Qualcomm® Hexagon™ V65 HVX

Programmer's Reference Manual

80-N2040-41 Rev. B

November 30, 2018

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

Contents

Figures

Tables

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 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 be mapped to SIMD parallel processing.

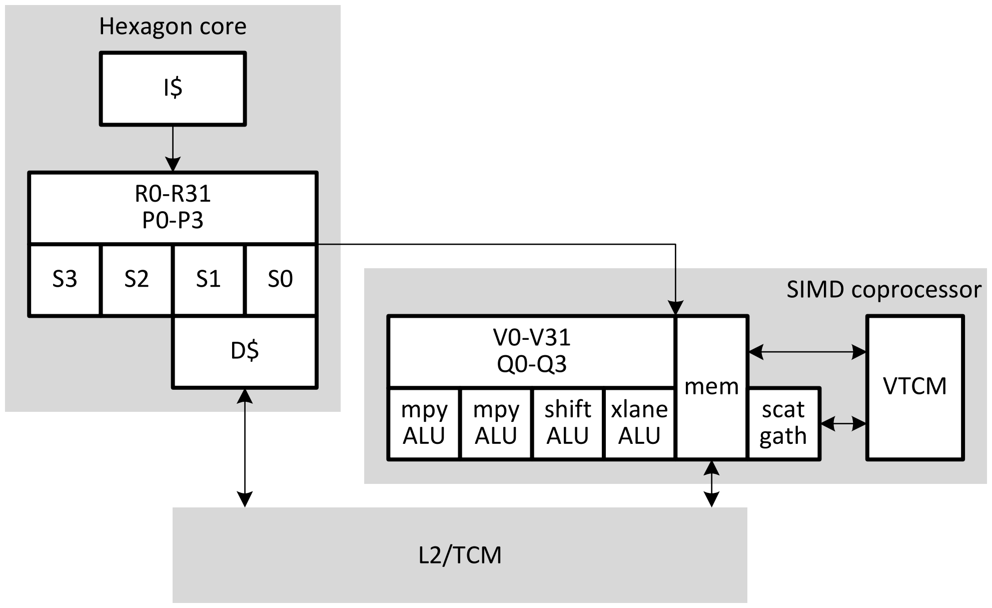


Figure 1-1 Hexagon core with attached SIMD coprocessor

The Hexagon instruction set architecture (ISA) is extended with new HVX instructions. These instructions use HVX compute resources and can be freely mixed with normal Hexagon instructions in a 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 be executed in parallel.

The main features of HVX are described in the following subsections.

1.2.1 Vector length

The HVX architecture has a scalable, but implementation-specific vector length. HVX supports two vector lengths selectable by the V2X bit in the core SYSCFG register.

- With V2X=0, a base vector length is used, which is 512 bits (64B). Each vector register (V) is 512-bit (64B), and each vector predicate register (Q) is 64 bits wide (1 bit per vector register byte). *This is a deprecated vector length and is not available in all implementations; dependence on this vector length should be avoided.*
- With V2X=1, the vector length is double: 1024 bits (128B). Each vector register (V) is 1024-bit (128B), and each vector predicate register (Q) is 128 bits wide.

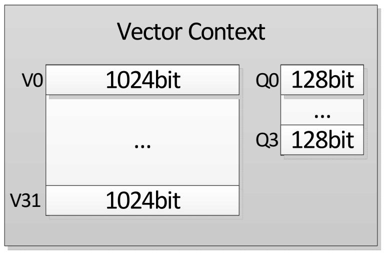


Figure 1-2 Registers using 128B with a vector length of 1024 bits

To minimize effort with porting HVX software to future hardware implementations, software should be written in a way that treats vector length as a constant power of two.

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.

Hexagon hardware threads can be dynamically attached 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 Hexagon 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.
- SSR:XA=6: HVX instructions use vector context 2. Typically, this applies only with V2X=0 (single vector length) which is currently 64B and deprecated.
- SSR:XA=7: HVX instructions use vector context 3. Typically, this applies only with V2X=0 (single vector length) which is currently 64B and deprecated.

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

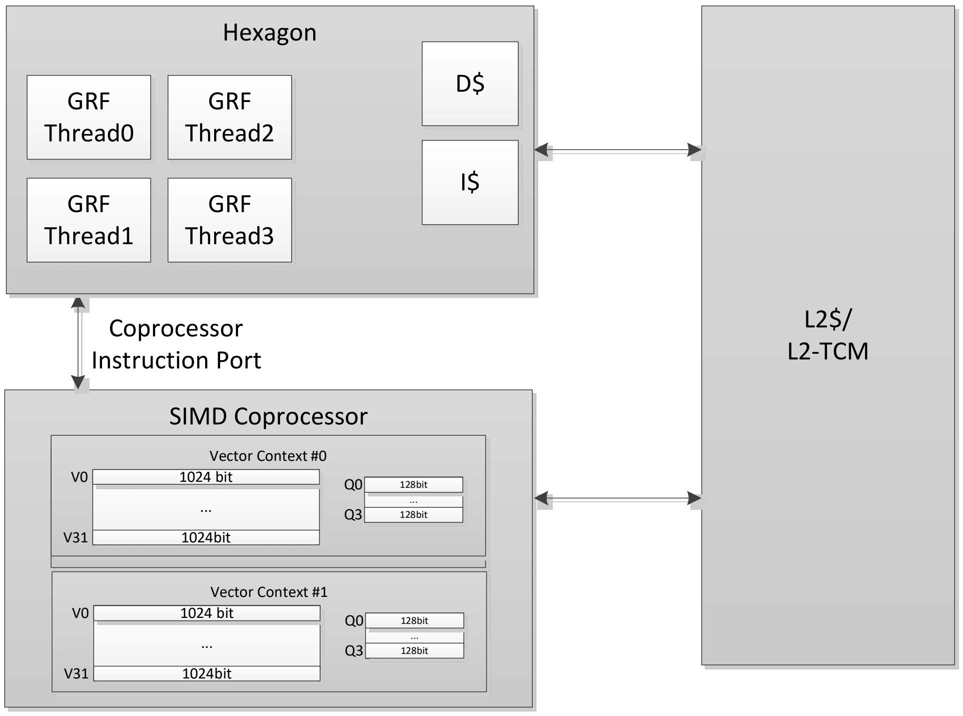


Figure 1-3 Four threads (two double-vector contexts and two scalar threads)

1.2.3 Memory access

The HVX memory instructions (referred to as VMEM instructions) use the Hexagon general registers (R0-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 4 registers each with 1 bit per byte of vector length. These registers provide operands to various compare, mux, and other special instructions.

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

- 32 words (32-bit elements)
- 64 half-words (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.

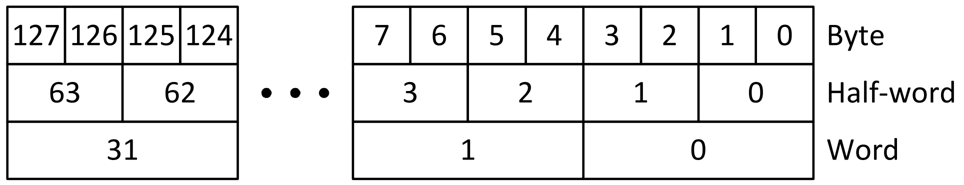


Figure 1-4 1024-bit SIMD register

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

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 16x16 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, submit a case to Qualcomm Technologies at https://support.cdmatech.com.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@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 used 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 (i.e., 64 bits for 64B vector length and 128 bits for 128B vector length).

2.1 Vector data registers

The HVX coprocessor contains 32 vector registers (named V0 through V31). These registers store operand data for all of 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 1024 bits 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 VRF-GRF transfers

Table 2-1 lists the Hexagon instructions used to 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-GRF transfer instructions

Syntax	Behavior	Description
`Rd.w=extractw(Vu,Rs)`	Rd = Vu.uw[Rs&0xF];	Extract word from a vector into Rd with location specified by Rs. Primarily meant for debug.
`Vx.w=insertw(Rss)`	Vx.uw[Rss.w[1]&0xF] = Rss.w[0];	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)
```

In this case, each 32-bit field of V2 and V5 are compared 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 vector predicate instruction is used frequently by the vmux 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 MMU for address translation and protection.

3.1 Alignment

Unlike on the scalar processor an unaligned pointer (i.e., one 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 (lower 6 for 64Bvector length and lower 7 for 128B vector length).

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 also supports a local memory called VTCM. The size of the memory is implementation defined. The size is discoverable from the configuration table defined in the V65 system architecture specification. VTCM needs normal virtual to physical translation just like other memory. This memory has higher performance and lower power. When possible, this memory should be used at least for intermediate vector data. 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. This includes speculative access.
- Scalar VTCM access when the HVX fuse is blown (disabled).
- Load-locked or store-conditional to VTCM.
- Physical load from VTCM while more than one thread is active.
- Accessing VTCM while HVX is not fully powered up or any 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 memory. Each element may specify an independent address that should be read (gather) or written (scatter). Gather for HVX is a vector copy from non-contiguous addresses to an aligned contiguous vector location.

Typically, these instructions are not ordered with following operations. Even accesses from elements of the same scatter or gather instruction are not ordered. The primary ordered case is loading from a gather result. To help with this ordering, a scatter store release instruction is provided, which is not performed until prior scatters and gathers are complete. A load consuming from the scatter store release stalls until those prior scatters and gathers are done.

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

It is illegal for HVX memory instructions (VMEM or scatter/gather) to target device-type memory. If this is done, a VMEM address error exception is raised. 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. Memory should be marked as L2-cacheable for L2 cache data and uncached for data that resides in L2TCM or VTCM.

3.5 Non-temporal

A VMEM instruction can have an optional non-temporal attribute. This is specified in assembly with a ":nt" appendix. Marking an instruction non-temporal indicates to the micro-architecture 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 VMEMU instructions that happen to be 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 any bytes are performed or not. In other words, the state of the Q register is not considered when checking permissions.

3.7 Ordering

To improve performance and reduce hardware overhead, only a subset of memory operations within a thread are ordered. The following pairs of operations are ordered:

Table 3-1 Ordered operations within a thread

Older	Newer
Any store	Any load (same address)
Scalar load/store	Any store
Vector load/store	Any store (same memory type)
Scalar load/store	Scalar load/store (inter-packet)
Vector load	Vector load/store (same memory type)
Vector store	Scalar load/store (same memory type)
Gather write	Any load (same address)
Scatter release	Any load (same address)
Any load/store	Scatter/gather
Anything	Scatter release

Table 3-2 lists pairs of operations within a thread that are not ordered. The barriers listed can be put between the operations to achieve ordering between them.

Table 3-2 Unordered operations in a thread and barriers to order

Older	Newer	Minimal barrier
Scalar load	Scalar load (intra-packet, in data cache)	Separate into different packets
Scalar store	Scalar load (intra-packet, in data cache, different address)	Separate into different packets
Vector load	Scalar load (with external observer - bus)	Invalidate scalar in data cache
Vector store (VTCM)	Vector load (VTCM, different address)	Load (overlap with store)
Vector load/store (VTCM)	Load or store (non-VTCM)	Scalar load/store (VTCM)

Table 3-2 Unordered operations in a thread and barriers to order

Older	Newer	Minimal barrier
Store or vector load (non-VTCM)	Vector load (VTCM)	Scalar load/store (VTCM)
Scatter	Scalar load/store (same address)	Invalidate in data cache Scatter release -> load
Scatter	Scatter/gather Vector load/store Scalar load/store	Scatter release -> load
Gather read	Scatter/gather Load/store	Scatter release -> load
Gather write Scatter release	Scatter/gather Vector store	Scatter release -> load
Gather write Scatter release	Vector load Scalar load/store (different address)	Scatter release -> load
Unknown (non-VTCM)	Unknown (non-VTCM)	SynchT or invalidate data cache
Unknown	Unknown	Invalidate data cache Scatter release -> load

Scatter and gather operations apply to an element at a time in the table above (i.e., no ordering between elements of a scatter/gather instruction).

Ordering is also ensured between two memory operations when the newer operation deterministically depends on the older operation. For example, when a memory write needs the data from a prior memory read, the write is ordered after the read. This applies to the read followed by write operations per element of gathers and scatter accumulates.

3.8 Atomicity

Table 3-3 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 3-3 Atomicity of types of memory accesses

Access type	Atomic size
Scalar A mem-op is 2 accesses	Access size
Aligned vector	Base vector size
Unaligned vector	1B
Scatter	1B
Scatter-accumulate (read-modify-write)	1B A larger read-modify-write may be decomposed into multiple equivalent smaller read-modify-writes.

Table 3-3 Atomicity of types of memory accesses

Access type	Atomic size
Gather read	1B
Gather write	1B

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

3.9 Performance considerations

This section describes best-practices for 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 vector register file (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 it could be hosted in VRF instead.

3.9.2 Use aligned data

VMEMU instruction access multiple L2 cache lines and are expensive in bandwidth and power. Where possible, data structures should be aligned 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

The L2FETCH instruction should be used to pre-populate the L2 with data prior to using VMEM loads.

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

All L2 cacheable data that is used by VMEM should be prefetched, even if it is not used in the computation. Software pipelined loops often overload data that will not be used. 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, data in memory should be arranged so that it is accessed contiguously. For example, instead of repeatedly striding through memory, data might be first tiled, striped, or decimated so that it can be accessed contiguously.

The following techniques achieve better spatial locality in memory to help avoid various 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.
- Set aliasing. Caches hold a number of 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 non-temporal 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 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.

It is common for algorithms to use the vector engine to produce some results that must be further processed 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. The DCFETCH can be executed 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 the resulting latency can be long, therefore extra care is required to avoid stalls. The following techniques should 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. For V65, address bits [7:1] are important to avoid conflicts. Ideally this would apply 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 given thread (for V65).
- Defer loading from a gather result or a scatter store release. If the in-flight scatters and gathers (including from other threads) avoid conflicts, a distance of 12 or more packets should be sufficient. Double that distance is needed if the addresses of in-flight accesses are not correlated.

Table 3-4 Peak scatter/gather performance for V65

Operation	Addressing	Vector Bandwidth (per packet)	Latency (packets)
Scatter	Conflict-free	2/3	8
Gather	Conflict-free	1/2	12
Scatter	Random	1/3	16
Gather	Random	1/6	32

4 Vector Instructions

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

Section 4.7 gives a summary of all hexagon slot, HVX resource, and instruction latency for all instruction categories.

4.1 VLIW packing rules

HVX provides six 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 endloop. The instructions inside the packet must obey the packet grouping rules described in section 4.1.3.

NOTE Invalid packet combinations should be checked and flagged by the assembler. In the case that an invalid packet is executed, 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

The following table 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	Any one of shift, permute, or multiply
Aligned memory	Any one of shift, permute, or multiply and one of load or store
Aligned memory (.tmp/.new)	Load or store only
Scatter (single vector indexing)	Store and any 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 any one of shift, permute, or multiply
Gather (double vector indexing)	Load and either shift and permute or both multiply

4.1.3 Vector instruction

In addition to vector resource assignment, vector instructions also map to certain Hexagon slots. A special subset of ALU instructions that require either the full 32 bits of the scalar Rt register (or 64 bits of Rtt) are mapped to slots 2 and 3. These include lookup table, splat, insert, and add/sub with Rt.

Table 4-2 HVX

Instruction	Used Hexagon Slots	Additional Restriction
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

Table 4-2 HVX

Instruction	Used Hexagon Slots	Additional Restriction
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 will not be overwritten). Since the "load-temp" instruction does not write to the register file, it does not consume a vector ALU resource.

```
{  V2.tmp = vmem(R1+#1)         // Data loaded into 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 is written 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.

```
vmem(R1+#1)= V20.new // Store V20 that was generate 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 partial byte-enabled store can be issued and controlled with a vector predicate register

```
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 non-contiguous memory access of vector data. With scatter and gather, each element can independently index into a region of memory. This allows for applications that would not otherwise map well to the SIMD parallelism that HVX provides.

A scatter transfers data from a contiguous vector to non-contiguous memory locations. Similarly, gather transfers data from non-contiguous memory locations to a contiguous vector. In HVX, scatter is a vector register to non-contiguous memory transfer and gather is a non-contiguous 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 all non-contiguous accesses must be in. 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. The following table describes the three sources that specify the non-contiguous accesses of a scatter or gather:

Table 4-3 Sources used for non-contiguous accesses: (Rt, Mu, Vv)

Source	Meaning
Rt	Base address of the region
Mu	Byte offset of last valid byte of the region (i.e. 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, the following table describes the basic forms of scatter and gather instructions:

Table 4-4 Basic scatter and gather instructions

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

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	Non-memory
VMEM St	Non-memory
VMEM Ld	Scalar Ld
Scalar St	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	Non-memory
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 is divided into four histogram tables each of 256 entries (32 registers by 8 halfwords). A line is fetched from memory via a temporary VMEM load instruction. 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. All the registers must be cleared before use by the programmer.

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 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 so that a packet can be issued and completed on every context cycle. The contexts are time interleaved to share the hardware such that using all contexts may be required to reach peak compute bandwidth.

With a few exceptions (i.e., histogram and extract), all results of all packets are generated 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 register that is being shifted or counted is considered early source. Accumulators are not early sources.
- Input to permute instructions. Only registers that are being permuted are considered early source (not an acccumulator).
- Unaligned store data is an early source.

If an early source register is produced in the previous vector packet, an interlock stall can be incurred. Software should strive to schedule an intervening packet between the producer and an early source consumer.

