu.v[0].uw[i].ub[1] * c11);
    Vdd.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] * c12);
} else if (#u == 3) {
    Vdd.v[1].w[i] = (Vuu.v[0].uw[i].ub[2] * c00);
    Vdd.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c01);
    Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[2] * c02);
    Vdd.v[0].w[i] = (Vuu.v[0].uw[i].ub[2] * c10);
    Vdd.v[0].w[i] += (Vuu.v[0].uw[i].ub[3] * c11);
    Vdd.v[0].w[i] += (Vuu.v[1].uw[i].ub[2] * c12);
    Vdd.v[0].w[i] += (Vuu.v[0].uw[i].ub[0] * c00);
    Vdd.v[0].w[i] += (Vuu.v[0].uw[i].ub[1] * c01);
    Vdd.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] * c02);
};
}
```

```
Vxx.w+=v6mpy(Vuu.ub,Vvv.b,#u2):h
```

```
for (i = 0; i < VELEM(32); i++) {
    c00=(((Vvv.v[0].uw[i].ub[3] >> (2 * 0)) & 3)
    << 8) | Vvv.v[0].uw[i].ub[0]) << 6) >> 6;
    c01=(((Vvv.v[0].uw[i].ub[3] >> (2 * 1)) & 3)
    << 8) | Vvv.v[0].uw[i].ub[1]) << 6) >> 6;
    c02=(((Vvv.v[0].uw[i].ub[3] >> (2 * 2)) & 3)
    << 8) | Vvv.v[0].uw[i].ub[2]) << 6) >> 6;
    c10=(((Vvv.v[1].uw[i].ub[3] >> (2 * 0)) & 3)
    << 8) | Vvv.v[1].uw[i].ub[0]) << 6) >> 6;
    c11=(((Vvv.v[1].uw[i].ub[3] >> (2 * 1)) & 3)
    << 8) | Vvv.v[1].uw[i].ub[1]) << 6) >> 6;
    c12=(((Vvv.v[1].uw[i].ub[3] >> (2 * 2)) & 3)
    << 8) | Vvv.v[1].uw[i].ub[2]) << 6) >> 6;
```

Syntax

```
Vxx.w+=v6mpy(Vuu.ub,Vvv.b,#u2):h
(cont)
```

Behavior

```
if (#u == 0) {
    Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[3] * c10);
    Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[1] * c11);
    Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c12);
    Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[2] * c00);
    Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[0] * c01);
    Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[2] * c02);
    Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[2] * c10);
    Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] * c11);
    Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[2] * c12);
} else if (#u == 1) {
    Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[3] * c00);
    Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[1] * c01);
    Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c02);
    Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[3] * c10);
    Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[1] * c11);
    Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[3] * c12);
    Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[2] * c00);
    Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] * c01);
    Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[2] * c02);
} else if (#u == 2) {
    Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[1] * c10);
    Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c11);
    Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[1] * c12);
    Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[0] * c00);
    Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[2] * c01);
    Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[0] * c02);
    Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] * c10);
    Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[2] * c11);
    Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[0] * c12);
} else if (#u == 3) {
    Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[1] * c00);
    Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c01);
    Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[1] * c02);
    Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[1] * c10);
    Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[3] * c11);
    Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[1] * c12);
    Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] * c00);
    Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[2] * c01);
    Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[0] * c02);
};
}
```

```
Vxx.w+=v6mpy(Vuu.ub,Vvv.b,#u2):v
```

```
for (i = 0; i < VELEM(32); i++) {
    c00=(((Vvv.v[0].uw[i].ub[3] >> (2 * 0)) & 3)
    << 8) | Vvv.v[0].uw[i].ub[0] << 6) >> 6;
    c01=(((Vvv.v[0].uw[i].ub[3] >> (2 * 1)) & 3)
    << 8) | Vvv.v[0].uw[i].ub[1] << 6) >> 6;
    c02=(((Vvv.v[0].uw[i].ub[3] >> (2 * 2)) & 3)
    << 8) | Vvv.v[0].uw[i].ub[2] << 6) >> 6;
    c10=(((Vvv.v[1].uw[i].ub[3] >> (2 * 0)) & 3) << 8)
    | Vvv.v[1].uw[i].ub[0] << 6) >> 6;
    c11=(((Vvv.v[1].uw[i].ub[3] >> (2 * 1)) & 3)
    << 8) | Vvv.v[1].uw[i].ub[1] << 6) >> 6;
    c12=(((Vvv.v[1].uw[i].ub[3] >> (2 * 2)) & 3)
    << 8) | Vvv.v[1].uw[i].ub[2] << 6) >> 6;
```

Syntax	Behavior
<pre>Vxx.w+=v6mpy(Vuu.ub,Vvv.b,#u2):v (cont)</pre>	<pre>if (#u == 0) { Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c10); Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[2] * c11); Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[3] * c12); Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[1] * c00); Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[0] * c01); Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[1] * c02); Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[1] * c10); Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] * c11); Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[1] * c12); } else if (#u == 1) { Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c00); Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[2] * c01); Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[3] * c02); Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[3] * c10); Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[2] * c11); Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[3] * c12); Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[1] * c00); Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] * c01); Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[1] * c02); } else if (#u == 2) { Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[2] * c10); Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c11); Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[2] * c12); Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[0] * c00); Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[1] * c01); Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[0] * c02); Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[0] * c10); Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[1] * c11); Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] * c12); } else if (#u == 3) { Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[2] * c00); Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c01); Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[2] * c02); Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[2] * c10); Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[3] * c11); Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[2] * c12); Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[0] * c00); Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[1] * c01); Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] * c02); }; }</pre>

Class: COPROC_VX (slots 2,3)**Notes**

- This instruction uses both HVX multiply resources.
- The accumulator (Vxx) source of this instruction must be generate in the previous packet to avoid a stall. The accumulator cannot come from a .tmp operation.

Intrinsics

<code>Vdd.w=v6mpy(Vuu.ub,Vvv.b,#u2):h</code>	HVX_VectorPair Q6_Ww_v6mpy_WubWbI_h(HVX_VectorPair Vuu, HVX_VectorPair Vvv, Word32 Iu2)
<code>Vdd.w=v6mpy(Vuu.ub,Vvv.b,#u2):v</code>	HVX_VectorPair Q6_Ww_v6mpy_WubWbI_v(HVX_VectorPair Vuu, HVX_VectorPair Vvv, Word32 Iu2)
<code>Vxx.w+=v6mpy(Vuu.ub,Vvv.b,#u2):h</code>	HVX_VectorPair Q6_Ww_v6mpyacc_WwWubWbI_h(HVX_VectorPair Vxx, HVX_VectorPair Vuu, HVX_VectorPair Vvv, Word32 Iu2)
<code>Vxx.w+=v6mpy(Vuu.ub,Vvv.b,#u2):v</code>	HVX_VectorPair Q6_Ww_v6mpyacc_WwWubWbI_v(HVX_VectorPair Vxx, HVX_VectorPair Vuu, HVX_VectorPair Vvv, Word32 Iu2)

Encoding

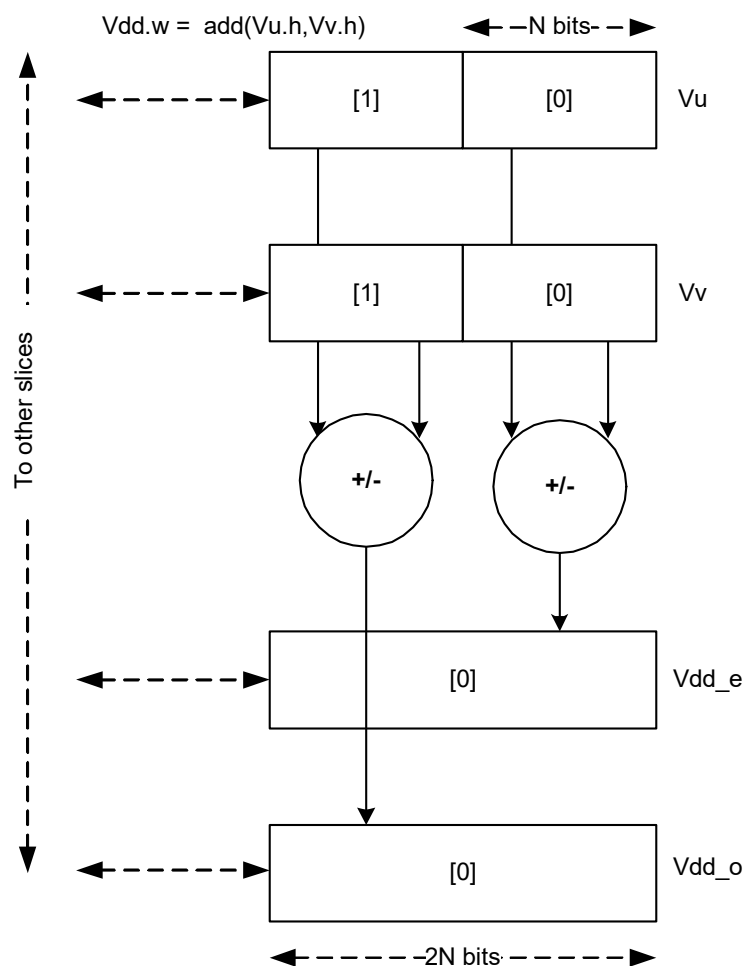
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS															Parse		u5										x5					
0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	i	i	x	x	x	x	x	Vxx.w+=v6mpy(Vuu.ub,Vvv.b,#u2):v
0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	i	i	x	x	x	x	x	Vxx.w+=v6mpy(Vuu.ub,Vvv.b,#u2):h
ICLASS															Parse		u5										d5					
0	0	0	1	1	1	1	1	0	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	i	i	d	d	d	d	d	Vdd.w=v6mpy(Vuu.ub,Vvv.b,#u2):v
0	0	0	1	1	1	1	1	0	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	i	i	d	d	d	d	d	Vdd.w=v6mpy(Vuu.ub,Vvv.b,#u2):h

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v
x5	Field to encode register x

Arithmetic widening

Add or subtract the elements of vector registers Vu and Vv. The resulting elements are double the width of the input size to capture any data growth in the result. The result is placed in a double vector register.

Supports unsigned byte, and signed and unsigned halfword.



Syntax

```
Vdd.h=vadd(Vu.ub,Vv.ub)
```

```
Vdd.h=vsub(Vu.ub,Vv.ub)
```

Behavior

```
for (i = 0; i < VELEM(16); i++) {
    Vdd.v[0].h[i] = Vu.ub[i].ub[0] + Vv.ub[i].ub[0];
    Vdd.v[1].h[i] = Vu.ub[i].ub[1] + Vv.ub[i].ub[1];
}
```

```
for (i = 0; i < VELEM(16); i++) {
    Vdd.v[0].h[i] = Vu.ub[i].ub[0] - Vv.ub[i].ub[0];
    Vdd.v[1].h[i] = Vu.ub[i].ub[1] - Vv.ub[i].ub[1];
}
```

Syntax	Behavior
<code>Vdd.w=vadd(Vu.h,Vv.h)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vdd.v[0].w[i] = Vu.w[i].h[0] + Vv.w[i].h[0]; Vdd.v[1].w[i] = Vu.w[i].h[1] + Vv.w[i].h[1]; }</pre>
<code>Vdd.w=vadd(Vu.uh,Vv.uh)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vdd.v[0].w[i] = Vu.uw[i].uh[0] + Vv.uw[i].uh[0]; Vdd.v[1].w[i] = Vu.uw[i].uh[1] + Vv.uw[i].uh[1]; }</pre>
<code>Vdd.w=vsub(Vu.h,Vv.h)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vdd.v[0].w[i] = Vu.w[i].h[0] - Vv.w[i].h[0]; Vdd.v[1].w[i] = Vu.w[i].h[1] - Vv.w[i].h[1]; }</pre>
<code>Vdd.w=vsub(Vu.uh,Vv.uh)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vdd.v[0].w[i] = Vu.uw[i].uh[0] - Vv.uw[i].uh[0]; Vdd.v[1].w[i] = Vu.uw[i].uh[1] - Vv.uw[i].uh[1]; }</pre>
<code>Vxx.h+=vadd(Vu.ub,Vv.ub)</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vxx.v[0].h[i] += Vu.h[i].ub[0] + Vv.h[i].ub[0]; Vxx.v[1].h[i] += Vu.h[i].ub[1] + Vv.h[i].ub[1]; }</pre>
<code>Vxx.w+=vadd(Vu.h,Vv.h)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vxx.v[0].w[i] += Vu.w[i].h[0] + Vv.w[i].h[0]; Vxx.v[1].w[i] += Vu.w[i].h[1] + Vv.w[i].h[1]; }</pre>
<code>Vxx.w+=vadd(Vu.uh,Vv.uh)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vxx.v[0].w[i] += Vu.w[i].uh[0] + Vv.w[i].uh[0]; Vxx.v[1].w[i] += Vu.w[i].uh[1] + Vv.w[i].uh[1]; }</pre>

Class: COPROC_VX (slots 2,3)**Notes**

- This instruction uses both HVX multiply resources.

Intrinsics

Vdd.h=vadd(Vu.ub,Vv.ub)	HVX_VectorPair Q6_Wh_vadd_VubVub(HVX_Vector Vu, HVX_Vector Vv)
Vdd.h=vsub(Vu.ub,Vv.ub)	HVX_VectorPair Q6_Wh_vsub_VubVub(HVX_Vector Vu, HVX_Vector Vv)
Vdd.w=vadd(Vu.h,Vv.h)	HVX_VectorPair Q6_Ww_vadd_VhVh(HVX_Vector Vu, HVX_Vector Vv)
Vdd.w=vadd(Vu.uh,Vv.uh)	HVX_VectorPair Q6_Ww_vadd_VuhVuh(HVX_Vector Vu, HVX_Vector Vv)
Vdd.w=vsub(Vu.h,Vv.h)	HVX_VectorPair Q6_Ww_vsub_VhVh(HVX_Vector Vu, HVX_Vector Vv)
Vdd.w=vsub(Vu.uh,Vv.uh)	HVX_VectorPair Q6_Ww_vsub_VuhVuh(HVX_Vector Vu, HVX_Vector Vv)
Vxx.h+=vadd(Vu.ub,Vv.ub)	HVX_VectorPair Q6_Wh_vaddacc_WhVubVub(HVX_VectorPair Vxx, HVX_Vector Vu, HVX_Vector Vv)
Vxx.w+=vadd(Vu.h,Vv.h)	HVX_VectorPair Q6_Ww_vaddacc_WwVhVh(HVX_VectorPair Vxx, HVX_Vector Vu, HVX_Vector Vv)
Vxx.w+=vadd(Vu.uh,Vv.uh)	HVX_VectorPair Q6_Ww_vaddacc_WwVuhVuh(HVX_VectorPair Vxx, HVX_Vector Vu, HVX_Vector Vv)

Encoding

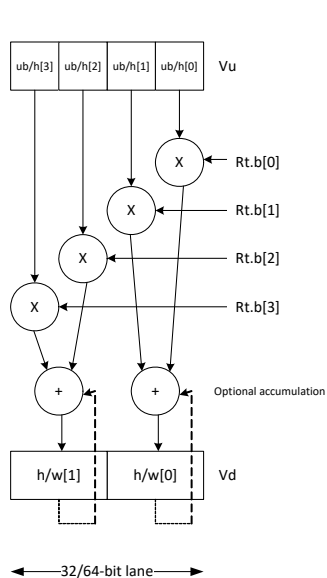
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS															Parse		u5										x5					
0	0	0	1	1	1	0	0	0	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	0	x	x	x	x	x	Vxx.w+=vadd(Vu.h,Vv.h)
0	0	0	1	1	1	0	0	0	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	x	x	x	x	x	Vxx.w+=vadd(Vu.uh,Vv.uh)
0	0	0	1	1	1	0	0	0	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	1	x	x	x	x	x	Vxx.h+=vadd(Vu.ub,Vv.ub)
ICLASS															Parse		u5										d5					
0	0	0	1	1	1	0	0	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vdd.h=vadd(Vu.ub,Vv.ub)
0	0	0	1	1	1	0	0	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vdd.w=vadd(Vu.uh,Vv.uh)
0	0	0	1	1	1	0	0	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vdd.w=vadd(Vu.h,Vv.h)
0	0	0	1	1	1	0	0	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vdd.h=vsub(Vu.ub,Vv.ub)
0	0	0	1	1	1	0	0	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vdd.w=vsub(Vu.uh,Vv.uh)
0	0	0	1	1	1	0	0	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vdd.w=vsub(Vu.h,Vv.h)

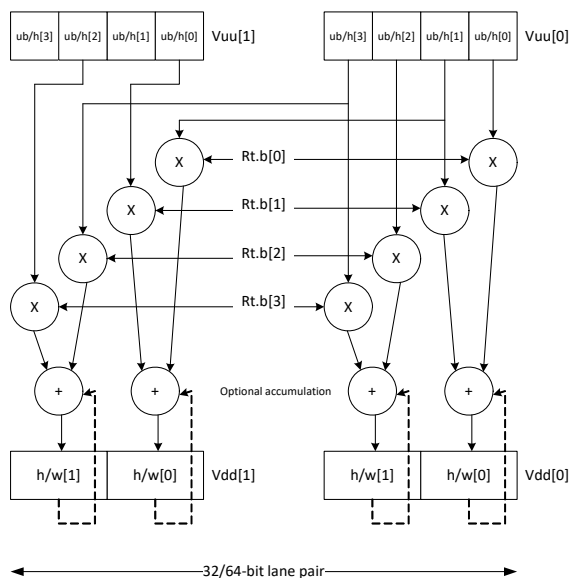
Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v
x5	Field to encode register x

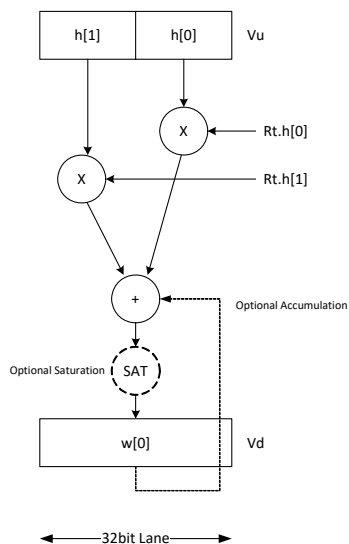
Multiply with 2-wide reduction

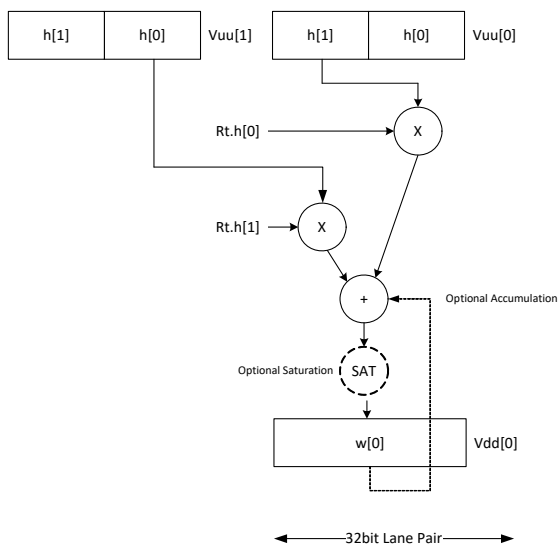
Multiply elements from Vu by the corresponding elements in the scalar register Rt. The products are added in pairs to yield a by-2 reduction. The products can optionally be accumulated with Vx, with optional saturation after summation.

Supports multiplication of unsigned bytes by bytes, halfwords by signed bytes, and halfwords by halfwords. The double-vector version performs a sliding window 2-way reduction, where the odd register output contains the offset computation.

$$Vd.h[+] = \text{vdmpy}(Vu.ub, Rt.b) / Vd.w[+] = \text{vdmpy}(Vu.h, Rt.b)$$


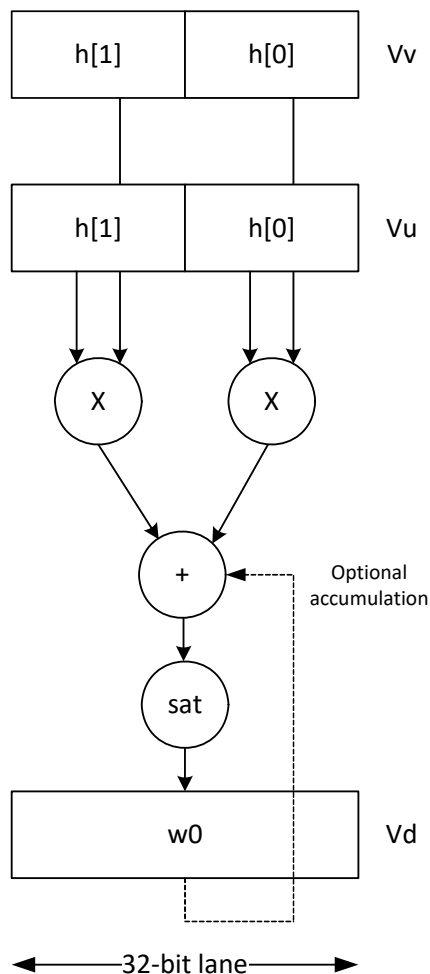
$$Vdd.h[+] = \text{vdmpy}(Vuu.ub, Rt.b) / Vdd.w[+] = \text{vdmpy}(Vuu.h, Rt.b)$$


$$Vd.w[+] = \text{vdmpy}(Vu.h, Rt.h) : \text{sat}$$


$$Vdd.w[+] = \text{vdmpy}(Vuu.h, Rt.h) : \text{sat}$$


Multiply halfword elements from vector register Vu by the corresponding halfword elements in the vector register Vv. The products are added in pairs to make a 32-bit wide sum. The sum is optionally accumulated with the vector register destination Vx, and then saturated to 32 bits.

$Vd.w[+] = vdmpy(Vu.h, Vv.h) : sat$



Syntax

$Vd.w = vdmpy(Vuu.h, Rt.h) : sat$

$Vd.w = vdmpy(Vuu.h, Rt.uh, \#1) : sat$

Behavior

```
for (i = 0; i < VELEM(32); i++) {
    accum = (Vuu.v[0].w[i].h[1] * Rt.h[0]);
    accum += (Vuu.v[1].w[i].h[0] * Rt.h[1]);
    Vd.w[i] = sat32(accum);
}
```

```
for (i = 0; i < VELEM(32); i++) {
    accum = (Vuu.v[0].w[i].h[1] * Rt.uh[0]);
    accum += (Vuu.v[1].w[i].h[0] * Rt.uh[1]);
    Vd.w[i] = sat32(accum);
}
```

Syntax	Behavior
<code>Vdd.h = vdmpy(Vuu.ub,Rt.b)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vdd.v[0].h[i] = (Vuu.v[0].uh[i].ub[0] * Rt.b[(2*i) % 4]); Vdd.v[0].h[i] += (Vuu.v[0].uh[i].ub[1] * Rt.b[(2*i+1)%4]); Vdd.v[1].h[i] = (Vuu.v[0].uh[i].ub[1] * Rt.b[(2*i) % 4]); Vdd.v[1].h[i] += (Vuu.v[1].uh[i].ub[0] * Rt.b[(2*i+1)%4]); } </pre>
<code>Vdd.w = vdmpy(Vuu.h,Rt.b)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vdd.v[0].w[i] = (Vuu.v[0].w[i].h[0] * Rt.b[(2*i+0)%4]); Vdd.v[0].w[i] += (Vuu.v[0].w[i].h[1] * Rt.b[(2*i+1)%4]); Vdd.v[1].w[i] = (Vuu.v[0].w[i].h[1] * Rt.b[(2*i+0)%4]); Vdd.v[1].w[i] += (Vuu.v[1].w[i].h[0] * Rt.b[(2*i+1)%4]); } </pre>
<code>Vx.w += vdmpy(Vu.h,Vv.h):sat</code>	<pre> for (i = 0; i < VELEM(32); i++) { accum = (Vx.w[i].h[0] * Vv.w[i].h[0]); accum += (Vu.w[i].h[1] * Vv.w[i].h[1]); Vx.w[i] = sat₃₂(Vx.w[i]+accum); } </pre>
<code>Vx.w += vdmpy(Vuu.h,Rt.h):sat</code>	<pre> for (i = 0; i < VELEM(32); i++) { accum = Vx.w[i]; accum += (Vuu.v[0].w[i].h[1] * Rt.h[0]); accum += (Vuu.v[1].w[i].h[0] * Rt.h[1]); Vx.w[i] = sat₃₂(accum); } </pre>
<code>Vx.w += vdmpy(Vuu.h, Rt.uh, #1):sat</code>	<pre> for (i = 0; i < VELEM(32); i++) { accum=Vx.w[i]; accum += (Vuu.v[0].w[i].h[1] * Rt.uh[0]); accum += (Vuu.v[1].w[i].h[0] * Rt.uh[1]); Vx.w[i] = sat₃₂(accum); } </pre>
<code>Vxx.h+=vdmpy(Vuu.ub,Rt.b)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vxx.v[0].h[i] += (Vuu.v[0].uh[i].ub[0] * Rt.b[(2*i) % 4]); Vxx.v[0].h[i] += (Vuu.v[0].uh[i].ub[1] * Rt.b[(2*i+1)%4]); Vxx.v[1].h[i] += (Vuu.v[0].uh[i].ub[1] * Rt.b[(2*i) % 4]); Vxx.v[1].h[i] += (Vuu.v[1].uh[i].ub[0] * Rt.b[(2*i+1)%4]); } </pre>

Syntax	Behavior
<code>Vxx.w+=vdmpy(Vuu.h,Rt.b)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vxx.v[0].w[i] += (Vuu.v[0].w[i].h[0] * Rt.b[(2*i+0)%4]); Vxx.v[0].w[i] += (Vuu.v[0].w[i].h[1] * Rt.b[(2*i+1)%4]); Vxx.v[1].w[i] += (Vuu.v[0].w[i].h[1] * Rt.b[(2*i+0)%4]); Vxx.v[1].w[i] += (Vuu.v[1].w[i].h[0] * Rt.b[(2*i+1)%4]); } </pre>

Class: COPROC_VX (slots 2,3)**Notes**

- This instruction uses both HVX multiply resources.

Intrinsics

<code>Vd.w=vdmpy(Vuu.h,Rt.h):sat</code>	<code>HVX_Vector Q6_Vw_vdmpy_WhRh_sat(HVX_VectorPair Vuu, Word32 Rt)</code>
<code>Vd.w=vdmpy(Vuu.h,Rt.uh,#1):sat</code>	<code>HVX_Vector Q6_Vw_vdmpy_WhRuh_sat(HVX_VectorPair Vuu, Word32 Rt)</code>
<code>Vdd.h=vdmpy(Vuu.ub,Rt.b)</code>	<code>HVX_VectorPair Q6_Wh_vdmpy_WubRb(HVX_VectorPair Vuu, Word32 Rt)</code>
<code>Vdd.w=vdmpy(Vuu.h,Rt.b)</code>	<code>HVX_VectorPair Q6_Ww_vdmpy_WhRb(HVX_VectorPair Vuu, Word32 Rt)</code>
<code>Vx.w+=vdmpy(Vu.h,Vv.h):sat</code>	<code>HVX_Vector Q6_Vw_vdmpyacc_VwVhVh_sat(HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vx.w+=vdmpy(Vuu.h,Rt.h):sat</code>	<code>HVX_Vector Q6_Vw_vdmpyacc_VwWhRh_sat(HVX_Vector Vx, HVX_VectorPair Vuu, Word32 Rt)</code>
<code>Vx.w += vdmpy(Vuu.h,Rt.uh,#1):sat</code>	<code>HVX_Vector Q6_Vw_vdmpyacc_VwWhRuh_sat(HVX_Vector Vx, HVX_VectorPair Vuu, Word32 Rt)</code>
<code>Vxx.h+=vdmpy(Vuu.ub,Rt.b)</code>	<code>HVX_VectorPair Q6_Wh_vdmpyacc_WhWubRb(HVX_VectorPair Vxx, HVX_VectorPair Vuu, Word32 Rt)</code>
<code>Vxx.w+=vdmpy(Vuu.h,Rt.b)</code>	<code>HVX_VectorPair Q6_Ww_vdmpyacc_WwWhRb(HVX_VectorPair Vxx, HVX_VectorPair Vuu, Word32 Rt)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											t5					Parse		u5										d5				
0	0	0	1	1	0	0	1	0	0	0	t	t	t	t	t	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vdd.h+=vdmpy(Vuu.ub,Rt.b)
ICLASS											t5					Parse		u5										x5				
0	0	0	1	1	0	0	1	0	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	1	1	1	x	x	x	x	x	Vxx.h+=vdmpy(Vuu.ub,Rt.b)
ICLASS											t5					Parse		u5										d5				
0	0	0	1	1	0	0	1	0	0	1	t	t	t	t	t	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.w+=vdmpy(Vuu.h,Rt.uh,#1):sat
0	0	0	1	1	0	0	1	0	0	1	t	t	t	t	t	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.w+=vdmpy(Vuu.h,Rt.h):sat
0	0	0	1	1	0	0	1	0	0	1	t	t	t	t	t	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vdd.w+=vdmpy(Vuu.h,Rt.b)
ICLASS											t5					Parse		u5										x5				
0	0	0	1	1	0	0	1	0	0	1	t	t	t	t	t	P	P	1	u	u	u	u	u	0	0	1	x	x	x	x	x	Vx.w+=vdmpy(Vuu.h,Rt.uh,#1):sat
0	0	0	1	1	0	0	1	0	0	1	t	t	t	t	t	P	P	1	u	u	u	u	u	0	1	0	x	x	x	x	x	Vx.w+=vdmpy(Vuu.h,Rt.h):sat
0	0	0	1	1	0	0	1	0	0	1	t	t	t	t	t	P	P	1	u	u	u	u	u	1	0	0	x	x	x	x	x	Vxx.w+=vdmpy(Vuu.h,Rt.b)
ICLASS																Parse		u5										x5				
0	0	0	1	1	1	0	0	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	x	x	x	x	x	Vx.w+=vdmpy(Vu.h,Vv.h):sat

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t5	Field to encode register t
u5	Field to encode register u
v5	Field to encode register v
x5	Field to encode register x

Lookup table for piecewise from 64-bit scalar

The `vlut4` instruction implements a four entry lookup table that is specified in scalar register pair, `Rtt`.

Syntax

```
Vd.h=vlut4(Vu.uh,Rtt.h)
```

Behavior

```
for (i = 0; i < VELEM(16); i++) {
    Vd.h[i]= Rtt.h[ ((Vu.h[i]>>14)&0x3) ];
}
```

Class: COPROC_VX (slots 2)

Notes

- This instruction uses both HVX multiply resources.

Intrinsics

```
Vd.h=vlut4(Vu.uh,Rtt.h)
```

```
HVX_Vector Q6_Vh_vlut4_VuhPh(HVX_Vector Vu, Word64
Rtt)
```

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											t5				Parse		u5								d5							
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.h=vlut4(Vu.uh,Rtt.h)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t5	Field to encode register t
u5	Field to encode register u

Multiply with piecewise add/sub from 64-bit scalar

Instructions to help nonlinear function calculations.

Syntax	Behavior
<code>Vx.h=vmpa(Vx.h,Vu.h,Rtt.h):sat</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vx.h[i]= sat₁₆(((Vx.h[i] * Vu.h[i])<<1) + (Rtt.h[(Vu.h[i]>>14)&0x3]<<15))>>16); }</pre>
<code>Vx.h=vmpa(Vx.h,Vu.uh,Rtt.uh):sat</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vx.h[i]= sat₁₆((Vx.h[i] * Vu.uh[i]) + (Rtt.uh[(Vu.uh[i]>>14)&0x3]<<15))>>16); }</pre>
<code>Vx.h=vmps(Vx.h,Vu.uh,Rtt.uh):sat</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vx.h[i]= sat₁₆((Vx.h[i] * Vu.uh[i]) - (Rtt.uh[(Vu.uh[i]>>14)&0x3]<<15))>>16);}</pre>

Class: COPROC_VX (slots 2)

Notes

- This instruction uses both HVX multiply resources.

Intrinsics

<code>Vx.h=vmpa(Vx.h,Vu.h,Rtt.h):sat</code>	<code>HVX_Vector Q6_Vh_vmpa_VhVhVhPh_sat(HVX_Vector Vx, HVX_Vector Vu, Word64 Rtt)</code>
<code>Vx.h=vmpa(Vx.h,Vu.uh,Rtt.uh):sat</code>	<code>HVX_Vector Q6_Vh_vmpa_VhVhVuhPuh_sat(HVX_Vector Vx, HVX_Vector Vu, Word64 Rtt)</code>
<code>Vx.h=vmps(Vx.h,Vu.uh,Rtt.uh):sat</code>	<code>HVX_Vector Q6_Vh_vmps_VhVhVuhPuh_sat(HVX_Vector Vx, HVX_Vector Vu, Word64 Rtt)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											t5				Parse		u5								x5							
0	0	0	1	1	0	0	1	1	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	1	0	0	x	x	x	x	x	Vx.h=vmpa(Vx.h,Vu.h,Rtt.h):sat
0	0	0	1	1	0	0	1	1	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	1	0	1	x	x	x	x	x	Vx.h=vmpa(Vx.h,Vu.uh,Rtt.uh):sat
0	0	0	1	1	0	0	1	1	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	1	1	0	x	x	x	x	x	Vx.h=vmps(Vx.h,Vu.uh,Rtt.uh):sat

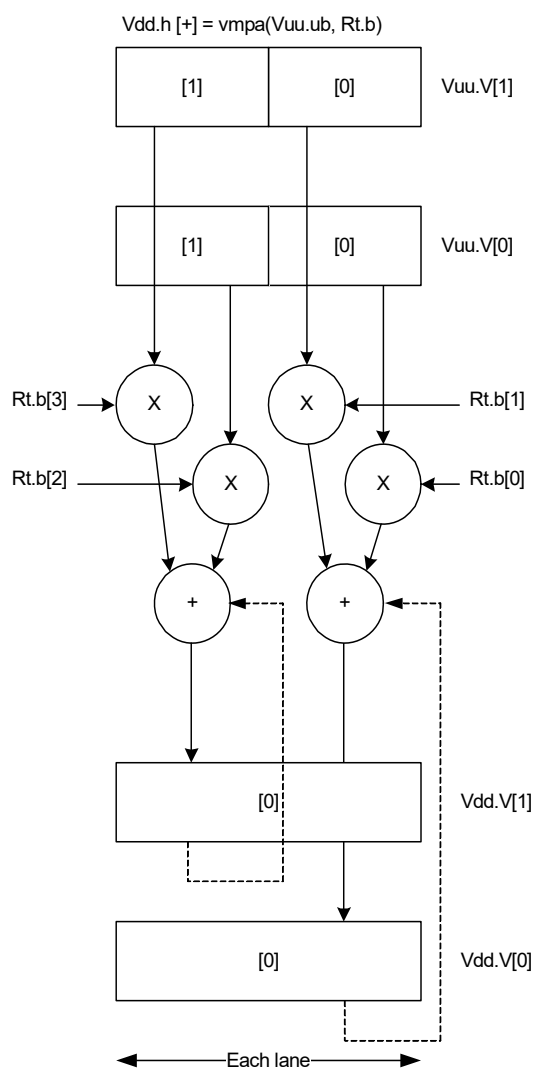
Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
t5	Field to encode register t
u5	Field to encode register u
x5	Field to encode register x

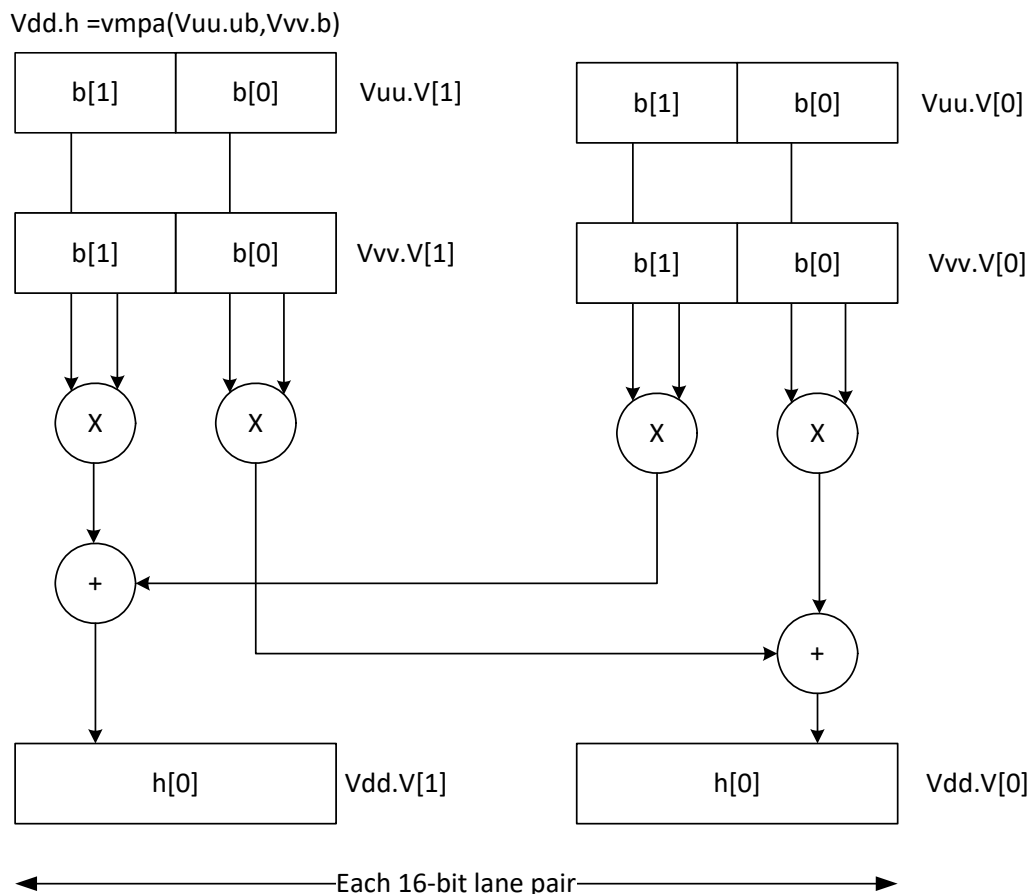
Multiply-add

Compute the sum of two byte multiplies. The two products consist of either unsigned bytes or signed halfwords coming from the vector registers Vuu and Vvv. These are multiplied by a signed byte coming from a scalar register Rt. The result of the summation is a signed halfword or word. Each corresponding pair of elements in Vuu and Vvv is weighted, using Rt.b[0] and Rt.b[1] for the even elements, and Rt.b[2] and Rt.b[3] for the odd elements.

Optionally accumulates the product with the destination vector register Vxx.

For vector by vector, compute the sum of two byte multiplies. The two products consist of an unsigned byte vector operand multiplied by a signed byte scalar. The result of the summation is a signed halfword. Even elements from the input vector register pairs Vuu and Vvv are multiplied together and placed in the even register of Vdd. Odd elements are placed in the odd register of Vdd.



**Syntax**
 $Vdd.h = vmpa(Vuu.ub, Rt.b)$
 $Vdd.h = vmpa(Vuu.ub, Rt.ub)$
 $Vdd.h = vmpa(Vuu.ub, Vvv.b)$
Behavior

```
for (i = 0; i < VELEM(16); i++) {
    Vdd.v[0].h[i] = (Vuu.v[0].uh[i].ub[0] *
Rt.b[0]) + (Vuu.v[1].uh[i].ub[0] * Rt.b[1]);
    Vdd.v[1].h[i] = (Vuu.v[0].uh[i].ub[1] *
Rt.b[2]) + (Vuu.v[1].uh[i].ub[1] * Rt.b[3]);
}
```

```
for (i = 0; i < VELEM(16); i++) {
    Vdd.v[0].uh[i] = (Vuu.v[0].uh[i].ub[0] *
Rt.ub[0]) + (Vuu.v[1].uh[i].ub[0] * Rt.ub[1]);
    Vdd.v[1].uh[i] = (Vuu.v[0].uh[i].ub[1] *
Rt.ub[2]) + (Vuu.v[1].uh[i].ub[1] * Rt.ub[3]);
}
```