The following example shows various 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
```

4.7 Slot/resource/latency summary

Table 4-6 summarizes the Hexagon slot, HVX resource, and latency requirements for all HVX instruction types.

Table 4-6 HVX slot/resource/latency summary

		Core slots				Vector resources						Input
Category	Variation	3	2	1	0	ld	mpy	mpy	shift	xlane	st	latency
ALU	1 vec	any					any					1
	2 vec	any					either pair					1
	Rt	either					either					1
Abs-diff	1 vec	either					either					2
	2 vec	either					▨	▨				2
Multiply	by 8b; 1 vec	either					either					2
	by 8b; 2 vec	either					▨	▨				2
	by 16b	either					▨	▨				2
Cross-lane	1 vec	any								▨		2
	2 vec	any							▨	▨		2
Shift or count	1 vec	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 vec			▨		▨	any					1
	2 vec			▨		▨	either pair					1
scatter	1 vec				▨		any				▨	1
	2 vec				▨		either pair				▨	1
histogram (needs .tmp load)		any					▨	▨	▨	▨	▨	2
extract		░	░	░	▨	▨	▨	▨	▨	▨	▨	1

5 Instruction Set

This chapter describes the instruction set for version 6 of the Hexagon processor. The HVX instruction class includes instructions which perform vector operations on 512- or 1024-bit data.

The 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
- Any C intrinsic functions that provide access to the instruction
- Instruction encoding

5.1 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-V31, and all four instruction pipes. The instruction also takes four execution packets to complete. The basic unit of the histogram instruction is a 128-bit wide slice - there can be 4 or 8 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 65535 8-bit elements of the same value to be accumulated. Each histogram is 128 bits wide and 32 elements deep, giving 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 then 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 are processed 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 are 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.

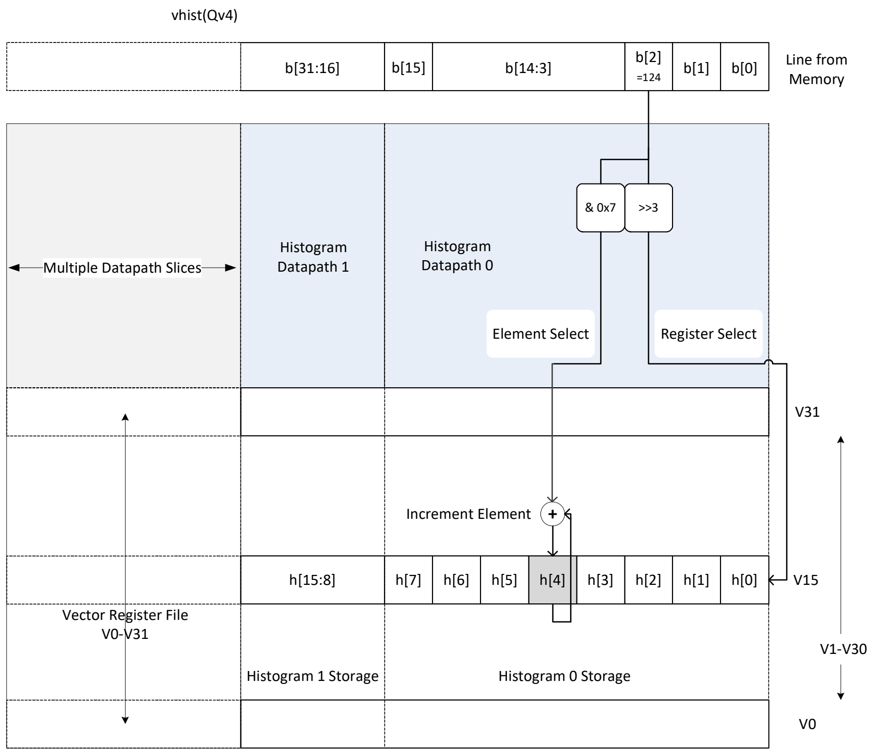
vhist is the only instruction that occupies all pipes and resources.

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

vhist 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 eight-bit element in position two of the source data. The value is 124, the register number assigned to this is 124 >> 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 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(bytes per vector)/4(packets per operation) * 4(threads) = 64 bytes per clock cycle per core, assuming all threads are performing histogramming.



Syntax	Behavior

`vhist`	`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);`
	`        tmp.uh[(128/16)*lane+(element)]++;`
	`        WRITE_EXT_VREG(regno,tmp,EXT_NEW);`
	`    }`
	`}`

`vhist(Qv4)`	`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);`
	`        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	**P**	**P**	1	-	0	0	0	-	1	0	0	-	-	-	-	-	vhist
0	0	0	1	1	1	1	0	**v**	**v**	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-V31, and all four instruction pipes. The instruction also 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, giving a total of 256 histogram bins.

For vwhist128, 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, giving 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 five 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 bye is ignored.

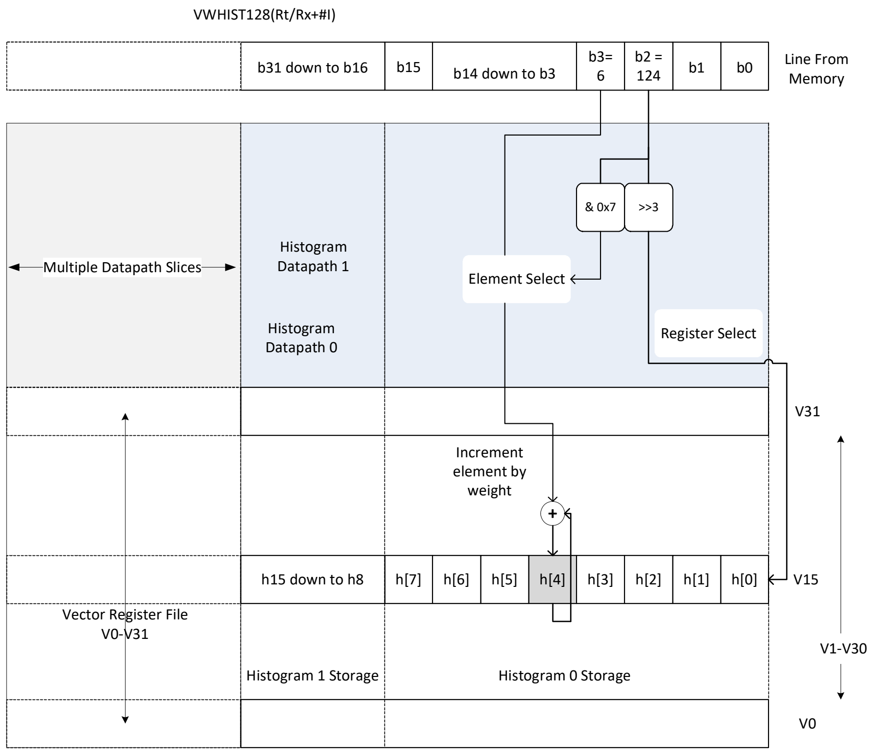
The register file entry is then incremented by corresponding weight from the odd byte.

Like vhist, vwhist also occupies all pipes and resources.

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

vwhist 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 two of the source data with corresponding weight in byte position 3.



The value is 124, the register number assigned to this is 124 >> 3 = V15, and the element

Syntax	Behavior
`vwhist128`	`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);` `tmp.uw[elindex] = (tmp.uw[elindex] + weight);` `WRITE_EXT_VREG(vindex,tmp,EXT_NEW);` `}`
`vwhist128(#u1)`	`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);` `if ((bucket & 1) == #u) tmp.uw[elindex] =` `(tmp.uw[elindex] + weight);` `WRITE_EXT_VREG(vindex,tmp,EXT_NEW);` `}`
`vwhist128(Qv4)`	`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);` `if (QvV[2*i]) tmp.uw[elindex] = (tmp.uw[elindex] +` `weight);` `WRITE_EXT_VREG(vindex,tmp,EXT_NEW);` `}`
`vwhist128(Qv4,#u1)`	`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);` `if (((bucket & 1) == #u) && QvV[2*i]) tmp.uw[elindex] =` `(tmp.uw[elindex] + weight);` `WRITE_EXT_VREG(vindex,tmp,EXT_NEW);` `}`
`vwhist256`	`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);` `tmp.uh[elindex] = (tmp.uh[elindex] + weight);` `WRITE_EXT_VREG(vindex,tmp,EXT_NEW);` `}`

Syntax	Behavior

vwhist256(Qv4)	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);
	    if (QvV[2*i]) tmp.uh[elindex] = (tmp.uh[elindex] +
	weight);
	    WRITE_EXT_VREG(vindex,tmp,EXT_NEW);
	}

vwhist256(Qv4):sat	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);
	    if (QvV[2*i]) tmp.uh[elindex] = $usat_{16}$(tmp.uh[elindex]
	+ weight);
	    WRITE_EXT_VREG(vindex,tmp,EXT_NEW);
	}

vwhist256:sat	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);
	    tmp.uh[elindex] = $usat_{16}$(tmp.uh[elindex] + weight);
	    WRITE_EXT_VREG(vindex,tmp,EXT_NEW);
	}

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(bytes per vector)/4(packets per operation) * 4(threads) = 64 bytes per clock cycle per core, assuming all threads are performing histogramming.

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	**P**	**P**	1	-	0	0	1	0	1	0	0	-	-	-	-	-	vwhist256
0	0	0	1	1	1	1	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	**P**	**P**	1	-	0	1	0	-	1	0	0	-	-	-	-	-	vwhist128

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	-	-	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	-	-	-	1	0	**P**	**P**	1	-	-	0	1	0	1	0	0	-	-	-	-	-	vwhist256(Qv4)
0	0	0	1	1	1	1	0	**v**	**v**	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	-	-	-	1	0	**P**	**P**	1	-	-	1	0	-	1	0	0	-	-	-	-	-	vwhist128(Qv4)
0	0	0	1	1	1	1	0	**v**	**v**	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

5.2 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 like vsat or vround. Forms that match word-to-halfword and halfword-to-byte shuffling are available.

Syntax	Behavior

`Qd4.b=vshuffe(Qs4.h,Qt4.h)`	`for (i = 0; i < VELEM(8); i++) {`
	`    QdV[i]=(i & 1) ? QsV[i-1] : QtV[i];`
	`}`

`Qd4.h=vshuffe(Qs4.w,Qt4.w)`	`for (i = 0; i < VELEM(8); i++) {`
	`    QdV[i]=(i & 2) ? QsV[i-2] : QtV[i];`
	`}`

`Qd4=and(Qs4,[!]Qt4)`	`for (i = 0; i < VELEM(8); i++) {`
	`    QdV[i]=QsV[i] && [!]QtV[i];`
	`}`

`Qd4=or(Qs4,[!]Qt4)`	`for (i = 0; i < VELEM(8); i++) {`
	`    QdV[i]=QsV[i] || [!]QtV[i];`
	`}`

`Qd4=xor(Qs4,Qt4)`	`for (i = 0; i < VELEM(8); i++) {`
	`    QdV[i]=QsV[i] ^ QtV[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

`Qd4.b=vshuffe(Qs4.h,Qt4.h)`	`HVX_VectorPred Q6_Qb_vshuffe_QhQh(HVX_VectorPred Qs, HVX_VectorPred Qt)`
`Qd4.h=vshuffe(Qs4.w,Qt4.w)`	`HVX_VectorPred Q6_Qh_vshuffe_QwQw(HVX_VectorPred Qs, HVX_VectorPred Qt)`
`Qd4=and(Qs4,!Qt4)`	`HVX_VectorPred Q6_Q_and_QQn(HVX_VectorPred Qs, HVX_VectorPred Qt)`
`Qd4=and(Qs4,Qt4)`	`HVX_VectorPred Q6_Q_and_QQ(HVX_VectorPred Qs, HVX_VectorPred Qt)`
`Qd4=or(Qs4,!Qt4)`	`HVX_VectorPred Q6_Q_or_QQn(HVX_VectorPred Qs, HVX_VectorPred Qt)`
`Qd4=or(Qs4,Qt4)`	`HVX_VectorPred Q6_Q_or_QQ(HVX_VectorPred Qs, HVX_VectorPred Qt)`
`Qd4=xor(Qs4,Qt4)`	`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	**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	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

`Vdd=vcombine(Vu,Vv)`	`for (i = 0; i < VELEM(8); i++) {`
	`    Vdd.v[0].ub[i] = Vv.ub[i];`
	`    Vdd.v[1].ub[i] = Vu.ub[i];`
	`}`

`if ([!]Ps) Vdd=vcombine(Vu,Vv)`	`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;`
	`}`

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

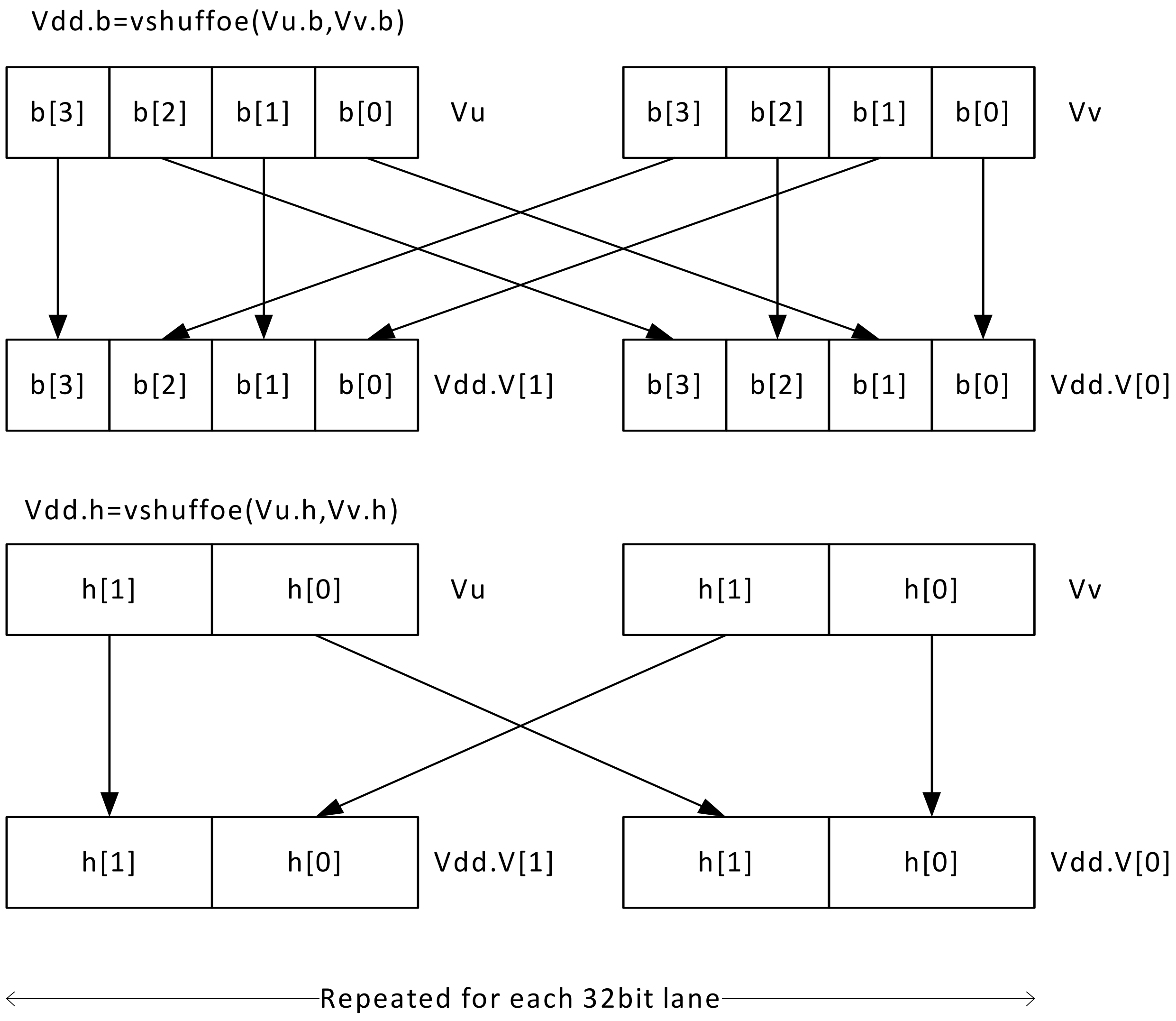
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse			u5						s2		d5					
0	0	0	1	1	0	1	0	0	1	0	v	v	v	v	v	P	P	-	u	u	u	u	u	-	s	s	d	d	d	d	d	if (!Ps) Vdd=vcombine(Vu,Vv)
0	0	0	1	1	0	1	0	0	1	1	v	v	v	v	v	P	P	-	u	u	u	u	u	-	s	s	d	d	d	d	d	if (Ps)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

Field name	Description
`s2`	Field to encode register s
`u5`	Field to encode register u
`v5`	Field to encode register v

In-lane shuffle

vshuffoe 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	Behavior

`Vdd.b=vshuffoe(Vu.b,Vv.b)`	`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];`
	`}`

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

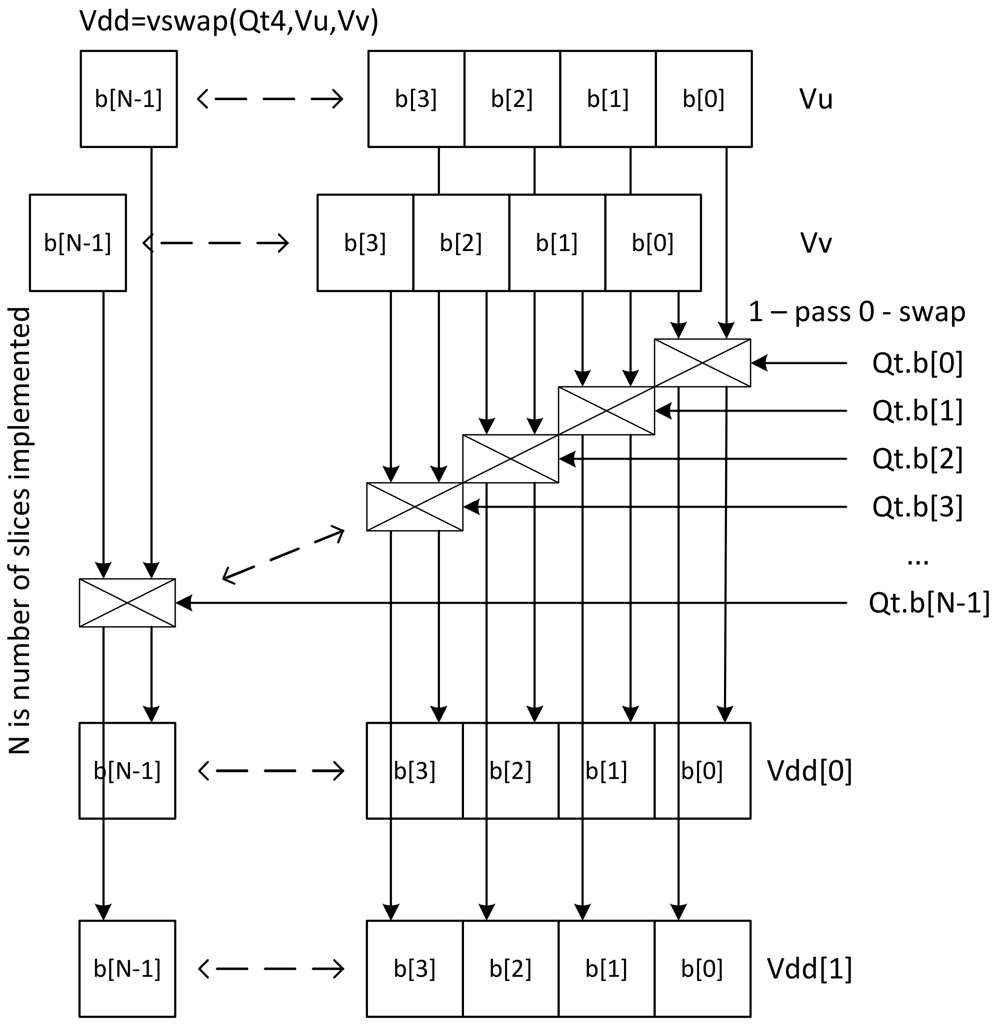
Encoding

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	Behavior

`Vdd=vswap(Qt4,Vu,Vv)`	`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];`
	`}`

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(Qt4,Vu,Vv)`	`HVX_VectorPair Q6_W_vswap_QVV(HVX_VectorPred Qt, HVX_Vector Vu, HVX_Vector Vv)`

Encoding

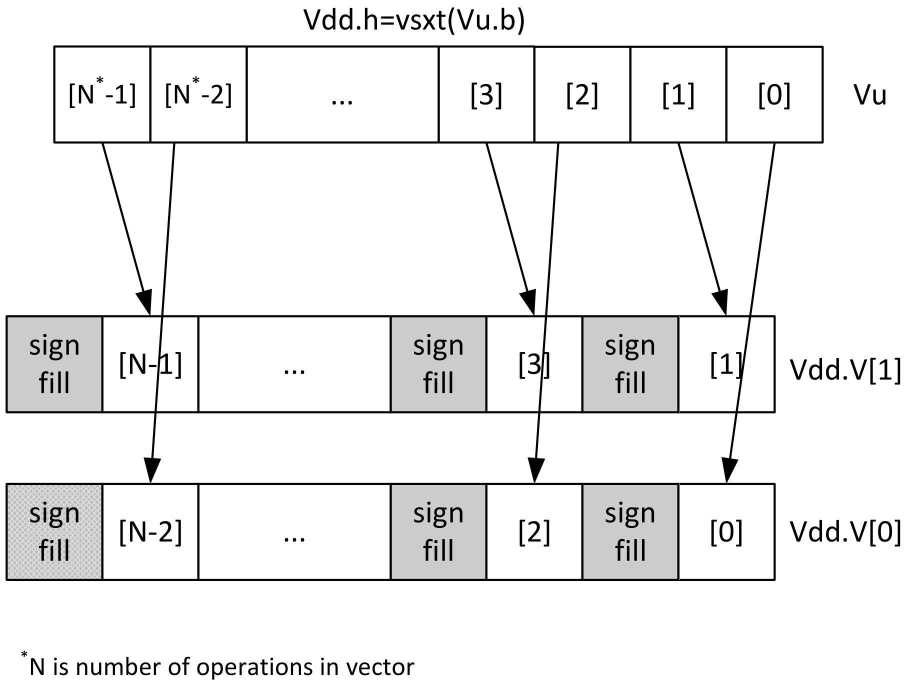
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse			u5						t2		d5					
0	0	0	1	1	1	1	0	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	-	t	t	d	d	d	d	d	Vdd=vswap(Qt4,Vu,Vv)

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
`v5`	Field to encode register v

Sign/Zero extension

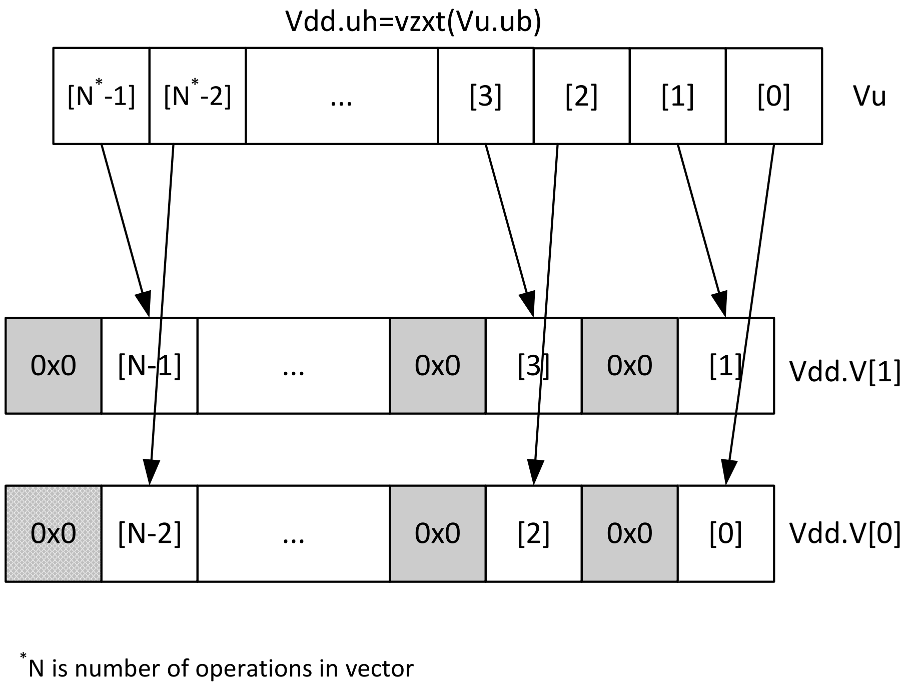
Perform sign extension on each even element in Vu, and place it in the even destination vector register Vdd[0]. Odd elements are sign-extended and placed in the odd destination vector register Vdd[1]. Bytes are converted to halfwords, and halfwords are converted to words.

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



Perform zero extension on each even element in Vu, and place it in the even destination vector register Vdd[0]. Odd elements are zero-extended and placed in the odd destination vector register Vdd[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.



Syntax	Behavior

`Vdd.h=vsxt(Vu.b)`	`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];`
	`}`

`Vdd.uh=vzxt(Vu.ub)`	`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];`
	`}`

`Vdd.uw=vzxt(Vu.uh)`	`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];`
	`}`

`Vdd.w=vsxt(Vu.h)`	`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];`
	`}`

`Vdd=vsxtb(Vu)`	`Assembler mapped to: "Vdd.h=vsxt(Vu.b)"`
`Vdd=vsxth(Vu)`	`Assembler mapped to: "Vdd.w=vsxt(Vu.h)"`
`Vdd=vzxtb(Vu)`	`Assembler mapped to: "Vdd.uh=vzxt(Vu.ub)"`
`Vdd=vzxth(Vu)`	`Assembler mapped to: "Vdd.uw=vzxt(Vu.uh)"`

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

`Vdd.b=vadd(Vuu.b,Vvv.b)[:sat]`	`for (i = 0; i < VELEM(8); i++) {`
	`    Vdd.v[0].b[i] = [sat`$_8$`](Vuu.v[0].b[i]+Vvv.v[0].b[i]);`
	`    Vdd.v[1].b[i] = [sat`$_8$`](Vuu.v[1].b[i]+Vvv.v[1].b[i]);`
	`}`

`Vdd.b=vsub(Vuu.b,Vvv.b)[:sat]`	`for (i = 0; i < VELEM(8); i++) {`
	`    Vdd.v[0].b[i] = [sat`$_8$`](Vuu.v[0].b[i]-Vvv.v[0].b[i]);`
	`    Vdd.v[1].b[i] = [sat`$_8$`](Vuu.v[1].b[i]-Vvv.v[1].b[i]);`
	`}`

`Vdd.h=vadd(Vuu.h,Vvv.h)[:sat]`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vdd.v[0].h[i] = [sat`$_{16}$`](Vuu.v[0].h[i]+Vvv.v[0].h[i]);`
	`    Vdd.v[1].h[i] = [sat`$_{16}$`](Vuu.v[1].h[i]+Vvv.v[1].h[i]);`
	`}`

`Vdd.h=vsub(Vuu.h,Vvv.h)[:sat]`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vdd.v[0].h[i] = [sat`$_{16}$`](Vuu.v[0].h[i]-Vvv.v[0].h[i]);`
	`    Vdd.v[1].h[i] = [sat`$_{16}$`](Vuu.v[1].h[i]-Vvv.v[1].h[i]);`
	`}`

`Vdd.ub=vadd(Vuu.ub,Vvv.ub):sat`	`for (i = 0; i < VELEM(8); i++) {`
	`    Vdd.v[0].ub[i] = usat`$_8$`(Vuu.v[0].ub[i]+Vvv.v[0].ub[i]);`
	`    Vdd.v[1].ub[i] = usat`$_8$`(Vuu.v[1].ub[i]+Vvv.v[1].ub[i]);`
	`}`

`Vdd.ub=vsub(Vuu.ub,Vvv.ub):sat`	`for (i = 0; i < VELEM(8); i++) {`
	`    Vdd.v[0].ub[i] = usat`$_8$`(Vuu.v[0].ub[i]-Vvv.v[0].ub[i]);`
	`    Vdd.v[1].ub[i] = usat`$_8$`(Vuu.v[1].ub[i]-Vvv.v[1].ub[i]);`
	`}`

`Vdd.uh=vadd(Vuu.uh,Vvv.uh):sat`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vdd.v[0].uh[i] = usat`$_{16}$`(Vuu.v[0].uh[i]+Vvv.v[0].uh[i]);`
	`    Vdd.v[1].uh[i] = usat`$_{16}$`(Vuu.v[1].uh[i]+Vvv.v[1].uh[i]);`
	`}`

Syntax	Behavior

`Vdd.uh=vsub(Vuu.uh,Vvv.uh):sat`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vdd.v[0].uh[i] = usat`$_{16}$`(Vuu.v[0].uh[i]-Vvv.v[0].uh[i]);`
	`    Vdd.v[1].uh[i] = usat`$_{16}$`(Vuu.v[1].uh[i]-Vvv.v[1].uh[i]);`
	`}`

`Vdd.uw=vadd(Vuu.uw,Vvv.uw):sat`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vdd.v[0].uw[i] = usat`$_{32}$`(Vuu.v[0].uw[i]+Vvv.v[0].uw[i]);`
	`    Vdd.v[1].uw[i] = usat`$_{32}$`(Vuu.v[1].uw[i]+Vvv.v[1].uw[i]);`
	`}`

`Vdd.uw=vsub(Vuu.uw,Vvv.uw):sat`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vdd.v[0].uw[i] = usat`$_{32}$`(Vuu.v[0].uw[i]-Vvv.v[0].uw[i]);`
	`    Vdd.v[1].uw[i] = usat`$_{32}$`(Vuu.v[1].uw[i]-Vvv.v[1].uw[i]);`
	`}`

`Vdd.w=vadd(Vuu.w,Vvv.w)[:sat]`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vdd.v[0].w[i] = [sat`$_{32}$`](Vuu.v[0].w[i]+Vvv.v[0].w[i]);`
	`    Vdd.v[1].w[i] = [sat`$_{32}$`](Vuu.v[1].w[i]+Vvv.v[1].w[i]);`
	`}`

`Vdd.w=vsub(Vuu.w,Vvv.w)[:sat]`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vdd.v[0].w[i] = [sat`$_{32}$`](Vuu.v[0].w[i]-Vvv.v[0].w[i]);`
	`    Vdd.v[1].w[i] = [sat`$_{32}$`](Vuu.v[1].w[i]-Vvv.v[1].w[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.b=vadd(Vuu.b,Vvv.b)`	`HVX_VectorPair Q6_Wb_vadd_WbWb(HVX_VectorPair Vuu, HVX_VectorPair Vvv)`
`Vdd.b=vadd(Vuu.b,Vvv.b):sat`	`HVX_VectorPair Q6_Wb_vadd_WbWb_sat(HVX_VectorPair Vuu, HVX_VectorPair Vvv)`
`Vdd.b=vsub(Vuu.b,Vvv.b)`	`HVX_VectorPair Q6_Wb_vsub_WbWb(HVX_VectorPair Vuu, HVX_VectorPair Vvv)`
`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

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	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

5.3 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 combinations are implemented: !Qs.

Syntax	Behavior

`Qd4=not(Qs4)`	`for (i = 0; i < VELEM(8); i++) {`
	`    QdV[i]=!QsV[i];`
	`}`

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

Notes

- This instruction can use any HVX resource.

Intrinsics

`Qd4=not(Qs4)`	`HVX_VectorPred Q6_Q_not_Q(HVX_VectorPred Qs)`

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	Behavior

`Vd=vand([!]Qv4,Vu)`	`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	-	-	-	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	-	-	-	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)`	`for (i = 0; i < VELEM(8); i++) {`
	`    Vd.b[i] = (Vu.b[i] > Vv.b[i]) ? Vu.b[i] : Vv.b[i];`
	`}`

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

`Vd.h=vmax(Vu.h,Vv.h)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.h[i] = (Vu.h[i] > Vv.h[i]) ? Vu.h[i] : Vv.h[i];`
	`}`

`Vd.h=vmin(Vu.h,Vv.h)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.h[i] = (Vu.h[i] < Vv.h[i]) ? Vu.h[i] : Vv.h[i];`
	`}`

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

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

`Vd.uh=vmax(Vu.uh,Vv.uh)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.uh[i] = (Vu.uh[i] > Vv.uh[i]) ? Vu.uh[i] : Vv.uh[i];`
	`}`

`Vd.uh=vmin(Vu.uh,Vv.uh)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.uh[i] = (Vu.uh[i] < Vv.uh[i]) ? Vu.uh[i] : Vv.uh[i];`
	`}`

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

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

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

Notes

- This instruction can use any HVX resource.

Intrinsics

`Vd.b=vmax(Vu.b,Vv.b)`	`HVX_Vector Q6_Vb_vmax_VbVb(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.b=vmin(Vu.b,Vv.b)`	`HVX_Vector Q6_Vb_vmin_VbVb(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.h=vmax(Vu.h,Vv.h)`	`HVX_Vector Q6_Vh_vmax_VhVh(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.h=vmin(Vu.h,Vv.h)`	`HVX_Vector Q6_Vh_vmin_VhVh(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.ub=vmax(Vu.ub,Vv.ub)`	`HVX_Vector Q6_Vub_vmax_VubVub(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.ub=vmin(Vu.ub,Vv.ub)`	`HVX_Vector Q6_Vub_vmin_VubVub(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.uh=vmax(Vu.uh,Vv.uh)`	`HVX_Vector Q6_Vuh_vmax_VuhVuh(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.uh=vmin(Vu.uh,Vv.uh)`	`HVX_Vector Q6_Vuh_vmin_VuhVuh(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.w=vmax(Vu.w,Vv.w)`	`HVX_Vector Q6_Vw_vmax_VwVw(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.w=vmin(Vu.w,Vv.w)`	`HVX_Vector Q6_Vw_vmin_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	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)

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 maximum negative value overflow case.

Syntax	Behavior

`Vd.b=vabs(Vu.b)[:sat]`	`for (i = 0; i < VELEM(8); i++) {`
	`    Vd.b[i] = [sat`$_8$`](ABS(Vu.b[i]));`
	`}`

`Vd.h=vabs(Vu.h)[:sat]`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.h[i] = [sat`$_{16}$`](ABS(Vu.h[i]));`
	`}`

`Vd.ub=vabs(Vu.b)`	`Assembler mapped to: "Vd.b=vabs(Vu.b)"`
`Vd.uh=vabs(Vu.h)`	`Assembler mapped to: "Vd.h=vabs(Vu.h)"`
`Vd.uw=vabs(Vu.w)`	`Assembler mapped to: "Vd.w=vabs(Vu.w)"`

`Vd.w=vabs(Vu.w)[:sat]`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.w[i] = [sat`$_{32}$`](ABS(Vu.w[i]));`
	`}`

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

Notes

- This instruction can use any HVX resource.
- This instruction may not work correctly in Napali V1.

Intrinsics

`Vd.b=vabs(Vu.b)`	`HVX_Vector Q6_Vb_vabs_Vb(HVX_Vector Vu)`
`Vd.b=vabs(Vu.b):sat`	`HVX_Vector Q6_Vb_vabs_Vb_sat(HVX_Vector Vu)`
`Vd.h=vabs(Vu.h)`	`HVX_Vector Q6_Vh_vabs_Vh(HVX_Vector Vu)`
`Vd.h=vabs(Vu.h):sat`	`HVX_Vector Q6_Vh_vabs_Vh_sat(HVX_Vector Vu)`
`Vd.w=vabs(Vu.w)`	`HVX_Vector Q6_Vw_vabs_Vw(HVX_Vector Vu)`
`Vd.w=vabs(Vu.w):sat`	`HVX_Vector Q6_Vw_vabs_Vw_sat(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**	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

Vd.b=vadd(Vu.b,Vv.b)[:sat]	for (i = 0; i < VELEM(8); i++) {
	    Vd.b[i] = [sat$_8$](Vu.b[i]+Vv.b[i]);
	}

Vd.b=vsub(Vu.b,Vv.b)[:sat]	for (i = 0; i < VELEM(8); i++) {
	    Vd.b[i] = [sat$_8$](Vu.b[i]-Vv.b[i]);
	}

Vd.h=vadd(Vu.h,Vv.h)[:sat]	for (i = 0; i < VELEM(16); i++) {
	    Vd.h[i] = [sat$_{16}$](Vu.h[i]+Vv.h[i]);
	}

Vd.h=vsub(Vu.h,Vv.h)[:sat]	for (i = 0; i < VELEM(16); i++) {
	    Vd.h[i] = [sat$_{16}$](Vu.h[i]-Vv.h[i]);
	}

Vd.ub=vadd(Vu.ub,Vv.b):sat	for (i = 0; i < VELEM(8); i++) {
	    Vd.ub[i] = usat$_8$(Vu.ub[i] + Vv.b[i]);
	}

Vd.ub=vadd(Vu.ub,Vv.ub):sat	for (i = 0; i < VELEM(8); i++) {
	    Vd.ub[i] = usat$_8$(Vu.ub[i]+Vv.ub[i]);
	}

Vd.ub=vsub(Vu.ub,Vv.b):sat	for (i = 0; i < VELEM(8); i++) {
	    Vd.ub[i] = usat$_8$(Vu.ub[i] - Vv.b[i]);
	}

Vd.ub=vsub(Vu.ub,Vv.ub):sat	for (i = 0; i < VELEM(8); i++) {
	    Vd.ub[i] = usat$_8$(Vu.ub[i]-Vv.ub[i]);
	}

Vd.uh=vadd(Vu.uh,Vv.uh):sat	for (i = 0; i < VELEM(16); i++) {
	    Vd.uh[i] = usat$_{16}$(Vu.uh[i]+Vv.uh[i]);
	}

Vd.uh=vsub(Vu.uh,Vv.uh):sat	for (i = 0; i < VELEM(16); i++) {
	    Vd.uh[i] = usat$_{16}$(Vu.uh[i]-Vv.uh[i]);
	}

Vd.uw=vadd(Vu.uw,Vv.uw):sat	for (i = 0; i < VELEM(32); i++) {
	    Vd.uw[i] = usat$_{32}$(Vu.uw[i]+Vv.uw[i]);
	}

Syntax	Behavior
Vd.uw=vsub(Vu.uw,Vv.uw):sat	for (i = 0; i < VELEM(32); i++) { Vd.uw[i] = usat$_{32}$(Vu.uw[i]-Vv.uw[i]); }
Vd.w=vadd(Vu.w,Vv.w)[:sat]	for (i = 0; i < VELEM(32); i++) { Vd.w[i] = [sat$_{32}$](Vu.w[i]+Vv.w[i]); }
Vd.w=vadd(Vu.w,Vv.w,Qx4):carry	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]); }
Vd.w=vsub(Vu.w,Vv.w)[:sat]	for (i = 0; i < VELEM(32); i++) { Vd.w[i] = [sat$_{32}$](Vu.w[i]-Vv.w[i]); }
Vd.w=vsub(Vu.w,Vv.w,Qx4):carry	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]); }

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

Notes

- This instruction can use any HVX resource.

Intrinsics

Vd.b=vadd(Vu.b,Vv.b)	HVX_Vector Q6_Vb_vadd_VbVb(HVX_Vector Vu, HVX_Vector Vv)
Vd.b=vadd(Vu.b,Vv.b):sat	HVX_Vector Q6_Vb_vadd_VbVb_sat(HVX_Vector Vu, HVX_Vector Vv)
Vd.b=vsub(Vu.b,Vv.b)	HVX_Vector Q6_Vb_vsub_VbVb(HVX_Vector Vu, HVX_Vector Vv)
Vd.b=vsub(Vu.b,Vv.b):sat	HVX_Vector Q6_Vb_vsub_VbVb_sat(HVX_Vector Vu, HVX_Vector Vv)
Vd.h=vadd(Vu.h,Vv.h)	HVX_Vector Q6_Vh_vadd_VhVh(HVX_Vector Vu, HVX_Vector Vv)
Vd.h=vadd(Vu.h,Vv.h):sat	HVX_Vector Q6_Vh_vadd_VhVh_sat(HVX_Vector Vu, HVX_Vector Vv)
Vd.h=vsub(Vu.h,Vv.h)	HVX_Vector Q6_Vh_vsub_VhVh(HVX_Vector Vu, HVX_Vector Vv)
Vd.h=vsub(Vu.h,Vv.h):sat	HVX_Vector Q6_Vh_vsub_VhVh_sat(HVX_Vector Vu, HVX_Vector Vv)
Vd.ub=vadd(Vu.ub,Vv.b):sat	HVX_Vector Q6_Vub_vadd_VubVb_sat(HVX_Vector Vu, HVX_Vector Vv)
Vd.ub=vadd(Vu.ub,Vv.ub):sat	HVX_Vector Q6_Vub_vadd_VubVub_sat(HVX_Vector Vu, HVX_Vector Vv)

`Vd.ub=vsub(Vu.ub,Vv.b):sat`	`HVX_Vector Q6_Vub_vsub_VubVb_sat(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.ub=vsub(Vu.ub,Vv.ub):sat`	`HVX_Vector Q6_Vub_vsub_VubVub_sat(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.uh=vadd(Vu.uh,Vv.uh):sat`	`HVX_Vector Q6_Vuh_vadd_VuhVuh_sat(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.uh=vsub(Vu.uh,Vv.uh):sat`	`HVX_Vector Q6_Vuh_vsub_VuhVuh_sat(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.uw=vadd(Vu.uw,Vv.uw):sat`	`HVX_Vector Q6_Vuw_vadd_VuwVuw_sat(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.uw=vsub(Vu.uw,Vv.uw):sat`	`HVX_Vector Q6_Vuw_vsub_VuwVuw_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
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								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	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

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	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
`x2`	Field to encode register x

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

`Vd=vand(Vu,Vv)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.uh[i] = Vu.uh[i] & Vv.h[i];`
	`}`

`Vd=vnot(Vu)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.uh[i] = ~Vu.uh[i];`
	`}`

`Vd=vor(Vu,Vv)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.uh[i] = Vu.uh[i] | Vv.h[i];`
	`}`

`Vd=vxor(Vu,Vv)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.uh[i] = Vu.uh[i] ^ Vv.h[i];`
	`}`

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

Notes

- This instruction can use any HVX resource.

Intrinsics

`Vd=vand(Vu,Vv)`	`HVX_Vector Q6_V_vand_VV(HVX_Vector Vu, HVX_Vector Vv)`
`Vd=vnot(Vu)`	`HVX_Vector Q6_V_vnot_V(HVX_Vector Vu)`
`Vd=vor(Vu,Vv)`	`HVX_Vector Q6_V_vor_VV(HVX_Vector Vu, HVX_Vector Vv)`
`Vd=vxor(Vu,Vv)`	`HVX_Vector Q6_V_vxor_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								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
`v5`	Field to encode register v

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

`if ([!]Ps) Vd=Vu`	`if ([!]Ps[0]) {`
	`    for (i = 0; i < VELEM(8); i++) {`
	`        Vd.ub[i] = Vu.ub[i];`
	`    }`
	`} else {`
	`    NOP;`
	`}`

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

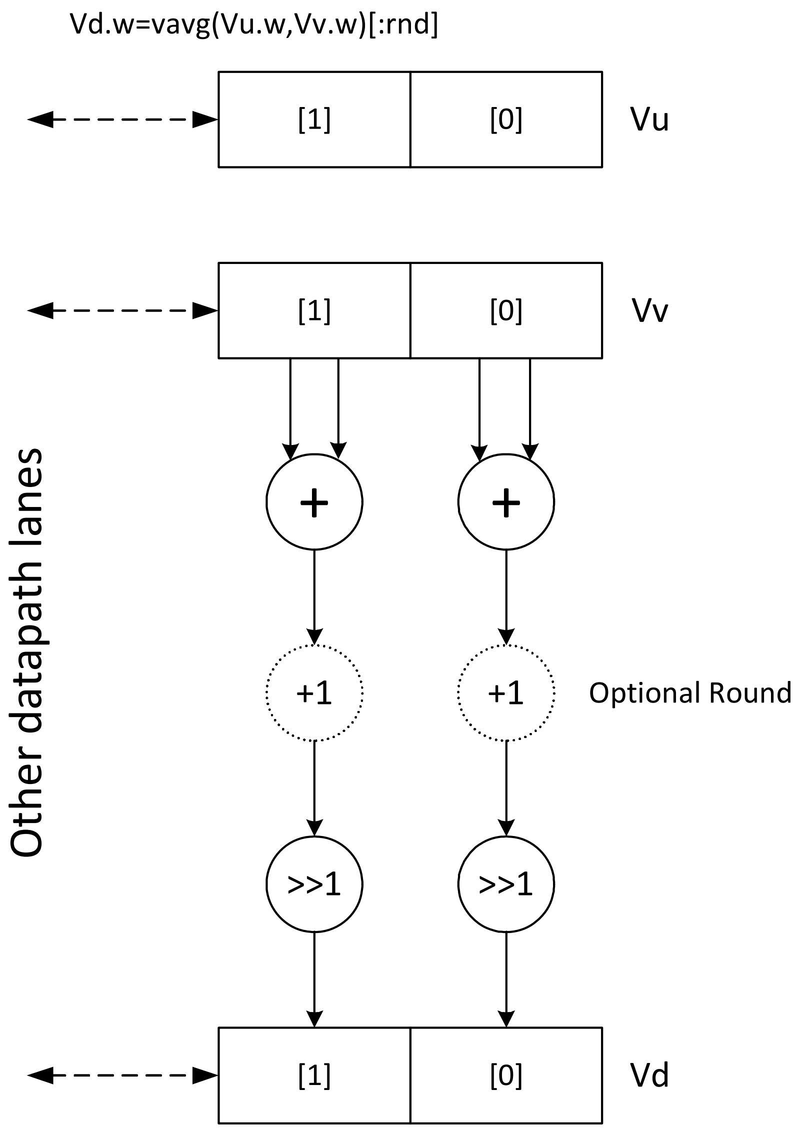
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse			u5						s2		d5					
0	0	0	1	1	0	1	0	0	0	0	-	-	-	-	-	**P**	**P**	-	**u**	**u**	**u**	**u**	**u**	-	**s**	**s**	**d**	**d**	**d**	**d**	**d**	if (Ps) Vd=Vu
0	0	0	1	1	0	1	0	0	0	1	-	-	-	-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

Average

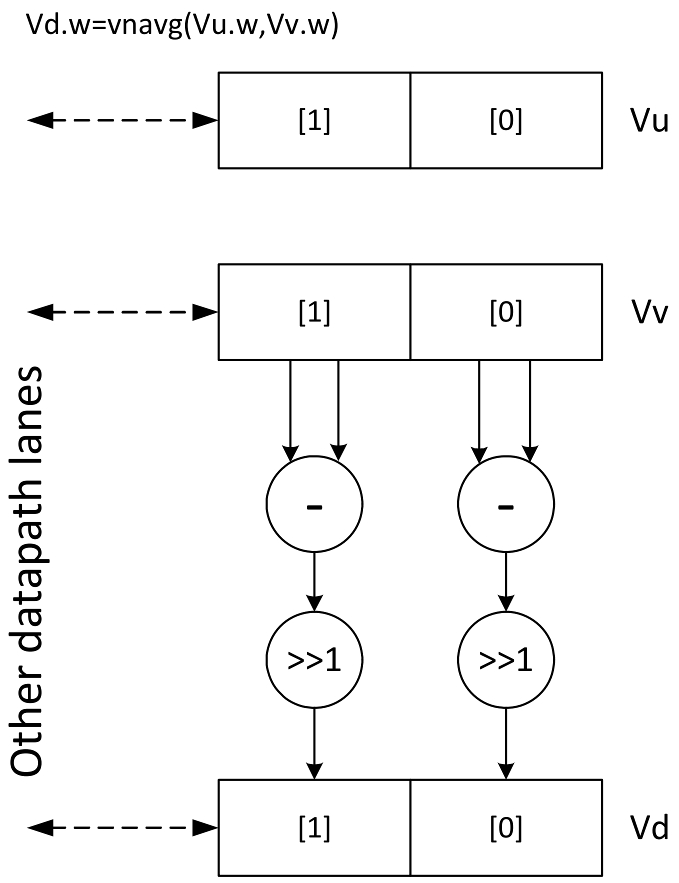
Add the elements of Vu to the respective elements of Vv, and shift the results right by one bit. The intermediate precision of the sum is larger than the input data precision. Optionally, a rounding constant 0x1 is added before shifting.

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



Subtract the elements of Vu from the respective elements of Vv, and shift the results right by one 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 datapath width.



Syntax	Behavior

`Vd.b=vavg(Vu.b,Vv.b)[:rnd]`	`for (i = 0; i < VELEM(8); i++) {`
	`    Vd.b[i] = (Vu.b[i]+Vv.b[i]+1)/2;`
	`}`

`Vd.b=vnavg(Vu.b,Vv.b)`	`for (i = 0; i < VELEM(8); i++) {`
	`    Vd.b[i] = (Vu.b[i]-Vv.b[i])/2;`
	`}`

`Vd.b=vnavg(Vu.ub,Vv.ub)`	`for (i = 0; i < VELEM(8); i++) {`
	`    Vd.b[i] = (Vu.ub[i]-Vv.ub[i])/2;`
	`}`

`Vd.h=vavg(Vu.h,Vv.h)[:rnd]`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.h[i] = (Vu.h[i]+Vv.h[i]+1)/2;`
	`}`

`Vd.h=vnavg(Vu.h,Vv.h)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.h[i] = (Vu.h[i]-Vv.h[i])/2;`
	`}`

Syntax	Behavior

`Vd.ub=vavg(Vu.ub,Vv.ub)[:rnd]`	`for (i = 0; i < VELEM(8); i++) {`
	`    Vd.ub[i] = (Vu.ub[i]+Vv.ub[i]+1)/2;`
	`}`

`Vd.uh=vavg(Vu.uh,Vv.uh)[:rnd]`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.uh[i] = (Vu.uh[i]+Vv.uh[i]+1)/2;`
	`}`

`Vd.uw=vavg(Vu.uw,Vv.uw)[:rnd]`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.uw[i] = (Vu.uw[i]+Vv.uw[i]+1)/2;`
	`}`

`Vd.w=vavg(Vu.w,Vv.w)[:rnd]`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.w[i] = (Vu.w[i]+Vv.w[i]+1)/2;`
	`}`

`Vd.w=vnavg(Vu.w,Vv.w)`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.w[i] = (Vu.w[i]-Vv.w[i])/2;`
	`}`

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

Notes

- This instruction can use any HVX resource.
- This instruction may not work correctly in Napali V1.

Intrinsics

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

`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

`Qd4=vcmp.eq(Vu.b,Vv.b)`	`for( i = 0; i < VWIDTH; i += 1) {`
	`    QdV[i+1-1:i] = ((Vu.b[i/1] == Vv.b[i/1]) ? 0x1 : 0);`
	`}`

`Qd4=vcmp.eq(Vu.h,Vv.h)`	`for( i = 0; i < VWIDTH; i += 2) {`
	`    QdV[i+2-1:i] = ((Vu.h[i/2] == Vv.h[i/2]) ? 0x3 : 0);`
	`}`

`Qd4=vcmp.eq(Vu.ub,Vv.ub)`	`Assembler mapped to: "Qd4=vcmp.eq(Vu." "b" ",Vv." "b" ")"`
`Qd4=vcmp.eq(Vu.uh,Vv.uh)`	`Assembler mapped to: "Qd4=vcmp.eq(Vu." "h" ",Vv." "h" ")"`
`Qd4=vcmp.eq(Vu.uw,Vv.uw)`	`Assembler mapped to: "Qd4=vcmp.eq(Vu." "w" ",Vv." "w" ")"`

`Qd4=vcmp.eq(Vu.w,Vv.w)`	`for( i = 0; i < VWIDTH; i += 4) {`
	`    QdV[i+4-1:i] = ((Vu.w[i/4] == Vv.w[i/4]) ? 0xF : 0);`
	`}`

`Qd4=vcmp.gt(Vu.b,Vv.b)`	`for( i = 0; i < VWIDTH; i += 1) {`
	`    QdV[i+1-1:i] = ((Vu.b[i/1] > Vv.b[i/1]) ? 0x1 : 0);`
	`}`

`Qd4=vcmp.gt(Vu.h,Vv.h)`	`for( i = 0; i < VWIDTH; i += 2) {`
	`    QdV[i+2-1:i] = ((Vu.h[i/2] > Vv.h[i/2]) ? 0x3 : 0);`
	`}`

`Qd4=vcmp.gt(Vu.ub,Vv.ub)`	`for( i = 0; i < VWIDTH; i += 1) {`
	`    QdV[i+1-1:i] = ((Vu.ub[i/1] > Vv.ub[i/1]) ? 0x1 : 0);`
	`}`

`Qd4=vcmp.gt(Vu.uh,Vv.uh)`	`for( i = 0; i < VWIDTH; i += 2) {`
	`    QdV[i+2-1:i] = ((Vu.uh[i/2] > Vv.uh[i/2]) ? 0x3 : 0);`
	`}`

`Qd4=vcmp.gt(Vu.uw,Vv.uw)`	`for( i = 0; i < VWIDTH; i += 4) {`
	`    QdV[i+4-1:i] = ((Vu.uw[i/4] > Vv.uw[i/4]) ? 0xF : 0);`
	`}`

`Qd4=vcmp.gt(Vu.w,Vv.w)`	`for( i = 0; i < VWIDTH; i += 4) {`
	`    QdV[i+4-1:i] = ((Vu.w[i/4] > Vv.w[i/4]) ? 0xF : 0);`
	`}`

`Qx4[&|]=vcmp.eq(Vu.b,Vv.b)`	`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);`
	`}`

Syntax	Behavior

Qx4[&|]=vcmp.eq(Vu.h,Vv.h)	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);
	}

Qx4[&|]=vcmp.eq(Vu.ub,Vv.ub)	Assembler mapped to: "Qx4[|&]=vcmp.eq(Vu." "b" ",Vv." "b" ")"
Qx4[&|]=vcmp.eq(Vu.uh,Vv.uh)	Assembler mapped to: "Qx4[|&]=vcmp.eq(Vu." "h" ",Vv." "h" ")"
Qx4[&|]=vcmp.eq(Vu.uw,Vv.uw)	Assembler mapped to: "Qx4[|&]=vcmp.eq(Vu." "w" ",Vv." "w" ")"

Qx4[&|]=vcmp.eq(Vu.w,Vv.w)	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);
	}

Qx4[&|]=vcmp.gt(Vu.b,Vv.b)	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);
	}

Qx4[&|]=vcmp.gt(Vu.h,Vv.h)	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);
	}

Qx4[&|]=vcmp.gt(Vu.ub,Vv.ub)	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);
	}

Qx4[&|]=vcmp.gt(Vu.uh,Vv.uh)	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);
	}

Qx4[&|]=vcmp.gt(Vu.uw,Vv.uw)	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);
	}

Qx4[&|]=vcmp.gt(Vu.w,Vv.w)	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);
	}

Qx4^=vcmp.eq(Vu.b,Vv.b)	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);
	}

Qx4^=vcmp.eq(Vu.h,Vv.h)	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);
	}

Qx4^=vcmp.eq(Vu.ub,Vv.ub)	Assembler mapped to: "Qx4^=vcmp.eq(Vu." "b" ",Vv." "b" ")"
Qx4^=vcmp.eq(Vu.uh,Vv.uh)	Assembler mapped to: "Qx4^=vcmp.eq(Vu." "h" ",Vv." "h" ")"
Qx4^=vcmp.eq(Vu.uw,Vv.uw)	Assembler mapped to: "Qx4^=vcmp.eq(Vu." "w" ",Vv." "w" ")"

Syntax	Behavior

`Qx4^=vcmp.eq(Vu.w,Vv.w)`	`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);`
	`}`

`Qx4^=vcmp.gt(Vu.b,Vv.b)`	`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);`
	`}`

`Qx4^=vcmp.gt(Vu.h,Vv.h)`	`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);`
	`}`

`Qx4^=vcmp.gt(Vu.ub,Vv.ub)`	`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);`
	`}`

`Qx4^=vcmp.gt(Vu.uh,Vv.uh)`	`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);`
	`}`

`Qx4^=vcmp.gt(Vu.uw,Vv.uw)`	`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);`
	`}`

`Qx4^=vcmp.gt(Vu.w,Vv.w)`	`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);`
	`}`

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

Notes

- This instruction can use any HVX resource.

Intrinsics

`Qd4=vcmp.eq(Vu.b,Vv.b)`	`HVX_VectorPred Q6_Q_vcmp_eq_VbVb(HVX_Vector Vu, HVX_Vector Vv)`
`Qd4=vcmp.eq(Vu.h,Vv.h)`	`HVX_VectorPred Q6_Q_vcmp_eq_VhVh(HVX_Vector Vu, HVX_Vector Vv)`
`Qd4=vcmp.eq(Vu.w,Vv.w)`	`HVX_VectorPred Q6_Q_vcmp_eq_VwVw(HVX_Vector Vu, HVX_Vector Vv)`
`Qd4=vcmp.gt(Vu.b,Vv.b)`	`HVX_VectorPred Q6_Q_vcmp_gt_VbVb(HVX_Vector Vu, HVX_Vector Vv)`
`Qd4=vcmp.gt(Vu.h,Vv.h)`	`HVX_VectorPred Q6_Q_vcmp_gt_VhVh(HVX_Vector Vu, HVX_Vector Vv)`
`Qd4=vcmp.gt(Vu.ub,Vv.ub)`	`HVX_VectorPred Q6_Q_vcmp_gt_VubVub(HVX_Vector Vu, HVX_Vector Vv)`
`Qd4=vcmp.gt(Vu.uh,Vv.uh)`	`HVX_VectorPred Q6_Q_vcmp_gt_VuhVuh(HVX_Vector Vu, HVX_Vector Vv)`

`Qd4=vcmp.gt(Vu.uw,Vv.uw)`	`HVX_VectorPred Q6_Q_vcmp_gt_VuwVuw(HVX_Vector Vu, HVX_Vector Vv)`
`Qd4=vcmp.gt(Vu.w,Vv.w)`	`HVX_VectorPred Q6_Q_vcmp_gt_VwVw(HVX_Vector Vu, HVX_Vector Vv)`
`Qx4&=vcmp.eq(Vu.b,Vv.b)`	`HVX_VectorPred Q6_Q_vcmp_eqand_QVbVb(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4&=vcmp.eq(Vu.h,Vv.h)`	`HVX_VectorPred Q6_Q_vcmp_eqand_QVhVh(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4&=vcmp.eq(Vu.w,Vv.w)`	`HVX_VectorPred Q6_Q_vcmp_eqand_QVwVw(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4&=vcmp.gt(Vu.b,Vv.b)`	`HVX_VectorPred Q6_Q_vcmp_gtand_QVbVb(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4&=vcmp.gt(Vu.h,Vv.h)`	`HVX_VectorPred Q6_Q_vcmp_gtand_QVhVh(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4&=vcmp.gt(Vu.ub,Vv.ub)`	`HVX_VectorPred Q6_Q_vcmp_gtand_QVubVub(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4&=vcmp.gt(Vu.uh,Vv.uh)`	`HVX_VectorPred Q6_Q_vcmp_gtand_QVuhVuh(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4&=vcmp.gt(Vu.uw,Vv.uw)`	`HVX_VectorPred Q6_Q_vcmp_gtand_QVuwVuw(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4&=vcmp.gt(Vu.w,Vv.w)`	`HVX_VectorPred Q6_Q_vcmp_gtand_QVwVw(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4^=vcmp.eq(Vu.b,Vv.b)`	`HVX_VectorPred Q6_Q_vcmp_eqxacc_QVbVb(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4^=vcmp.eq(Vu.h,Vv.h)`	`HVX_VectorPred Q6_Q_vcmp_eqxacc_QVhVh(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4^=vcmp.eq(Vu.w,Vv.w)`	`HVX_VectorPred Q6_Q_vcmp_eqxacc_QVwVw(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4^=vcmp.gt(Vu.b,Vv.b)`	`HVX_VectorPred Q6_Q_vcmp_gtxacc_QVbVb(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4^=vcmp.gt(Vu.h,Vv.h)`	`HVX_VectorPred Q6_Q_vcmp_gtxacc_QVhVh(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4^=vcmp.gt(Vu.ub,Vv.ub)`	`HVX_VectorPred Q6_Q_vcmp_gtxacc_QVubVub(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4^=vcmp.gt(Vu.uh,Vv.uh)`	`HVX_VectorPred Q6_Q_vcmp_gtxacc_QVuhVuh(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`

`Qx4^=vcmp.gt(Vu.uw,Vv.uw)`	`HVX_VectorPred Q6_Q_vcmp_gtxacc_QVuwVuw(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4^=vcmp.gt(Vu.w,Vv.w)`	`HVX_VectorPred Q6_Q_vcmp_gtxacc_QVwVw(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4|=vcmp.eq(Vu.b,Vv.b)`	`HVX_VectorPred Q6_Q_vcmp_eqor_QVbVb(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4|=vcmp.eq(Vu.h,Vv.h)`	`HVX_VectorPred Q6_Q_vcmp_eqor_QVhVh(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4|=vcmp.eq(Vu.w,Vv.w)`	`HVX_VectorPred Q6_Q_vcmp_eqor_QVwVw(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4|=vcmp.gt(Vu.b,Vv.b)`	`HVX_VectorPred Q6_Q_vcmp_gtor_QVbVb(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4|=vcmp.gt(Vu.h,Vv.h)`	`HVX_VectorPred Q6_Q_vcmp_gtor_QVhVh(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4|=vcmp.gt(Vu.ub,Vv.ub)`	`HVX_VectorPred Q6_Q_vcmp_gtor_QVubVub(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4|=vcmp.gt(Vu.uh,Vv.uh)`	`HVX_VectorPred Q6_Q_vcmp_gtor_QVuhVuh(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4|=vcmp.gt(Vu.uw,Vv.uw)`	`HVX_VectorPred Q6_Q_vcmp_gtor_QVuwVuw(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`
`Qx4|=vcmp.gt(Vu.w,Vv.w)`	`HVX_VectorPred Q6_Q_vcmp_gtor_QVwVw(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)`

Encoding

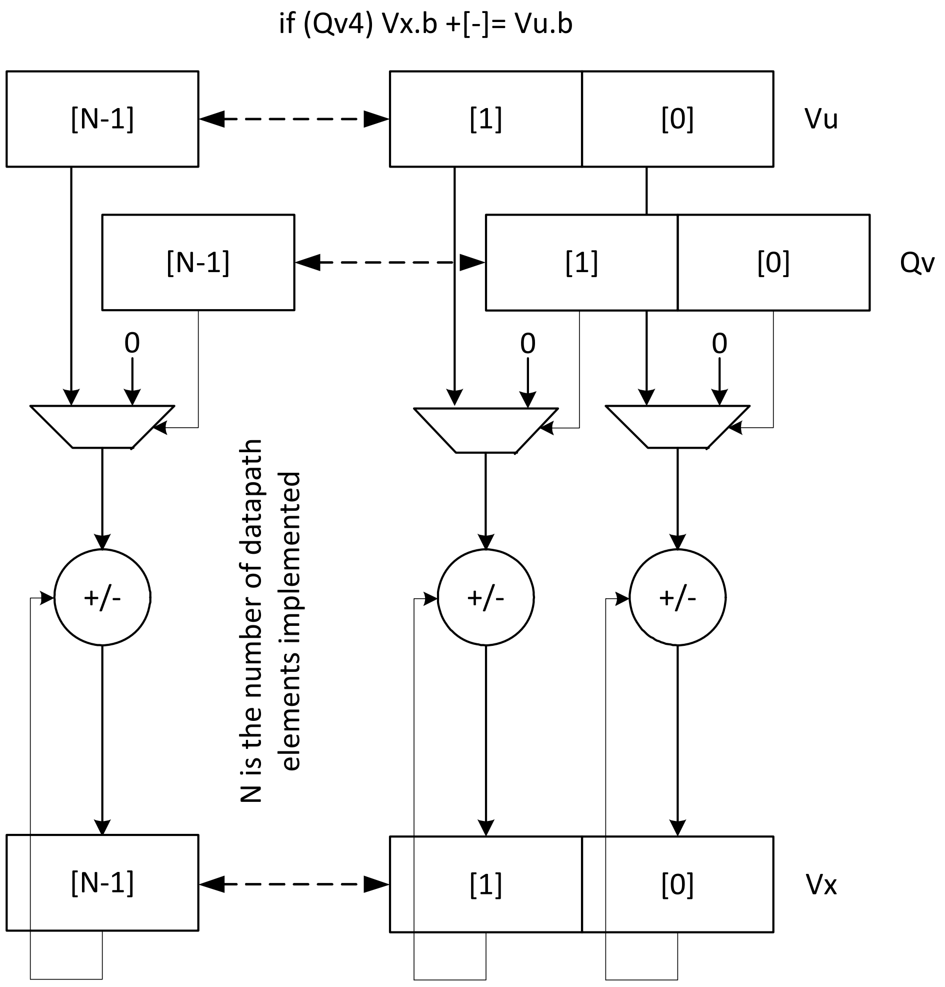
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse			u5											x2		
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	0	0	0	x	x	Qx4&=vcmp.eq(Vu.b,Vv.b)
0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	0	0	1	x	x	Qx4&=vcmp.eq(Vu.h,Vv.h)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

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	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	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	1	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	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)
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	Behavior

`if ([!]Qv4) Vx.b[+-]=Vu.b`	`for (i = 0; i < VELEM(8); i[+-][+-]) {`
	`    Vx.ub[i]=QvV.i ? Vx.ub[i] : Vx.ub[i][+-]Vu.ub[i];`
	`}`

`if ([!]Qv4) Vx.h[+-]=Vu.h`	`for (i = 0; i < VELEM(16); i[+-][+-]) {`
	`    Vx.h[i]=select_bytes(QvV,i,Vx.h[i],Vx.h[i][+-]Vu.h[i]);`
	`}`

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

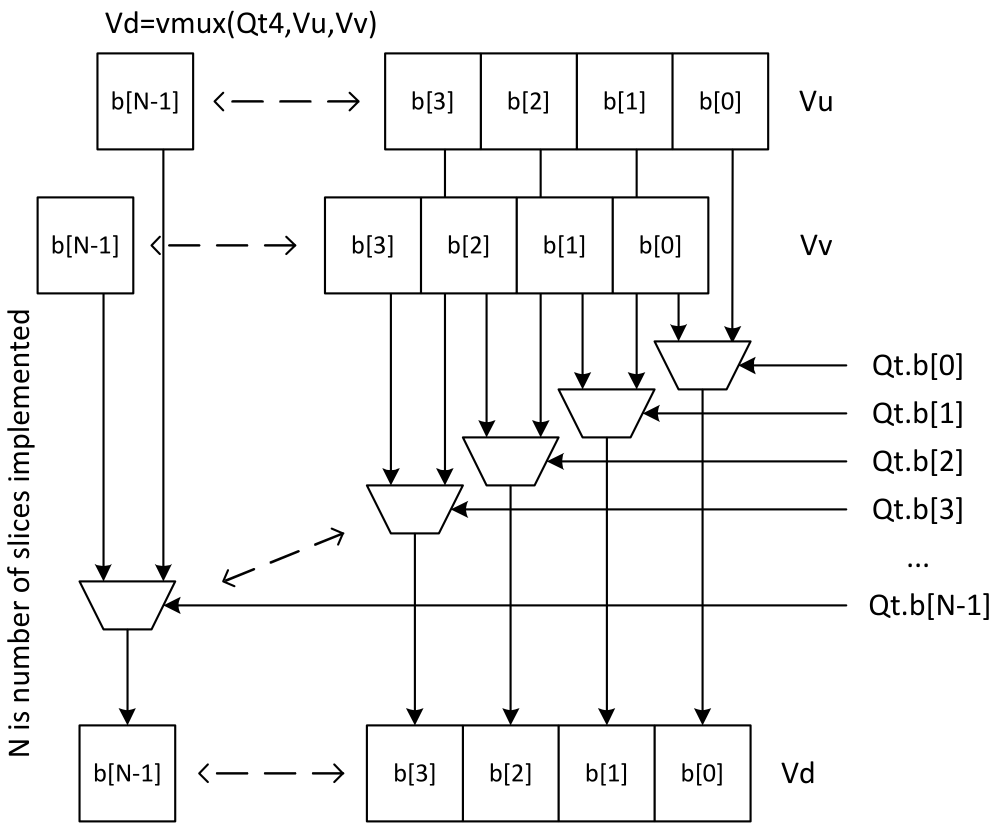
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse			u5								x5					
0	0	0	1	1	1	1	0	v	v	0	-	-	-	0	1	P	P	1	u	u	u	u	u	0	0	0	x	x	x	x	x	if (Qv4) Vx.b+=Vu.b
0	0	0	1	1	1	1	0	v	v	0	-	-	-	0	1	P	P	1	u	u	u	u	u	0	0	1	x	x	x	x	x	if (Qv4) Vx.h+=Vu.h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

`Vd=vmux(Qt4,Vu,Vv)`	`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
`v5`	Field to encode register v

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
`Vd.h=vsat(Vu.w,Vv.w)`	for (i = 0; i < VELEM(32); i++) { Vd.w[i].h[0]=sat$_{16}$(Vv.w[i]); Vd.w[i].h[1]=sat$_{16}$(Vu.w[i]); }
`Vd.ub=vsat(Vu.h,Vv.h)`	for (i = 0; i < VELEM(16); i++) { Vd.uh[i].b[0]=usat$_{8}$(Vv.h[i]); Vd.uh[i].b[1]=usat$_{8}$(Vu.h[i]); }
`Vd.uh=vsat(Vu.uw,Vv.uw)`	for (i = 0; i < VELEM(32); i++) { Vd.w[i].h[0]=usat$_{16}$(Vv.uw[i]); Vd.w[i].h[1]=usat$_{16}$(Vu.uw[i]); }

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

Notes

- This instruction can use any HVX resource.

Intrinsics

`Vd.h=vsat(Vu.w,Vv.w)`	`HVX_Vector Q6_Vh_vsat_VwVw(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.ub=vsat(Vu.h,Vv.h)`	`HVX_Vector Q6_Vub_vsat_VhVh(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.uh=vsat(Vu.uw,Vv.uw)`	`HVX_Vector Q6_Vuh_vsat_VuwVuw(HVX_Vector Vu, HVX_Vector Vv)`

Encoding

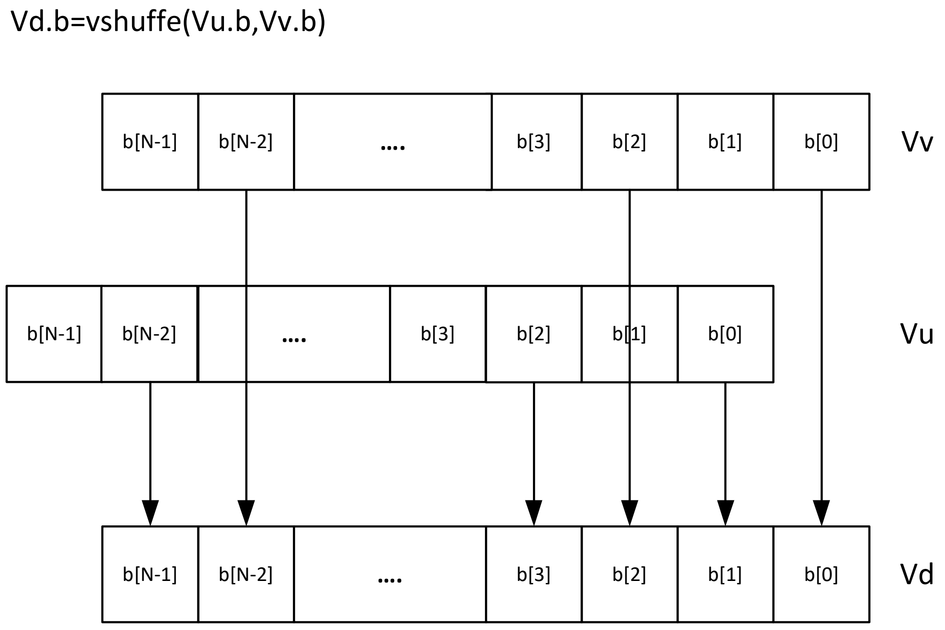
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse			u5								d5					
0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d	Vd.uh=vsat(Vu.uw,Vv.uw)
0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.ub=vsat(Vu.h,Vv.h)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

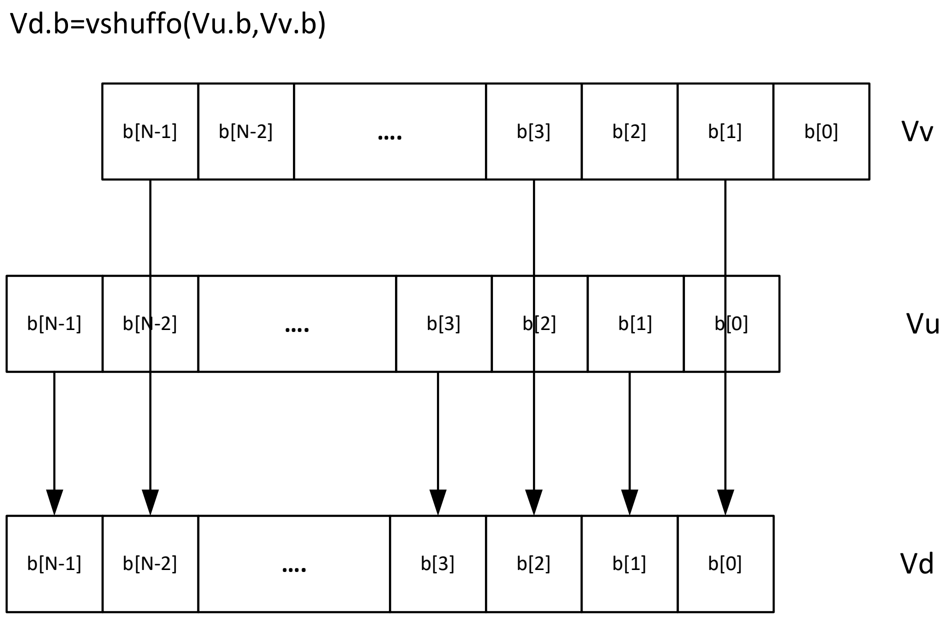
Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits

Field name	Description
`d5`	Field to encode register d
`u5`	Field to encode register u
`v5`	Field to encode register v

In-lane shuffle

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





This group of shuffles is limited to bytes and halfwords.

Syntax	Behavior

`Vd.b=vshuffe(Vu.b,Vv.b)`	`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];`
	`}`

`Vd.b=vshuffo(Vu.b,Vv.b)`	`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];`
	`}`

`Vd.h=vshuffe(Vu.h,Vv.h)`	`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];`
	`}`

`Vd.h=vshuffo(Vu.h,Vv.h)`	`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];`
	`}`

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

Notes

- This instruction can use any HVX resource.

Intrinsics

`Vd.b=vshuffe(Vu.b,Vv.b)`	`HVX_Vector Q6_Vb_vshuffe_VbVb(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.b=vshuffo(Vu.b,Vv.b)`	`HVX_Vector Q6_Vb_vshuffo_VbVb(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.h=vshuffe(Vu.h,Vv.h)`	`HVX_Vector Q6_Vh_vshuffe_VhVh(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.h=vshuffo(Vu.h,Vv.h)`	`HVX_Vector Q6_Vh_vshuffo_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	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

Field name	Description
`d5`	Field to encode register d
`u5`	Field to encode register u
`v5`	Field to encode register v

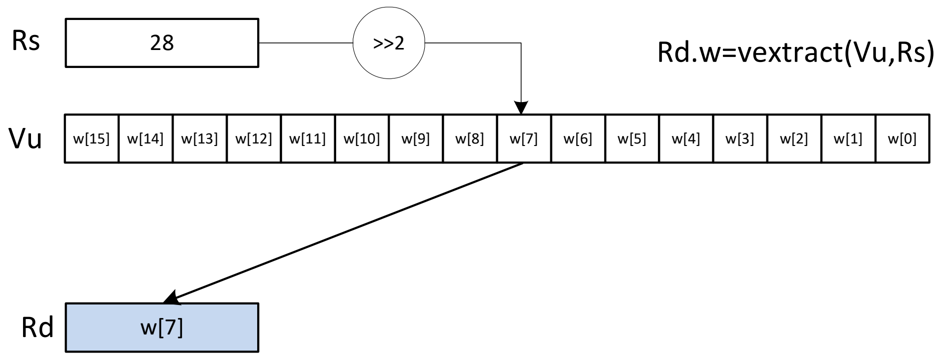
5.4 HVX/DEBUG

The HVX/DEBUG instruction subclass includes instructions used for debugging.

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	Behavior
`Rd.w=vextract(Vu,Rs)`	`Assembler mapped to: "Rd=vextract(Vu,Rs)"`
`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			UN	s5					Parse			u5								d5					
1	0	0	1	0	0	1	0	0	0	0	s	s	s	s	s	P	P	-	u	u	u	u	u	-	-	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

5.5 HVX/GATHER-DOUBLE-RESOURCE

The HVX/GATHER-DOUBLE-RESOURCE instruction subclass includes instructions that perform gather operations in the vector TCM.

Vector gather

Vector gather 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 gather region's end, 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.

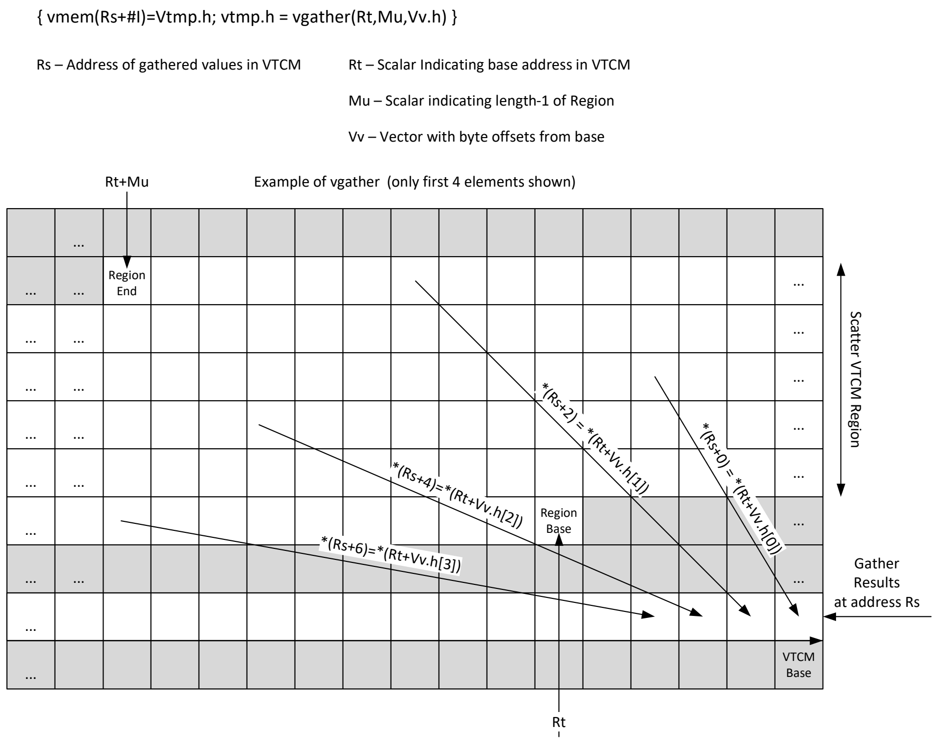
The gather instruction must be paired with a VMEM .new store that uses a tmp register source.

For example:

{ VMEM(R0+#0) = Vtmp.new; Vtmp.h = vgather(R1,M0, 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+#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
Rt+Mu
Example of vgather (only first 4 elements shown)
Region End
Scatter VTCM Region
*(Rs+2) = *(Rt+Vv.h[1])
*(Rs+0) = *(Rt+Vv.h[0])
(Rs+4)=(Rt+Vv.h[2])
(Rs+6)=(Rt+Vv.h[3])
Region Base
Gather Results at address Rs
VTCM Base
Rt

Syntax	Behavior

`if (Qs4) vtmp.h=vgather(Rt,Mu,Vvv.w).h`	`MuV | (element_size-1);`
	`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;`
	`    }`
	`}`

`vtmp.h=vgather(Rt,Mu,Vvv.w).h`	`MuV | (element_size-1);`
	`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;`
	`    }`
	`}`

Class: COPROC_VMEM (slots 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_VectorAddress A, HVX_VectorPred Qs, Word32 Rt, Word32 Mu, HVX_VectorPair Vvv)`
`vtmp.h=vgather(Rt,Mu,Vvv.w).h`	`void Q6_vgather_ARMWw(HVX_VectorAddress A, Word32 Rt, 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

5.6 HVX/GATHER

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

Vector gather

Vector gather 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 gather region's end, 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.

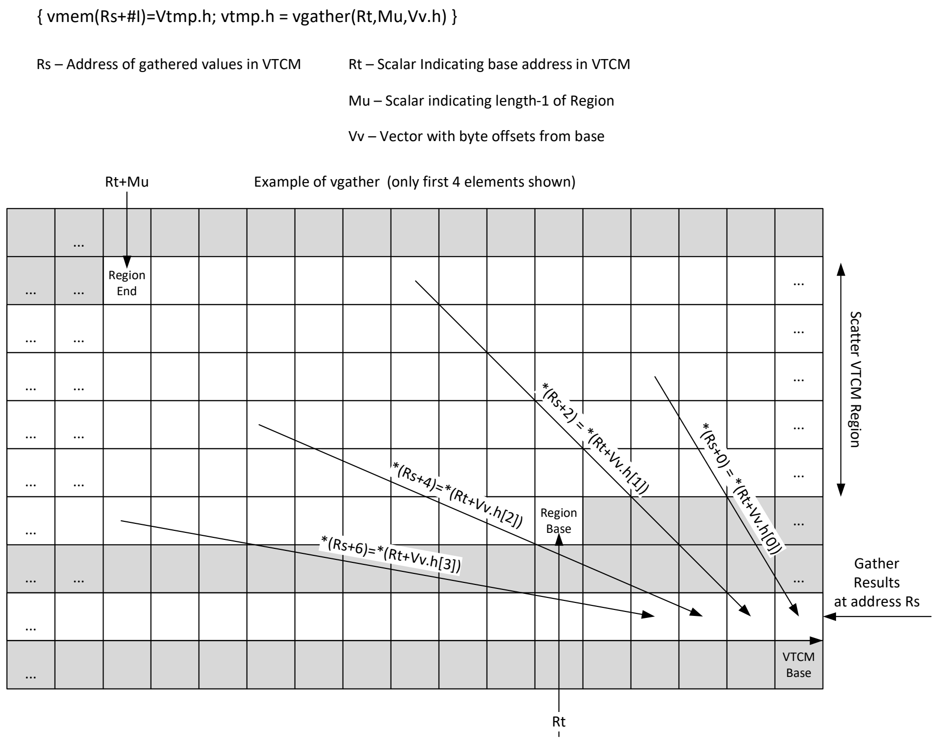
The gather instruction must be paired with a VMEM .new store that uses a temporary register source.

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
Rt+Mu
Example of vgather (only first 4 elements shown)
Region End
Region Base
VTCM Base
Scatter VTCM Region
Gather Results at address Rs
*(Rs+0) = *(Rt+Vv.h[0])
*(Rs+2) = *(Rt+Vv.h[1])
(Rs+4)=(Rt+Vv.h[2])
(Rs+6)=(Rt+Vv.h[3])
Rt

Syntax	Behavior

`if (Qs4) vtmp.h=vgather(Rt,Mu,Vv.h).h`	`MuV | (element_size-1);`
	`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;`
	`}`

`if (Qs4) vtmp.w=vgather(Rt,Mu,Vv.w).w`	`MuV | (element_size-1);`
	`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;`
	`}`

`vtmp.h=vgather(Rt,Mu,Vv.h).h`	`MuV | (element_size-1);`
	`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;`
	`}`

`vtmp.w=vgather(Rt,Mu,Vv.w).w`	`MuV | (element_size-1);`
	`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;`
	`}`

Class: COPROC_VMEM (slots 1)

Notes

- This instruction can use any HVX resource.

Intrinsics

`if (Qs4) vtmp.h=vgather(Rt,Mu,Vv.h).h`	`void Q6_vgather_AQRMVh(HVX_VectorAddress A, HVX_VectorPred Qs, Word32 Rt, Word32 Mu, HVX_Vector Vv)`
`if (Qs4) vtmp.w=vgather(Rt,Mu,Vv.w).w`	`void Q6_vgather_AQRMVw(HVX_VectorAddress A, HVX_VectorPred Qs, Word32 Rt, Word32 Mu, HVX_Vector Vv)`
`vtmp.h=vgather(Rt,Mu,Vv.h).h`	`void Q6_vgather_ARMVh(HVX_VectorAddress A, Word32 Rt, Word32 Mu, HVX_Vector Vv)`
`vtmp.w=vgather(Rt,Mu,Vv.w).w`	`void Q6_vgather_ARMVw(HVX_VectorAddress A, Word32 Rt, Word32 Mu, 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									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

5.7 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):nt`	`Assembler mapped to: "Vd=vmem(Rt+#0):nt"`

`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(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++Mu)`	`EA=Rx;`
	`Vd = *(EA&~(ALIGNMENT-1));`
	`Rx=Rx+MuV;`

`Vd=vmem(Rx++Mu):nt`	`EA=Rx;`
	`Vd = *(EA&~(ALIGNMENT-1));`
	`Rx=Rx+MuV;`

`if ([!]Pv) Vd=vmem(Rt)`	`Assembler mapped to: "if ([!]Pv) Vd=vmem(Rt+#0)"`
`if ([!]Pv) Vd=vmem(Rt):nt`	`Assembler mapped to: "if ([!]Pv) Vd=vmem(Rt+#0):nt"`

`if ([!]Pv) Vd=vmem(Rt+#s4)`	`if ([!]Pv[0]) {`
	`    EA=Rt+#s*VBYTES;`
	`    Vd = *(EA&~(ALIGNMENT-1));`
	`} else {`
	`    NOP;`
	`}`

Syntax	Behavior

`if ([!]Pv) Vd=vmem(Rt+#s4):nt`	`if ([!]Pv[0]) {`
	`    EA=Rt+#s*VBYTES;`
	`    Vd = *(EA&~(ALIGNMENT-1));`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) Vd=vmem(Rx++#s3)`	`if ([!]Pv[0]) {`
	`    EA=Rx;`
	`    Vd = *(EA&~(ALIGNMENT-1));`
	`    Rx=Rx+#s*VBYTES;`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) Vd=vmem(Rx++#s3):nt`	`if ([!]Pv[0]) {`
	`    EA=Rx;`
	`    Vd = *(EA&~(ALIGNMENT-1));`
	`    Rx=Rx+#s*VBYTES;`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) Vd=vmem(Rx++Mu)`	`if ([!]Pv[0]) {`
	`    EA=Rx;`
	`    Vd = *(EA&~(ALIGNMENT-1));`
	`    Rx=Rx+MuV;`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) Vd=vmem(Rx++Mu):nt`	`if ([!]Pv[0]) {`
	`    EA=Rx;`
	`    Vd = *(EA&~(ALIGNMENT-1));`
	`    Rx=Rx+MuV;`
	`} else {`
	`    NOP;`
	`}`

Class: COPROC_VMEM (slots 0,1)

Notes

- This instruction can use any HVX resource.
- An optional "nontemporal" hint to the micro-architecture 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 (!Pv) Vd=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	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 (!Pv) 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 (!Pv) 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 (!Pv) Vd=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	0	0	d	d	d	d	d	Vd=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	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 (!Pv) 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 (!Pv) Vd=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 - 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.

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

"Vd.tmp" 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. Note that this form does not consume any 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
`if ([!]Pv) Vd.cur=vmem(Rt+#s4):nt`	`if ([!]Pv[0]) { EA=Rt+#s*VBYTES; Vd = *(EA&~(ALIGNMENT-1)); } else { NOP; }`
`if ([!]Pv) Vd.cur=vmem(Rx++#s3)`	`if ([!]Pv[0]) { EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+#s*VBYTES; } else { NOP; }`
`if ([!]Pv) Vd.cur=vmem(Rx++#s3):nt`	`if ([!]Pv[0]) { EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+#s*VBYTES; } else { NOP; }`
`if ([!]Pv) Vd.cur=vmem(Rx++Mu)`	`if ([!]Pv[0]) { EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+MuV; } else { NOP; }`
`if ([!]Pv) Vd.cur=vmem(Rx++Mu):nt`	`if ([!]Pv[0]) { EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+MuV; } else { NOP; }`

Class: COPROC_VMEM (slots 0,1)

Notes

- This instruction can use any HVX resource.
- An optional "nontemporal" hint to the micro-architecture 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	0	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.

"Vd.tmp" 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. Note that this form does not consume any 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

`if ([!]Pv) Vd.tmp=vmem(Rt+#s4):nt`	`if ([!]Pv[0]) {`
	`    EA=Rt+#s*VBYTES;`
	`    Vd = *(EA&~(ALIGNMENT-1));`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) Vd.tmp=vmem(Rx++#s3)`	`if ([!]Pv[0]) {`
	`    EA=Rx;`
	`    Vd = *(EA&~(ALIGNMENT-1));`
	`    Rx=Rx+#s*VBYTES;`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) Vd.tmp=vmem(Rx++#s3):nt`	`if ([!]Pv[0]) {`
	`    EA=Rx;`
	`    Vd = *(EA&~(ALIGNMENT-1));`
	`    Rx=Rx+#s*VBYTES;`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) Vd.tmp=vmem(Rx++Mu)`	`if ([!]Pv[0]) {`
	`    EA=Rx;`
	`    Vd = *(EA&~(ALIGNMENT-1));`
	`    Rx=Rx+MuV;`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) Vd.tmp=vmem(Rx++Mu):nt`	`if ([!]Pv[0]) {`
	`    EA=Rx;`
	`    Vd = *(EA&~(ALIGNMENT-1));`
	`    Rx=Rx+MuV;`
	`} else {`
	`    NOP;`
	`}`

Class: COPROC_VMEM (slots 0,1)

Notes

- This instruction can use any HVX resource.
- An optional "nontemporal" hint to the micro-architecture 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	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 (!Pv) 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 (!Pv) 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 (!Pv) 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 (!Pv) 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 (!Pv) 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 (!Pv) 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.

Note that this instruction uses both slot 0 and slot 1, allowing only three instructions at most 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		u1									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
`d5`	Field to encode register d
`t5`	Field to encode register t
`u1`	Field to encode register u
`x5`	Field to encode register x

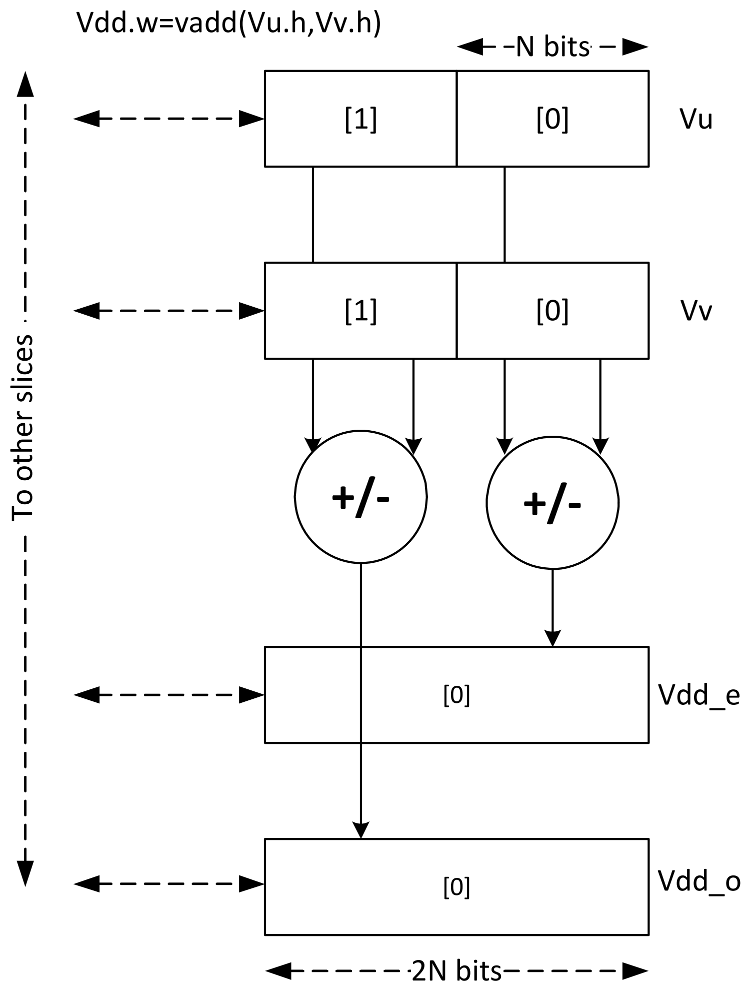
5.8 HVX/MPY-DOUBLE-RESOURCE

The HVX/MPY-DOUBLE-RESOURCE instruction subclass includes instructions that use both HVX multiply resources.

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	Behavior

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

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

`Vdd.w=vadd(Vu.h,Vv.h)`	`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];`
	`}`

`Vdd.w=vadd(Vu.uh,Vv.uh)`	`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];`
	`}`

`Vdd.w=vsub(Vu.h,Vv.h)`	`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];`
	`}`

`Vdd.w=vsub(Vu.uh,Vv.uh)`	`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];`
	`}`

`Vxx.h+=vadd(Vu.ub,Vv.ub)`	`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];`
	`}`

`Vxx.w+=vadd(Vu.h,Vv.h)`	`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];`
	`}`

`Vxx.w+=vadd(Vu.uh,Vv.uh)`	`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];`
	`}`

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

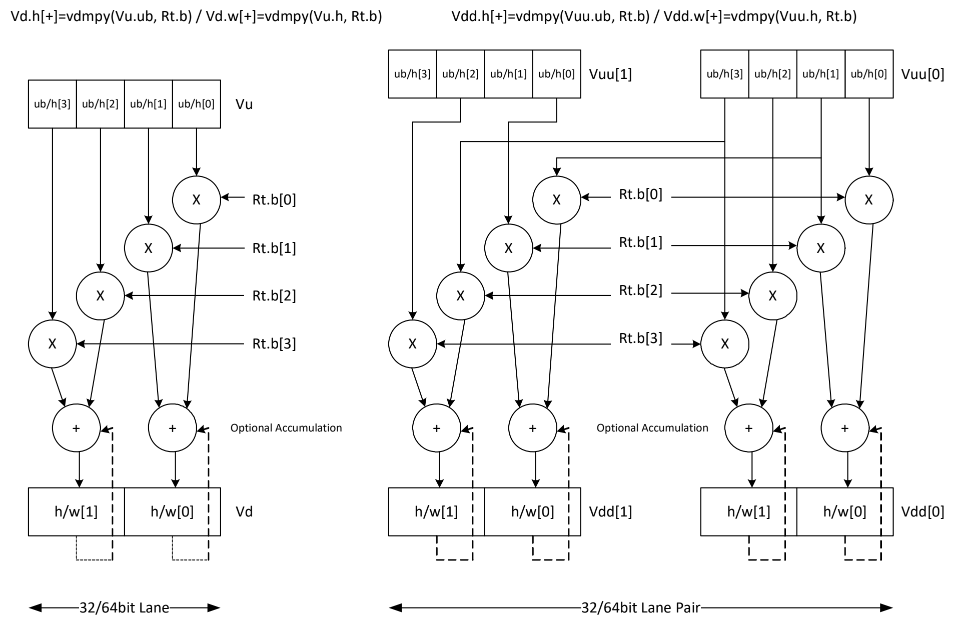
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse			u5								x5					
0	0	0	1	1	1	0	0	0	0	1	**v**	**v**	**v**	**v**	**v**	**P**	**P**	1	**u**	**u**	**u**	**u**	**u**	0	1	0	**x**	**x**	**x**	**x**	**x**	Vxx.w+=vadd(Vu.h,Vv.h)
0	0	0	1	1	1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

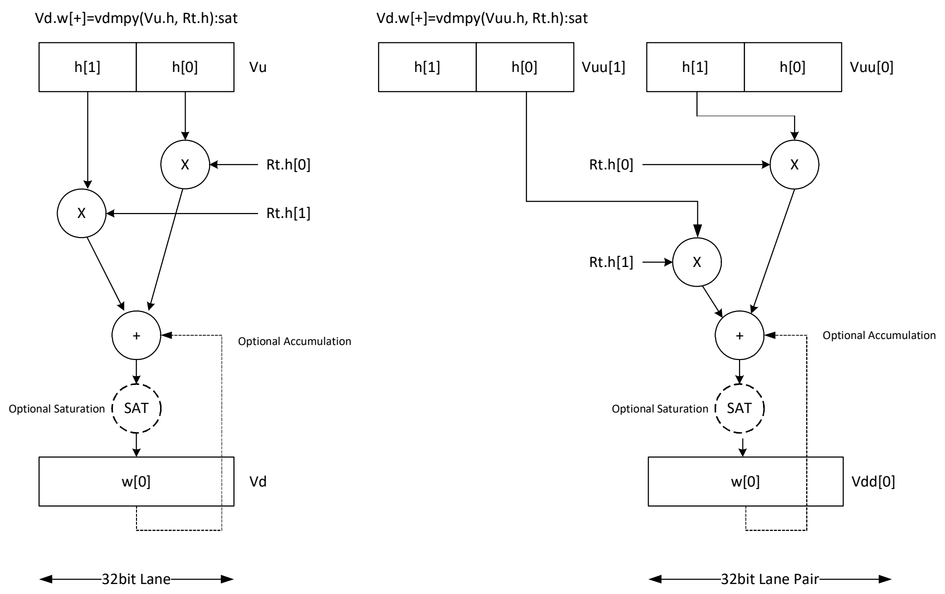
Field name	**Description**
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`u5`	Field to encode register u
`v5`	Field to encode register v
`x5`	Field to encode register x

Multiply with two-wide reduction

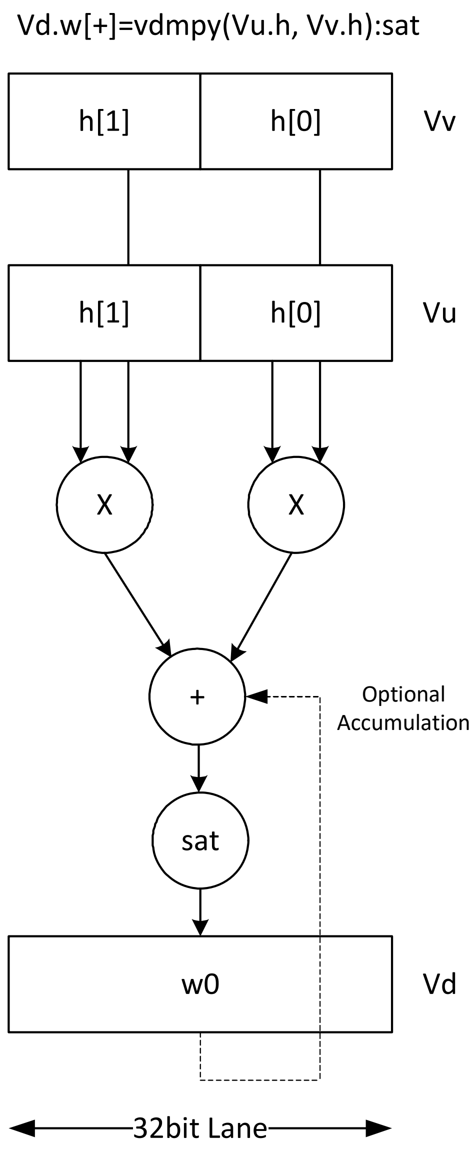
Multiply elements from Vu by the corresponding elements in the scalar register Rt. The products are added in pairs to yield a by-two reduction. The products can optionally be accumulated with Vx, with optional saturation after summation.

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





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.



Syntax	Behavior

`Vd.w=vdmpy(Vu.h,Rt.h):sat`	`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] = sat`$_{32}$`(accum);`
	`}`

`Vd.w=vdmpy(Vu.h,Rt.uh):sat`	`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] = sat`$_{32}$`(accum);`
	`}`

`Vd.w=vdmpy(Vu.h,Vv.h):sat`	`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] = sat`$_{32}$`(accum);`
	`}`

Syntax	Behavior

Vd.w=vdmpy(Vuu.h,Rt.h):sat	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] = sat$_{32}$(accum);
	}

Vd.w=vdmpy(Vuu.h,Rt.uh,#1):sat	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] = sat$_{32}$(accum);
	}

Vdd.h=vdmpy(Vuu.ub,Rt.b)	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]);
	}

Vdd.w=vdmpy(Vuu.h,Rt.b)	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]);
	}

Vx.w+=vdmpy(Vu.h,Rt.h):sat	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] = sat$_{32}$(accum);
	}

Vx.w+=vdmpy(Vu.h,Rt.uh):sat	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] = sat$_{32}$(accum);
	}

Vx.w+=vdmpy(Vu.h,Vv.h):sat	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]);
	    Vx.w[i] = sat$_{32}$(Vx.w[i]+accum);
	}

Vx.w+=vdmpy(Vuu.h,Rt.h):sat	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] = sat$_{32}$(accum);
	}

Syntax	Behavior

`Vx.w+=vdmpy(Vuu.h,Rt.uh,#1):sat`	`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] = sat`$_{32}$`(accum);`
	`}`

`Vxx.h+=vdmpy(Vuu.ub,Rt.b)`	`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]);`
	`}`

`Vxx.w+=vdmpy(Vuu.h,Rt.b)`	`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]);`
	`}`

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.

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)`
`Vd.w=vdmpy(Vuu.h,Rt.h):sat`	`HVX_Vector Q6_Vw_vdmpy_WhRh_sat(HVX_VectorPair Vuu, Word32 Rt)`
`Vd.w=vdmpy(Vuu.h,Rt.uh,#1):sat`	`HVX_Vector Q6_Vw_vdmpy_WhRuh_sat(HVX_VectorPair Vuu, Word32 Rt)`
`Vdd.h=vdmpy(Vuu.ub,Rt.b)`	`HVX_VectorPair Q6_Wh_vdmpy_WubRb(HVX_VectorPair Vuu, Word32 Rt)`
`Vdd.w=vdmpy(Vuu.h,Rt.b)`	`HVX_VectorPair Q6_Ww_vdmpy_WhRb(HVX_VectorPair Vuu, Word32 Rt)`
`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)`
`Vx.w+=vdmpy(Vu.h,Vv.h):sat`	`HVX_Vector Q6_Vw_vdmpyacc_VwVhVh_sat(HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv)`
`Vx.w+=vdmpy(Vuu.h,Rt.h):sat`	`HVX_Vector Q6_Vw_vdmpyacc_VwWhRh_sat(HVX_Vector Vx, HVX_VectorPair Vuu, Word32 Rt)`
`Vx.w+=vdmpy(Vuu.h,Rt.uh,#1):sat`	`HVX_Vector Q6_Vw_vdmpyacc_VwWhRuh_sat(HVX_Vector Vx, HVX_VectorPair Vuu, Word32 Rt)`
`Vxx.h+=vdmpy(Vuu.ub,Rt.b)`	`HVX_VectorPair Q6_Wh_vdmpyacc_WhWubRb(HVX_VectorPair Vxx, HVX_VectorPair Vuu, Word32 Rt)`
`Vxx.w+=vdmpy(Vuu.h,Rt.b)`	`HVX_VectorPair Q6_Ww_vdmpyacc_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	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	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	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	0	d	d	d	d	d	Vd.w=vdmpy(Vu.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	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	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	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	0	1	1	x	x	x	x	x	Vx.w+=vdmpy(Vu.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								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
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

Field name	Description
`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 a scalar register pair, Rtt.

Syntax	Behavior

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

`Vx.h=vmpa(Vx.h,Vu.h,Rtt.h):sat`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vx.h[i]= sat`$_{16}$`(( ( (Vx.h[i] * Vu.h[i])<<1) + (Rtt.h[( (Vu.h[i]>>14)&0x3)]<<15))>>16);`
	`}`

`Vx.h=vmpa(Vx.h,Vu.uh,Rtt.uh):sat`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vx.h[i]= sat`$_{16}$`(( (Vx.h[i] * Vu.uh[i]) + (Rtt.uh[((Vu.uh[i]>>14)&0x3)]<<15))>>16);`
	`}`

`Vx.h=vmps(Vx.h,Vu.uh,Rtt.uh):sat`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vx.h[i]= sat`$_{16}$`(( (Vx.h[i] * Vu.uh[i]) - (Rtt.uh[((Vu.uh[i]>>14)&0x3)]<<15))>>16);`
	`}`

Class: COPROC_VX (slots 2)

Notes

- This instruction uses both HVX multiply resources.

Intrinsics

`Vx.h=vmpa(Vx.h,Vu.h,Rtt.h):sat`	`HVX_Vector Q6_Vh_vmpa_VhVhVhPh_sat(HVX_Vector Vx, HVX_Vector Vu, Word64 Rtt)`
`Vx.h=vmpa(Vx.h,Vu.uh,Rtt.uh):sat`	`HVX_Vector Q6_Vh_vmpa_VhVhVuhPuh_sat(HVX_Vector Vx, HVX_Vector Vu, Word64 Rtt)`
`Vx.h=vmps(Vx.h,Vu.uh,Rtt.uh):sat`	`HVX_Vector Q6_Vh_vmps_VhVhVuhPuh_sat(HVX_Vector Vx, 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								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

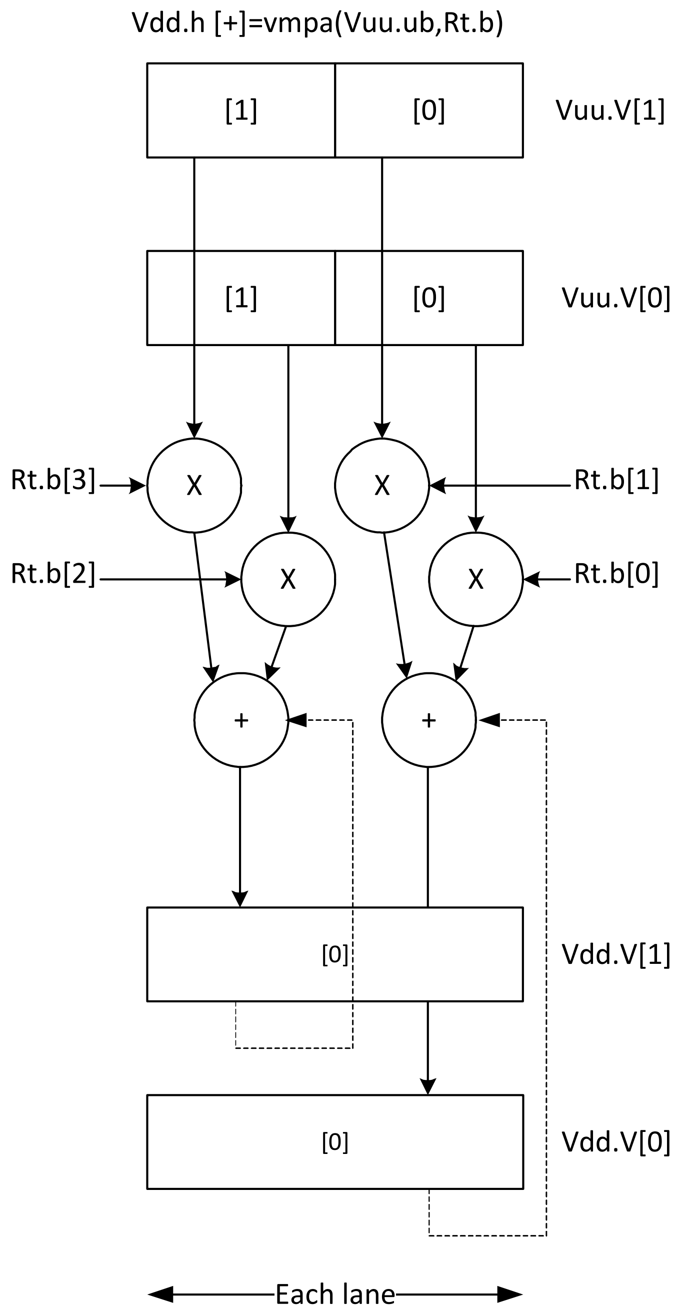
Field name	Description
`t5`	Field to encode register t
`u5`	Field to encode register u
`x5`	Field to encode register x

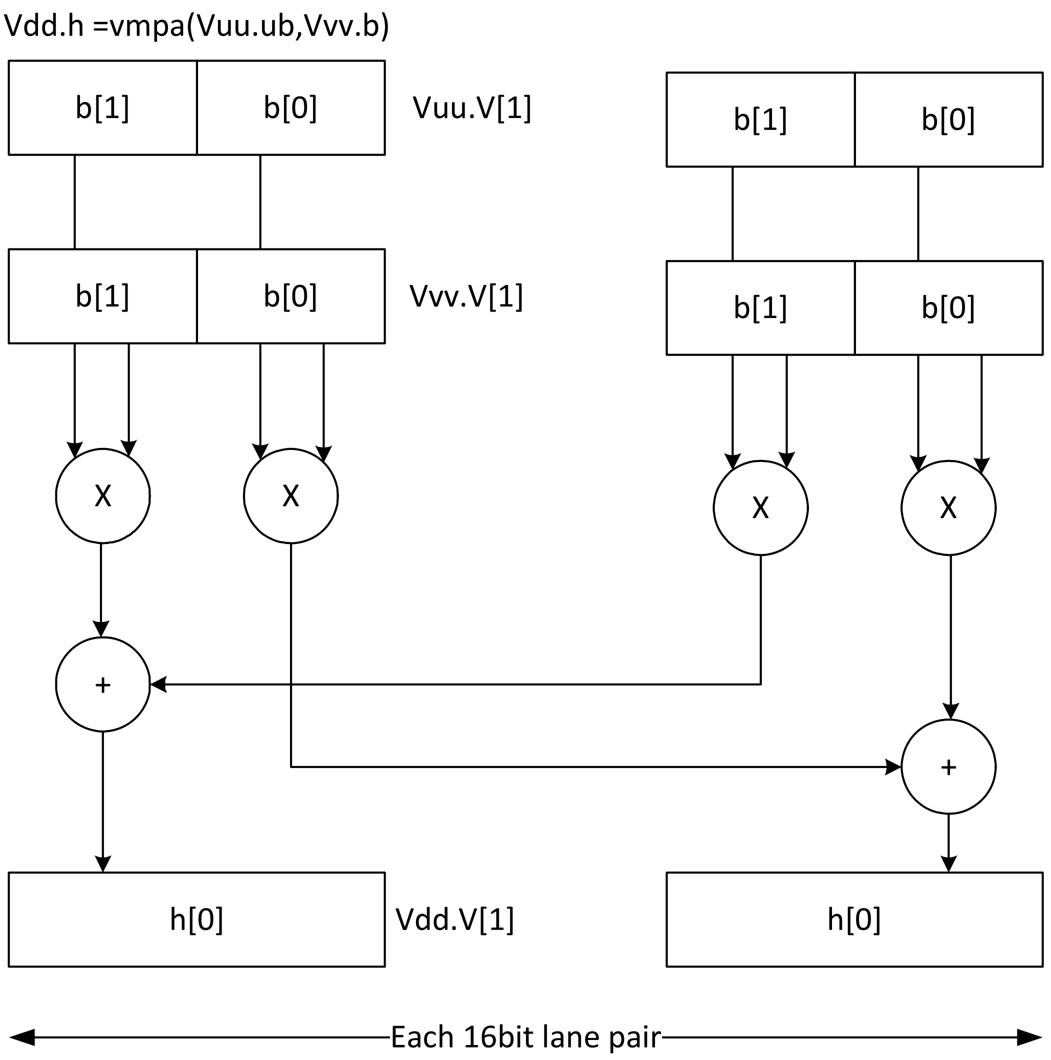
Multiply - add

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] amd 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.


Vdd.h [+]=vmpa(Vuu.ub,Rt.b)
[1]
[0]
Vuu.V[1]
[1]
[0]
Vuu.V[0]
Rt.b[3]
Rt.b[2]
Rt.b[1]
Rt.b[0]
X
X
X
X
+
+
[0]
Vdd.V[1]
[0]
Vdd.V[0]
Each lane



Syntax	Behavior

`Vdd.h=vmpa(Vuu.ub,Rt.b)`	`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]);`
	`}`

`Vdd.h=vmpa(Vuu.ub,Rt.ub)`	`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]);`
	`}`

`Vdd.h=vmpa(Vuu.ub,Vvv.b)`	`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

`Vdd.h=vmpa(Vuu.ub,Vvv.ub)`	`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]);`
	`}`

`Vdd.w=vmpa(Vuu.h,Rt.b)`	`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]);`
	`}`

`Vdd.w=vmpa(Vuu.uh,Rt.b)`	`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]);`
	`}`

`Vxx.h+=vmpa(Vuu.ub,Rt.b)`	`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]);`
	`}`

`Vxx.h+=vmpa(Vuu.ub,Rt.ub)`	`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]);`
	`}`

`Vxx.w+=vmpa(Vuu.h,Rt.b)`	`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]);`
	`}`

`Vxx.w+=vmpa(Vuu.uh,Rt.b)`	`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]);`
	`}`

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.
- This instruction may not work correctly in Napali V1.

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.uh,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.uh,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.uh,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.uh,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

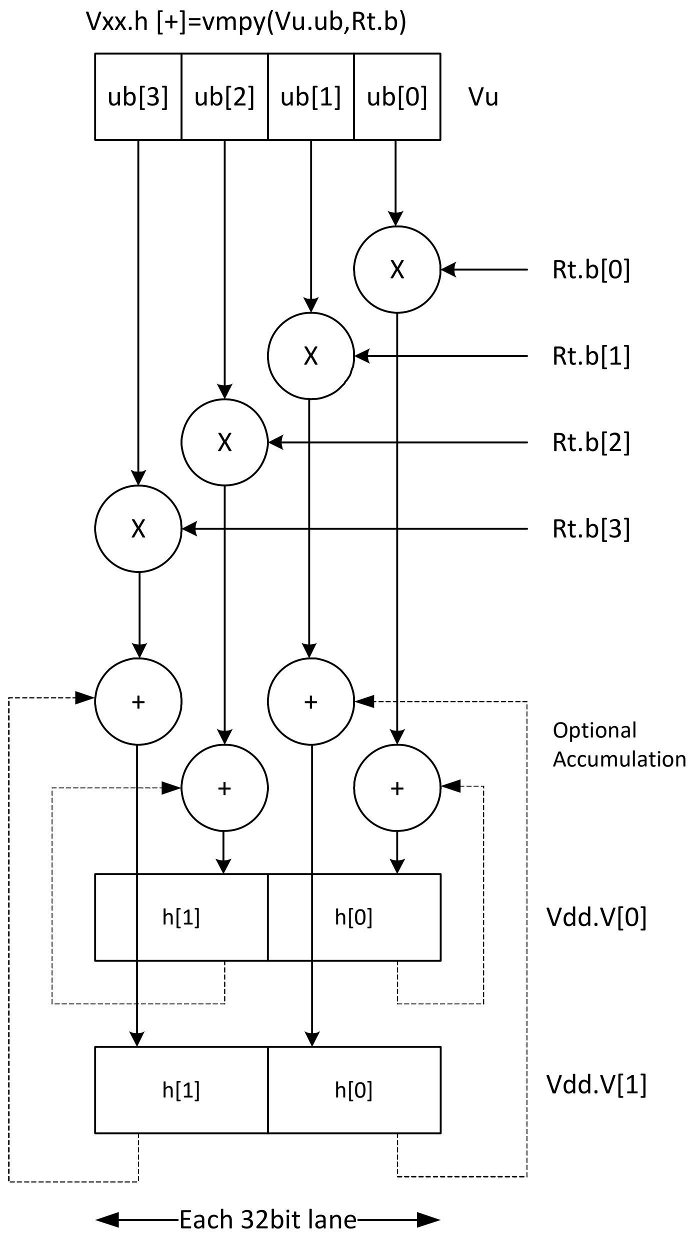
Multiply - vector by scalar

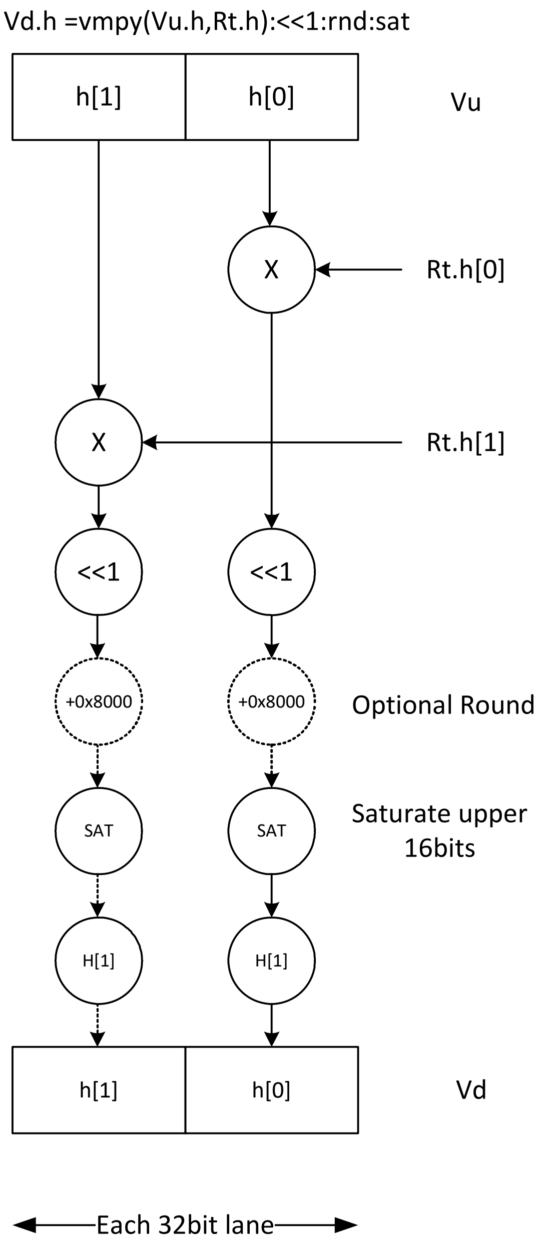
Multiply groups of elements in the vector Vu by the corresponding elements in the scalar register Rt.

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.

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





Syntax	Behavior

`Vd.h=vmpy(Vu.h,Rt.h):<<1:rnd:sat`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.w[i].h[0]=sat`$_{16}$`(sat`$_{32}$`(round(((Vu.w[i].h[0] * Rt.h[0])<<1))).h[1]);`
	`    Vd.w[i].h[1]=sat`$_{16}$`(sat`$_{32}$`(round(((Vu.w[i].h[1] * Rt.h[1])<<1))).h[1]);`
	`}`

Syntax	Behavior

`Vd.h=vmpy(Vu.h,Rt.h):<<1:sat`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.w[i].h[0]=sat`$_{16}$`(sat`$_{32}$`(((Vu.w[i].h[0] * Rt.h[0])<<1)).h[1]);`
	`    Vd.w[i].h[1]=sat`$_{16}$`(sat`$_{32}$`(((Vu.w[i].h[1] * Rt.h[1])<<1)).h[1]);`
	`}`

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

`Vdd.uh=vmpy(Vu.ub,Rt.ub)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vdd.v[0].uh[i] = (Vu.uh[i].ub[0] * Rt.ub[(2*i+0)%4]);`
	`    Vdd.v[1].uh[i] = (Vu.uh[i].ub[1] * Rt.ub[(2*i+1)%4]);`
	`}`

`Vdd.uw=vmpy(Vu.uh,Rt.uh)`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vdd.v[0].uw[i] = (Vu.uw[i].uh[0] * Rt.uh[0]);`
	`    Vdd.v[1].uw[i] = (Vu.uw[i].uh[1] * Rt.uh[1]);`
	`}`

`Vdd.w=vmpy(Vu.h,Rt.h)`	`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]);`
	`}`

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

`Vxx.uh+=vmpy(Vu.ub,Rt.ub)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vxx.v[0].uh[i] += (Vu.uh[i].ub[0] * Rt.ub[(2*i+0)%4]);`
	`    Vxx.v[1].uh[i] += (Vu.uh[i].ub[1] * Rt.ub[(2*i+1)%4]);`
	`}`

`Vxx.uw+=vmpy(Vu.uh,Rt.uh)`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vxx.v[0].uw[i] += (Vu.uw[i].uh[0] * Rt.uh[0]);`
	`    Vxx.v[1].uw[i] += (Vu.uw[i].uh[1] * Rt.uh[1]);`
	`}`

`Vxx.w+=vmpy(Vu.h,Rt.h)`	`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]);`
	`}`

Syntax	Behavior

`Vxx.w+=vmpy(Vu.h,Rt.h):sat`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vxx.v[0].w[i] = sat`$_{32}$`(Vxx.v[0].w[i].s64 + (Vu.w[i].h[0] * Rt.h[0]));`
	`    Vxx.v[1].w[i] = sat`$_{32}$`(Vxx.v[1].w[i].s64 + (Vu.w[i].h[1] * Rt.h[1]));`
	`}`

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.
- This instruction may not work correctly in Napali V1.

Intrinsics

`Vd.h=vmpy(Vu.h,Rt.h):<<1:rnd:sat`	`HVX_Vector Q6_Vh_vmpy_VhRh_s1_rnd_sat(HVX_Vector Vu, Word32 Rt)`
`Vd.h=vmpy(Vu.h,Rt.h):<<1:sat`	`HVX_Vector Q6_Vh_vmpy_VhRh_s1_sat(HVX_Vector Vu, Word32 Rt)`
`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	0	1	**d**	**d**	**d**	**d**	**d**	Vd.h=vmpy(Vu.h,Rt.h):<<1:sat
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):<<1:rnd:sat
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
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

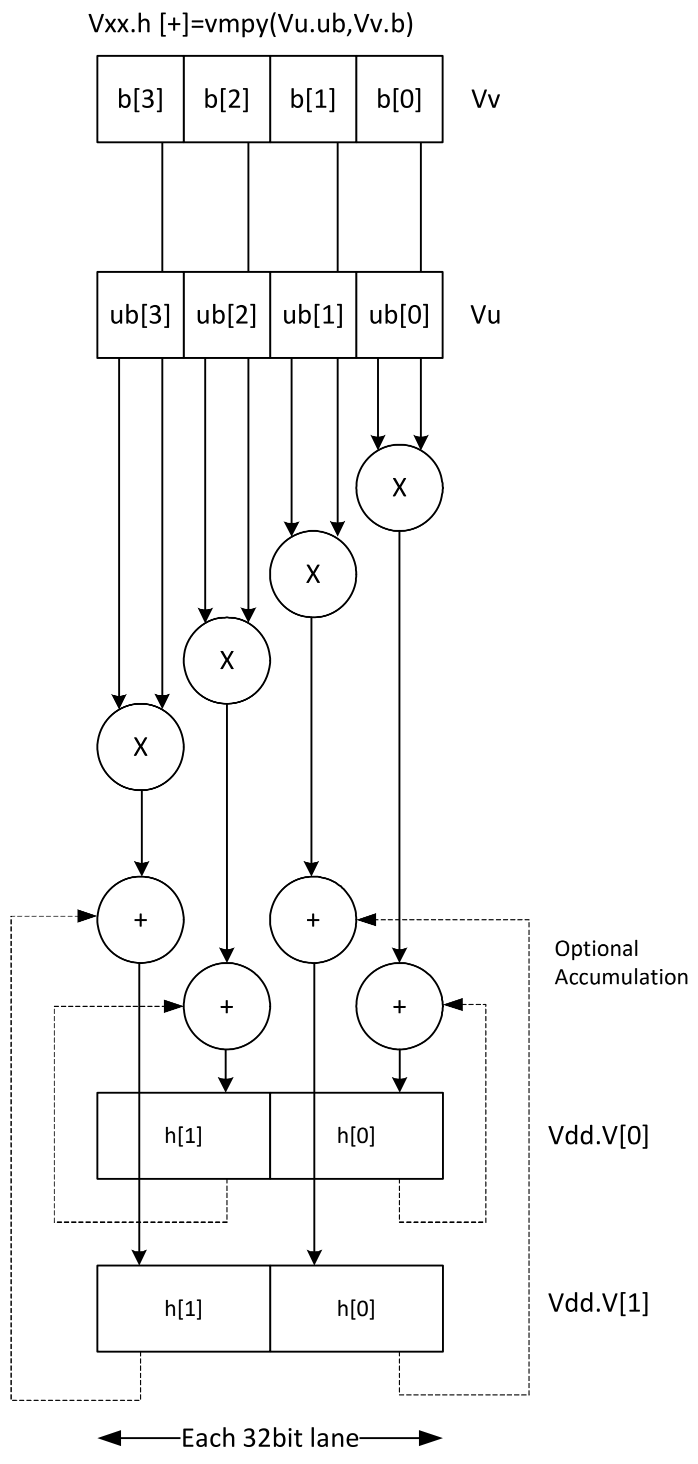
Multiply - vector by vector

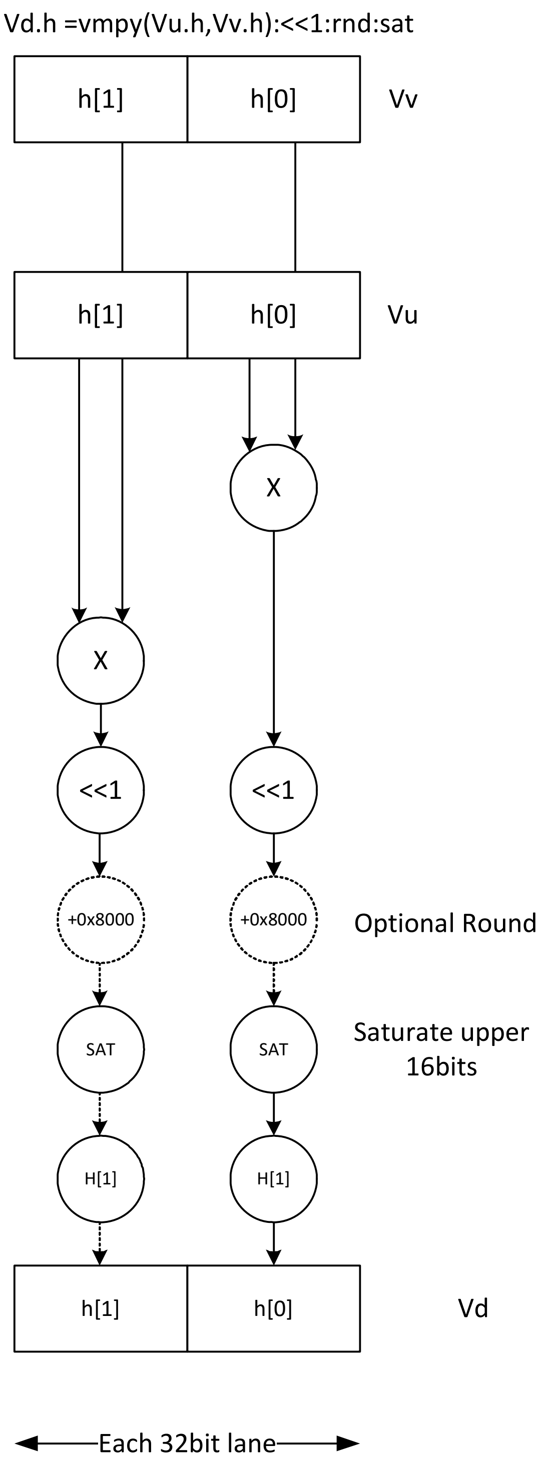
Multiply groups of elements in the vector Vu by the corresponding elements in the vector register Vv.

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 one, saturating the product to 32 bits, and placing the upper 16 bits in the output. Optional rounding of the result is supported.


Vxx.h [+]=vmpy(Vu.ub,Vv.b)
b[3] b[2] b[1] b[0]
Vv
ub[3] ub[2] ub[1] ub[0]
Vu
X
+
Optional Accumulation
h[1] h[0]
Vdd.V[0]
h[1] h[0]
Vdd.V[1]
Each 32bit lane


Vd.h =vmpy(Vu.h,Vv.h):<<1:rnd:sat
h[1] h[0]
Vv
h[1] h[0]
Vu
X
<<1
+0x8000
Optional Round
SAT
Saturate upper 16bits
H[1]
h[1] h[0]
Vd
Each 32bit lane

Syntax	Behavior

`Vd.h=vmpy(Vu.h,Vv.h):<<1:rnd:sat`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.h[i] = sat`$_{16}$`(sat`$_{32}$`(round(((Vu.h[i] *`
	`Vv.h[i])<<1))).h[1]);`
	`}`

`Vdd.h=vmpy(Vu.b,Vv.b)`	`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]);`
	`}`

`Vdd.h=vmpy(Vu.ub,Vv.b)`	`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]);`
	`}`

`Vdd.uh=vmpy(Vu.ub,Vv.ub)`	`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]);`
	`}`

`Vdd.uw=vmpy(Vu.uh,Vv.uh)`	`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]);`
	`}`

`Vdd.w=vmpy(Vu.h,Vv.h)`	`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]);`
	`}`

`Vdd.w=vmpy(Vu.h,Vv.uh)`	`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]);`
	`}`

`Vxx.h+=vmpy(Vu.b,Vv.b)`	`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]);`
	`}`

Syntax	Behavior
`Vxx.h+=vmpy(Vu.ub,Vv.b)`	`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]); }`
`Vxx.uh+=vmpy(Vu.ub,Vv.ub)`	`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]); }`
`Vxx.uw+=vmpy(Vu.uh,Vv.uh)`	`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]); }`
`Vxx.w+=vmpy(Vu.h,Vv.h)`	`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]); }`
`Vxx.w+=vmpy(Vu.h,Vv.uh)`	`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]); }`

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.

Intrinsics

`Vd.h=vmpy(Vu.h,Vv.h):<<1:rnd:sat`	`HVX_Vector Q6_Vh_vmpy_VhVh_s1_rnd_sat(HVX_Vector Vu, HVX_Vector Vv)`
`Vdd.h=vmpy(Vu.b,Vv.b)`	`HVX_VectorPair Q6_Wh_vmpy_VbVb(HVX_Vector Vu, HVX_Vector Vv)`
`Vdd.h=vmpy(Vu.ub,Vv.b)`	`HVX_VectorPair Q6_Wh_vmpy_VubVb(HVX_Vector Vu, HVX_Vector Vv)`
`Vdd.uh=vmpy(Vu.ub,Vv.ub)`	`HVX_VectorPair Q6_Wuh_vmpy_VubVub(HVX_Vector Vu, HVX_Vector Vv)`
`Vdd.uw=vmpy(Vu.uh,Vv.uh)`	`HVX_VectorPair Q6_Wuw_vmpy_VuhVuh(HVX_Vector Vu, HVX_Vector Vv)`
`Vdd.w=vmpy(Vu.h,Vv.h)`	`HVX_VectorPair Q6_Ww_vmpy_VhVh(HVX_Vector Vu, HVX_Vector Vv)`

`Vdd.w=vmpy(Vu.h,Vv.uh)`	`HVX_VectorPair Q6_Ww_vmpy_VhVuh(HVX_Vector Vu, HVX_Vector Vv)`
`Vxx.h+=vmpy(Vu.b,Vv.b)`	`HVX_VectorPair Q6_Wh_vmpyacc_WhVbVb(HVX_VectorPair Vxx, HVX_Vector Vu, HVX_Vector Vv)`
`Vxx.h+=vmpy(Vu.ub,Vv.b)`	`HVX_VectorPair Q6_Wh_vmpyacc_WhVubVb(HVX_VectorPair Vxx, HVX_Vector Vu, HVX_Vector Vv)`
`Vxx.uh+=vmpy(Vu.ub,Vv.ub)`	`HVX_VectorPair Q6_Wuh_vmpyacc_WuhVubVub(HVX_VectorPair Vxx, HVX_Vector Vu, HVX_Vector Vv)`
`Vxx.uw+=vmpy(Vu.uh,Vv.uh)`	`HVX_VectorPair Q6_Wuw_vmpyacc_WuwVuhVuh(HVX_VectorPair Vxx, HVX_Vector Vu, HVX_Vector Vv)`
`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

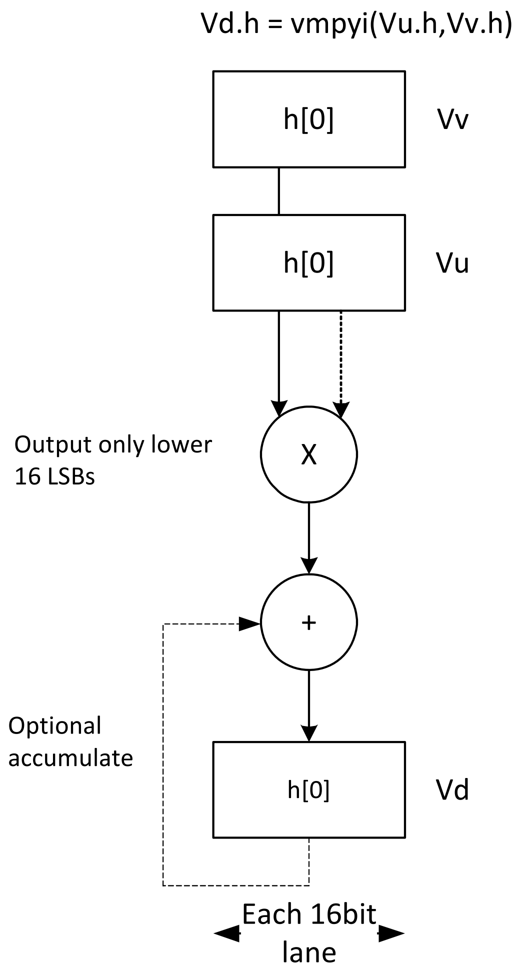
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse			u5								d5					
0	0	0	1	1	1	0	0	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vdd.h=vmpy(Vu.b,Vv.b)
0	0	0	1	1	1	0	0	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	1	d	d	d	d	d	Vdd.uh=vmpy(Vu.ub,Vv.ub)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

Field name	Description
`u5`	Field to encode register u
`v5`	Field to encode register v
`x5`	Field to encode register x

Integer multiply - vector by vector

Multiply corresponding elements in Vu by the corresponding elements in Vv, and place the lower half of the result in the destination vector register Vd. Supports signed halfwords, and optional accumulation of the product with the destination vector register Vx.



Syntax	Behavior

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

`Vx.h+=vmpyi(Vu.h,Vv.h)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vx.h[i] += (Vu.h[i] * Vv.h[i]);`
	`}`

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.

Intrinsics

`Vd.h=vmpyi(Vu.h,Vv.h)`	`HVX_Vector Q6_Vh_vmpyi_VhVh(HVX_Vector Vu, HVX_Vector Vv)`
`Vx.h+=vmpyi(Vu.h,Vv.h)`	`HVX_Vector Q6_Vh_vmpyiacc_VhVhVh(HVX_Vector Vx, 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	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
`v5`	Field to encode register v
`x5`	Field to encode register x

Integer multiply (32x16)

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 32x32 multiply synthesis.

Syntax	Behavior

`Vd.w=vmpyie(Vu.w,Vv.uh)`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.w[i] = (Vu.w[i] * Vv.w[i].uh[0]);`
	`}`

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

`Vx.w+=vmpyie(Vu.w,Vv.h)`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vx.w[i] = Vx.w[i] + (Vu.w[i] * Vv.w[i].h[0]);`
	`}`

`Vx.w+=vmpyie(Vu.w,Vv.uh)`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vx.w[i] = Vx.w[i] + (Vu.w[i] * Vv.w[i].uh[0]);`
	`}`

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.

Intrinsics

`Vd.w=vmpyie(Vu.w,Vv.uh)`	`HVX_Vector Q6_Vw_vmpyie_VwVuh(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.w=vmpyio(Vu.w,Vv.h)`	`HVX_Vector Q6_Vw_vmpyio_VwVh(HVX_Vector Vu, HVX_Vector Vv)`
`Vx.w+=vmpyie(Vu.w,Vv.h)`	`HVX_Vector Q6_Vw_vmpyieacc_VwVwVh(HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv)`
`Vx.w+=vmpyie(Vu.w,Vv.uh)`	`HVX_Vector Q6_Vw_vmpyieacc_VwVwVuh(HVX_Vector Vx, 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	1	0	1	x	x	x	x	x	Vx.w+=vmpyie(Vu.w,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	0	0	0	x	x	x	x	x	Vx.w+=vmpyie(Vu.w,Vv.h)
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	Vd.w=vmpyie(Vu.w,Vv.uh)
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	Vd.w=vmpyio(Vu.w,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

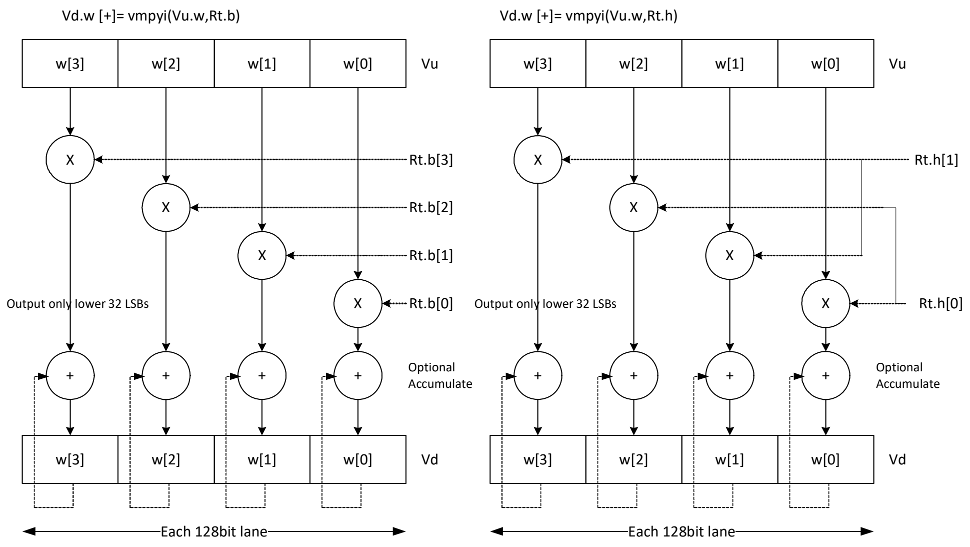
Integer multiply accumulate even/odd

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 one form: Signed words multiplied by halfwords in Rt.

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	Behavior

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

`Vx.w+=vmpyi(Vu.w,Rt.h)`	`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)`	`HVX_Vector Q6_Vw_vmpyi_VwRh(HVX_Vector Vu, Word32 Rt)`
`Vx.w+=vmpyi(Vu.w,Rt.h)`	`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 (32x16)

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 32x32 multiply synthesis.

An important operation is a 32 x 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 }
```