```
for (i = 0; i < VELEM(16); i++) {
    Vdd.v[0].h[i] = (Vuu.v[0].uh[i].ub[0] *
Vvv.v[0].uh[i].b[0]) + (Vuu.v[1].uh[i].ub[0] *
Vvv.v[1].uh[i].b[0]);
    Vdd.v[1].h[i] = (Vuu.v[0].uh[i].ub[1] *
Vvv.v[0].uh[i].b[1]) + (Vuu.v[1].uh[i].ub[1] *
Vvv.v[1].uh[i].b[1]);
}
```

Syntax	Behavior
<code>Vdd.h=vmpa (Vuu.ub,Vvv.ub)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vdd.v[0].h[i] = (Vuu.v[0].uh[i].ub[0] * Vvv.v[0].uh[i].ub[0]) + (Vuu.v[1].uh[i].ub[0] * Vvv.v[1].uh[i].ub[0]); Vdd.v[1].h[i] = (Vuu.v[0].uh[i].ub[1] * Vvv.v[0].uh[i].ub[1]) + (Vuu.v[1].uh[i].ub[1] * Vvv.v[1].uh[i].ub[1]); } </pre>
<code>Vdd.w=vmpa (Vuu.h,Rt.b)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vdd.v[0].w[i] = (Vuu.v[0].w[i].h[0] * Rt.b[0]) + (Vuu.v[1].w[i].h[0] * Rt.b[1]); Vdd.v[1].w[i] = (Vuu.v[0].w[i].h[1] * Rt.b[2]) + (Vuu.v[1].w[i].h[1] * Rt.b[3]); } </pre>
<code>Vdd.w=vmpa (Vuu.uh,Rt.b)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vdd.v[0].w[i] = (Vuu.v[0].w[i].uh[0] * Rt.b[0]) + (Vuu.v[1].w[i].uh[0] * Rt.b[1]); Vdd.v[1].w[i] = (Vuu.v[0].w[i].uh[1] * Rt.b[2]) + (Vuu.v[1].w[i].uh[1] * Rt.b[3]); } </pre>
<code>Vxx.h+=vmpa (Vuu.ub,Rt.b)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vxx.v[0].h[i] += (Vuu.v[0].uh[i].ub[0] * Rt.b[0]) + (Vuu.v[1].uh[i].ub[0] * Rt.b[1]); Vxx.v[1].h[i] += (Vuu.v[0].uh[i].ub[1] * Rt.b[2]) + (Vuu.v[1].uh[i].ub[1] * Rt.b[3]); } </pre>
<code>Vxx.h+=vmpa (Vuu.ub,Rt.ub)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vxx.v[0].uh[i] += (Vuu.v[0].uh[i].ub[0] * Rt.ub[0]) + (Vuu.v[1].uh[i].ub[0] * Rt.ub[1]); Vxx.v[1].uh[i] += (Vuu.v[0].uh[i].ub[1] * Rt.ub[2]) + (Vuu.v[1].uh[i].ub[1] * Rt.ub[3]); } </pre>
<code>Vxx.w+=vmpa (Vuu.h,Rt.b)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vxx.v[0].w[i] += (Vuu.v[0].w[i].h[0] * Rt.b[0]) + (Vuu.v[1].w[i].h[0] * Rt.b[1]); Vxx.v[1].w[i] += (Vuu.v[0].w[i].h[1] * Rt.b[2]) + (Vuu.v[1].w[i].h[1] * Rt.b[3]); } </pre>
<code>Vxx.w+=vmpa (Vuu.uh,Rt.b)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vxx.v[0].w[i] += (Vuu.v[0].w[i].uh[0] * Rt.b[0]) + (Vuu.v[1].w[i].uh[0] * Rt.b[1]); Vxx.v[1].w[i] += (Vuu.v[0].w[i].uh[1] * Rt.b[2]) + (Vuu.v[1].w[i].uh[1] * Rt.b[3]); } </pre>

Class: COPROC_VX (slots 2,3)**Notes**

- This instruction uses both HVX multiply resources.

Intrinsics

Vdd.h=vmpa (Vuu.ub,Rt.b)	HVX_VectorPair Q6_Wh_vmpa_WubRb (HVX_VectorPair Vuu, Word32 Rt)
Vdd.h=vmpa (Vuu.ub,Rt.ub)	HVX_VectorPair Q6_Wh_vmpa_WubRub (HVX_VectorPair Vuu, Word32 Rt)
Vdd.h=vmpa (Vuu.ub,Vvv.b)	HVX_VectorPair Q6_Wh_vmpa_WubWb (HVX_VectorPair Vuu, HVX_VectorPair Vvv)
Vdd.h=vmpa (Vuu.ub,Vvv.ub)	HVX_VectorPair Q6_Wh_vmpa_WubWub (HVX_VectorPair Vuu, HVX_VectorPair Vvv)
Vdd.w=vmpa (Vuu.h,Rt.b)	HVX_VectorPair Q6_Ww_vmpa_WhRb (HVX_VectorPair Vuu, Word32 Rt)
Vdd.w=vmpa (Vuu.ub,Rt.b)	HVX_VectorPair Q6_Ww_vmpa_WuhRb (HVX_VectorPair Vuu, Word32 Rt)
Vxx.h+=vmpa (Vuu.ub,Rt.b)	HVX_VectorPair Q6_Wh_vmpaacc_WhWubRb (HVX_VectorPair Vxx, HVX_VectorPair Vuu, Word32 Rt)
Vxx.h+=vmpa (Vuu.ub,Rt.ub)	HVX_VectorPair Q6_Wh_vmpaacc_WhWubRub (HVX_VectorPair Vxx, HVX_VectorPair Vuu, Word32 Rt)
Vxx.w+=vmpa (Vuu.h,Rt.b)	HVX_VectorPair Q6_Ww_vmpaacc_WwWhRb (HVX_VectorPair Vxx, HVX_VectorPair Vuu, Word32 Rt)
Vxx.w+=vmpa (Vuu.ub,Rt.b)	HVX_VectorPair Q6_Ww_vmpaacc_WwWuhRb (HVX_VectorPair Vxx, HVX_VectorPair Vuu, Word32 Rt)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS										t5					Parse		u5										d5					
0	0	0	1	1	0	0	1	0	0	1	t	t	t	t	t	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vdd.h=vmpa(Vuu.ub,Rt.b)
0	0	0	1	1	0	0	1	0	0	1	t	t	t	t	t	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vdd.w=vmpa(Vuu.h,Rt.b)
ICLASS										t5					Parse		u5										x5					
0	0	0	1	1	0	0	1	0	0	1	t	t	t	t	t	P	P	1	u	u	u	u	u	1	1	0	x	x	x	x	x	Vxx.h+=vmpa(Vuu.ub,Rt.b)
0	0	0	1	1	0	0	1	0	0	1	t	t	t	t	t	P	P	1	u	u	u	u	u	1	1	1	x	x	x	x	x	Vxx.w+=vmpa(Vuu.h,Rt.b)
ICLASS										t5					Parse		u5										d5					
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vdd.h=vmpa(Vuu.ub,Rt.ub)
0	0	0	1	1	0	0	1	1	0	0	t	t	t	t	t	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vdd.w=vmpa(Vuu.ub,Rt.b)
ICLASS										t5					Parse		u5										x5					
0	0	0	1	1	0	0	1	1	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	0	1	0	x	x	x	x	x	Vxx.w+=vmpa(Vuu.ub,Rt.b)
0	0	0	1	1	0	0	1	1	0	1	t	t	t	t	t	P	P	1	u	u	u	u	u	1	0	0	x	x	x	x	x	Vxx.h+=vmpa(Vuu.ub,Rt.ub)
ICLASS															Parse		u5										d5					
0	0	0	1	1	1	0	0	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vdd.h=vmpa(Vuu.ub,Vvv.b)
0	0	0	1	1	1	0	0	1	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vdd.h=vmpa(Vuu.ub,Vvv.ub)

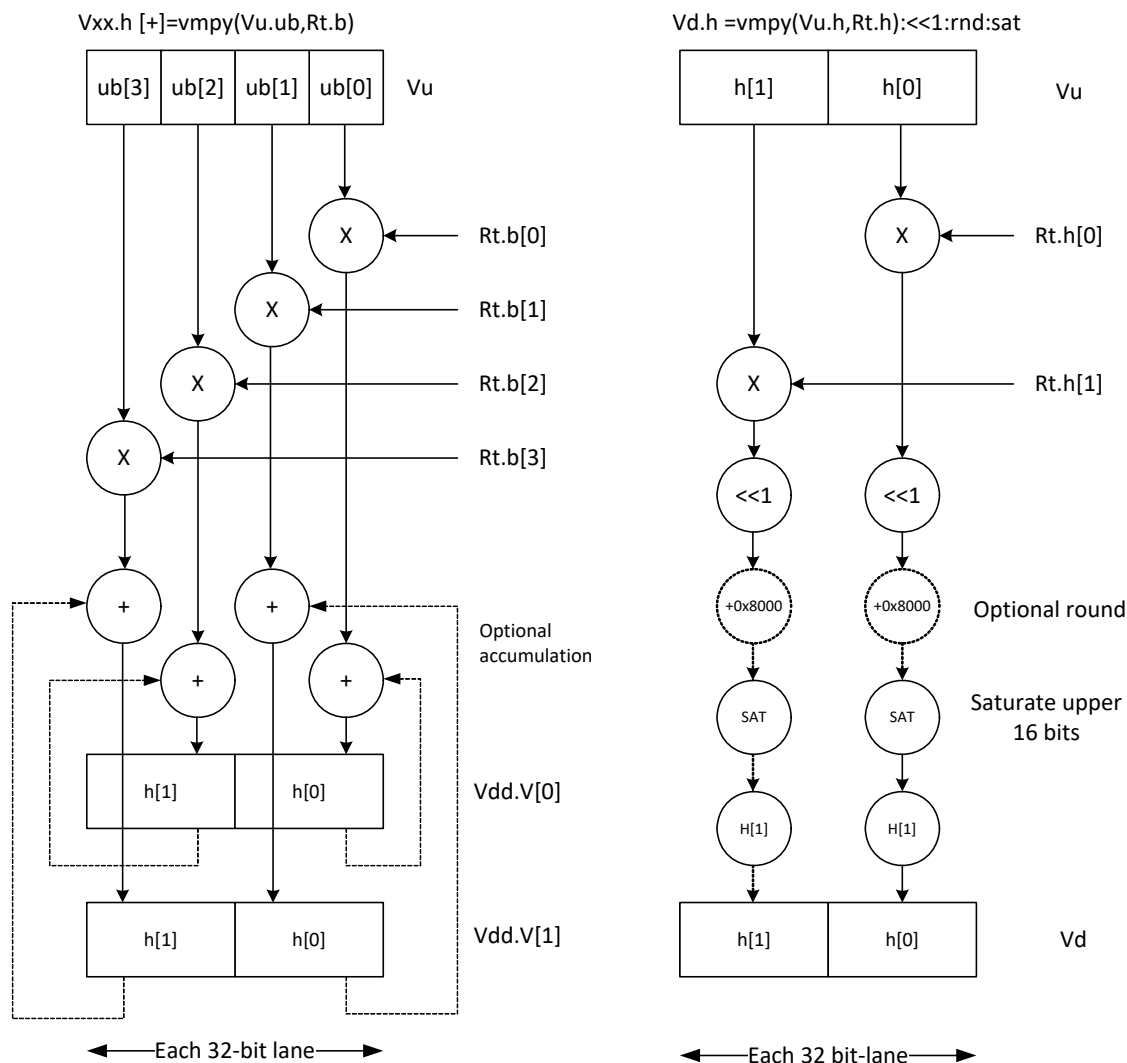
Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t5	Field to encode register t
u5	Field to encode register u
v5	Field to encode register v
x5	Field to encode register x

Multiply - vector by scalar

Multiply groups of elements in the vector Vu by the corresponding elements in the scalar register Rt.

This operation has two forms. In the first form the product is not modified, and is optionally accumulated with the destination register. The even results are placed in the even vector register of the destination register pair, while the odd results are placed in the odd vector register.

Supports signed by signed halfword, unsigned by unsigned byte, unsigned by signed byte, and unsigned halfword by unsigned halfword.



Syntax	Behavior
<code>Vdd.h=vmpy (Vu.ub,Rt.b)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vdd.v[0].h[i] = (Vu.ub[i].ub[0] * Rt.b[(2*i+0)%4]); Vdd.v[1].h[i] = (Vu.ub[i].ub[1] * Rt.b[(2*i+1)%4]); } </pre>
<code>Vdd.uh=vmpy (Vu.ub,Rt.ub)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vdd.v[0].uh[i] = (Vu.ub[i].ub[0] * Rt.ub[(2*i+0)%4]); Vdd.v[1].uh[i] = (Vu.ub[i].ub[1] * Rt.ub[(2*i+1)%4]); } </pre>
<code>Vdd.uw=vmpy (Vu.uh,Rt.uh)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vdd.v[0].uw[i] = (Vu.uh[i].uh[0] * Rt.uh[0]); Vdd.v[1].uw[i] = (Vu.uh[i].uh[1] * Rt.uh[1]); } </pre>
<code>Vdd.w=vmpy (Vu.h,Rt.h)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vdd.v[0].w[i] = (Vu.w[i].h[0] * Rt.h[0]); Vdd.v[1].w[i] = (Vu.w[i].h[1] * Rt.h[1]); } </pre>
<code>Vxx.h+=vmpy (Vu.ub,Rt.b)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vxx.v[0].h[i] += (Vu.ub[i].ub[0] * Rt.b[(2*i+0)%4]); Vxx.v[1].h[i] += (Vu.ub[i].ub[1] * Rt.b[(2*i+1)%4]); } </pre>
<code>Vxx.uh+=vmpy (Vu.ub,Rt.ub)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vxx.v[0].uh[i] += (Vu.ub[i].ub[0] * Rt.ub[(2*i+0)%4]); Vxx.v[1].uh[i] += (Vu.ub[i].ub[1] * Rt.ub[(2*i+1)%4]); } </pre>
<code>Vxx.uw+=vmpy (Vu.uh,Rt.uh)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vxx.v[0].uw[i] += (Vu.uh[i].uh[0] * Rt.uh[0]); Vxx.v[1].uw[i] += (Vu.uh[i].uh[1] * Rt.uh[1]); } </pre>
<code>Vxx.w+=vmpy (Vu.h,Rt.h)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vxx.v[0].w[i] = Vxx.v[0].w[i].s64 + (Vu.w[i].h[0] * Rt.h[0]); Vxx.v[1].w[i] = Vxx.v[1].w[i].s64 + (Vu.w[i].h[1] * Rt.h[1]); } </pre>
<code>Vxx.w+=vmpy (Vu.h,Rt.h):sat</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vxx.v[0].w[i] = sat₃₂(Vxx.v[0].w[i].s64 + (Vu.w[i].h[0] * Rt.h[0])); Vxx.v[1].w[i] = sat₃₂(Vxx.v[1].w[i].s64 + (Vu.w[i].h[1] * Rt.h[1])); } </pre>

Class: COPROC_VX (slots 2,3)**Notes**

- This instruction uses both HVX multiply resources.

Intrinsics

Vdd.h=vmpy(Vu.ub,Rt.b)	HVX_VectorPair Q6_Wh_vmpy_VubRb(HVX_Vector Vu, Word32 Rt)
Vdd.uh=vmpy(Vu.ub,Rt.ub)	HVX_VectorPair Q6_Wuh_vmpy_VubRub(HVX_Vector Vu, Word32 Rt)
Vdd.uw=vmpy(Vu.uh,Rt.uh)	HVX_VectorPair Q6_Wuw_vmpy_VuhRuh(HVX_Vector Vu, Word32 Rt)
Vdd.w=vmpy(Vu.h,Rt.h)	HVX_VectorPair Q6_Ww_vmpy_VhRh(HVX_Vector Vu, Word32 Rt)
Vxx.h+=vmpy(Vu.ub,Rt.b)	HVX_VectorPair Q6_Wh_vmpyacc_WhVubRb(HVX_VectorPair Vxx, HVX_Vector Vu, Word32 Rt)
Vxx.uh+=vmpy(Vu.ub,Rt.ub)	HVX_VectorPair Q6_Wuh_vmpyacc_WuhVubRub(HVX_VectorPair Vxx, HVX_Vector Vu, Word32 Rt)
Vxx.uw+=vmpy(Vu.uh,Rt.uh)	HVX_VectorPair Q6_Wuw_vmpyacc_WuwVuhRuh(HVX_VectorPair Vxx, HVX_Vector Vu, Word32 Rt)
Vxx.w+=vmpy(Vu.h,Rt.h)	HVX_VectorPair Q6_Ww_vmpyacc_WwVhRh(HVX_VectorPair Vxx, HVX_Vector Vu, Word32 Rt)
Vxx.w+=vmpy(Vu.h,Rt.h):sat	HVX_VectorPair Q6_Ww_vmpyacc_WwVhRh_sat(HVX_VectorPair Vxx, HVX_Vector Vu, Word32 Rt)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											t5				Parse		u5								d5							
0	0	0	1	1	0	0	1	0	0	1	t	t	t	t	t	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vdd.h=vmpy(Vu.ub,Rt.b)
ICLASS											t5				Parse		u5								x5							
0	0	0	1	1	0	0	1	0	0	1	t	t	t	t	t	P	P	1	u	u	u	u	u	1	0	1	x	x	x	x	x	Vxx.h+=vmpy(Vu.ub,Rt.b)
ICLASS											t5				Parse		u5								d5							
0	0	0	1	1	0	0	1	0	1	0	t	t	t	t	t	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vdd.w=vmpy(Vu.h,Rt.h)
0	0	0	1	1	0	0	1	0	1	0	t	t	t	t	t	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vdd.uw=vmpy(Vu.uh,Rt.uh)
ICLASS											t5				Parse		u5								x5							
0	0	0	1	1	0	0	1	0	1	0	t	t	t	t	t	P	P	1	u	u	u	u	u	0	0	0	x	x	x	x	x	Vxx.w+=vmpy(Vu.h,Rt.h):sat

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	1	1	0	0	1	0	1	0	t	t	t	t	t	P	P	1	u	u	u	u	u	0	0	1	x	x	x	x	x	Vxx.uw+=vmpy(Vu.uh,Rt.uh)
0	0	0	1	1	0	0	1	1	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	0	0	0	x	x	x	x	x	Vxx.uh+=vmpy(Vu.ub,Rt.ub)
0	0	0	1	1	0	0	1	1	0	1	t	t	t	t	t	P	P	1	u	u	u	u	u	1	1	0	x	x	x	x	x	Vxx.w+=vmpy(Vu.h,Rt.h)
ICLASS												t5				Parse		u5						d5								
0	0	0	1	1	0	0	1	1	1	0	t	t	t	t	t	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vdd.uh+=vmpy(Vu.ub,Rt.ub)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t5	Field to encode register t
u5	Field to encode register u
x5	Field to encode register x

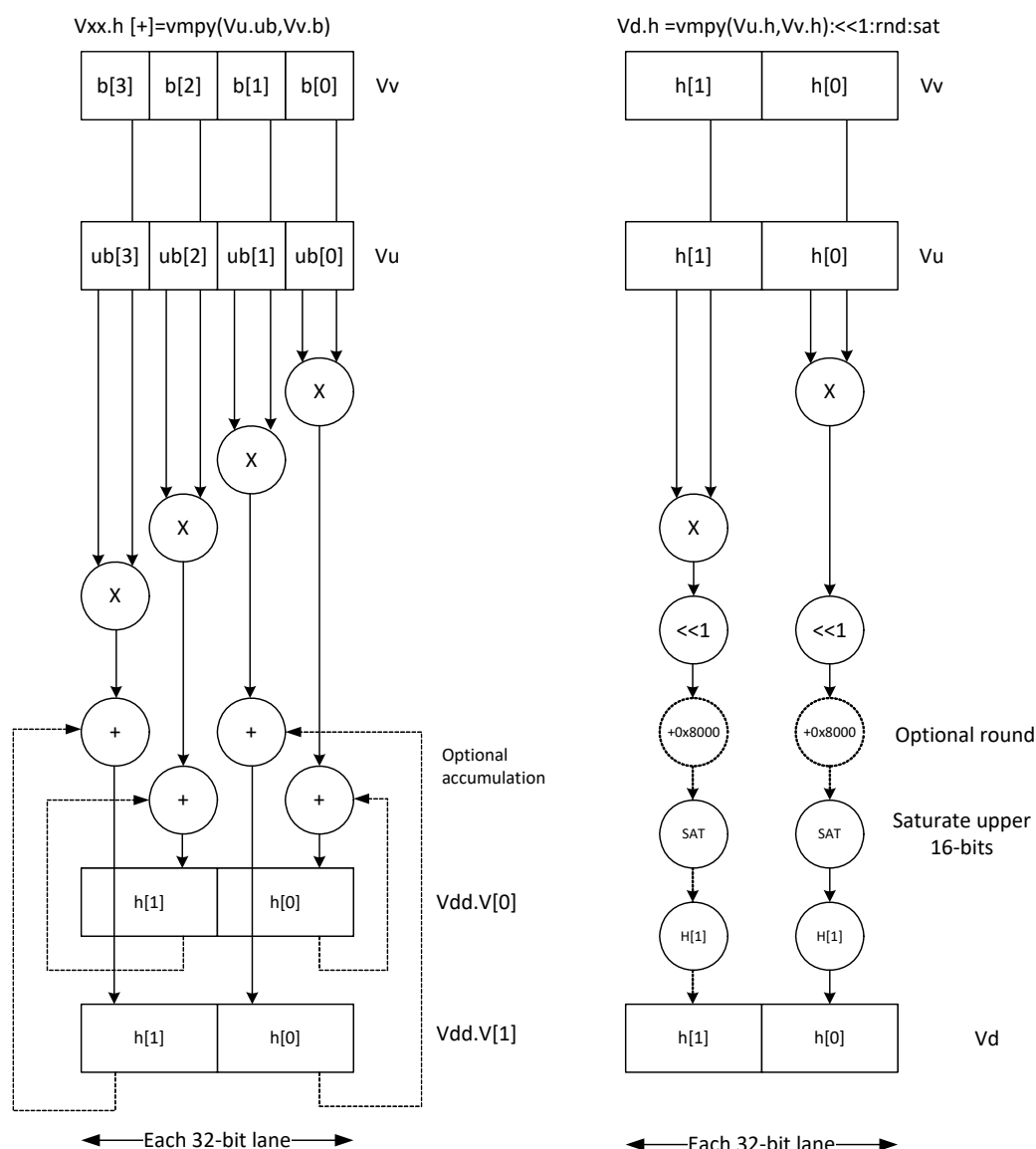
Multiply - vector by vector

Multiply groups of elements in the vector V_u by the corresponding elements in the vector register V_v .

This operation has two forms. In the first form the product is not modified, and is optionally accumulated with the destination register. The even results are placed in the even vector register of the destination register pair, while the odd results are placed in the odd vector register.

Supports signed by signed halfword, unsigned by unsigned byte, unsigned by signed byte, and unsigned halfword by unsigned halfword.

The second form of this operation keeps the output precision the same as the input width by shifting the product left by 1, saturating the product to 32 bits, and placing the upper 16 bits in the output.



Syntax	Behavior
<code>Vdd.h=vmpy(Vu.b,Vv.b)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vdd.v[0].h[i] = (Vu.h[i].b[0] * Vv.h[i].b[0]); Vdd.v[1].h[i] = (Vu.h[i].b[1] * Vv.h[i].b[1]); } </pre>
<code>Vdd.h=vmpy(Vu.ub,Vv.ub)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vdd.v[0].h[i] = (Vu.uh[i].ub[0] * Vv.h[i].b[0]); Vdd.v[1].h[i] = (Vu.uh[i].ub[1] * Vv.h[i].b[1]); } </pre>
<code>Vdd.uh=vmpy(Vu.ub,Vv.ub)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vdd.v[0].uh[i] = (Vu.uh[i].ub[0] * Vv.uh[i].ub[0]); Vdd.v[1].uh[i] = (Vu.uh[i].ub[1] * Vv.uh[i].ub[1]); } </pre>
<code>Vdd.uw=vmpy(Vu.uh,Vv.uh)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vdd.v[0].uw[i] = (Vu.uw[i].uh[0] * Vv.uw[i].uh[0]); Vdd.v[1].uw[i] = (Vu.uw[i].uh[1] * Vv.uw[i].uh[1]); } </pre>
<code>Vdd.w=vmpy(Vu.h,Vv.h)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vdd.v[0].w[i] = (Vu.w[i].h[0] * Vv.w[i].h[0]); Vdd.v[1].w[i] = (Vu.w[i].h[1] * Vv.w[i].h[1]); } </pre>
<code>Vdd.w=vmpy(Vu.h,Vv.uh)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vdd.v[0].w[i] = (Vu.w[i].h[0] * Vv.uw[i].uh[0]); Vdd.v[1].w[i] = (Vu.w[i].h[1] * Vv.uw[i].uh[1]); } </pre>
<code>Vxx.h+=vmpy(Vu.b,Vv.b)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vxx.v[0].h[i] += (Vu.h[i].b[0] * Vv.h[i].b[0]); Vxx.v[1].h[i] += (Vu.h[i].b[1] * Vv.h[i].b[1]); } </pre>
<code>Vxx.h+=vmpy(Vu.ub,Vv.b)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vxx.v[0].h[i] += (Vu.uh[i].ub[0] * Vv.h[i].b[0]); Vxx.v[1].h[i] += (Vu.uh[i].ub[1] * Vv.h[i].b[1]); } </pre>
<code>Vxx.uh+=vmpy(Vu.ub,Vv.ub)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vxx.v[0].uh[i] += (Vu.uh[i].ub[0] * Vv.uh[i].ub[0]); Vxx.v[1].uh[i] += (Vu.uh[i].ub[1] * Vv.uh[i].ub[1]); } </pre>

Syntax	Behavior
<code>Vxx.uw+=vmpy(Vu.uh,Vv.uh)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vxx.v[0].uw[i] += (Vu.uw[i].uh[0] * Vv.uw[i].uh[0]); Vxx.v[1].uw[i] += (Vu.uw[i].uh[1] * Vv.uw[i].uh[1]); }</pre>
<code>Vxx.w+=vmpy(Vu.h,Vv.h)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vxx.v[0].w[i] += (Vu.w[i].h[0] * Vv.w[i].h[0]); Vxx.v[1].w[i] += (Vu.w[i].h[1] * Vv.w[i].h[1]); }</pre>
<code>Vxx.w+=vmpy(Vu.h,Vv.uh)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vxx.v[0].w[i] += (Vu.w[i].h[0] * Vv.uw[i].uh[0]); Vxx.v[1].w[i] += (Vu.w[i].h[1] * Vv.uw[i].uh[1]); }</pre>

Class: COPROC_VX (slots 2,3)**Notes**

- This instruction uses both HVX multiply resources.

Intrinsics

<code>Vdd.h=vmpy(Vu.b,Vv.b)</code>	<code>HVX_VectorPair Q6_Wh_vmpy_VbVb(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vdd.h=vmpy(Vu.ub,Vv.b)</code>	<code>HVX_VectorPair Q6_Wh_vmpy_VubVb(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vdd.uh=vmpy(Vu.ub,Vv.ub)</code>	<code>HVX_VectorPair Q6_Wuh_vmpy_VubVub(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vdd.uw=vmpy(Vu.uh,Vv.uh)</code>	<code>HVX_VectorPair Q6_Wuw_vmpy_VuhVuh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vdd.w=vmpy(Vu.h,Vv.h)</code>	<code>HVX_VectorPair Q6_Ww_vmpy_VhVh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vdd.w=vmpy(Vu.h,Vv.uh)</code>	<code>HVX_VectorPair Q6_Ww_vmpy_VhVuh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vxx.h+=vmpy(Vu.b,Vv.b)</code>	<code>HVX_VectorPair Q6_Wh_vmpyacc_WhVbVb(HVX_VectorPair Vxx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vxx.h+=vmpy(Vu.ub,Vv.b)</code>	<code>HVX_VectorPair Q6_Wh_vmpyacc_WhVubVb(HVX_VectorPair Vxx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vxx.uh+=vmpy(Vu.ub,Vv.ub)</code>	<code>HVX_VectorPair Q6_Wuh_vmpyacc_WuhVubVub(HVX_VectorPair Vxx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vxx.uw+=vmpy(Vu.uh,Vv.uh)</code>	<code>HVX_VectorPair Q6_Wuw_vmpyacc_WuwVuhVuh(HVX_VectorPair Vxx, HVX_Vector Vu, HVX_Vector Vv)</code>

Vxx.w+=vmpy(Vu.h,Vv.h)	HVX_VectorPair Q6_Ww_vmpyacc_WwVhVh(HVX_VectorPair Vxx, HVX_Vector Vu, HVX_Vector Vv)
Vxx.w+=vmpy(Vu.h,Vv.uh)	HVX_VectorPair Q6_Ww_vmpyacc_WwVhVuh(HVX_VectorPair Vxx, HVX_Vector Vu, HVX_Vector Vv)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS															Parse		u5										d5					
0	0	0	1	1	1	0	0	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vdd.h=vmpy(Vu.b,Vv.b)
0	0	0	1	1	1	0	0	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vdd.uh=vmpy(Vu.ub,Vv.ub)
0	0	0	1	1	1	0	0	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vdd.h=vmpy(Vu.ub,Vv.b)
0	0	0	1	1	1	0	0	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vdd.w=vmpy(Vu.h,Vv.h)
ICLASS															Parse		u5										x5					
0	0	0	1	1	1	0	0	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	x	x	x	x	x	Vxx.h+=vmpy(Vu.b,Vv.b)
0	0	0	1	1	1	0	0	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	1	x	x	x	x	x	Vxx.uh+=vmpy(Vu.ub,Vv.ub)
0	0	0	1	1	1	0	0	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	0	x	x	x	x	x	Vxx.h+=vmpy(Vu.ub,Vv.b)
0	0	0	1	1	1	0	0	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	x	x	x	x	x	Vxx.w+=vmpy(Vu.h,Vv.h)
ICLASS															Parse		u5										d5					
0	0	0	1	1	1	0	0	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vdd.uw=vmpy(Vu.uh,Vv.uh)
0	0	0	1	1	1	0	0	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vdd.w=vmpy(Vu.h,Vv.uh)
ICLASS															Parse		u5										x5					
0	0	0	1	1	1	0	0	0	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	x	x	x	x	x	Vxx.uw+=vmpy(Vu.uh,Vv.uh)
0	0	0	1	1	1	0	0	0	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	x	x	x	x	x	Vxx.w+=vmpy(Vu.h,Vv.uh)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v
x5	Field to encode register x

Multiply - half precision vector by vector

These instructions perform a vectorized single precision floating point multiply. The inputs are either both IEEE single precision, both 16-bit Qfloat, or a combination of each. The result is either 16-bit Qfloat vector or a double vector of widened 32-bit Qfloat.

Syntax	Behavior
<code>Vd.qf16=vmpy(Vu.hf,Vv.hf)</code>	<pre> for (i = 0; i < VELEM(16); i++) { u = Vu.hf[i]; v = Vv.hf[i]; Vd.qf16[i] = rnd_sat(u.exp + v.exp,u.sig*v.sig,0); if(u.sign^v.sign) Vd.qf16[i] = - (Vd.qf16[i]); } </pre>
<code>Vd.qf16=vmpy(Vu.qf16,Vv.hf)</code>	<pre> for (i = 0; i < VELEM(16); i++) { u = Vu.qf16[i]; v = Vv.hf[i]; Vd.qf16[i] = rnd_sat(u.exp + v.exp,u.sig*v.sig,0); if(v.sign) Vd.qf16[i] = -(Vd.qf16[i]); } </pre>
<code>Vd.qf16=vmpy(Vu.qf16,Vv.qf16)</code>	<pre> for (i = 0; i < VELEM(16); i++) { u = Vu.qf16[i]; v = Vv.qf16[i]; Vd.qf16[i] = rnd_sat(u.exp + v.exp,u.sig*v.sig,0); } </pre>
<code>Vdd.qf32=vmpy(Vu.hf,Vv.hf)</code>	<pre> for (i = 0; i < VELEM(32); i++) { u0 = Vu.w[i] & 0xFFFF; u1 = (Vu.w[i]>>16) & 0xFFFF; v0 = Vv.w[i] & 0xFFFF; v1 = (Vv.w[i]>>16) & 0xFFFF; Vdd.v[0].qf32[i] = rnd_sat(u0.exp + v0.exp,u0.sig*v0.sig,0); Vdd.v[1].qf32[i] = rnd_sat(u1.exp + v1.exp,u1.sig*v1.sig,0); if(u0.sign^v0.sign) Vdd.v[0].qf32[i] = - (Vdd.v[0].qf32[i]); if(u1.sign^v1.sign) Vdd.v[1].qf32[i] = - (Vdd.v[1].qf32[i]); } </pre>

Syntax	Behavior
<code>Vdd.qf32=vmpy(Vu.qf16,Vv.hf)</code>	<pre> for (i = 0; i < VELEM(32); i++) { u0 = Vu.w[i] & 0xFFFF; u1 = (Vu.w[i]>>16) & 0xFFFF; v0 = Vv.w[i] & 0xFFFF; v1 = (Vv.w[i]>>16) & 0xFFFF; Vdd.v[0].qf32[i] = rnd_sat(u0.exp + v0.exp,u0.sig*v0.sig,0); Vdd.v[1].qf32[i] = rnd_sat(u1.exp + v1.exp, u1.sig*v1.sig,0); if(v0.sign) Vdd.v[0].qf32[i] = - (Vdd.v[0].qf32[i]); if(v1.sign) Vdd.v[1].qf32[i] = - (Vdd.v[1].qf32[i]); } </pre>
<code>Vdd.qf32=vmpy(Vu.qf16,Vv.qf16)</code>	<pre> for (i = 0; i < VELEM(32); i++) { u0 = Vu.w[i] & 0xFFFF; u1 = (Vu.w[i]>>16) & 0xFFFF; v0 = Vv.w[i] & 0xFFFF; v1 = (Vv.w[i]>>16) & 0xFFFF; Vdd.v[0].qf32[i] = rnd_sat(u0.exp + v0.exp,u0.sig*v0.sig,0); Vdd.v[1].qf32[i] = rnd_sat(u1.exp + v1.exp,u1.sig*v1.sig,0); } </pre>

Class: COPROC_VX (slots 2,3)**Notes**

- This instruction uses both HVX multiply resources.

Intrinsics

<code>Vd.qf16=vmpy(Vu.hf,Vv.hf)</code>	<code>HVX_Vector Q6_Vqf16_vmpy_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.qf16=vmpy(Vu.qf16,Vv.hf)</code>	<code>HVX_Vector Q6_Vqf16_vmpy_Vqf16Vhf(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.qf16=vmpy(Vu.qf16,Vv.qf16)</code>	<code>HVX_Vector Q6_Vqf16_vmpy_Vqf16Vqf16(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vdd.qf32=vmpy(Vu.hf,Vv.hf)</code>	<code>HVX_VectorPair Q6_Wqf32_vmpy_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vdd.qf32=vmpy(Vu.qf16,Vv.hf)</code>	<code>HVX_VectorPair Q6_Wqf32_vmpy_Vqf16Vhf(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vdd.qf32=vmpy(Vu.qf16,Vv.qf16)</code>	<code>HVX_VectorPair Q6_Wqf32_vmpy_Vqf16Vqf16(HVX_Vector Vu, HVX_Vector Vv)</code>

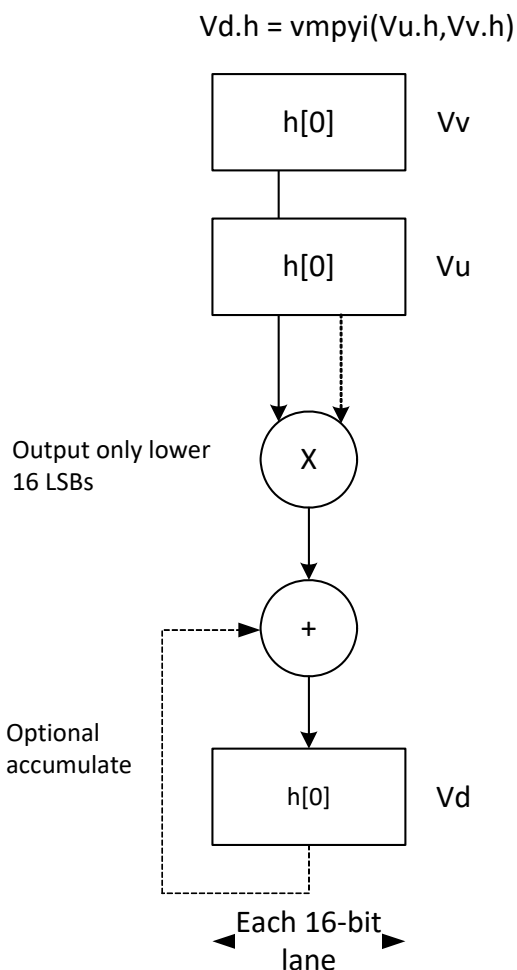
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
ICLASS																Parse		u5										d5					
0	0	0	1	1	1	1	1	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	d	d	d	d	d	Vdd.qf32=vmpy(Vu.qf16,Vv.hf)	
0	0	0	1	1	1	1	1	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.qf16=vmpy(Vu.qf16,Vv.qf16)	
0	0	0	1	1	1	1	1	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.qf16=vmpy(Vu.hf,Vv.hf)	
0	0	0	1	1	1	1	1	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.qf16=vmpy(Vu.qf16,Vv.hf)	
0	0	0	1	1	1	1	1	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	0	d	d	d	d	d	Vdd.qf32=vmpy(Vu.qf16,Vv.qf16)	
0	0	0	1	1	1	1	1	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	d	d	d	d	d	Vdd.qf32=vmpy(Vu.hf,Vv.hf)	

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Integer multiply - vector by vector

Multiply corresponding elements in V_u by the corresponding elements in V_v , and place the lower half of the result in the destination vector register V_d . Supports signed halfwords, and optional accumulation of the product with the destination vector register V_x .



Syntax	Behavior
$V_d.h = \text{vmpyi}(V_u.h, V_v.h)$	<pre>for (i = 0; i < VELEM(16); i++) { Vd.h[i] = (Vu.h[i] * Vv.h[i]); }</pre>
$V_x.h += \text{vmpyi}(V_u.h, V_v.h)$	<pre>for (i = 0; i < VELEM(16); i++) { Vx.h[i] += (Vu.h[i] * Vv.h[i]); }</pre>

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.

Intrinsics

<code>Vd.h=vmpyi (Vu.h,Vv.h)</code>	<code>HVX_Vector Q6_Vh_vmpyi_VhVh (HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vx.h+=vmpyi (Vu.h,Vv.h)</code>	<code>HVX_Vector Q6_Vh_vmpyiacc_VhVhVh (HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS															Parse		u5										d5					
0	0	0	1	1	1	0	0	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.h=vmpyi(Vu.h,Vv.h)
ICLASS															Parse		u5										x5					
0	0	0	1	1	1	0	0	0	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	x	x	x	x	x	Vx.h+=vmpyi(Vu.h,Vv.h)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
x5	Field to encode register x

Integer multiply (32×16)

Multiply words in one vector by even or odd halfwords in another vector. Take the lower part. Some versions of this operation perform unusual shifts to facilitate 32×32 multiply synthesis.

Syntax	Behavior
<code>Vd.w=vmpyie(Vu.w,Vv.uh)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i] = (Vu.w[i] * Vv.w[i].uh[0]); }</pre>
<code>Vd.w=vmpyio(Vu.w,Vv.h)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i] = (Vu.w[i] * Vv.w[i].h[1]); }</pre>
<code>Vx.w+=vmpyie(Vu.w,Vv.h)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vx.w[i] = Vx.w[i] + (Vu.w[i] * Vv.w[i].h[0]); }</pre>
<code>Vx.w+=vmpyie(Vu.w,Vv.uh)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vx.w[i] = Vx.w[i] + (Vu.w[i] * Vv.w[i].uh[0]); }</pre>

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.

Intrinsics

<code>Vd.w=vmpyie(Vu.w,Vv.uh)</code>	<code>HVX_Vector Q6_Vw_vmpyie_VwVuh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.w=vmpyio(Vu.w,Vv.h)</code>	<code>HVX_Vector Q6_Vw_vmpyio_VwVh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vx.w+=vmpyie(Vu.w,Vv.h)</code>	<code>HVX_Vector Q6_Vw_vmpyieacc_VwVwVh(HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vx.w+=vmpyie(Vu.w,Vv.uh)</code>	<code>HVX_Vector Q6_Vw_vmpyieacc_VwVwVuh(HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse		u5										x5				
0	0	0	1	1	1	0	0	0	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	1	x	x	x	x	x	<code>Vx.w+=vmpyie(Vu.w,Vv.uh)</code>
0	0	0	1	1	1	0	0	0	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	x	x	x	x	x	<code>Vx.w+=vmpyie(Vu.w,Vv.h)</code>
ICLASS																Parse		u5										d5				
0	0	0	1	1	1	1	1	1	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	<code>Vd.w=vmpyie(Vu.w,Vv.uh)</code>
0	0	0	1	1	1	1	1	1	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	<code>Vd.w=vmpyio(Vu.w,Vv.h)</code>

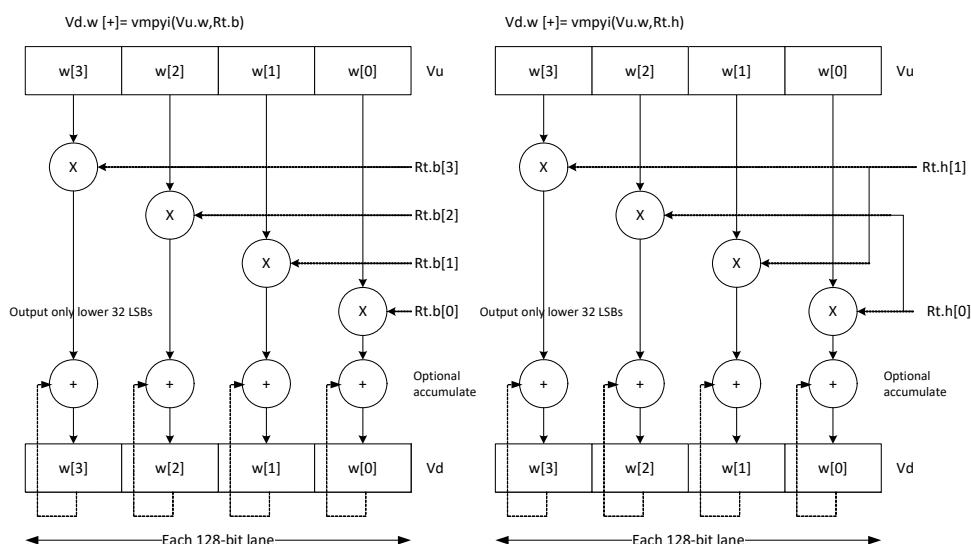
Field name		Description
ICLASS	Instruction class	
Parse	Packet loop parse bits	
d5	Field to encode register d	
u5	Field to encode register u	
v5	Field to encode register v	
x5	Field to encode register x	

Integer multiply accumulate even/odd

Multiply groups of words in vector register Vu by the elements in Rt. The lower 32-bit results are placed in vector register Vd.

The operation has two forms: signed words or halfwords in Vu, multiplied by signed bytes in Rt.

Optionally accumulates the product with the destination vector register Vx.



Syntax

$Vd.w = vmpyi(Vu.w, Rt.h)$

$Vx.w += vmpyi(Vu.w, Rt.h)$

Behavior

```
for (i = 0; i < VELEM(32); i++) {
    Vd.w[i] = (Vu.w[i] * Rt.h[i % 2]);
}
```

```
for (i = 0; i < VELEM(32); i++) {
    Vx.w[i] += (Vu.w[i] * Rt.h[i % 2]);
}
```

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.

Intrinsics

$Vd.w = vmpyi(Vu.w, Rt.h)$

$Vx.w += vmpyi(Vu.w, Rt.h)$

HVX_Vector Q6_Vw_vmpyi_VwRh(HVX_Vector Vu, Word32 Rt)

HVX_Vector Q6_Vw_vmpyiacc_VwVwRh(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS										t5					Parse		u5										x5					
0	0	0	1	1	0	0	1	0	1	0	t	t	t	t	t	P	P	1	u	u	u	u	u	0	1	1	x	x	x	x	x	Vx.w+=vmpyi(Vu.w,Rt.h)
ICLASS										t5					Parse		u5										d5					
0	0	0	1	1	0	0	1	1	0	0	t	t	t	t	t	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.w=vmpyi(Vu.w,Rt.h)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t5	Field to encode register t
u5	Field to encode register u
x5	Field to encode register x

Multiply - single precision vector by vector

These instructions perform a vectorized single precision floating point multiply. The inputs are either both IEEE single precision or both 32-bit Qfloat. The result is a 32-bit Qfloat vector.

Syntax	Behavior
<code>Vd.qf32=vmpy(Vu.qf32,Vv.qf32)</code>	<pre>for (i = 0; i < VELEM(32); i++) { u = Vu.qf32[i]; v = Vv.qf32[i]; Vd.qf32[i] = rnd_sat(u.exp+v.exp,u.sig*v.sig,0); }</pre>
<code>Vd.qf32=vmpy(Vu.sf,Vv.sf)</code>	<pre>for (i = 0; i < VELEM(32); i++) { u = Vu.sf[i]; v = Vv.sf[i]; Vd.qf32[i] = rnd_sat(u.exp+v.exp,u.sig*v.sig,0); if(u.sign^v.sign) Vd.qf32[i] = - (Vd.qf32[i]); }</pre>

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.

Intrinsics

<code>Vd.qf32=vmpy(Vu.qf32,Vv.qf32)</code>	<code>HVX_Vector Q6_Vqf32_vmpy_Vqf32Vqf32(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.qf32=vmpy(Vu.sf,Vv.sf)</code>	<code>HVX_Vector Q6_Vqf32_vmpy_VsfVsf(HVX_Vector Vu, HVX_Vector Vv)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS															Parse		u5										d5					
0	0	0	1	1	1	1	1	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.qf32=vmpy(Vu.qf32,Vv.qf32)
0	0	0	1	1	1	1	1	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.qf32=vmpy(Vu.sf,Vv.sf)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Multiply (32 × 16)

Multiply words in one vector by even or odd halfwords in another vector. Take the upper part. Some versions of this operation perform specific shifts to facilitate 32× 32 multiply synthesis.

An important operation is a 32 × 32 fractional multiply, equivalent to (OP1 * OP2) >> 31. The case of fn(0x80000000, 0x80000000) must saturate to 0x7fffffff.

The rounding fractional multiply:

```
vectorize(sat32(x * y + 0x40000000) >> 31) )
```

equivalent to:

```
{V2 = vmpye(V0.w, V1.uh)} {V2+= vmpyo(V0.w, V1.h):<<1:rnd:sat:shift}
```

The nonrounding fractional multiply version:

```
vectorize(sat32(x * y)>>31) ) equivalent to: {V2 = vmpye(V0.w, V1.uh) } {  
V2+= vmpyo(V0.w, V1.h):<<1:sat:shift}
```

A key function is a 32-bit x 32-bit signed multiply where the 64-bit result is kept.

```
vectorize( (int64) x * (int64) y )
```

equivalent to:

```
{ V3:2 = vmpye(V0.w, V1.uh) } { V3:2+= vmpyo(V0.w, V1.h) }
```

The lower 32 bits of products are in V2 and the upper 32 bits in V3. If only `vmpye` is performed, the result is a 48-bit product of 32 signed × 16-bit unsigned asserted into the upper 48 bits of Vdd. If only `vmpyo` is performed, assuming `Vxx = #0`, the result is a 32 signed × 16 signed product asserted into the upper 48 bits of Vxx.

Syntax	Behavior
<code>Vd.w=vmpye(Vu.w,Vv.uh)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i] = (Vu.w[i] * Vv.w[i].uh[0]) >> 16; }</pre>
<code>Vd.w=vmpyo(Vu.w,Vv.h):<<1[:rnd]:sat</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i] = sat₃₂(((((Vu.w[i] * Vv.w[i].h[1]) >> 14) + 1) >> 1)); }</pre>
<code>Vdd=vmpye(Vu.w,Vv.uh)</code>	<pre>for (i = 0; i < VELEM(32); i++) { prod = (Vu.w[i] * Vv.w[i].uh[0]); Vdd.v[1].w[i] = prod >> 16; Vdd.v[0].w[i] = prod << 16; }</pre>
<code>Vx.w+=vmpyo(Vu.w,Vv.h):<<1[:rnd]:sat:shift</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vx.w[i] = sat₃₂(((((Vx.w[i] + (Vu.w[i] * Vv.w[i].h[1])) >> 14) + 1) >> 1)); }</pre>

Syntax	Behavior
<code>Vxx+=vmpyo (Vu.w,Vv.h)</code>	<pre> for (i = 0; i < VELEM(32); i++) { prod = (Vu.w[i] * Vv.w[i].h[1]) + Vxx.v[1].w[i]; Vxx.v[1].w[i] = prod >> 16; Vxx.v[0].w[i].h[0]=Vxx.v[0].w[i] >> 16; Vxx.v[0].w[i].h[1]=prod & 0x0000ffff; } </pre>

Class: COPROC_VX (slots 2,3)**Notes**

- This instruction uses both HVX multiply resources.

Intrinsics

<code>Vd.w=vmpye (Vu.w,Vv.uh)</code>	<code>HVX_Vector Q6_Vw_vmpye_VwVuh (HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.w=vmpyo (Vu.w,Vv.h) :<<1:rnd:sat</code>	<code>HVX_Vector Q6_Vw_vmpyo_VwVh_s1_rnd_sat (HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.w=vmpyo (Vu.w,Vv.h) :<<1:sat</code>	<code>HVX_Vector Q6_Vw_vmpyo_VwVh_s1_sat (HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vdd=vmpye (Vu.w,Vv.uh)</code>	<code>HVX_VectorPair Q6_W_vmpye_VwVuh (HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vx.w+=vmpyo (Vu.w,Vv.h) :<<1:rnd:sat:shift</code>	<code>HVX_Vector Q6_Vw_vmpyoacc_VwVwVh_s1_rnd_sat_shift (HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vx.w+=vmpyo (Vu.w,Vv.h) :<<1:sat:shift</code>	<code>HVX_Vector Q6_Vw_vmpyoacc_VwVwVh_s1_sat_shift (HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vxx+=vmpyo (Vu.w,Vv.h)</code>	<code>HVX_VectorPair Q6_W_vmpyoacc_WwVwVh (HVX_VectorPair Vxx, HVX_Vector Vu, HVX_Vector Vv)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse		u5										x5				
0	0	0	1	1	1	0	0	0	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	x	x	x	x	x	Vxx+=vmpyo(Vu.w,Vv.h)
0	0	0	1	1	1	0	0	0	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	0	x	x	x	x	x	Vx.w+=vmpyo(Vu.w,Vv.h):<1:sat:shift
0	0	0	1	1	1	0	0	0	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	x	x	x	x	x	Vx.w+=vmpyo(Vu.w,Vv.h):<1:rnd:sat:shift

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS										Parse						u5					d5											
0	0	0	1	1	1	1	0	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vdd=vmpye(Vu.w,Vv.uh)
0	0	0	1	1	1	1	1	0	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.w=vmpyo(Vu.w,Vv.h):<<1:rnd:sat
0	0	0	1	1	1	1	1	1	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.w=vmpye(Vu.w,Vv.uh)
0	0	0	1	1	1	1	1	1	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.w=vmpyo(Vu.w,Vv.h):<<1:sat

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v
x5	Field to encode register x

Multiply bytes with 4-wide reduction - vector by scalar

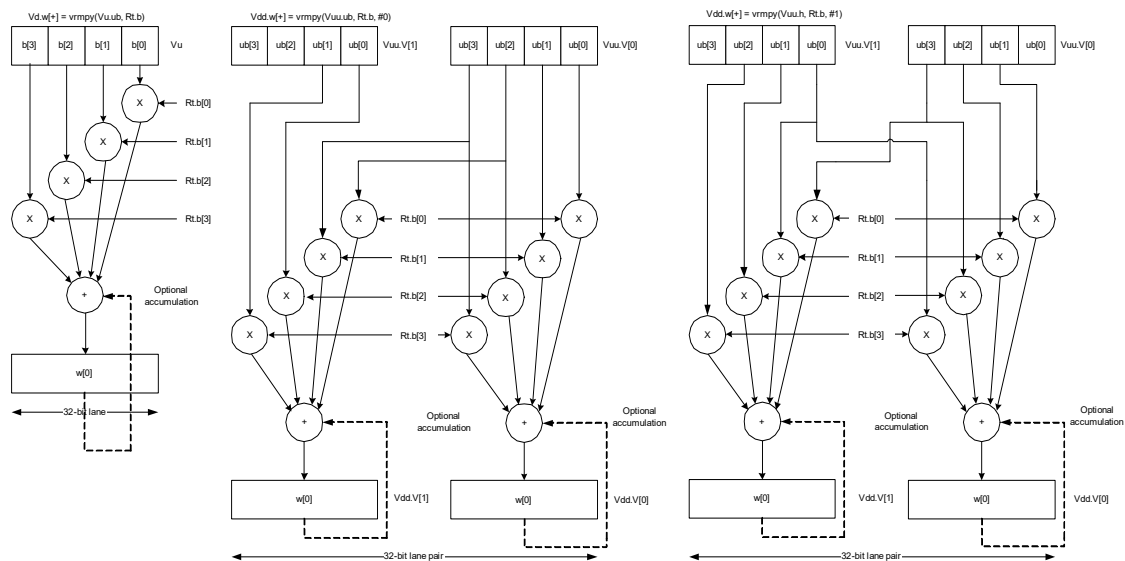
Perform multiplication between the elements in vector Vu and the corresponding elements in the scalar register Rt, followed by a 4-way reduction to a word in each 32-bit lane. Accumulate the result in Vx or Vxx.

Supports the multiplication of unsigned byte data by signed or unsigned bytes in the scalar.

The operation has two forms: the first performs simple dot product of four elements into a single result. The second form takes a one bit immediate input and generates a vector register pair.

For #1 = 0, the even destination contains a simple dot product, the odd destination contains a dot product of the coefficients rotated by two elements and the upper two data elements taken from the even register of Vuu.

For #u = 1, the even destination takes coefficients rotated by -1 and data element 0 from the odd register of Vuu. The odd destination uses coefficients rotated by -1 and takes data element 3 from the even register of Vuu.



Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.

Syntax	Behavior
<code>Vdd.uw=vrmpy (Vuu.ub, Rt.ub, #u1)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vdd.v[0].uw[i] = (Vuu.v[#u ? 1:0].uw[i].ub[0] * Rt.ub[(0-#u) & 0x3]); Vdd.v[0].uw[i] += (Vuu.v[0].uw[i].ub[1] * Rt.ub[(1-#u) & 0x3]); Vdd.v[0].uw[i] += (Vuu.v[0].uw[i].ub[2] * Rt.ub[(2-#u) & 0x3]); Vdd.v[0].uw[i] += (Vuu.v[0].uw[i].ub[3] * Rt.ub[(3-#u) & 0x3]); Vdd.v[1].uw[i] = (Vuu.v[1].uw[i].ub[0] * Rt.ub[(2-#u) & 0x3]); Vdd.v[1].uw[i] += (Vuu.v[1].uw[i].ub[1] * Rt.ub[(3-#u) & 0x3]); Vdd.v[1].uw[i] += (Vuu.v[#u ? 1:0].uw[i].ub[2] * Rt.ub[(0-#u) & 0x3]); Vdd.v[1].uw[i] += (Vuu.v[0].uw[i].ub[3] * Rt.ub[(1-#u) & 0x3]); } </pre>
<code>Vdd.w=vrmpy (Vuu.ub, Rt.b, #u1)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vdd.v[0].w[i] = (Vuu.v[#u ? 1:0].uw[i].ub[0] * Rt.b[(0-#u) & 0x3]); Vdd.v[0].w[i] += (Vuu.v[0].uw[i].ub[1] * Rt.b[(1-#u) & 0x3]); Vdd.v[0].w[i] += (Vuu.v[0].uw[i].ub[2] * Rt.b[(2-#u) & 0x3]); Vdd.v[0].w[i] += (Vuu.v[0].uw[i].ub[3] * Rt.b[(3-#u) & 0x3]); Vdd.v[1].w[i] = (Vuu.v[1].uw[i].ub[0] * Rt.b[(2-#u) & 0x3]); Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[1] * Rt.b[(3-#u) & 0x3]); Vdd.v[1].w[i] += (Vuu.v[#u ? 1:0].uw[i].ub[2] * Rt.b[(0-#u) & 0x3]); Vdd.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * Rt.b[(1-#u) & 0x3]); } </pre>
<code>Vxx.uw+=vrmpy (Vuu.ub, Rt.ub, #u1)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vxx.v[0].uw[i] += (Vuu.v[#u ? 1:0].uw[i].ub[0] * Rt.ub[(0-#u) & 0x3]); Vxx.v[0].uw[i] += (Vuu.v[0].uw[i].ub[1] * Rt.ub[(1-#u) & 0x3]); Vxx.v[0].uw[i] += (Vuu.v[0].uw[i].ub[2] * Rt.ub[(2-#u) & 0x3]); Vxx.v[0].uw[i] += (Vuu.v[0].uw[i].ub[3] * Rt.ub[(3-#u) & 0x3]); Vxx.v[1].uw[i] += (Vuu.v[1].uw[i].ub[0] * Rt.ub[(2-#u) & 0x3]); Vxx.v[1].uw[i] += (Vuu.v[1].uw[i].ub[1] * Rt.ub[(3-#u) & 0x3]); Vxx.v[1].uw[i] += (Vuu.v[#u ? 1:0].uw[i].ub[2] * Rt.ub[(0-#u) & 0x3]); Vxx.v[1].uw[i] += (Vuu.v[0].uw[i].ub[3] * Rt.ub[(1-#u) & 0x3]); } </pre>

Syntax

```
Vxx.w+=vrmpy(Vuu.ub,Rt.b,#u1)
```

Behavior

```
for (i = 0; i < VELEM(32); i++) {
    Vxx.v[0].w[i] += (Vuu.v[#u ?
1:0].uw[i].ub[0] * Rt.b[(0-#u) & 0x3]);
    Vxx.v[0].w[i] += (Vuu.v[0 ].uw[i].ub[1] *
Rt.b[(1-#u) & 0x3]);
    Vxx.v[0].w[i] += (Vuu.v[0 ].uw[i].ub[2] *
Rt.b[(2-#u) & 0x3]);
    Vxx.v[0].w[i] += (Vuu.v[0 ].uw[i].ub[3] *
Rt.b[(3-#u) & 0x3]);
    Vxx.v[1].w[i] += (Vuu.v[1 ].uw[i].ub[0] *
Rt.b[(2-#u) & 0x3]);
    Vxx.v[1].w[i] += (Vuu.v[1 ].uw[i].ub[1] *
Rt.b[(3-#u) & 0x3]);
    Vxx.v[1].w[i] += (Vuu.v[#u ?
1:0].uw[i].ub[2] * Rt.b[(0-#u) & 0x3]);
    Vxx.v[1].w[i] += (Vuu.v[0 ].uw[i].ub[3] *
Rt.b[(1-#u) & 0x3]);
}
```

Intrinsics

```
Vdd.uw=vrmpy(Vuu.ub,Rt.ub,#u1)
```

```
HVX_VectorPair
Q6_Wuw_vrmpy_WubRbI(HVX_VectorPair Vuu,
Word32 Rt, Word32 Iu1)
```

```
Vdd.w=vrmpy(Vuu.ub,Rt.b,#u1)
```

```
HVX_VectorPair
Q6_Ww_vrmpy_WubRbI(HVX_VectorPair Vuu, Word32
Rt, Word32 Iu1)
```

```
Vxx.uw+=vrmpy(Vuu.ub,Rt.ub,#u1)
```

```
HVX_VectorPair
Q6_Wuw_vrmpyacc_WuwWubRbI(HVX_VectorPair
Vxx, HVX_VectorPair Vuu, Word32 Rt, Word32
Iu1)
```

```
Vxx.w+=vrmpy(Vuu.ub,Rt.b,#u1)
```

```
HVX_VectorPair
Q6_Ww_vrmpyacc_WwWubRbI(HVX_VectorPair Vxx,
HVX_VectorPair Vuu, Word32 Rt, Word32 Iu1)
```

Encoding

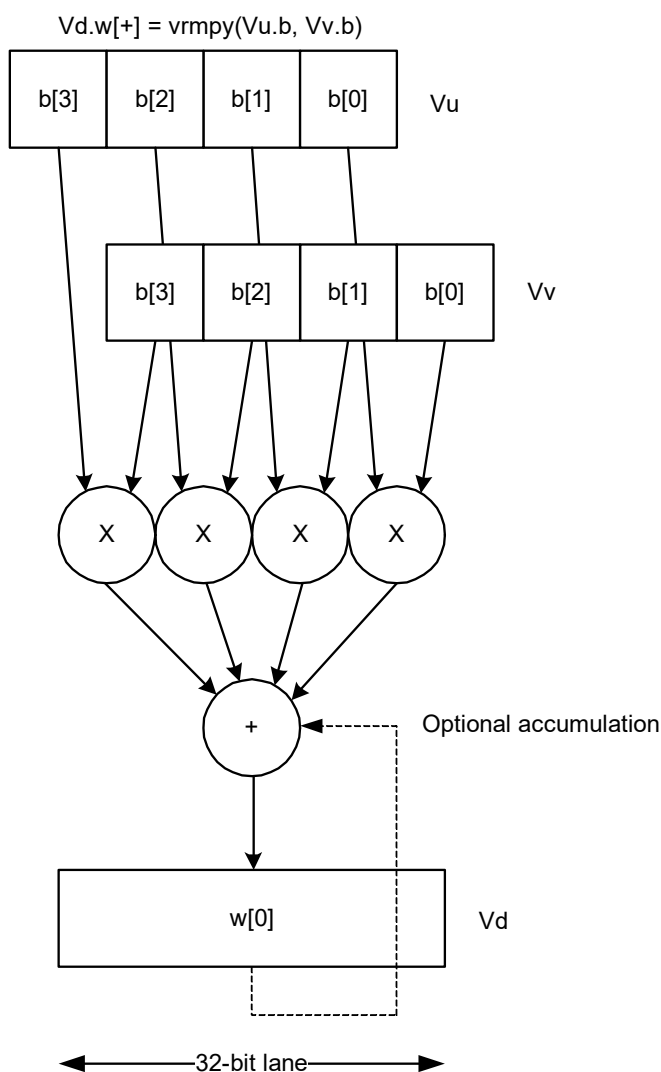
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											t5				Parse		u5										d5					
0	0	0	1	1	0	0	1	0	1	0	t	t	t	t	t	P	P	0	u	u	u	u	u	1	0	i	d	d	d	d	d	Vdd.w=vrmpy(Vuu.ub,Rt.b,#u1)
ICLASS											t5				Parse		u5										x5					
0	0	0	1	1	0	0	1	0	1	0	t	t	t	t	t	P	P	1	u	u	u	u	u	1	0	i	x	x	x	x	x	Vxx.w+=vrmpy(Vuu.ub,Rt.b,#u1)
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	1	u	u	u	u	u	1	1	i	x	x	x	x	x	Vxx.uw+=vrmpy(Vuu.ub,Rt.ub,#u1)
ICLASS											t5				Parse		u5										d5					
0	0	0	1	1	0	0	1	1	0	1	t	t	t	t	t	P	P	0	u	u	u	u	u	1	1	i	d	d	d	d	d	Vdd.uw=vrmpy(Vuu.ub,Rt.ub,#u1)

Field name		Description
ICLASS		Instruction class
Parse		Packet loop parse bits
d5		Field to encode register d
t5		Field to encode register t
u5		Field to encode register u
x5		Field to encode register x

Multiply by byte with accumulate and 4-wide reduction - vector by vector

The `vrmpy` instruction performs a dot product function between 4 byte elements in vector register `Vu` and 4 byte elements in `Vv`. The sum of products can be optionally accumulated into `Vx` or written into `Vd` as words within each 32-bit lane.

Data types can be unsigned by unsigned, signed by signed, or unsigned by signed.



Syntax

```
Vx.uw+=vrmpy(Vu.ub,Vv.ub)
```

Behavior

```
for (i = 0; i < VELEM(32); i++) {
    Vx.uw[i] += (Vu.uw[i].ub[0] * Vv.uw[i].ub[0]);
    Vx.uw[i] += (Vu.uw[i].ub[1] * Vv.uw[i].ub[1]);
    Vx.uw[i] += (Vu.uw[i].ub[2] * Vv.uw[i].ub[2]);
    Vx.uw[i] += (Vu.uw[i].ub[3] * Vv.uw[i].ub[3]);
}
```

Syntax	Behavior
<code>Vx.w+=vrmpy(Vu.b,Vv.b)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vx.w[i] += (Vu.w[i].b[0] * Vv.w[i].b[0]); Vx.w[i] += (Vu.w[i].b[1] * Vv.w[i].b[1]); Vx.w[i] += (Vu.w[i].b[2] * Vv.w[i].b[2]); Vx.w[i] += (Vu.w[i].b[3] * Vv.w[i].b[3]); } </pre>
<code>Vx.w+=vrmpy(Vu.ub,Vv.b)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vx.w[i] += (Vu.uw[i].ub[0] * Vv.w[i].b[0]); Vx.w[i] += (Vu.uw[i].ub[1] * Vv.w[i].b[1]); Vx.w[i] += (Vu.uw[i].ub[2] * Vv.w[i].b[2]); Vx.w[i] += (Vu.uw[i].ub[3] * Vv.w[i].b[3]); } </pre>

Class: COPROC_VX (slots 2,3)**Notes**

- This instruction uses a HVX multiply resource.

Intrinsics

<code>Vx.uw+=vrmpy(Vu.ub,Vv.ub)</code>	<code>HVX_Vector Q6_Vuw_vrmpyacc_VuwVubVub(HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vx.w+=vrmpy(Vu.b,Vv.b)</code>	<code>HVX_Vector Q6_Vw_vrmpyacc_VwVbVb(HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vx.w+=vrmpy(Vu.ub,Vv.b)</code>	<code>HVX_Vector Q6_Vw_vrmpyacc_VwVubVb(HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv)</code>

Encoding

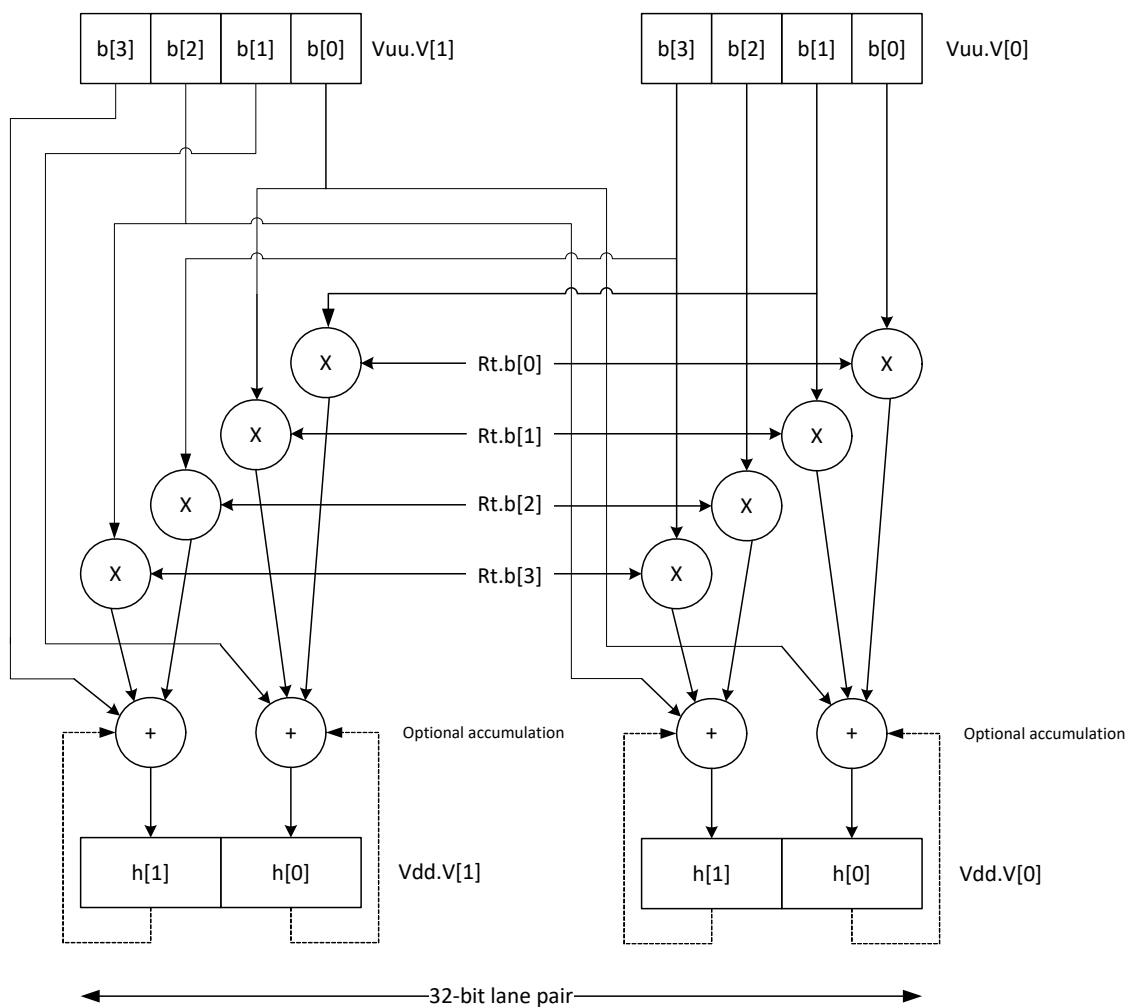
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
ICLASS																Parse		u5								x5				Vx.uw+=vrmpy(Vu.ub,Vv.ub) Vx.w+=vrmpy(Vu.b,Vv.b) Vx.w+=vrmpy(Vu.ub,Vv.b)			
0	0	0	1	1	1	0	0	0	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	x	x		x	x	x
0	0	0	1	1	1	0	0	0	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	x	x		x	x	x
0	0	0	1	1	1	0	0	0	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	0	x	x		x	x	x

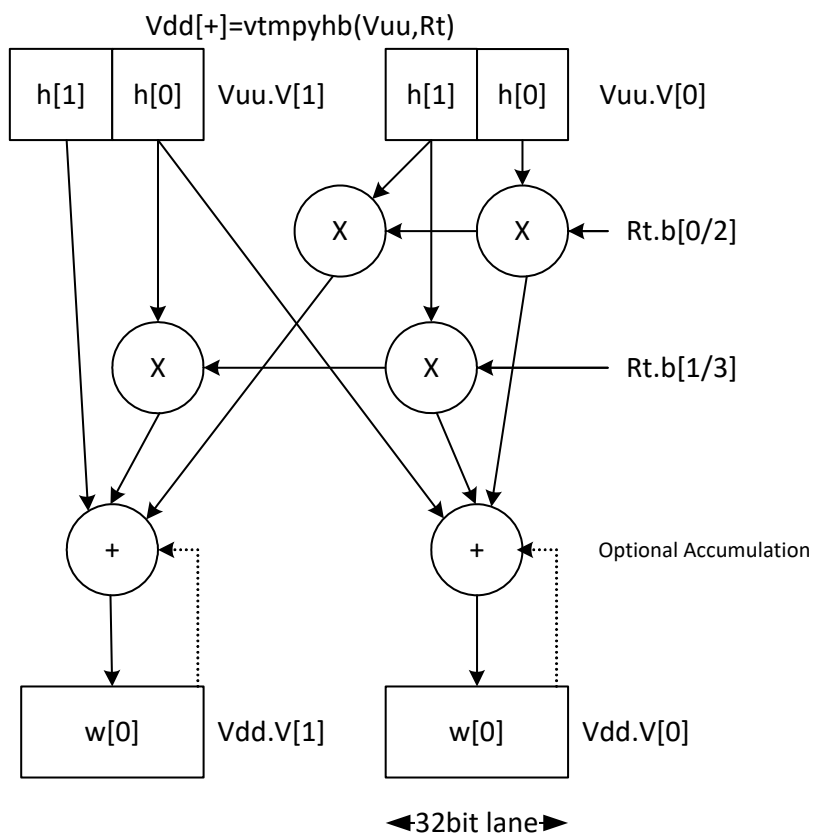
Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
u5	Field to encode register u
v5	Field to encode register v
x5	Field to encode register x

Multiply with 3-wide reduction

Perform a 3-element sliding window pattern operation consisting of a two multiplies with an additional accumulation. Data elements are stored in the vector register pair Vuu, and coefficients in the scalar register Rt.

$Vdd.h[+] = vtmpr(Vuu.b, Rt.b)$



**Syntax**