and the non rounding 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 }
```

Also a key function is a 32bit x 32bit signed multiply where the 64bit 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 x 16-bit unsigned asserted into the upper 48 bits of Vdd. If vmpyo only is performed, assuming Vxx = #0, the result is a 32 signed x 16 signed product asserted into the upper 48 bits of Vxx.

Syntax	Behavior

`Vd.w=vmpye(Vu.w,Vv.uh)`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.w[i] = (Vu.w[i] * Vv.w[i].uh[0]) >> 16;`
	`}`

`Vd.w=vmpyo(Vu.w,Vv.h):<<1[:rnd]:sat`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.w[i] = sat32((((Vu.w[i] * Vv.w[i].h[1])`
	`>> 14) + 1) >> 1));`
	`}`

`Vdd=vmpye(Vu.w,Vv.uh)`	`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;`
	`}`

`Vx.w+=vmpyo(Vu.w,Vv.h):<<1[:rnd]:sat:shift`	`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));`
	`}`

`Vxx+=vmpyo(Vu.w,Vv.h)`	`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;`
	`}`

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.

Intrinsics

`Vd.w=vmpye(Vu.w,Vv.uh)`	`HVX_Vector Q6_Vw_vmpye_VwVuh(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.w=vmpyo(Vu.w,Vv.h):<<1:rnd:sat`	`HVX_Vector Q6_Vw_vmpyo_VwVh_s1_rnd_sat(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.w=vmpyo(Vu.w,Vv.h):<<1:sat`	`HVX_Vector Q6_Vw_vmpyo_VwVh_s1_sat(HVX_Vector Vu, HVX_Vector Vv)`
`Vdd=vmpye(Vu.w,Vv.uh)`	`HVX_VectorPair Q6_W_vmpye_VwVuh(HVX_Vector Vu, HVX_Vector Vv)`
`Vx.w+=vmpyo(Vu.w,Vv.h):<<1:rnd:sat:shift`	`HVX_Vector Q6_Vw_vmpyoacc_VwVwVh_s1_rnd_sat_shift(HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv)`
`Vx.w+=vmpyo(Vu.w,Vv.h):<<1:sat:shift`	`HVX_Vector Q6_Vw_vmpyoacc_VwVwVh_s1_sat_shift(HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv)`
`Vxx+=vmpyo(Vu.w,Vv.h)`	`HVX_VectorPair Q6_W_vmpyoacc_WVwVh(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	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
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

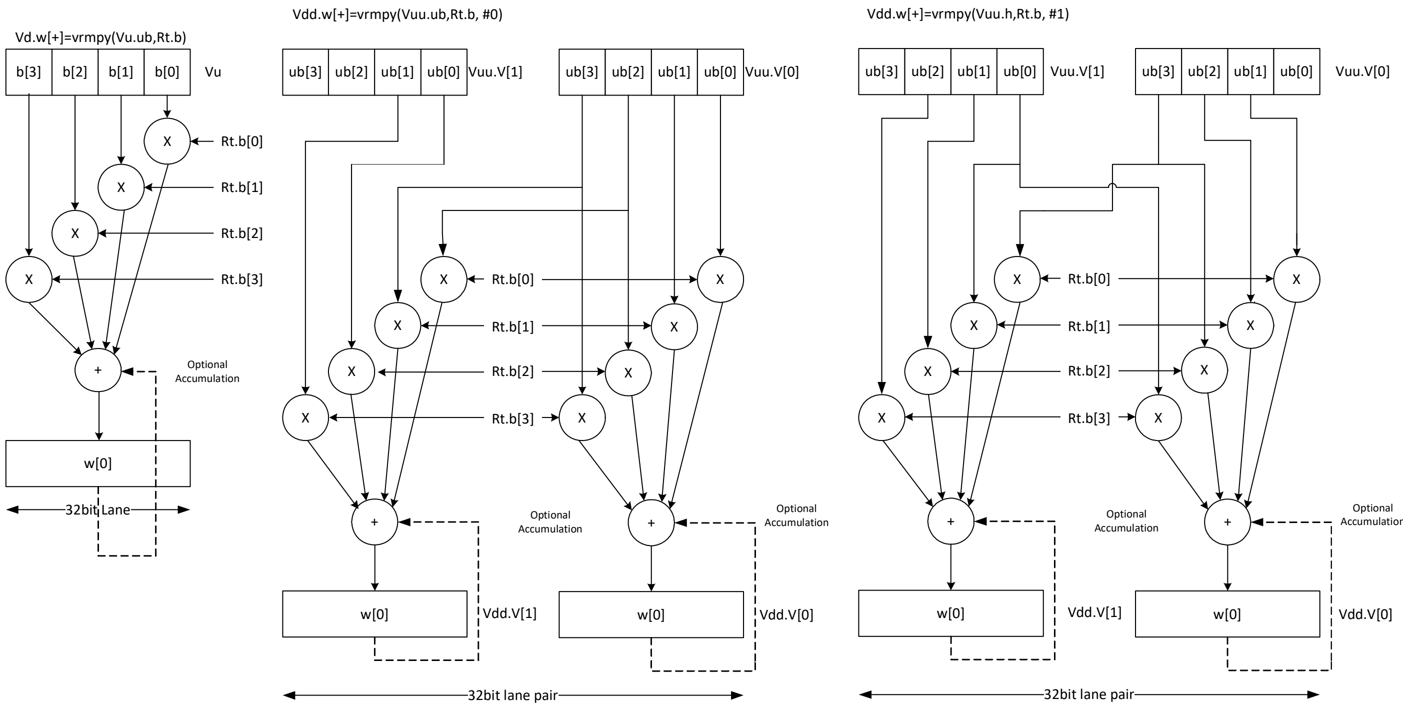
Field name	Description
`u5`	Field to encode register u
`v5`	Field to encode register v
`x5`	Field to encode register x

Multiply bytes with four-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 four-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 three from the even register of Vuu.



Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.

Syntax	Behavior

Vdd.uw=vrmpy(Vuu.ub,Rt.ub,#u1)	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]);
	}

Vdd.w=vrmpy(Vuu.ub,Rt.b,#u1)	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]);
	}

Vxx.uw+=vrmpy(Vuu.ub,Rt.ub,#u1)	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]);
	}

Syntax **Behavior**

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