```
Vdd.h=vtmpy(Vuu.b, Rt.b)
```

Behavior

```
for (i = 0; i < VELEM(16); i++) {
    Vdd.v[0].h[i] = (Vuu.v[0].h[i].b[0] *
    Rt.b[(2*i) % 4]);
    Vdd.v[0].h[i] += (Vuu.v[0].h[i].b[1] *
    Rt.b[(2*i+1) % 4]);
    Vdd.v[0].h[i] += Vuu.v[1].h[i].b[0];
    Vdd.v[1].h[i] = (Vuu.v[0].h[i].b[1] *
    Rt.b[(2*i) % 4]);
    Vdd.v[1].h[i] += (Vuu.v[1].h[i].b[0] *
    Rt.b[(2*i+1) % 4]);
    Vdd.v[1].h[i] += Vuu.v[1].h[i].b[1];
}
```

Syntax	Behavior
<code>Vdd.h=vtmpy(Vuu.ub,Rt.b)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vdd.v[0].h[i] = (Vuu.v[0].uh[i].ub[0] * Rt.b[(2*i)%4]); Vdd.v[0].h[i] += (Vuu.v[0].uh[i].ub[1] * Rt.b[(2*i+1)%4]); Vdd.v[0].h[i] += Vuu.v[1].uh[i].ub[0]; Vdd.v[1].h[i] = (Vuu.v[0].uh[i].ub[1] * Rt.b[(2*i)%4]); Vdd.v[1].h[i] += (Vuu.v[1].uh[i].ub[0] * Rt.b[(2*i+1)%4]); Vdd.v[1].h[i] += Vuu.v[1].uh[i].ub[1]; } </pre>
<code>Vdd.w=vtmpy(Vuu.h,Rt.b)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vdd.v[0].w[i] = (Vuu.v[0].w[i].h[0] * Rt.b[(2*i+0)%4]); Vdd.v[0].w[i] += (Vuu.v[0].w[i].h[1] * Rt.b[(2*i+1)%4]); Vdd.v[0].w[i] += Vuu.v[1].w[i].h[0]; Vdd.v[1].w[i] = (Vuu.v[0].w[i].h[1] * Rt.b[(2*i+0)%4]); Vdd.v[1].w[i] += (Vuu.v[1].w[i].h[0] * Rt.b[(2*i+1)%4]); Vdd.v[1].w[i] += Vuu.v[1].w[i].h[1]; } </pre>
<code>Vxx.h+=vtmpy(Vuu.b,Rt.b)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vxx.v[0].h[i] += (Vuu.v[0].h[i].b[0] * Rt.b[(2*i)%4]); Vxx.v[0].h[i] += (Vuu.v[0].h[i].b[1] * Rt.b[(2*i+1)%4]); Vxx.v[0].h[i] += Vuu.v[1].h[i].b[0]; Vxx.v[1].h[i] += (Vuu.v[0].h[i].b[1] * Rt.b[(2*i)%4]); Vxx.v[1].h[i] += (Vuu.v[1].h[i].b[0] * Rt.b[(2*i+1)%4]); Vxx.v[1].h[i] += Vuu.v[1].h[i].b[1]; } </pre>
<code>Vxx.h+=vtmpy(Vuu.ub,Rt.b)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vxx.v[0].h[i] += (Vuu.v[0].uh[i].ub[0] * Rt.b[(2*i)%4]); Vxx.v[0].h[i] += (Vuu.v[0].uh[i].ub[1] * Rt.b[(2*i+1)%4]); Vxx.v[0].h[i] += Vuu.v[1].uh[i].ub[0]; Vxx.v[1].h[i] += (Vuu.v[0].uh[i].ub[1] * Rt.b[(2*i)%4]); Vxx.v[1].h[i] += (Vuu.v[1].uh[i].ub[0] * Rt.b[(2*i+1)%4]); Vxx.v[1].h[i] += Vuu.v[1].uh[i].ub[1]; } </pre>

Syntax	Behavior
Vxx.w+=vtmpy(Vuu.h,Rt.b)	<pre> for (i = 0; i < VELEM(32); i++) { Vxx.v[0].w[i] += (Vuu.v[0].w[i].h[0] * Rt.b[(2*i+0)%4]); Vxx.v[0].w[i] += (Vuu.v[0].w[i].h[1] * Rt.b[(2*i+1)%4]); Vxx.v[0].w[i] += Vuu.v[1].w[i].h[0]; Vxx.v[1].w[i] += (Vuu.v[0].w[i].h[1] * Rt.b[(2*i+0)%4]); Vxx.v[1].w[i] += (Vuu.v[1].w[i].h[0] * Rt.b[(2*i+1)%4]); Vxx.v[1].w[i] += Vuu.v[1].w[i].h[1]; } </pre>

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.

Intrinsics

Vdd.h=vtmpy(Vuu.b,Rt.b)	HVX_VectorPair Q6_Wh_vtmpy_WbRb(HVX_VectorPair Vuu, Word32 Rt)
Vdd.h=vtmpy(Vuu.ub,Rt.b)	HVX_VectorPair Q6_Wh_vtmpy_WubRb(HVX_VectorPair Vuu, Word32 Rt)
Vdd.w=vtmpy(Vuu.h,Rt.b)	HVX_VectorPair Q6_Ww_vtmpy_WhRb(HVX_VectorPair Vuu, Word32 Rt)
Vxx.h+=vtmpy(Vuu.b,Rt.b)	HVX_VectorPair Q6_Wh_vtmpyacc_WhWbRb(HVX_VectorPair Vxx, HVX_VectorPair Vuu, Word32 Rt)
Vxx.h+=vtmpy(Vuu.ub,Rt.b)	HVX_VectorPair Q6_Wh_vtmpyacc_WhWubRb(HVX_VectorPair Vxx, HVX_VectorPair Vuu, Word32 Rt)
Vxx.w+=vtmpy(Vuu.h,Rt.b)	HVX_VectorPair Q6_Ww_vtmpyacc_WwWhRb(HVX_VectorPair Vxx, HVX_VectorPair Vuu, Word32 Rt)

Encoding

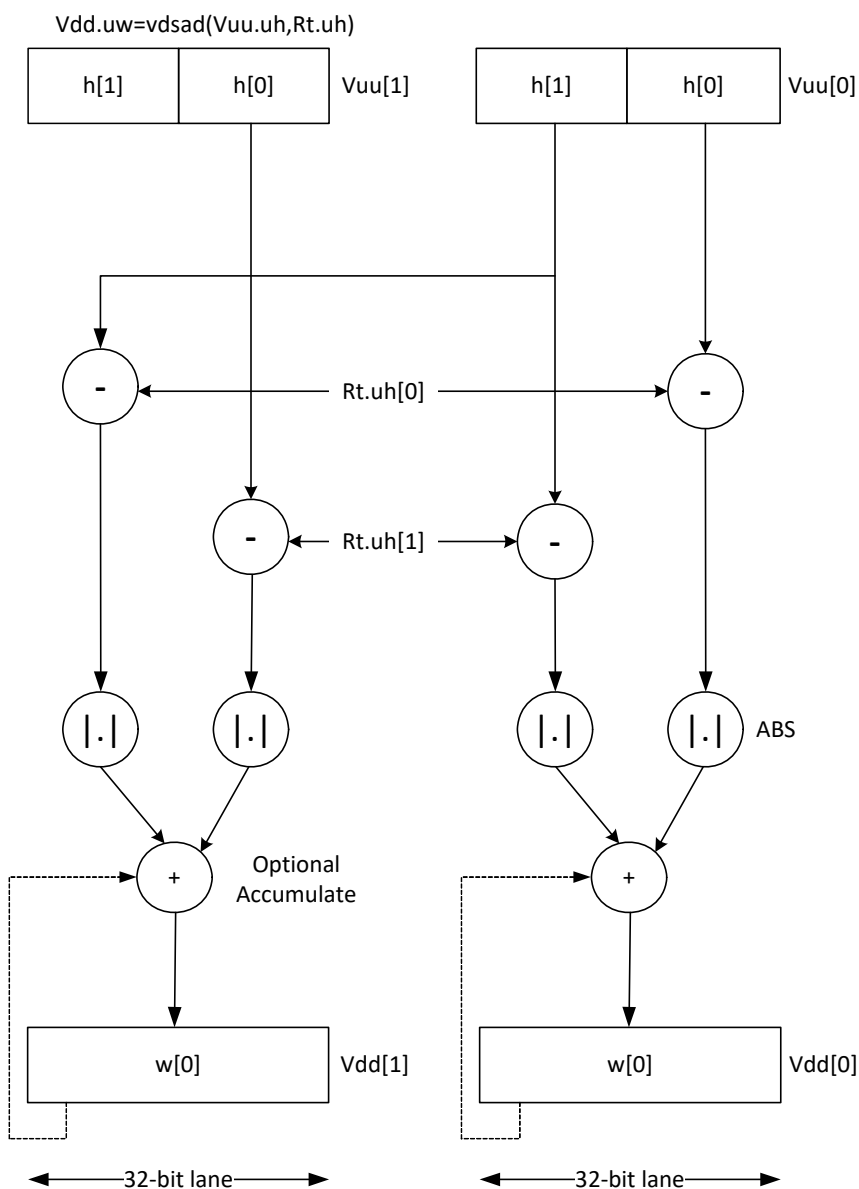
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											t5				Parse		u5										d5					
0	0	0	1	1	0	0	1	0	0	0	t	t	t	t	t	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vdd.h=vtmpy(Vuu.b,Rt.b)
0	0	0	1	1	0	0	1	0	0	0	t	t	t	t	t	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vdd.h=vtmpy(Vuu.ub,Rt.b)
ICLASS											t5				Parse		u5										x5					
0	0	0	1	1	0	0	1	0	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	0	0	0	x	x	x	x	x	Vxx.h+=vtmpy(Vuu.b,Rt.b)
0	0	0	1	1	0	0	1	0	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	0	0	1	x	x	x	x	x	Vxx.h+=vtmpy(Vuu.ub,Rt.b)
0	0	0	1	1	0	0	1	0	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	0	1	0	x	x	x	x	x	Vxx.w+=vtmpy(Vuu.h,Rt.b)
ICLASS											t5				Parse		u5										d5					
0	0	0	1	1	0	0	1	1	0	1	t	t	t	t	t	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vdd.w=vtmpy(Vuu.h,Rt.b)

Field name		Description
ICLASS		Instruction class
Parse		Packet loop parse bits
d5		Field to encode register d
t5		Field to encode register t
u5		Field to encode register u
x5		Field to encode register x

Sum of reduction of absolute differences halfwords

Takes groups of two unsigned halfwords from the vector register source Vuu , subtracts the halfwords from the scalar register Rt , and takes the absolute value as an unsigned result. These are summed together and optionally added to the destination register Vxx , or written directly to Vdd . The even destination register contains the data from $Vuu[0]$ and Rt , $Vdd[1]$ contains the absolute difference of half of the data from $Vuu[0]$ and half from $Vuu[1]$.

This operation is used to implement a sliding window.



Syntax	Behavior
<code>Vdd.uw=vdsad(Vuu.uh,Rt.uh)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vdd.v[0].uw[i] = ABS(Vuu.v[0].uw[i].uh[0] - Rt.uh[0]); Vdd.v[0].uw[i] += ABS(Vuu.v[0].uw[i].uh[1] - Rt.uh[1]); Vdd.v[1].uw[i] = ABS(Vuu.v[0].uw[i].uh[1] - Rt.uh[0]); Vdd.v[1].uw[i] += ABS(Vuu.v[1].uw[i].uh[0] - Rt.uh[1]); } </pre>
<code>Vxx.uw+=vdsad(Vuu.uh,Rt.uh)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vxx.v[0].uw[i] += ABS(Vuu.v[0].uw[i].uh[0] - Rt.uh[0]); Vxx.v[0].uw[i] += ABS(Vuu.v[0].uw[i].uh[1] - Rt.uh[1]); Vxx.v[1].uw[i] += ABS(Vuu.v[0].uw[i].uh[1] - Rt.uh[0]); Vxx.v[1].uw[i] += ABS(Vuu.v[1].uw[i].uh[0] - Rt.uh[1]); } </pre>

Class: COPROC_VX (slots 2,3)**Notes**

- This instruction uses both HVX multiply resources.

Intrinsics

<code>Vdd.uw=vdsad(Vuu.uh,Rt.uh)</code>	<code>HVX_VectorPair Q6_Wuw_vdsad_WuhRuh(HVX_VectorPair Vuu, Word32 Rt)</code>
<code>Vxx.uw+=vdsad(Vuu.uh,Rt.uh)</code>	<code>HVX_VectorPair Q6_Wuw_vdsadacc_WuwWuhRuh(HVX_VectorPair Vxx, HVX_VectorPair Vuu, Word32 Rt)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
ICLASS											t5					Parse			u5								d5						
0	0	0	1	1	0	0	1	0	0	0	t	t	t	t	t	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vdd.uw=vdsad(Vuu.uh,Rt.uh)	
ICLASS											t5					Parse			u5								x5						
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	1	u	u	u	u	u	0	0	0	x	x	x	x	x	Vxx.uw+=vdsad(Vuu.uh,Rt.uh)	

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t5	Field to encode register t
u5	Field to encode register u
x5	Field to encode register x

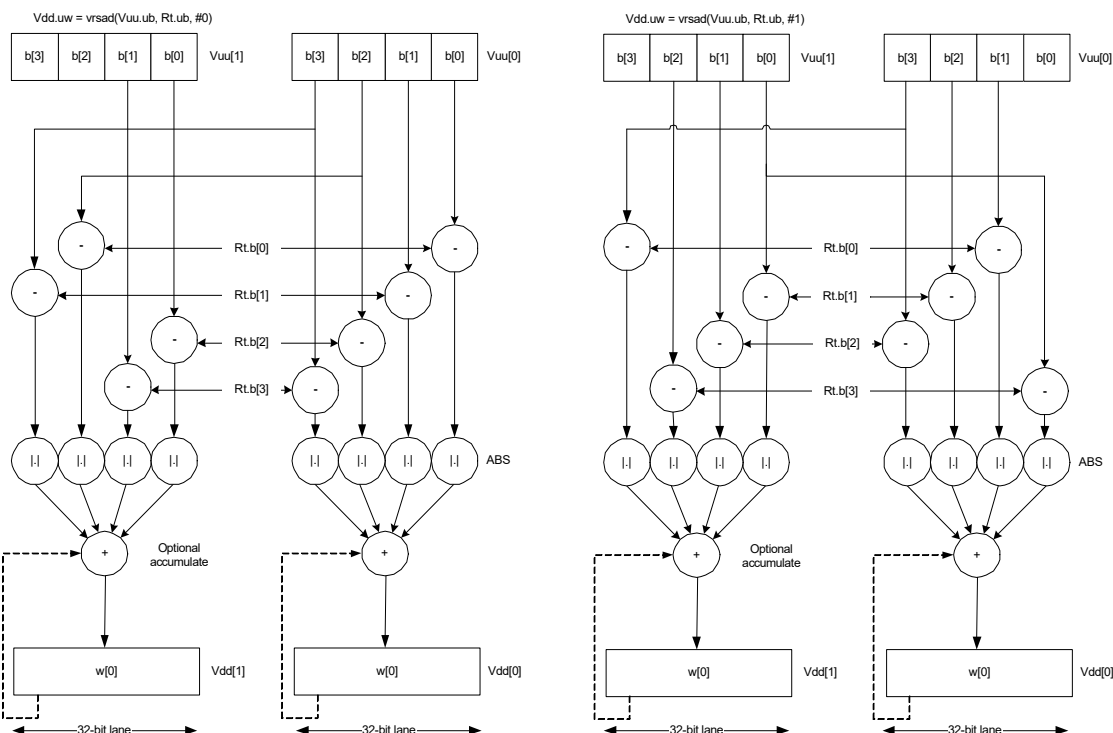
Sum of absolute differences byte

Takes groups of four bytes from the vector register source Vuu, subtracts the bytes from the scalar register Rt, and takes the absolute value as an unsigned result. These are summed together and optionally added to the destination register Vxx, or written directly to Vdd.

If #u1 is 0, the even destination register contains the data from Vuu[0] and Rt, Vdd[1] contains the absolute difference of half of the data from Vuu[0] and half from Vuu[1].

If #u1 is 1, Vdd[0] takes byte 0 from Vuu[1] and bytes 1, 2, and 3 from Vuu[0], while Vdd[1] takes byte 3 from Vuu[0] and the rest from Vuu[1].

This operation is used to implement a sliding window between data in Vuu and Rt.



Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.

Syntax	Behavior
<code>Vdd.uw=vrsad(Vuu.ub,Rt.ub,#u1)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vdd.v[0].uw[i] = ABS(Vuu.v[#u?1:0].uw[i].ub[0] - Rt.ub[(0- #u)&3]); Vdd.v[0].uw[i] += ABS(Vuu.v[0].uw[i].ub[1] - Rt.ub[(1-#u)&3]); Vdd.v[0].uw[i] += ABS(Vuu.v[0].uw[i].ub[2] - Rt.ub[(2-#u)&3]); Vdd.v[0].uw[i] += ABS(Vuu.v[0].uw[i].ub[3] - Rt.ub[(3-#u)&3]); Vdd.v[1].uw[i] = ABS(Vuu.v[1].uw[i].ub[0] - Rt.ub[(2-#u)&3]); Vdd.v[1].uw[i] += ABS(Vuu.v[1].uw[i].ub[1] - Rt.ub[(3-#u)&3]); Vdd.v[1].uw[i] += ABS(Vuu.v[#u?1:0].uw[i].ub[2] - Rt.ub[(0- #u)&3]); Vdd.v[1].uw[i] += ABS(Vuu.v[0].uw[i].ub[3] - Rt.ub[(1-#u)&3]); } </pre>
<code>Vxx.uw+=vrsad(Vuu.ub,Rt.ub,#u1)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vxx.v[0].uw[i] += ABS(Vuu.v[#u?1:0].uw[i].ub[0] - Rt.ub[(0- #u)&3]); Vxx.v[0].uw[i] += ABS(Vuu.v[0].uw[i].ub[1] - Rt.ub[(1-#u)&3]); Vxx.v[0].uw[i] += ABS(Vuu.v[0].uw[i].ub[2] - Rt.ub[(2-#u)&3]); Vxx.v[0].uw[i] += ABS(Vuu.v[0].uw[i].ub[3] - Rt.ub[(3-#u)&3]); Vxx.v[1].uw[i] += ABS(Vuu.v[1].uw[i].ub[0] - Rt.ub[(2-#u)&3]); Vxx.v[1].uw[i] += ABS(Vuu.v[1].uw[i].ub[1] - Rt.ub[(3-#u)&3]); Vxx.v[1].uw[i] += ABS(Vuu.v[#u?1:0].uw[i].ub[2] - Rt.ub[(0- #u)&3]); Vxx.v[1].uw[i] += ABS(Vuu.v[0].uw[i].ub[3] - Rt.ub[(1-#u)&3]); } </pre>

Intrinsics

<code>Vdd.uw=vrsad(Vuu.ub,Rt.ub,#u1)</code>	<pre> HVX_VectorPair Q6_Wuw_vrsad_WubRubI(HVX_VectorPair Vuu, Word32 Rt, Word32 Iu1) </pre>
<code>Vxx.uw+=vrsad(Vuu.ub,Rt.ub,#u1)</code>	<pre> HVX_VectorPair Q6_Wuw_vrsadacc_WuwWubRubI(HVX_VectorPair Vxx, HVX_VectorPair Vuu, Word32 Rt, Word32 Iu1) </pre>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS										t5					Parse		u5										d5					
0	0	0	1	1	0	0	1	0	1	0	t	t	t	t	t	P	P	0	u	u	u	u	u	1	1	i	d	d	d	d	d	Vdd.uw=vrsad(Vuu.ub,Rt.ub,#u1)
ICLASS										t5					Parse		u5										x5					
0	0	0	1	1	0	0	1	0	1	0	t	t	t	t	t	P	P	1	u	u	u	u	u	1	1	i	x	x	x	x	x	Vxx.uw+=vrsad(Vuu.ub,Rt.ub,#u1)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t5	Field to encode register t
u5	Field to encode register u
x5	Field to encode register x

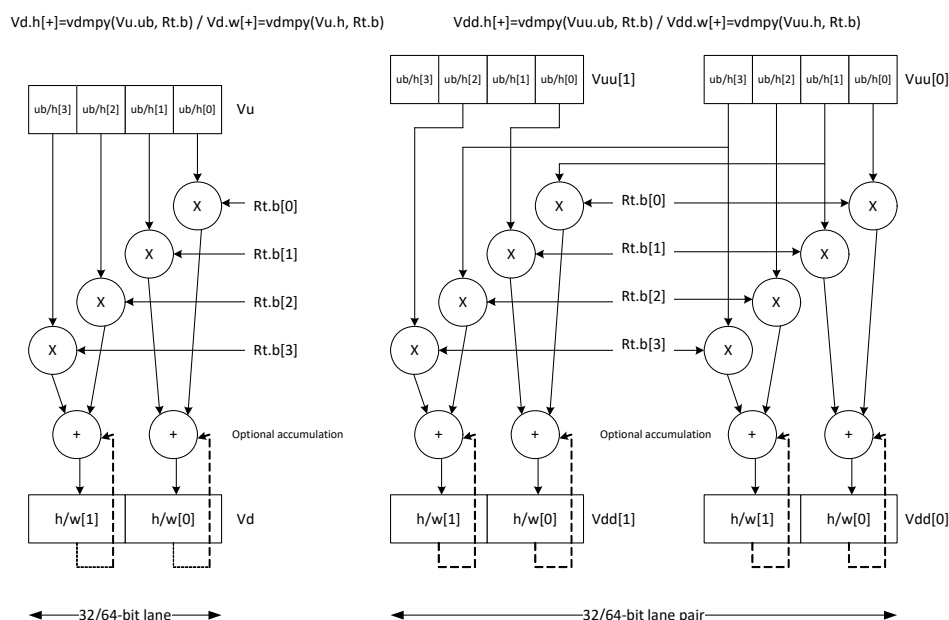
6.9 HVX MPY resource

The HVX MPY resource instruction subclass includes instructions that use a single HVX multiply resource.

Multiply by byte with 2-wide reduction

Multiply elements from Vu by the corresponding elements in the scalar register Rt. The products are added in pairs to yield a by-2 reduction. The products can optionally be accumulated with Vx.

Supports multiplication of unsigned bytes by bytes, and halfwords by signed bytes. The double-vector version performs a sliding-window 2-way reduction, where the odd register output contains the offset computation.



Syntax

$Vd.h = vdmpy(Vu.ub, Rt.b)$

$Vd.w = vdmpy(Vu.h, Rt.b)$

$Vx.h += vdmpy(Vu.ub, Rt.b)$

Behavior

```
for (i = 0; i < VELEM(16); i++) {
    Vd.h[i] = (Vu.ub[i].ub[0] * Rt.b[(2*i) % 4]);
    Vd.h[i] += (Vu.ub[i].ub[1] * Rt.b[(2*i+1) % 4]);
}
```

```
for (i = 0; i < VELEM(32); i++) {
    Vd.w[i] = (Vu.h[i].h[0] * Rt.b[(2*i+0) % 4]);
    Vd.w[i] += (Vu.h[i].h[1] * Rt.b[(2*i+1) % 4]);
}
```

```
for (i = 0; i < VELEM(16); i++) {
    Vx.h[i] += (Vu.ub[i].ub[0] * Rt.b[(2*i) % 4]);
    Vx.h[i] += (Vu.ub[i].ub[1] * Rt.b[(2*i+1) % 4]);
}
```

Syntax	Behavior
<code>Vx.w+=vdmpy(Vu.h,Rt.b)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vx.w[i] += (Vu.w[i].h[0] * Rt.b[(2*i+0)%4]); Vx.w[i] += (Vu.w[i].h[1] * Rt.b[(2*i+1)%4]); } </pre>

Class: COPROC_VX (slots 2,3)**Notes**

- This instruction uses a HVX multiply resource.

Intrinsics

<code>Vd.h=vdmpy(Vu.ub,Rt.b)</code>	<code>HVX_Vector Q6_Vh_vdmpy_VubRb(HVX_Vector Vu, Word32 Rt)</code>
<code>Vd.w=vdmpy(Vu.h,Rt.b)</code>	<code>HVX_Vector Q6_Vw_vdmpy_VhRb(HVX_Vector Vu, Word32 Rt)</code>
<code>Vx.h+=vdmpy(Vu.ub,Rt.b)</code>	<code>HVX_Vector Q6_Vh_vdmpyacc_VhVubRb(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)</code>
<code>Vx.w+=vdmpy(Vu.h,Rt.b)</code>	<code>HVX_Vector Q6_Vw_vdmpyacc_VwVhRb(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)</code>

Encoding

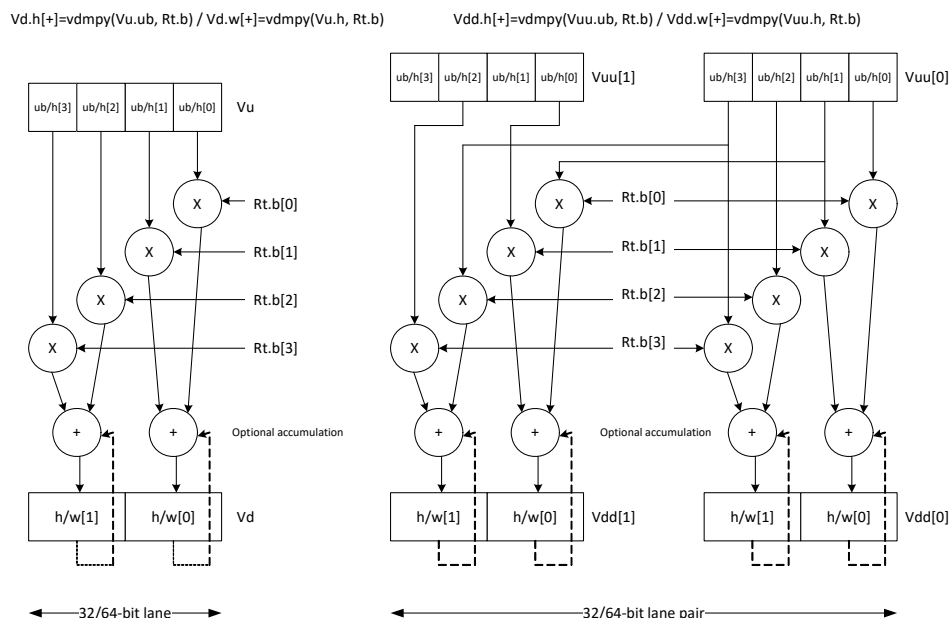
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS												t5				Parse		u5										d5				
0	0	0	1	1	0	0	1	0	0	0	t	t	t	t	t	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.w=vdmpy(Vu.h,Rt.b)
0	0	0	1	1	0	0	1	0	0	0	t	t	t	t	t	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.h=vdmpy(Vu.ub,Rt.b)
ICLASS												t5				Parse		u5										x5				
0	0	0	1	1	0	0	1	0	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	0	1	1	x	x	x	x	x	Vx.w+=vdmpy(Vu.h,Rt.b)
0	0	0	1	1	0	0	1	0	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	1	1	0	x	x	x	x	x	Vx.h+=vdmpy(Vu.ub,Rt.b)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t5	Field to encode register t
u5	Field to encode register u
x5	Field to encode register x

Multiply by halfword with 2-wide reduction

Multiply elements from Vu by the corresponding elements in the scalar register Rt. The products are added in pairs to yield a by-2 reduction. The products can optionally be accumulated with Vx.

Supports multiplication of unsigned bytes by bytes, and halfwords by signed bytes. The double-vector version performs a sliding-window 2-way reduction, where the odd register output contains the offset computation.



Syntax

Vd.w=vdmpy(Vu.h,Rt.h):sat

Vd.w=vdmpy(Vu.h,Rt.uh):sat

Vd.w=vdmpy(Vu.h,Vv.h):sat

Vx.w+=vdmpy(Vu.h,Rt.h):sat

Behavior

```
for (i = 0; i < VELEM(32); i++) {
    accum = (Vu.w[i].h[0] * Rt.h[0]);
    accum += (Vu.w[i].h[1] * Rt.h[1]);
    Vd.w[i] = sat32(accum);
}
```

```
for (i = 0; i < VELEM(32); i++) {
    accum = (Vu.w[i].h[0] * Rt.uh[0]);
    accum += (Vu.w[i].h[1] * Rt.uh[1]);
    Vd.w[i] = sat32(accum);
}
```

```
for (i = 0; i < VELEM(32); i++) {
    accum = (Vu.w[i].h[0] * Vv.w[i].h[0]);
    accum += (Vu.w[i].h[1] * Vv.w[i].h[1]);
    Vd.w[i] = sat32(accum);
}
```

```
for (i = 0; i < VELEM(32); i++) {
    accum = Vx.w[i];
    accum += (Vu.w[i].h[0] * Rt.h[0]);
    accum += (Vu.w[i].h[1] * Rt.h[1]);
    Vx.w[i] = sat32(accum);
}
```

Syntax

```
Vx.w+=vdmpy(Vu.h,Rt.uh):sat
```

Behavior

```
for (i = 0; i < VELEM(32); i++) {
    accum=Vx.w[i];
    accum += (Vu.w[i].h[0] * Rt.uh[0]);
    accum += (Vu.w[i].h[1] * Rt.uh[1]);
    Vx.w[i] = sat32(accum);
}
```

Class: COPROC_VX (slots 2,3)**Notes**

- This instruction uses a HVX multiply resource.

Intrinsics

```
Vd.w=vdmpy(Vu.h,Rt.h):sat
```

```
HVX_Vector Q6_Vw_vdmpy_VhRh_sat(HVX_Vector Vu,
Word32 Rt)
```

```
Vd.w=vdmpy(Vu.h,Rt.uh):sat
```

```
HVX_Vector Q6_Vw_vdmpy_VhRuh_sat(HVX_Vector Vu,
Word32 Rt)
```

```
Vd.w=vdmpy(Vu.h,Vv.h):sat
```

```
HVX_Vector Q6_Vw_vdmpy_VhVh_sat(HVX_Vector Vu,
HVX_Vector Vv)
```

```
Vx.w+=vdmpy(Vu.h,Rt.h):sat
```

```
HVX_Vector Q6_Vw_vdmpyacc_VwVhRh_sat(HVX_Vector
Vx, HVX_Vector Vu, Word32 Rt)
```

```
Vx.w+=vdmpy(Vu.h,Rt.uh):sat
```

```
HVX_Vector Q6_Vw_vdmpyacc_VwVhRuh_sat(HVX_Vector
Vx, HVX_Vector Vu, Word32 Rt)
```

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											t5				Parse		u5							d5								
0	0	0	1	1	0	0	1	0	0	1	t	t	t	t	t	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.w=vdmpy(Vu.h,Rt.uh):sat
0	0	0	1	1	0	0	1	0	0	1	t	t	t	t	t	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.w=vdmpy(Vu.h,Rt.h):sat
ICLASS											t5				Parse		u5							x5								
0	0	0	1	1	0	0	1	0	0	1	t	t	t	t	t	P	P	1	u	u	u	u	u	0	0	0	x	x	x	x	x	Vx.w+=vdmpy(Vu.h,Rt.uh):sat
0	0	0	1	1	0	0	1	0	0	1	t	t	t	t	t	P	P	1	u	u	u	u	u	0	1	1	x	x	x	x	x	Vx.w+=vdmpy(Vu.h,Rt.h):sat
ICLASS															Parse		u5							d5								
0	0	0	1	1	1	0	0	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.w=vdmpy(Vu.h,Vv.h):sat

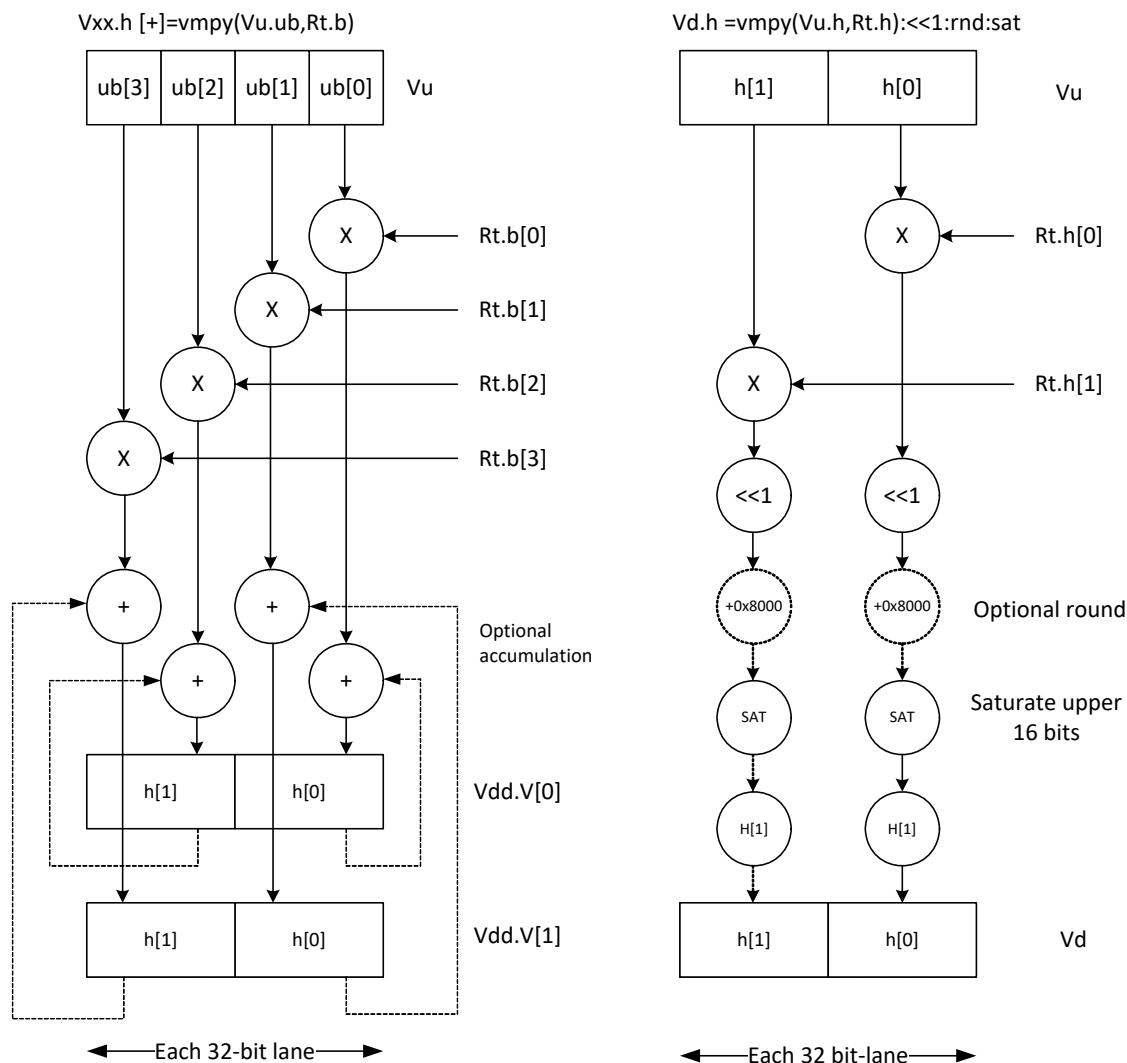
Field name**Description**

ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t5	Field to encode register t
u5	Field to encode register u
v5	Field to encode register v
x5	Field to encode register x

Multiply - vector by scalar non-widening

Multiply groups of elements in the vector V_u by the corresponding elements in the scalar register R_t .

This operation keeps the output precision the same as the input width by shifting the product left by 1, saturating the product to 32 bits, and placing the upper 16 bits in the output. Optional rounding of the result is supported.



Syntax	Behavior
<code>Vd.h=vmpy(Vu.h,Rt.h):<<1:rnd:sat</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vd.w[i].h[0]=sat₁₆(sat₃₂(round((Vu.w[i].h[0] * Rt.h[0])<<1))) .h[1]); Vd.w[i].h[1]=sat₁₆(sat₃₂(round((Vu.w[i].h[1] * Rt.h[1])<<1))) .h[1]); } </pre>
<code>Vd.h=vmpy(Vu.h,Rt.h):<<1:sat</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vd.w[i].h[0]=sat₁₆(sat₃₂((Vu.w[i].h[0] * Rt.h[0])<<1)) .h[1]); Vd.w[i].h[1]=sat₁₆(sat₃₂((Vu.w[i].h[1] * Rt.h[1])<<1)) .h[1]); } </pre>

Class: COPROC_VX (slots 2,3)**Notes**

- This instruction uses a HVX multiply resource.

Intrinsics

<code>Vd.h=vmpy(Vu.h,Rt.h):<<1:rnd:sat</code>	<code>HVX_Vector Q6_Vh_vmpy_VhRh_s1_rnd_sat(HVX_Vector Vu, Word32 Rt)</code>
<code>Vd.h=vmpy(Vu.h,Rt.h):<<1:sat</code>	<code>HVX_Vector Q6_Vh_vmpy_VhRh_s1_sat(HVX_Vector Vu, Word32 Rt)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											t5				Parse		u5								d5							
0	0	0	1	1	0	0	1	0	1	0	t	t	t	t	t	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.h=vmpy(Vu.h,Rt.h):<<1sat
0	0	0	1	1	0	0	1	0	1	0	t	t	t	t	t	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.h=vmpy(Vu.h,Rt.h):<<1rnd:sat

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t5	Field to encode register t
u5	Field to encode register u

Multiply - vector by vector - non-widening

Multiply elements in the vector Vu by the corresponding elements in the vector register Vv and take the upper halfword result.

Syntax	Behavior
<code>Vd.h=vmpy(Vu.h,Vv.h):<<1:rnd:sat</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.h[i] = sat₁₆(sat₃₂(round(((Vu.h[i] * Vv.h[i])<<1)))>>1)); }</pre>
<code>Vd.uh=vmpy(Vu.uh,Vv.uh):>>16</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = (Vu.uh[i] * Vv.uh[i]).uh[1]; }</pre>

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses a HVX multiply resource.

Intrinsics

<code>Vd.h=vmpy(Vu.h,Vv.h):<<1:rnd:sat</code>	<code>HVX_Vector Q6_Vh_vmpy_VhVh_s1_rnd_sat(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.uh=vmpy(Vu.uh,Vv.uh):>>16</code>	<code>HVX_Vector Q6_Vuh_vmpy_VuhVuh_rs16(HVX_Vector Vu, HVX_Vector Vv)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse		u5										d5				
0	0	0	1	1	1	0	0	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.h=vmpy(Vu.h,Vv.h):<<1:md:sat
0	0	0	1	1	1	1	1	1	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.uh=vmpy(Vu.uh,Vv.uh):>>16

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Multiply half of the elements (16 × 16)

Multiply even elements of Vu by odd elements of Vv, shift the result left by 16 bits, and place the result in each lane of Vd. This instruction is useful for 32 × 32 low-half multiplies.

Syntax

```
Vd.w=vmpyieo(Vu.h,Vv.h)
```

Behavior

```
for (i = 0; i < VELEM(32); i++) {
    Vd.w[i] = (Vu.w[i].h[0]*Vv.w[i].h[1]) << 16;
}
```

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses a HVX multiply resource.

Intrinsics

```
Vd.w=vmpyieo(Vu.h,Vv.h)
```

```
HVX_Vector Q6_Vw_vmpyieo_VhVh(HVX_Vector Vu,
HVX_Vector Vv)
```

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
ICLASS																Parse		u5										d5					
0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.w=vmpyieo(Vu.h,Vv.h)	

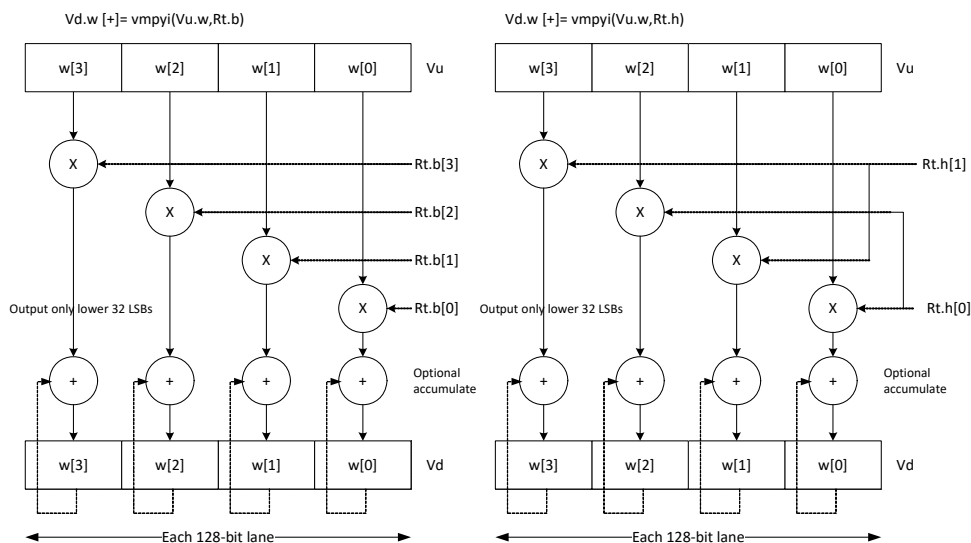
Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Integer multiply by byte

Multiply groups of words in vector register Vu by the elements in Rt. The lower 32-bit results are placed in vector register Vd.

The operation has one forms: signed words in Vu multiplied by signed bytes in Rt.

Optionally accumulates the product with the destination vector register Vx.



Syntax

Vd.h=vmipyi (Vu.h,Rt.b)

Vd.w=vmipyi (Vu.w,Rt.b)

Vd.w=vmipyi (Vu.w,Rt.ub)

Vx.h+=vmipyi (Vu.h,Rt.b)

Vx.w+=vmipyi (Vu.w,Rt.b)

Vx.w+=vmipyi (Vu.w,Rt.ub)

Behavior

```
for (i = 0; i < VELEM(16); i++) {
    Vd.h[i] = (Vu.h[i] * Rt.b[i % 4]);
}
```

```
for (i = 0; i < VELEM(32); i++) {
    Vd.w[i] = (Vu.w[i] * Rt.b[i % 4]);
}
```

```
for (i = 0; i < VELEM(32); i++) {
    Vd.w[i] = (Vu.w[i] * Rt.ub[i % 4]);
}
```

```
for (i = 0; i < VELEM(16); i++) {
    Vx.h[i] += (Vu.h[i] * Rt.b[i % 4]);
}
```

```
for (i = 0; i < VELEM(32); i++) {
    Vx.w[i] += (Vu.w[i] * Rt.b[i % 4]);
}
```

```
for (i = 0; i < VELEM(32); i++) {
    Vx.w[i] += (Vu.w[i] * Rt.ub[i % 4]);
}
```

Class: COPROC_VX (slots 2,3)**Notes**

- This instruction uses a HVX multiply resource.

Intrinsics

Vd.h=vmpyi (Vu.h,Rt.b)	HVX_Vector Q6_Vh_vmpyi_VhRb(HVX_Vector Vu, Word32 Rt)
Vd.w=vmpyi (Vu.w,Rt.b)	HVX_Vector Q6_Vw_vmpyi_VwRb(HVX_Vector Vu, Word32 Rt)
Vd.w=vmpyi (Vu.w,Rt.ub)	HVX_Vector Q6_Vw_vmpyi_VwRub(HVX_Vector Vu, Word32 Rt)
Vx.h+=vmpyi (Vu.h,Rt.b)	HVX_Vector Q6_Vh_vmpyiacc_VhVhRb(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)
Vx.w+=vmpyi (Vu.w,Rt.b)	HVX_Vector Q6_Vw_vmpyiacc_VwVwRb(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)
Vx.w+=vmpyi (Vu.w,Rt.ub)	HVX_Vector Q6_Vw_vmpyiacc_VwVwRub(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS										t5					Parse		u5										x5					
0	0	0	1	1	0	0	1	0	1	0	t	t	t	t	t	P	P	1	u	u	u	u	u	0	1	0	x	x	x	x	x	Vx.w+=vmpyi(Vu.w,Rt.b)
ICLASS										t5					Parse		u5										d5					
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.h=vmpyi(Vu.h,Rt.b)
ICLASS										t5					Parse		u5										x5					
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	1	u	u	u	u	u	0	0	1	x	x	x	x	x	Vx.h+=vmpyi(Vu.h,Rt.b)
ICLASS										t5					Parse		u5										d5					
0	0	0	1	1	0	0	1	1	0	0	t	t	t	t	t	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.w=vmpyi(Vu.w,Rt.ub)
ICLASS										t5					Parse		u5										x5					
0	0	0	1	1	0	0	1	1	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	0	0	1	x	x	x	x	x	Vx.w+=vmpyi(Vu.w,Rt.ub)
ICLASS										t5					Parse		u5										d5					
0	0	0	1	1	0	0	1	1	0	1	t	t	t	t	t	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.w=vmpyi(Vu.w,Rt.b)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t5	Field to encode register t
u5	Field to encode register u
x5	Field to encode register x

Multiply half of the elements with scalar (16 ×16)

Unsigned 16× 16 multiply of the lower halfword of each word in the vector with the lower halfword of the 32-bit scalar.

Syntax	Behavior
<code>Vd.uw=vmpye(Vu.uh,Rt.uh)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.uw[i] = (Vu.uw[i].uh[0] * Rt.uh[0]); }</pre>
<code>Vx.uw+=vmpye(Vu.uh,Rt.uh)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vx.uw[i] += (Vu.uw[i].uh[0] * Rt.uh[0]); }</pre>

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses a HVX multiply resource.

Intrinsics

<code>Vd.uw=vmpye(Vu.uh,Rt.uh)</code>	<code>HVX_Vector Q6_Vuw_vmpye_VuhRuh(HVX_Vector Vu, Word32 Rt)</code>
<code>Vx.uw+=vmpye(Vu.uh,Rt.uh)</code>	<code>HVX_Vector Q6_Vuw_vmpyeacc_VuwVuhRuh(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
ICLASS											t5					Parse		u5										d5					Vd.uw=vmpye(Vu.uh,Rt.uh)
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d		
ICLASS											t5					Parse		u5										x5					Vx.uw+=vmpye(Vu.uh,Rt.uh)
0	0	0	1	1	0	0	1	1	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	0	1	1	x	x	x	x	x		

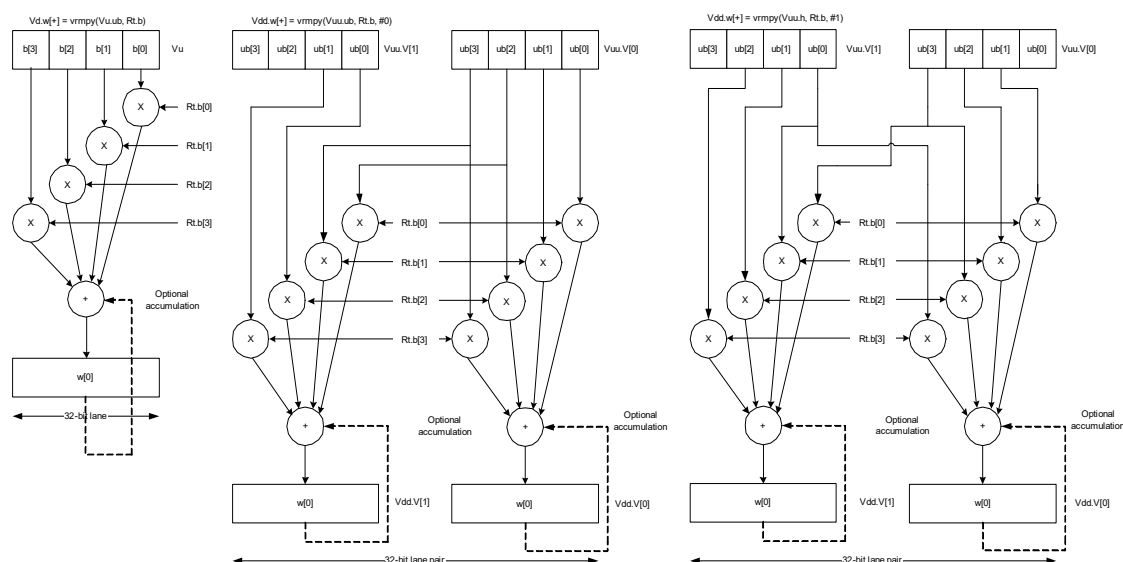
Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t5	Field to encode register t
u5	Field to encode register u
x5	Field to encode register x

Multiply bytes with 4-wide reduction - vector by scalar

Perform multiplication between the elements in vector Vu and the corresponding elements in the scalar register Rt, followed by a 4-way reduction to a word in each 32-bit lane.

Supports the multiplication of unsigned byte data by signed or unsigned bytes in the scalar.

The operation has two forms: the first performs simple dot product of four elements into a single result. The second form takes a one bit immediate input and generates a vector register pair. For #1 = 0, the even destination contains a simple dot product, the odd destination contains a dot product of the coefficients rotated by two elements and the upper two data elements taken from the even register of Vuu. For #u = 1, the even destination takes coefficients rotated by -1 and data element 0 from the odd register of Vuu. The odd destination uses coefficients rotated by -1 and takes data element 3 from the even register of Vuu.



Syntax

`Vd.uw=vrmppy(Vu.ub,Rt.ub)`

`Vd.w=vrmppy(Vu.ub,Rt.b)`

`Vx.uw+=vrmppy(Vu.ub,Rt.ub)`

Behavior

```
for (i = 0; i < VELEM(32); i++) {
    Vd.uw[i] = (Vu.uw[i].ub[0] * Rt.ub[0]);
    Vd.uw[i] += (Vu.uw[i].ub[1] * Rt.ub[1]);
    Vd.uw[i] += (Vu.uw[i].ub[2] * Rt.ub[2]);
    Vd.uw[i] += (Vu.uw[i].ub[3] * Rt.ub[3]);
}
```

```
for (i = 0; i < VELEM(32); i++) {
    Vd.w[i] = (Vu.uw[i].ub[0] * Rt.b[0]);
    Vd.w[i] += (Vu.uw[i].ub[1] * Rt.b[1]);
    Vd.w[i] += (Vu.uw[i].ub[2] * Rt.b[2]);
    Vd.w[i] += (Vu.uw[i].ub[3] * Rt.b[3]);
}
```

```
for (i = 0; i < VELEM(32); i++) {
    Vx.uw[i] += (Vu.uw[i].ub[0] * Rt.ub[0]);
    Vx.uw[i] += (Vu.uw[i].ub[1] * Rt.ub[1]);
    Vx.uw[i] += (Vu.uw[i].ub[2] * Rt.ub[2]);
    Vx.uw[i] += (Vu.uw[i].ub[3] * Rt.ub[3]);
}
```

Syntax	Behavior
<code>Vx.w+=vrmpy(Vu.ub,Rt.b)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vx.w[i] += (Vu.uw[i].ub[0] * Rt.b[0]); Vx.w[i] += (Vu.uw[i].ub[1] * Rt.b[1]); Vx.w[i] += (Vu.uw[i].ub[2] * Rt.b[2]); Vx.w[i] += (Vu.uw[i].ub[3] * Rt.b[3]); } </pre>

Class: COPROC_VX (slots 2,3)**Notes**

- This instruction uses a HVX multiply resource.

Intrinsics

<code>Vd.uw=vrmpy(Vu.ub,Rt.ub)</code>	<code>HVX_Vector Q6_Vuw_vrmpy_VubRub(HVX_Vector Vu, Word32 Rt)</code>
<code>Vd.w=vrmpy(Vu.ub,Rt.b)</code>	<code>HVX_Vector Q6_Vw_vrmpy_VubRb(HVX_Vector Vu, Word32 Rt)</code>
<code>Vx.uw+=vrmpy(Vu.ub,Rt.ub)</code>	<code>HVX_Vector Q6_Vuw_vrmpyacc_VuwVubRub(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)</code>
<code>Vx.w+=vrmpy(Vu.ub,Rt.b)</code>	<code>HVX_Vector Q6_Vw_vrmpyacc_VwVubRb(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)</code>

Encoding

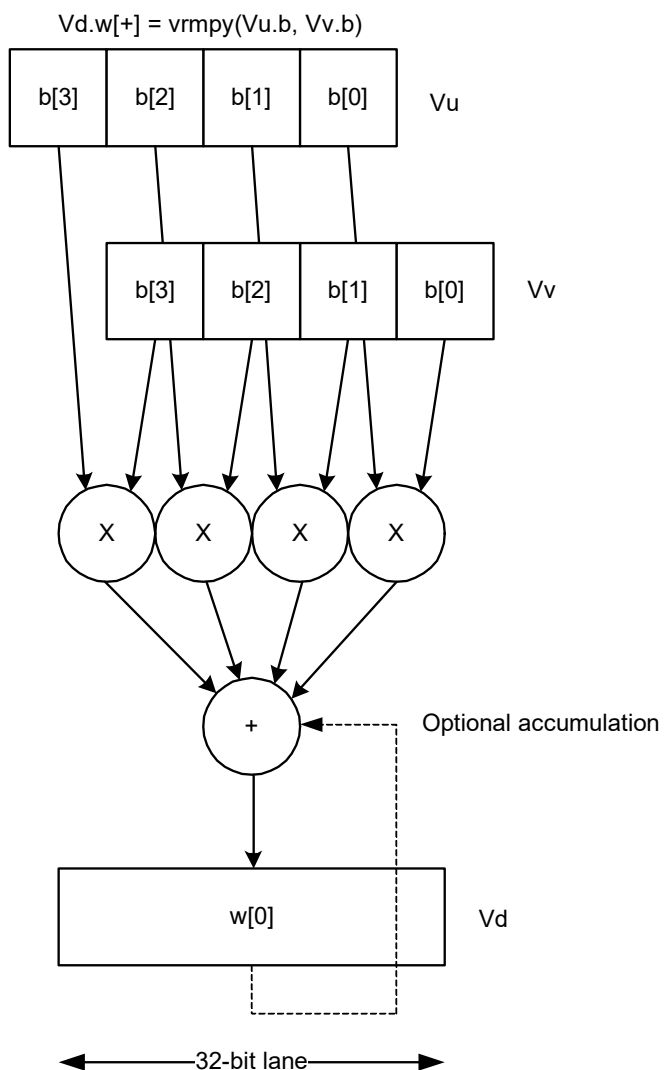
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											t5					Parse		u5										d5				
0	0	0	1	1	0	0	1	0	0	0	t	t	t	t	t	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.uw=vrmpy(Vu.ub,Rt.ub)
0	0	0	1	1	0	0	1	0	0	0	t	t	t	t	t	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.w=vrmpy(Vu.ub,Rt.b)
ICLASS											t5					Parse		u5										x5				
0	0	0	1	1	0	0	1	0	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	1	0	0	x	x	x	x	x	Vx.uw+=vrmpy(Vu.ub,Rt.ub)
0	0	0	1	1	0	0	1	0	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	1	0	1	x	x	x	x	x	Vx.w+=vrmpy(Vu.ub,Rt.b)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t5	Field to encode register t
u5	Field to encode register u
x5	Field to encode register x

Multiply by byte with 4-wide reduction - vector by vector

The `vrmpy` instruction performs a dot product function between 4-byte elements in vector register `Vu`, and 4-byte elements in `Vv`. The sum of the products is written into `Vd` as words within each 32-bit lane.

Data types can be unsigned by unsigned, signed by signed or unsigned by signed.



Syntax