```
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_WubRubI(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_WuwWubRubI(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)

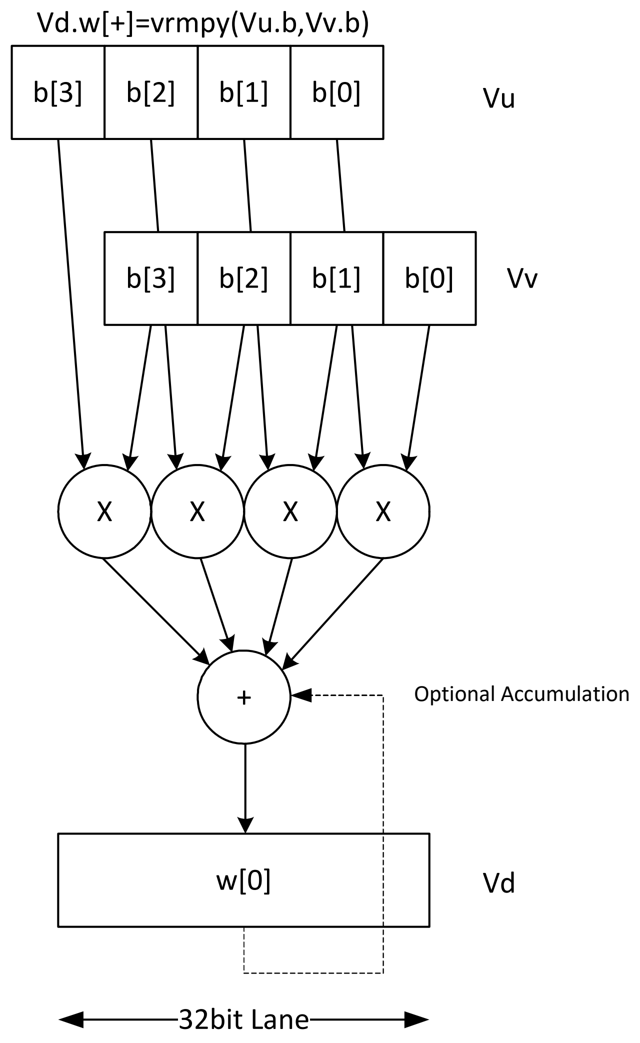
Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d

Field name	Description
`t5`	Field to encode register t
`u5`	Field to encode register u
`x5`	Field to encode register x

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

vrmpy performs a dot product function between four byte elements in vector register Vu and four 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	Behavior

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

`Vx.w+=vrmpy(Vu.b,Vv.b)`	`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]);`
	`}`

`Vx.w+=vrmpy(Vu.ub,Vv.b)`	`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]);`
	`}`

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.