```
Vd.uw=vrmpy(Vu.ub,Vv.ub)
```

Behavior

```
for (i = 0; i < VELEM(32); i++) {
    Vd.uw[i] = (Vu.uw[i].ub[0] * Vv.uw[i].ub[0]);
    Vd.uw[i] += (Vu.uw[i].ub[1] * Vv.uw[i].ub[1]);
    Vd.uw[i] += (Vu.uw[i].ub[2] * Vv.uw[i].ub[2]);
    Vd.uw[i] += (Vu.uw[i].ub[3] * Vv.uw[i].ub[3]);
}
```

Syntax	Behavior
<code>Vd.w=vrmpy(Vu.b,Vv.b)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vd.w[i] = (Vu.w[i].b[0] * Vv.w[i].b[0]); Vd.w[i] += (Vu.w[i].b[1] * Vv.w[i].b[1]); Vd.w[i] += (Vu.w[i].b[2] * Vv.w[i].b[2]); Vd.w[i] += (Vu.w[i].b[3] * Vv.w[i].b[3]); } </pre>
<code>Vd.w=vrmpy(Vu.ub,Vv.b)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vd.w[i] = (Vu.uw[i].ub[0] * Vv.w[i].b[0]); Vd.w[i] += (Vu.uw[i].ub[1] * Vv.w[i].b[1]); Vd.w[i] += (Vu.uw[i].ub[2] * Vv.w[i].b[2]); Vd.w[i] += (Vu.uw[i].ub[3] * Vv.w[i].b[3]); } </pre>

Class: COPROC_VX (slots 2,3)**Notes**

- This instruction uses a HVX multiply resource.

Intrinsics

<code>Vd.uw=vrmpy(Vu.ub,Vv.ub)</code>	<code>HVX_Vector Q6_Vuw_vrmpy_VubVub(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.w=vrmpy(Vu.b,Vv.b)</code>	<code>HVX_Vector Q6_Vw_vrmpy_VbVb(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.w=vrmpy(Vu.ub,Vv.b)</code>	<code>HVX_Vector Q6_Vw_vrmpy_VubVb(HVX_Vector Vu, HVX_Vector Vv)</code>

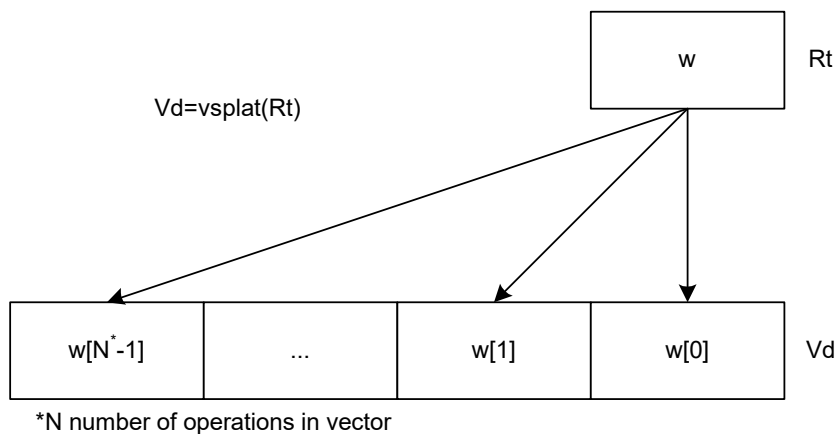
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
ICLASS																Parse		u5										d5					
0	0	0	1	1	1	0	0	0	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.uw=vrmpy(Vu.ub,Vv.ub)
0	0	0	1	1	1	0	0	0	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.w=vrmpy(Vu.b,Vv.b)
0	0	0	1	1	1	0	0	0	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.w=vrmpy(Vu.ub,Vv.b)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Splat from scalar

Set all destination vector register words to the value specified by the contents of scalar register Rt.



Syntax	Behavior
<code>Vd.b=vsplat(Rt)</code>	<pre>for (i = 0; i < VELEM(8); i++) { Vd.ub[i] = Rt ; }</pre>
<code>Vd.h=vsplat(Rt)</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = Rt ; }</pre>
<code>Vd=vsplat(Rt)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.uw[i] = Rt ; }</pre>

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses a HVX multiply resource.

Intrinsics

<code>Vd.b=vsplat(Rt)</code>	<code>HVX_Vector Q6_Vb_vsplat_R(Word32 Rt)</code>
<code>Vd.h=vsplat(Rt)</code>	<code>HVX_Vector Q6_Vh_vsplat_R(Word32 Rt)</code>
<code>Vd=vsplat(Rt)</code>	<code>HVX_Vector Q6_V_vsplat_R(Word32 Rt)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											t5					Parse									d5							
0	0	0	1	1	0	0	1	1	0	1	t	t	t	t	t	P	P	0	-	-	-	-	0	0	0	1	d	d	d	d	d	Vd=vsplat(Rt)
0	0	0	1	1	0	0	1	1	1	0	t	t	t	t	t	P	P	0	-	-	-	-	-	0	0	1	d	d	d	d	d	Vd.h=vsplat(Rt)
0	0	0	1	1	0	0	1	1	1	0	t	t	t	t	t	P	P	0	-	-	-	-	-	0	1	0	d	d	d	d	d	Vd.b=vsplat(Rt)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t5	Field to encode register t

Vector to predicate transfer

Copy bits into the destination vector predicate register, under the control of the scalar register Rt and the input vector register Vu. Instead of a direct write, the destination can also be OR'd with the result. If the corresponding byte i of Vu matches any of the bits in Rt byte[i%4], the destination Qd is OR'd with or set to 1 or 0.

If Rt contains 0x01010101, Qt can effectively be filled with the LSBs of Vu, one bit per byte.

Syntax	Behavior
<code>Qd4=vand(Vu,Rt)</code>	<pre>for (i = 0; i < VELEM(8); i++) { QdV[i]=((Vu.ub[i] & Rt.ub[i % 4]) != 0) ? 1 : 0 ; }</pre>
<code>Qx4 =vand(Vu,Rt)</code>	<pre>for (i = 0; i < VELEM(8); i++) { QxV[i]=QxV[i] ((Vu.ub[i] & Rt.ub[i % 4]) != 0) ? 1 : 0); }</pre>

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses a HVX multiply resource.

Intrinsics

<code>Qd4=vand(Vu,Rt)</code>	<code>HVX_VectorPred Q6_Q_vand_VR(HVX_Vector Vu, Word32 Rt)</code>
<code>Qx4 =vand(Vu,Rt)</code>	<code>HVX_VectorPred Q6_Q_vandor_QVR(HVX_VectorPred Qx, HVX_Vector Vu, Word32 Rt)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS												t5				Parse		u5												x2		
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	1	u	u	u	u	u	1	0	0	-	-	-	x	x	Qx4 =vand(Vu,Rt)
ICLASS												t5				Parse		u5												d2		
0	0	0	1	1	0	0	1	1	0	1	t	t	t	t	t	P	P	0	u	u	u	u	u	0	1	0	-	1	0	d	d	Qd4=vand(Vu,Rt)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d2	Field to encode register d
t5	Field to encode register t
u5	Field to encode register u
x2	Field to encode register x

Predicate to vector transfer

Copy the byte elements of scalar register Rt into the destination vector register Vd, under the control of the vector predicate register. Instead of a direct write, the destination can also be OR'd with the result. If the corresponding bit i of Qu is set, the contents of byte[i % 4] are written or OR'd into Vd or Vx. If Rt contains 0x01010101, Qt can effectively be expanded into Vd or Vx, one bit per byte.

Syntax	Behavior
Vd=vand(!Qu4,Rt)	<pre>for (i = 0; i < VELEM(8); i++) { Vd.ub[i] = !QuV[i] ? Rt.ub[i % 4] : 0; }</pre>
Vx =vand(!Qu4,Rt)	<pre>for (i = 0; i < VELEM(8); i++) { Vx.ub[i] = !(QuV[i]) ? Rt.ub[i % 4] : 0; }</pre>

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses a HVX multiply resource.

Intrinsics

Vd=vand(!Qu4,Rt)	HVX_Vector Q6_V_vand_QnR(HVX_VectorPred Qu, Word32 Rt)
Vd=vand(Qu4,Rt)	HVX_Vector Q6_V_vand_QR(HVX_VectorPred Qu, Word32 Rt)
Vx =vand(!Qu4,Rt)	HVX_Vector Q6_V_vandor_VQnR(HVX_Vector Vx, HVX_VectorPred Qu, Word32 Rt)
Vx =vand(Qu4,Rt)	HVX_Vector Q6_V_vandor_VQR(HVX_Vector Vx, HVX_VectorPred Qu, Word32 Rt)

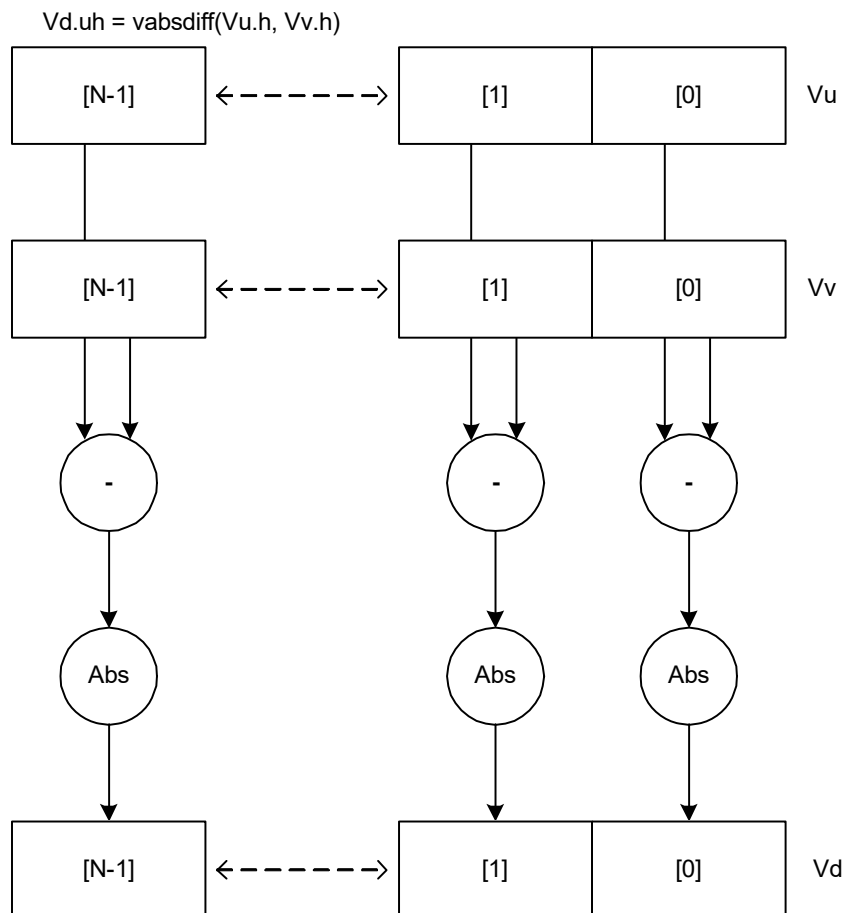
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											t5				Parse							u2							x5			
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	1	-	-	0	u	u	0	1	1	x	x	x	x	x	Vx =vand(Qu4,Rt)
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	1	-	-	1	u	u	0	1	1	x	x	x	x	x	Vx =vand(!Qu4,Rt)
ICLASS											t5				Parse							u2							d5			
0	0	0	1	1	0	0	1	1	0	1	t	t	t	t	t	P	P	0	-	-	0	u	u	1	0	1	d	d	d	d	d	Vd=vand(Qu4,Rt)
0	0	0	1	1	0	0	1	1	0	1	t	t	t	t	t	P	P	0	-	-	1	u	u	1	0	1	d	d	d	d	d	Vd=vand(!Qu4,Rt)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t5	Field to encode register t
u2	Field to encode register u
x5	Field to encode register x

Absolute value of difference

Return the absolute value of the difference between corresponding elements in vector registers Vu and Vv, and place the result in Vd. Supports unsigned byte, signed and unsigned halfword, and signed word.



N is the number of elements implemented in a vector register.

Syntax	Behavior
<code>Vd.ub=vabsdiff(Vu.ub,Vv.ub)</code>	<pre>for (i = 0; i < VELEM(8); i++) { Vd.ub[i] = (Vu.ub[i] > Vv.ub[i]) ? (Vu.ub[i] - Vv.ub[i]) : (Vv.ub[i] - Vu.ub[i]); }</pre>
<code>Vd.uh=vabsdiff(Vu.h,Vv.h)</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = (Vu.h[i] > Vv.h[i]) ? (Vu.h[i] - Vv.h[i]) : (Vv.h[i] - Vu.h[i]); }</pre>
<code>Vd.uh=vabsdiff(Vu.uh,Vv.uh)</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = (Vu.uh[i] > Vv.uh[i]) ? (Vu.uh[i] - Vv.uh[i]) : (Vv.uh[i] - Vu.uh[i]); }</pre>

Syntax

```
Vd.uw=vabsdiff(Vu.w,Vv.w)
```

Behavior

```
for (i = 0; i < VELEM(32); i++) {
    Vd.uw[i] = (Vu.w[i] > Vv.w[i]) ? (Vu.w[i] -
    Vv.w[i]) : (Vv.w[i] - Vu.w[i]);
}
```

Class: COPROC_VX (slots 2,3)**Notes**

- This instruction uses a HVX multiply resource.

Intrinsics

```
Vd.ub=vabsdiff(Vu.ub,Vv.ub) HVX_Vector Q6_Vub_vabsdiff_VubVub(HVX_Vector Vu,
HVX_Vector Vv)
```

```
Vd.uh=vabsdiff(Vu.h,Vv.h) HVX_Vector Q6_Vuh_vabsdiff_VhVh(HVX_Vector Vu,
HVX_Vector Vv)
```

```
Vd.uh=vabsdiff(Vu.uh,Vv.uh) HVX_Vector Q6_Vuh_vabsdiff_VuhVuh(HVX_Vector Vu,
HVX_Vector Vv)
```

```
Vd.uw=vabsdiff(Vu.w,Vv.w) HVX_Vector Q6_Vuw_vabsdiff_VwVw(HVX_Vector Vu,
HVX_Vector Vv)
```

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS															Parse		u5					d5										
0	0	0	1	1	1	0	0	1	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.ub=vabsdiff(Vu.ub,Vv.ub)
0	0	0	1	1	1	0	0	1	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.uh=vabsdiff(Vu.h,Vv.h)
0	0	0	1	1	1	0	0	1	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.uh=vabsdiff(Vu.uh,Vv.uh)
0	0	0	1	1	1	0	0	1	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.uw=vabsdiff(Vu.w,Vv.w)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Insert element

Insert a 32-bit element in Rt into the destination vector register Vx, at the word element 0.

Syntax

```
Vx.w=vininsert(Rt)
```

Behavior

```
Vx.uw[0] = Rt;
```

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses a HVX multiply resource.

Intrinsics

```
Vx.w=vininsert(Rt)
```

```
HVX_Vector Q6_Vw_vininsert_VwR(HVX_Vector Vx,  
Word32 Rt)
```

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											t5					Parse										x5						
0	0	0	1	1	0	0	1	1	0	1	t	t	t	t	t	P	P	1	-	-	-	-	-	0	0	1	x	x	x	x	x	Vx.w=vininsert(Rt)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
t5	Field to encode register t
x5	Field to encode register x

6.10 HVX permute resource

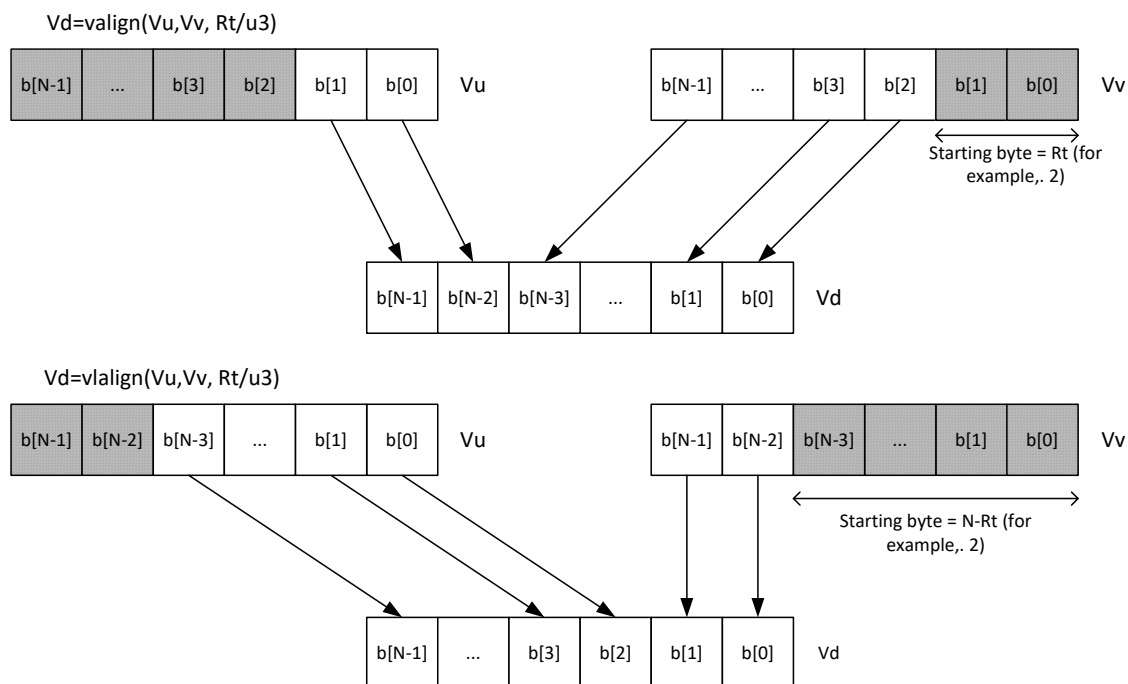
The HVX permute resource instruction subclass includes instructions that use the HVX permute resource.

Byte alignment

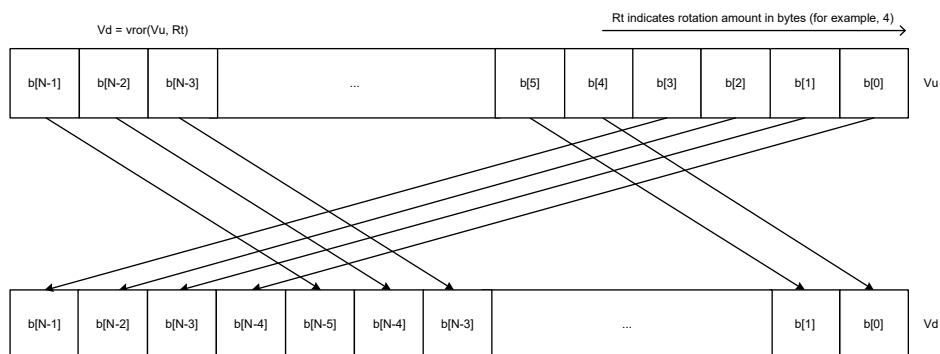
Select a continuous group of bytes the size of a vector register from vector registers Vu and Vv. The starting location is provided by the lower bits of Rt (modulo the vector length) or by a 3-bit immediate value.

There are two forms of the operation, The first, `valign`, uses the Rt or immediate input directly to specify the beginning of the block. The second, `vlalign`, uses the inverse of the input value by subtracting it from the vector length.

The operation can be used to implement a non-aligned vector load, using two aligned loads (above and below the pointer) and a `valign` where the pointer is used as the control input.



Perform a right rotate vector operation on vector register Vu, by the number of bytes specified by the lower bits of Rt. The result is written into Vd. Byte[i] moves to Byte[(i+N-R)%N], where R is the right rotate amount in bytes, and N is the vector register size in bytes.

**Syntax****Behavior**

Vd=valign(Vu,Vv,#u3)

```
for(i = 0; i < VWIDTH; i++) {
    Vd.ub[i] = (i+#u>=VWIDTH) ? Vu.ub[i+#u-VWIDTH] :
    Vv.ub[i+#u];
}
```

Vd=valign(Vu,Vv,Rt)

```
unsigned shift = Rt & (VWIDTH-1);
for(i = 0; i < VWIDTH; i++) {
    Vd.ub[i] = (i+shift>=VWIDTH) ? Vu.ub[i+shift-
    VWIDTH] : Vv.ub[i+shift];
}
```

Vd=vlalign(Vu,Vv,#u3)

```
unsigned shift = VWIDTH - #u;
for(i = 0; i < VWIDTH; i++) {
    Vd.ub[i] = (i+shift>=VWIDTH) ? Vu.ub[i+shift-
    VWIDTH] : Vv.ub[i+shift];
}
```

Vd=vlalign(Vu,Vv,Rt)

```
unsigned shift = VWIDTH - (Rt & (VWIDTH-1));
for(i = 0; i < VWIDTH; i++) {
    Vd.ub[i]=(i+shift>=VWIDTH) ? Vu.ub[i+shift-VWIDTH] :
    Vv.ub[i+shift];
}
```

Vd=vror(Vu,Rt)

```
for (k=0;k<VWIDTH;k++) {
    Vd.ub[k] = Vu.ub[(k+Rt) & (VWIDTH-1)];
}
```

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction uses the HVX permute resource.
- Input scalar register Rt is limited to registers 0 through 7

Intrinsics

<code>Vd=valign(Vu,Vv,#u3)</code>	<code>HVX_Vector Q6_V_valign_VVI(HVX_Vector Vu, HVX_Vector Vv, Word32 Iu3)</code>
<code>Vd=valign(Vu,Vv,Rt)</code>	<code>HVX_Vector Q6_V_valign_VVR(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)</code>
<code>Vd=vlalign(Vu,Vv,#u3)</code>	<code>HVX_Vector Q6_V_vlalign_VVI(HVX_Vector Vu, HVX_Vector Vv, Word32 Iu3)</code>
<code>Vd=vlalign(Vu,Vv,Rt)</code>	<code>HVX_Vector Q6_V_vlalign_VVR(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)</code>
<code>Vd=vrór(Vu,Rt)</code>	<code>HVX_Vector Q6_V_vróór_VR(HVX_Vector Vu, Word32 Rt)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS												t5				Parse		u5										d5				
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd=vrór(Vu,Rt)
ICLASS												t3				Parse		u5										d5				
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd=valign(Vu,Vv,Rt)
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd=vlalign(Vu,Vv,Rt)
ICLASS																Parse		u5										d5				
0	0	0	1	1	1	1	0	0	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	i	i	i	d	d	d	d	d	Vd=valign(Vu,Vv,#u3)
0	0	0	1	1	1	1	0	0	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	i	i	i	d	d	d	d	d	Vd=vlalign(Vu,Vv,#u3)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t3	Field to encode register t
t5	Field to encode register t
u5	Field to encode register u

General permute network

Perform permutation and re-arrangement of the 64 input bytes, which is the width of a data slice. The input data is passed through a network of switch boxes, these are able to take two inputs and based on the two controls can pass through, swap, replicate the first input, or replicate the second input. Though the functionality is powerful the algorithms to compute the controls are complex.

The input vector of bytes is passed through six levels of switches that have an increasing stride varying from 1 to 32 at the last stage.

There are two main forms of data rearrangement. One uses a simple reverse butterfly network shown as `vrdelta`, and a butterfly network `vdelta` shown in [Figure 6-1](#). These are known as blocking networks, as not all possible paths can be allowed, simultaneously from input to output. The data does not have to be a permutation, defined as a one-to-one mapping of every input to its own output position. A subset of data rearrangement such as data replication can be accommodated. It can handle a family of patterns that have symmetric properties.

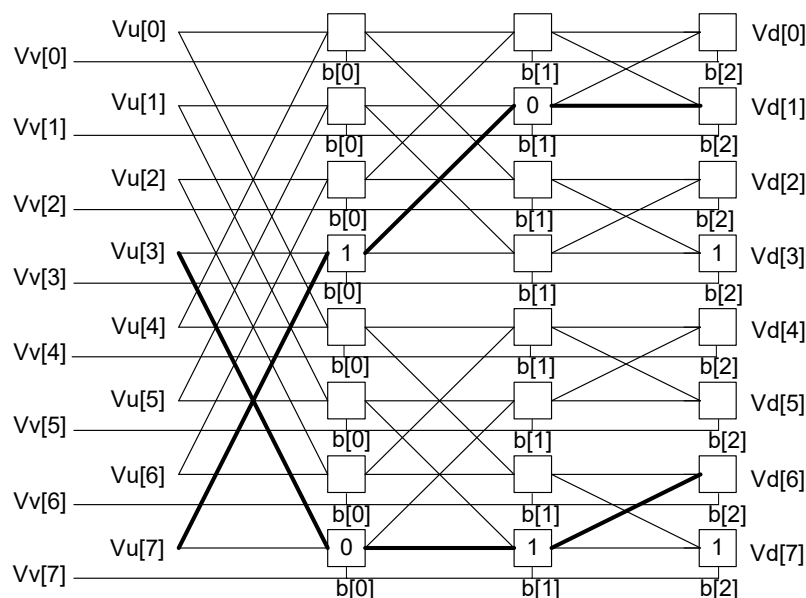


Figure 6-1 vdelta example

[Figure 6-2](#) shows an example of such a valid pattern using an 8-element `vrdelta` network for clarity: 0, 2, 4, 6, 7, 5, 3, 1.

[Figure 6-2](#) shows the `vrdelta` network, the `vdelta` network is the mirror image, with the largest stride first followed by smaller strides down to 1. Each stage output is controlled by the control inputs in the vector register `Vv`. For each stage (for example stage 3), the bit at that position looks at the corresponding bit (bit 3) in the control byte. This is shown in the switch box in the diagram.

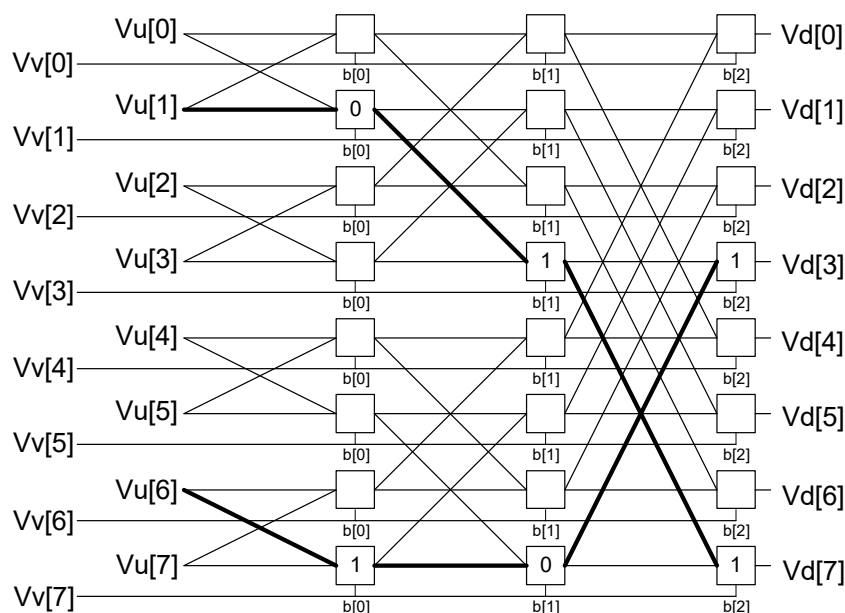


Figure 6-2 vrdelta example

The desired pattern 0, 2, 4, 6, 1, 3, 5, 7 is not possible, as this overuses available paths in the trellis. The position of the output for a particular input is determined by using the bit sequence produced by the destination position D from source position S. The bit vector for the path through the trellis is a function of this destination bit sequence. In the example $D = 7$, $S = 1$, the element in position 1 is to be moved to position 7. The first switch box control bit at position 1 is 0, the next control bit at position 3 is 1, and finally the bit at position 7 is 1, yielding the sequence 0, 1, 1. Also, element 6 is moved to position 3, with the control vector 1, 0, 1. Bits must be placed at the appropriate position in the control bytes to guide the inputs to the desired positions. Every input can be placed into any output, but certain combinations conflict for resources, and so the rearrangement is not possible. A total of 512 control bits are required for a single vrdelta or vdelta slice.

Example of a permitted arrangement:

0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,52,54,56,58,60,62,63,
61,59,57,55,53,51,49,47,45,43,41,39,37,35,33,31,29,27,25,23,21,19,17,15,13,11,9,7,5,3,1

controls =

{0x00,0x02,0x05,0x07,0x0A,0x08,0x0F,0x0D,0x14,0x16,0x11,0x13,0x1E,0x1C,0x1B,0x19,0x28,0x2A,0x2D,0x2F,0x22,0x20,0x27,0x25,0x3C,0x3E,0x39,0x3B,0x36,0x34,0x33,0x31,0x10,0x12,0x15,0x17,0x1A,0x18,0x1F,0x1D,0x04,0x06,0x01,0x03,0x0E,0x0C,0x0B,0x09,0x38,0x3A,0x3D,0x3F,0x32,0x30,0x37,0x35,0x2C,0x2E,0x29,0x2B,0x26,0x24,0x23,0x21}

Similarly, here is a function that replicates every fourth element:

0,0,0,0,4,4,4,4,8,8,8,8,12,12,12,12,16,16,16,16,20,20,20,20,24,24,24,24,28,28,28,28,32,32,32,32,
36,36,36,36,40,40,40,40,44,44,44,44,48,48,48,48,52,52,52,52,56,56,56,56,60,60,60,60

[illegible]

The other general form of permute is a Benes network, which requires a vrdelta immediately followed by a vdelta operation. This form is non-blocking: any possible permute, however random, can be accommodated, though it must be a permutation, each input must have a position in the output. Replication can be performed by using a pre- or post-conditioning vrdelta pass to perform the replications before or after the permute.

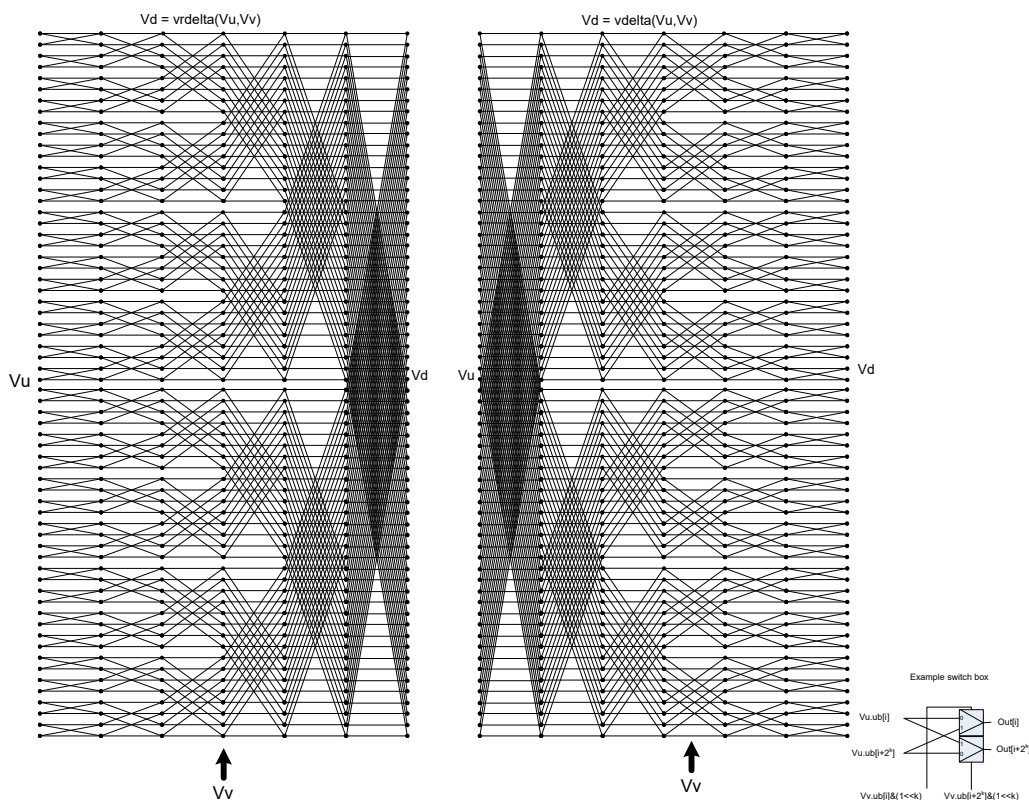
Element sizes larger than a byte can be implemented by grouping bytes together and moving them to a group in the output. An example of a general permute is the following random mix, where the 64 inputs are put in the following output positions: 33,42,40,61,28, 6,17,16,12,38,57,21,58,63,37,13,26,51,50,23,46, 5,52,53, 0,25,39, 7,10,19,18,56,44,41,11,14,43,45, 3,35,32,60,15,55,22,24,48, 9, 4,31,27, 8, 2,62,30,34,54,20,49,59,29,47,36

```
vrdelta controls = {0x00, 0x00, 0x21, 0x21, 0x20, 0x02, 0x00, 0x02, 0x20, 0x22, 0x00, 0x06, 0x23,
0x23, 0x02, 0x26, 0x06, 0x04, 0x2A, 0x0C, 0x2D, 0x2F, 0x20, 0x2E, 0x04, 0x00, 0x09, 0x29, 0x0C,
0x0A, 0x20, 0x0A, 0x05, 0x0F, 0x29, 0x2B, 0x2C, 0x0E, 0x11, 0x13, 0x31, 0x2F, 0x08, 0x0A, 0x2A,
0x3E, 0x02, 0x32, 0x0B, 0x07, 0x26, 0x0E, 0x2A, 0x2E, 0x36, 0x36, 0x1D, 0x07, 0x01, 0x2B, 0x0C,
0x1E, 0x21, 0x13}
```

```
vdelta controls={ 0x1D, 0x01, 0x00, 0x00, 0x1D, 0x1B, 0x00, 0x1A, 0x1E, 0x02, 0x13, 0x03, 0x0C,
0x18, 0x10, 0x08, 0x1A, 0x06, 0x07, 0x03, 0x11, 0x1D, 0x0D, 0x11, 0x19, 0x03, 0x15, 0x03, 0x03,
0x19, 0x1F, 0x01, 0x1B, 0x1B, 0x06, 0x12, 0x18, 0x00, 0x1D, 0x09, 0x1A, 0x0E, 0x02, 0x02, 0x0B,
0x05, 0x0A, 0x18, 0x1D, 0x1F, 0x01, 0x17, 0x14, 0x06, 0x19, 0x0F, 0x1D, 0x0D, 0x05, 0x01, 0x06,
0x06, 0x0F, 0x1B}
```

Use these applications to find your vdelta/vrdelta controls for a Benes-type network or vrdelta only for a simple Delta network.

For the Benes control, all outputs must be used. In the Delta network, X is a don't-care output and replication is allowed.

**Syntax**

`Vd=vdelta(Vu,Vv)`

```
for (offset=VWIDTH; (offset>=1)>0; ) {
    for (k = 0; k<VWIDTH; k++) {
        Vd.ub[k] = (Vv.ub[k]&offset) ? Vu.ub[k^offset] :
Vu.ub[k];
    }
    for (k = 0; k<VWIDTH; k++) {
        Vu.ub[k] = Vd.ub[k];
    }
}
```

`Vd=vrdelta(Vu,Vv)`

```
for (offset=1; offset<VWIDTH; offset<=1){
    for (k = 0; k<VWIDTH; k++) {
        Vd.ub[k] = (Vv.ub[k]&offset) ? Vu.ub[k^offset] :
Vu.ub[k];
    }
    for (k = 0; k<VWIDTH; k++) {
        Vu.ub[k] = Vd.ub[k];
    }
}
```

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction uses the HVX permute resource.

Intrinsics

<code>Vd=vdelta(Vu,Vv)</code>	<code>HVX_Vector Q6_V_vdelta_VV(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd=vrdelta(Vu,Vv)</code>	<code>HVX_Vector Q6_V_vrdelta_VV(HVX_Vector Vu, HVX_Vector Vv)</code>

Encoding

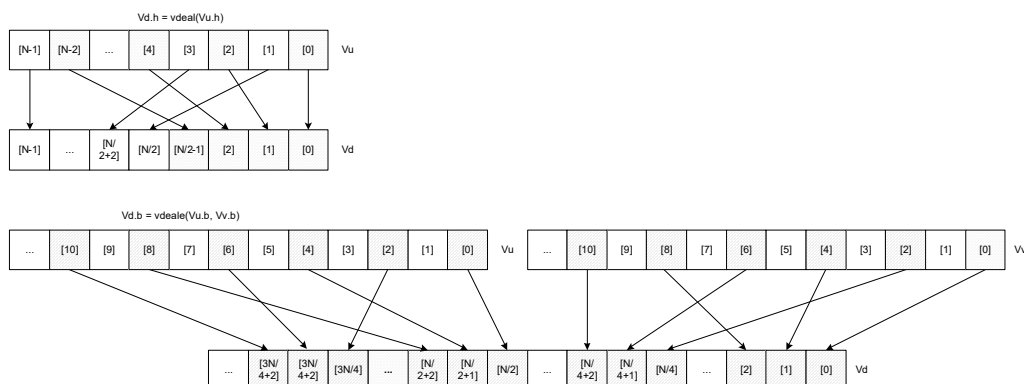
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
ICLASS																Parse								u5								d5				
0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd=vdelta(Vu,Vv)				
0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd=vrdelta(Vu,Vv)				

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Shuffle - deal

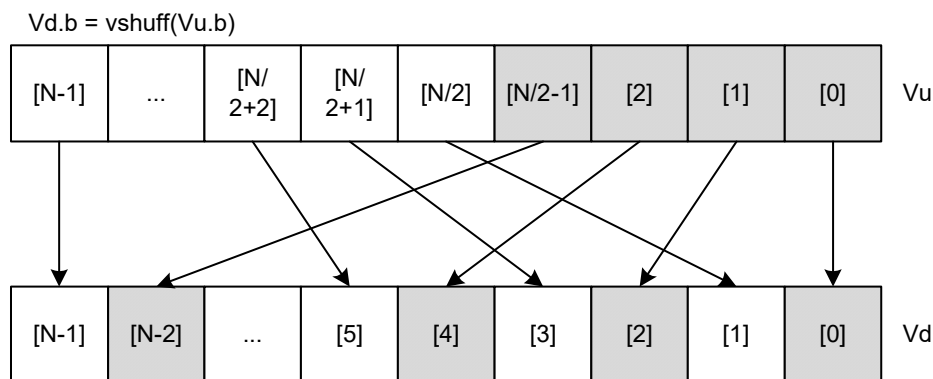
Deal or deinterleave the elements into the destination register Vd. Even elements of Vu are placed in the lower half of Vd, and odd elements are placed in the upper half.

For `vdeale`, the even elements of Vv are dealt into the lower half of the destination vector register Vd, and the even elements of Vu are dealt into the upper half of Vd. The deal operation takes even-even elements of Vv and places them in the lower quarter of Vd, while odd-even elements of Vv are placed in the second quarter of Vd. Similarly, even-even elements of Vu are placed in the third quarter of Vd, while odd-even elements of Vu are placed in the fourth quarter of Vd.



Shuffle elements within a vector. Elements from the same position - but in the upper half of the vector register - are packed together in even and odd element pairs, and then placed in the destination vector register Vd.

Supports byte and halfword. Operates on a single register input, similar to `vshuffoe`.



*N is the number of element operations allowed in the vector

Syntax

```
Vd.b=vdeal(Vu.b)
```

Behavior

```
for (i = 0; i < VELEM(16); i++) {
    Vd.ub[i] = Vu.uh[i].ub[0];
    Vd.ub[i+VBITS/16] = Vu.uh[i].ub[1] ;
}
```

Syntax	Behavior
Vd.b=vdeale(Vu.b,Vv.b)	<pre> for (i = 0; i < VELEM(32); i++) { Vd.ub[0+i] = Vv.uw[i].ub[0]; Vd.ub[VBITS/32+i] = Vv.uw[i].ub[2]; Vd.ub[2*VBITS/32+i] = Vu.uw[i].ub[0]; Vd.ub[3*VBITS/32+i] = Vu.uw[i].ub[2] ; } </pre>
Vd.b=vshuff(Vu.b)	<pre> for (i = 0; i < VELEM(16); i++) { Vd.uh[i].b[0]=Vu.ub[i]; Vd.uh[i].b[1]=Vu.ub[i+VBITS/16] ; } </pre>
Vd.h=vdeal(Vu.h)	<pre> for (i = 0; i < VELEM(32); i++) { Vd.uh[i] = Vu.uw[i].uh[0]; Vd.uh[i+VBITS/32] = Vu.uw[i].uh[1] ; } </pre>
Vd.h=vshuff(Vu.h)	<pre> for (i = 0; i < VELEM(32); i++) { Vd.uw[i].h[0]=Vu.uh[i]; Vd.uw[i].h[1]=Vu.uh[i+VBITS/32] ; } </pre>

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction uses the HVX permute resource.

Intrinsics

Vd.b=vdeal(Vu.b)	HVX_Vector Q6_Vb_vdeal_Vb(HVX_Vector Vu)
Vd.b = vdeale(Vu.b,Vv.b)	HVX_Vector Q6_Vb_vdeale_VbVb(HVX_Vector Vu, HVX_Vector Vv)
Vd.b=vshuff(Vu.b)	HVX_Vector Q6_Vb_vshuff_Vb(HVX_Vector Vu)
Vd.h=vdeal(Vu.h)	HVX_Vector Q6_Vh_vdeal_Vh(HVX_Vector Vu)
Vd.h=vshuff(Vu.h)	HVX_Vector Q6_Vh_vshuff_Vh(HVX_Vector Vu)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse				u5							d5					
0	0	0	1	1	1	1	0	-	-	0	-	-	0	0	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.h=vdeal(Vu.h)	
0	0	0	1	1	1	1	0	-	-	0	-	-	0	0	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.b=vdeal(Vu.b)	
0	0	0	1	1	1	1	0	-	-	0	-	-	0	1	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.h=vshuff(Vu.h)	
0	0	0	1	1	1	1	0	-	-	0	-	-	1	0	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.b=vshuff(Vu.b)	
0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.b=vdeale(Vu.b,Vv.b)

Field name		Description
ICLASS		Instruction class
Parse		Packet loop parse bits
d5		Field to encode register d
u5		Field to encode register u
v5		Field to encode register v

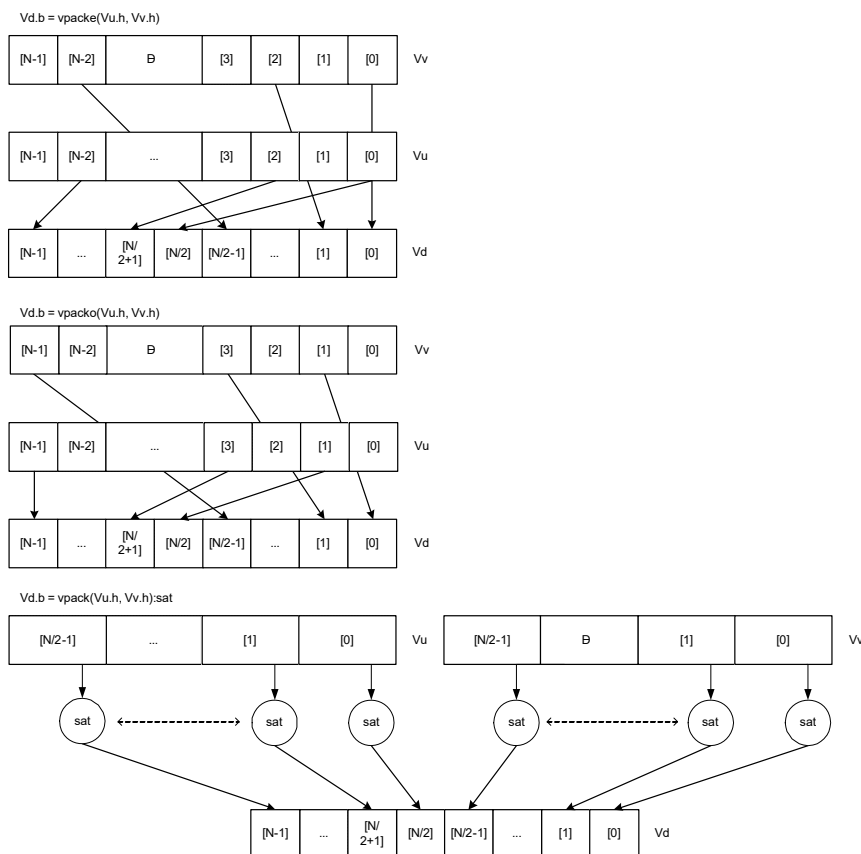
Pack

The `vpack` operation has three forms. All of them pack elements from the vector registers `Vu` and `Vv` into the destination vector register `Vd`.

The `vpacke` instruction writes even elements from `Vv` and `Vu` into the lower half and upper half of `Vd` respectively.

The `vpacko` instruction writes odd elements from `Vv` and `Vu` into the lower half and upper half of `Vd` respectively.

The `vpack` instruction takes all elements from `Vv` and `Vu`, saturates them to the next smallest element size, and writes them into `Vd`.



Syntax

```
Vd.b=vpack(Vu.h,Vv.h):sat
```

```
Vd.b=vpacke(Vu.h,Vv.h)
```

Behavior

```
for (i = 0; i < VELEM(16); i++) {
    Vd.b[i] = sat8(Vv.h[i]);
    Vd.b[i+VBITS/16] = sat8(Vu.h[i]);
}
```

```
for (i = 0; i < VELEM(16); i++) {
    Vd.ub[i] = Vv.uh[i].ub[0];
    Vd.ub[i+VBITS/16] = Vu.uh[i].ub[0];
}
```

Syntax	Behavior
<code>Vd.b=vpacko(Vu.h,Vv.h)</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.ub[i] = Vv.uh[i].ub[1]; Vd.ub[i+VBITS/16] = Vu.uh[i].ub[1] ; }</pre>
<code>Vd.h=vpack(Vu.w,Vv.w):sat</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.h[i] = sat₁₆(Vv.w[i]); Vd.h[i+VBITS/32] = sat₁₆(Vu.w[i]); }</pre>
<code>Vd.h=vpacke(Vu.w,Vv.w)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.uh[i] = Vv.uw[i].uh[0]; Vd.uh[i+VBITS/32] = Vu.uw[i].uh[0] ; }</pre>
<code>Vd.h=vpacko(Vu.w,Vv.w)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.uh[i] = Vv.uw[i].uh[1]; Vd.uh[i+VBITS/32] = Vu.uw[i].uh[1] ; }</pre>
<code>Vd.ub=vpack(Vu.h,Vv.h):sat</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.ub[i] = usat₈(Vv.h[i]); Vd.ub[i+VBITS/16] = usat₈(Vu.h[i]); }</pre>
<code>Vd.uh=vpack(Vu.w,Vv.w):sat</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.uh[i] = usat₁₆(Vv.w[i]); Vd.uh[i+VBITS/32] = usat₁₆(Vu.w[i]); }</pre>

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction uses the HVX permute resource.

Intrinsics

<code>Vd.b=vpack(Vu.h,Vv.h):sat</code>	<code>HVX_Vector Q6_Vb_vpack_VhVh_sat(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.b=vpacke(Vu.h,Vv.h)</code>	<code>HVX_Vector Q6_Vb_vpacke_VhVh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.b=vpacko(Vu.h,Vv.h)</code>	<code>HVX_Vector Q6_Vb_vpacko_VhVh(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.h=vpack(Vu.w,Vv.w):sat</code>	<code>HVX_Vector Q6_Vh_vpack_VwVw_sat(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.h=vpacke(Vu.w,Vv.w)</code>	<code>HVX_Vector Q6_Vh_vpacke_VwVw(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.h=vpacko(Vu.w,Vv.w)</code>	<code>HVX_Vector Q6_Vh_vpacko_VwVw(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.ub=vpack(Vu.h,Vv.h):sat</code>	<code>HVX_Vector Q6_Vub_vpack_VhVh_sat(HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.uh=vpack(Vu.w,Vv.w):sat</code>	<code>HVX_Vector Q6_Vuh_vpack_VwVw_sat(HVX_Vector Vu, HVX_Vector Vv)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS															Parse		u5										d5					
0	0	0	1	1	1	1	1	1	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.b=vpacke(Vu.h,Vv.h)
0	0	0	1	1	1	1	1	1	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.h=vpacke(Vu.w,Vv.w)
0	0	0	1	1	1	1	1	1	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.ub=vpack(Vu.h,Vv.h):sat
0	0	0	1	1	1	1	1	1	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.b=vpack(Vu.h,Vv.h):sat
0	0	0	1	1	1	1	1	1	1	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.uh=vpack(Vu.w,Vv.w):sat
0	0	0	1	1	1	1	1	1	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.h=vpack(Vu.w,Vv.w):sat
0	0	0	1	1	1	1	1	1	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.b=vpacko(Vu.h,Vv.h)
0	0	0	1	1	1	1	1	1	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.h=vpacko(Vu.w,Vv.w)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Set predicate

Set a vector predicate register with a sequence of 1's based on the lower bits of the scalar register Rt.

Rt = 0x11 : Qd4 = 0-----00111111111111111b

Rt = 0x07 : Qd4 = 0-----0000000000001111111b

The operation is element-size agnostic, and typically is used to create a mask to predicate an operation if it does not span a whole vector register width.

Syntax	Behavior
Qd4=vsetq(Rt)	for(i = 0; i < VWIDTH; i++) QdV[i]=(i < (Rt & (VWIDTH-1))) ? 1 : 0;
Qd4=vsetq2(Rt)	for(i = 0; i < VWIDTH; i++) QdV[i]=(i <= ((Rt-1) & (VWIDTH-1))) ? 1 : 0;

Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction uses the HVX permute resource.

Intrinsics

Qd4=vsetq(Rt)

HVX_VectorPred Q6_Q_vsetq_R(Word32 Rt)

Qd4=vsetq2(Rt)

HVX_VectorPred Q6_Q_vsetq2_R(Word32 Rt)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											t5					Parse												d2				
0	0	0	1	1	0	0	1	1	0	1	t	t	t	t	t	P	P	0	-	-	-	-	-	0	1	0	-	0	1	d	d	Qd4=vsetq(Rt)
0	0	0	1	1	0	0	1	1	0	1	t	t	t	t	t	P	P	0	-	-	-	-	-	0	1	0	-	1	1	d	d	Qd4=vsetq2(Rt)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d2	Field to encode register d
t5	Field to encode register t

Vector in-lane lookup table

The `vlut` instructions are used to implement fast vectorized lookup-tables. The lookup table is contained in the `Vv` register while the indexes are held in `Vu`. Table elements can be either 8-bit or 16-bit. An aggregation feature is used to implement tables larger than 64 bytes in 64 byte mode and 128 bytes in 128 byte mode. This explanation discusses both the 64 byte and 128 byte modes of operation. In both 64 and 128 byte modes, the maximum amount of lookup table accessible is 32 bytes for byte lookups (`vlut32`) and 16 half words in hwords lookup (`vlut16`).

8-bit elements

For 64 byte mode, tables with 8-bit elements support 32 entry lookup tables using the `vlut32` instructions. The required entry is conditionally selected by using the lower five bits of the input byte for the respective output byte. A control input register, `Rt`, contains match and select bits. The lower three bits of `Rt` must match the upper three bits of the input byte index for the table entry to write to or OR'd with the destination vector register byte in `Vd` or `Vx` respectively. The LSB of `Rt` selects odd or even (32 entry) lookup tables in `Vv`.

A 256 byte table stored naturally in memory looks like the following:

```
127,126,.....66, 65, 64, 63, 62,.....2, 1, 0
255,254,....194,193,192,191,190,.....130,129,128
```

To prepare it for use with the `vlut` instruction in 64 byte mode, it must be shuffled in blocks of 32 bytes.

```
63, 31, 62, 30,.....36, 4, 35, 3, 34, 2, 33, 1, 32, 0 Rt=0, Rt=1
127, 95,126, 94,.....100, 68, 99, 67, 98, 66, 97, 65, 96, 64 Rt=2, Rt=3
```

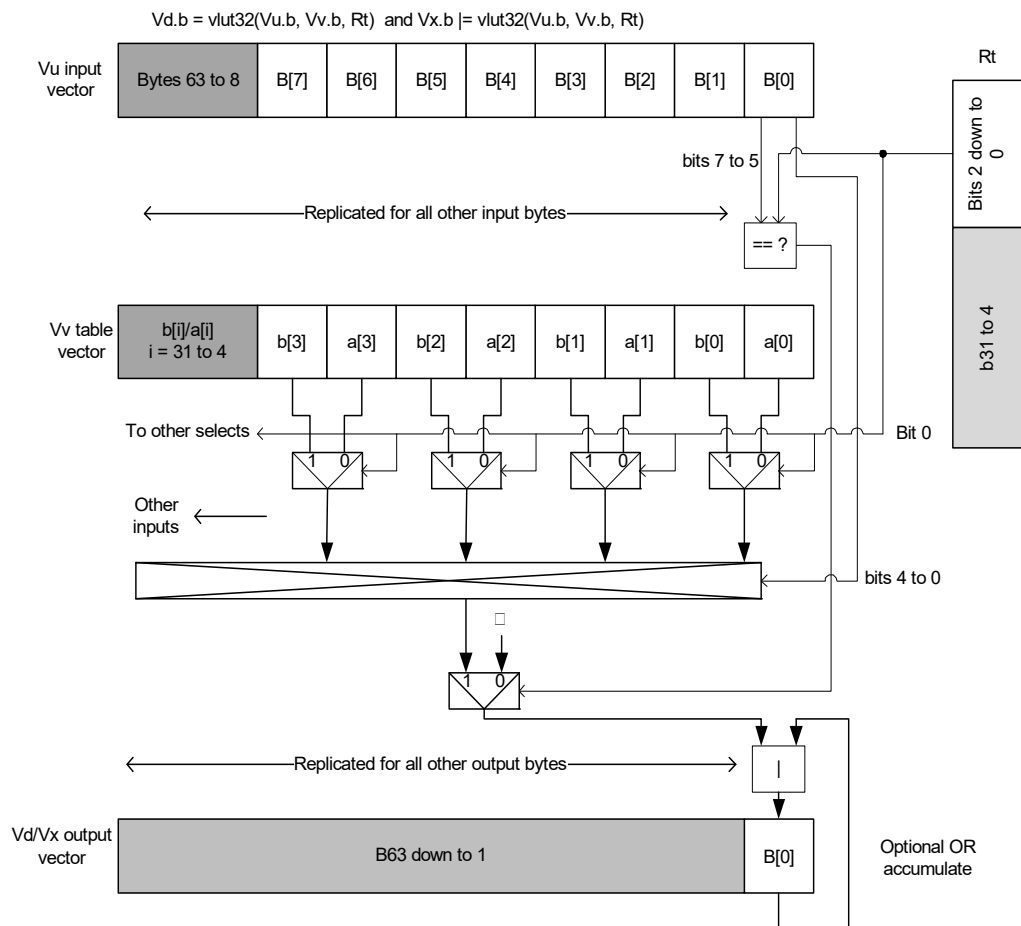
same ordering for bytes 128-255 Rt=4, 5, 6, 7

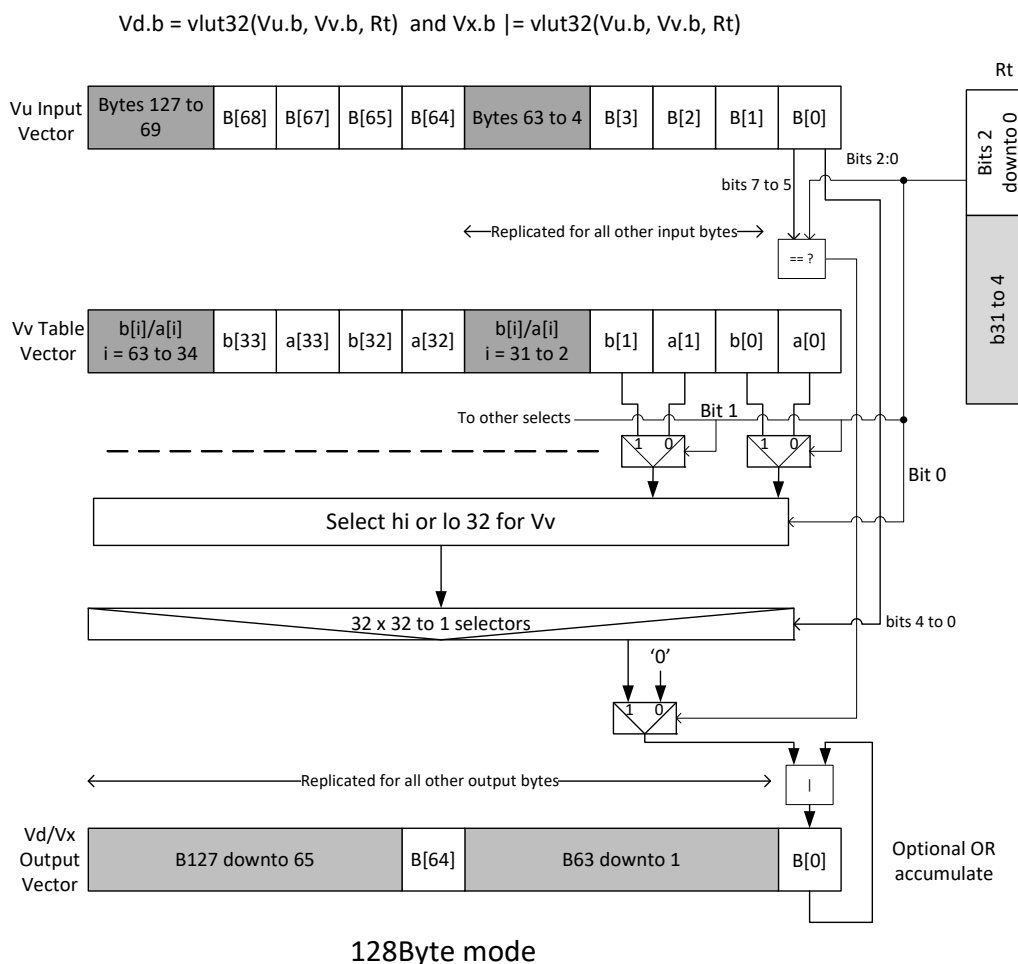
For 128 byte mode, the data must be shuffled in blocks of 64 bytes.

```
127, 63,126, 62,.....68, 4, 67, 3, 66, 2, 65, 1, 64, 0 Rt=0,1,2,3
```

same ordering for bytes 128 through 255 Rt=4,5,6,7

Data stored in this way that is accessed with 64 or 128 byte mode produces the same results. For 128 byte mode, bit 1 of `Rt` selects whether to use the odd or even packed table and bit 0 chooses the high of low 32 elements of that high or low table.





For tables with 16-bit elements, the basic unit is a 16-entry lookup table in 64 byte mode and 128 byte mode. Supported by the `vlut16` instructions. The even byte entries conditionally select using the lower four bits for the even destination register Vdd0, the odd byte entries select table entries into the odd vector destination register Vdd1. A control input register, Rt, contains match and select bits in the same way as the byte table case. For 64 byte mode, the lower four bits of Rt must match the upper four bits of the input bytes for the table entry to write to or OR'd with the destination vector register bytes in Vdd or Vxx respectively. Bit 0 of Rt selects the even or odd 16 entries in Vv. For 128 byte, only the upper four bits of input bytes must also match the lower four of Rt. Bit 1 of Rt selects odd or even hwords and bit 0 selects the lower or upper 16 entries in the Vv register.

For larger than 32-element tables in the hword case (for example 256 entries), the user must access the main lookup table in eight different 32 hword sections.

A 256H table stored naturally in memory looks like the following example:

```

63, 62, .....2, 1, 0
127, 126, .....66, 65, 64
191, 190, .....130, 129, 128
255, 254, .....194, 193, 192

```

To prepare it for use with the `vlut` instruction in 64 byte mode, it must be shuffled in blocks of 16 hwords, the LSB of `Rt` is used to choose the even or odd 16 entry hword tables in `Vv`.

31, 15, 30, 14, 20, 4, 19, 3, 18, 2, 17, 1, 16, 0 `Rt=0`, `Rt=1`
 63, 47, 62, 46, 52, 36, 51, 35, 50, 34, 49, 33, 48, 32 `Rt=2`, `Rt=3`

same ordering for bytes 64-255 `Rt=4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15`

For 128 byte mode, the data must be shuffled in blocks of 32 hwords. Bit 1 of `Rt` is used to choose between the even or odd 32 hwords in `Vv`. Bit 0 accesses the high or low 16 half words of the odd or even set.

63, 31, 62, 30, 36, 4, 35, 3, 34, 2, 33, 1, 32, `gh0` `Rt=0,1` `Rt=2,3`

same ordering for bytes 128-255 `Rt=4,5`, `Rt=6,7`, `Rt=8,9`, `Rt=10,11`, `Rt=12,13`, `Rt=14,15`

Figure 6-3 shows `vlut16` with even bytes being used to look up a table value, with the result written into the even destination register. Odd values go into the odd destination, 64 byte and 128 byte modes are shown.

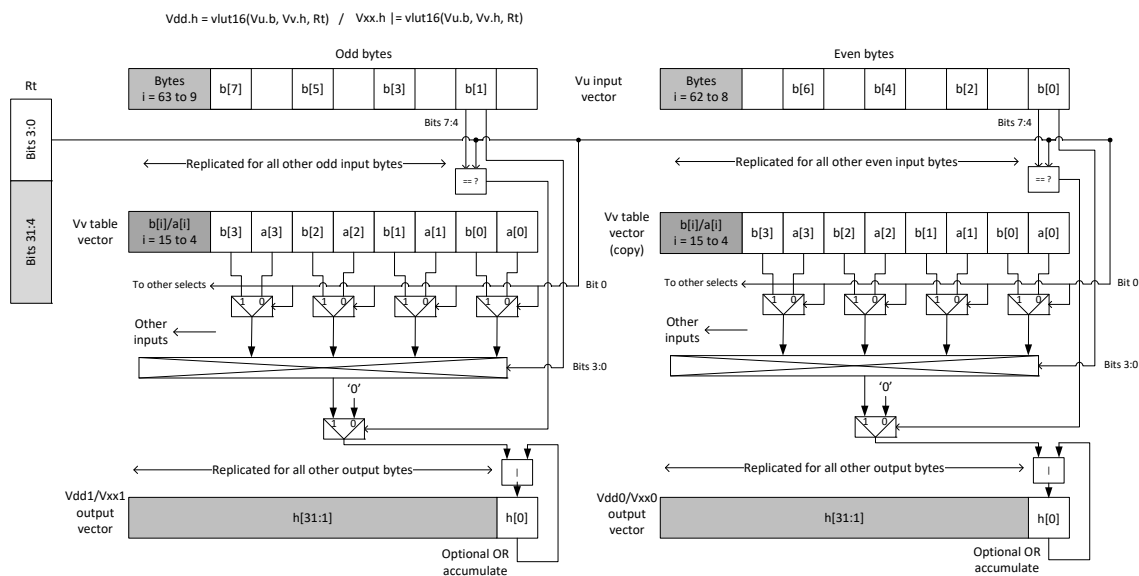
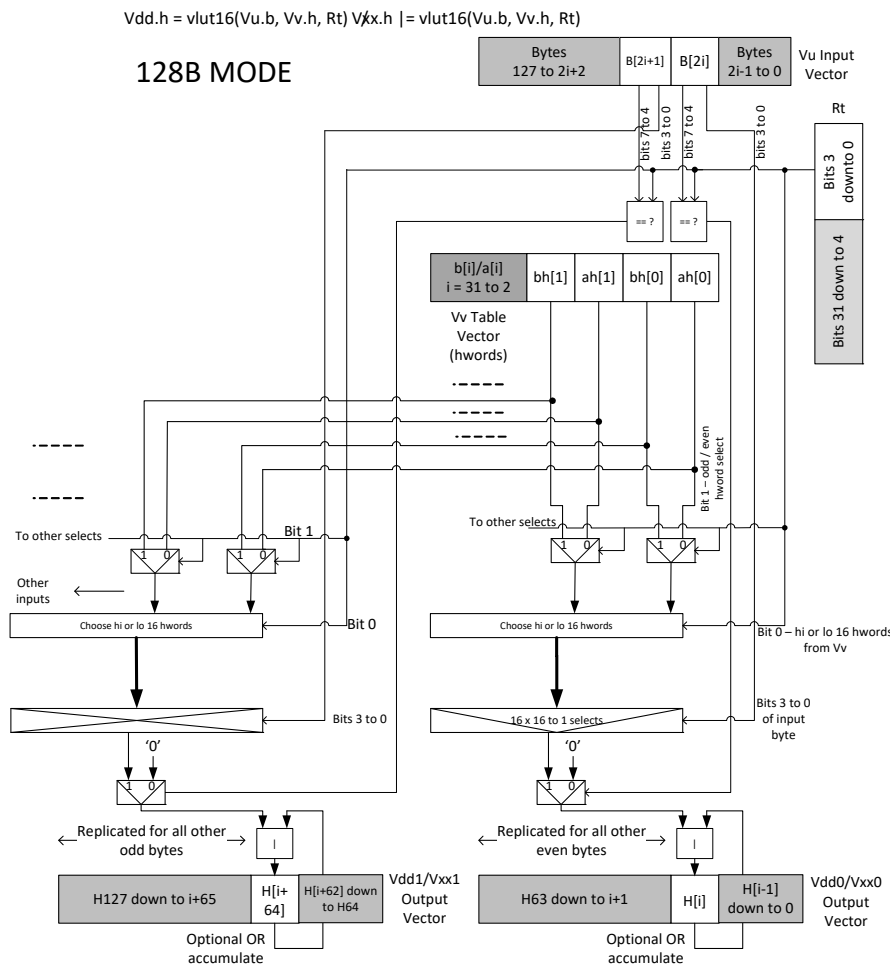


Figure 6-3 Vlut16 with even bytes used to look up a table value



`vluts` with the `nomatch` extension do not look at the upper bits and always produce a result. These are for small lookup tables.

Syntax

`Vd.b=vlut32(Vu.b,Vv.b,#u3)`

`Vd.b=vlut32(Vu.b,Vv.b,Rt)`

Behavior

```
for (i = 0; i < VELEM(8); i++) {
    matchval = #u & 0x7;
    oddhalf = (#u >> (log2(VECTOR_SIZE)-6)) & 0x1;
    idx = Vu.ub[i];
    Vd.b[i] = ((idx & 0xE0) == (matchval << 5)) ?
    Vv.h[idx % VBITS/16].b[oddhalf] : 0 ;
}
```

```
for (i = 0; i < VELEM(8); i++) {
    matchval = Rt & 0x7;
    oddhalf = (Rt >> (log2(VECTOR_SIZE)-6)) & 0x1;
    idx = Vu.ub[i];
    Vd.b[i] = ((idx & 0xE0) == (matchval << 5)) ?
    Vv.h[idx % VBITS/16].b[oddhalf] : 0 ;
}
```

Syntax

```
Vd.b=vlut32(Vu.b,Vv.b,Rt):no
match
```

Behavior

```
for (i = 0; i < VELEM(8); i++) {
    matchval = Rt & 0x7;
    oddhalf = (Rt >> (log2(VECTOR_SIZE)-6)) & 0x1;
    idx = Vu.ub[i];
    idx = (idx&0x1F) | (matchval<<5);
    Vd.b[i] = Vv.h[idx % VBITS/16].b[oddhalf] ;
}
```

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction uses the HVX permute resource.
- Input scalar register Rt is limited to registers 0 through 7

Intrinsics

```
Vd.b = vlut32(Vu.b,Vv.b,#u3) HVX_Vector Q6_Vb_vlut32_VbVbI(HVX_Vector Vu,
HVX_Vector Vv, Word32 Iu3)
```

```
Vd.b = vlut32(Vu.b,Vv.b,Rt) HVX_Vector Q6_Vb_vlut32_VbVbR(HVX_Vector Vu,
HVX_Vector Vv, Word32 Rt)
```

```
Vd.b = vlut32(Vu.b, Vv.b,
Rt):nomatch HVX_Vector Q6_Vb_vlut32_VbVbR_nomatch(HVX_Vector
Vu, HVX_Vector Vv, Word32 Rt)
```

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS													t3			Parse		u5								d5						
0	0	0	1	1	0	0	0	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.b=vlut32(Vu.b,Vv.b,Rt): omatch
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	1	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.b=vlut32(Vu.b,Vv.b,Rt)
ICLASS													Parse			u5								d5								
0	0	0	1	1	1	1	0	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	i	i	i	d	d	d	d	d	Vd.b=vlut32(Vu.b,Vv.b,#u3)

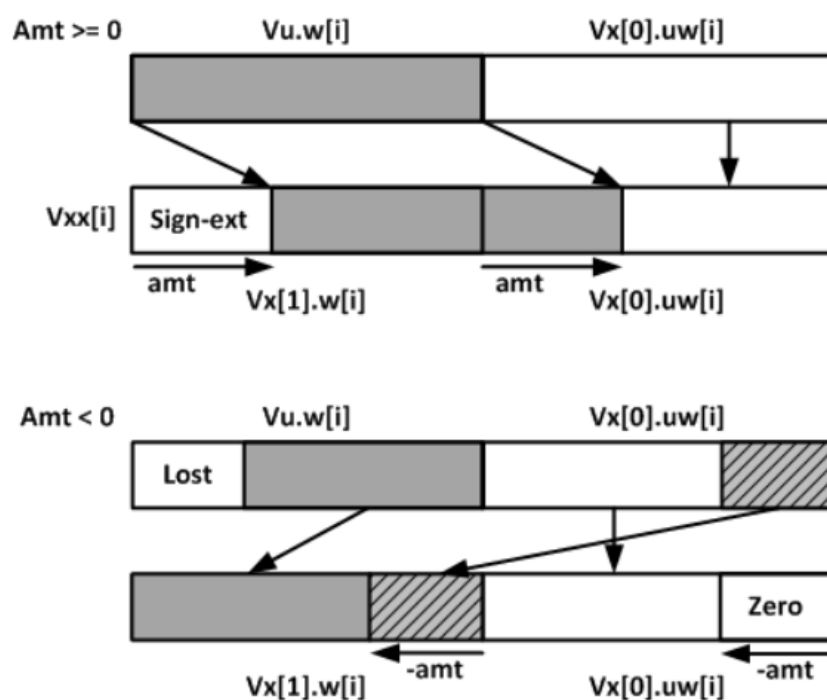
Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t3	Field to encode register t
u5	Field to encode register u
v5	Field to encode register v

6.11 HVX permute shift resource

The HVX permute shift resource instruction subclass includes instructions that use both the HVX permute and shift resources.

Vector ASR overlay

The primary use of this instruction is to complete a 64 bit bidirectional arithmetic shift right (ASR) by shifting the high-word source ($Vu.w[i]$) and merging with the destination register. This assumes a rotate on the low-word source was already performed and placed in the low-word of the destination register. However, this instruction can also be used to concatenate LSB portions of the source and destination registers and placed into the high or low word of the destination depending on the shift amount.



Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction uses the HVX permute resource.
- This instruction uses the HVX shift resource.

Syntax

```
Vxx.w=vasrinto(Vu.w,Vv.w)
```

Behavior

```
for (i = 0; i < VELEM(32); i++) {
    shift = (Vu.w[i] << 32);
    mask = (((Vxx.v[0].w[i]) << 32) |
Vxx.v[0].w[i]);
    lomask = ((1) << 32) - 1);
    count = -(0x40 & Vv.w[i]) + (Vv.w[i] & 0x3f);
    result = (count == -0x40) ? 0 : (((count < 0) ?
((shift << -(count)) | (mask & (lomask << -
(count)))) : ((shift >> count) | (mask & (lomask >>
count)))));
    Vxx.v[1].w[i] = ((result >> 32) & 0xffffffff);
    Vxx.v[0].w[i] = (result & 0xffffffff);
}
```

Intrinsics

```
Vxx.w=vasrinto(Vu.w,Vv.w) HVX_VectorPair Q6_Ww_vasrinto_WwVwVw(HVX_VectorPair
Vxx, HVX_Vector Vu, HVX_Vector Vv)
```

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse				u5						x5						
0	0	0	1	1	0	1	0	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	x	x	x	x	x	Vxx.w=vasrinto(Vu.w,Vv.w)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
u5	Field to encode register u
v5	Field to encode register v
x5	Field to encode register x

Vector shuffle and deal cross-lane

`vshuff` (formerly `vtrans2x2`) and `vdeal` perform a multiple-level transpose operation between groups of elements in two vectors. The element size is specified by the scalar register `Rt`. `Rt = 1` indicates an element size of one byte, `Rt = 2` indicates halfwords, `Rt = 4` indicates words, `Rt = 8` indicates eight bytes, `Rt=16` indicates 16 bytes, and `Rt=32` indicates 32 bytes. The data in the two registers should be considered as two rows of 64 bytes each. Each two-by-two group is transposed. For example, `Rt = 4` indicates that each element contains four bytes. The matrix of four of these elements is made up of two elements from the even register and two corresponding elements of the odd register. This two-by-two array is transposed, and the resulting elements are presented in the two destination registers. A value of `Rt = 0` leaves the input unchanged.

Examples for `Rt = 1, 2, 4, 8, 16`, and `32` are shown below. `vdeal` and `vshuff` perform the same operation. [Figure 6-4](#) is valid for `vshuff` and `vdeal`.

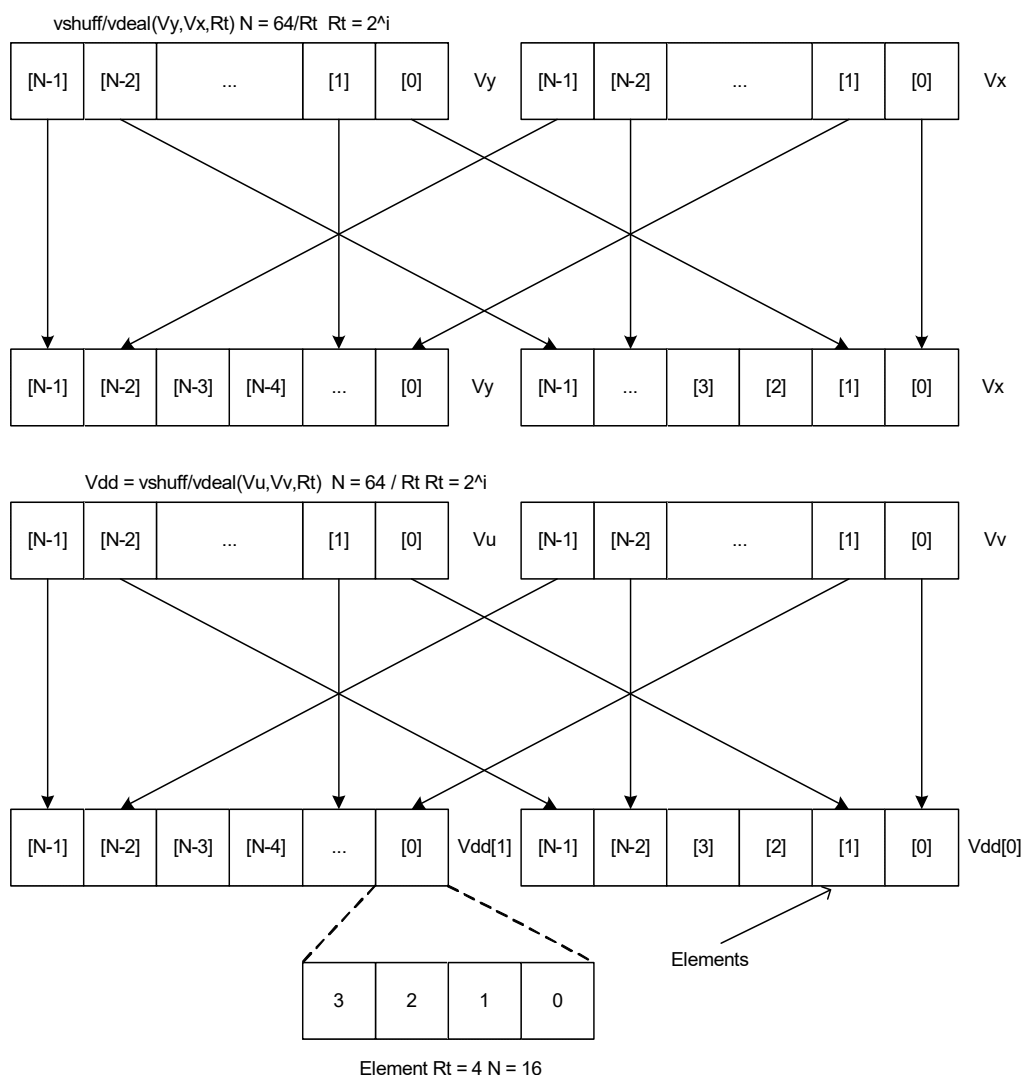


Figure 6-4 `vshuff/vdeal` (`Vy, Vx, Rt`) $N = 64/Rt \quad Rt = 2^i$

When a value of R_t other than 1, 2, 4, 8, 16, or 32 is used, the effect is a compound hierarchical transpose. For example, if the value 23 is used, $23 = 1+2+4+16$ indicates that the transformation is the same as performing the `vshuff` instruction with $R_t=1$, then $R_t=2$ on that result, then $R_t=4$ on its result, then $R_t=16$ on its result. The order is in increasing element size. For `vdeal`, the order is reversed, starting with the largest element size first, then working down to the smallest.

When the R_t value is the negated power of 2: -1, -2, -4, -8, -16, or -32, it performs a perfect shuffle for `vshuff`, or a deal for `vdeal` of the smallest element size. For example, if $R_t = -24$, this is a multiple of 8, so 8 is the smallest element size. With a -ve value of R_t , all the upper bits of the value R_t are set. For example, with $R_t=-8$ this is the same as $32+16+8$. Figure 6-5 shows the effect of this transform for both `vshuff` and `vdeal`.

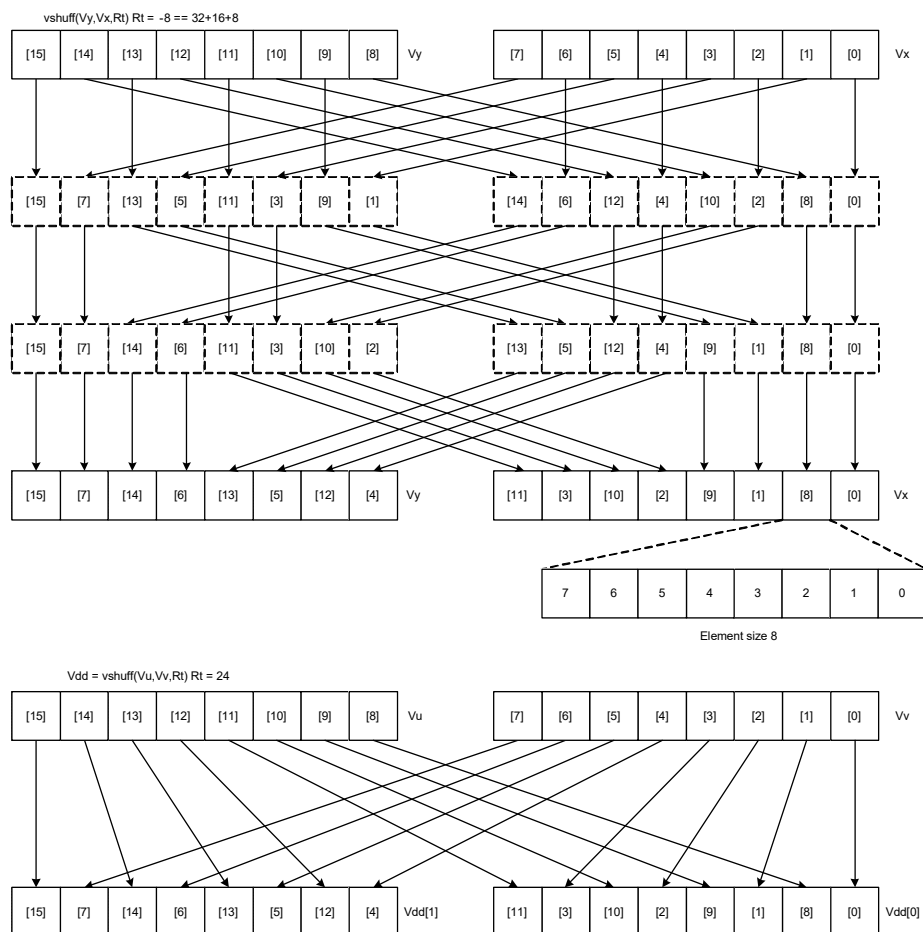
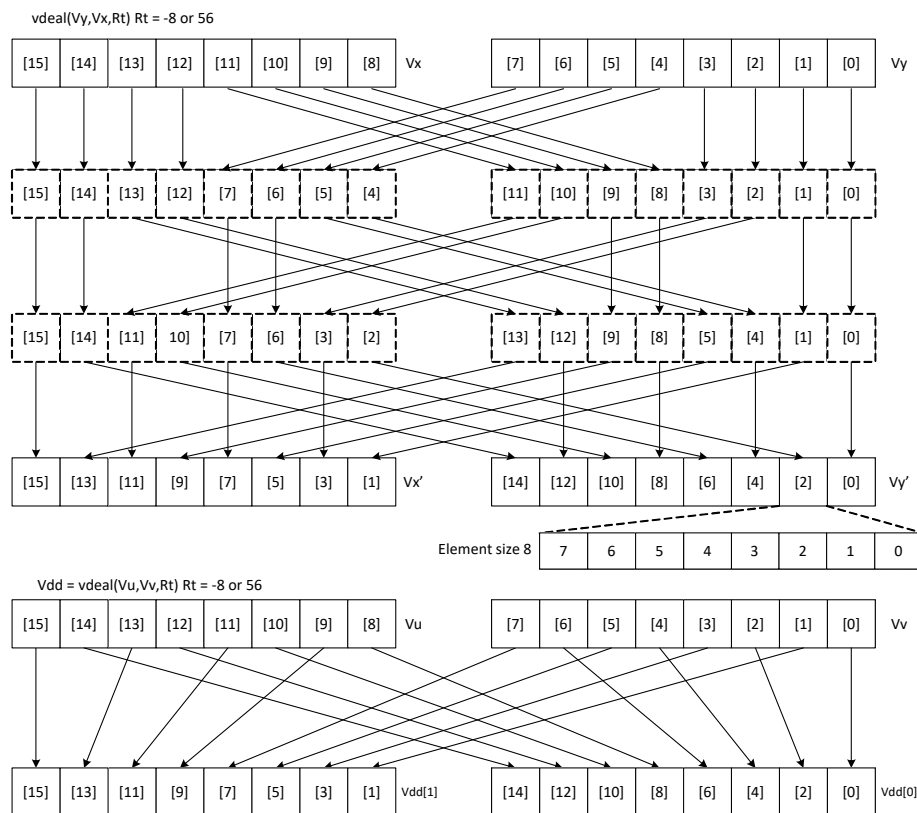


Figure 6-5 vshuff(Vy, Vx, Rt) Rt = -8 == 32 + 16 + 8

Figure 6-6 $vdeal(Vy, Vx, Rt) \text{ Rt} = -8 \text{ or } 56$ **Syntax**

```
Vdd=vdeal (Vu,Vv,Rt)
```

Behavior

```
Vdd.v[0] = Vv;
Vdd.v[1] = Vu;
for (offset=VWIDTH>>1; offset>0; offset>>=1) {
    if ( Rt & offset) {
        for (k = 0; k < VELEM(8); k++) {
            if (!( k & offset)) {
                SWAP(Vdd.v[1].ub[k],Vdd.v[0].ub[k+offset]);
            }
        }
    }
}
```

Syntax	Behavior
<code>Vdd=vshuff(Vu,Vv,Rt)</code>	<pre> Vdd.v[0] = Vv; Vdd.v[1] = Vu; for (offset=1; offset<VWIDTH; offset<=&=1) { if (Rt & offset) { for (k = 0; k < VELEM(8); k++) { if (!(k & offset)) { SWAP(Vdd.v[1].ub[k],Vdd.v[0].ub[k+offset]); } } } } </pre>
<code>vdeal(Vy,Vx,Rt)</code>	<pre> for (offset=VWIDTH>>1; offset>0; offset>>=1) { if (Rt & offset) { for (k = 0; k < VELEM(8); k++) { if (!(k & offset)) { SWAP(Vy.ub[k],Vx.ub[k+offset]); } } } } </pre>
<code>vshuff(Vy,Vx,Rt)</code>	<pre> for (offset=1; offset<VWIDTH; offset<=&=1) { if (Rt & offset) { for (k = 0; k < VELEM(8); k++) { if (!(k & offset)) { SWAP(Vy.ub[k],Vx.ub[k+offset]); } } } } </pre>
<code>vtrans2x2(Vy,Vx,Rt)</code>	Assembler mapped to: <code>"vshuff(Vy,Vx,Rt)"</code>

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction uses the HVX permute resource.
- Input scalar register Rt is limited to registers 0 through 7
- This instruction uses the HVX shift resource.

Intrinsics

<code>Vdd=vdeal(Vu,Vv,Rt)</code>	<code>HVX_VectorPair Q6_W_vdeal_VVR(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)</code>
<code>Vdd=vshuff(Vu,Vv,Rt)</code>	<code>HVX_VectorPair Q6_W_vshuff_VVR(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
ICLASS											t5					Parse			y5								x5						
0	0	0	1	1	0	0	1	1	1	1	t	t	t	t	t	P	P	1	y	y	y	y	y	0	0	1	x	x	x	x	x	vshuff(Vy,Vx,Rt)	
0	0	0	1	1	0	0	1	1	1	1	t	t	t	t	t	P	P	1	y	y	y	y	y	0	1	0	x	x	x	x	x	vdeal(Vy,Vx,Rt)	
ICLASS											t3					Parse			u5								d5						
0	0	0	1	1	0	1	1	v	v	v	v	v	v	t	t	t	P	P	1	u	u	u	u	u	0	1	1	d	d	d	d	d	Vdd=vshuff(Vu,Vv,Rt)
0	0	0	1	1	0	1	1	v	v	v	v	v	v	t	t	t	P	P	1	u	u	u	u	u	1	0	0	d	d	d	d	d	Vdd=vdeal(Vu,Vv,Rt)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t3	Field to encode register t
t5	Field to encode register t
u5	Field to encode register u
v5	Field to encode register v
x5	Field to encode register x
y5	Field to encode register y

Vector in-lane lookup table

The `vlut` instructions are used to implement fast vectorized lookup-tables. The lookup table is contained in the `Vv` register while the indexes are held in `Vu`. Table elements can be either 8-bit or 16-bit. An aggregation feature is used to implement tables larger than 64 bytes in 64 byte mode and 128 bytes in 128 byte mode. This explanation discusses both the 64 byte and 128 byte modes of operation. In both 64 and 128 byte modes, the maximum amount of lookup table accessible is 32 bytes for byte lookups (`vlut32`) and 16 half words in hwords lookup (`vlut16`).

8-bit elements

For 64 byte mode, tables with 8-bit elements support 32 entry lookup tables using the `vlut32` instructions. The required entry is conditionally selected by using the lower five bits of the input byte for the respective output byte. A control input register, `Rt`, contains match and select bits. The lower three bits of `Rt` must match the upper three bits of the input byte index for the table entry to write to or OR'd with the destination vector register byte in `Vd` or `Vx` respectively. The LSB of `Rt` selects odd or even (32 entry) lookup tables in `Vv`.

A 256 byte table stored naturally in memory looks like the following:

```
127,126,.....66, 65, 64, 63, 62,.....2, 1, 0
255,254,....194,193,192,191,190,.....130,129,128
```

To prepare it for use with the `vlut` instruction in 64 byte mode, it must be shuffled in blocks of 32 bytes.

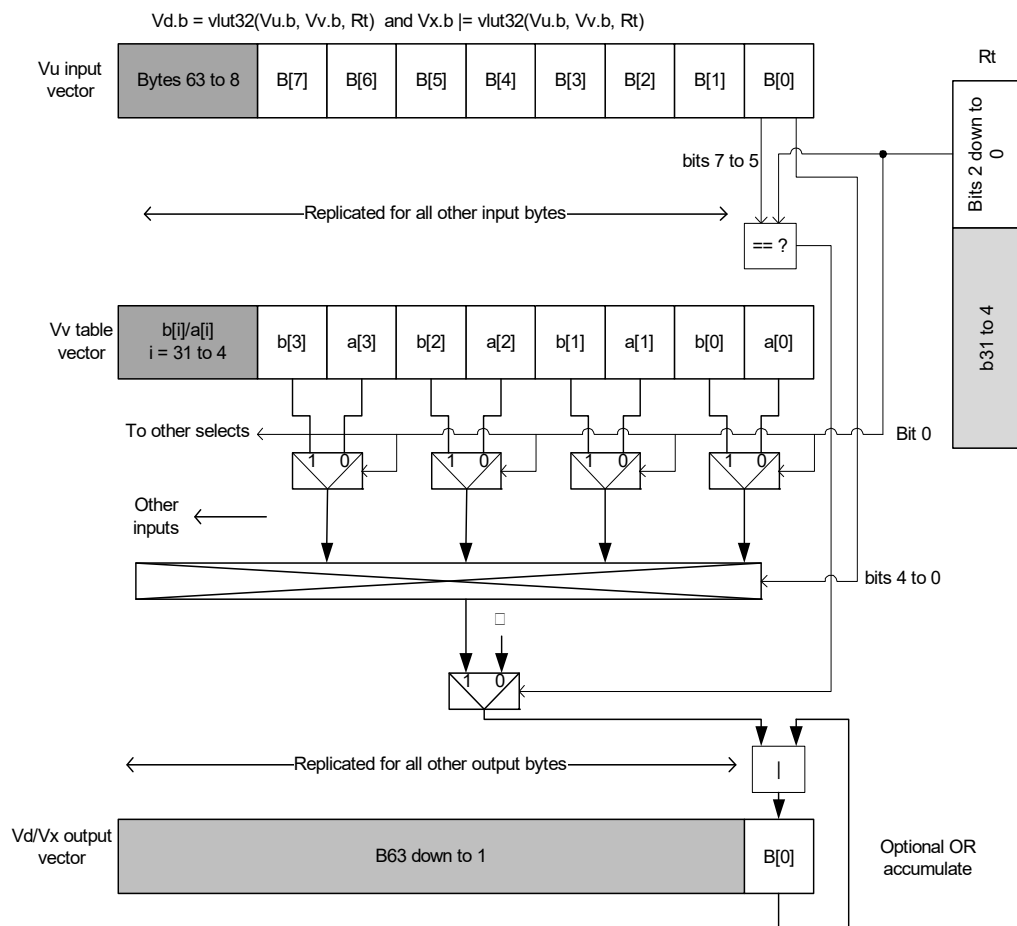
```
63, 31, 62, 30,.....36, 4, 35, 3, 34, 2, 33, 1, 32, 0 Rt=0, Rt=1
127, 95,126, 94,.....100, 68, 99, 67, 98, 66, 97, 65, 96, 64 Rt=2, Rt=3
same ordering for bytes 128-255 Rt=4, 5, 6, 7
```

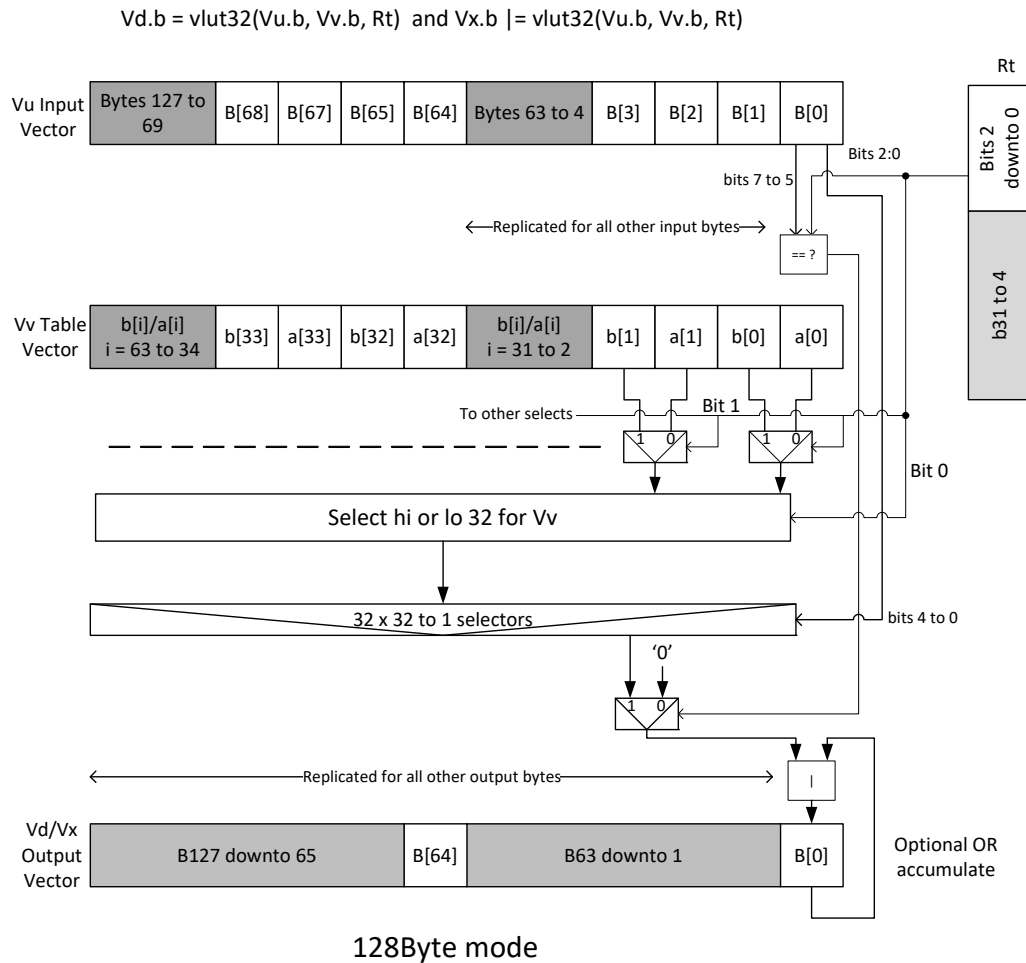
For 128 byte mode, the data must be shuffled in blocks of 64 bytes.

```
127, 63,126, 62,.....68, 4, 67, 3, 66, 2, 65, 1, 64, 0 Rt = 0, 1, 2, 3
same ordering for bytes 128 through 255 Rt = 4, 5, 6, 7
```

Access to data is stored in this way with 64 or 128 byte mode produces the same results.

For 128 byte mode, bit 1 of Rt selects whether to use the odd or even packed table and bit 0 chooses the high of low 32 elements of that high or low table.





For tables with 16-bit elements, the basic unit is a 16-entry lookup table in 64 byte mode and 128 byte mode. Supported by the `vlut16` instructions. The even byte entries conditionally select using the lower four bits for the even destination register `Vdd0`, the odd byte entries select table entries into the odd vector destination register `Vdd1`. A control input register, `Rt`, contains match and select bits in the same way as the byte table case. For 64 byte mode, the lower four bits of `Rt` must match the upper four bits of the input bytes for the table entry to write to or OR'd with the destination vector register bytes in `Vdd` or `Vxx`, respectively. Bit 0 of `Rt` selects the even or odd 16 entries in `Vv`. For 128 byte, only the upper four bits of input bytes must also match the lower four of `Rt`. Bit 1 of `Rt` selects odd or even hwords and bit 0 selects the lower or upper 16 entries in the `Vv` register.

For larger than 32-element tables in the hword case (for example 256 entries), the user must access the main lookup table in eight different 32 hword sections.

A 256H table stored naturally in memory looks like the following:

```

63, 62, .....2, 1, 0
127,126, .....66, 65, 64
191,190, .....130,129,128
255,254, .....194,193,192

```

To prepare it for use with the `vlut` instruction in 64 byte mode, it must be shuffled in blocks of 16 hwords. The LSB of `Rt` is used to choose the even or odd 16 entry hword tables in `Vv`.

31, 15, 30, 14, 20, 4, 19, 3, 18, 2, 17, 1, 16, 0 `Rt=0`, `Rt=1`
 63, 47, 62, 46, 52, 36, 51, 35, 50, 34, 49, 33, 48, 32 `Rt=2`, `Rt=3`

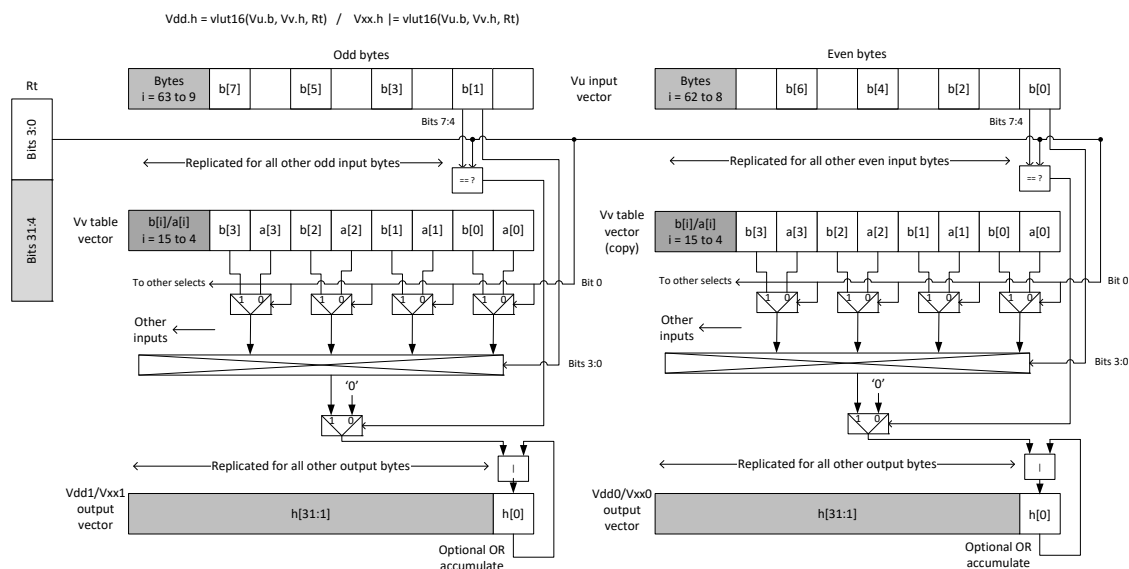
same ordering for bytes 64 through 255 `Rt=4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15`

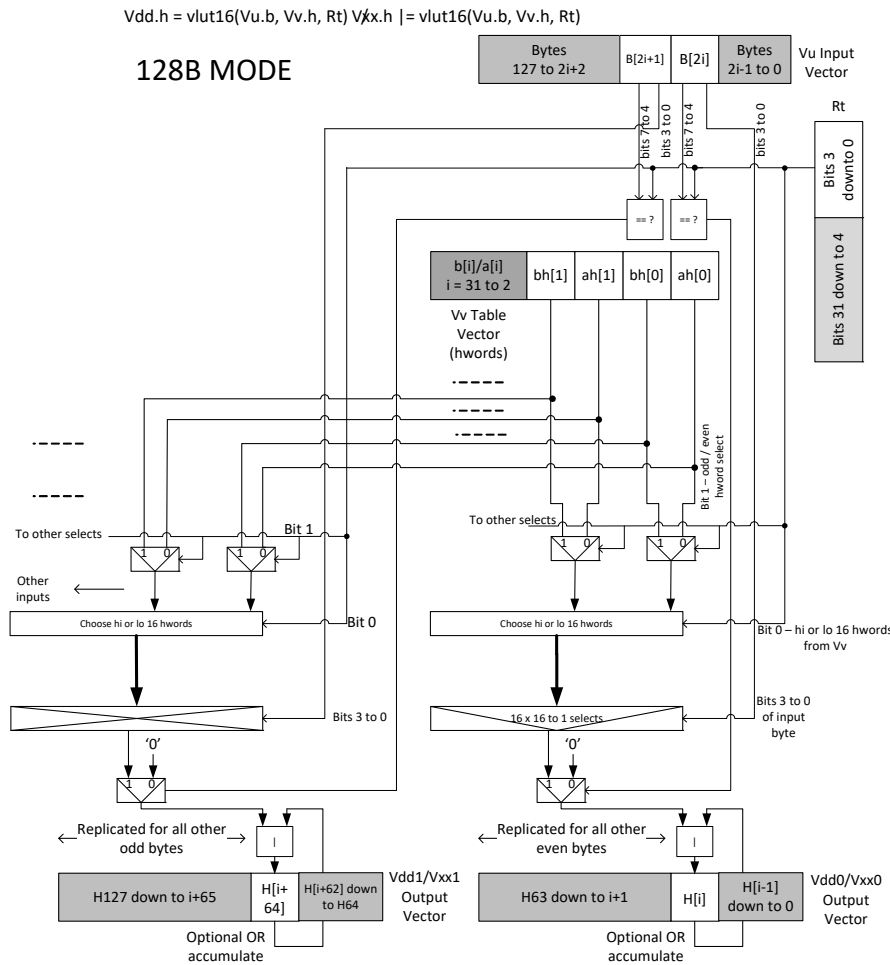
For 128 byte mode, the data must be shuffled in blocks of 32 hwords. Bit 1 of `Rt` is used to choose between the even or odd 32 hwords in `Vv`. Bit 0 accesses the high or lo 16 half words of the odd or even set.

63, 31, 62, 30, 36, 4, 35, 3, 34, 2, 33, 1, 32, 0 `Rt=0, 1` `Rt=2, 3`

same ordering for bytes 128 through 255 `Rt=4, 5`, `Rt=6, 7`, `Rt=8, 9`, `Rt=10, 11`, `Rt=12, 13`,
`Rt=14, 15`

The following diagram shows `vlut16` with even bytes being used to look up a table value, with the result written into the even destination register. Odd values going into the odd destination, 64 byte and 128 byte modes are shown.





`vluts` with the `nomatch` extension do not look at the upper bits and always produce a result. These are for small lookup tables.

Syntax

```
Vdd.h=vlut16(Vu.b,Vv.h,#u3)
```

Behavior

```
for (i = 0; i < VELEM(16); i++) {
    matchval = #u & 0xF;
    oddhalf = (#u >> (log2(VECTOR_SIZE)-6))
    & 0x1;
    idx = Vu.uh[i].ub[0];
    Vdd.v[0].h[i] = ((idx & 0xF0) ==
(matchval << 4)) ? Vv.w[idx %
VBITS/32].h[oddhalf] : 0;
    idx = Vu.uh[i].ub[1];
    Vdd.v[1].h[i] = ((idx & 0xF0) ==
(matchval << 4)) ? Vv.w[idx %
VBITS/32].h[oddhalf] : 0;
}
```

Syntax	Behavior
<code>Vdd.h=vlut16(Vu.b,Vv.h,Rt)</code>	<pre> for (i = 0; i < VELEM(16); i++) { matchval = Rt & 0xF; oddhalf = (Rt >> (log2(VECTOR_SIZE)-6)) & 0x1; idx = Vu.uh[i].ub[0]; Vdd.v[0].h[i] = ((idx & 0xF0) == (matchval << 4)) ? Vv.w[idx % VBITS/32].h[oddhalf] : 0; idx = Vu.uh[i].ub[1]; Vdd.v[1].h[i] = ((idx & 0xF0) == (matchval << 4)) ? Vv.w[idx % VBITS/32].h[oddhalf] : 0; } </pre>
<code>Vdd.h=vlut16(Vu.b,Vv.h,Rt):nomatch</code>	<pre> for (i = 0; i < VELEM(16); i++) { matchval = Rt & 0xF; oddhalf = (Rt >> (log2(VECTOR_SIZE)-6)) & 0x1; idx = Vu.uh[i].ub[0]; idx = (idx&0x0F) (matchval<<4); Vdd.v[0].h[i] = Vv.w[idx % VBITS/32].h[oddhalf]; idx = Vu.uh[i].ub[1]; idx = (idx&0x0F) (matchval<<4); Vdd.v[1].h[i] = Vv.w[idx % VBITS/32].h[oddhalf]; } </pre>
<code>Vx.b =vlut32(Vu.b,Vv.b,#u3)</code>	<pre> for (i = 0; i < VELEM(8); i++) { matchval = #u & 0x7; oddhalf = (#u >> (log2(VECTOR_SIZE)-6)) & 0x1; idx = Vu.ub[i]; Vx.b[i] = ((idx & 0xE0) == (matchval << 5)) ? Vv.h[idx % VBITS/16].b[oddhalf] : 0; } </pre>
<code>Vx.b =vlut32(Vu.b,Vv.b,Rt)</code>	<pre> for (i = 0; i < VELEM(8); i++) { matchval = Rt & 0x7; oddhalf = (Rt >> (log2(VECTOR_SIZE)-6)) & 0x1; idx = Vu.ub[i]; Vx.b[i] = ((idx & 0xE0) == (matchval << 5)) ? Vv.h[idx % VBITS/16].b[oddhalf] : 0; } </pre>

Syntax	Behavior
<code>Vxx.h =vlut16(Vu.b,Vv.h,#u3)</code>	<pre> for (i = 0; i < VELEM(16); i++) { matchval = #u & 0xF; oddhalf = (#u >> (log2(VECTOR_SIZE)-6)) & 0x1; idx = Vu.uh[i].ub[0]; Vxx.v[0].h[i] = ((idx & 0xF0) == (matchval << 4)) ? Vv.w[idx % VBITS/32].h[oddhalf] : 0; idx = Vu.uh[i].ub[1]; Vxx.v[1].h[i] = ((idx & 0xF0) == (matchval << 4)) ? Vv.w[idx % VBITS/32].h[oddhalf] : 0; } </pre>
<code>Vxx.h =vlut16(Vu.b,Vv.h,Rt)</code>	<pre> for (i = 0; i < VELEM(16); i++) { matchval = Rt.ub[0] & 0xF; oddhalf = (Rt >> (log2(VECTOR_SIZE)-6)) & 0x1; idx = Vu.uh[i].ub[0]; Vxx.v[0].h[i] = ((idx & 0xF0) == (matchval << 4)) ? Vv.w[idx % VBITS/32].h[oddhalf] : 0; idx = Vu.uh[i].ub[1]; Vxx.v[1].h[i] = ((idx & 0xF0) == (matchval << 4)) ? Vv.w[idx % VBITS/32].h[oddhalf] : 0; } </pre>

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction uses the HVX permute resource.
- Input scalar register Rt is limited to registers 0 through 7
- This instruction uses the HVX shift resource.

Intrinsics

<code>Vdd.h=vlut16(Vu.b,Vv.h,#u3)</code>	<code>HVX_VectorPair Q6_Wh_vlut16_VbVhI(HVX_Vector Vu, HVX_Vector Vv, Word32 Iu3)</code>
<code>Vdd.h=vlut16(Vu.b,Vv.h,Rt)</code>	<code>HVX_VectorPair Q6_Wh_vlut16_VbVhR(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)</code>
<code>Vdd.h=vlut16(Vu.b,Vv.h,Rt) : nomatch</code>	<code>HVX_VectorPair Q6_Wh_vlut16_VbVhR_nomatch(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)</code>
<code>Vx.b =vlut32(Vu.b,Vv.b,#u3)</code>	<code>HVX_Vector Q6_Vb_vlut32or_VbVbVbI(HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv, Word32 Iu3)</code>
<code>Vx.b =vlut32(Vu.b,Vv.b,Rt)</code>	<code>HVX_Vector Q6_Vb_vlut32or_VbVbVbR(HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)</code>

```
Vxx.h|=vlut16(Vu.b,Vv.h,#u3) HVX_VectorPair
Q6_Wh_vlut16or_WhVbVhI(HVX_VectorPair Vxx,
HVX_Vector Vu, HVX_Vector Vv, Word32 Iu3)

Vxx.h|=vlut16(Vu.b,Vv.h,Rt) HVX_VectorPair
Q6_Wh_vlut16or_WhVbVhR(HVX_VectorPair Vxx,
HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
```

Encoding

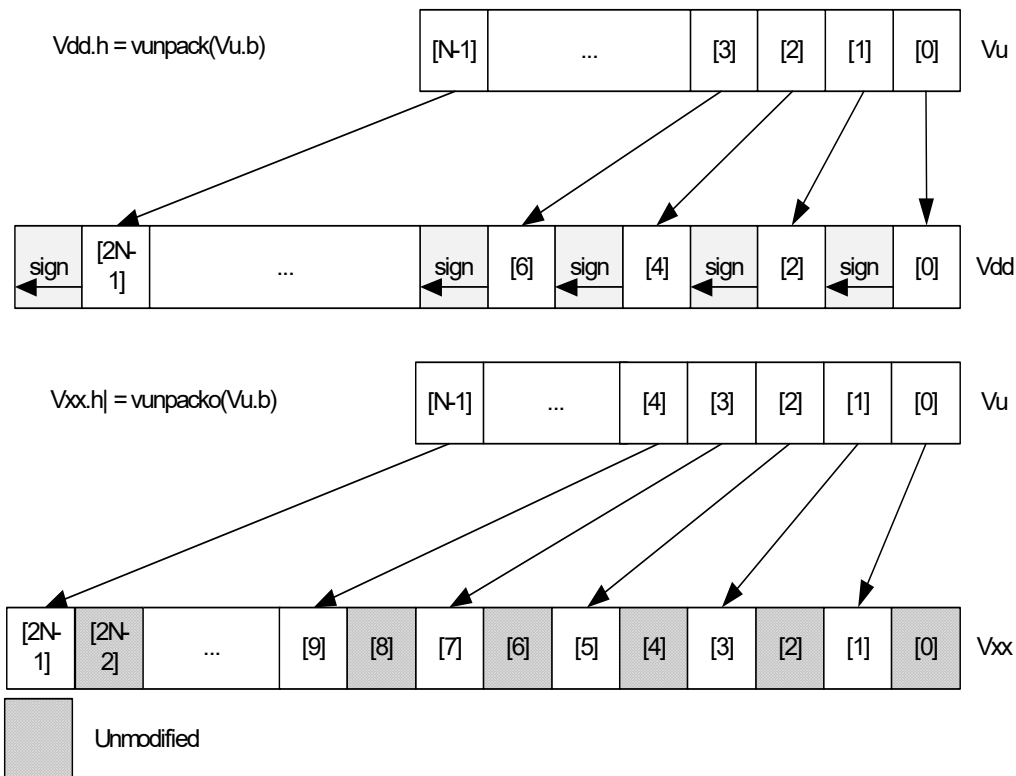
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																t3		Parse		u5										d5		
0	0	0	1	1	0	0	0	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vdd.h=vlut16(Vu.b,Vv.h,Rt):nomatch
ICLASS																t3		Parse		u5										x5		
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	1	u	u	u	u	u	1	0	1	x	x	x	x	x	Vx.b =vlut32(Vu.b,Vv.b,Rt)
ICLASS																t3		Parse		u5										d5		
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	1	u	u	u	u	u	1	1	0	d	d	d	d	d	Vdd.h=vlut16(Vu.b,Vv.h,Rt)
ICLASS																t3		Parse		u5										x5		
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	1	u	u	u	u	u	1	1	1	x	x	x	x	x	Vxx.h =vlut16(Vu.b,Vv.h,Rt)
ICLASS																		Parse		u5										x5		
0	0	0	1	1	1	0	0	1	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	i	i	i	x	x	x	x	x	Vx.b =vlut32(Vu.b,Vv.b,#u3)
0	0	0	1	1	1	0	0	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	i	i	i	x	x	x	x	x	Vxx.h =vlut16(Vu.b,Vv.h,#u3)
ICLASS																		Parse		u5										d5		
0	0	0	1	1	1	1	0	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	i	i	i	d	d	d	d	d	Vdd.h=vlut16(Vu.b,Vv.h,#u3)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t3	Field to encode register t
u5	Field to encode register u
v5	Field to encode register v
x5	Field to encode register x

Unpack

The unpack operation has two forms. The first form takes each element in vector register Vu and either zero or sign extends it to the next largest element size. The results are written into the vector register Vdd . This operation supports the unpacking of signed or unsigned byte to halfword, signed or unsigned halfword to word, and unsigned word to unsigned double.

The second form inserts elements from Vu into the odd element locations of Vxx . The even elements of Vxx are not changed. This operation supports the unpacking of signed or unsigned byte to halfword, and signed or unsigned halfword to word.



Syntax

$Vdd.h = \text{vunpack}(Vu.b)$

$Vdd.uh = \text{vunpack}(Vu.ub)$

$Vdd.uw = \text{vunpack}(Vu.uh)$

$Vdd.w = \text{vunpack}(Vu.h)$

Behavior

```
for (i = 0; i < VELEM(8); i++) {
    Vdd.h[i] = Vu.b[i] ;
}
```

```
for (i = 0; i < VELEM(8); i++) {
    Vdd.uh[i] = Vu.ub[i] ;
}
```

```
for (i = 0; i < VELEM(16); i++) {
    Vdd.uw[i] = Vu.uh[i] ;
}
```

```
for (i = 0; i < VELEM(16); i++) {
    Vdd.w[i] = Vu.h[i] ;
}
```

Syntax	Behavior
Vxx.h =vunpacko(Vu.b)	for (i = 0; i < VELEM(8); i++) { Vxx.uh[i] = Vu.ub[i]<<8 ; }
Vxx.w =vunpacko(Vu.h)	for (i = 0; i < VELEM(16); i++) { Vxx.uw[i] = Vu.uh[i]<<16 ; }

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction uses the HVX permute resource.
- This instruction uses the HVX shift resource.

Intrinsics

Vdd.h=vunpack(Vu.b)	HVX_VectorPair Q6_Wh_vunpack_Vb(HVX_Vector Vu)
Vdd.uh=vunpack(Vu.ub)	HVX_VectorPair Q6_Wuh_vunpack_Vub(HVX_Vector Vu)
Vdd.uw=vunpack(Vu.uh)	HVX_VectorPair Q6_Wuw_vunpack_Vuh(HVX_Vector Vu)
Vdd.w=vunpack(Vu.h)	HVX_VectorPair Q6_Ww_vunpack_Vh(HVX_Vector Vu)
Vxx.h =vunpacko(Vu.b)	HVX_VectorPair Q6_Wh_vunpackoor_WhVb(HVX_VectorPair Vxx, HVX_Vector Vu)
Vxx.w =vunpacko(Vu.h)	HVX_VectorPair Q6_Ww_vunpackoor_WwVh(HVX_VectorPair Vxx, HVX_Vector Vu)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse		u5										d5				
0	0	0	1	1	1	1	0	-	-	0	-	-	-	0	1	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vdd.uh=vunpack(Vu.ub)
0	0	0	1	1	1	1	0	-	-	0	-	-	-	0	1	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vdd.uw=vunpack(Vu.uh)
0	0	0	1	1	1	1	0	-	-	0	-	-	-	0	1	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vdd.h=vunpack(Vu.b)
0	0	0	1	1	1	1	0	-	-	0	-	-	-	0	1	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vdd.w=vunpack(Vu.h)
ICLASS																Parse		u5										x5				
0	0	0	1	1	1	1	0	-	-	0	-	-	0	0	0	P	P	1	u	u	u	u	u	0	0	0	x	x	x	x	x	Vxx.h =vunpacko(Vu.b)
0	0	0	1	1	1	1	0	-	-	0	-	-	0	0	0	P	P	1	u	u	u	u	u	0	0	1	x	x	x	x	x	Vxx.w =vunpacko(Vu.h)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
x5	Field to encode register x

6.12 HVX scatter double resource

The HVX scatter double resource instruction subclass includes instructions that perform scatter operations to the vector TCM.

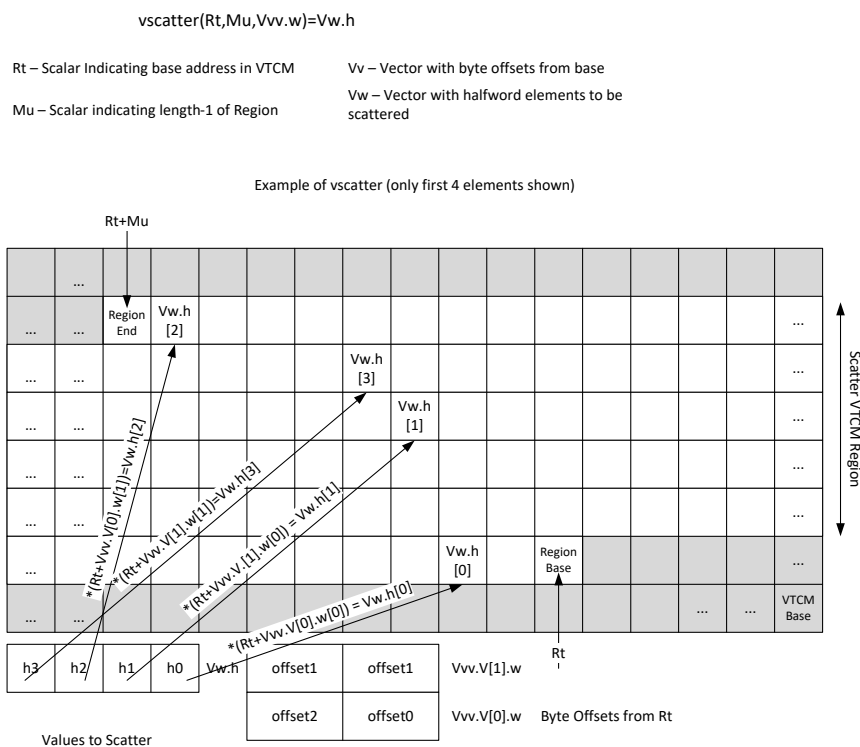
Vector scatter

Vector scatter instructions perform scatter operations to the vector TCM. Scatter operations copy values from the register file to a region in VTCM. This region of memory is specified by two scalar registers: Rt32 is the base and Mu2 specified the length-1 of the region in bytes. This region must reside in VTCM and cannot cross a page boundary. A vector register, Vvv32, specifies byte offsets in this region. Elements of either halfword or word granularity, specified by Vw32, are sent to addresses pointed to by $Rt + Vvv32$ for each element. In the memory, the element is either write to memory or accumulated with the memory (scatter-accumulate).

If multiple values are written to the same memory location, ordering is not guaranteed. This applies to a single scatter or multiple scatters.

The offset vector, Vvv32, can contain byte offsets specified in word sizes. The vector pair contains even element offsets in the lower vector and the odd in the upper vector. The final element addresses do not have to be byte aligned for regular scatter operations. However, for scatter accumulate instructions, the addresses are aligned. If an offset crosses the end of the scatter region end, it is dropped. Offsets must be positive, otherwise they are dropped.

All vectors registers can be used immediately after the scatter operation.



Syntax	Behavior
<pre>if (Qs4) vscatter(Rt,Mu,Vvv.w).h=Vw32</pre>	<pre>MuV = MuV (element_size-1); Rt = Rt & ~(element_size-1); for (i = 0; i < VELEM(32); i++) { for(j = 0; j < 2; j++) { EA = Rt+Vvv.v[j].uw[i]; if ((Rt <= EA <= Rt + MuV) & QsV) *EA = VwV.w[i].uh[j]; } }</pre>
<pre>if (Qs4) vscatter(Rt,Mu,Vvv.w)=Vw32.h</pre>	Assembler mapped to: "if (Qs4) vscatter(Rt,Mu2,Vvv.w).h=Vw32"
<pre>vscatter(Rt,Mu,Vvv.w)+=Vw32.h</pre>	Assembler mapped to: "vscatter(Rt,Mu2,Vvv.w).h+=Vw32"
<pre>vscatter(Rt,Mu,Vvv.w).h+=Vw32</pre>	<pre>MuV = MuV (element_size-1); Rt = Rt & ~(element_size-1); for (i = 0; i < VELEM(32); i++) { for(j = 0; j < 2; j++) { EA = Rt + Vvv.v[j].uw[i] = Vvv.v[j].uw[i] & ~(ALIGNMENT-1); if (Rt <= EA <= Rt + MuV) *EA += VwV.w[i].uh[j]; } }</pre>
<pre>vscatter(Rt,Mu,Vvv.w).h=Vw32</pre>	<pre>MuV = MuV (element_size-1); Rt = Rt & ~(element_size-1); for (i = 0; i < VELEM(32); i++) { for(j = 0; j < 2; j++) { EA = Rt+Vvv.v[j].uw[i]; if (Rt <= EA <= Rt + MuV) *EA = VwV.w[i].uh[j]; } }</pre>
<pre>vscatter(Rt,Mu,Vvv.w)=Vw32.h</pre>	Assembler mapped to: "vscatter(Rt,Mu2,Vvv.w).h=Vw32"

Class: COPROC_VMEM (slots 0)**Notes**

- This instruction uses any pair of the HVX resources (both multiply or shift/permute).

Intrinsics

<pre>if (Qs4) vscatter(Rt,Mu,Vvv.w).h=Vw32</pre>	<pre>void Q6_vscatter_QRMWwV(HVX_VectorPred Qs, HVX_Vector* Rb, Word32 Mu, HVX_VectorPair Vvv, HVX_Vector Vw)</pre>
<pre>vscatter(Rt,Mu,Vvv.w).h+=Vw32</pre>	<pre>void Q6_vscatteracc_RMWwV(HVX_Vector* Rb, Word32 Mu, HVX_VectorPair Vvv, HVX_Vector Vw)</pre>
<pre>vscatter(Rt,Mu,Vvv.w).h=Vw32</pre>	<pre>void Q6_vscatter_RMWwV(HVX_Vector* Rb, Word32 Mu, HVX_VectorPair Vvv, HVX_Vector Vw)</pre>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS								NT		t5				Parse		u1																
0	0	1	0	1	1	1	1	0	0	1	t	t	t	t	t	P	P	u	v	v	v	v	v	0	1	0	w	w	w	w	w	vscatter(Rt,Mu,Vvv.w).h=Vw32
0	0	1	0	1	1	1	1	0	0	1	t	t	t	t	t	P	P	u	v	v	v	v	v	1	1	0	w	w	w	w	w	vscatter(Rt,Mu,Vvv.w).h+=Vw32
ICLASS								NT		t5				Parse		u1					s2											
0	0	1	0	1	1	1	1	1	0	1	t	t	t	t	t	P	P	u	v	v	v	v	v	0	s	s	w	w	w	w	w	if (Qs4) vscatter(Rt,Mu,Vvv.w).h=Vw32

Field name	Description
ICLASS	Instruction class
NT	Nontemporal
Parse	Packet loop parse bits
s2	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
v5	Field to encode register v

6.13 HVX scatter

The HVX scatter instruction subclass includes instructions that perform scatter operations to the vector TCM.

Vector scatter

Vector scatter instructions perform scatter operations to the vector TCM. Scatter operations copy values from the register file to a region in VTCM. This region of memory is specified by two scalar registers: Rt32 is the base and Mu2 specifies the length - 1 of the region in bytes. This region must reside in VTCM and cannot cross a page boundary. A vector register, Vv32, specifies byte offsets in this region. Elements of either halfword or word granularity, specified by Vw32, are sent to addresses pointed to by Rt + Vv32 for each element. In memory, the element is either write to memory or accumulated with the memory (scatter-accumulate).

If multiple values are written to the same memory location, ordering is not guaranteed.

The offset vector, Vv32, can contain byte offsets specified in either halfword or word sizes. The final element addresses do not have to be byte aligned for regular scatter operations. However, for scatter accumulate instructions, the addresses are aligned. If an offset crosses the end of the scatter region, it is dropped. Offsets must be positive, otherwise they are dropped.

All vectors registers can be used immediately after the scatter operation.