Intrinsics

`Vx.uw+=vrmpy(Vu.ub,Vv.ub)`	`HVX_Vector Q6_Vuw_vrmpyacc_VuwVubVub(HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv)`
`Vx.w+=vrmpy(Vu.b,Vv.b)`	`HVX_Vector Q6_Vw_vrmpyacc_VwVbVb(HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv)`
`Vx.w+=vrmpy(Vu.ub,Vv.b)`	`HVX_Vector Q6_Vw_vrmpyacc_VwVubVb(HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv)`

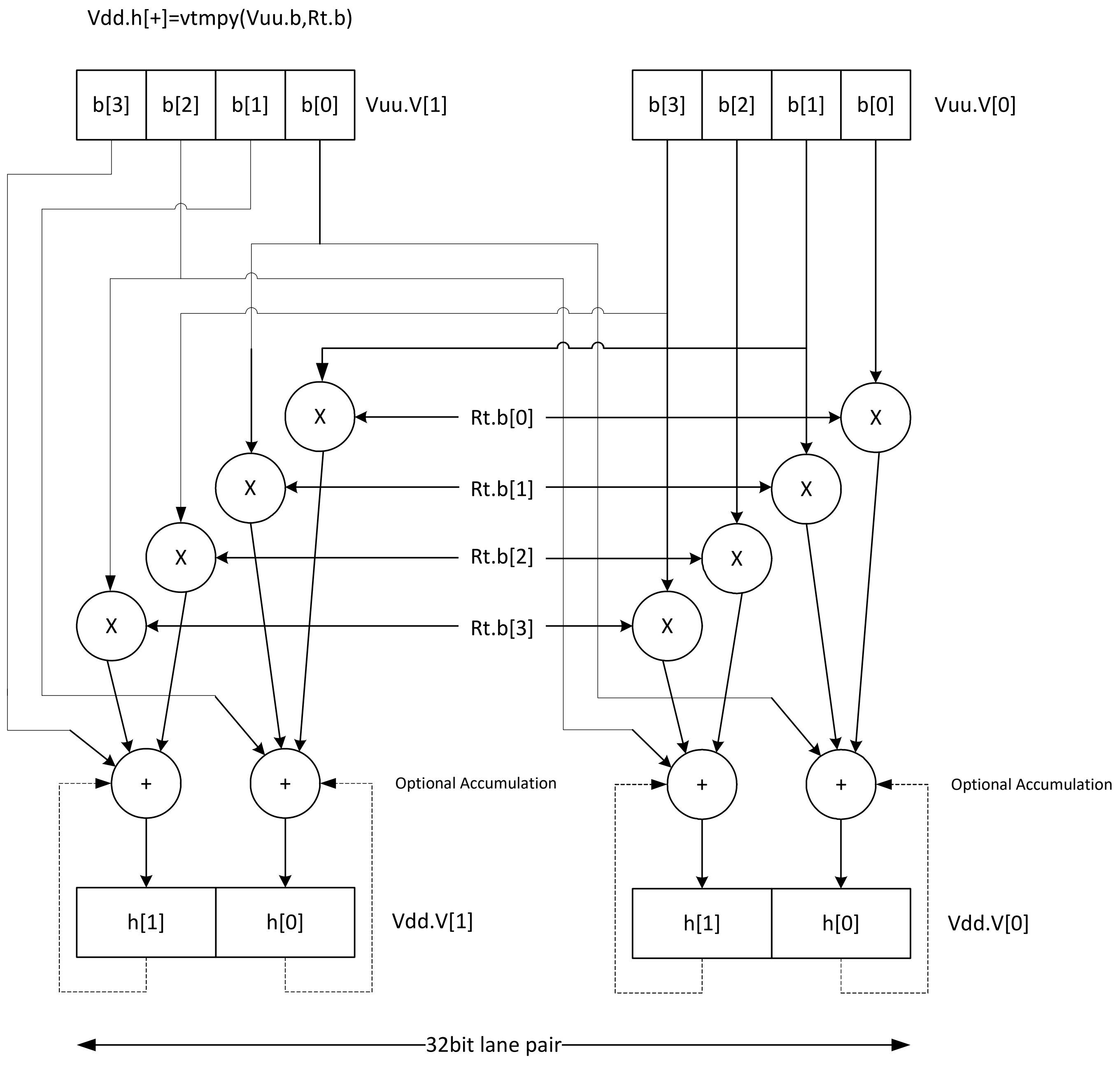
Encoding

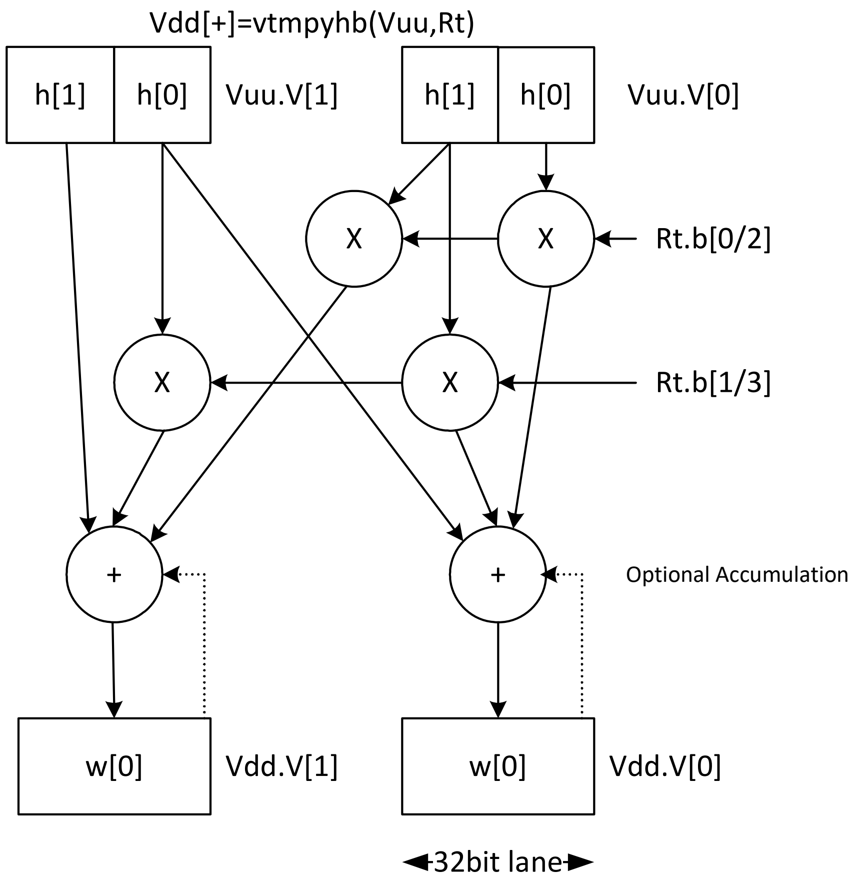
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse			u5								x5					
0	0	0	1	1	1	0	0	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	x	x	x	x	x	Vx.uw+=vrmpy(Vu.ub,Vv.ub)
0	0	0	1	1	1	0	0	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	x	x	x	x	x	Vx.w+=vrmpy(Vu.b,Vv.b)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

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 three-wide reduction

Perform a three-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.





Syntax	Behavior

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

`Vdd.h=vtmpy(Vuu.ub,Rt.b)`	
	```
	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];
	}
	```

Syntax	Behavior
`Vdd.w=vtmpy(Vuu.h,Rt.b)`	`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];` `}`
`Vxx.h+=vtmpy(Vuu.b,Rt.b)`	`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];` `}`
`Vxx.h+=vtmpy(Vuu.ub,Rt.b)`	`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];` `}`
`Vxx.w+=vtmpy(Vuu.h,Rt.b)`	`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];` `}`

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

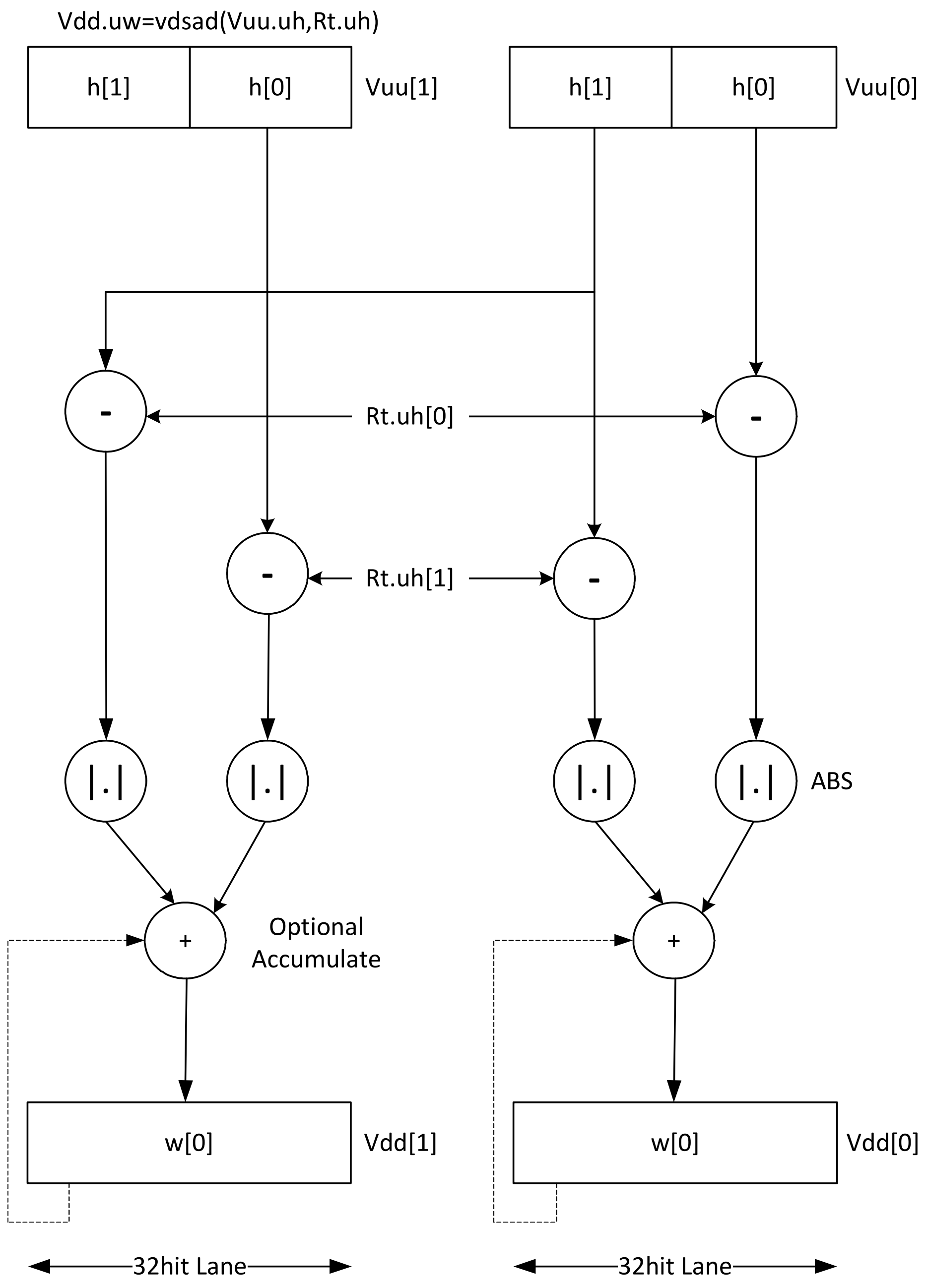
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											t5					Parse			u5								d5					
0	0	0	1	1	0	0	1	0	0	0	**t**	**t**	**t**	**t**	**t**	**P**	**P**	0	**u**	**u**	**u**	**u**	**u**	0	0	0	**d**	**d**	**d**	**d**	**d**	Vdd.h=vtmpy(Vuu.b,Rt.b)
0	0	0	1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


Vdd.uw=vdsad(Vuu.uh,Rt.uh)
h[1]
h[0]
Vuu[1]
h[1]
h[0]
Vuu[0]
-
Rt.uh[0]
-
-
Rt.uh[1]
-
\|.\|
\|.\|
\|.\|
\|.\|
ABS
+
Optional Accumulate
+
w[0]
Vdd[1]
w[0]
Vdd[0]
32hit Lane
32hit Lane

Syntax	Behavior

`Vdd.uw=vdsad(Vuu.uh,Rt.uh)`	`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]);`
	`}`

`Vxx.uw+=vdsad(Vuu.uh,Rt.uh)`	`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]);`
	`}`

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.

Intrinsics

`Vdd.uw=vdsad(Vuu.uh,Rt.uh)`	`HVX_VectorPair Q6_Wuw_vdsad_WuhRuh(HVX_VectorPair Vuu, Word32 Rt)`
`Vxx.uw+=vdsad(Vuu.uh,Rt.uh)`	`HVX_VectorPair Q6_Wuw_vdsadacc_WuwWuhRuh(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	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)

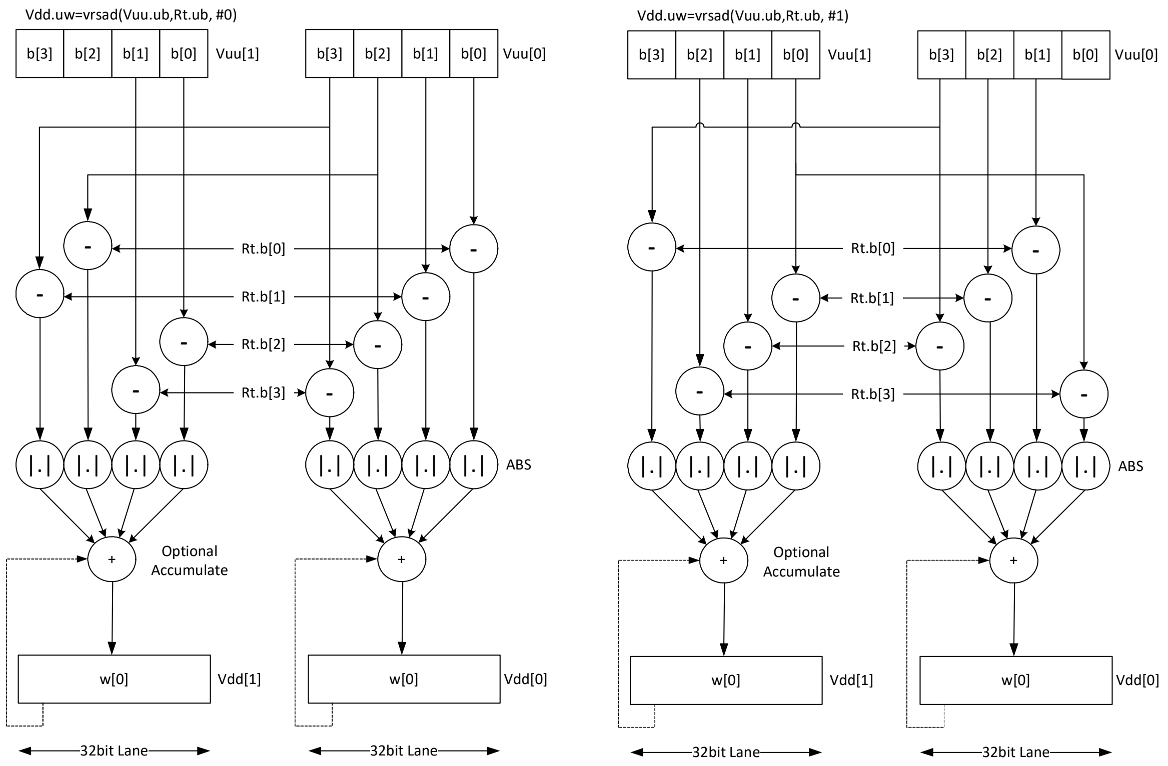
Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d

Field name	Description
`t5`	Field to encode register t
`u5`	Field to encode register u
`x5`	Field to encode register x

Sum of absolute differences byte

Take groups of four bytes from the vector register source Vuu, subtract 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 btye 0 from Vuu[1] and bytes 1,2,3 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.



Syntax	Behavior

`Vdd.uw=vrsad(Vuu.ub,Rt.ub,#u1)`	`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]);`
	`}`

`Vxx.uw+=vrsad(Vuu.ub,Rt.ub,#u1)`	`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]);`
	`}`

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses both HVX multiply resources.

Intrinsics

`Vdd.uw=vrsad(Vuu.ub,Rt.ub,#u1)`	`HVX_VectorPair Q6_Wuw_vrsad_WubRubI(HVX_VectorPair Vuu, Word32 Rt, Word32 Iu1)`
`Vxx.uw+=vrsad(Vuu.ub,Rt.ub,#u1)`	`HVX_VectorPair Q6_Wuw_vrsadacc_WuwWubRubI(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	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

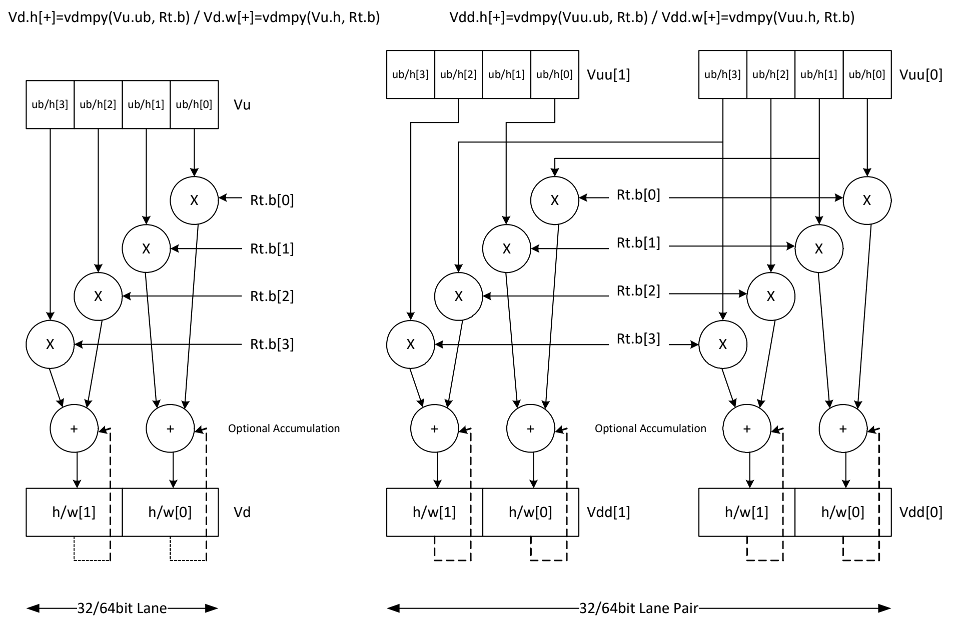
5.9 HVX/MPY-RESOURCE

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

Multiply by byte with two-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-two 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 two-way reduction, where the odd register output contains the offset computation.



Syntax	Behavior

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

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

Syntax	Behavior

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

`Vx.w+=vdmpy(Vu.h,Rt.b)`	`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]);`
	`}`

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses a HVX multiply resource.

Intrinsics

`Vd.h=vdmpy(Vu.ub,Rt.b)`	`HVX_Vector Q6_Vh_vdmpy_VubRb(HVX_Vector Vu, Word32 Rt)`
`Vd.w=vdmpy(Vu.h,Rt.b)`	`HVX_Vector Q6_Vw_vdmpy_VhRb(HVX_Vector Vu, Word32 Rt)`
`Vx.h+=vdmpy(Vu.ub,Rt.b)`	`HVX_Vector Q6_Vh_vdmpyacc_VhVubRb(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)`
`Vx.w+=vdmpy(Vu.h,Rt.b)`	`HVX_Vector Q6_Vw_vdmpyacc_VwVhRb(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	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 half of the elements (16x16)

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 32x32 low-half multiplies.

Syntax	Behavior

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

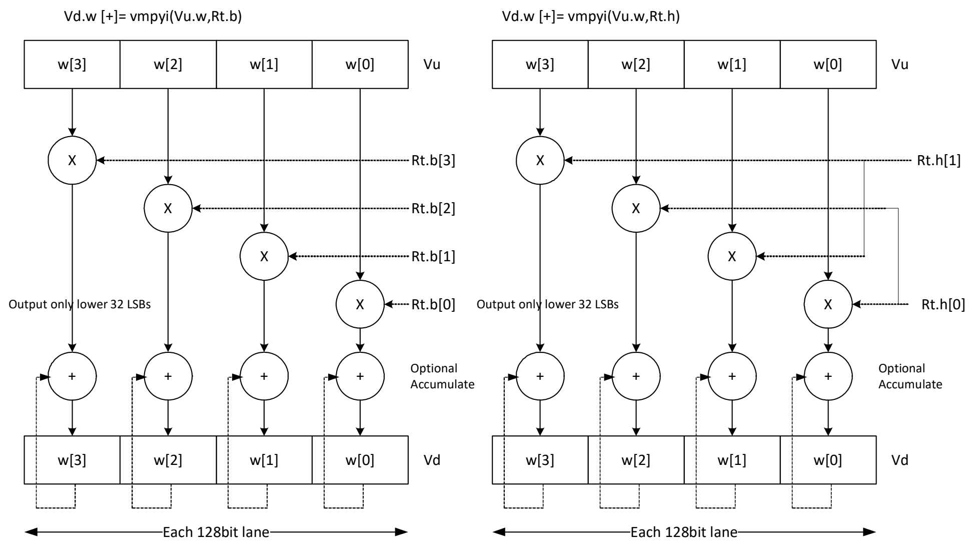
Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`u5`	Field to encode register u
`v5`	Field to encode register v

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	Behavior

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

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

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

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

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

`Vx.w+=vmpyi(Vu.w,Rt.ub)`	`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 x 16)

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

Syntax	Behavior

`Vd.uw=vmpye(Vu.uh,Rt.uh)`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.uw[i] = (Vu.uw[i].uh[0] * Rt.uh[0]);`
	`}`

`Vx.uw+=vmpye(Vu.uh,Rt.uh)`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vx.uw[i] += (Vu.uw[i].uh[0] * Rt.uh[0]);`
	`}`

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses a HVX multiply resource.
- This instruction may not work correctly in Napali V1.

Intrinsics

`Vd.uw=vmpye(Vu.uh,Rt.uh)`	`HVX_Vector Q6_Vuw_vmpye_VuhRuh(HVX_Vector Vu, Word32 Rt)`
`Vx.uw+=vmpye(Vu.uh,Rt.uh)`	`HVX_Vector Q6_Vuw_vmpyeacc_VuwVuhRuh(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	1	1	t	t	t	t	t	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d	Vd.uw=vmpye(Vu.uh,Rt.uh)
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	1	x	x	x	x	x	Vx.uw+=vmpye(Vu.uh,Rt.uh)

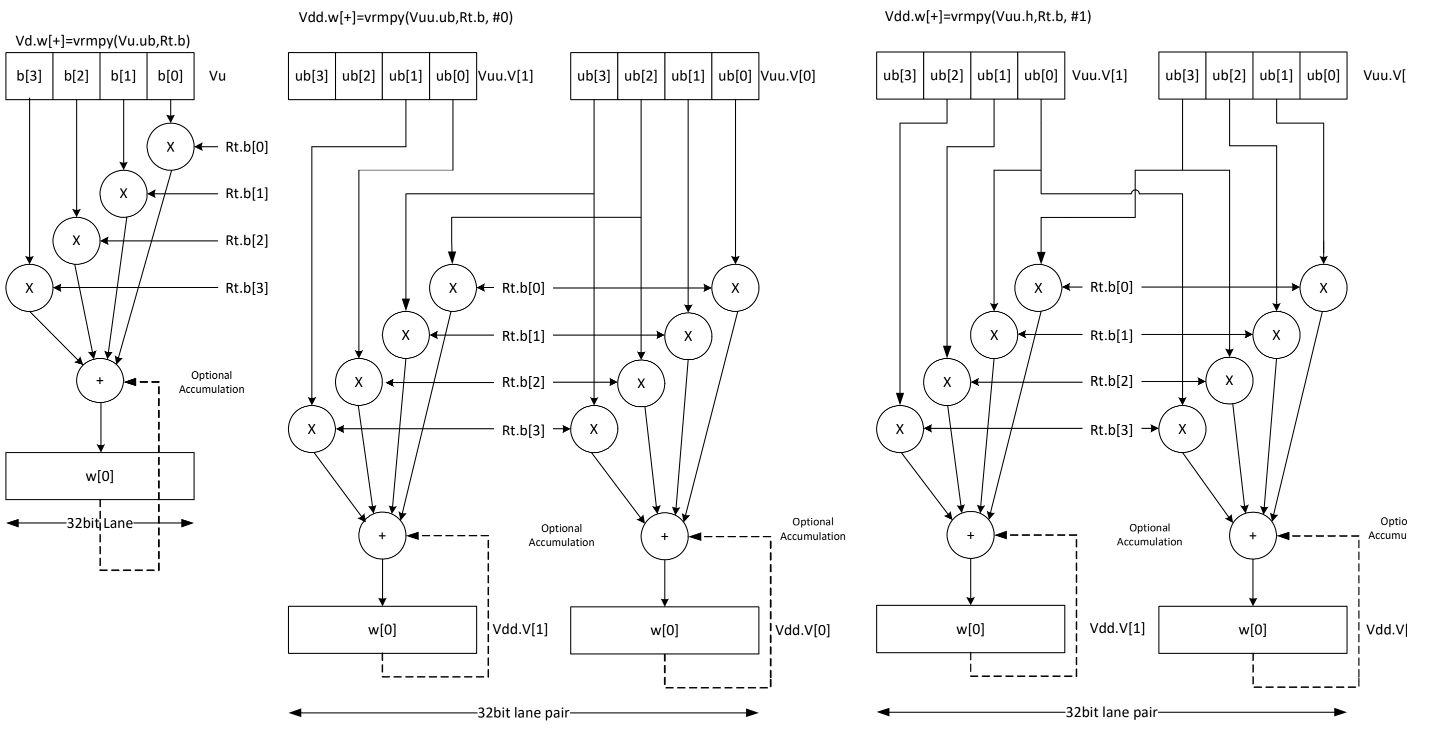
Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`t5`	Field to encode register t
`u5`	Field to encode register u
`x5`	Field to encode register x

Multiply bytes with four-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 four-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 three from the even register of Vuu.



Syntax	Behavior

`Vd.uw=vrmpy(Vu.ub,Rt.ub)`	`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]);`
	`}`

`Vd.w=vrmpy(Vu.ub,Rt.b)`	`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]);`
	`}`

`Vx.uw+=vrmpy(Vu.ub,Rt.ub)`	`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]);`
	`}`

`Vx.w+=vrmpy(Vu.ub,Rt.b)`	`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]);`
	`}`

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses a HVX multiply resource.

Intrinsics

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

Encoding

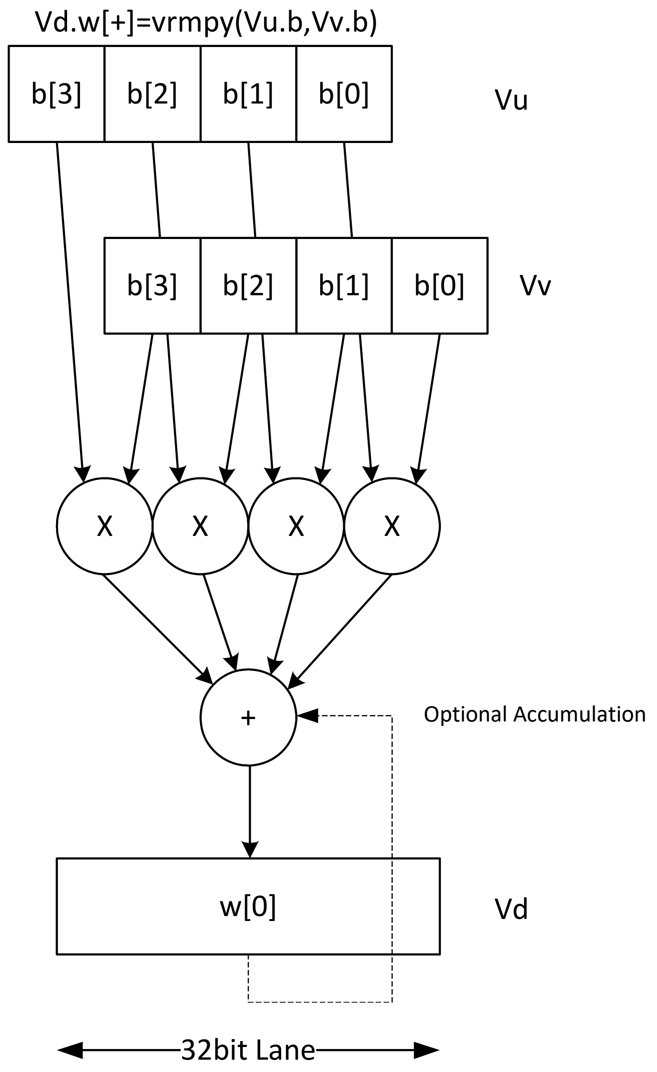
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											t5					Parse			u5								d5					
0	0	0	1	1	0	0	1	0	0	0	**t**	**t**	**t**	**t**	**t**	**P**	**P**	0	**u**	**u**	**u**	**u**	**u**	0	1	1	**d**	**d**	**d**	**d**	**d**	Vd.uw=vrmpy(Vu.ub,Rt.ub)
0	0	0	1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

vrmpy performs a dot product function between four-byte elements in vector register Vu, and four-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	Behavior

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

`Vd.w=vrmpy(Vu.b,Vv.b)`	`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]);`
	`}`

`Vd.w=vrmpy(Vu.ub,Vv.b)`	`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]);`
	`}`

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses a HVX multiply resource.

Intrinsics

`Vd.uw=vrmpy(Vu.ub,Vv.ub)`	`HVX_Vector Q6_Vuw_vrmpy_VubVub(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.w=vrmpy(Vu.b,Vv.b)`	`HVX_Vector Q6_Vw_vrmpy_VbVb(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.w=vrmpy(Vu.ub,Vv.b)`	`HVX_Vector Q6_Vw_vrmpy_VubVb(HVX_Vector Vu, HVX_Vector Vv)`

Encoding

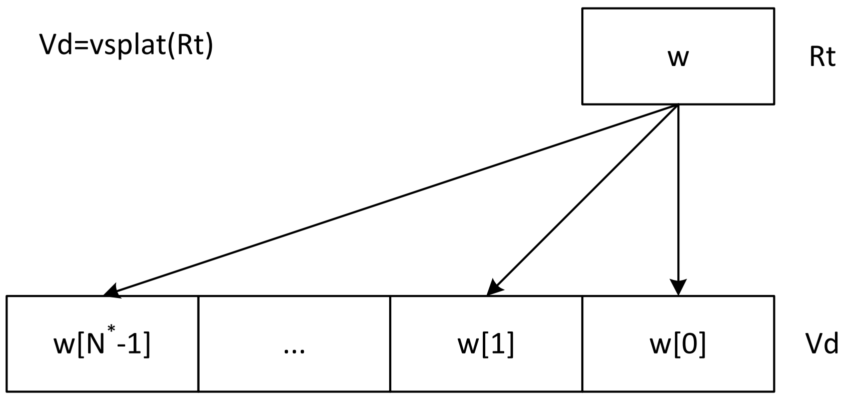
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse			u5								d5					
0	0	0	1	1	1	0	0	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.uw=vrmpy(Vu.ub,Vv.ub)
0	0	0	1	1	1	0	0	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.w=vrmpy(Vu.b,Vv.b)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

Field name	Description
`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.



*N number of operations in vector

Syntax	Behavior

`Vd.b=vsplat(Rt)`	`for (i = 0; i < VELEM(8); i++) {`
	`    Vd.ub[i] = Rt;`
	`}`

`Vd.h=vsplat(Rt)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.uh[i] = Rt;`
	`}`

`Vd=vsplat(Rt)`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.uw[i] = Rt;`
	`}`

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses a HVX multiply resource.

Intrinsics

`Vd.b=vsplat(Rt)`	`HVX_Vector Q6_Vb_vsplat_R(Word32 Rt)`
`Vd.h=vsplat(Rt)`	`HVX_Vector Q6_Vh_vsplat_R(Word32 Rt)`
`Vd=vsplat(Rt)`	`HVX_Vector Q6_V_vsplat_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											d5					
0	0	0	1	1	0	0	1	1	0	1	**t**	**t**	**t**	**t**	**t**	**P**	**P**	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 then Qt can effectively be filled with the lsb's of Vu, one bit per byte.

Syntax	Behavior

`Qd4=vand(Vu,Rt)`	`for (i = 0; i < VELEM(8); i++) {`
	`    QdV[i]=((Vu.ub[i] & Rt.ub[i % 4]) != 0) ? 1 : 0;`
	`}`

`Qx4|=vand(Vu,Rt)`	`for (i = 0; i < VELEM(8); i++) {`
	`    QxV[i]=QxV[i]|(((Vu.ub[i] & Rt.ub[i % 4]) != 0) ? 1 : 0);`
	`}`

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses a HVX multiply resource.

Intrinsics

`Qd4=vand(Vu,Rt)`	`HVX_VectorPred Q6_Q_vand_VR(HVX_Vector Vu, Word32 Rt)`
`Qx4|=vand(Vu,Rt)`	`HVX_VectorPred Q6_Q_vandor_QVR(HVX_VectorPred Qx, 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											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 then Qt can effectively be expanded into Vd or Vx, one bit per byte.