```
vscatter(Rt,Mu,Vv.h)=Vw.h
```

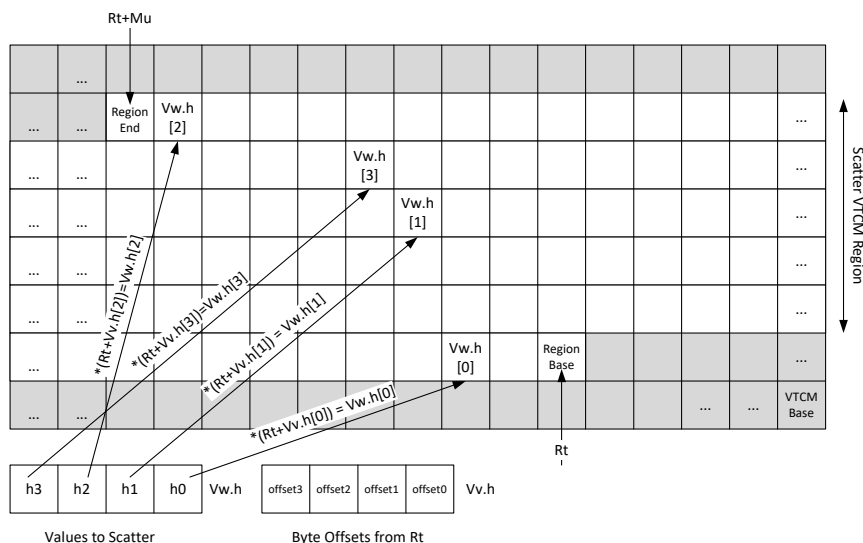
Rt – Scalar Indicating base address in VTCM

Vv – Vector with byte offsets from base

Mu – Scalar indicating length-1 of Region

Vw – Vector with halfword elements to be scattered

Example of vscatter (only first 4 elements shown)



Syntax	Behavior
<pre>if (Qs4) vscatter(Rt,Mu,Vv.h).h=Vw32</pre>	<pre>MuV = MuV (element_size-1); Rt = Rt & ~(element_size-1); for (i = 0; i < VELEM(16); i++) { EA = Rt+Vv.uh[i]; if ((Rt <= EA <= Rt + MuV) & QsV) *EA = VwV.uh[i]; }</pre>
<pre>if (Qs4) vscatter(Rt,Mu,Vv.h)=Vw32.h</pre>	Assembler mapped to: "if (Qs4) vscatter(Rt,Mu2,Vv.h).h = Vw32"
<pre>if (Qs4) vscatter(Rt,Mu,Vv.w).w=Vw32</pre>	<pre>MuV = MuV (element_size-1); Rt = Rt & ~(element_size-1); for (i = 0; i < VELEM(32); i++) { EA = Rt+Vv.uw[i]; if ((Rt <= EA <= Rt + MuV) & QsV) *EA = VwV.uw[i]; }</pre>
<pre>if (Qs4) vscatter(Rt,Mu,Vv.w)=Vw32.w</pre>	Assembler mapped to: "if (Qs4) vscatter(Rt,Mu2,Vv.w).w=Vw32"
<pre>vscatter(Rt,Mu,Vv.h)+=Vw32.h</pre>	Assembler mapped to: "vscatter(Rt,Mu2,Vv.h).h+=Vw32"
<pre>vscatter(Rt,Mu,Vv.h).h+=Vw32</pre>	<pre>MuV = MuV (element_size-1); Rt = Rt & ~(element_size-1); for (i = 0; i < VELEM(16); i++) { EA = (Rt+Vv.uh[i] = Vv.uh[i] & ~(ALIGNMENT- 1)); if (Rt <= EA <= Rt + MuV) *EA += VwV.uh[i]; }</pre>
<pre>vscatter(Rt,Mu,Vv.h).h=Vw32</pre>	<pre>MuV = MuV (element_size-1); Rt = Rt & ~(element_size-1); for (i = 0; i < VELEM(16); i++) { EA = Rt+Vv.uh[i]; if (Rt <= EA <= Rt + MuV) *EA = VwV.uh[i]; }</pre>
<pre>vscatter(Rt,Mu,Vv.h)=Vw32.h</pre>	Assembler mapped to: "vscatter(Rt,Mu2,Vv.h).h=Vw32"
<pre>vscatter(Rt,Mu,Vv.w)+=Vw32.w</pre>	Assembler mapped to: "vscatter(Rt,Mu2,Vv.w).w+=Vw32"
<pre>vscatter(Rt,Mu,Vv.w).w+=Vw32</pre>	<pre>MuV = MuV (element_size-1); Rt = Rt & ~(element_size-1); for (i = 0; i < VELEM(32); i++) { EA = (Rt+Vv.uw[i] = Vv.uw[i] & ~(ALIGNMENT- 1)); if (Rt <= EA <= Rt + MuV) *EA += VwV.uw[i]; }</pre>
<pre>vscatter(Rt,Mu,Vv.w).w=Vw32</pre>	<pre>MuV = MuV (element_size-1); Rt = Rt & ~(element_size-1); for (i = 0; i < VELEM(32); i++) { EA = Rt+Vv.uw[i]; if (Rt <= EA <= Rt + MuV) *EA = VwV.uw[i]; }</pre>

Syntax

```
vscatter(Rt,Mu,Vv.w)=Vw32.w
```

Behavior

Assembler mapped to:
"vscatter(Rt,Mu2,Vv.w).w=Vw32"

Class: COPROC_VMEM (slots 0)**Notes**

- This instruction can use any HVX resource.

Intrinsics

```
if (Qs4)
vscatter(Rt,Mu,Vv.h).h=Vw32
```

```
void Q6_vscatter_QRMVhV(HVX_VectorPred Qs,
HVX_Vector* Rb, Word32 Mu, HVX_Vector Vv,
HVX_Vector Vw)
```

```
if (Qs4)
vscatter(Rt,Mu,Vv.w).w=Vw32
```

```
void Q6_vscatter_QRMVwV(HVX_VectorPred Qs,
HVX_Vector* Rb, Word32 Mu, HVX_Vector Vv,
HVX_Vector Vw)
```

```
vscatter(Rt,Mu,Vv.h).h+=Vw32
```

```
void Q6_vscatteracc_RMVhV(HVX_Vector* Rb,
Word32 Mu, HVX_Vector Vv, HVX_Vector Vw)
```

```
vscatter(Rt,Mu,Vv.h).h=Vw32
```

```
void Q6_vscatter_RMVhV(HVX_Vector* Rb,
Word32 Mu, HVX_Vector Vv, HVX_Vector Vw)
```

```
vscatter(Rt,Mu,Vv.w).w+=Vw32
```

```
void Q6_vscatteracc_RMVwV(HVX_Vector* Rb,
Word32 Mu, HVX_Vector Vv, HVX_Vector Vw)
```

```
vscatter(Rt,Mu,Vv.w).w=Vw32
```

```
void Q6_vscatter_RMVwV(HVX_Vector* Rb,
Word32 Mu, HVX_Vector Vv, HVX_Vector Vw)
```

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS									NT				t5				Parse		u1													
0	0	1	0	1	1	1	1	0	0	1	t	t	t	t	t	P	P	u	v	v	v	v	v	0	0	0	w	w	w	w	w	vscatter(Rt,Mu,Vv.w).w=Vw32
0	0	1	0	1	1	1	1	0	0	1	t	t	t	t	t	P	P	u	v	v	v	v	v	0	0	1	w	w	w	w	w	vscatter(Rt,Mu,Vv.h).h=Vw32
0	0	1	0	1	1	1	1	0	0	1	t	t	t	t	t	P	P	u	v	v	v	v	v	1	0	0	w	w	w	w	w	vscatter(Rt,Mu,Vv.w).w+=Vw32
0	0	1	0	1	1	1	1	0	0	1	t	t	t	t	t	P	P	u	v	v	v	v	v	1	0	1	w	w	w	w	w	vscatter(Rt,Mu,Vv.h).h+=Vw32
ICLASS									NT		t5				Parse		u1							s2								
0	0	1	0	1	1	1	1	1	0	0	t	t	t	t	t	P	P	u	v	v	v	v	v	0	s	s	w	w	w	w	w	if (Qs4) vscatter(Rt,Mu,Vv.w).w=Vw32
0	0	1	0	1	1	1	1	1	0	0	t	t	t	t	t	P	P	u	v	v	v	v	v	1	s	s	w	w	w	w	w	if (Qs4) vscatter(Rt,Mu,Vv.h).h=Vw32

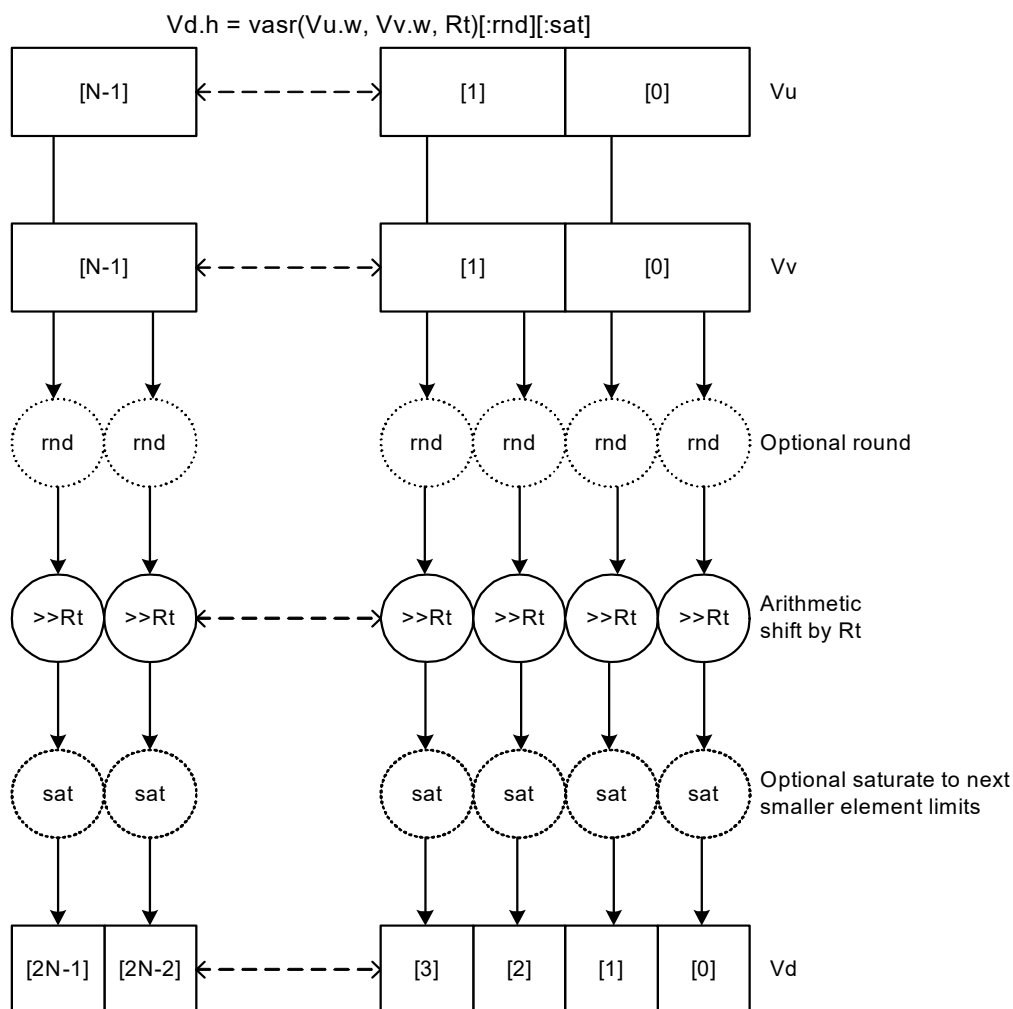
Field name		Description
ICLASS	Instruction class	
NT	Nontemporal	
Parse	Packet loop parse bits	
s2	Field to encode register s	
t5	Field to encode register t	
u1	Field to encode register u	
v5	Field to encode register v	

6.14 HVX shift resource

The HVX shift resource instruction subclass includes instructions that use the HVX shift resource.

Narrowing shift

Arithmetically shift-right the elements in vector registers Vu and Vv by the lower bits of the scalar register Rt. Each result is optionally saturated, rounded to infinity, and packed into a single destination vector register. Each even element in the destination vector register Vd comes from the vector register Vv. Each odd element in Vd comes from the vector register Vu.



Class: COPROC_VX (slots 0,1,2,3)

Notes

- Input scalar register Rt is limited to registers 0 through 7

Syntax	Behavior
<code>Vd.b=vasr(Vu.h,Vv.h,Rt)[:rnd]:sat</code>	<pre> for (i = 0; i < VELEM(16); i++) { shamt = Rt & 0x7; Vd.h[i].b[0]=sat₈(Vv.h[i] + (1<<(shamt-1)) >> shamt); Vd.h[i].b[1]=sat₈(Vu.h[i] + (1<<(shamt-1)) >> shamt); } </pre>
<code>Vd.h=vasr(Vu.w,Vv.w,Rt):rnd:sat</code>	<pre> for (i = 0; i < VELEM(32); i++) { shamt = Rt & 0xF; Vd.w[i].h[0]=sat₁₆(Vv.w[i] + (1 << (shamt-1)) >> shamt); Vd.w[i].h[1]=sat₁₆(Vu.w[i] + (1 << (shamt - 1)) >> shamt); } </pre>
<code>Vd.h=vasr(Vu.w,Vv.w,Rt)[:sat]</code>	<pre> for (i = 0; i < VELEM(32); i++) { shamt = Rt & 0xF; Vd.w[i].h[0]=[sat₁₆](Vv.w[i] >> shamt); Vd.w[i].h[1]=[sat₁₆](Vu.w[i] >> shamt); } </pre>
<code>Vd.ub=vasr(Vu.h,Vv.h,Rt)[:rnd]:sat</code>	<pre> for (i = 0; i < VELEM(16); i++) { shamt = Rt & 0x7; Vd.h[i].b[0]=usat₈(Vv.h[i] + (1 << (shamt-1)) >> shamt); Vd.h[i].b[1]=usat₈(Vu.h[i] + (1 << (shamt-1)) >> shamt); } </pre>
<code>Vd.ub=vasr(Vu.uh,Vv.uh,Rt)[:rnd]:sat</code>	<pre> for (i = 0; i < VELEM(16); i++) { shamt = Rt & 0x7; Vd.uh[i].b[0]=usat₈(Vv.uh[i] + (1<<(shamt - 1)) >> shamt); Vd.uh[i].b[1]=usat₈(Vu.uh[i] + (1<<(shamt-1)) >> shamt); } </pre>
<code>Vd.uh=vasr(Vu.uw,Vv.uw,Rt)[:rnd]:sat</code>	<pre> for (i = 0; i < VELEM(32); i++) { shamt = Rt & 0xF; Vd.uw[i].h[0]=usat₁₆(Vv.uw[i] + (1<<(shamt-1)) >> shamt); Vd.uw[i].h[1]=usat₁₆(Vu.uw[i] + (1<<(shamt-1)) >> shamt); } </pre>
<code>Vd.uh=vasr(Vu.w,Vv.w,Rt)[:rnd]:sat</code>	<pre> for (i = 0; i < VELEM(32); i++) { shamt = Rt & 0xF; Vd.w[i].h[0]=usat₁₆(Vv.w[i] + (1<<(shamt-1)) >> shamt); Vd.w[i].h[1]=usat₁₆(Vu.w[i] + (1<<(shamt-1)) >> shamt); } </pre>

- This instruction uses the HVX shift resource.

Intrinsics

Vd.b=vasr(Vu.h,Vv.h,Rt):rnd:sat	HVX_Vector Q6_Vb_vasr_VhVhR_rnd_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.b=vasr(Vu.h,Vv.h,Rt):sat	HVX_Vector Q6_Vb_vasr_VhVhR_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.h=vasr(Vu.w,Vv.w,Rt)	HVX_Vector Q6_Vh_vasr_VwVwR(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.h=vasr(Vu.w,Vv.w,Rt):rnd:sat	HVX_Vector Q6_Vh_vasr_VwVwR_rnd_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.h=vasr(Vu.w,Vv.w,Rt):sat	HVX_Vector Q6_Vh_vasr_VwVwR_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.ub=vasr(Vu.h,Vv.h,Rt):rnd:sat	HVX_Vector Q6_Vub_vasr_VhVhR_rnd_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.ub=vasr(Vu.h,Vv.h,Rt):sat	HVX_Vector Q6_Vub_vasr_VhVhR_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.ub=vasr(Vu.uh,Vv.uh,Rt):rnd:sat	HVX_Vector Q6_Vub_vasr_VuhVuhR_rnd_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.ub=vasr(Vu.uh,Vv.uh,Rt):sat	HVX_Vector Q6_Vub_vasr_VuhVuhR_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.uh=vasr(Vu.uw,Vv.uw,Rt):rnd:sat	HVX_Vector Q6_Vuh_vasr_VuwVuwR_rnd_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.uh=vasr(Vu.uw,Vv.uw,Rt):sat	HVX_Vector Q6_Vuh_vasr_VuwVuwR_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.uh=vasr(Vu.w,Vv.w,Rt):rnd:sat	HVX_Vector Q6_Vuh_vasr_VwVwR_rnd_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.uh=vasr(Vu.w,Vv.w,Rt):sat	HVX_Vector Q6_Vuh_vasr_VwVwR_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS													t3			Parse		u5								d5						
0	0	0	1	1	0	0	0	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.b=vasr(Vu.h,Vv.h,Rt):sat
0	0	0	1	1	0	0	0	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.uh=vasr(Vu.uw,Vv.uw,Rt):rnd:sat
0	0	0	1	1	0	0	0	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.uh=vasr(Vu.w,Vv.w,Rt):rnd:sat
0	0	0	1	1	0	0	0	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.ub=vasr(Vu.uh,Vv.uh,Rt):rnd:sat

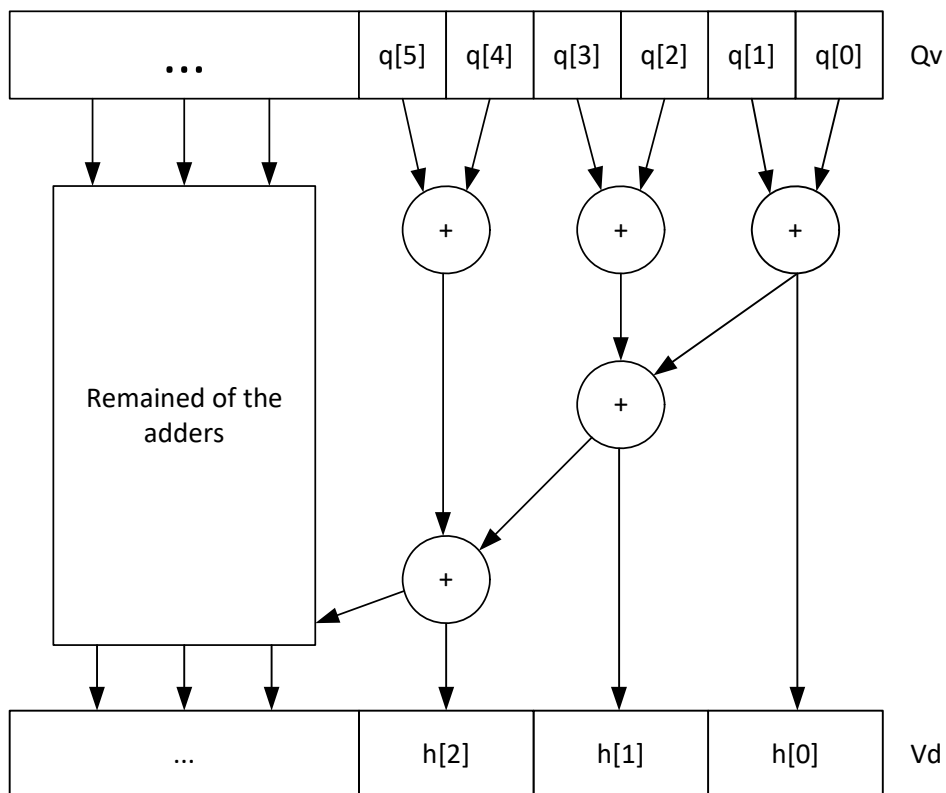
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	1	1	0	0	0	v	v	v	v	v	t	t	t	P	P	1	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.uh=vasr(Vu.uw,Vv.uw,Rt):sat
0	0	0	1	1	0	0	0	v	v	v	v	v	t	t	t	P	P	1	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.ub=vasr(Vu.uh,Vv.uh,Rt):sat
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.h=vasr(Vu.w,Vv.w,Rt)
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.h=vasr(Vu.w,Vv.w,Rt):sat
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.h=vasr(Vu.w,Vv.w,Rt):rnd:sat
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.uh=vasr(Vu.w,Vv.w,Rt):sat
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.ub=vasr(Vu.h,Vv.h,Rt):sat
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.ub=vasr(Vu.h,Vv.h,Rt):rnd:sat
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	1	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.b=vasr(Vu.h,Vv.h,Rt):rnd:sat

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t3	Field to encode register t
u5	Field to encode register u
v5	Field to encode register v

Compute contiguous offsets for valid positions

Perform a cumulative sum of the bits in the predicate register.

$Vd32.h = \text{prefixsum}(qv4)$



Syntax

$Vd.b = \text{prefixsum}(Qv4)$

$Vd.h = \text{prefixsum}(Qv4)$

$Vd.w = \text{prefixsum}(Qv4)$

Behavior

```
for (i = 0; i < VELEM(8); i++) {
    acc += QvV[i];
    Vd.ub[i] = acc;
}
```

```
for (i = 0; i < VELEM(16); i++) {
    acc += QvV[i*2+0];
    acc += QvV[i*2+1];
    Vd.uh[i] = acc;
}
```

```
for (i = 0; i < VELEM(32); i++) {
    acc += QvV[i*4+0];
    acc += QvV[i*4+1];
    acc += QvV[i*4+2];
    acc += QvV[i*4+3];
    Vd.uw[i] = acc;
}
```

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction uses the HVX shift resource.

Intrinsics

Vd.b=prefixsum(Qv4)	HVX_Vector Q6_Vb_prefixsum_Q(HVX_VectorPred Qv)
Vd.h=prefixsum(Qv4)	HVX_Vector Q6_Vh_prefixsum_Q(HVX_VectorPred Qv)
Vd.w=prefixsum(Qv4)	HVX_Vector Q6_Vw_prefixsum_Q(HVX_VectorPred Qv)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS									Parse									d5														
0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	1	P	P	1	-	-	0	0	0	0	1	0	d	d	d	d	d	Vd.b=prefixsum(Qv4)
0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	1	P	P	1	-	-	0	0	1	0	1	0	d	d	d	d	d	Vd.h=prefixsum(Qv4)
0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	1	P	P	1	-	-	0	1	0	0	1	0	d	d	d	d	d	Vd.w=prefixsum(Qv4)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
v2	Field to encode register v

Add - half precision vector by vector

Perform a vectorized half precision floating point add. The inputs are either both IEEE single precision, both 16-bit Qfloat, or one of each. The result is a 16-bit Qfloat vector.

Syntax	Behavior
<code>Vd.qf16=vadd(Vu.hf,Vv.hf)</code>	<pre> for (i = 0; i < VELEM(16); i++) { u = Vu.hf[i]; v = Vv.hf[i]; if (u.exp>v.exp) { exp = u.exp+((u.sig==0.0)? (-(FRAC_HF + 1)):ilogb(u.sig)); if (exp < v.exp) exp = v.exp; } else { exp = v.exp+((v.sig==0.0)? (-(FRAC_HF + 1)):ilogb(v.sig)); if (exp<u.exp) exp = u.exp; } sig_u = ldexp(u.sig, u.exp-exp); sig_v = ldexp(v.sig, v.exp-exp); if((u.sign^v.sign)==0){ sig = sig_u + sig_v; sig_low = (u.exp>v.exp) ? (sig_u-sig) + sig_v : (sig_v-sig)+sig_u; } else if((u.sign==0) && (v.sign==1)) { sig = sig_u - sig_v; sig_low = (u.exp>v.exp) ? (sig_u-sig)- sig_v : sig_u-(sig_v+sig); } else{ sig = sig_v - sig_u; sig_low = (v.exp>u.exp) ? (sig_v-sig)-sig_u : sig_v-(sig_u+sig); } Vd.qf16[i] = rnd_sat(exp,sig,sig_low); if(u.sign && v.sign) Vd.qf16[i] = - (Vd.qf16[i]); } </pre>
<code>Vd.qf16=vadd(Vu.qf16,Vv.hf)</code>	<pre> for (i = 0; i < VELEM(16); i++) { u = Vu.qf16[i]; v = Vv.hf[i]; if(v.sign) v.sig = (-1.0)*v.sig; if (u.exp>v.exp) { exp = u.exp+((u.sig==0.0)? (-(FRAC_HF + 1)):ilogb(u.sig)); if (exp<v.exp) exp = v.exp; } else { exp = v.exp+((v.sig==0.0)? (-(FRAC_HF+ 1)):ilogb(v.sig)); if (exp<u.exp) exp = u.exp; } sig_u = ldexp(u.sig, u.exp-exp); sig_v = ldexp(v.sig, v.exp-exp); sig = sig_u + sig_v; sig_low = (u.exp>v.exp) ? (sig_u-sig)+sig_v : (sig_v-sig)+sig_u; Vd.qf16[i] = rnd_sat(exp, sig, sig_low);} </pre>

Syntax

```
Vd.qf16=vadd(Vu.qf16,Vv.qf16)
```

Behavior

```
for (i = 0; i < VELEM(16); i++) {
    u = Vu.qf16[i];
    v = Vv.qf16[i];
    if (u.exp>v.exp) {
        exp = u.exp+((u.sig==0.0)? -(FRAC_HF+
1)):ilogb(u.sig));
        if (exp<v.exp) exp = v.exp;
    } else {
        exp = v.exp+((v.sig==0.0)? -(FRAC_HF+
1)):ilogb(v.sig));
        if (exp<u.exp) exp = u.exp;
    }
    sig_u = ldexp(u.sig, u.exp-exp);
    sig_v = ldexp(v.sig, v.exp-exp);
    sig = sig_u + sig_v;
    sig_low = (u.exp>v.exp) ? (sig_u-sig)+sig_v
: (sig_v-sig)+sig_u;
    Vd.qf16[i] = rnd_sat(exp,sig,sig_low);
}
```

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction uses the HVX shift resource.

Intrinsics

```
Vd.qf16=vadd(Vu.hf,Vv.hf)
```

```
HVX_Vector Q6_Vqf16_vadd_VhfVhf(HVX_Vector Vu,
HVX_Vector Vv)
```

```
Vd.qf16=vadd(Vu.qf16,Vv.hf)
```

```
HVX_Vector Q6_Vqf16_vadd_Vqf16Vhf(HVX_Vector
Vu, HVX_Vector Vv)
```

```
Vd.qf16=vadd(Vu.qf16,Vv.qf16)
```

```
HVX_Vector Q6_Vqf16_vadd_Vqf16Vqf16(HVX_Vector
Vu, HVX_Vector Vv)
```

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse		u5										d5				
0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.qf16=vadd(Vu.qf16,Vv.qf16)
0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.qf16=vadd(Vu.hf,Vv.hf)
0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.qf16=vadd(Vu.qf16,Vv.hf)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Add - single precision vector by vector

Perform a vectorized single precision floating point add. The inputs are either both IEEE single precision, both 32-bit Qfloat, or one of each. The result is a 32-bit Qfloat vector.

Syntax	Behavior
<code>Vd.qf32=vadd(Vu.qf32,Vv.qf32)</code>	<pre> for (i = 0; i < VELEM(32); i++) { u = Vu.qf32[i]; v = Vv.qf32[i]; if (u.exp>v.exp) { exp = u.exp+((u.sig==0.0)? (- (FRAC_SF+1)):ilogb(u.sig)); if (exp<v.exp) exp = v.exp; } else { exp = v.exp+((v.sig==0.0)? (- (FRAC_SF+1)):ilogb(v.sig)); if (exp<u.exp) exp = u.exp; } sig_u = ldexp(u.sig, u.exp-exp); sig_v = ldexp(v.sig, v.exp-exp); sig = sig_u + sig_v; sig_low = (u.exp>v.exp) ? (sig_u-sig)+sig_v : (sig_v-sig)+sig_u; Vd.qf32[i] = rnd_sat(exp,sig,sig_low); } </pre>
<code>Vd.qf32=vadd(Vu.qf32,Vv.sf)</code>	<pre> for (i = 0; i < VELEM(32); i++) { u = Vu.qf32[i]; v = Vv.sf[i]; if(v.sign) v.sig = (-1.0)*v.sig; if (u.exp>v.exp) { exp = u.exp+((u.sig==0.0)? (- (FRAC_SF+1)):ilogb(u.sig)); if (exp<v.exp) exp = v.exp; } else { exp = v.exp+((v.sig==0.0)? (- (FRAC_SF+1)):ilogb(v.sig)); if (exp<u.exp) exp = u.exp; } sig_u = ldexp(u.sig, u.exp-exp); sig_v = ldexp(v.sig, v.exp-exp); sig = sig_u + sig_v; sig_low = (u.exp>v.exp) ? (sig_u-sig)+sig_v : (sig_v-sig)+sig_u; Vd.qf32[i] = rnd_sat(exp,sig,sig_low); } </pre>
<code>Vd.qf32 = vadd(Vu.sf,Vv.sf)</code>	<pre> for (i = 0; i < VELEM(32); i++) { u = Vu.sf[i]; v = Vv.sf[i]; if (u.exp>v.exp) { exp = u.exp+((u.sig==0.0)? (- (FRAC_SF+ 1)):ilogb(u.sig)); if (exp<v.exp) exp = v.exp; } else { exp = v.exp+((v.sig==0.0)? (- (FRAC_SF+ 1)):ilogb(v.sig)); if (exp<u.exp) exp = u.exp;} </pre>

Syntax

```
Vd.qf32 = vadd(Vu.sf,Vv.sf)
(cont.)
```

Behavior

```
sig_u = ldexp(u.sig, u.exp-exp);
sig_v = ldexp(v.sig, v.exp-exp);
if((u.sign^v.sign)==0){
    sig = sig_u + sig_v;
    sig_low = (u.exp>v.exp) ? (sig_u-sig)+sig_v
: (sig_v-sig)+sig_u;
} else if((u.sign==0) && (v.sign==1)) {
    sig = sig_u - sig_v;
    sig_low = (u.exp>v.exp) ? (sig_u-sig)-sig_v
: sig_u-(sig_v+sig);
} else{
    sig = sig_v - sig_u;
    sig_low = (v.exp>u.exp) ? (sig_v-sig)-sig_u
: sig_v-(sig_u+sig);
}
Vd.qf32[i] = rnd_sat(exp,sig,sig_low);
if(u.sign && v.sign) Vd.qf32[i] = -
(Vd.qf32[i]);
}
```

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction uses the HVX shift resource.

Intrinsics

Vd.qf32=vadd(Vu.qf32,Vv.qf32)	HVX_Vector Q6_Vqf32_vadd_Vqf32Vqf32 (HVX_Vector Vu, HVX_Vector Vv)
Vd.qf32=vadd(Vu.qf32,Vv.sf)	HVX_Vector Q6_Vqf32_vadd_Vqf32Vsf (HVX_Vector Vu, HVX_Vector Vv)
Vd.qf32=vadd(Vu.sf,Vv.sf)	HVX_Vector Q6_Vqf32_vadd_VsfVsf (HVX_Vector Vu, HVX_Vector Vv)

Encoding

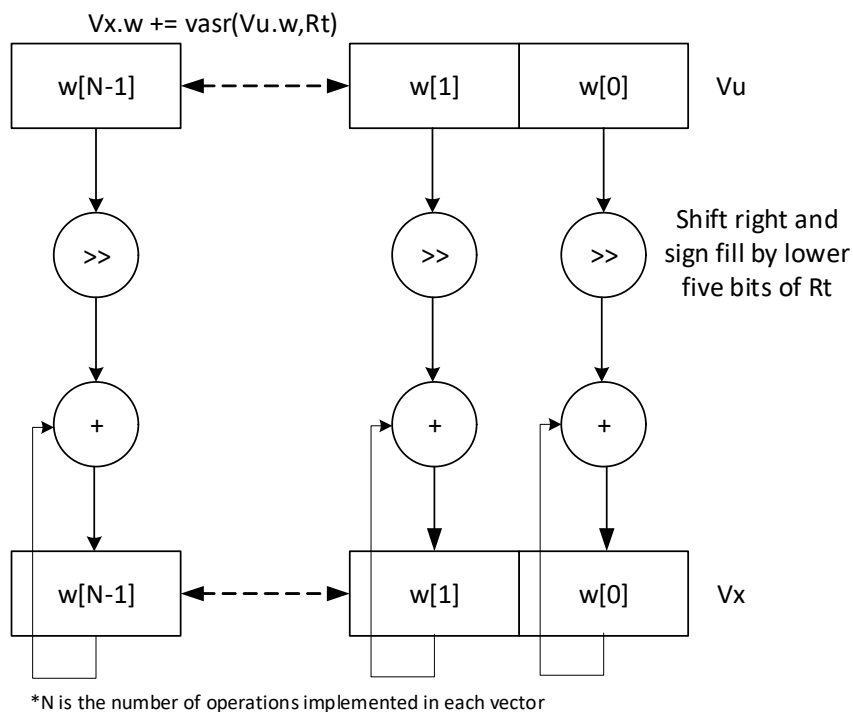
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse		u5					d5									
0	0	0	1	1	1	1	1	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.qf32=vadd(Vu.qf32,Vv.qf32)
0	0	0	1	1	1	1	1	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.qf32=vadd(Vu.sf,Vv.sf)
0	0	0	1	1	1	1	1	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.qf32=vadd(Vu.qf32,Vv.sf)

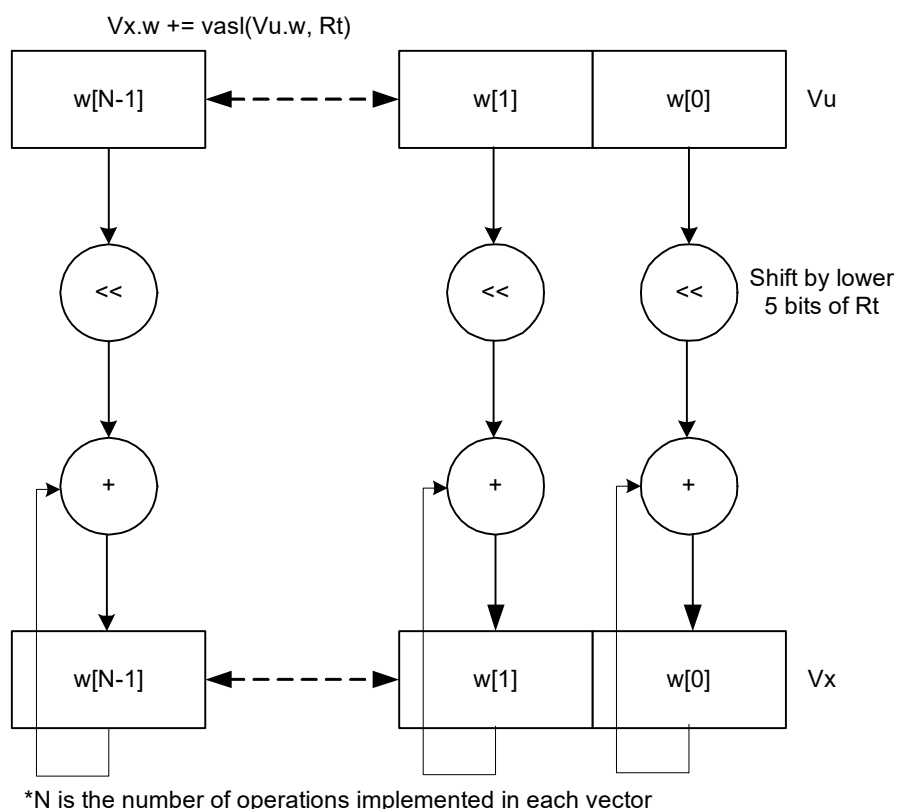
Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Shift and add

Each element in the vector register V_u is arithmetically shifted right by the value specified by the lower bits of the scalar register R_t . The result is then added to the destination vector register V_x . For signed word shifts the lower 5 bits of R_t specify the shift amount.

The left shift does not saturate the result to the element size.





Syntax	Behavior
$Vx.h += vasl(Vu.h, Rt)$	<pre>for (i = 0; i < VELEM(16); i++) { Vx.h[i] += (Vu.h[i] << (Rt & (16-1))); }</pre>
$Vx.h += vasr(Vu.h, Rt)$	<pre>for (i = 0; i < VELEM(16); i++) { Vx.h[i] += (Vu.h[i] >> (Rt & (16-1))); }</pre>
$Vx.w += vasl(Vu.w, Rt)$	<pre>for (i = 0; i < VELEM(32); i++) { Vx.w[i] += (Vu.w[i] << (Rt & (32-1))); }</pre>
$Vx.w += vasr(Vu.w, Rt)$	<pre>for (i = 0; i < VELEM(32); i++) { Vx.w[i] += (Vu.w[i] >> (Rt & (32-1))); }</pre>

Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction uses the HVX shift resource.

Intrinsics

<code>Vx.h+=vasl(Vu.h,Rt)</code>	<code>HVX_Vector Q6_Vh_vaslace_VhVhR(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)</code>
<code>Vx.h+=vasr(Vu.h,Rt)</code>	<code>HVX_Vector Q6_Vh_vasrace_VhVhR(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)</code>
<code>Vx.w+=vasl(Vu.w,Rt)</code>	<code>HVX_Vector Q6_Vw_vaslace_VwVwR(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)</code>
<code>Vx.w+=vasr(Vu.w,Rt)</code>	<code>HVX_Vector Q6_Vw_vasrace_VwVwR(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS												t5				Parse		u5										x5				
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	1	u	u	u	u	u	0	1	0	x	x	x	x	x	Vx.w+=vasl(Vu.w,Rt)
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	1	u	u	u	u	u	1	0	1	x	x	x	x	x	Vx.w+=vasr(Vu.w,Rt)
0	0	0	1	1	0	0	1	1	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	1	1	1	x	x	x	x	x	Vx.h+=vasr(Vu.h,Rt)
0	0	0	1	1	0	0	1	1	0	1	t	t	t	t	t	P	P	1	u	u	u	u	u	1	0	1	x	x	x	x	x	Vx.h+=vasl(Vu.h,Rt)

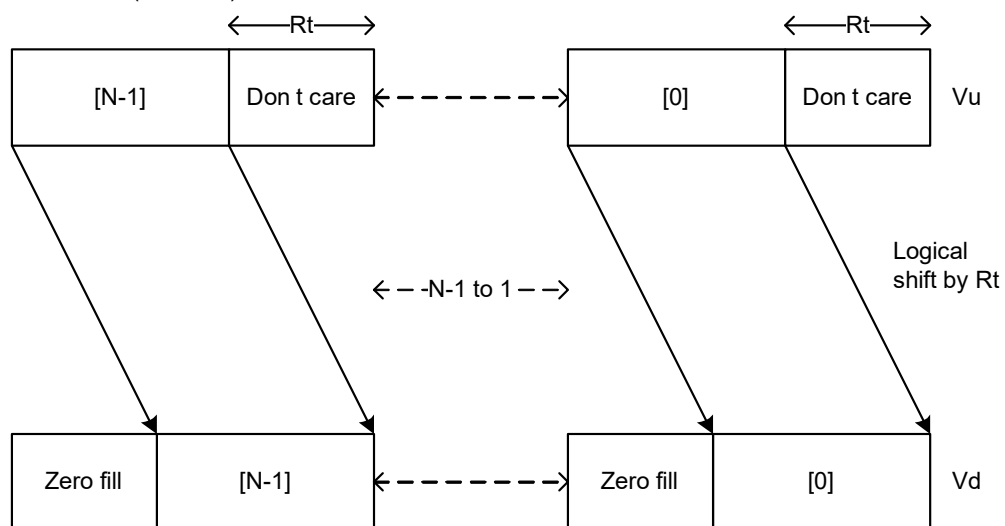
Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
t5	Field to encode register t
u5	Field to encode register u
x5	Field to encode register x

Shift

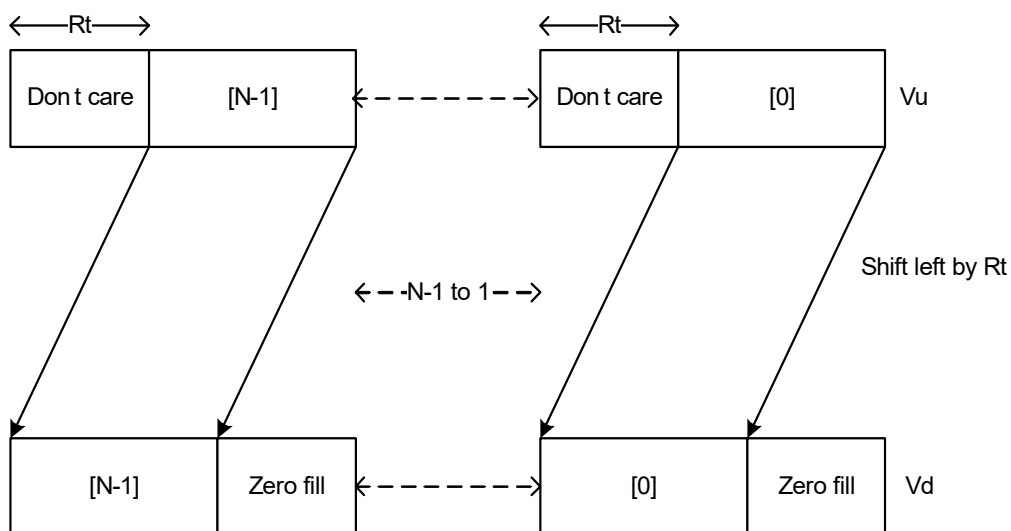
Each element in the vector register V_u is arithmetically (logically) shifted right (left) by the value specified in the lower bits of the corresponding element of vector register V_v (or scalar register R_t). For halfword shifts, the lower four bits are used, while for word shifts the lower five bits are used.

The logical left shift does not saturate the result to the element size.

$$Vd.w = vlsl(Vu.w, Rt)$$



$$Vd.w = vasl(Vu.w, Rt)$$



Syntax	Behavior
<code>Vd.b=vasr(Vu.h,Vv.h,Rt)[:rnd]:sat</code>	<pre> for (i = 0; i < VELEM(16); i++) { shamt = Rt & 0x7; Vd.h[i].b[0]=sat₈(Vv.h[i] + (1<<(shamt-1)) >> shamt); Vd.h[i].b[1]=sat₈(Vu.h[i] + (1<<(shamt-1)) >> shamt); } </pre>
<code>Vd.h=vasl(Vu.h,Rt)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vd.h[i] = (Vu.h[i] << (Rt & (16-1))); } </pre>
<code>Vd.h=vasl(Vu.h,Vv.h)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vd.h[i] = (sxt_{(4+1)->16}(Vv.h[i]) > 0)?(Vu.h[i] << sxt_{(4+1)->16}(Vv.h[i])):(Vu.h[i] >> sxt_{(4+1)->16}(Vv.h[i])); } </pre>
<code>Vd.h=vasr(Vu.h,Rt)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vd.h[i] = (Vu.h[i] >> (Rt & (16-1))); } </pre>
<code>Vd.h=vasr(Vu.h,Vv.h)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vd.h[i] = (sxt_{(4+1)->16}(Vv.h[i]) > 0)?(Vu.h[i] >> sxt_{(4+1)->16}(Vv.h[i])):(Vu.h[i] << sxt_{(4+1)->16}(Vv.h[i])); } </pre>
<code>Vd.h=vasr(Vu.w,Vv.w,Rt):rnd:sat</code>	<pre> for (i = 0; i < VELEM(32); i++) { shamt = Rt & 0xF; Vd.w[i].h[0]=sat₁₆(Vv.w[i] + (1<<(shamt-1)) >> shamt); Vd.w[i].h[1]=sat₁₆(Vu.w[i] + (1<<(shamt-1)) >> shamt); } </pre>
<code>Vd.h=vasr(Vu.w,Vv.w,Rt)[:sat]</code>	<pre> for (i = 0; i < VELEM(32); i++) { shamt = Rt & 0xF; Vd.w[i].h[0]=[sat₁₆](Vv.w[i] >> shamt); Vd.w[i].h[1]=[sat₁₆](Vu.w[i] >> shamt); } </pre>
<code>Vd.h=vlsr(Vu.h,Vv.h)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = (sxt_{(4+1)->16}(Vv.h[i]) > 0)?(Vu.uh[i] >>> sxt_{(4+1)->16}(Vv.h[i])):(Vu.uh[i] << sxt_{(4+1)->16}(Vv.h[i])); } </pre>

Syntax	Behavior
<code>Vd.ub = vasr(Vu.h, Vv.h, Rt)[:rnd]:sat</code>	<pre> for (i = 0; i < VELEM(16); i++) { shamt = Rt & 0x7; Vd.h[i].b[0]=usat₈(Vv.h[i] + (1<<(shamt-1)) >> shamt); Vd.h[i].b[1]=usat₈(Vu.h[i] + (1<<(shamt-1)) >> shamt); } </pre>
<code>Vd.ub = vasr(Vu.uh, Vv.uh, Rt)[:rnd]:sat</code>	<pre> for (i = 0; i < VELEM(16); i++) { shamt = Rt & 0x7; Vd.uh[i].b[0]=usat₈(Vv.uh[i] + (1<<(shamt-1)) >> shamt); Vd.uh[i].b[1]=usat₈(Vu.uh[i] + (1<<(shamt-1)) >> shamt); } </pre>
<code>Vd.ub = vlsr(Vu.ub, Rt)</code>	<pre> for (i = 0; i < VELEM(8); i++) { Vd.b[i] = Vu.ub[i] >> (Rt & 0x7); } </pre>
<code>Vd.uh = vasr(Vu.uw, Vv.uw, Rt)[:rnd]:sat</code>	<pre> for (i = 0; i < VELEM(32); i++) { shamt = Rt & 0xF; Vd.uw[i].h[0]=usat₁₆(Vv.uw[i] + (1<<(shamt-1)) >> shamt); Vd.uw[i].h[1]=usat₁₆(Vu.uw[i] + (1<<(shamt-1)) >> shamt); } </pre>
<code>Vd.uh=vasr(Vu.w,Vv.w,Rt)[:rnd]:sat</code>	<pre> for (i = 0; i < VELEM(32); i++) { shamt = Rt & 0xF; Vd.w[i].h[0]=usat₁₆(Vv.w[i] + (1<<(shamt-1)) >> shamt); Vd.w[i].h[1]=usat₁₆(Vu.w[i] + (1<<(shamt-1)) >> shamt); } </pre>
<code>Vd.uh=vlsr(Vu.uh,Rt)</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = (Vu.uh[i] >> (Rt & (16-1))); } </pre>
<code>Vd.uw=vlsr(Vu.uw,Rt)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vd.uw[i] = (Vu.uw[i] >> (Rt & (32-1))); } </pre>
<code>Vd.w=vasl(Vu.w,Rt)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vd.w[i] = (Vu.w[i] << (Rt & (32-1))); } </pre>
<code>Vd.w=vasl(Vu.w,Vv.w)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vd.w[i] = (sxt_{(5+1)->32}(Vv.w[i]) > 0) ? (Vu.w[i] << sxt_{(5+1)->32}(Vv.w[i])) : (Vu.w[i] >> sxt_{(5+1)->32}(Vv.w[i])); } </pre>
<code>Vd.w=vasr(Vu.w,Rt)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vd.w[i] = (Vu.w[i] >> (Rt & (32-1))); } </pre>

Syntax	Behavior
<code>Vd.w=vasr(Vu.w,Vv.w)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vd.w[i] = (sxt₍₅₊₁₎->₃₂(Vv.w[i]) > 0)?(Vu.w[i] >> sxt₍₅₊₁₎- >₃₂(Vv.w[i])):(Vu.w[i] << sxt₍₅₊₁₎- >₃₂(Vv.w[i])); } </pre>
<code>Vd.w=vlsr(Vu.w,Vv.w)</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vd.w[i] = (sxt₍₅₊₁₎->₃₂(Vv.w[i]) > 0)?(Vu.w[i] >>> sxt₍₅₊₁₎- >₃₂(Vv.w[i])):(Vu.w[i] << sxt₍₅₊₁₎- >₃₂(Vv.w[i])); } </pre>

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- Input scalar register Rt is limited to registers 0 through 7
- This instruction uses the HVX shift resource.

Intrinsics

<code>Vd.b=vasr(Vu.h,Vv.h,Rt):rnd:sat</code>	HVX_Vector Q6_Vb_vasr_VhVhR_rnd_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
<code>Vd.b=vasr(Vu.h,Vv.h,Rt):sat</code>	HVX_Vector Q6_Vb_vasr_VhVhR_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
<code>Vd.h=vasl(Vu.h,Rt)</code>	HVX_Vector Q6_Vh_vasl_VhR(HVX_Vector Vu, Word32 Rt)
<code>Vd.h=vasl(Vu.h,Vv.h)</code>	HVX_Vector Q6_Vh_vasl_VhVh(HVX_Vector Vu, HVX_Vector Vv)
<code>Vd.h=vasr(Vu.h,Rt)</code>	HVX_Vector Q6_Vh_vasr_VhR(HVX_Vector Vu, Word32 Rt)
<code>Vd.h=vasr(Vu.h,Vv.h)</code>	HVX_Vector Q6_Vh_vasr_VhVh(HVX_Vector Vu, HVX_Vector Vv)
<code>Vd.h=vasr(Vu.w,Vv.w,Rt)</code>	HVX_Vector Q6_Vh_vasr_VwVwR(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
<code>Vd.h=vasr(Vu.w,Vv.w,Rt):rnd:sat</code>	HVX_Vector Q6_Vh_vasr_VwVwR_rnd_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
<code>Vd.h=vasr(Vu.w,Vv.w,Rt):sat</code>	HVX_Vector Q6_Vh_vasr_VwVwR_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
<code>Vd.h=vlsr(Vu.h,Vv.h)</code>	HVX_Vector Q6_Vh_vlsr_VhVh(HVX_Vector Vu, HVX_Vector Vv)

Vd.ub=vasr(Vu.h,Vv.h,Rt):rnd:sat	HVX_Vector Q6_Vub_vasr_VhVhR_rnd_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.ub=vasr(Vu.h,Vv.h,Rt):sat	HVX_Vector Q6_Vub_vasr_VhVhR_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.ub=vasr(Vu.uh,Vv.uh,Rt):rnd:sat	HVX_Vector Q6_Vub_vasr_VuhVuhR_rnd_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.ub=vasr(Vu.uh,Vv.uh,Rt):sat	HVX_Vector Q6_Vub_vasr_VuhVuhR_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.ub=vlsr(Vu.ub,Rt)	HVX_Vector Q6_Vub_vlsr_VubR(HVX_Vector Vu, Word32 Rt)
Vd.uh=vasr(Vu.uw,Vv.uw,Rt):rnd:sat	HVX_Vector Q6_Vuh_vasr_VuwVuwR_rnd_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.uh=vasr(Vu.uw,Vv.uw,Rt):sat	HVX_Vector Q6_Vuh_vasr_VuwVuwR_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.uh=vasr(Vu.w,Vv.w,Rt):rnd:sat	HVX_Vector Q6_Vuh_vasr_VwVwR_rnd_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.uh=vasr(Vu.w,Vv.w,Rt):sat	HVX_Vector Q6_Vuh_vasr_VwVwR_sat(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.uh=vlsr(Vu.uh,Rt)	HVX_Vector Q6_Vuh_vlsr_VuhR(HVX_Vector Vu, Word32 Rt)
Vd.uw=vlsr(Vu.uw,Rt)	HVX_Vector Q6_Vuw_vlsr_VuwR(HVX_Vector Vu, Word32 Rt)
Vd.w=vasl(Vu.w,Rt)	HVX_Vector Q6_Vw_vasl_VwR(HVX_Vector Vu, Word32 Rt)
Vd.w=vasl(Vu.w,Vv.w)	HVX_Vector Q6_Vw_vasl_VwVw(HVX_Vector Vu, HVX_Vector Vv)
Vd.w=vasr(Vu.w,Rt)	HVX_Vector Q6_Vw_vasr_VwR(HVX_Vector Vu, Word32 Rt)
Vd.w=vasr(Vu.w,Vv.w)	HVX_Vector Q6_Vw_vasr_VwVw(HVX_Vector Vu, HVX_Vector Vv)
Vd.w=vlsr(Vu.w,Vv.w)	HVX_Vector Q6_Vw_vlsr_VwVw(HVX_Vector Vu, HVX_Vector Vv)

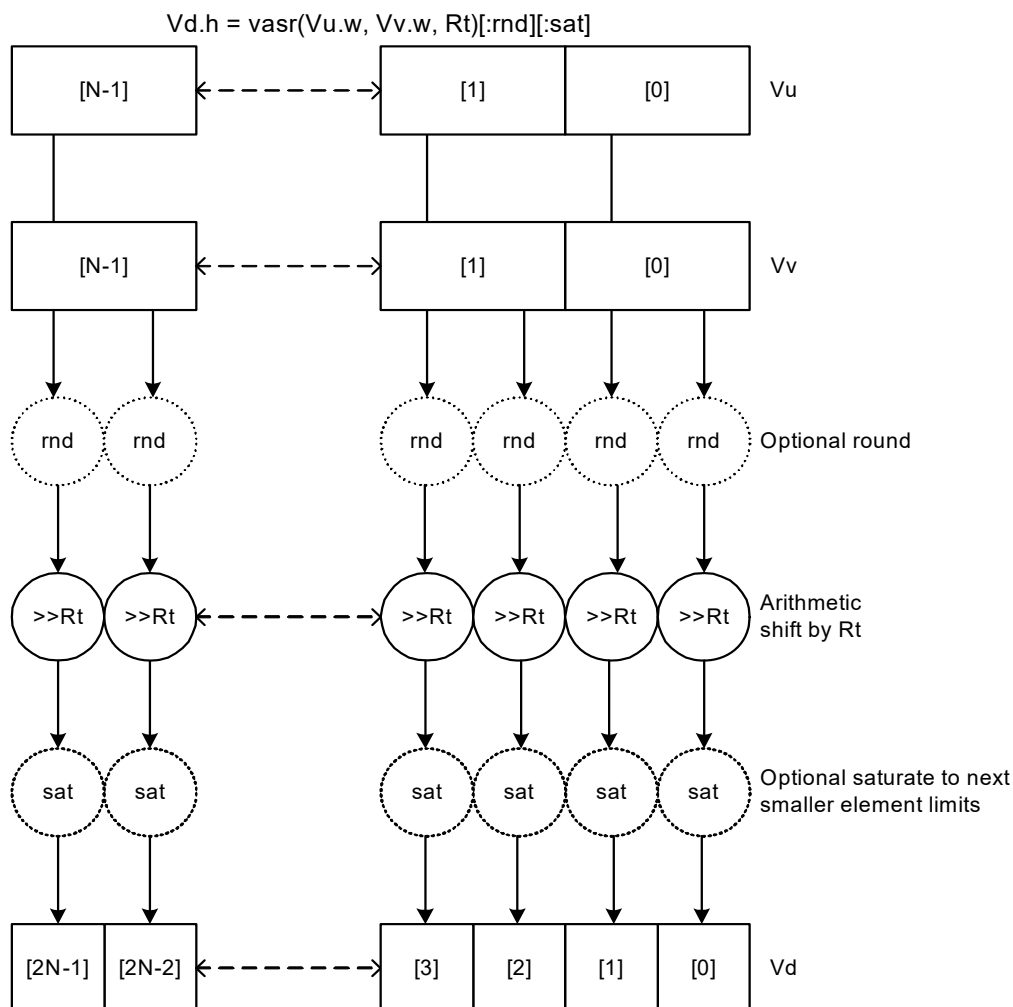
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
ICLASS														t3			Parse		u5										d5					
0	0	0	1	1	0	0	0	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.b=vasr(Vu.h,Vv.h,Rt):sat		
0	0	0	1	1	0	0	0	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.uh=vasr(Vu.uw,Vv.uw,Rt):rnd:sat		
0	0	0	1	1	0	0	0	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.uh=vasr(Vu.w,Vv.w,Rt):rnd:sat		
0	0	0	1	1	0	0	0	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.ub=vasr(Vu.uh,Vv.uh,Rt):rnd:sat		
0	0	0	1	1	0	0	0	v	v	v	v	v	t	t	t	P	P	1	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.uh=vasr(Vu.uw,Vv.uw,Rt):sat		
0	0	0	1	1	0	0	0	v	v	v	v	v	t	t	t	P	P	1	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.ub=vasr(Vu.uh,Vv.uh,Rt):sat		
ICLASS														t5			Parse		u5										d5					
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.w=vasr(Vu.w,Rt)		
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.h=vasr(Vu.h,Rt)		
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.w=vasl(Vu.w,Rt)		
0	0	0	1	1	0	0	1	1	0	0	t	t	t	t	t	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.h=vasl(Vu.h,Rt)		
0	0	0	1	1	0	0	1	1	0	0	t	t	t	t	t	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.uw=vlsr(Vu.uw,Rt)		
0	0	0	1	1	0	0	1	1	0	0	t	t	t	t	t	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.uh=vlsr(Vu.uh,Rt)		
0	0	0	1	1	0	0	1	1	0	0	t	t	t	t	t	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.ub=vlsr(Vu.ub,Rt)		
ICLASS														t3			Parse		u5										d5					
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.h=vasr(Vu.w,Vv.w,Rt)		
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.h=vasr(Vu.w,Vv.w,Rt):sat		
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.h=vasr(Vu.w,Vv.w,Rt):rnd:sat		
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.uh=vasr(Vu.w,Vv.w,Rt):sat		
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.ub=vasr(Vu.h,Vv.h,Rt):sat		
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.ub=vasr(Vu.h,Vv.h,Rt):rnd:sat		
0	0	0	1	1	0	1	1	v	v	v	v	v	t	t	t	P	P	1	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.b=vasr(Vu.h,Vv.h,Rt):rnd:sat		
ICLASS																Parse		u5										d5						
0	0	0	1	1	1	1	1	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.w=vasr(Vu.w,Vv.w)		
0	0	0	1	1	1	1	1	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.w=vlsr(Vu.w,Vv.w)		
0	0	0	1	1	1	1	1	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.h=vlsr(Vu.h,Vv.h)		
0	0	0	1	1	1	1	1	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.h=vasr(Vu.h,Vv.h)		
0	0	0	1	1	1	1	1	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.w=vasl(Vu.w,Vv.w)		
0	0	0	1	1	1	1	1	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.h=vasl(Vu.h,Vv.h)		

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
t3	Field to encode register t
t5	Field to encode register t
u5	Field to encode register u
v5	Field to encode register v

Narrowing shift by vector

Arithmetically shift-right the elements in vector register pair V_{uu} by the lower bits of the elements in vector register V_v . Each result is optionally saturated, rounded to infinity, and packed into a single destination vector register. Each even element in the destination vector register V_d comes from the vector register V_{u+1} . Each odd element in V_d comes from the vector register V_u .



Syntax

```
Vd.ub=vasr(Vuu.ub,Vv.ub)[:rnd]:sat
```

```
Vd.uh=vasr(Vuu.w,Vv.uh)[:rnd]:sat
```

Behavior

```
for (i = 0; i < VELEM(16); i++) {
    shamt = Vv.ub[2*i+0] & 0x7;
    Vd.uh[i].b[0]=usat8(Vuu.v[0].uh[i] + (1
    << (shamt-1)) >> shamt);
    shamt = Vv.ub[2*i+1] & 0x7;
    Vd.uh[i].b[1]=usat8(Vuu.v[1].uh[i] + (1
    << (shamt-1)) >> shamt);
}
```

```
for (i = 0; i < VELEM(32); i++) {
    shamt = Vv.uh[2*i+0] & 0xF;
    Vd.w[i].h[0]=usat16(Vuu.v[0].w[i] + (1
    << (shamt-1)) >> shamt);
}
```

Syntax

```
Vd.uh=vasr(Vuu.w,Vv.uh)[:rnd]:sat
(cont.)
```

Behavior

```
shamt = Vv.uh[2*i+1] & 0xF;
Vd.w[i].h[1]=usat16(Vuu.v[1].w[i] + (1
<< (shamt-1)) >> shamt);
}
```

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction cannot be paired with a HVX permute instruction
- This instruction uses the HVX shift resource.
- If a packet contains this instruction and a HVX ALU op, the ALU OP must be unary.

Intrinsics

Vd.ub=vasr(Vuu.uh,Vv.ub):rnd:sat	HVX_Vector Q6_Vub_vasr_WuhVub_rnd_sat(HVX_VectorPair Vuu, HVX_Vector Vv)
Vd.ub=vasr(Vuu.uh,Vv.ub):sat	HVX_Vector Q6_Vub_vasr_WuhVub_sat(HVX_VectorPair Vuu, HVX_Vector Vv)
Vd.uh=vasr(Vuu.w,Vv.uh):rnd:sat	HVX_Vector Q6_Vuh_vasr_WwVuh_rnd_sat(HVX_VectorPair Vuu, HVX_Vector Vv)
Vd.uh=vasr(Vuu.w,Vv.uh):sat	HVX_Vector Q6_Vuh_vasr_WwVuh_sat(HVX_VectorPair Vuu, HVX_Vector Vv)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS															Parse		u5										d5					
0	0	0	1	1	1	0	1	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.uh=vasr(Vuu.w,Vv.uh):sat
0	0	0	1	1	1	0	1	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.uh=vasr(Vuu.w,Vv.uh):rnd:sat
0	0	0	1	1	1	0	1	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.ub=vasr(Vuu.uh,Vv.ub):sat
0	0	0	1	1	1	0	1	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.ub=vasr(Vuu.uh,Vv.ub):rnd:sat

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Convert IEEE floating point to IEEE integer

Convert input IEEE 754 floating point output vector registers to an IEEE integer vector register.

Syntax	Behavior
<code>Vd.h=Vu.hf</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.h[i] = conv_h_hf(Vu.hf[i]); }</pre>
<code>Vd.w=Vu.sf</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i] = conv_w_sf(Vu.sf[i]); }</pre>

Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction uses the HVX shift resource.

Intrinsics

<code>Vd.h=Vu.hf</code>	<code>HVX_Vector Q6_Vh_equals_Vhf(HVX_Vector Vu)</code>
<code>Vd.w=Vu.sf</code>	<code>HVX_Vector Q6_Vw_equals_Vsf(HVX_Vector Vu)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse			u5								d5					
0	0	0	1	1	1	1	0	-	-	0	-	-	1	0	1	P	P	1	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.w=Vu.sf
0	0	0	1	1	1	1	0	-	-	0	-	-	1	0	1	P	P	1	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.h=Vu.hf

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u

Convert IEEE integer to IEEE floating point

These instructions convert input IEEE integer vector registers to an IEEE 754 floating point output vector register.

Syntax	Behavior
<code>Vd.hf=Vu.h</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.hf[i] = conv_hf_h(Vu.h[i]); }</pre>
<code>Vd.sf=Vu.w</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.sf[i] = conv_sf_w(Vu.w[i]); }</pre>

Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction uses the HVX shift resource.

Intrinsics

`Vd.hf=Vu.h`

`HVX_Vector Q6_Vhf_equals_Vh(HVX_Vector Vu)`

`Vd.sf=Vu.w`

`HVX_Vector Q6_Vsf_equals_Vw(HVX_Vector Vu)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS								Parse								u5								d5								
0	0	0	1	1	1	1	0	-	-	0	-	-	1	0	1	P	P	1	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.sf=Vu.w
0	0	0	1	1	1	1	0	-	-	0	-	-	1	0	1	P	P	1	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.hf=Vu.h

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u

Convert Qfloat to IEEE floating point

These instructions convert Qfloat input vector register(s) to an IEEE output vector register.

Syntax	Behavior
Vd.hf=Vu.qf16	<pre>for (i = 0; i < VELEM(16); i++) { u = Vu.qf16[i]; Vd.hf[i] = rnd_sat(u.exp,u.sig); }</pre>
Vd.hf=Vuu.qf32	<pre>for (i = 0; i < VELEM(32); i++) { u0 = Vuu.v[0].qf32[i]; u1 = Vuu.v[1].qf32[i]; Vd.hf[2*i] = rnd_sat(u0.exp,u0.sig); Vd.hf[2*i+1] = rnd_sat(u1.exp,u1.sig); }</pre>
Vd.sf=Vu.qf32	<pre>for (i = 0; i < VELEM(32); i++) { u = Vu.qf32[i]; Vd.sf[i] = rnd_sat(u.exp,u.sig); }</pre>

Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction uses the HVX shift resource.

Intrinsics

Vd.hf=Vu.qf16	HVX_Vector Q6_Vhf_equals_Vqf16(HVX_Vector Vu)
Vd.hf=Vuu.qf32	HVX_Vector Q6_Vhf_equals_Wqf32(HVX_VectorPair Vuu)
Vd.sf=Vu.qf32	HVX_Vector Q6_Vsf_equals_Vqf32(HVX_Vector Vu)

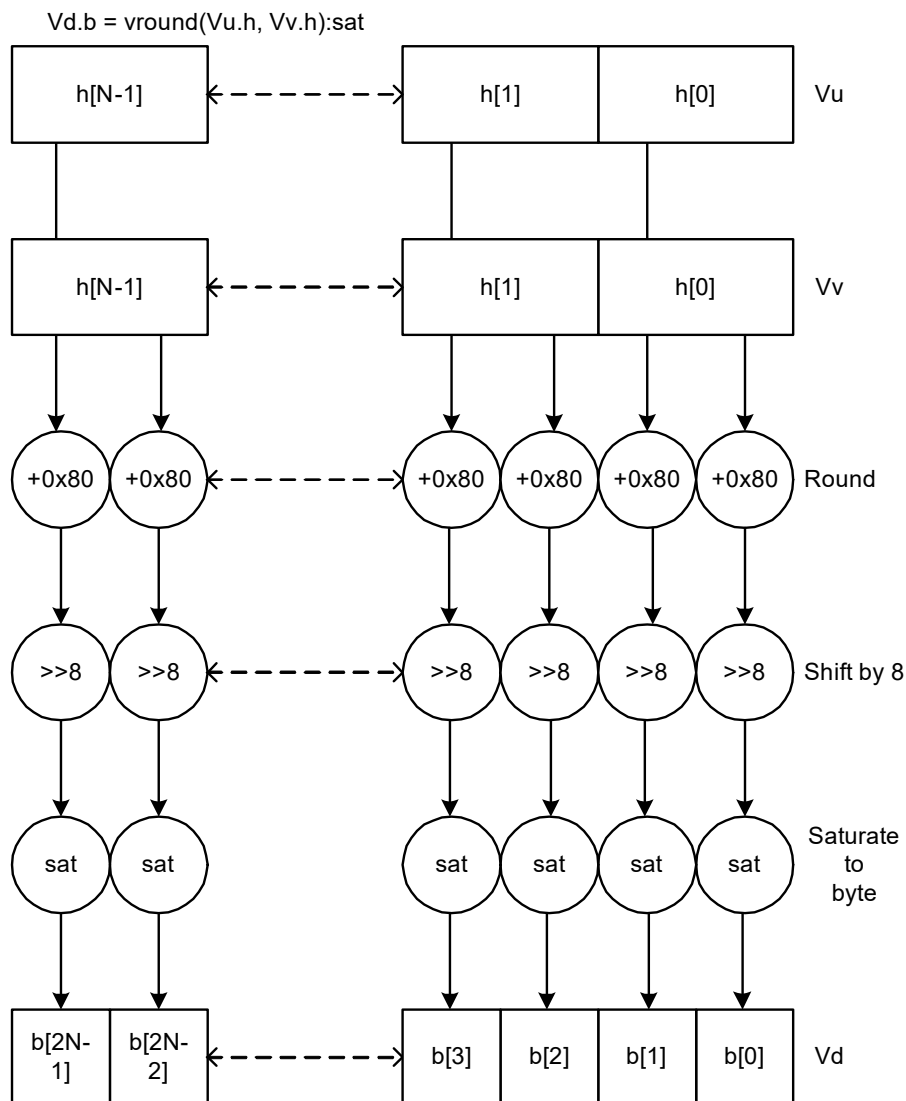
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse				u5								d5				
0	0	0	1	1	1	1	0	-	-	0	-	-	1	0	0	P	P	1	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.sf=Vu.qf32
0	0	0	1	1	1	1	0	-	-	0	-	-	1	0	0	P	P	1	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.hf=Vu.qf16
0	0	0	1	1	1	1	0	-	-	0	-	-	1	0	0	P	P	1	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.hf=Vuu.qf32

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u

Round to next smaller element size

Pack signed words to signed or unsigned halfwords, add 0x8000 to the lower 16 bits, logically or arithmetically right-shift by 16, and saturate the results to unsigned or signed halfwords, respectively. Alternatively, pack signed halfwords to signed or unsigned bytes, add 0x80 to the lower eight bits, logically or arithmetically right-shift by eight, and saturate the results to unsigned or signed bytes, respectively. The odd elements in the destination vector register Vd come from vector register Vv, and the even elements from Vu.



Syntax

```
Vd.b=vround(Vu.h,Vv.h):sat
```

Behavior

```
for (i = 0; i < VELEM(16); i++) {
    Vd.uh[i].b[0]=sat8((Vv.h[i] + 0x80) >> 8);
    Vd.uh[i].b[1]=sat8((Vu.h[i] + 0x80) >> 8);
}
```

Syntax	Behavior
<code>Vd.h=vround(Vu.w,Vv.w):sat</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vd.uw[i].h[0]=sat₁₆((Vv.w[i] + 0x8000) >> 16); Vd.uw[i].h[1]=sat₁₆((Vu.w[i] + 0x8000) >> 16); } </pre>
<code>Vd.ub=vround(Vu.h,Vv.h):sat</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vd.uh[i].b[0]=usat₈((Vv.h[i] + 0x80) >> 8); Vd.uh[i].b[1]=usat₈((Vu.h[i] + 0x80) >> 8); } </pre>
<code>Vd.ub=vround(Vu.uh,Vv.uh):sat</code>	<pre> for (i = 0; i < VELEM(16); i++) { Vd.uh[i].b[0]=usat₈((Vv.uh[i] + 0x80) >> 8); Vd.uh[i].b[1]=usat₈((Vu.uh[i] + 0x80) >> 8); } </pre>
<code>Vd.uh=vround(Vu.uw,Vv.uw):sat</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vd.uw[i].h[0]=usat₁₆((Vv.uw[i] + 0x8000) >> 16); Vd.uw[i].h[1]=usat₁₆((Vu.uw[i] + 0x8000) >> 16); } </pre>
<code>Vd.uh=vround(Vu.w,Vv.w):sat</code>	<pre> for (i = 0; i < VELEM(32); i++) { Vd.uw[i].h[0]=usat₁₆((Vv.w[i] + 0x8000) >> 16); Vd.uw[i].h[1]=usat₁₆((Vu.w[i] + 0x8000) >> 16); } </pre>

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction uses the HVX shift resource.

Intrinsics

<code>Vd.b=vround(Vu.h,Vv.h):sat</code>	<code>HVX_Vector Q6_Vb_vround_VhVh_sat (HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.h=vround(Vu.w,Vv.w):sat</code>	<code>HVX_Vector Q6_Vh_vround_VwVw_sat (HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.ub=vround(Vu.h,Vv.h):sat</code>	<code>HVX_Vector Q6_Vub_vround_VhVh_sat (HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.ub=vround(Vu.uh,Vv.uh):sat</code>	<code>HVX_Vector Q6_Vub_vround_VuhVuh_sat (HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.uh=vround(Vu.uw,Vv.uw):sat</code>	<code>HVX_Vector Q6_Vuh_vround_VuwVuw_sat (HVX_Vector Vu, HVX_Vector Vv)</code>
<code>Vd.uh=vround(Vu.w,Vv.w):sat</code>	<code>HVX_Vector Q6_Vuh_vround_VwVw_sat (HVX_Vector Vu, HVX_Vector Vv)</code>

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse		u5										d5				
0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.h=vround(Vu.w,Vv.w):sat
0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.uh=vround(Vu.w,Vv.w):sat
0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.b=vround(Vu.h,Vv.h):sat
0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.ub=vround(Vu.h,Vv.h):sat
0	0	0	1	1	1	1	1	1	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.ub=vround(Vu.uh,Vv.uh):sat
0	0	0	1	1	1	1	1	1	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.uh=vround(Vu.uw,Vv.uw):sat

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Vector rotate right word

Rotate right each element of Vu.w by the unsigned amount specified by bits 4:0 of the corresponding element of Vv.w. Place the result in respective elements of Vd.w

Syntax

```
Vd.uw=vrotr(Vu.uw,Vv.uw)
```

Behavior

```
for (i = 0; i < VELEM(32); i++) {
    Vd.uw[i] = ((Vu.uw[i] >> (Vv.uw[i] & 0x1f)) |
    (Vu.uw[i] << (32 - (Vv.uw[i] & 0x1f))));
}
```

Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction uses the HVX shift resource.

Intrinsics

```
Vd.uw=vrotr(Vu.uw,Vv.uw)  HVX_Vector Q6_Vuw_vrotr_VuwVuw(HVX_Vector Vu,
HVX_Vector Vv)
```

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse				u5						d5						
0	0	0	1	1	0	1	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.uw=vrotr(Vu.uw,Vv.uw)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Subtract - half precision vector by vector

Perform a vectorized half precision floating point subtract. The inputs are either both IEEE single precision, both 16-bit Qfloat, or one of each. The result is a 16-bit Qfloat vector.

Syntax	Behavior
<code>Vd.qf16=vsub(Vu.hf,Vv.hf)</code>	<pre> for (i = 0; i < VELEM(16); i++) { u = Vu.hf[i]; v = Vv.hf[i]; if (u.exp>v.exp) { exp = u.exp+((u.sig==0.0)? (-(FRAC_HF + 1)):ilogb(u.sig)); if (exp<v.exp) exp = v.exp; } else { exp = v.exp+((v.sig==0.0)? (-(FRAC_HF + 1)):ilogb(v.sig)); if (exp<u.exp) exp = u.exp; } sig_u = ldexp(u.sig, u.exp-exp); sig_v = ldexp(v.sig, v.exp-exp); if((u.sign==0) && (v.sign==0)) { sig = sig_u - sig_v; sig_low = (u.exp>v.exp) ? (sig_u-sig)- sig_v : (sig_u-(sig_v+sig)); } else if(u.sign ^ v.sign){ sig = sig_u + sig_v; sig_low = (u.exp>v.exp) ? (sig_u-sig) + sig_v : (sig_v-sig)+sig_u; } else{ sig = sig_v - sig_u; sig_low = (v.exp>u.exp) ? (sig_v-sig)- sig_u : sig_v-(sig_u+sig); } Vd.qf16[i] = rnd_sat(exp,sig,sig_low); if((u.sign==1) && (v.sign==0)) Vd.qf16[i] = - (Vd.qf16[i]); } </pre>
<code>Vd.qf16=vsub(Vu.qf16,Vv.hf)</code>	<pre> for (i = 0; i < VELEM(16); i++) { u = Vu.qf16[i]; v = Vv.hf[i]; if(v.sign) v.sig = (-1.0)*v.sig; if (u.exp>v.exp) { exp = u.exp+((u.sig==0.0)? (-(FRAC_HF+ 1)):ilogb(u.sig)); if (exp < v.exp) exp = v.exp; } else { exp = v.exp+((v.sig==0.0)? (-(FRAC_HF+ 1)):ilogb(v.sig)); if (exp < u.exp) exp = u.exp; } sig_u = ldexp(u.sig, u.exp-exp); sig_v = ldexp(v.sig, v.exp-exp); sig = sig_u - sig_v; sig_low = (u.exp>v.exp) ? (sig_u-sig)-sig_v : (sig_u - (sig_v + sig)); Vd.qf16[i] = rnd_sat(exp, sig, sig_low);} </pre>

Syntax	Behavior
Vd.qf16 = vsub(Vu.qf16, Vv.qf16)	<pre> for (i = 0; i < VELEM(16); i++) { u = Vu.qf16[i]; v = Vv.qf16[i]; if (u.exp>v.exp) { exp = u.exp+((u.sig==0.0)? -(FRAC_HF+1)):ilogb(u.sig)); if (exp<v.exp) exp = v.exp; } else { exp = v.exp+((v.sig==0.0)? -(FRAC_HF+1)):ilogb(v.sig)); if (exp<u.exp) exp = u.exp; } sig_u = ldexp(u.sig, u.exp-exp); sig_v = ldexp(v.sig, v.exp-exp); sig = sig_u - sig_v; sig_low = (u.exp>v.exp) ? (sig_u-sig)-sig_v : (sig_u-(sig_v+sig)); Vd.qf16[i] = rnd_sat(exp,sig,sig_low); } </pre>

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction uses the HVX shift resource.

Intrinsics

Vd.qf16=vsub(Vu.hf,Vv.hf)	HVX_Vector Q6_Vqf16_vsub_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf16=vsub(Vu.qf16,Vv.hf)	HVX_Vector Q6_Vqf16_vsub_Vqf16Vhf(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf16=vsub(Vu.qf16,Vv.qf16)	HVX_Vector Q6_Vqf16_vsub_Vqf16Vqf16(HVX_Vector Vu, HVX_Vector Vv)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse		u5										d5				
0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.qf16=vsub(Vu.qf16,Vv.qf16)
0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.qf16=vsub(Vu.hf,Vv.hf)
0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.qf16=vsub(Vu.qf16,Vv.hf)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Subtract - single precision vector by vector

These instructions perform a vectorized single precision floating point subtract. The inputs are either both IEEE single precision, both 32-bit Qfloat, or one of each. The result is a 32-bit Qfloat vector.

Syntax	Behavior
<code>Vd.qf32=vsub(Vu.qf32,Vv.qf32)</code>	<pre> for (i = 0; i < VELEM(32); i++) { u = Vu.qf32[i]; v = Vv.qf32[i]; if (u.exp>v.exp) { exp = u.exp+((u.sig==0.0)? (-(FRAC_SF+ 1)):ilogb(u.sig)); if (exp<v.exp) exp = v.exp; } else { exp = v.exp+((v.sig==0.0)? (-(FRAC_SF+ 1)):ilogb(v.sig)); if (exp < u.exp) exp = u.exp; } sig_u = ldexp(u.sig, u.exp-exp); sig_v = ldexp(v.sig, v.exp-exp); sig = sig_u - sig_v; sig_low = (u.exp > v.exp) ? (sig_u -sig) - sig_v : (sig_u-(sig_v + sig)); Vd.qf32[i] = rnd_sat(exp,sig,sig_low); } </pre>
<code>Vd.qf32=vsub(Vu.qf32,Vv.sf)</code>	<pre> for (i = 0; i < VELEM(32); i++) { u = Vu.qf32[i]; v = Vv.sf[i]; if(v.sign) v.sig = (-1.0)* v.sig; if (u.exp>v.exp) { exp = u.exp+((u.sig==0.0)? (-(FRAC_SF+ 1)):ilogb(u.sig)); if (exp<v.exp) exp = v.exp; } else { exp = v.exp+((v.sig==0.0)? (-(FRAC_SF+ 1)):ilogb(v.sig)); if (exp<u.exp) exp = u.exp; } sig_u = ldexp(u.sig, u.exp-exp); sig_v = ldexp(v.sig, v.exp-exp); sig = sig_u - sig_v; sig_low = (u.exp>v.exp) ? (sig_u-sig)-sig_v : (sig_u-(sig_v+sig)); Vd.qf32[i] = rnd_sat(exp, sig, sig_low); } </pre>
<code>Vd.qf32=vsub(Vu.sf,Vv.sf)</code>	<pre> for (i = 0; i < VELEM(32); i++) { u = Vu.sf[i]; v = Vv.sf[i]; if (u.exp>v.exp) { exp = u.exp +((u.sig==0.0)? (-(FRAC_SF+ 1)):ilogb(u.sig)); if (exp<v.exp) exp = v.exp; } else { exp = v.exp+((v.sig==0.0)? (-(FRAC_SF+ 1)):ilogb(v.sig)); } } </pre>

Syntax