Syntax	Behavior

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

`Vx|=vand([!]Qu4,Rt)`	`for (i = 0; i < VELEM(8); i++) {`
	`    Vx.ub[i] |= [!](QuV[i]) ? Rt.ub[i % 4] : 0;`
	`}`

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)`

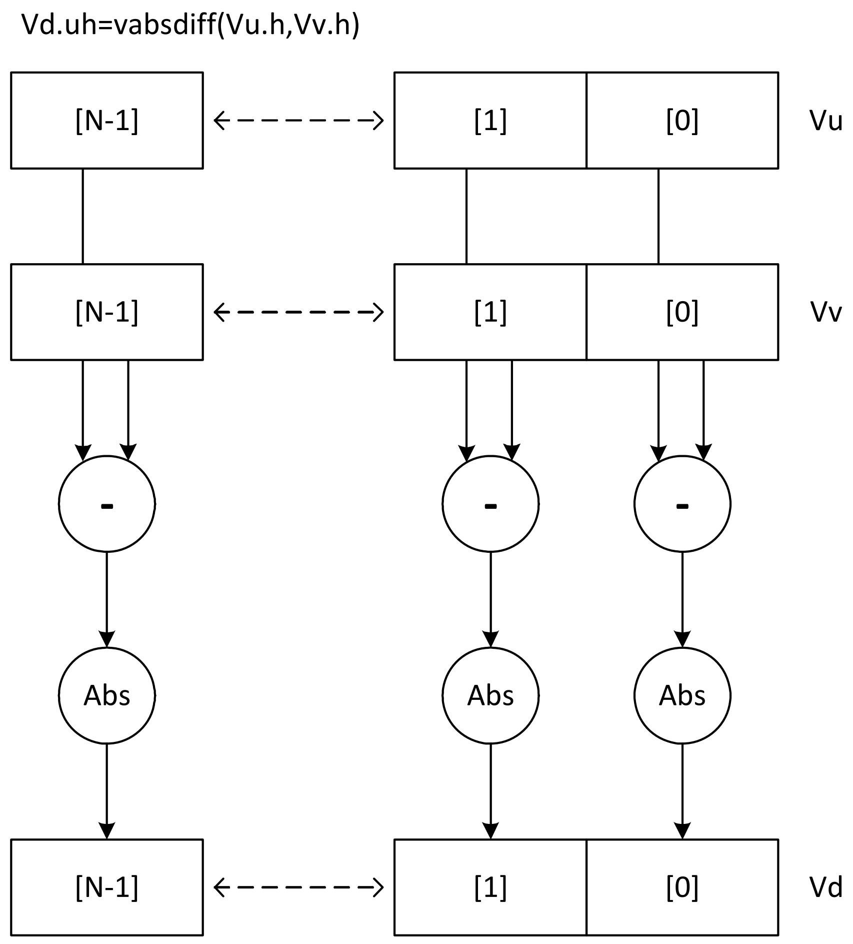
Encoding

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

`Vd.ub=vabsdiff(Vu.ub,Vv.ub)`	`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]);`
	`}`

`Vd.uh=vabsdiff(Vu.h,Vv.h)`	`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]);`
	`}`

`Vd.uh=vabsdiff(Vu.uh,Vv.uh)`	`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]);`
	`}`

`Vd.uw=vabsdiff(Vu.w,Vv.w)`	`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	Behavior
`Vx.w=vinsert(Rt)`	`Vx.uw[0] = Rt;`

Class: COPROC_VX (slots 2,3)

Notes

- This instruction uses a HVX multiply resource.

Intrinsics

`Vx.w=vinsert(Rt)`	`HVX_Vector Q6_Vw_vinsert_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=vinsert(Rt)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`t5`	Field to encode register t
`x5`	Field to encode register x

5.10 HVX/PERMUTE-RESOURCE

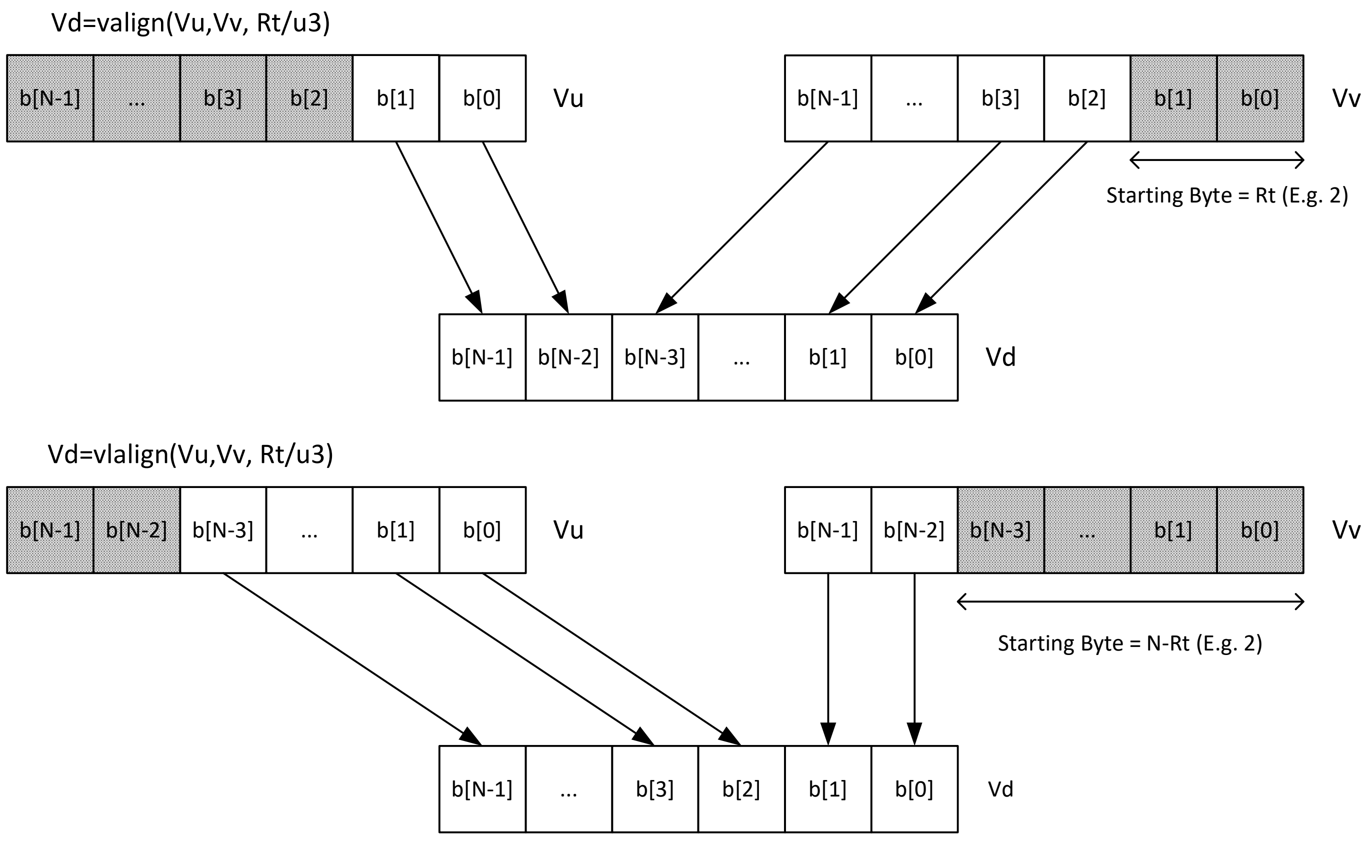
The HVX/PERMUTE-RESOURCE instruction subclass includes instructions that use the HVX permute resource.

Byte alignment

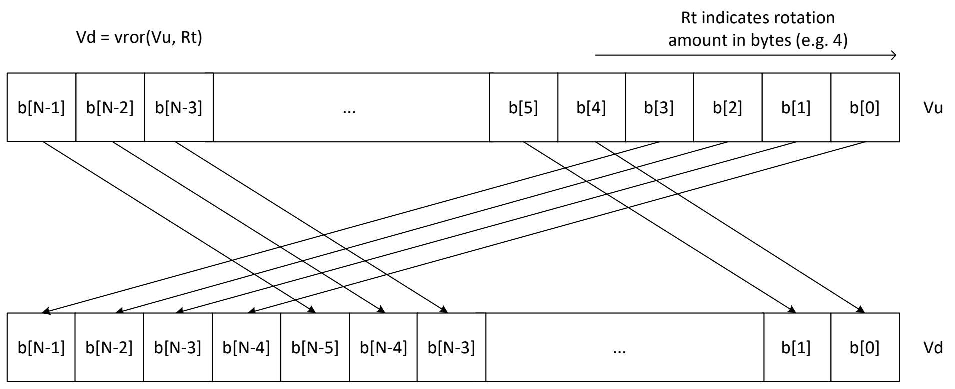
Select a continuous group of bytes the size of a vector register from vector registers Vu and Vv. The starting location is provided by the lower bits of Rt (modulo the vector length) or by a 3-bit immediate value.

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.


Vd=valign(Vu,Vv, Rt/u3)
b[N-1] ... b[3] b[2] b[1] b[0] Vu
b[N-1] ... b[3] b[2] b[1] b[0] Vv
Starting Byte = Rt (E.g. 2)
b[N-1] b[N-2] b[N-3] ... b[1] b[0] Vd
Vd=vlalign(Vu,Vv, Rt/u3)
b[N-1] b[N-2] b[N-3] ... b[1] b[0] Vu
b[N-1] b[N-2] b[N-3] ... b[1] b[0] Vv
Starting Byte = N-Rt (E.g. 2)
b[N-1] ... b[3] b[2] b[1] b[0] Vd

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

`Vd=valign(Vu,Vv,#u3)`	`HVX_Vector Q6_V_valign_VVI(HVX_Vector Vu, HVX_Vector Vv, Word32 Iu3)`
`Vd=valign(Vu,Vv,Rt)`	`HVX_Vector Q6_V_valign_VVR(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)`
`Vd=vlalign(Vu,Vv,#u3)`	`HVX_Vector Q6_V_vlalign_VVI(HVX_Vector Vu, HVX_Vector Vv, Word32 Iu3)`
`Vd=vlalign(Vu,Vv,Rt)`	`HVX_Vector Q6_V_vlalign_VVR(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)`
`Vd=vror(Vu,Rt)`	`HVX_Vector Q6_V_vror_VR(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	1	1	t	t	t	t	t	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd=vror(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)

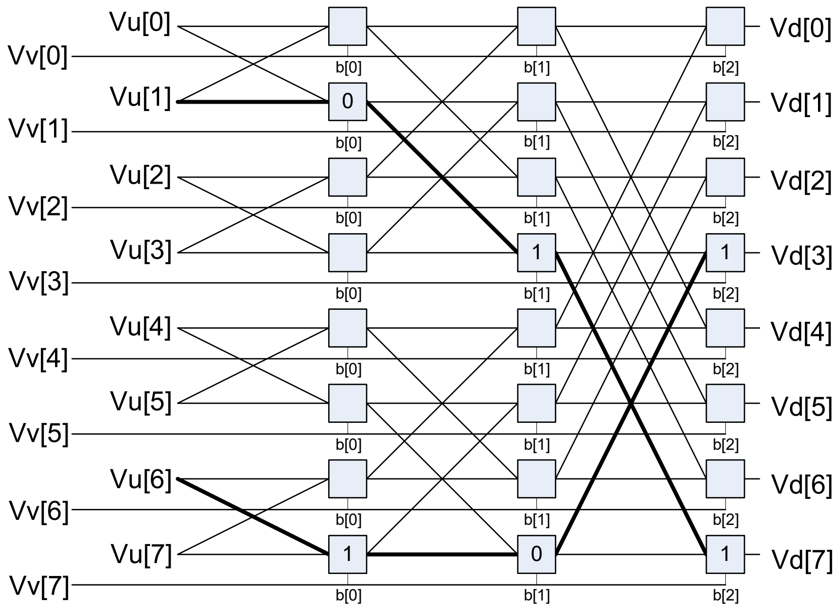
Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`t3`	Field to encode register t
`t5`	Field to encode register t
`u5`	Field to encode register u
`v2`	Field to encode register v
`v3`	Field to encode register v
`v5`	Field to encode register v

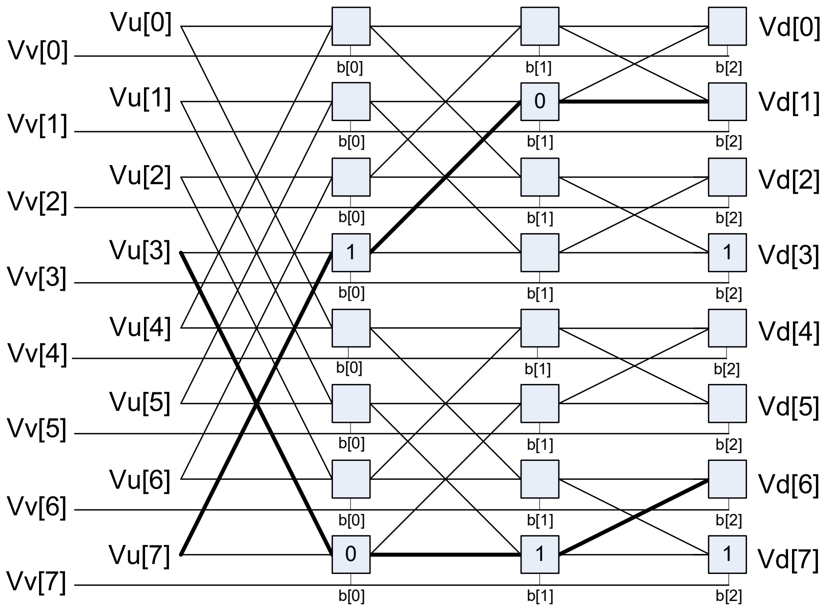
General permute network

Perform permutation and rearrangement 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 switch boxes 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 which have an increasing stride varying from 1 to 32 at the last stage. The diagram below 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 three), the bit at that position would look at the corresponding bit (bit three) in the control byte. This is shown in the switch box in the diagram.

There are two main forms of data rearrangement. One uses a simple reverse butterfly network shown as vrdelta, and a butterfly network vdelta shown below. 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.





An example is shown in the diagram above of such a valid pattern using an eight-element vrdelta network for clarity: 0,2,4,6,7,5,3,1.

However 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 4th 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

Valid controls =
{0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03}

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 has to 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.

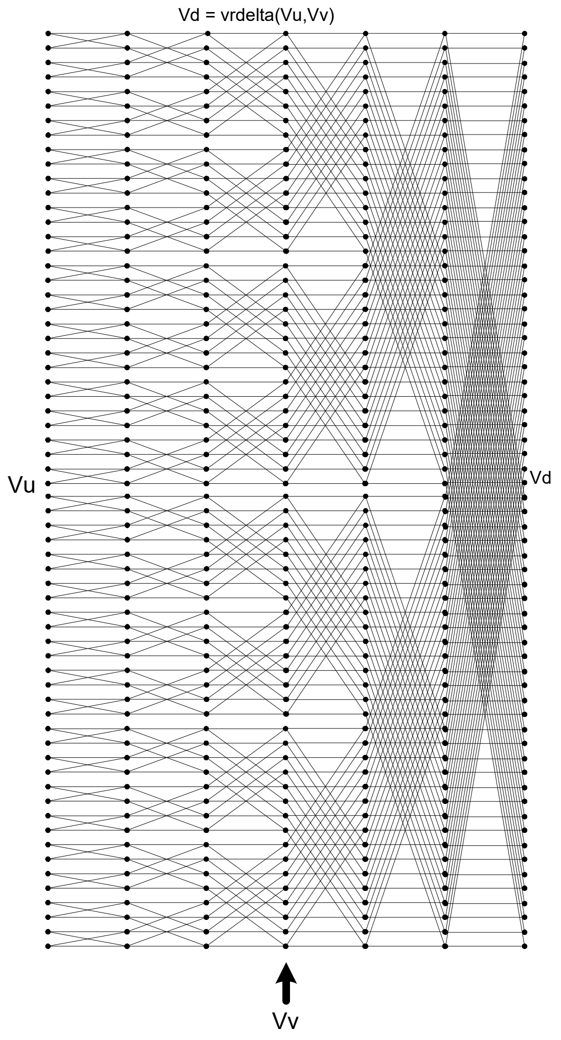
Element sizes larger than a byte can be implemented by grouping bytes together and moving them to a group in the output. An example of a general permute is the following random mix, where the 64 inputs are put in the following output positions: 33,42,40,61,28, 6,17,16,12,38,57,21,58,63,37,13,26,51,50,23,46, 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

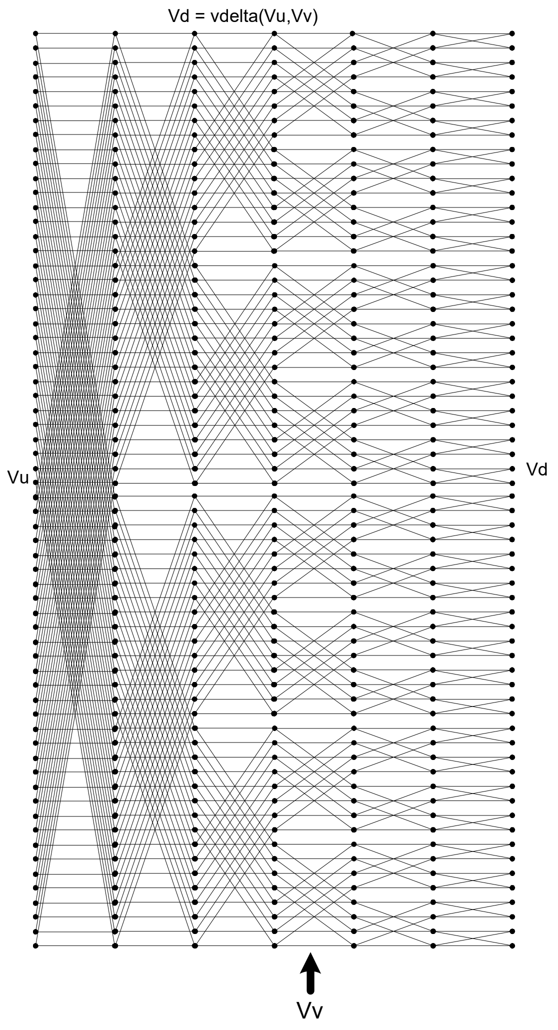
vrdelta controls ={0x00, 0x00, 0x21, 0x21, 0x20, 0x02, 0x00, 0x02, 0x20, 0x22, 0x00, 0x06, 0x23, 0x23, 0x02, 0x26, 0x06, 0x04, 0x2A, 0x0C, 0x2D, 0x2F, 0x20, 0x2E, 0x04, 0x00, 0x09, 0x29, 0x0C, 0x0A, 0x20, 0x0A, 0x05, 0x0F, 0x29, 0x2B, 0x2C, 0x0E, 0x11, 0x13, 0x31, 0x2F, 0x08, 0x0A, 0x2A, 0x3E, 0x02, 0x32, 0x0B, 0x07, 0x26, 0x0E, 0x2A, 0x2E, 0x36, 0x36, 0x1D, 0x07, 0x01, 0x2B, 0x0C, 0x1E, 0x21, 0x13}

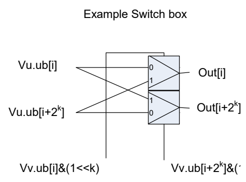
vdelta controls={ 0x1D, 0x01, 0x00, 0x00, 0x1D, 0x1B, 0x00, 0x1A, 0x1E, 0x02, 0x13, 0x03, 0x0C, 0x18, 0x10, 0x08, 0x1A, 0x06, 0x07, 0x03, 0x11, 0x1D, 0x0D, 0x11, 0x19, 0x03, 0x15, 0x03, 0x03, 0x19, 0x1F, 0x01, 0x1B, 0x1B, 0x06, 0x12, 0x18, 0x00, 0x1D, 0x09, 0x1A, 0x0E, 0x02, 0x02, 0x0B, 0x05, 0x0A, 0x18, 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. The Delta network X is a do-not-care output and replication is allowed.







Syntax	**Behavior**
`Vd=vdelta(Vu,Vv)`	see below

```
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];
    }
}
```