```
Vd.qf32=vsub(Vu.sf,Vv.sf)
(cont.)
```

Behavior

```
if (exp<u.exp) exp = u.exp;
}
sig_u = ldexp(u.sig, u.exp-exp);
sig_v = ldexp(v.sig, v.exp-exp);
if((u.sign==0) && (v.sign==0)) {
    sig = sig_u - sig_v;
    sig_low = (u.exp>v.exp) ? (sig_u -sig) -
sig_v : (sig_u -(sig_v+sig));
} else if(u.sign ^ v.sign){
    sig = sig_u + sig_v;
    sig_low = (u.exp > v.exp) ? (sig_u-sig)+
sig_v : (sig_v-sig)+sig_u;
} else{
    sig = sig_v - sig_u;
    sig_low = (v.exp > u.exp) ? (sig_v - sig)-
sig_u : sig_v - (sig_u+sig);
}
Vd.qf32[i] = rnd_sat(exp,sig,sig_low);
if((u.sign==1) && (v.sign==0)) Vd.qf32[i] =
-(Vd.qf32[i]);}
```

Class: COPROC_VX (slots 0,1,2,3)**Notes**

- This instruction uses the HVX shift resource.

Intrinsics

Vd.qf32 = vsub(Vu.qf32,Vv.qf32)	HVX_Vector Q6_Vqf32_vsub_Vqf32Vqf32(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf32 = vsub(Vu.qf32,Vv.sf)	HVX_Vector Q6_Vqf32_vsub_Vqf32Vsf(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf32 = vsub(Vu.sf,Vv.sf)	HVX_Vector Q6_Vqf32_vsub_VsfVsf(HVX_Vector Vu, HVX_Vector Vv)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse		u5								d5						
0	0	0	1	1	1	1	1	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd.qf32=vsub(Vu.qf32,Vv.qf32)
0	0	0	1	1	1	1	1	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.qf32=vsub(Vu.sf,Vv.sf)
0	0	0	1	1	1	1	1	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.qf32=vsub(Vu.qf32,Vv.sf)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u
v5	Field to encode register v

Bit counting

The bit counting operations are applied to each vector element in a vector register Vu, and place the result in the corresponding element in the vector destination register Vd.

Count leading zeros (`vc10`) counts the number of consecutive zeros starting with the most significant bit. It supports unsigned halfword and word.

Population count (`vpopcount`) counts the number of nonzero bits in a halfword element.

Normalization amount (`vnormamt`) counts the number of bits for normalization (consecutive sign bits minus one, with zero treated specially). Count leading identical bits, and add a value to it for each lane

Syntax	Behavior
<code>Vd.h=vadd(vclb(Vu.h),Vv.h)</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.h[i] = max(count_leading_ones(~Vu.h[i]), count_leading_ones(Vu.h[i])) + Vv.h[i]; }</pre>
<code>Vd.h=vnormamt(Vu.h)</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.h[i]=max(count_leading_ones(~Vu.h[i]),count_ _leading_ones(Vu.h[i]))-1; }</pre>
<code>Vd.h=vpopcount(Vu.h)</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.uh[i]=count_ones(Vu.uh[i]); }</pre>
<code>Vd.uh=vc10(Vu.uh)</code>	<pre>for (i = 0; i < VELEM(16); i++) { Vd.uh[i]=count_leading_ones(~Vu.uh[i]); }</pre>
<code>Vd.uw=vc10(Vu.uw)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.uw[i]=count_leading_ones(~Vu.uw[i]); }</pre>
<code>Vd.w=vadd(vclb(Vu.w),Vv.w)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i] = max(count_leading_ones(~Vu.w[i]), count_leading_ones(Vu.w[i])) + Vv.w[i] ; }</pre>
<code>Vd.w=vnormamt(Vu.w)</code>	<pre>for (i = 0; i < VELEM(32); i++) { Vd.w[i]=max(count_leading_ones(~Vu.w[i]),count_ _leading_ones(Vu.w[i]))-1; }</pre>

Class: COPROC_VX (slots 0,1,2,3)

Notes

- This instruction uses the HVX shift resource.

Intrinsics

Vd.h=vadd(vclb(Vu.h),Vv.h)	HVX_Vector Q6_Vh_vadd_vclb_VhVh(HVX_Vector Vu, HVX_Vector Vv)
Vd.h=vnormamt(Vu.h)	HVX_Vector Q6_Vh_vnormamt_Vh(HVX_Vector Vu)
Vd.h=vpopcount(Vu.h)	HVX_Vector Q6_Vh_vpopcount_Vh(HVX_Vector Vu)
Vd.uh=vcl0(Vu.uh)	HVX_Vector Q6_Vuh_vcl0_Vuh(HVX_Vector Vu)
Vd.uw=vcl0(Vu.uw)	HVX_Vector Q6_Vuw_vcl0_Vuw(HVX_Vector Vu)
Vd.w=vadd(vclb(Vu.w),Vv.w)	HVX_Vector Q6_Vw_vadd_vclb_VwVw(HVX_Vector Vu, HVX_Vector Vv)
Vd.w=vnormamt(Vu.w)	HVX_Vector Q6_Vw_vnormamt_Vw(HVX_Vector Vu)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS															Parse		u5										d5					
0	0	0	1	1	1	1	0	-	-	0	-	-	-	1	0	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.uw=vcl0(Vu.uw)
0	0	0	1	1	1	1	0	-	-	0	-	-	-	1	0	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.h=vpopcount(Vu.h)
0	0	0	1	1	1	1	0	-	-	0	-	-	-	1	0	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d	Vd.uh=vcl0(Vu.uh)
0	0	0	1	1	1	1	0	-	-	0	-	-	-	1	1	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vd.w=vnormamt(Vu.w)
0	0	0	1	1	1	1	0	-	-	0	-	-	-	1	1	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vd.h=vnormamt(Vu.h)
ICLASS															Parse		u5										d5					
0	0	0	1	1	1	1	1	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.h=vadd(vclb(Vu.h),Vv.h)
0	0	0	1	1	1	1	1	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.w=vadd(vclb(Vu.w),Vv.w)

Field name	Description
ICLASS	Instruction class
Parse	Packet loop parse bits
d5	Field to encode register d
u5	Field to encode register u

6.15 HVX store

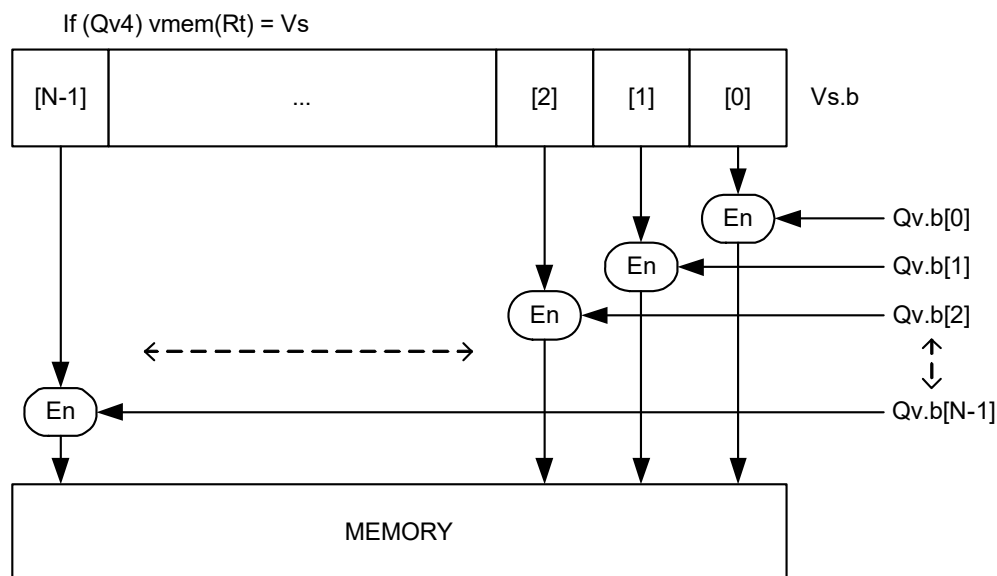
The HVX store instruction subclass includes memory store instructions.

Store - byte-enabled aligned

Of the bytes in vector register *Vs*, store to memory only the bytes where the corresponding bit in the predicate register *Qv* is enabled. The block of memory to store into is at a vector-size-aligned address. The operation has three ways to generate the memory pointer address: *Rt* with a constant 4-bit signed offset, *Rx* with a signed post-increment, and *Rx* with a modifier register *Mu* post-increment. For the immediate forms, the value indicates the number of vectors worth of data. *Mu* contains the actual byte offset.

If all bits in *Qv* are set to zero, no data is stored to memory, but the post-increment of the pointer in *Rt* occurs.

If the pointer presented to the instruction is not aligned, the instruction ignores the lower bits, yielding an aligned address.



Syntax	Behavior
<code>if (!<i>Qv</i>4) <i>vmem</i>(<i>Rt</i>):nt=<i>Vs</i></code>	Assembler mapped to: " <code>if (!<i>Qv</i>4) <i>vmem</i>(<i>Rt</i>+#0):nt=<i>Vs</i>"</code>
<code>if (!<i>Qv</i>4) <i>vmem</i>(<i>Rt</i>)=<i>Vs</i></code>	Assembler mapped to: " <code>if (!<i>Qv</i>4) <i>vmem</i>(<i>Rt</i>+#0)=<i>Vs</i>"</code>
<code>if (!<i>Qv</i>4) <i>vmem</i>(<i>Rt</i>+#<i>s</i>4):nt=<i>Vs</i></code>	<i>EA</i> = <i>Rt</i> +# <i>s</i> *VBYTES; *(<i>EA</i> &~(ALIGNMENT-1)) = <i>Vs</i> ;
<code>if (!<i>Qv</i>4) <i>vmem</i>(<i>Rt</i>+#<i>s</i>4)=<i>Vs</i></code>	<i>EA</i> = <i>Rt</i> +# <i>s</i> *VBYTES; *(<i>EA</i> &~(ALIGNMENT-1)) = <i>Vs</i> ;
<code>if (!<i>Qv</i>4) <i>vmem</i>(<i>Rx</i>++#<i>s</i>3):nt=<i>Vs</i></code>	<i>EA</i> = <i>Rx</i> ; *(<i>EA</i> &~(ALIGNMENT-1)) = <i>Vs</i> ; <i>Rx</i> = <i>Rx</i> +# <i>s</i> *VBYTES;

Syntax	Behavior
if (!Qv4) vmem(Rx++#s3)=Vs	EA=Rx; *(EA&~(ALIGNMENT-1)) = Vs; Rx=Rx+#s*VBYTES;
if (!Qv4) vmem(Rx++Mu):nt=Vs	EA=Rx; *(EA&~(ALIGNMENT-1)) = Vs; Rx=Rx+MuV;
if (!Qv4) vmem(Rx++Mu)=Vs	EA=Rx; *(EA&~(ALIGNMENT-1)) = Vs; Rx=Rx+MuV;

Class: COPROC_VMEM (slots 0)**Notes**

- This instruction can use any HVX resource.
- An optional non-temporal hint to the microarchitecture can be specified to indicate the data has no reuse.
- Immediates used in address computation are specified in multiples of vector length.

Intrinsics

if (!Qv4) vmem(Rt+#s4):nt=Vs	void Q6_vmem_QnRIV_nt(HVX_VectorPred Qv, HVX_Vector* A, HVX_Vector Vs)
if (!Qv4) vmem(Rt+#s4)=Vs	void Q6_vmem_QnRIV(HVX_VectorPred Qv, HVX_Vector* A, HVX_Vector Vs)
if (Qv4) vmem(Rt+#s4):nt=Vs	void Q6_vmem_QRIV_nt(HVX_VectorPred Qv, HVX_Vector* A, HVX_Vector Vs)
if (Qv4) vmem(Rt+#s4)=Vs	void Q6_vmem_QRIV(HVX_VectorPred Qv, HVX_Vector* A, HVX_Vector Vs)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS										NT		t5				Parse										s5						
0	0	1	0	1	0	0	0	1	0	0	t	t	t	t	t	P	P	i	v	v	i	i	i	0	0	0	s	s	s	s	s	if (Qv4) vmem(Rt+#s4)=Vs
0	0	1	0	1	0	0	0	1	0	0	t	t	t	t	t	P	P	i	v	v	i	i	i	0	0	1	s	s	s	s	s	if (!Qv4) vmem(Rt+#s4)=Vs
0	0	1	0	1	0	0	0	1	1	0	t	t	t	t	t	P	P	i	v	v	i	i	i	0	0	0	s	s	s	s	s	if (Qv4) vmem(Rt+#s4):nt=Vs
0	0	1	0	1	0	0	0	1	1	0	t	t	t	t	t	P	P	i	v	v	i	i	i	0	0	1	s	s	s	s	s	if (!Qv4) vmem(Rt+#s4):nt=Vs
ICLASS										NT		x5				Parse										s5						
0	0	1	0	1	0	0	1	1	0	0	x	x	x	x	x	P	P	-	v	v	i	i	i	0	0	0	s	s	s	s	s	if (Qv4) vmem(Rx++#s3)=Vs
0	0	1	0	1	0	0	1	1	0	0	x	x	x	x	x	P	P	-	v	v	i	i	i	0	0	1	s	s	s	s	s	if (!Qv4) vmem(Rx++#s3)=Vs
0	0	1	0	1	0	0	1	1	1	0	x	x	x	x	x	P	P	-	v	v	i	i	i	0	0	0	s	s	s	s	s	if (Qv4) vmem(Rx++#s3):nt=Vs

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	1	0	0	1	1	1	0	x	x	x	x	x	P	P	-	v	v	i	i	i	0	0	1	s	s	s	s	s	if (!Qv4) vmem(Rx++#s3):nt=Vs
ICLASS					NT					x5					Parse		u1												s5			
0	0	1	0	1	0	1	1	1	0	0	x	x	x	x	x	P	P	u	v	v	-	-	-	0	0	0	s	s	s	s	s	if (Qv4) vmem(Rx++Mu)=Vs
0	0	1	0	1	0	1	1	1	0	0	x	x	x	x	x	P	P	u	v	v	-	-	-	0	0	1	s	s	s	s	s	if (!Qv4) vmem(Rx++Mu)=Vs
0	0	1	0	1	0	1	1	1	1	0	x	x	x	x	x	P	P	u	v	v	-	-	-	0	0	0	s	s	s	s	s	if (Qv4) vmem(Rx++Mu):nt=Vs
0	0	1	0	1	0	1	1	1	1	0	x	x	x	x	x	P	P	u	v	v	-	-	-	0	0	1	s	s	s	s	s	if (!Qv4) vmem(Rx++Mu):nt=Vs

Field name	Description
ICLASS	Instruction class
NT	Nontemporal
Parse	Packet loop parse bits
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x

Store - new

Store the result of an operation in the current packet to memory, using a vector-aligned address. The result is also written to the vector register file at the vector register location.

For example, in the instruction `vmem(R8++#1) = V12.new`, the value in V12 in this packet is written to memory, and V12 is also written to the vector register file.

The operation has three ways to generate the memory pointer address: Rt with a constant 4-bit signed offset, Rx with a 3-bit signed post-increment, and Rx with a modifier register Mu post-increment. For the immediate forms, the value indicates the number of vectors worth of data. Mu contains the actual byte offset.

The store is conditional, based on the value of the scalar predicate register Pv. If the condition evaluates false, the operation becomes a NOP.

Syntax	Behavior
<code>if (!Pv) vmem(Rt+#s4):nt=Os8.new</code>	<pre>if (!Pv[0]) { EA=Rt+#s*VBYTES; *(EA&~(ALIGNMENT-1)) = OsN.new; } else { NOP; }</pre>
<code>if (!Pv) vmem(Rt+#s4)=Os8.new</code>	<pre>if (!Pv[0]) { EA=Rt+#s*VBYTES; *(EA&~(ALIGNMENT-1)) = OsN.new; } else { NOP; }</pre>
<code>if (!Pv) vmem(Rx++#s3):nt=Os8.new</code>	<pre>if (!Pv[0]) { EA=Rx; *(EA&~(ALIGNMENT-1)) = OsN.new; Rx=Rx+#s*VBYTES; } else { NOP; }</pre>
<code>if (!Pv) vmem(Rx++#s3)=Os8.new</code>	<pre>if (!Pv[0]) { EA=Rx; *(EA&~(ALIGNMENT-1)) = OsN.new; Rx=Rx+#s*VBYTES; } else { NOP; }</pre>
<code>if (!Pv) vmem(Rx++Mu):nt=Os8.new</code>	<pre>if (!Pv[0]) { EA=Rx; *(EA&~(ALIGNMENT-1)) = OsN.new; Rx=Rx+MuV; } else { NOP; }</pre>

Syntax	Behavior
if (!Pv) vmem(Rx++Mu)=Os8.new	if (!Pv[0]) { EA=Rx; *(EA&~(ALIGNMENT-1)) = OsN.new; Rx=Rx+MuV; } else { NOP; }
vmem(Rt):nt=Os8.new	Assembler mapped to: "vmem(Rt+#0):nt=Os8.new"
vmem(Rt)=Os8.new	Assembler mapped to: "vmem(Rt+#0)=Os8.new"
vmem(Rt+#s4):nt=Os8.new	EA=Rt+#s*VBYTES; *(EA&~(ALIGNMENT-1)) = OsN.new;
vmem(Rt+#s4)=Os8.new	EA=Rt+#s*VBYTES; *(EA&~(ALIGNMENT-1)) = OsN.new;
vmem(Rx++#s3):nt=Os8.new	EA=Rx; *(EA&~(ALIGNMENT-1)) = OsN.new; Rx=Rx+#s*VBYTES;
vmem(Rx++#s3)=Os8.new	EA=Rx; *(EA&~(ALIGNMENT-1)) = OsN.new; Rx=Rx+#s*VBYTES;
vmem(Rx++Mu):nt=Os8.new	EA=Rx; *(EA&~(ALIGNMENT-1)) = OsN.new; Rx=Rx+MuV;
vmem(Rx++Mu)=Os8.new	EA=Rx; *(EA&~(ALIGNMENT-1)) = OsN.new; Rx=Rx+MuV;

Class: COPROC_VMEM (slots 0)**Notes**

- This instruction can use any HVX resource.
- An optional non-temporal hint to the microarchitecture can be specified to indicate the data has no reuse.
- Immediates used in address computation are specified in multiples of vector length.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS									NT		t5					Parse													s3			
0	0	1	0	1	0	0	0	0	0	1	t	t	t	t	t	P	P	i	-	-	i	i	i	0	0	1	-	0	s	s	s	vmem(Rt+#s4)=Os8.new
0	0	1	0	1	0	0	0	0	1	1	t	t	t	t	t	P	P	i	-	-	i	i	i	0	0	1	-	-	s	s	s	vmem(Rt+#s4):nt=Os8.new
0	0	1	0	1	0	0	0	1	0	1	t	t	t	t	t	P	P	i	v	v	i	i	i	0	1	0	0	0	s	s	s	if (Pv) vmem(Rt+#s4)=Os8.new
0	0	1	0	1	0	0	0	1	0	1	t	t	t	t	t	P	P	i	v	v	i	i	i	0	1	1	0	1	s	s	s	if (!Pv) vmem(Rt+#s4)=Os8.new
0	0	1	0	1	0	0	0	1	1	1	t	t	t	t	t	P	P	i	v	v	i	i	i	0	1	0	1	0	s	s	s	if (Pv) vmem(Rt+#s4):nt=Os8.new

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	1	0	0	0	1	1	1	t	t	t	t	t	P	P	i	v	v	i	i	i	0	1	1	1	1	s	s	s	if (!Pv) vmem(Rt++s4):nt=Os8.new
ICLASS								NT		x5				Parse												s3						
0	0	1	0	1	0	0	1	0	0	1	x	x	x	x	x	P	P	-	-	-	i	i	i	0	0	1	-	0	s	s	s	vmem(Rx++s3)=Os8.new
0	0	1	0	1	0	0	1	0	1	1	x	x	x	x	x	P	P	-	-	-	i	i	i	0	0	1	-	-	s	s	s	vmem(Rx++s3):nt=Os8.new
0	0	1	0	1	0	0	1	1	0	1	x	x	x	x	x	P	P	-	v	v	i	i	i	0	1	0	0	0	s	s	s	if (Pv) vmem(Rx++s3)=Os8.new
0	0	1	0	1	0	0	1	1	0	1	x	x	x	x	x	P	P	-	v	v	i	i	i	0	1	1	0	1	s	s	s	if (!Pv) vmem(Rx++s3)=Os8.new
0	0	1	0	1	0	0	1	1	1	1	x	x	x	x	x	P	P	-	v	v	i	i	i	0	1	0	1	0	s	s	s	if (Pv) vmem(Rx++s3):nt=Os8.new
0	0	1	0	1	0	0	1	1	1	1	x	x	x	x	x	P	P	-	v	v	i	i	i	0	1	1	1	1	s	s	s	if (!Pv) vmem(Rx++s3):nt=Os8.new
ICLASS								NT		x5				Parse				u1								s3						
0	0	1	0	1	0	1	1	0	0	1	x	x	x	x	x	P	P	u	-	-	-	-	-	0	0	1	-	0	s	s	s	vmem(Rx++Mu)=Os8.new
0	0	1	0	1	0	1	1	0	1	1	x	x	x	x	x	P	P	u	-	-	-	-	-	0	0	1	-	-	s	s	s	vmem(Rx++Mu):nt=Os8.new
0	0	1	0	1	0	1	1	1	0	1	x	x	x	x	x	P	P	u	v	v	-	-	-	0	1	0	0	0	s	s	s	if (Pv) vmem(Rx++Mu)=Os8.new
0	0	1	0	1	0	1	1	1	0	1	x	x	x	x	x	P	P	u	v	v	-	-	-	0	1	1	0	1	s	s	s	if (!Pv) vmem(Rx++Mu)=Os8.new
0	0	1	0	1	0	1	1	1	1	1	x	x	x	x	x	P	P	u	v	v	-	-	-	0	1	0	1	0	s	s	s	if (Pv) vmem(Rx++Mu):nt=Os8.new
0	0	1	0	1	0	1	1	1	1	1	x	x	x	x	x	P	P	u	v	v	-	-	-	0	1	1	1	1	s	s	s	if (!Pv) vmem(Rx++Mu):nt=Os8.new

Field name	Description
ICLASS	Instruction class
NT	Nontemporal
Parse	Packet loop parse bits
s3	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x

Store - aligned

Write a full vector register Vs to memory, using a vector-size-aligned address. The operation has three ways to generate the memory pointer address: Rt with a constant 4-bit signed offset, Rx with a signed post-increment, and Rx with a modifier register Mu post-increment. For the immediate forms, the value indicates the number of vectors worth of data. Mu contains the actual byte offset.

If the pointer presented to the instruction is not aligned, the instruction ignores the lower bits, yielding an aligned address.

If a scalar predicate register Pv evaluates true, store a full vector register Vs to memory, using a vector-size-aligned address. Otherwise, the operation becomes a NOP

Syntax	Behavior
<code>if (!Pv) vmem(Rt):nt=Vs</code>	Assembler mapped to: <code>"if (!Pv) vmem(Rt+#0):nt=Vs"</code>
<code>if (!Pv) vmem(Rt)=Vs</code>	Assembler mapped to: <code>"if (!Pv) vmem(Rt+#0)=Vs"</code>
<code>if (!Pv) vmem(Rt+#s4):nt=Vs</code>	<pre>if (!Pv[0]) { EA=Rt+#s*VBYTES; *(EA&~(ALIGNMENT-1)) = Vs; } else { NOP; }</pre>
<code>if (!Pv) vmem(Rt+#s4)=Vs</code>	<pre>if (!Pv[0]) { EA=Rt+#s*VBYTES; *(EA&~(ALIGNMENT-1)) = Vs; } else { NOP; }</pre>
<code>if (!Pv) vmem(Rx++#s3):nt=Vs</code>	<pre>if (!Pv[0]) { EA=Rx; *(EA&~(ALIGNMENT-1)) = Vs; Rx=Rx+#s*VBYTES; } else { NOP; }</pre>
<code>if (!Pv) vmem(Rx++#s3)=Vs</code>	<pre>if (!Pv[0]) { EA=Rx; *(EA&~(ALIGNMENT-1)) = Vs; Rx=Rx+#s*VBYTES; } else { NOP; }</pre>
<code>if (!Pv) vmem(Rx++Mu):nt=Vs</code>	<pre>if (!Pv[0]) { EA=Rx; *(EA&~(ALIGNMENT-1)) = Vs; Rx=Rx+MuV; } else { NOP; }</pre>

Syntax	Behavior
<code>if (!Pv) vmem(Rx++Mu)=Vs</code>	<pre> if (!Pv[0]) { EA=Rx; *(EA&~(ALIGNMENT-1)) = Vs; Rx=Rx+MuV; } else { NOP; } </pre>
<code>vmem(Rt):nt=Vs</code>	Assembler mapped to: "vmem(Rt+#0):nt=Vs"
<code>vmem(Rt)=Vs</code>	Assembler mapped to: "vmem(Rt+#0)=Vs"
<code>vmem(Rt+#s10:6)=Vs</code>	<pre> EA=Rt+#s; L2S[EA&~(128-1)] = Vs; </pre>
<code>vmem(Rt+#s4):nt=Vs</code>	<pre> EA=Rt+#s*VBYTES; *(EA&~(ALIGNMENT-1)) = Vs; </pre>
<code>vmem(Rt+#s4)=Vs</code>	<pre> EA=Rt+#s*VBYTES; *(EA&~(ALIGNMENT-1)) = Vs; </pre>
<code>vmem(Rx++#s10:6)=Vs</code>	<pre> EA=Rx; L2S[EA&~(128-1)] = Vs; Rx=Rx+#s; </pre>
<code>vmem(Rx++#s3):nt=Vs</code>	<pre> EA=Rx; *(EA&~(ALIGNMENT-1)) = Vs; Rx=Rx+#s*VBYTES; </pre>
<code>vmem(Rx++#s3)=Vs</code>	<pre> EA=Rx; *(EA&~(ALIGNMENT-1)) = Vs; Rx=Rx+#s*VBYTES; </pre>
<code>vmem(Rx++I:circ(Mu))=Vs</code>	<pre> EA=Rx; L2S[EA&~(128-1)] = Vs; Rx=Rx=circ_add(Rx,I,MuV); </pre>
<code>vmem(Rx++I:circ(Mu),Rt)=Vs</code>	<pre> EA=Rx; int bytecount =0; int aligned_store =0; int write_mask_en =0; int write_mask =0; if ((Rt == 0x80000000)) { bytecount = 128; VTMP = Vs; } else if ((Rt == 0x80000400)) { bytecount = 128; VTMP = Vs; VTMP=SWAP_IQ(VTMP); } else if ((Rt == 0x80000600)) { bytecount = 128; VTMP = Vs; VTMP=SWAP_IQ(VTMP); } else if ((Rt == 0x80000001)) { bytecount = 40; VTMP = PACK_TO_PCFL_10_4(Vs); } else if ((Rt == 0x80000002)) { bytecount = 48; </pre>

Syntax	Behavior
<pre>vmem(Rx++I:circ(Mu), Rt)=Vs (cont.)</pre>	<pre>VTMP = PACK TO PCFL 12_4(Vs); } else if ((Rt == 0x80000003)) { bytecount = 64; VTMP = PACK TO PCFL 16_4(Vs); } else if ((Rt == 0x80000004)) { bytecount = 80; VTMP = PACK TO PCFL 20_4(Vs); } else if ((Rt == 0x80000005)) { bytecount = 96; VTMP = PACK TO PCFL 24_4(Vs); } else if ((Rt == 0x8000000B)) { bytecount = 64; VTMP = PACK TO PCFL 16_2(Vs); } else if ((Rt == 0x80000010)) { bytecount = 32; VTMP = PACK TO UFL 8_5 (Vs); } else if ((Rt == 0x80000011)) { bytecount = 40; VTMP = PACK TO UFL 10_5 (Vs); } else if ((Rt == 0x80000012)) { bytecount = 48; VTMP = PACK TO UFL 12_5 (Vs); } else if ((Rt == 0x80000013)) { bytecount = 64; VTMP = PACK TO UFL 16_5 (Vs); } else if ((Rt == 0x8000001b)) { bytecount = 64; VTMP = PACK TO SFL 16_5 (Vs); } else if (((Rt & ~(7 <<12)) == 0x8000002c)) { bytecount = 80; VTMP = PACK TO PCFL20_H (Vs); Rx=Rx=circ_add(Rx,I,MuV); } else if (((Rt & ~(3 <<12)) == 0x8000002d)) { bytecount = 96; VTMP = PACK TO PCFL24_H (Vs); } else if ((Rt == 0x80000030)) { bytecount = 96; VTMP = PACK TO LLR6 (Vs); } else if ((Rt & 0xFFFF0000) == 0x80010000)) { bytecount = 16; write_mask_en = 1; VTMP = Vs; write_mask = Rt & 0xFFFF; } else if (((Rt & 0xFFFF0000) == 0x80020000)) { bytecount = 64; write_mask_en = 1; VTMP = Vs; write_mask = Rt & 0xFFFF; } else if (((Rt & 0xFFFF0000) == 0x80030000)) { bytecount = 128; write_mask_en = 1; VTMP = Vs; write_mask = Rt & 0xFFFF;</pre>

Syntax	Behavior
<code>vmem(Rx++I:circ(Mu), Rt)=Vs (cont.)</code>	<pre> } else { bytecount = 128; VTMP = Vs; aligned_store = 1; }; if (aligned_store) { } else if (write_mask_en) { L2S[EA] = VTMP; } else { { int k; int k2; unsigned int sh=0; L2S[EA&~(128-1)] = VTMP; if(Rt>>31) sh = ((EA & get_vStw_mask(thread - > processor_ptr -> arch_proc_options -> QDSP6_SLVR_VER) & 0x7f)); for(k=0; k<128-sh; k++) V0.V8u[sh+k] = VTMP.V8u[k]; for(k2=0; k2<sh; k2++,k++) V128.V8u[k2] = VTMP.V8u[k]; }; foreach(byteno) if (((byteno)>=(EA&(128-1))) & ((byteno) < (EA&(128-1)+bytecount))) { L2S[EA&~(128-1)+byteno] = Vmemptr[byteno]; bytecountleft--; foreach(byteno) if (((byteno)<(bytecountleft))) L2S[(EA+128)&~(128-1)+byteno] = Vmemptrplus128[byteno]; }; }; </pre>
<code>vmem(Rx++Mu):nt=Vs</code>	<pre> EA=Rx; *(EA&~(ALIGNMENT-1)) = Vs; Rx=Rx+MuV; </pre>
<code>vmem(Rx++Mu)=Vs</code>	<pre> EA=Rx; L2S[EA&~(128-1)] = Vs; Rx=Rx+MuV; </pre>
<code>vmem(Rx++Mu)=Vs</code>	<pre> EA=Rx; *(EA&~(ALIGNMENT-1)) = Vs; Rx=Rx+MuV; </pre>
<code>vmem(Rx++Mu, Rt)=Vs</code>	<pre> EA=Rx; int bytecount =0; int aligned_store =0; int write_mask_en =0; int write_mask =0; if ((Rt == 0x80000000)) { bytecount = 128; VTMP = Vs; } else if ((Rt == 0x80000400)) { bytecount = 128; </pre>

Syntax	Behavior
vmem (Rx++Mu, Rt)=Vs (cont.)	<pre> VTMP = Vs; VTMP=SWAP_IQ(VTMP); } else if ((Rt == 0x80000600)) { bytecount = 128; VTMP = Vs; VTMP=SWAP_IQ(VTMP); } else if ((Rt == 0x80000001)) { bytecount = 40; VTMP = PACK TO PCFL 10_4(Vs); } else if ((Rt == 0x80000002)) { bytecount = 48; VTMP = PACK TO PCFL 12_4(Vs); } else if ((Rt == 0x80000003)){ bytecount = 64; VTMP = PACK TO PCFL 16_4(Vs); } else if ((Rt == 0x80000004)){ bytecount = 80; VTMP = PACK TO PCFL 20_4(Vs); } else if ((Rt == 0x80000005)){ bytecount = 96; VTMP = PACK TO PCFL 24_4(Vs); } else if ((Rt == 0x8000000B)){ bytecount = 64; VTMP = PACK TO PCFL 16_2(Vs); } else if((Rt == 0x80000010)){ bytecount = 32; VTMP = PACK TO UFL 8_5 (Vs); } else if((Rt == 0x80000011)){ bytecount = 40; VTMP = PACK TO UFL 10_5 (Vs); } else if((Rt == 0x80000012)){ bytecount = 48; VTMP = PACK TO UFL 12_5 (Vs); } else if ((Rt == 0x80000013)){ bytecount = 64; VTMP = PACK TO UFL 16_5 (Vs); } else if ((Rt == 0x8000001b)){ bytecount = 64; VTMP = PACK TO SFL 16_5 (Vs); } else if (((Rt & ~(7 <<12)) == 0x8000002c)) { bytecount = 80; VTMP = PACK TO PCFL20_H (Vs); } else if (((Rt & ~(3 <<12)) == 0x8000002d)){ bytecount = 96; VTMP = PACK TO PCFL24_H (Vs); } else if ((Rt == 0x80000030)) { bytecount = 96; VTMP = PACK TO LLR6 (Vs); } else if (((Rt & 0xFFFF0000) == 0x80010000)) { bytecount = 16; write_mask_en = 1; VTMP = Vs; write_mask = Rt & 0xFFFF; } else if (((Rt & 0xFFFF0000) == 0x80020000)) { bytecount = 64; write_mask_en = 1; VTMP = Vs; </pre>

Syntax	Behavior
<pre>vmem (Rx++Mu, Rt) = Vs (cont.)</pre>	<pre> write_mask = Rt & 0xFFFF; } else if (((Rt & 0xFFFF0000) == 0x80030000)) { bytecount = 128; write_mask_en = 1; VTMP = Vs; write_mask = Rt & 0xFFFF; } else { bytecount = 128; VTMP = Vs; aligned_store = 1; }; if (aligned_store) { L2S[EA&~(128-1)] = VTMP; } else if (write_mask_en) { L2S[EA] = VTMP; } else { { int k; int k2; unsigned int sh=0; if(Rt>>31) sh = ((EA & get_vStw_mask(thread -> processor_ptr -> arch_proc_options -> QDSP6_SLVR_VER) & 0x7f)); for(k=0; k<128-sh; k++) V0.V8u[sh+k] = VTMP.V8u[k]; for(k2=0; k2<sh; k2++, k++) V128.V8u[k2] = VTMP.V8u[k]; }; foreach(byteno) if (((byteno)>=(EA&(128 -1))) & ((byteno) < (EA&(128 - 1) + bytecount))) { L2S[EA&~(128 - 1) + byteno] = Vmemptr[byteno]; bytecountleft--; foreach(byteno) if (((byteno)<(bytecountleft))) L2S[(EA + 128)&~(128-1) + byteno] = Vmemptrplus128[byteno];};} Rx=Rx+MuV; </pre>

Class: COPROC_VMEM (slots 0)**Notes**

- This instruction can use any HVX resource.
- An optional non-temporal hint to the microarchitecture can be specified to indicate the data has no reuse.
- Immediates used in address computation are specified in multiples of vector length.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0											
ICLASS													NT				t5					Parse					s5															
0	0	1	0	1	0	0	0	0	0	1	1	t	t	t	t	t	P	P	i	-	-	i	i	i	0	0	0	s	s	s	s	s	vmem(Rt+#s4)=Vs									
0	0	1	0	1	0	0	0	0	0	1	1	t	t	t	t	t	P	P	i	-	-	i	i	i	0	0	0	s	s	s	s	s	vmem(Rt+#s4):nt=Vs									
0	0	1	0	1	0	0	0	0	1	0	1	t	t	t	t	t	P	P	i	v	v	i	i	i	0	0	0	s	s	s	s	s	if (Pv) vmem(Rt+#s4)=Vs									
0	0	1	0	1	0	0	0	0	1	0	1	t	t	t	t	t	P	P	i	v	v	i	i	i	0	0	1	s	s	s	s	s	if (!Pv) vmem(Rt+#s4)=Vs									
0	0	1	0	1	0	0	0	0	1	1	1	t	t	t	t	t	P	P	i	v	v	i	i	i	0	0	0	s	s	s	s	s	if (Pv) vmem(Rt+#s4):nt=Vs									
0	0	1	0	1	0	0	0	0	1	1	1	t	t	t	t	t	P	P	i	v	v	i	i	i	0	0	1	s	s	s	s	s	if (!Pv) vmem(Rt+#s4):nt=Vs									
ICLASS													NT				x5					Parse					s5															
0	0	1	0	1	0	0	0	1	0	0	1	x	x	x	x	x	P	P	-	-	-	i	i	i	0	0	0	s	s	s	s	s	vmem(Rx++#s3)=Vs									
0	0	1	0	1	0	0	0	1	0	1	1	x	x	x	x	x	P	P	-	-	-	i	i	i	0	0	0	s	s	s	s	s	vmem(Rx++#s3):nt=Vs									
0	0	1	0	1	0	0	0	1	1	0	1	x	x	x	x	x	P	P	-	v	v	i	i	i	0	0	0	s	s	s	s	s	if (Pv) vmem(Rx++#s3)=Vs									
0	0	1	0	1	0	0	0	1	1	0	1	x	x	x	x	x	P	P	-	v	v	i	i	i	0	0	1	s	s	s	s	s	if (!Pv) vmem(Rx++#s3)=Vs									
0	0	1	0	1	0	0	0	1	1	1	1	x	x	x	x	x	P	P	-	v	v	i	i	i	0	0	0	s	s	s	s	s	if (Pv) vmem(Rx++#s3):nt=Vs									
0	0	1	0	1	0	0	0	1	1	1	1	x	x	x	x	x	P	P	-	v	v	i	i	i	0	0	1	s	s	s	s	s	if (!Pv) vmem(Rx++#s3):nt=Vs									
ICLASS													NT				x5					Parse					u1					s5										
0	0	1	0	1	0	1	0	1	0	0	0	x	x	x	x	x	P	P	u	-	-	-	-	-	s	s	s	s	s	0	-	0	vmem(Rx++l:circ(Mu))=Vs									
ICLASS													NT				x5					Parse					u1					t5					s5					
0	0	1	0	1	0	1	0	1	0	0	1	x	x	x	x	x	P	P	u	t	t	t	t	t	s	s	s	s	s	-	-	-	vmem(Rx++l:circ(Mu),Rt)=Vs									
ICLASS													NT				x5					Parse					u1					s5										
0	0	1	0	1	0	1	0	1	1	0	0	1	x	x	x	x	x	P	P	u	-	-	-	-	-	0	0	0	s	s	s	s	s	vmem(Rx++Mu)=Vs								
0	0	1	0	1	0	1	0	1	1	0	1	1	x	x	x	x	x	P	P	u	-	-	-	-	-	0	0	0	s	s	s	s	s	vmem(Rx++Mu):nt=Vs								
ICLASS													NT				x5					Parse					u1					s5										
0	0	1	0	1	0	1	0	1	1	0	0	x	x	x	x	x	P	P	u	-	-	-	-	-	s	s	s	s	s	0	-	0	vmem(Rx++Mu)=Vs									
ICLASS													NT				x5					Parse					u1					t5					s5					
0	0	1	0	1	0	1	0	1	1	0	1	x	x	x	x	x	P	P	u	t	t	t	t	t	s	s	s	s	s	-	-	-	vmem(Rx++Mu,Rt)=Vs									
ICLASS													NT				x5					Parse					u1					s5										
0	0	1	0	1	0	1	0	1	1	0	1	x	x	x	x	x	P	P	u	v	v	-	-	-	0	0	0	s	s	s	s	s	if (Pv) vmem(Rx++Mu)=Vs									
0	0	1	0	1	0	1	0	1	1	0	1	x	x	x	x	x	P	P	u	v	v	-	-	-	0	0	1	s	s	s	s	s	if (!Pv) vmem(Rx++Mu)=Vs									
0	0	1	0	1	0	1	0	1	1	1	1	x	x	x	x	x	P	P	u	v	v	-	-	-	0	0	0	s	s	s	s	s	if (Pv) vmem(Rx++Mu):nt=Vs									
0	0	1	0	1	0	1	0	1	1	1	1	x	x	x	x	x	P	P	u	v	v	-	-	-	0	0	1	s	s	s	s	s	if (!Pv) vmem(Rx++Mu):nt=Vs									
ICLASS													NT				x5					Parse					s5															
0	0	1	0	1	1	0	0	1	0	i	x	x	x	x	x	P	P	i	i	i	i	i	i	i	s	s	s	s	s	i	i	i	vmem(Rx++#s10:6)=Vs									
ICLASS													NT				t5					Parse					s5															
0	0	1	0	1	1	1	1	1	1	1	i	t	t	t	t	t	P	P	i	i	i	i	i	i	s	s	s	s	s	i	i	i	vmem(Rt+#s10:6)=Vs									

Field name	Description
ICLASS	Instruction class
NT	Nontemporal
Parse	Packet loop parse bits
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x

Store - unaligned

Write a full vector register Vs to memory, using an arbitrary byte-aligned address. The operation has three ways to generate the memory pointer address: Rt with a constant 4-bit signed offset, Rx with a 3-bit signed post-increment, and Rx with a modifier register Mu post-increment. For the immediate forms, the value indicates the number of vectors worth of data. Mu contains the actual byte offset.

Unaligned memory operations require two accesses to the memory system, and thus incur increased power and bandwidth over aligned accesses. However, they require fewer instructions. When possible, use aligned memory operations and combinations of permute operations.

This instruction uses both slot 0 and slot 1, allowing at most three instructions to execute in a packet that contains `vmemu`.

If the scalar predicate register Pv is true, store a full vector register Vs to memory, using an arbitrary byte-aligned address. Otherwise, the operation becomes a NOP.

Syntax	Behavior
<code>if (!Pv) vmemu(Rt)=Vs</code>	Assembler mapped to: <code>"if (!Pv) vmemu(Rt+#0)=Vs"</code>
<code>if (!Pv) vmemu(Rt+#s4)=Vs</code>	<pre>if (!Pv[0]) { EA=Rt+#s*VBYTES; *EA = Vs; } else { NOP; }</pre>
<code>if (!Pv) vmemu(Rx++#s3)=Vs</code>	<pre>if (!Pv[0]) { EA=Rx; *EA = Vs; Rx=Rx+#s*VBYTES; } else { NOP; }</pre>
<code>if (!Pv) vmemu(Rx++Mu)=Vs</code>	<pre>if (!Pv[0]) { EA=Rx; *EA = Vs; Rx=Rx+MuV; } else { NOP; }</pre>
<code>vmemu(Rt)=Vs</code>	Assembler mapped to: <code>"vmemu(Rt+#0)=Vs"</code>
<code>vmemu(Rt+#s4)=Vs</code>	<pre>EA=Rt+#s*VBYTES; *EA = Vs;</pre>
<code>vmemu(Rx++#s3)=Vs</code>	<pre>EA=Rx; *EA = Vs; Rx=Rx+#s*VBYTES;</pre>
<code>vmemu(Rx++Mu)=Vs</code>	<pre>EA=Rx; *EA = Vs; Rx=Rx+MuV;</pre>

Class: COPROC_VMEM (slots 0)**Notes**

- This instruction uses the HVX permute resource.
- Immediates used in address computation are specified in multiples of vector length.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS									NT		t5					Parse									s5							
0	0	1	0	1	0	0	0	0	0	1	t	t	t	t	t	P	P	i	-	-	i	i	i	1	1	1	s	s	s	s	s	vmemu(Rt+#s4)=Vs
0	0	1	0	1	0	0	0	1	0	1	t	t	t	t	t	P	P	i	v	v	i	i	i	1	1	0	s	s	s	s	s	if (Pv) vmemu(Rt+#s4)=Vs
0	0	1	0	1	0	0	0	1	0	1	t	t	t	t	t	P	P	i	v	v	i	i	i	1	1	1	s	s	s	s	s	if (!Pv) vmemu(Rt+#s4)=Vs
ICLASS									NT		x5					Parse									s5							
0	0	1	0	1	0	0	1	0	0	1	x	x	x	x	x	P	P	-	-	-	i	i	i	1	1	1	s	s	s	s	s	vmemu(Rx++#s3)=Vs
0	0	1	0	1	0	0	1	1	0	1	x	x	x	x	x	P	P	-	v	v	i	i	i	1	1	0	s	s	s	s	s	if (Pv) vmemu(Rx++#s3)=Vs
0	0	1	0	1	0	0	1	1	0	1	x	x	x	x	x	P	P	-	v	v	i	i	i	1	1	1	s	s	s	s	s	if (!Pv) vmemu(Rx++#s3)=Vs
ICLASS									NT		x5					Parse									u1	s5						
0	0	1	0	1	0	1	1	0	0	1	x	x	x	x	x	P	P	u	-	-	-	-	-	1	1	1	s	s	s	s	s	vmemu(Rx++Mu)=Vs
0	0	1	0	1	0	1	1	1	0	1	x	x	x	x	x	P	P	u	v	v	-	-	-	1	1	0	s	s	s	s	s	if (Pv) vmemu(Rx++Mu)=Vs
0	0	1	0	1	0	1	1	1	0	1	x	x	x	x	x	P	P	u	v	v	-	-	-	1	1	1	s	s	s	s	s	if (!Pv) vmemu(Rx++Mu)=Vs

Field name	Description
ICLASS	Instruction class
NT	Nontemporal
Parse	Packet loop parse bits
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x

Instruction Index

A

and

Qd4=and(Qs4,[!]*Qt*4) 40

B

bf

Qd4=vcmp.gt(Vu.bf,Vv.bf) 74

Qx4[&]=vcmp.gt(Vu.bf,Vv.bf) 76

Qx4^=vcmp.gt(Vu.bf,Vv.bf) 77

H

hf

Vd.h=Vu.hf 265

Vd.hf=Vu.h 266

Vd.hf=Vu.qf16 267

Vd.hf=Vu.qf32 267

Vd.qf16=vadd(Vu.hf,Vv.hf) 250

Vd.qf16=vadd(Vu.qf16,Vv.hf) 250

Vd.qf16=vmpy(Vu.hf,Vv.hf) 148

Vd.qf16=vmpy(Vu.qf16,Vv.hf) 148

Vd.qf16=vsub(Vu.hf,Vv.hf) 272

Vd.qf16=vsub(Vu.qf16,Vv.hf) 272

Vdd.qf32=vmpy(Vu.hf,Vv.hf) 148

Vdd.qf32=vmpy(Vu.qf16,Vv.hf) 149

N

no mnemonic

if ([!]*Ps*) Vd=Vu 67, 67

if ([!]*Qv*4) Vx.b[+]=Vu.b 82

if ([!]*Qv*4) Vx.h[+]=Vu.h 82

if ([!]*Qv*4) Vx.w[+]=Vu.w 82

Vd.tmp=Vu 69

Vd=Vu 67

not

Qd4=not(Qs4) 54

O

or

Qd4=or(Qs4,[!]*Qt*4) 40

P

prefixsum

Vd.b=prefixsum(*Qv*4) 248

Vd.h=prefixsum(*Qv*4) 248

Vd.w=prefixsum(*Qv*4) 248

Q

qf16

Vd.qf16=vadd(Vu.qf16,Vv.qf16) 251

Vd.qf16=vmpy(Vu.qf16,Vv.qf16) 148

Vd.qf16=vsub(Vu.qf16,Vv.qf16) 273

Vdd.qf32=vmpy(Vu.qf16,Vv.qf16) 149

qf32

Vd.qf32=vadd(Vu.qf32,Vv.qf32) [252](#)
Vd.qf32=vmpy(Vu.qf32,Vv.qf32) [157](#)
Vd.qf32=vsub(Vu.qf32,Vv.qf32) [274](#)

S**sf**

Vd.qf32=vadd(Vu.qf32,Vv.sf) [252](#)
Vd.qf32=vadd(Vu.sf,Vv.sf) [252](#), [253](#)
Vd.qf32=vmpy(Vu.sf,Vv.sf) [157](#)
Vd.qf32=vsub(Vu.qf32,Vv.sf) [274](#)
Vd.qf32=vsub(Vu.sf,Vv.sf) [275](#), [275](#)
Vd.sf=Vu.qf32 [267](#)
Vd.sf=Vu.w [266](#)
Vd.w=Vu.sf [265](#)

U

ub

Qd4=vcmp.eq(Vu.ub,Vv.ub) 74
 Qd4=vcmp.gt(Vu.ub,Vv.ub) 75
 Qx4 [&|]=vcmp.eq(Vu.ub,Vv.ub) 75
 Qx4 [&|]=vcmp.gt(Vu.ub,Vv.ub) 76
 Qx4^=vcmp.eq(Vu.ub,Vv.ub) 76
 Qx4^=vcmp.gt(Vu.ub,Vv.ub) 77
 Vd.b=vnavg(Vu.ub,Vv.ub) 71
 Vd.h=vdmpy(Vu.ub,Rt.b) 177
 Vd.ub=vabs(Vu.b) 59
 Vd.ub=vabsdiff(Vu.ub,Vv.ub) 196
 Vd.ub=vadd(Vu.ub,Vv.b):sat 61
 Vd.ub=vadd(Vu.ub,Vv.ub):sat 61
 Vd.ub=vasr(Vu.h,Vv.h,Rt)[:rnd]:sat 245, 259
 Vd.ub=vavg(Vu.ub,Vv.ub)[:rnd] 72
 Vd.ub=vlsr(Vu.ub,Rt) 259
 Vd.ub=vmax(Vu.ub,Vv.ub) 56
 Vd.ub=vmin(Vu.ub,Vv.ub) 56
 Vd.ub=vpack(Vu.h,Vv.h):sat 211
 Vd.ub=vround(Vu.h,Vv.h):sat 269
 Vd.ub=vsat(Vu.h,Vv.h) 87
 Vd.ub=vsb(Vu.ub,Vv.b):sat 61
 Vd.ub=vsb(Vu.ub,Vv.ub):sat 61
 Vd.uw=vrmPy(Vu.ub,Rt.ub) 188
 Vd.uw=vrmPy(Vu.ub,Vv.ub) 190
 Vd.w=vmpyi(Vu.w,Rt.ub) 185
 Vd.w=vrmPy(Vu.ub,Rt.b) 188
 Vd.w=vrmPy(Vu.ub,Vv.b) 191
 Vdd.h=vadd(Vu.ub,Vv.ub) 125
 Vdd.h=vdmpy(Vuu.ub,Rt.b) 130
 Vdd.h=vmpa(Vuu.ub,Rt.b) 136
 Vdd.h=vmpa(Vuu.ub,Rt.ub) 136
 Vdd.h=vmpa(Vuu.ub,Vvv.b) 136
 Vdd.h=vmpa(Vuu.ub,Vvv.ub) 137
 Vdd.h=vmpy(Vu.ub,Rt.b) 141
 Vdd.h=vmpy(Vu.ub,Vv.b) 145
 Vdd.h=vsub(Vu.ub,Vv.ub) 125
 Vdd.h=vtmPy(Vuu.ub,Rt.b) 169
 Vdd.ub=vadd(Vuu.ub,Vvv.ub):sat 50
 Vdd.ub=vsub(Vuu.ub,Vvv.ub):sat 50
 Vdd.uw=vrmPy(Vuu.ub,Rt.ub,#u1) 162
 Vdd.uw=vrsad(Vuu.ub,Rt.ub,#u1) 175
 Vdd.w=v6mPy(Vuu.ub,Vvv.b,#u2):h 119, 120
 Vdd.w=v6mPy(Vuu.ub,Vvv.b,#u2):v 121
 Vdd.w=vrmPy(Vuu.ub,Rt.b,#u1) 162
 Vx.h+=vdmpy(Vu.ub,Rt.b) 177
 Vx.uw+=vrmPy(Vu.ub,Rt.ub) 188
 Vx.uw+=vrmPy(Vu.ub,Vv.ub) 165
 Vx.w+=vmpyi(Vu.w,Rt.ub) 185
 Vx.w+=vrmPy(Vu.ub,Rt.b) 189
 Vx.w+=vrmPy(Vu.ub,Vv.b) 166
 Vxx.h+=vadd(Vu.ub,Vv.ub) 126
 Vxx.h+=vdmpy(Vuu.ub,Rt.b) 130
 Vxx.h+=vmpa(Vuu.ub,Rt.b) 137
 Vxx.h+=vmpa(Vuu.ub,Rt.ub) 137
 Vxx.h+=vmpy(Vu.ub,Rt.b) 141
 Vxx.h+=vmpy(Vu.ub,Vv.b) 145
 Vxx.h+=vtmPy(Vuu.ub,Rt.b) 169
 Vxx.uw+=vrmPy(Vuu.ub,Rt.ub,#u1) 162
 Vxx.uw+=vrsad(Vuu.ub,Rt.ub,#u1) 175
 Vxx.w+=v6mPy(Vuu.ub,Vvv.b,#u2):h 122
 Vxx.w+=v6mPy(Vuu.ub,Vvv.b,#u2):v 123
 Vxx.w+=vrmPy(Vuu.ub,Rt.b,#u1) 163

uh

Qd4=vcmp.eq (Vu.uh, Vv.uh) 74
 Qd4=vcmp.gt (Vu.uh, Vv.uh) 75
 Qx4 [&|]=vcmp.eq (Vu.uh, Vv.uh) 75
 Qx4 [&|]=vcmp.gt (Vu.uh, Vv.uh) 76
 Qx4^=vcmp.eq (Vu.uh, Vv.uh) 76
 Qx4^=vcmp.gt (Vu.uh, Vv.uh) 77
 Vd.h=vlut4 (Vu.uh, Rtt.h) 133
 Vd.ub=vasr (Vu.uh, Vv.uh, Rt) [:rnd]:sat 245, 259
 Vd.ub=vasr (Vuu.uh, Vv.ub) [:rnd]:sat 263
 Vd.ub=vround (Vu.uh, Vv.uh):sat 269
 Vd.uh=vabs (Vu.h) 59
 Vd.uh=vabsdiff (Vu.h, Vv.h) 196
 Vd.uh=vabsdiff (Vu.uh, Vv.uh) 196
 Vd.uh=vadd (Vu.uh, Vv.uh):sat 61
 Vd.uh=vasr (Vu.uw, Vv.uw, Rt) [:rnd]:sat 245, 259
 Vd.uh=vasr (Vu.w, Vv.w, Rt) [:rnd]:sat 245, 259
 Vd.uh=vasr (Vuu.w, Vv.uh) [:rnd]:sat 263, 264
 Vd.uh=vavg (Vu.uh, Vv.uh) [:rnd] 72
 Vd.uh=vcl0 (Vu.uh) 276
 Vd.uh=vlsr (Vu.uh, Rt) 259
 Vd.uh=vmax (Vu.uh, Vv.uh) 56
 Vd.uh=vmin (Vu.uh, Vv.uh) 56
 Vd.uh=vmpy (Vu.uh, Vv.uh):>>16 183
 Vd.uh=vpack (Vu.w, Vv.w):sat 211
 Vd.uh=vround (Vu.uw, Vv.uw):sat 269
 Vd.uh=vround (Vu.w, Vv.w):sat 269
 Vd.uh=vsat (Vu.uw, Vv.uw) 87
 Vd.uh=vsb (Vu.uh, Vv.uh):sat 61
 Vd.uw=vmpye (Vu.uh, Rt.uh) 187
 Vd.w=vdmpy (Vu.h, Rt.uh):sat 179
 Vd.w=vdmpy (Vuu.h, Rt.uh, #1):sat 129
 Vd.w=vmpye (Vu.w, Vv.uh) 158
 Vd.w=vmpyie (Vu.w, Vv.uh) 153
 Vdd.uh=vadd (Vuu.uh, Vvv.uh):sat 50
 Vdd.uh=vmpy (Vu.ub, Rt.ub) 141
 Vdd.uh=vmpy (Vu.ub, Vv.ub) 145
 Vdd.uh=vsub (Vuu.uh, Vvv.uh):sat 51
 Vdd.uh=vunpack (Vu.ub) 235
 Vdd.uh=vzxt (Vu.ub) 48
 Vdd.uw=vsad (Vuu.uh, Rt.uh) 173
 Vdd.uw=vmpy (Vu.uh, Rt.uh) 141
 Vdd.uw=vmpy (Vu.uh, Vv.uh) 145
 Vdd.uw=vunpack (Vu.uh) 235
 Vdd.uw=vzxt (Vu.uh) 48
 Vdd.w=vadd (Vu.uh, Vv.uh) 126
 Vdd.w=vmpa (Vuu.uh, Rt.b) 137
 Vdd.w=vmpy (Vu.h, Vv.uh) 145
 Vdd.w=vsub (Vu.uh, Vv.uh) 126
 Vdd=vmpye (Vu.w, Vv.uh) 158
 Vx.h=vmpa (Vx.h, Vu.uh, Rtt.uh):sat 134
 Vx.h=vmps (Vx.h, Vu.uh, Rtt.uh):sat 134
 Vx.uw+=vmpye (Vu.uh, Rt.uh) 187
 Vx.w+=vdmpy (Vu.h, Rt.uh):sat 180
 Vx.w+=vdmpy (Vuu.h, Rt.uh, #1):sat 130
 Vx.w+=vmpyie (Vu.w, Vv.uh) 153
 Vxx.uh+=vmpy (Vu.ub, Rt.ub) 141
 Vxx.uh+=vmpy (Vu.ub, Vv.ub) 145
 Vxx.uw+=vsad (Vuu.uh, Rt.uh) 173
 Vxx.uw+=vmpy (Vu.uh, Rt.uh) 141
 Vxx.uw+=vmpy (Vu.uh, Vv.uh) 146
 Vxx.w+=vadd (Vu.uh, Vv.uh) 126
 Vxx.w+=vmpa (Vuu.uh, Rt.b) 137
 Vxx.w+=vmpy (Vu.h, Vv.uh) 146

V**vabs**

Vd.b=vabs(Vu.b) [:sat] 59
 Vd.h=vabs(Vu.h) [:sat] 59
 Vd.uw=vabs(Vu.w) 59
 Vd.w=vabs(Vu.w) [:sat] 59

vabsdiff

Vd.uw=vabsdiff(Vu.w, Vv.w) 197

vadd

Vd.b=vadd(Vu.b, Vv.b) [:sat] 61
 Vd.h=vadd(vclb(Vu.h), Vv.h) 276
 Vd.h=vadd(Vu.h, Vv.h) [:sat] 61
 Vd.uw=vadd(Vu.uw, Vv.uw) :sat 61
 Vd.w, Qe4=vadd(Vu.w, Vv.w) :carry 64
 Vd.w=vadd(vclb(Vu.w), Vv.w) 276
 Vd.w=vadd(Vu.w, Vv.w, Qs4) :carry:sat 64
 Vd.w=vadd(Vu.w, Vv.w, Qx4) :carry 64
 Vd.w=vadd(Vu.w, Vv.w) [:sat] 61
 Vdd.b=vadd(Vuu.b, Vvv.b) [:sat] 50
 Vdd.h=vadd(Vuu.h, Vvv.h) [:sat] 50
 Vdd.uw=vadd(Vuu.uw, Vvv.uw) :sat 51
 Vdd.w=vadd(Vu.h, Vv.h) 126
 Vdd.w=vadd(Vuu.w, Vvv.w) [:sat] 51
 Vxx.w+=vadd(Vu.h, Vv.h) 126

valign

Vd=valign(Vu, Vv, #u3) 200
 Vd=valign(Vu, Vv, Rt) 200

vand

Qd4=vand(Vu, Rt) 194
 Qx4 |=vand(Vu, Rt) 194
 Vd=vand([!]Qu4, Rt) 195
 Vd=vand([!]Qv4, Vu) 55
 Vd=vand(Vu, Vv) 66
 Vx |=vand([!]Qu4, Rt) 195

vasl

Vd.h=vasl(Vu.h, Rt) 258
 Vd.h=vasl(Vu.h, Vv.h) 258
 Vd.w=vasl(Vu.w, Rt) 259
 Vd.w=vasl(Vu.w, Vv.w) 259
 Vx.h+=vasl(Vu.h, Rt) 255
 Vx.w+=vasl(Vu.w, Rt) 255

vasr

Vd.b=vasr(Vu.h, Vv.h, Rt) [:rnd] :sat 245, 258
 Vd.h=vasr(Vu.h, Rt) 258
 Vd.h=vasr(Vu.h, Vv.h) 258
 Vd.h=vasr(Vu.w, Vv.w, Rt) :rnd:sat 245, 258
 Vd.h=vasr(Vu.w, Vv.w, Rt) [:sat] 245, 258
 Vd.w=vasr(Vu.w, Rt) 259
 Vd.w=vasr(Vu.w, Vv.w) 260
 Vx.h+=vasr(Vu.h, Rt) 255
 Vx.w+=vasr(Vu.w, Rt) 255

vasrinto

Vxx.w=vasrinto(Vu.w, Vv.w) 221

vavg

Vd.b=vavg(Vu.b, Vv.b) [:rnd] 71
 Vd.h=vavg(Vu.h, Vv.h) [:rnd] 71
 Vd.uw=vavg(Vu.uw, Vv.uw) [:rnd] 72
 Vd.w=vavg(Vu.w, Vv.w) [:rnd] 72

vc10
 Vd.uw=vc10 (Vu.uw) 276

vcmp.eq
 Qd4=vcmp.eq (Vu.b, Vv.b) 74
 Qd4=vcmp.eq (Vu.h, Vv.h) 74
 Qd4=vcmp.eq (Vu.uw, Vv.uw) 74
 Qd4=vcmp.eq (Vu.w, Vv.w) 74
 Qx4 [&|] =vcmp.eq (Vu.b, Vv.b) 75
 Qx4 [&|] =vcmp.eq (Vu.h, Vv.h) 75
 Qx4 [&|] =vcmp.eq (Vu.uw, Vv.uw) 75
 Qx4 [&|] =vcmp.eq (Vu.w, Vv.w) 75
 Qx4^=vcmp.eq (Vu.b, Vv.b) 76
 Qx4^=vcmp.eq (Vu.h, Vv.h) 76
 Qx4^=vcmp.eq (Vu.uw, Vv.uw) 76
 Qx4^=vcmp.eq (Vu.w, Vv.w) 77

vcmp.gt
 Qd4=vcmp.gt (Vu.b, Vv.b) 74
 Qd4=vcmp.gt (Vu.h, Vv.h) 74
 Qd4=vcmp.gt (Vu.hf, Vv.hf) 75
 Qd4=vcmp.gt (Vu.sf, Vv.sf) 75
 Qd4=vcmp.gt (Vu.uw, Vv.uw) 75
 Qd4=vcmp.gt (Vu.w, Vv.w) 75
 Qx4 [&|] =vcmp.gt (Vu.b, Vv.b) 75
 Qx4 [&|] =vcmp.gt (Vu.h, Vv.h) 76
 Qx4 [&|] =vcmp.gt (Vu.hf, Vv.hf) 76
 Qx4 [&|] =vcmp.gt (Vu.sf, Vv.sf) 76
 Qx4 [&|] =vcmp.gt (Vu.uw, Vv.uw) 76
 Qx4 [&|] =vcmp.gt (Vu.w, Vv.w) 76
 Qx4^=vcmp.gt (Vu.b, Vv.b) 77
 Qx4^=vcmp.gt (Vu.h, Vv.h) 77
 Qx4^=vcmp.gt (Vu.hf, Vv.hf) 77
 Qx4^=vcmp.gt (Vu.sf, Vv.sf) 77
 Qx4^=vcmp.gt (Vu.uw, Vv.uw) 77
 Qx4^=vcmp.gt (Vu.w, Vv.w) 77

vcombine
 if ([!] Ps) Vdd=vcombine (Vu, Vv) 42
 Vdd.tmp=vcombine (Vu, Vv) 69
 Vdd=vcombine (Vu, Vv) 42

vdeal
 Vd.b=vdeal (Vu.b) 207
 Vd.h=vdeal (Vu.h) 208
 Vdd=vdeal (Vu, Vv, Rt) 224
 vdeal (Vy, Vx, Rt) 225

vdeale
 Vd.b=vdeale (Vu.b, Vv.b) 208

vdelta
 Vd=vdelta (Vu, Vv) 205

vdmpy
 Vd.w=vdmpy (Vu.h, Rt.b) 177
 Vd.w=vdmpy (Vu.h, Rt.h) :sat 179
 Vd.w=vdmpy (Vu.h, Vv.h) :sat 179
 Vd.w=vdmpy (Vuu.h, Rt.h) :sat 129
 Vdd.w=vdmpy (Vuu.h, Rt.b) 130
 Vx.w+=vdmpy (Vu.h, Rt.b) 178
 Vx.w+=vdmpy (Vu.h, Rt.h) :sat 179
 Vx.w+=vdmpy (Vu.h, Vv.h) :sat 130
 Vx.w+=vdmpy (Vuu.h, Rt.h) :sat 130
 Vxx.w+=vdmpy (Vuu.h, Rt.b) 131

vextract
 Rd.w=vextract (Vu, Rs) 91
 Rd=vextract (Vu, Rs) 91

vgather
 if (Qs4) vtmp.w=vgather(Rt,Mu,Vv.w).w [96](#)
 vtmp.w=vgather(Rt,Mu,Vv.w).w [96](#)

vhist
 vhist [33](#), [33](#), [33](#), [34](#)
 vhist(Qv4) [35](#)

vininsert
 Vx.w=vininsert(Rt) [198](#)

vlalign
 Vd=vlalign(Vu,Vv,#u3) [200](#)
 Vd=vlalign(Vu,Vv,Rt) [200](#)

vlsr
 Vd.h=vlsr(Vu.h,Vv.h) [258](#)
 Vd.uw=vlsr(Vu.uw,Rt) [259](#)
 Vd.w=vlsr(Vu.w,Vv.w) [260](#)

vlut16
 Vdd.h=vlut16(Vu.b,Vv.h,#u3) [231](#)
 Vdd.h=vlut16(Vu.b,Vv.h,Rt) [232](#)
 Vdd.h=vlut16(Vu.b,Vv.h,Rt):nomatch [232](#)
 Vxx.h|=vlut16(Vu.b,Vv.h,#u3) [233](#)
 Vxx.h|=vlut16(Vu.b,Vv.h,Rt) [233](#)

vlut32
 Vd.b=vlut32(Vu.b,Vv.b,#u3) [218](#)
 Vd.b=vlut32(Vu.b,Vv.b,Rt) [218](#)
 Vd.b=vlut32(Vu.b,Vv.b,Rt):nomatch [219](#)
 Vx.b|=vlut32(Vu.b,Vv.b,#u3) [232](#)
 Vx.b|=vlut32(Vu.b,Vv.b,Rt) [232](#)

vmax
 Vd.b=vmax(Vu.b,Vv.b) [56](#)
 Vd.h=vmax(Vu.h,Vv.h) [56](#)
 Vd.hf=vmax(Vu.hf,Vv.hf) [56](#)
 Vd.sf=vmax(Vu.sf,Vv.sf) [56](#)
 Vd.w=vmax(Vu.w,Vv.w) [57](#)

vmem

if ([!] Pv) Vd.cur=vmem(Rt) 104
 if ([!] Pv) Vd.cur=vmem(Rt):nt 104
 if ([!] Pv) Vd.cur=vmem(Rt+#s4) 104
 if ([!] Pv) Vd.cur=vmem(Rt+#s4):nt 105
 if ([!] Pv) Vd.cur=vmem(Rx++#s3) 105
 if ([!] Pv) Vd.cur=vmem(Rx++#s3):nt 105
 if ([!] Pv) Vd.cur=vmem(Rx++Mu) 105
 if ([!] Pv) Vd.cur=vmem(Rx++Mu):nt 105
 if ([!] Pv) Vd.tmp=vmem(Rt) 107
 if ([!] Pv) Vd.tmp=vmem(Rt):nt 107
 if ([!] Pv) Vd.tmp=vmem(Rt+#s4) 107
 if ([!] Pv) Vd.tmp=vmem(Rt+#s4):nt 108
 if ([!] Pv) Vd.tmp=vmem(Rx++#s3) 108
 if ([!] Pv) Vd.tmp=vmem(Rx++#s3):nt 108
 if ([!] Pv) Vd.tmp=vmem(Rx++Mu) 108
 if ([!] Pv) Vd.tmp=vmem(Rx++Mu):nt 108
 if ([!] Pv) Vd=vmem(Rt) 101
 if ([!] Pv) Vd=vmem(Rt):nt 101
 if ([!] Pv) Vd=vmem(Rt+#s4) 101
 if ([!] Pv) Vd=vmem(Rt+#s4):nt 101
 if ([!] Pv) Vd=vmem(Rx++#s3) 101
 if ([!] Pv) Vd=vmem(Rx++#s3):nt 101
 if ([!] Pv) Vd=vmem(Rx++Mu) 101
 if ([!] Pv) Vd=vmem(Rx++Mu):nt 101
 if ([!] Pv) vmem(Rt):nt=Vs 284
 if ([!] Pv) vmem(Rt)=Vs 284
 if ([!] Pv) vmem(Rt+#s4):nt=Os8.new 281
 if ([!] Pv) vmem(Rt+#s4):nt=Vs 284
 if ([!] Pv) vmem(Rt+#s4)=Os8.new 281
 if ([!] Pv) vmem(Rt+#s4)=Vs 284
 if ([!] Pv) vmem(Rx++#s3):nt=Os8.new 281
 if ([!] Pv) vmem(Rx++#s3):nt=Vs 284
 if ([!] Pv) vmem(Rx++#s3)=Os8.new 281
 if ([!] Pv) vmem(Rx++#s3)=Vs 284
 if ([!] Pv) vmem(Rx++Mu):nt=Os8.new 281
 if ([!] Pv) vmem(Rx++Mu):nt=Vs 284
 if ([!] Pv) vmem(Rx++Mu)=Os8.new 282
 if ([!] Pv) vmem(Rx++Mu)=Vs 285
 if ([!] Qv4) vmem(Rt):nt=Vs 278
 if ([!] Qv4) vmem(Rt)=Vs 278
 if ([!] Qv4) vmem(Rt+#s4):nt=Vs 278
 if ([!] Qv4) vmem(Rt+#s4)=Vs 278
 if ([!] Qv4) vmem(Rx++#s3):nt=Vs 278
 if ([!] Qv4) vmem(Rx++#s3)=Vs 279
 if ([!] Qv4) vmem(Rx++Mu):nt=Vs 279
 if ([!] Qv4) vmem(Rx++Mu)=Vs 279
 Vd.cur=vmem(Rt+#s4) 104
 Vd.cur=vmem(Rt+#s4):nt 104
 Vd.cur=vmem(Rx++#s3) 104
 Vd.cur=vmem(Rx++#s3):nt 104
 Vd.cur=vmem(Rx++Mu) 104
 Vd.cur=vmem(Rx++Mu):nt 104
 Vd.tmp=vmem(Rt+#s4) 107
 Vd.tmp=vmem(Rt+#s4):nt 107
 Vd.tmp=vmem(Rx++#s3) 107
 Vd.tmp=vmem(Rx++#s3):nt 107
 Vd.tmp=vmem(Rx++Mu) 107
 Vd.tmp=vmem(Rx++Mu):nt 107
 Vd=vmem(Rt) 98, 98
 Vd=vmem(Rt):nt 98
 Vd=vmem(Rt+#s10:6) 98
 Vd=vmem(Rt+#s4) 98
 Vd=vmem(Rt+#s4):nt 98
 Vd=vmem(Rt+Rs) 98
 Vd=vmem(Rx++#s10:6) 98
 Vd=vmem(Rx++#s3) 98
 Vd=vmem(Rx++#s3):nt 98
 Vd=vmem(Rx++I:circ(Mu), Rt) 99
 Vd=vmem(Rx++I:circ(Mu)) 98
 Vd=vmem(Rx++Mu, Rt) 100
 Vd=vmem(Rx++Mu) 99, 100

```

Vd=vmem(Rx++Mu):nt 100
vmem(Rt):nt=Os8.new 282
vmem(Rt):nt=Vs 285
vmem(Rt)=Os8.new 282
vmem(Rt)=Vs 285
vmem(Rt+#s10:6)=Vs 285
vmem(Rt+#s4):nt=Os8.new 282
vmem(Rt+#s4):nt=Vs 285
vmem(Rt+#s4)=Os8.new 282
vmem(Rt+#s4)=Vs 285
vmem(Rx++#s10:6)=Vs 285
vmem(Rx++#s3):nt=Os8.new 282
vmem(Rx++#s3):nt=Vs 285
vmem(Rx++#s3)=Os8.new 282
vmem(Rx++#s3)=Vs 285
vmem(Rx++I:circ(Mu),Rt)=Vs 285, 286, 287
vmem(Rx++I:circ(Mu))=Vs 285
vmem(Rx++Mu,Rt)=Vs 287, 288, 289
vmem(Rx++Mu):nt=Os8.new 282
vmem(Rx++Mu):nt=Vs 287
vmem(Rx++Mu)=Os8.new 282
vmem(Rx++Mu)=Vs 287, 287

```

vmemu

```

if ([!]Pv) vmemu(Rt)=Vs 291
if ([!]Pv) vmemu(Rt+#s4)=Vs 291
if ([!]Pv) vmemu(Rx++#s3)=Vs 291
if ([!]Pv) vmemu(Rx++Mu)=Vs 291
Vd=vmemu(Rt) 110
Vd=vmemu(Rt+#s4) 110
Vd=vmemu(Rx++#s3) 110
Vd=vmemu(Rx++Mu) 110
vmemu(Rt)=Vs 291
vmemu(Rt+#s4)=Vs 291
vmemu(Rx++#s3)=Vs 291
vmemu(Rx++Mu)=Vs 291

```

vmin

```

Vd.b=vmin(Vu.b,Vv.b) 56
Vd.h=vmin(Vu.h,Vv.h) 56
Vd.hf=vmin(Vu.hf,Vv.hf) 56
Vd.sf=vmin(Vu.sf,Vv.sf) 56
Vd.w=vmin(Vu.w,Vv.w) 57

```

vmpa

```

Vdd.w=vmpa(Vuu.h,Rt.b) 137
Vx.h=vmpa(Vx.h,Vu.h,Rtt.h):sat 134
Vxx.w+=vmpa(Vuu.h,Rt.b) 137

```

vmpy

```

Vd.h=vmpy(Vu.h,Rt.h):<<1:rnd:sat 182
Vd.h=vmpy(Vu.h,Rt.h):<<1:sat 182
Vd.h=vmpy(Vu.h,Vv.h):<<1:rnd:sat 183
Vdd.h=vmpy(Vu.b,Vv.b) 145
Vdd.w=vmpy(Vu.h,Rt.h) 141
Vdd.w=vmpy(Vu.h,Vv.h) 145
Vxx.h+=vmpy(Vu.b,Vv.b) 145
Vxx.w+=vmpy(Vu.h,Rt.h) 141
Vxx.w+=vmpy(Vu.h,Rt.h):sat 141
Vxx.w+=vmpy(Vu.h,Vv.h) 146

```

vmpyi

```

Vd.h=vmpyi(Vu.h,Rt.b) 185
Vd.h=vmpyi(Vu.h,Vv.h) 151
Vd.w=vmpyi(Vu.w,Rt.b) 185
Vd.w=vmpyi(Vu.w,Rt.h) 155
Vx.h+=vmpyi(Vu.h,Rt.b) 185
Vx.h+=vmpyi(Vu.h,Vv.h) 151
Vx.w+=vmpyi(Vu.w,Rt.b) 185
Vx.w+=vmpyi(Vu.w,Rt.h) 155

```

vmpyie
 Vx.w+=vmpyie(Vu.w,Vv.h) 153

vmpyieo
 Vd.w=vmpyieo(Vu.h,Vv.h) 184

vmpyio
 Vd.w=vmpyio(Vu.w,Vv.h) 153

vmpyo
 Vd.w=vmpyo(Vu.w,Vv.h):<<1[:rnd]:sat 158
 Vx.w+=vmpyo(Vu.w,Vv.h):<<1[:rnd]:sat:shift 158
 Vxx+=vmpyo(Vu.w,Vv.h) 159

vmux
 Vd=vmux(Qt4,Vu,Vv) 85

vnavg
 Vd.b=vnavg(Vu.b,Vv.b) 71
 Vd.h=vnavg(Vu.h,Vv.h) 71
 Vd.w=vnavg(Vu.w,Vv.w) 72

vnormamt
 Vd.h=vnormamt(Vu.h) 276
 Vd.w=vnormamt(Vu.w) 276

vnot
 Vd=vnot(Vu) 66

vor
 Vd=vor(Vu,Vv) 66

vpack
 Vd.b=vpack(Vu.h,Vv.h):sat 210
 Vd.h=vpack(Vu.w,Vv.w):sat 211

vpacke
 Vd.b=vpacke(Vu.h,Vv.h) 210
 Vd.h=vpacke(Vu.w,Vv.w) 211

vpacko
 Vd.b=vpacko(Vu.h,Vv.h) 211
 Vd.h=vpacko(Vu.w,Vv.w) 211

vpopcount
 Vd.h=vpopcount(Vu.h) 276

vrdelta
 Vd=vrdelta(Vu,Vv) 205

vrmpy
 Vd.w=vrmpy(Vu.b,Vv.b) 191
 Vx.w+=vrmpy(Vu.b,Vv.b) 166

vror
 Vd=vror(Vu,Rt) 200

vrotr
 Vd.uw=vrotr(Vu.uw,Vv.uw) 271

vround
 Vd.b=vround(Vu.h,Vv.h):sat 268
 Vd.h=vround(Vu.w,Vv.w):sat 269

vsat
 Vd.h=vsat(Vu.w,Vv.w) 87

vsatdwVd.w=vsatdw(Vu.w, Vv.w) [87](#)**vscatter**

if (Qs4) vscatter(Rt, Mu, Vv.h) .h=Vw32 [241](#)
 if (Qs4) vscatter(Rt, Mu, Vv.h)=Vw32.h [241](#)
 if (Qs4) vscatter(Rt, Mu, Vv.w) .w=Vw32 [241](#)
 if (Qs4) vscatter(Rt, Mu, Vv.w)=Vw32.w [241](#)
 if (Qs4) vscatter(Rt, Mu, Vvv.w) .h=Vw32 [238](#)
 if (Qs4) vscatter(Rt, Mu, Vvv.w)=Vw32.h [238](#)
 vscatter(Rt, Mu, Vv.h) .h+=Vw32 [241](#)
 vscatter(Rt, Mu, Vv.h) .h=Vw32 [241](#)
 vscatter(Rt, Mu, Vv.h) +=Vw32.h [241](#)
 vscatter(Rt, Mu, Vv.h)=Vw32.h [241](#)
 vscatter(Rt, Mu, Vv.w) .w+=Vw32 [241](#)
 vscatter(Rt, Mu, Vv.w) .w=Vw32 [241](#)
 vscatter(Rt, Mu, Vv.w) +=Vw32.w [241](#)
 vscatter(Rt, Mu, Vv.w)=Vw32.w [242](#)
 vscatter(Rt, Mu, Vvv.w) .h+=Vw32 [238](#)
 vscatter(Rt, Mu, Vvv.w) .h=Vw32 [238](#)
 vscatter(Rt, Mu, Vvv.w) +=Vw32.h [238](#)
 vscatter(Rt, Mu, Vvv.w)=Vw32.h [238](#)

vsetqQd4=vsetq(Rt) [213](#)**vsetq2**Qd4=vsetq2(Rt) [213](#)**vshuff**

Vd.b=vshuff(Vu.b) [208](#)
 Vd.h=vshuff(Vu.h) [208](#)
 Vdd=vshuff(Vu, Vv, Rt) [225](#)
 vshuff(Vy, Vx, Rt) [225](#)

vshuffe

Qd4.b=vshuffe(Qs4.h, Qt4.h) [40](#)
 Qd4.h=vshuffe(Qs4.w, Qt4.w) [40](#)
 Vd.b=vshuffe(Vu.b, Vv.b) [89](#)
 Vd.h=vshuffe(Vu.h, Vv.h) [90](#)

vshuffo

Vd.b=vshuffo(Vu.b, Vv.b) [89](#)
 Vd.h=vshuffo(Vu.h, Vv.h) [90](#)

vshuffoe

Vdd.b=vshuffoe(Vu.b, Vv.b) [43](#)
 Vdd.h=vshuffoe(Vu.h, Vv.h) [44](#)

vsplat

Vd.b=vsplat(Rt) [192](#)
 Vd.h=vsplat(Rt) [192](#)
 Vd=vsplat(Rt) [192](#)

vsub

Vd.b=vsub(Vu.b, Vv.b) [:sat] [61](#)
 Vd.h=vsub(Vu.h, Vv.h) [:sat] [61](#)
 Vd.uw=vsub(Vu.uw, Vv.uw) :sat [61](#)
 Vd.w, Qe4=vsub(Vu.w, Vv.w) :carry [64](#)
 Vd.w=vsub(Vu.w, Vv.w, Qx4) :carry [64](#)
 Vd.w=vsub(Vu.w, Vv.w) [:sat] [62](#)
 Vdd.b=vsub(Vuu.b, Vvv.b) [:sat] [50](#)
 Vdd.h=vsub(Vuu.h, Vvv.h) [:sat] [50](#)
 Vdd.uw=vsub(Vuu.uw, Vvv.uw) :sat [51](#)
 Vdd.w=vsub(Vu.h, Vv.h) [126](#)
 Vdd.w=vsub(Vuu.w, Vvv.w) [:sat] [51](#)

vswap
 Vdd=vswap (Qt4, Vu, Vv) [45](#)
 Vdd=vswap (Qt8, Vu, Vv) [46](#)

vsxt
 Vdd.h=vsxt (Vu.b) [48](#)
 Vdd.w=vsxt (Vu.h) [48](#)

vtmp.h
 if (Qs4) vtmp.h=vgather (Rt, Mu, Vv.h) .h [96](#)
 if (Qs4) vtmp.h=vgather (Rt, Mu, Vvv.w) .h [94](#)
 vtmp.h=vgather (Rt, Mu, Vv.h) .h [96](#)
 vtmp.h=vgather (Rt, Mu, Vvv.w) .h [94](#)

vtmpy
 Vdd.h=vtmpy (Vuu.b, Rt.b) [168](#)
 Vdd.w=vtmpy (Vuu.h, Rt.b) [169](#)
 Vxx.h+=vtmpy (Vuu.b, Rt.b) [169](#)
 Vxx.w+=vtmpy (Vuu.h, Rt.b) [170](#)

vtrans2x2
 vtrans2x2 (Vy, Vx, Rt) [225](#)

vunpack
 Vdd.h=vunpack (Vu.b) [235](#)
 Vdd.w=vunpack (Vu.h) [235](#)

vunpacko
 Vxx.h|=vunpacko (Vu.b) [236](#)
 Vxx.w|=vunpacko (Vu.h) [236](#)

vwhist128
 vwhist128 [37](#)
 vwhist128 (#u1) [37](#)
 vwhist128 (Qv4, #u1) [38](#)
 vwhist128 (Qv4) [38](#)

vwhist256
 sat
 vwhist256:sat [39](#)
 vwhist256 [38](#)
 vwhist256 (Qv4) [38](#)
 vwhist256 (Qv4) :sat [39](#)

vxor
 Vd=vxor (Vu, Vv) [66](#)

X

xor
 Qd4=xor (Qs4, Qt4) [40](#)