Syntax	Behavior
`Vd=vrdelta(Vu,Vv)`	see below

```
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

`Vd=vdelta(Vu,Vv)`	`HVX_Vector Q6_V_vdelta_VV(HVX_Vector Vu, HVX_Vector Vv)`
`Vd=vrdelta(Vu,Vv)`	`HVX_Vector Q6_V_vrdelta_VV(HVX_Vector Vu, HVX_Vector Vv)`

Encoding

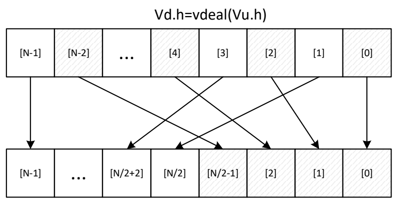
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse			u5								d5					
0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd=vdelta(Vu,Vv)
0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d	Vd=vrdelta(Vu,Vv)

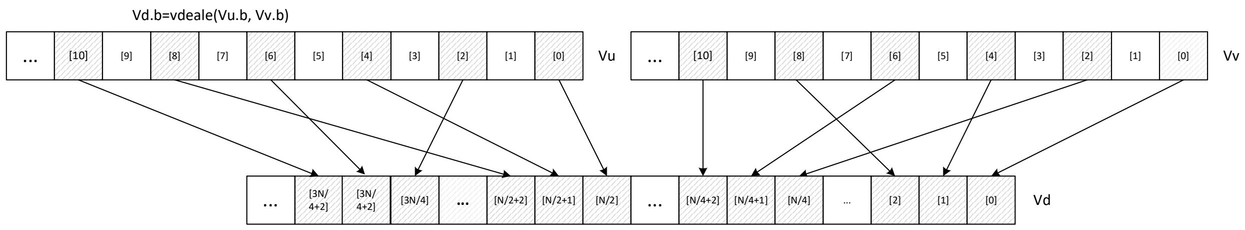
Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`u5`	Field to encode register u
`v5`	Field to encode register v

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.

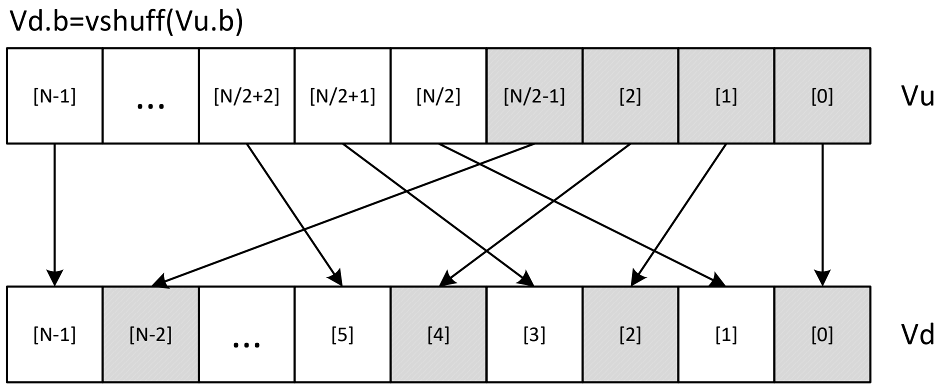
In the case of vdeale, the even elements of Vv are dealt into the lower half of the destination vector register Vd, and the even elements of Vu are dealt into the upper half of Vd. The deal operation takes even-even elements of Vv and places them in the lower quarter of Vd, while odd-even elements of 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.


Vd.h=vdeal(Vu.h)
[N-1] [N-2] ... [4] [3] [2] [1] [0] Vu
[N-1] ... [N/2+2] [N/2] [N/2-1] [2] [1] [0] Vd


Vd.b=vdeale(Vu.b, Vv.b)
... [10] [9] [8] [7] [6] [5] [4] [3] [2] [1] [0] Vu
... [10] [9] [8] [7] [6] [5] [4] [3] [2] [1] [0] Vv
... [3N/4+2] [3N/4+2] [3N/4] ... [N/2+2] [N/2+1] [N/2] ... [N/4+2] [N/4+1] [N/4] ... [2] [1] [0] 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, in a way similar to vshuffoe.


Vd.b=vshuff(Vu.b)
[N-1] ... [N/2+2] [N/2+1] [N/2] [N/2-1] [2] [1] [0] Vu
[N-1] [N-2] ... [5] [4] [3] [2] [1] [0] Vd

*N is the number of element operations allowed in the vector

Syntax	Behavior

`Vd.b=vdeal(Vu.b)`	`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];`
	`}`

`Vd.b=vdeale(Vu.b,Vv.b)`	`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];`
	`}`

`Vd.b=vshuff(Vu.b)`	`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];`
	`}`

`Vd.h=vdeal(Vu.h)`	`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];`
	`}`

`Vd.h=vshuff(Vu.h)`	`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];`
	`}`

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					

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	-	-	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

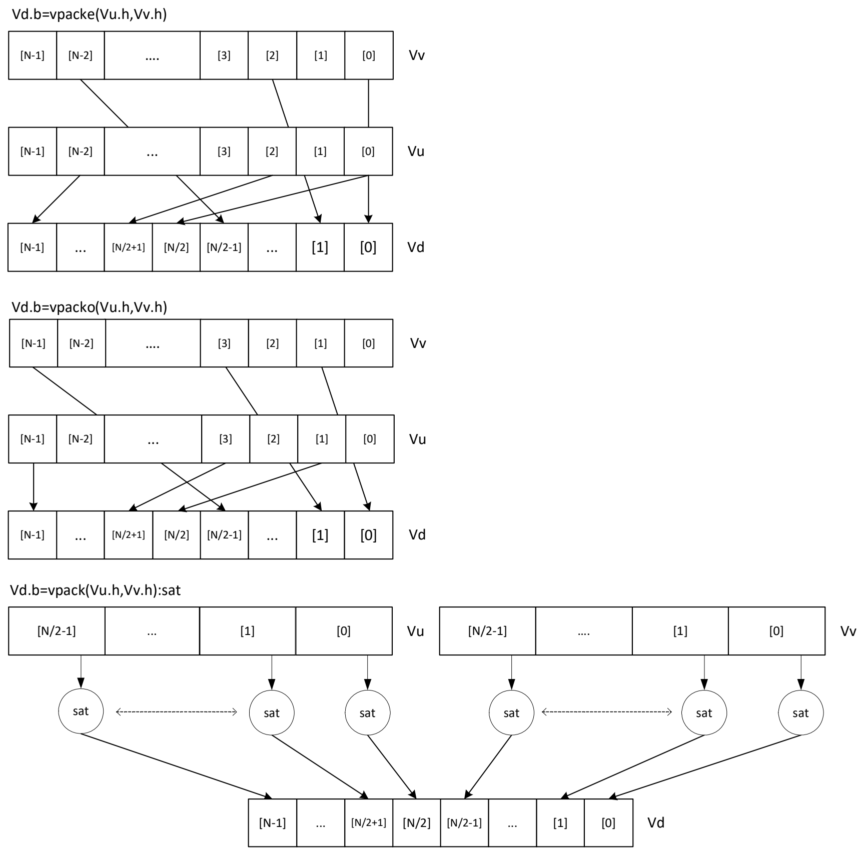
Pack

The vpack operation has three forms. All of them pack elements from the vector registers Vu and Vv into the destination vector register Vd.

vpacke writes even elements from Vv and Vu into the lower half and upper half of Vd respectively.

vpacko writes odd elements from Vv and Vu into the lower half and upper half of Vd respectively.

vpack takes all elements from Vv and Vu, saturates them to the next smallest element size, and writes them into Vd.



Syntax	Behavior

`Vd.b=vpack(Vu.h,Vv.h):sat`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.b[i] = sat`$_8$`(Vv.h[i]);`
	`    Vd.b[i+VBITS/16] = sat`$_8$`(Vu.h[i]);`
	`}`

`Vd.b=vpacke(Vu.h,Vv.h)`	`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];`
	`}`

`Vd.b=vpacko(Vu.h,Vv.h)`	`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];`
	`}`

`Vd.h=vpack(Vu.w,Vv.w):sat`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.h[i] = sat`$_{16}$`(Vv.w[i]);`
	`    Vd.h[i+VBITS/32] = sat`$_{16}$`(Vu.w[i]);`
	`}`

`Vd.h=vpacke(Vu.w,Vv.w)`	`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];`
	`}`

`Vd.h=vpacko(Vu.w,Vv.w)`	`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];`
	`}`

`Vd.ub=vpack(Vu.h,Vv.h):sat`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.ub[i] = usat`$_8$`(Vv.h[i]);`
	`    Vd.ub[i+VBITS/16] = usat`$_8$`(Vu.h[i]);`
	`}`

`Vd.uh=vpack(Vu.w,Vv.w):sat`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.uh[i] = usat`$_{16}$`(Vv.w[i]);`
	`    Vd.uh[i+VBITS/32] = usat`$_{16}$`(Vu.w[i]);`
	`}`

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

Notes

- This instruction uses the HVX permute resource.

Intrinsics

`Vd.b=vpack(Vu.h,Vv.h):sat`	`HVX_Vector Q6_Vb_vpack_VhVh_sat(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.b=vpacke(Vu.h,Vv.h)`	`HVX_Vector Q6_Vb_vpacke_VhVh(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.b=vpacko(Vu.h,Vv.h)`	`HVX_Vector Q6_Vb_vpacko_VhVh(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.h=vpack(Vu.w,Vv.w):sat`	`HVX_Vector Q6_Vh_vpack_VwVw_sat(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.h=vpacke(Vu.w,Vv.w)`	`HVX_Vector Q6_Vh_vpacke_VwVw(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.h=vpacko(Vu.w,Vv.w)`	`HVX_Vector Q6_Vh_vpacko_VwVw(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.ub=vpack(Vu.h,Vv.h):sat`	`HVX_Vector Q6_Vub_vpack_VhVh_sat(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.uh=vpack(Vu.w,Vv.w):sat`	`HVX_Vector Q6_Vuh_vpack_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	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-----0011111111111111111b

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 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 64B mode and 128 bytes in 128B mode. This explanation discusses both the 64B and 128B modes of operation. In both 64 and 128B 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

In the case of 64B 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 in order for the table entry to be written 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. If a 256B table is stored naturally in memory, it would look like the following example:

127,126,.....66, 65, 64, 63, 62,.........2, 1, 0
255,254,....194,193,192,191,190,.......130,129,128

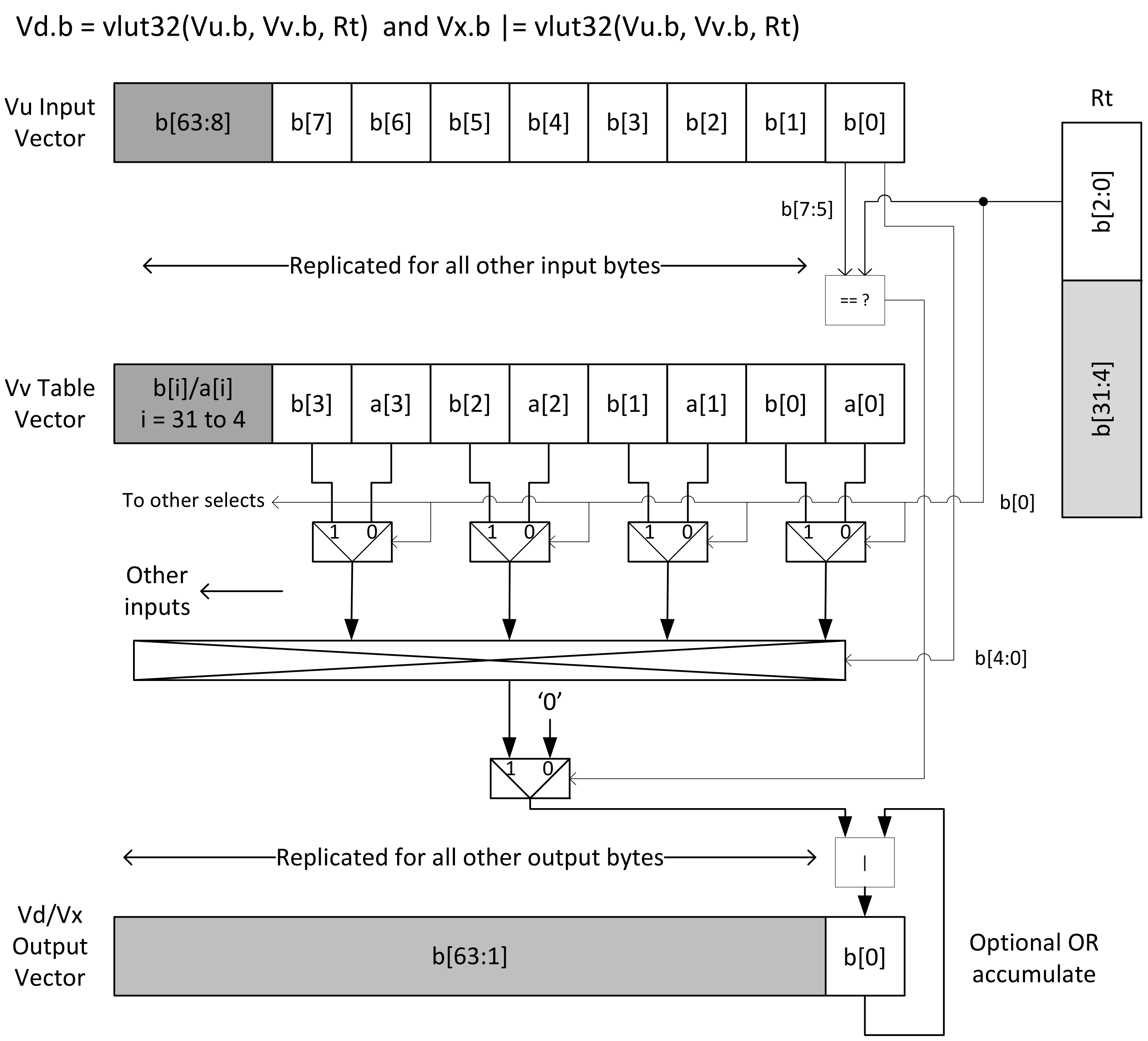
To prepare it for use with the vlut instruction in 64B mode, it must be shuffled in blocks of 32 bytes.

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

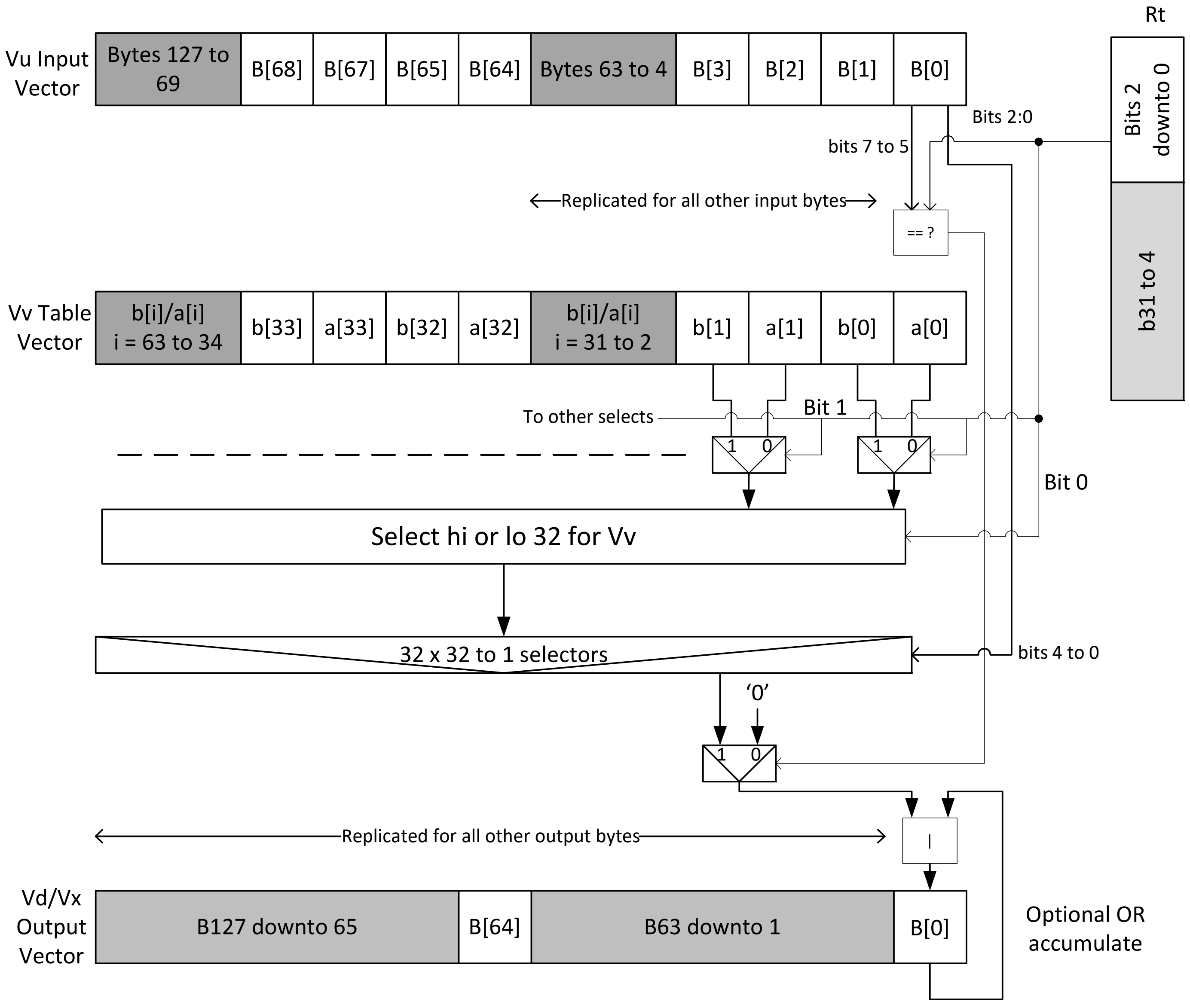
For 128B mode, the data must be shuffled in blocks of 64 bytes.

127, 63,126, 62,........68, 4, 67, 3, 66, 2, 65, 1, 64, 0 Rt=0,1,2,3 same ordering for bytes 128-255 Rt=4,5,6,7

If data is stored in this way, accessing this with 64 or 128B mode gives the same results. In the case of 128B mode, bit one 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.


Vd.b = vlut32(Vu.b, Vv.b, Rt) and Vx.b |= vlut32(Vu.b, Vv.b, Rt)
Vu Input Vector
b[63:8]
b[7]
b[6]
b[5]
b[4]
b[3]
b[2]
b[1]
b[0]
Rt
b[2:0]
b[31:4]
b[7:5]
Replicated for all other input bytes
== ?
Vv Table Vector
b[i]/a[i] i = 31 to 4
b[3]
a[3]
b[2]
a[2]
b[1]
a[1]
b[0]
a[0]
To other selects
b[0]
1 0
Other inputs
b[4:0]
'0'
1 0
|
Replicated for all other output bytes
Vd/Vx Output Vector
b[63:1]
b[0]
Optional OR accumulate

Vd.b = vlut32(Vu.b, Vv.b, Rt) and Vx.b |= vlut32(Vu.b, Vv.b, Rt)



128Byte mode

16-bit elements

For tables with 16-bit elements, the basic unit is a 16-entry lookup table in 64B mode and 128B 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. In the case of 64B mode, the lower four bits of Rt must match the upper four bits of the input bytes for the table entry to be written 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. In the 128B case, only the upper 4 bits of input bytes must also match the lower four of Rt. Bit one 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. If a 256H table is stored naturally in memory, it would look like the following example:

63, 62,.........2, 1, 0 127,126,.......66, 65, 64 191,190,......130,129,128 255,254,......194,193,192

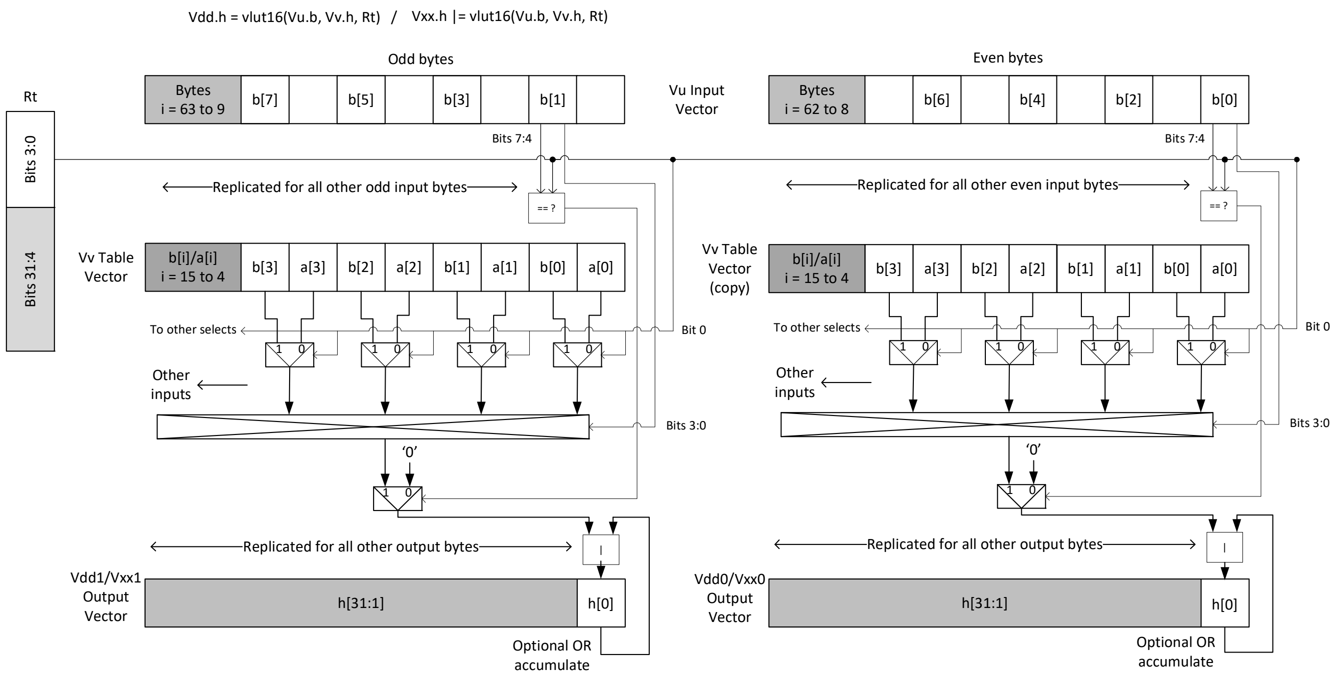
To prepare it for use with the vlut instruction in 64B mode, it must be shuffled in blocks of 16 hwords, the LSB of Rt is used to choose the even or odd 16 entry hword tables in Vv.

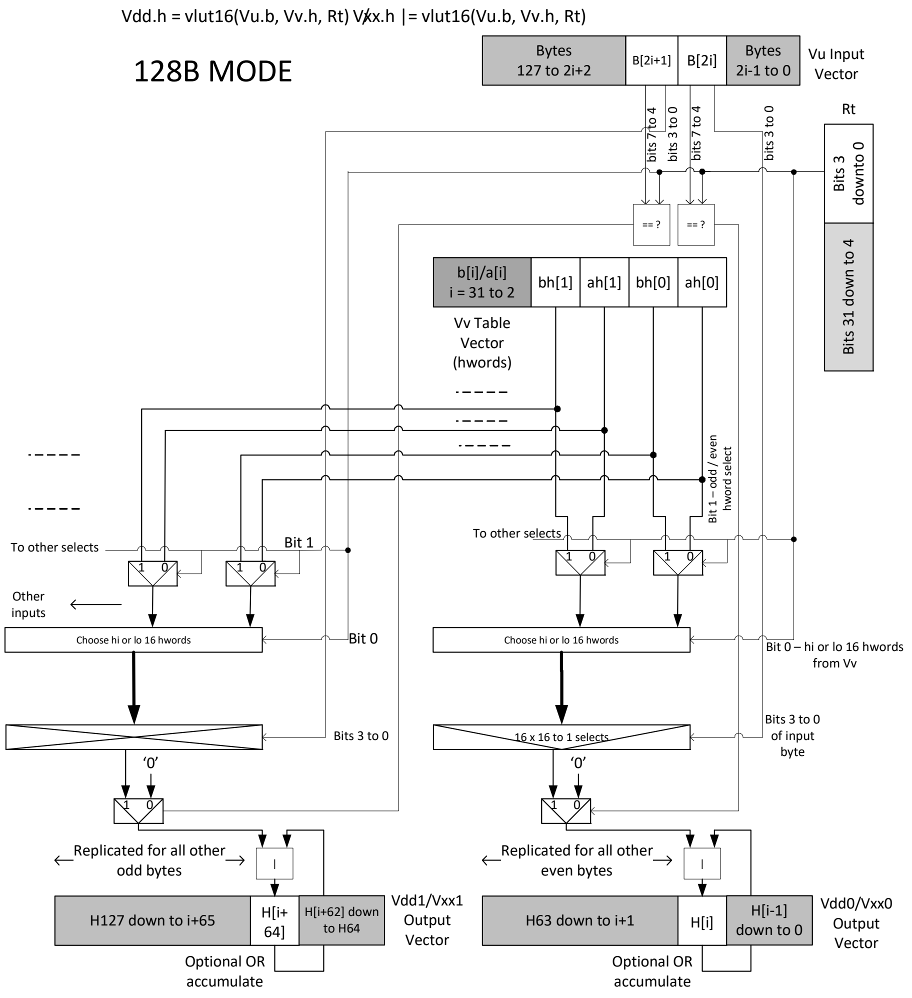
31, 15, 30, 14,......20, 4, 19, 3, 18, 2, 17, 1, 16, 0 Rt=0, Rt=1 63, 47, 62, 46,..... 52, 36, 51, 35, 50, 34, 49, 33, 48, 32 Rt=2, Rt=3 same ordering for bytes 64-255 Rt=4, 5, 6, 7, 8, 9, 10,11,12,13,14,15

In the case of the 128B mode, the data must be shuffled in blocks of 32 hwords. Bit one 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, 0 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

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 go into the odd destination, 64B and 128B 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	Behavior

`Vd.b=vlut32(Vu.b,Vv.b,#u3)`	`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;`
	`}`

`Vd.b=vlut32(Vu.b,Vv.b,Rt)`	`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;`
	`}`

`Vd.b=vlut32(Vu.b,Vv.b,Rt):nomatch`	`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):nomatch
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
v2	Field to encode register v
v3	Field to encode register v
v5	Field to encode register v

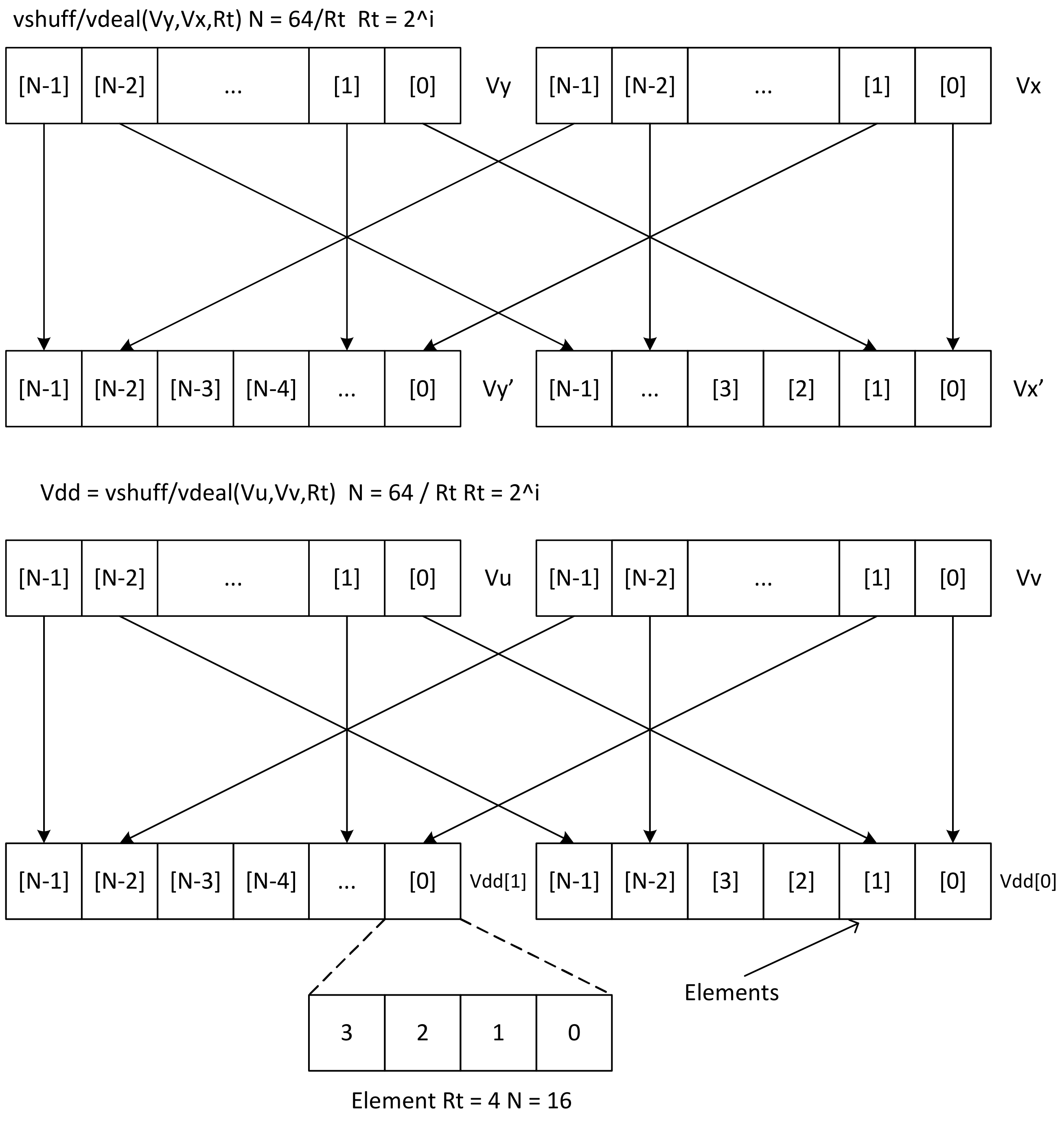
5.11 HVX/PERMUTE-SHIFT-RESOURCE

The HVX/PERMUTE-SHIFT-RESOURCE instruction subclass includes instructions that use both the HVX permute and shift resources.

Vector shuffle and deal cross-lane

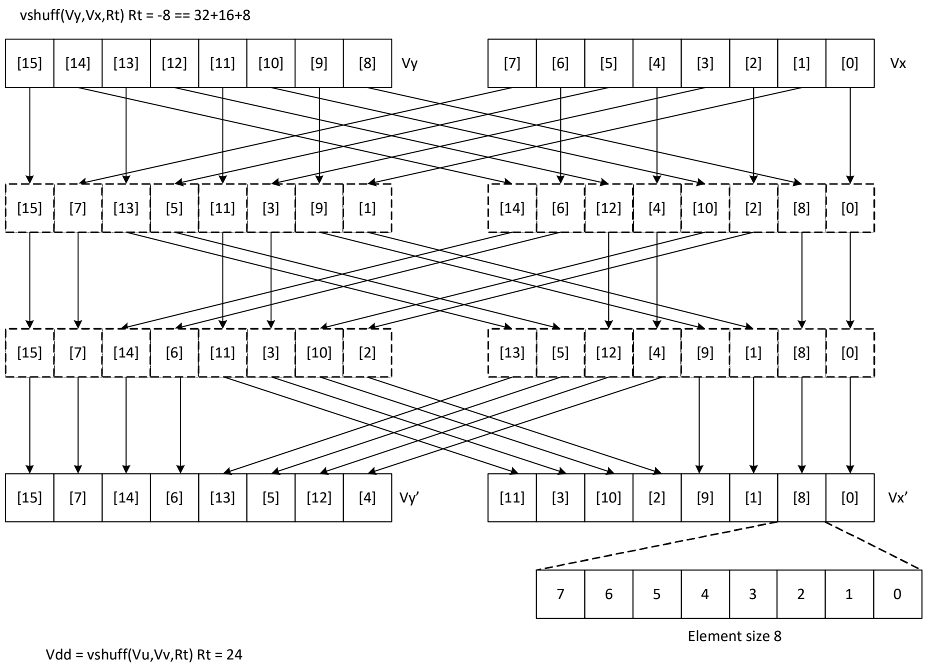
Vshuff (formerly vtrans2x2) and vdeal perform a multiple-level transpose operation between groups of elements in two vectors. The element size is specified by the scalar register Rt. Rt=1 indicates an element size of one byte, Rt=2 indicates halfwords, Rt=4 words, Rt=8 8 bytes, Rt=16 16 bytes, and Rt=32 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 then transposed, and the resulting elements are then presented in the two destination registers. Note that a value of Rt = 0 leaves the input unchanged.

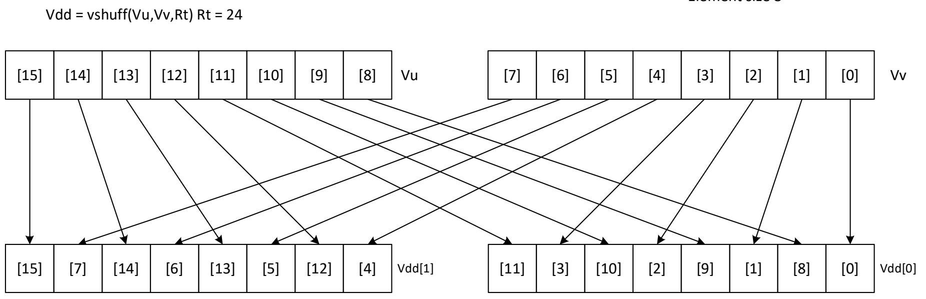
The following image shows examples for Rt = 1,2,4,8,16,32. In these cases vdeal and vshuff perform the same operation. The diagram is valid for vshuff and vdeal.

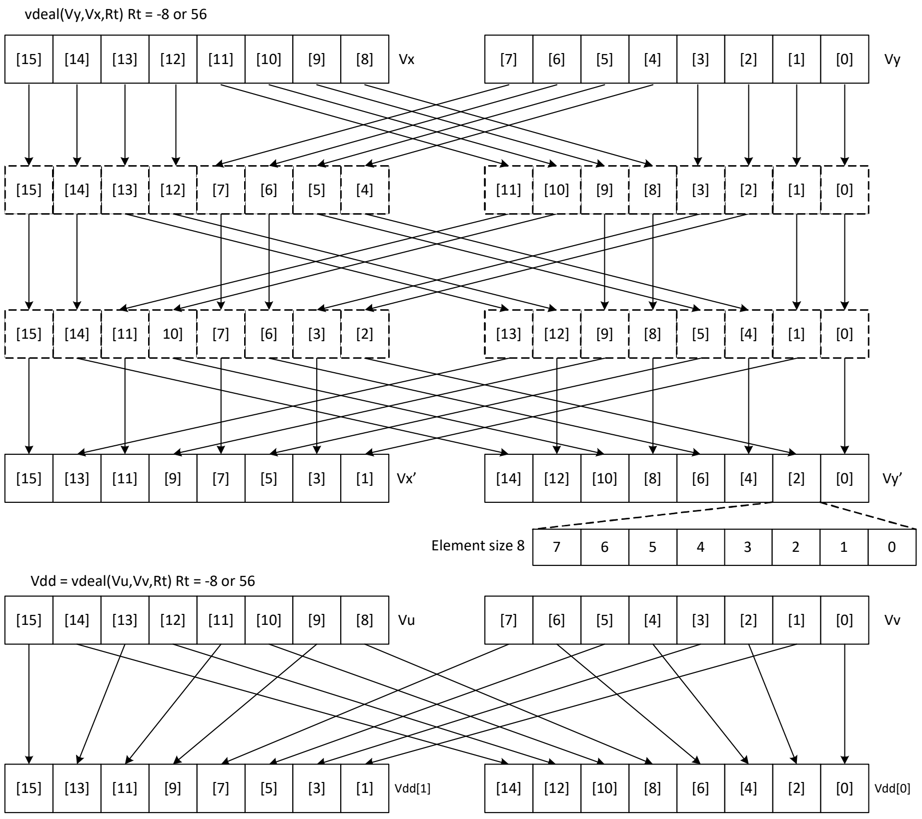


When a value of Rt other than 1,2,4,8,16,32 is used, the effect is a compound hierarchical transpose. For example, if the value 23 is used, 23 = 1+2+4+16. This indicates that the transformation is the same as performing the vshuff instruction with Rt=1, then Rt=2 on that result, then Rt = 4 on its result, then Rt = 16 on its result. Note that the order is in increasing element size. In the case of vdeal the order is reversed, starting with the largest element size first, then working down to the smallest.

When the Rt value is the negated power of 2: -1,-2,-4,-8,-16,-32, it performs a a perfect shuffle for vshuff, or a deal for vdeal of the smallest element size. For example, if Rt = -24 this is a multiple of eight, so eight is the smallest element size. With a -ve value of Rt, all of the upper bits of the value Rt are set. For example, with Rt=-8 this is the same as 32+16+8. The following diagram shows the effect of this transform for both vshuff and vdeal.


vshuff(Vy,Vx,Rt) Rt = -8 == 32+16+8
[15] [14] [13] [12] [11] [10] [9] [8] Vy
[7] [6] [5] [4] [3] [2] [1] [0] Vx
[15] [7] [13] [5] [11] [3] [9] [1]
[14] [6] [12] [4] [10] [2] [8] [0]
[15] [7] [14] [6] [11] [3] [10] [2]
[13] [5] [12] [4] [9] [1] [8] [0]
[15] [7] [14] [6] [13] [5] [12] [4] Vy’
[11] [3] [10] [2] [9] [1] [8] [0] Vx’
7 6 5 4 3 2 1 0
Element size 8


Vdd = vshuff(Vu,Vv,Rt) Rt = 24
[15] [14] [13] [12] [11] [10] [9] [8] Vu
[7] [6] [5] [4] [3] [2] [1] [0] Vv
[15] [7] [14] [6] [13] [5] [12] [4] Vdd[1]
[11] [3] [10] [2] [9] [1] [8] [0] Vdd[0]


vdeal(Vy,Vx,Rt) Rt = -8 or 56
[15] [14] [13] [12] [11] [10] [9] [8] Vx
[7] [6] [5] [4] [3] [2] [1] [0] Vy
[15] [14] [13] [12] [7] [6] [5] [4]
[11] [10] [9] [8] [3] [2] [1] [0]
[15] [14] [11] 10] [7] [6] [3] [2]
[13] [12] [9] [8] [5] [4] [1] [0]
[15] [13] [11] [9] [7] [5] [3] [1] Vx'
[14] [12] [10] [8] [6] [4] [2] [0] Vy'
Element size 8
7 6 5 4 3 2 1 0
Vdd = vdeal(Vu,Vv,Rt) Rt = -8 or 56
[15] [14] [13] [12] [11] [10] [9] [8] Vu
[7] [6] [5] [4] [3] [2] [1] [0] Vv
[15] [13] [11] [9] [7] [5] [3] [1] Vdd[1]
[14] [12] [10] [8] [6] [4] [2] [0] Vdd[0]

If in addition to this family of transformations a block size is defined B, and the element size is defined as E, if Rt = B - E, the resulting transformation is a set of B contiguous blocks, each containing perfectly shuffled or dealt elements of element size E. Each block B contains 128/B elements in the 64B vector case. This represents the majority of data transformations commonly used. When B is set to 0, the result is a shuffle or deal of elements across the whole vector register pair.

Syntax	Behavior

`Vdd=vdeal(Vu,Vv,Rt)`	`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]);`
	`            }`
	`        }`
	`    }`
	`}`

`Vdd=vshuff(Vu,Vv,Rt)`	`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]);`
	`            }`
	`        }`
	`    }`
	`}`

`vdeal(Vy,Vx,Rt)`	`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]);`
	`            }`
	`        }`
	`    }`
	`}`

`vshuff(Vy,Vx,Rt)`	`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]);`
	`            }`
	`        }`
	`    }`
	`}`

`vtrans2x2(Vy,Vx,Rt)`	`Assembler mapped to: "vshuff(Vy,Vx,Rt)"`

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

`Vdd=vdeal(Vu,Vv,Rt)`	`HVX_VectorPair Q6_W_vdeal_VVR(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)`
`Vdd=vshuff(Vu,Vv,Rt)`	`HVX_VectorPair Q6_W_vshuff_VVR(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											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**	**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**	**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
`v2`	Field to encode register v
`v3`	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 64B mode and 128 bytes in 128B mode. This explanation discusses both the 64B and 128B modes of operation. In both 64 and 128byte 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

In the case of 64Byte 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 be written 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. If a 256B table is stored naturally in memory, it would look like the following example:

127,126,.....66, 65, 64, 63, 62,.........2, 1, 0
255,254,....194,193,192,191,190,.......130,129,128

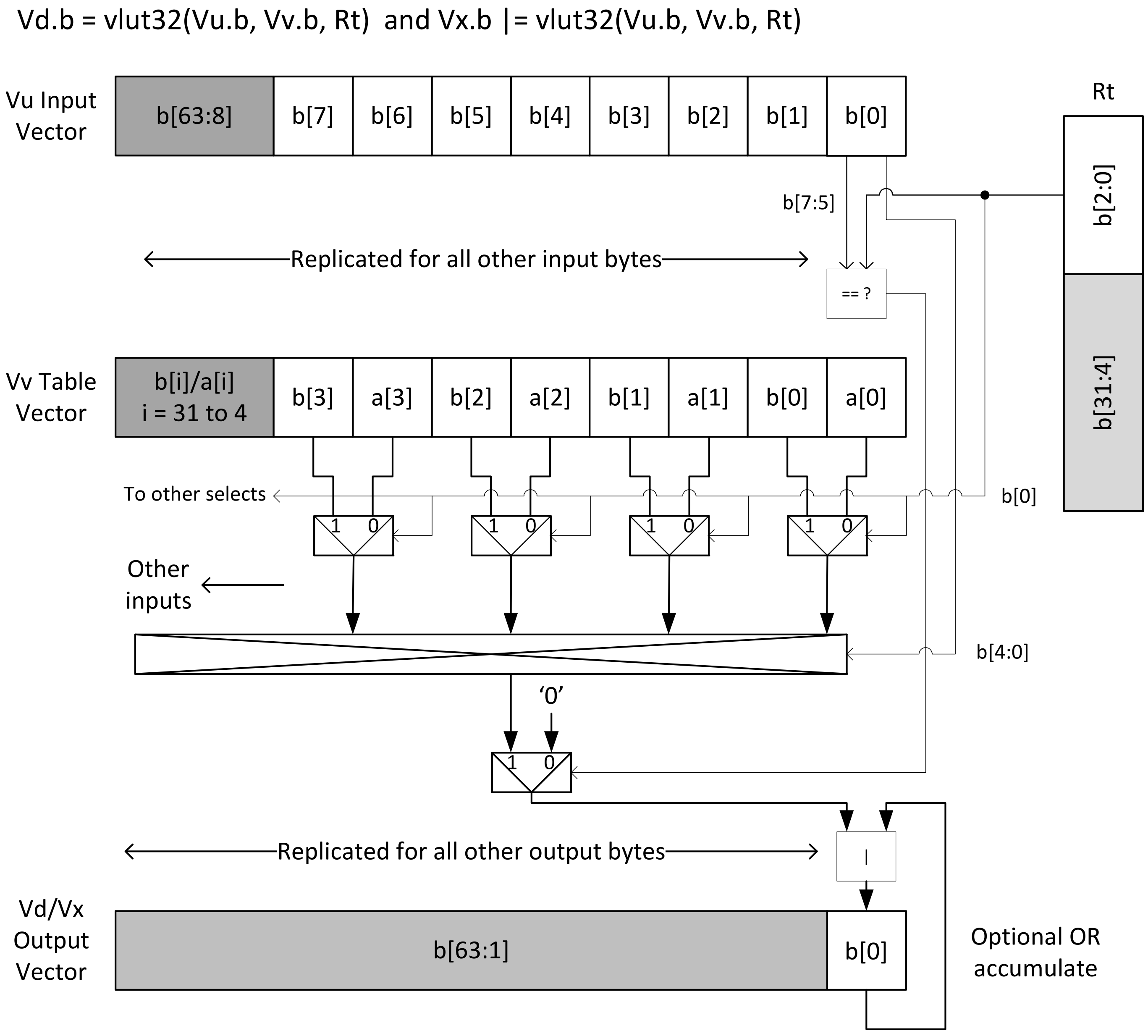
For use with the vlut instruction in 64B mode, it must be shuffled in blocks of 32 bytes

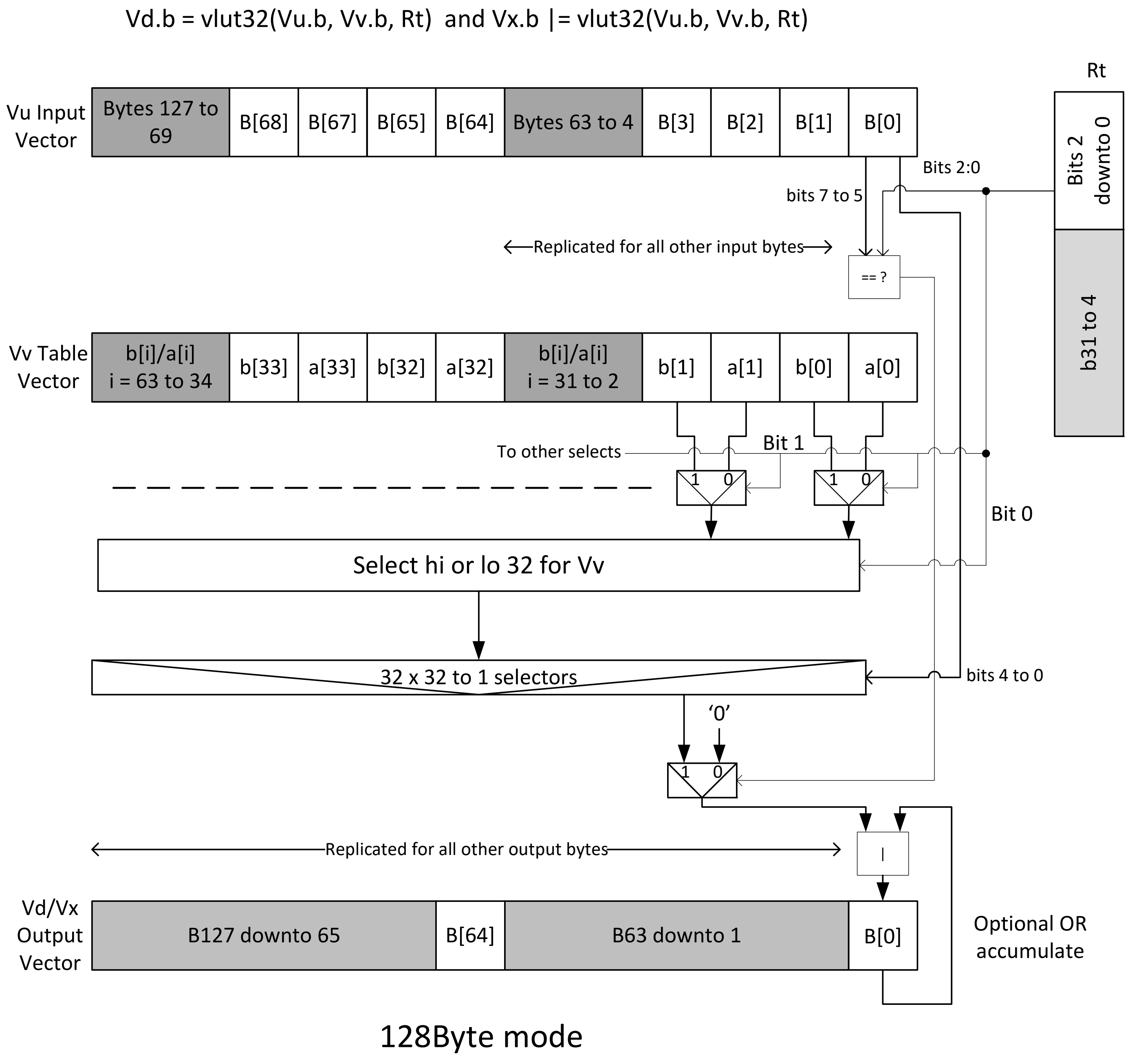
63, 31, 62, 30,......36, 4, 35, 3, 34, 2, 33, 1, 32, 0 Rt=0, Rt=1 127, 95,126, 94,.....100, 68, 99, 67, 98, 66, 97, 65, 96, 64 Rt=2, Rt=3 same ordering for bytes 128-255 Rt=4, 5, 6, 7

In the case of the 128B 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 128-255 Rt=4,5,6,7

If data is stored in this way, accessing this with 64 or 128B mode gives the same results. In the case of 128B mode, bit one 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.




Vd.b = vlut32(Vu.b, Vv.b, Rt) and Vx.b |= vlut32(Vu.b, Vv.b, Rt)
Rt
Vu Input Vector
Bytes 127 to 69
B[68]
B[67]
B[65]
B[64]
Bytes 63 to 4
B[3]
B[2]
B[1]
B[0]
Bits 2 downto 0
b31 to 4
Bits 2:0
bits 7 to 5
Replicated for all other input bytes
== ?
Vv Table Vector
b[i]/a[i] i = 63 to 34
b[33]
a[33]
b[32]
a[32]
b[i]/a[i] i = 31 to 2
b[1]
a[1]
b[0]
a[0]
To other selects
Bit 1
1
0
Bit 0
Select hi or lo 32 for Vv
32 x 32 to 1 selectors
bits 4 to 0
‘0’
Replicated for all other output bytes
|
Vd/Vx Output Vector
B127 downto 65
B[64]
B63 downto 1
B[0]
Optional OR accumulate
128Byte mode

16-bit elements

For tables with 16-bit elements, the basic unit is a 16-entry lookup table in 64B mode and 128B 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 64B mode, the lower four bits of Rt must match the upper four bits of the input bytes for the table entry to be written 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. In the 128B case, only the upper four bits of input bytes must also match the lower four of Rt. Bit one 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 8 different 32 hword sections. If a 256H table is stored naturally in memory, it would look like the following:

63, 62,.........2, 1, 0 127,126,.......66, 65, 64 191,190,......130,129,128 255,254,......194,193,192

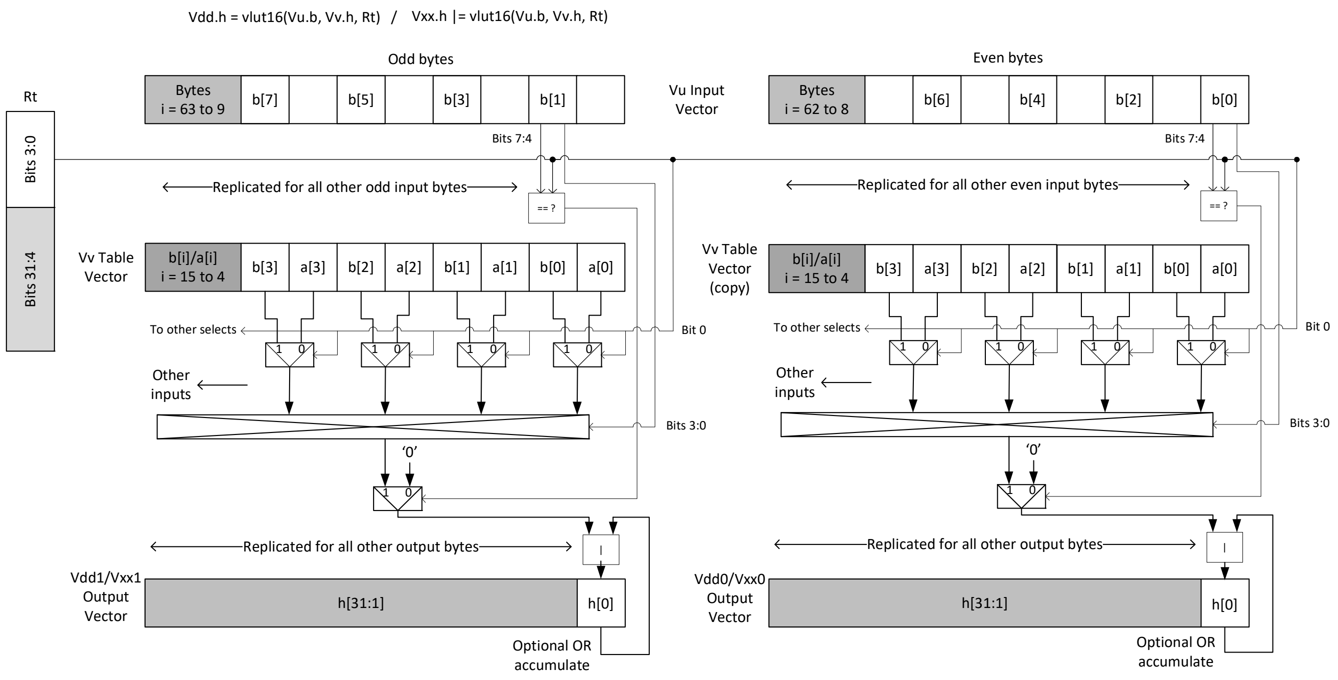
To prepare it for use with the vlut instruction in 64B mode, it must be shuffled in blocks of 16 hwords, the LSB of Rt is used to choose the even or odd 16 entry hword tables in Vv.

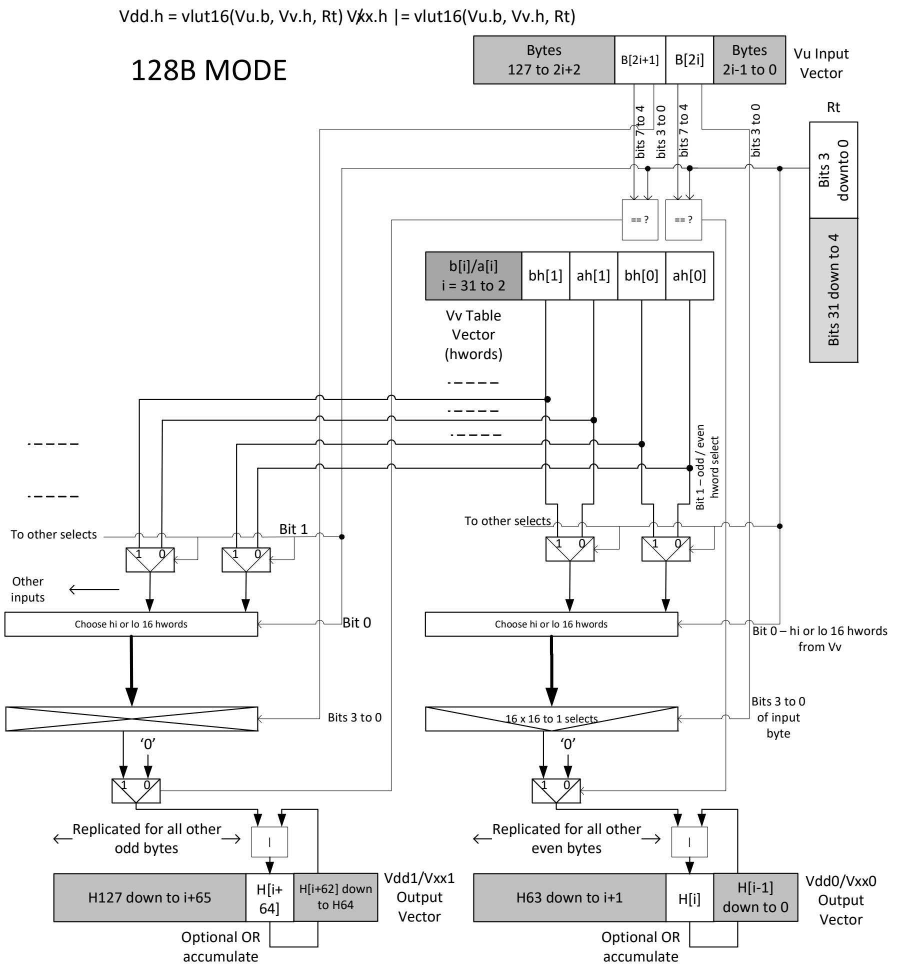
31, 15, 30, 14,......20, 4, 19, 3, 18, 2, 17, 1, 16, 0 Rt=0, Rt=1 63, 47, 62, 46,..... 52, 36, 51, 35, 50, 34, 49, 33, 48, 32 Rt=2, Rt=3 same ordering for bytes 64-255 Rt=4, 5, 6, 7, 8, 9, 10,11,12,13,14,15

For 128B 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, 0 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

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, 64B and 128B 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	Behavior

`Vdd.h=vlut16(Vu.b,Vv.h,#u3)`	`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

Vdd.h=vlut16(Vu.b,Vv.h,Rt)	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;
	}

Vdd.h=vlut16(Vu.b,Vv.h,Rt):nomatch	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];
	}

Vx.b|=vlut32(Vu.b,Vv.b,#u3)	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;
	}

Vx.b|=vlut32(Vu.b,Vv.b,Rt)	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;
	}

Vxx.h|=vlut16(Vu.b,Vv.h,#u3)	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;
	}

Vxx.h|=vlut16(Vu.b,Vv.h,Rt)	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;
	}

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

`Vdd.h=vlut16(Vu.b,Vv.h,#u3)`	`HVX_VectorPair Q6_Wh_vlut16_VbVhI(HVX_Vector Vu, HVX_Vector Vv, Word32 Iu3)`
`Vdd.h=vlut16(Vu.b,Vv.h,Rt)`	`HVX_VectorPair Q6_Wh_vlut16_VbVhR(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)`
`Vdd.h=vlut16(Vu.b,Vv.h,Rt):nomatch`	`HVX_VectorPair Q6_Wh_vlut16_VbVhR_nomatch(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)`
`Vx.b|=vlut32(Vu.b,Vv.b,#u3)`	`HVX_Vector Q6_Vb_vlut32or_VbVbVbI(HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv, Word32 Iu3)`
`Vx.b|=vlut32(Vu.b,Vv.b,Rt)`	`HVX_Vector Q6_Vb_vlut32or_VbVbVbR(HVX_Vector Vx, HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)`
`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

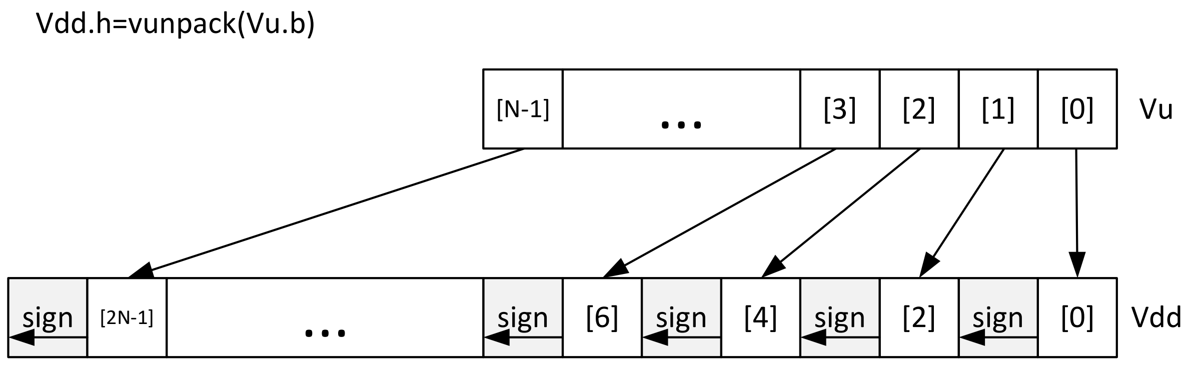
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS													t3			Parse			u5								d5					
0	0	0	1	1	0	0	0	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d	Vdd.h=vlut16(Vu.b,Vv.h,Rt):nomatch
ICLASS													t3			Parse			u5								x5					
0	0	0	1	1	0	1	1	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)

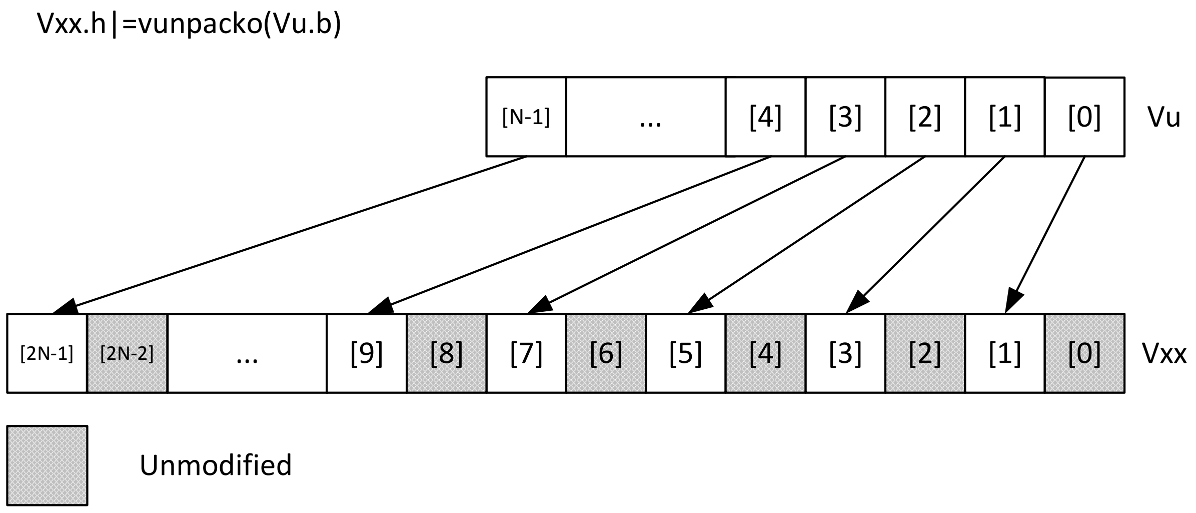
Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`t3`	Field to encode register t
`u5`	Field to encode register u
`v2`	Field to encode register v
`v3`	Field to encode register v
`v5`	Field to encode register v
`x5`	Field to encode register 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	Behavior
`Vdd.h=vunpack(Vu.b)`	`for (i = 0; i < VELEM(8); i++) { Vdd.h[i] = Vu.b[i]; }`
`Vdd.uh=vunpack(Vu.ub)`	`for (i = 0; i < VELEM(8); i++) { Vdd.uh[i] = Vu.ub[i]; }`
`Vdd.uw=vunpack(Vu.uh)`	`for (i = 0; i < VELEM(16); i++) { Vdd.uw[i] = Vu.uh[i]; }`
`Vdd.w=vunpack(Vu.h)`	`for (i = 0; i < VELEM(16); i++) { Vdd.w[i] = Vu.h[i]; }`
`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								x5					
0	0	0	1	1	1	1	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	**P**	**P**	1	**u**	**u**	**u**	**u**	**u**	0	0	1	**x**	**x**	**x**	**x**	**x**	Vxx.w|=vunpacko(Vu.h)

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)

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

5.12 HVX/SCATTER-DOUBLE-RESOURCE

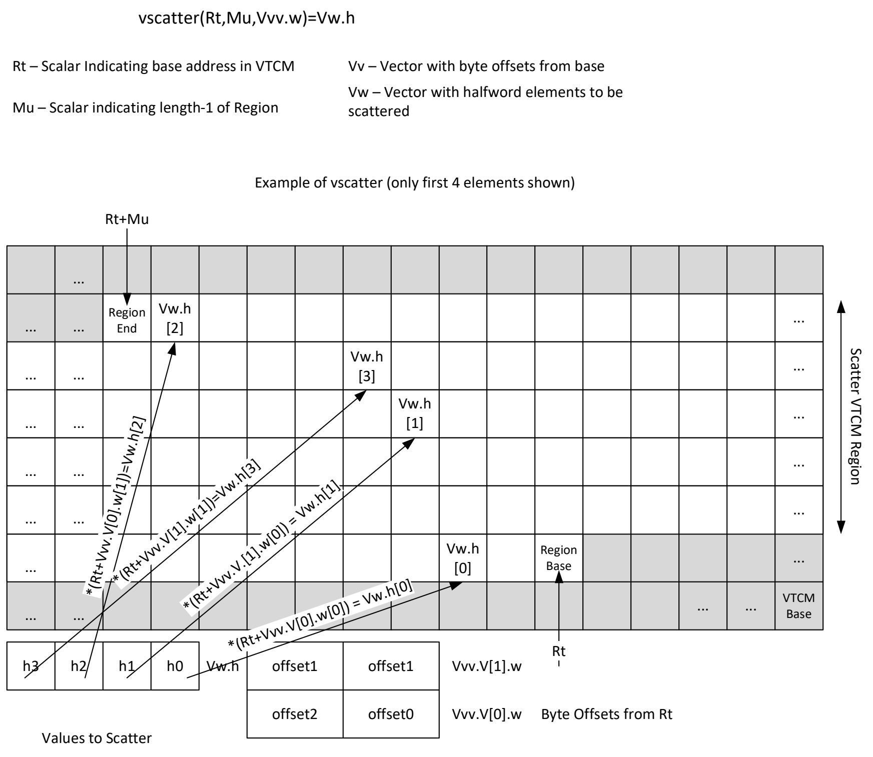
The HVX/SCATTER-DOUBLE-RESOURCE instruction subclass includes instructions that perform scatter operations to the vector TCM.

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 written 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, it is dropped. Offsets must be positive, otherwise they are dropped.


vscatter(Rt,Mu,Vvv.w)=Vw.h
Rt – Scalar Indicating base address in VTCM
Mu – Scalar indicating length-1 of Region
Vv – Vector with byte offsets from base
Vw – Vector with halfword elements to be scattered
Example of vscatter (only first 4 elements shown)
Rt+Mu
Region End
Vw.h [2]
Vw.h [3]
Vw.h [1]
Vw.h [0]
Region Base
VTCM Base
Scatter VTCM Region
*(Rt+Vvv.V[0].w[1])=Vw.h[2]
*(Rt+Vvv.V[1].w[1])=Vw.h[3]
*(Rt+Vvv.V[1].w[0]) = Vw.h[1]
*(Rt+Vvv.V[0].w[0]) = Vw.h[0]
h3 h2 h1 h0 Vw.h
Values to Scatter
offset1 offset1 Vvv.V[1].w
offset2 offset0 Vvv.V[0].w
Rt
Byte Offsets from Rt

Syntax	Behavior

if (Qs4)
vscatter(Rt,Mu,Vvv.w).h=Vw32
	MuV | (element_size-1);
	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];
	    }
	}

if (Qs4) vscatter(Rt,Mu,Vvv.w)=Vw32.h	Assembler mapped to: "if (Qs4) vscatter(Rt,Mu2,Vvv.w).h=Vw32"
vscatter(Rt,Mu,Vvv.w)+=Vw32.h	Assembler mapped to: "vscatter(Rt,Mu2,Vvv.w).h+=Vw32"

vscatter(Rt,Mu,Vvv.w).h+=Vw32	MuV | (element_size-1);
	Rt & ~(element_size-1);
	for (i = 0; i < VELEM(32); i++) {
	    for(j = 0; j < 2; j++) {
	        EA = Rt + Vvv.v[j].uw[i] & ~(ALIGNMENT-1);
	        if (Rt <= EA <= Rt + MuV) *EA += VwV.w[i].uh[j];
	    }
	}

vscatter(Rt,Mu,Vvv.w).h=Vw32	MuV | (element_size-1);
	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];
	    }
	}

vscatter(Rt,Mu,Vvv.w)=Vw32.h	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

if (Qs4) vscatter(Rt,Mu,Vvv.w).h=Vw32	void Q6_vscatter_QRMWwV(HVX_VectorPred Qs, Word32 Rt, Word32 Mu, HVX_VectorPair Vvv, HVX_Vector Vw)
vscatter(Rt,Mu,Vvv.w).h+=Vw32	void Q6_vscatteracc_RMWwV(Word32 Rt, Word32 Mu, HVX_VectorPair Vvv, HVX_Vector Vw)
vscatter(Rt,Mu,Vvv.w).h=Vw32	void Q6_vscatter_RMWwV(Word32 Rt, Word32 Mu, HVX_VectorPair Vvv, 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	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

5.13 HVX/SCATTER

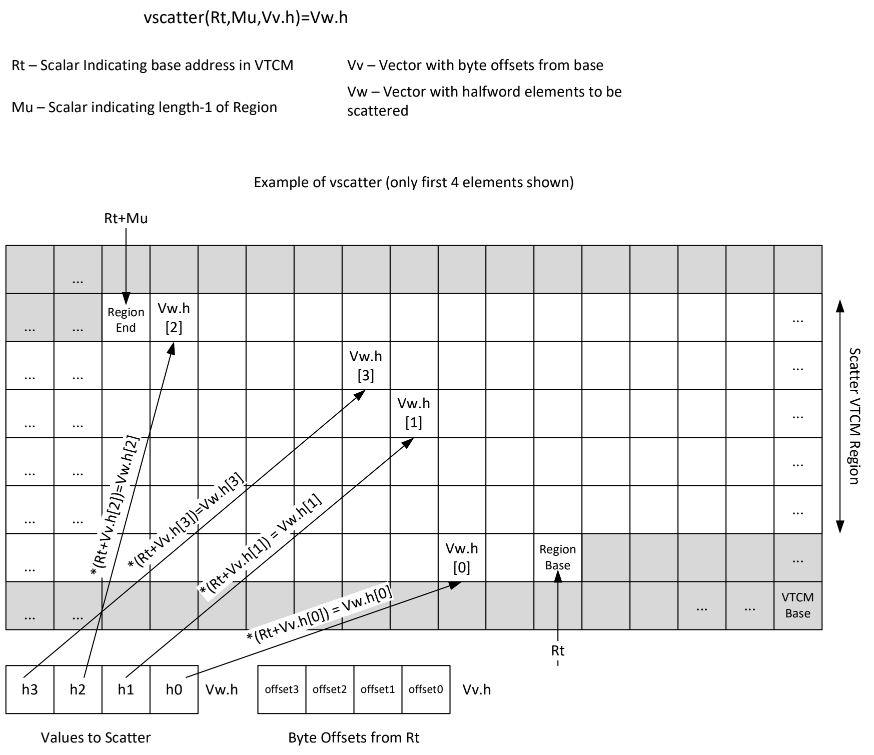
The HVX/SCATTER instruction subclass includes instructions that perform scatter operations to the vector TCM.

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 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 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 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.

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 scatter region's end, it is dropped. Offsets must be positive otherwise they are dropped.


vscatter(Rt,Mu,Vv.h)=Vw.h
Rt – Scalar Indicating base address in VTCM
Mu – Scalar indicating length-1 of Region
Vv – Vector with byte offsets from base
Vw – Vector with halfword elements to be scattered
Example of vscatter (only first 4 elements shown)
Rt+Mu
Region End
Vw.h [2]
Vw.h [3]
Vw.h [1]
Vw.h [0]
Region Base
VTCM Base
Scatter VTCM Region
*(Rt+Vv.h[2])=Vw.h[2]
*(Rt+Vv.h[3])=Vw.h[3]
*(Rt+Vv.h[1]) = Vw.h[1]
*(Rt+Vv.h[0]) = Vw.h[0]
Rt
h3 h2 h1 h0 Vw.h
offset3 offset2 offset1 offset0 Vv.h
Values to Scatter
Byte Offsets from Rt

Syntax	Behavior

if (Qs4)
vscatter(Rt,Mu,Vv.h).h=Vw32
	MuV | (element_size-1);
	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];
	}

if (Qs4) vscatter(Rt,Mu,Vv.h)=Vw32.h	Assembler mapped to: "if (Qs4) vscatter(Rt,Mu2,Vv.h).h=Vw32"

if (Qs4)
vscatter(Rt,Mu,Vv.w).w=Vw32
	MuV | (element_size-1);
	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];
	}

if (Qs4) vscatter(Rt,Mu,Vv.w)=Vw32.w	Assembler mapped to: "if (Qs4) vscatter(Rt,Mu2,Vv.w).w=Vw32"
vscatter(Rt,Mu,Vv.h)+=Vw32.h	Assembler mapped to: "vscatter(Rt,Mu2,Vv.h).h+=Vw32"

vscatter(Rt,Mu,Vv.h).h+=Vw32	MuV | (element_size-1);
	Rt & ~(element_size-1);
	for (i = 0; i < VELEM(16); i++) {
	    EA = (Rt+Vv.uh[i] & ~(ALIGNMENT-1));
	    if (Rt <= EA <= Rt + MuV) *EA += VwV.uw[i];
	}

vscatter(Rt,Mu,Vv.h).h=Vw32	MuV | (element_size-1);
	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];
	}

vscatter(Rt,Mu,Vv.h)=Vw32.h	Assembler mapped to: "vscatter(Rt,Mu2,Vv.h).h=Vw32"
vscatter(Rt,Mu,Vv.w)+=Vw32.w	Assembler mapped to: "vscatter(Rt,Mu2,Vv.w).w+=Vw32"

vscatter(Rt,Mu,Vv.w).w+=Vw32	MuV | (element_size-1);
	Rt & ~(element_size-1);
	for (i = 0; i < VELEM(32); i++) {
	    EA = (Rt+Vv.uw[i] & ~(ALIGNMENT-1));
	    if (Rt <= EA <= Rt + MuV) *EA += VwV.uw[i];
	}

vscatter(Rt,Mu,Vv.w).w=Vw32	MuV | (element_size-1);
	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];
	}

vscatter(Rt,Mu,Vv.w)=Vw32.w	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, Word32 Rt, Word32 Mu, HVX_Vector Vv, HVX_Vector Vw)`
`if (Qs4) vscatter(Rt,Mu,Vv.w).w=Vw32`	`void Q6_vscatter_QRMVwV(HVX_VectorPred Qs, Word32 Rt, Word32 Mu, HVX_Vector Vv, HVX_Vector Vw)`
`vscatter(Rt,Mu,Vv.h).h+=Vw32`	`void Q6_vscatteracc_RMVhV(Word32 Rt, Word32 Mu, HVX_Vector Vv, HVX_Vector Vw)`
`vscatter(Rt,Mu,Vv.h).h=Vw32`	`void Q6_vscatter_RMVhV(Word32 Rt, Word32 Mu, HVX_Vector Vv, HVX_Vector Vw)`
`vscatter(Rt,Mu,Vv.w).w+=Vw32`	`void Q6_vscatteracc_RMVwV(Word32 Rt, Word32 Mu, HVX_Vector Vv, HVX_Vector Vw)`
`vscatter(Rt,Mu,Vv.w).w=Vw32`	`void Q6_vscatter_RMVwV(Word32 Rt, 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

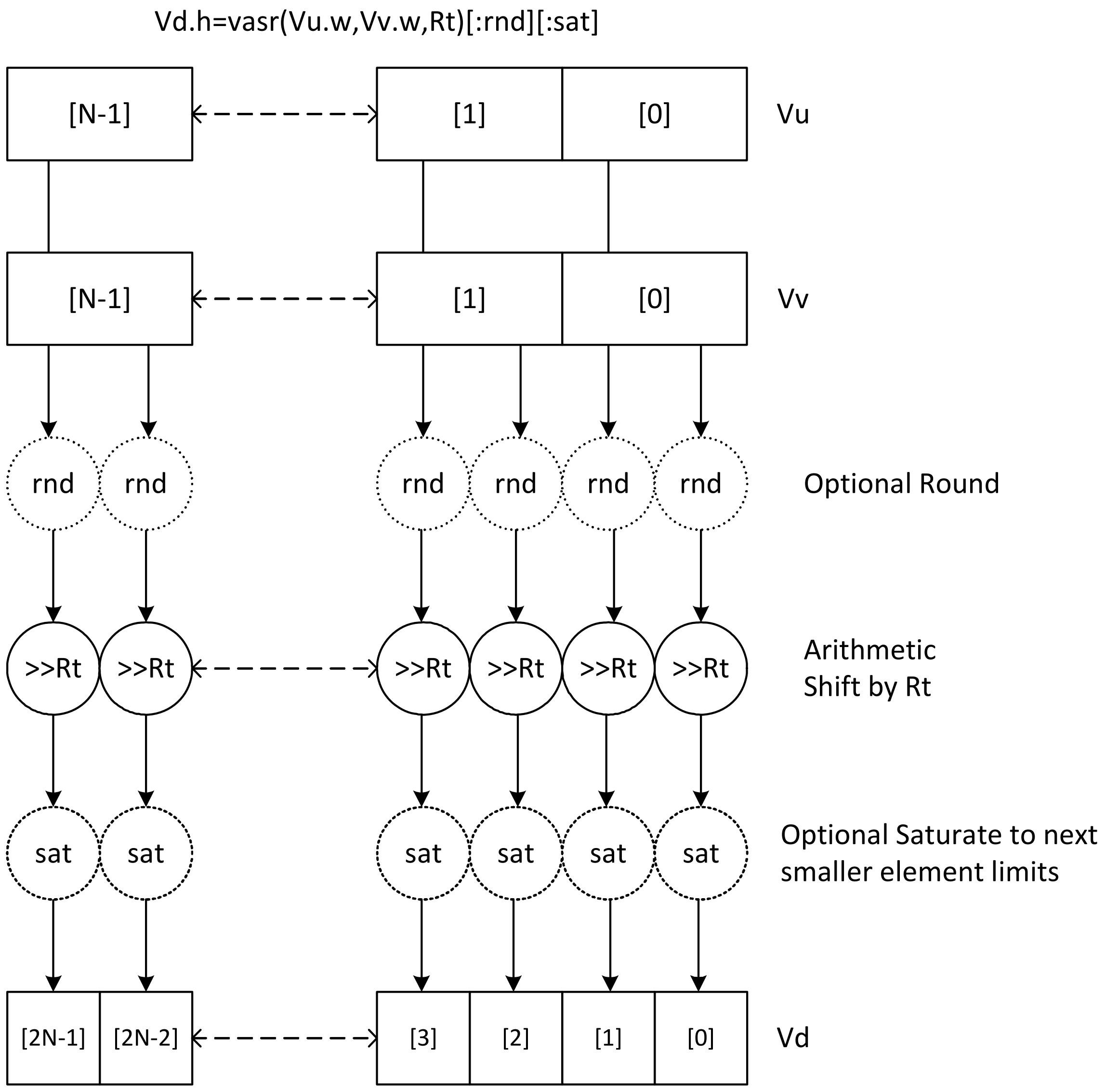
Field name	Description
`ICLASS`	Instruction Class
`NT`	NonTemporal
`Parse`	Packet/Loop parse bits
`s2`	Field to encode register s
`t5`	Field to encode register t
`u1`	Field to encode register u
`v5`	Field to encode register v

5.14 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, and each odd element in Vd comes from the vector register Vu.



Syntax	Behavior

`Vd.b=vasr(Vu.h,Vv.h,Rt)[:rnd]:sat`	`for (i = 0; i < VELEM(16); i++) {`
	`    shamt = Rt & 0x7;`
	`    Vd.h[i].b[0]=sat`$_8$`(Vv.h[i] + (1<<(shamt-1)) >> shamt);`
	`    Vd.h[i].b[1]=sat`$_8$`(Vu.h[i] + (1<<(shamt-1)) >> shamt);`
	`}`

`Vd.h=vasr(Vu.w,Vv.w,Rt):rnd:sat`	`for (i = 0; i < VELEM(32); i++) {`
	`    shamt = Rt & 0xF;`
	`    Vd.w[i].h[0]=sat`$_{16}$`(Vv.w[i] + (1<<(shamt-1)) >> shamt);`
	`    Vd.w[i].h[1]=sat`$_{16}$`(Vu.w[i] + (1<<(shamt-1)) >> shamt);`
	`}`

`Vd.h=vasr(Vu.w,Vv.w,Rt)[:sat]`	`for (i = 0; i < VELEM(32); i++) {`
	`    shamt = Rt & 0xF;`
	`    Vd.w[i].h[0]=[sat`$_{16}$`](Vv.w[i] >> shamt);`
	`    Vd.w[i].h[1]=[sat`$_{16}$`](Vu.w[i] >> shamt);`
	`}`

`Vd.ub=vasr(Vu.h,Vv.h,Rt)[:rnd]:sat`	`for (i = 0; i < VELEM(16); i++) {`
	`    shamt = Rt & 0x7;`
	`    Vd.h[i].b[0]=usat`$_8$`(Vv.h[i] + (1<<(shamt-1)) >> shamt);`
	`    Vd.h[i].b[1]=usat`$_8$`(Vu.h[i] + (1<<(shamt-1)) >> shamt);`
	`}`

`Vd.ub=vasr(Vu.uh,Vv.uh,Rt)[:rnd]:sat`	`for (i = 0; i < VELEM(16); i++) {`
	`    shamt = Rt & 0x7;`
	`    Vd.uh[i].b[0]=usat`$_8$`(Vv.uh[i] + (1<<(shamt-1)) >> shamt);`
	`    Vd.uh[i].b[1]=usat`$_8$`(Vu.uh[i] + (1<<(shamt-1)) >> shamt);`
	`}`

`Vd.uh=vasr(Vu.uw,Vv.uw,Rt)[:rnd]:sat`	`for (i = 0; i < VELEM(32); i++) {`
	`    shamt = Rt & 0xF;`
	`    Vd.uw[i].h[0]=usat`$_{16}$`(Vv.uw[i] + (1<<(shamt-1)) >> shamt);`
	`    Vd.uw[i].h[1]=usat`$_{16}$`(Vu.uw[i] + (1<<(shamt-1)) >> shamt);`
	`}`

`Vd.uh=vasr(Vu.w,Vv.w,Rt)[:rnd]:sat`	`for (i = 0; i < VELEM(32); i++) {`
	`    shamt = Rt & 0xF;`
	`    Vd.w[i].h[0]=usat`$_{16}$`(Vv.w[i] + (1<<(shamt-1)) >> shamt);`
	`    Vd.w[i].h[1]=usat`$_{16}$`(Vu.w[i] + (1<<(shamt-1)) >> shamt);`
	`}`

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

Notes

- This instruction may not work correctly in Napali V1.

- Input scalar register Rt is limited to registers 0 through 7
- 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

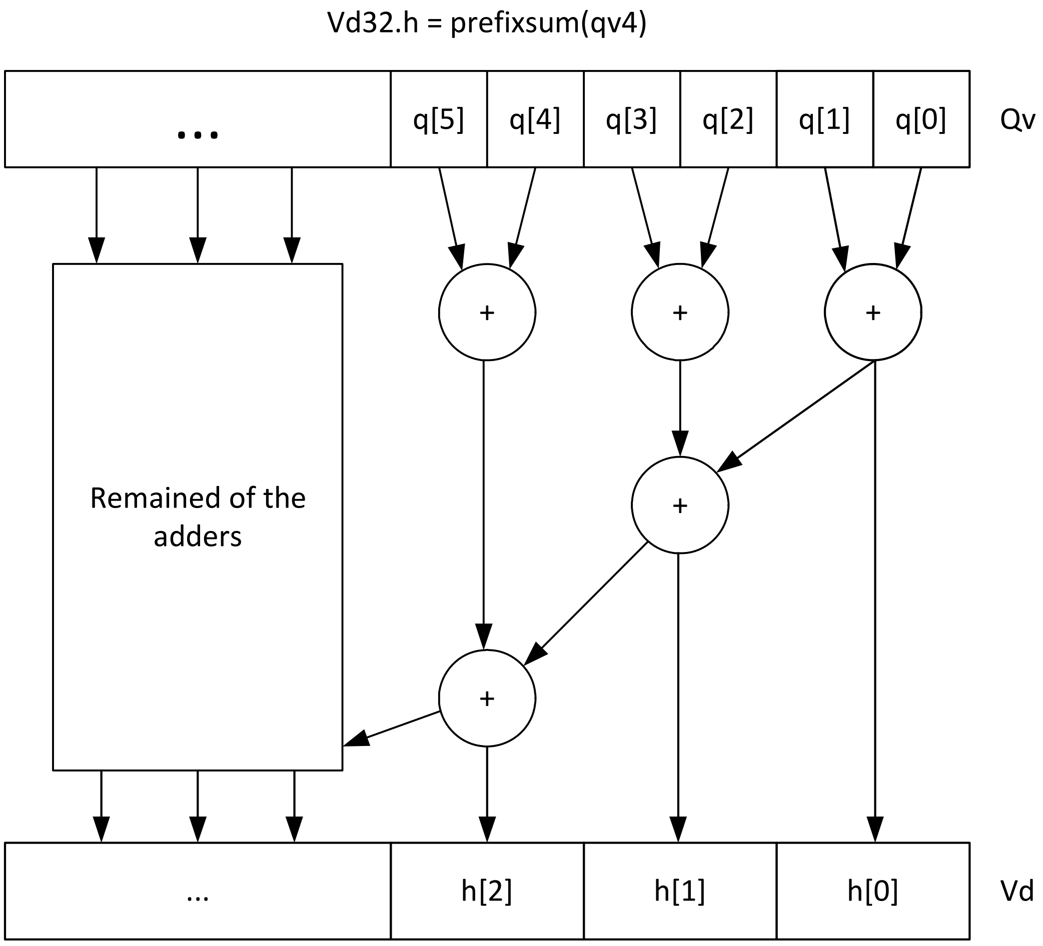
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS													t3			Parse			u5								d5					
0	0	0	1	1	0	0	0	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d	d	Vd.b=vasr(Vu.h,Vv.h,Rt):sat
0	0	0	1	1	0	0	0	v	v	v	v	v	t	t	t	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d	d	Vd.uh=vasr(Vu.uw,Vv.uw,Rt):rnd:sat
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
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)

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	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
`v2`	Field to encode register v
`v3`	Field to encode register v

Compute contiguous offsets for valid positions

Perform a cumulative sum of the bits in the predicate register.



Syntax	Behavior

`Vd.b=prefixsum(Qv4)`	`for (i = 0; i < VELEM(8); i++) {`
	`    acc += QvV[i];`
	`    Vd.ub[i] = acc;`
	`}`

`Vd.h=prefixsum(Qv4)`	`for (i = 0; i < VELEM(16); i++) {`
	`    acc += QvV[i*2+0];`
	`    acc += QvV[i*2+1];`
	`    Vd.uh[i] = acc;`
	`}`

`Vd.w=prefixsum(Qv4)`	`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

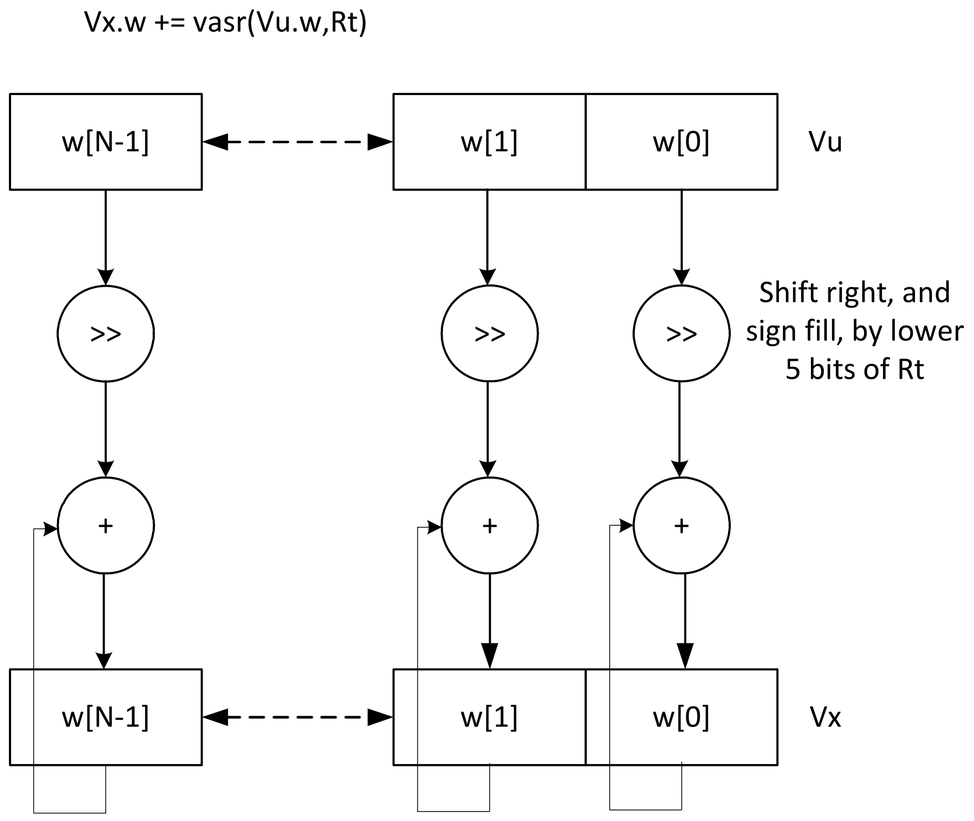
31	**30**	**29**	**28**	**27**	**26**	**25**	**24**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	
ICLASS																Parse											d5					
0	0	0	1	1	1	1	0	**v**	**v**	0	-	-	-	1	1	**P**	**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	-	-	-	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	-	-	-	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

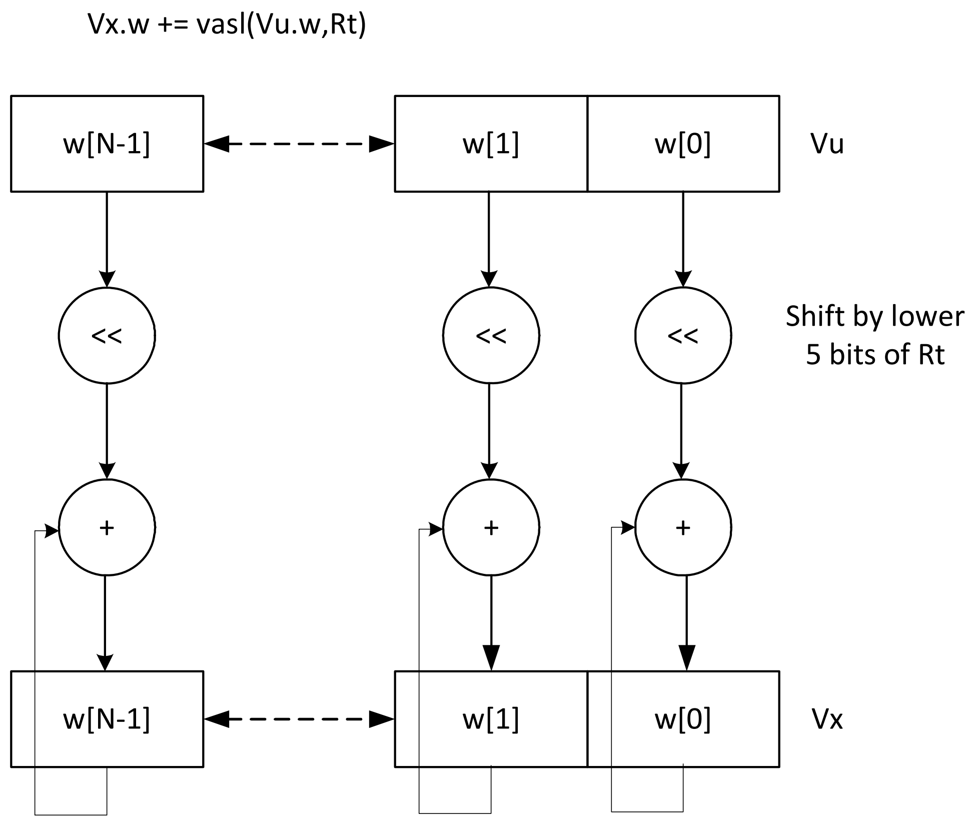
Shift and add

Each element in the vector register Vu is arithmetically shifted right by the value specified by the lower bits of the scalar register Rt. The result is added to the destination vector register Vx. For signed word shifts, the lower five bits of Rt specify the shift amount.

The left shift does not saturate the result to the element size.



*N is the number of operations implemented in each vector



*N is the number of operations implemented in each vector

Syntax	Behavior

`Vx.h+=vasl(Vu.h,Rt)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vx.h[i] += (Vu.h[i] << (Rt & (16-1)));`
	`}`

`Vx.h+=vasr(Vu.h,Rt)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vx.h[i] += (Vu.h[i] >> (Rt & (16-1)));`
	`}`

`Vx.w+=vasl(Vu.w,Rt)`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vx.w[i] += (Vu.w[i] << (Rt & (32-1)));`
	`}`

`Vx.w+=vasr(Vu.w,Rt)`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vx.w[i] += (Vu.w[i] >> (Rt & (32-1)));`
	`}`

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

Notes

- This instruction may not work correctly in Napali V1.

- This instruction uses the HVX shift resource.

Intrinsics

`Vx.h+=vasl(Vu.h,Rt)`	`HVX_Vector Q6_Vh_vaslacc_VhVhR(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)`
`Vx.h+=vasr(Vu.h,Rt)`	`HVX_Vector Q6_Vh_vasracc_VhVhR(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)`
`Vx.w+=vasl(Vu.w,Rt)`	`HVX_Vector Q6_Vw_vaslacc_VwVwR(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)`
`Vx.w+=vasr(Vu.w,Rt)`	`HVX_Vector Q6_Vw_vasracc_VwVwR(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)`

Encoding

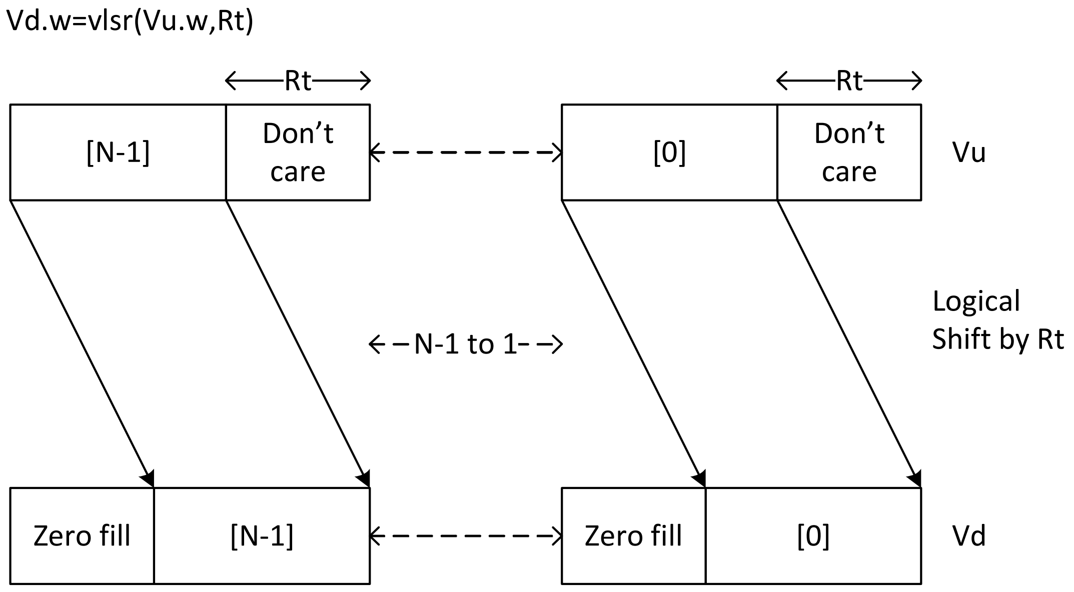
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											t5					Parse			u5								x5					
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	1	u	u	u	u	u	0	1	0	x	x	x	x	x	Vx.w+=vasl(Vu.w,Rt)
0	0	0	1	1	0	0	1	0	1	1	t	t	t	t	t	P	P	1	u	u	u	u	u	1	0	1	x	x	x	x	x	Vx.w+=vasr(Vu.w,Rt)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

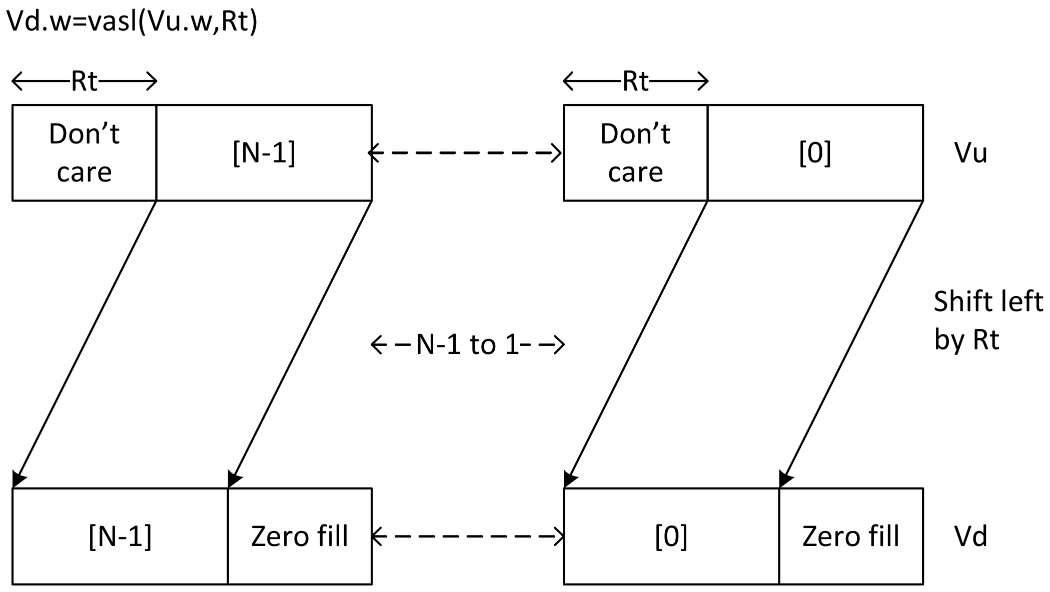
Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`t5`	Field to encode register t
`u5`	Field to encode register u
`x5`	Field to encode register x

Shift

Each element in the vector register Vu is arithmetically (logically) shifted right (left) by the value specified in the lower bits of the corresponding element of vector register Vv (or scalar register Rt). 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.





Syntax	**Behavior**
Vd.b=vasr(Vu.h,Vv.h,Rt)[:rnd]:sat	for (i = 0; i < VELEM(16); i++) { shamt = Rt & 0x7; Vd.h[i].b[0]=sat$_8$(Vv.h[i] + (1<<(shamt-1)) >> shamt); Vd.h[i].b[1]=sat$_8$(Vu.h[i] + (1<<(shamt-1)) >> shamt); }
Vd.h=vasl(Vu.h,Rt)	for (i = 0; i < VELEM(16); i++) { Vd.h[i] = (Vu.h[i] << (Rt & (16-1))); }
Vd.h=vasl(Vu.h,Vv.h)	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])); }
Vd.h=vasr(Vu.h,Rt)	for (i = 0; i < VELEM(16); i++) { Vd.h[i] = (Vu.h[i] >> (Rt & (16-1))); }
Vd.h=vasr(Vu.h,Vv.h)	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])); }
Vd.h=vasr(Vu.w,Vv.w,Rt):rnd:sat	for (i = 0; i < VELEM(32); i++) { shamt = Rt & 0xF; Vd.w[i].h[0]=sat$_{16}$(Vv.w[i] + (1<<(shamt-1)) >> shamt); Vd.w[i].h[1]=sat$_{16}$(Vu.w[i] + (1<<(shamt-1)) >> shamt); }
Vd.h=vasr(Vu.w,Vv.w,Rt)[:sat]	for (i = 0; i < VELEM(32); i++) { shamt = Rt & 0xF; Vd.w[i].h[0]=[sat$_{16}$](Vv.w[i] >> shamt); Vd.w[i].h[1]=[sat$_{16}$](Vu.w[i] >> shamt); }
Vd.h=vlsr(Vu.h,Vv.h)	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])); }
Vd.ub=vasr(Vu.h,Vv.h,Rt)[:rnd]:sat	for (i = 0; i < VELEM(16); i++) { shamt = Rt & 0x7; Vd.h[i].b[0]=usat$_8$(Vv.h[i] + (1<<(shamt-1)) >> shamt); Vd.h[i].b[1]=usat$_8$(Vu.h[i] + (1<<(shamt-1)) >> shamt); }

Syntax	Behavior

Vd.ub=vasr(Vu.uh,Vv.uh,Rt)[:rnd]:sat	for (i = 0; i < VELEM(16); i++) {
	    shamt = Rt & 0x7;
	    Vd.uh[i].b[0]=usat$_8$(Vv.uh[i] + (1<<(shamt-1)) >> shamt);
	    Vd.uh[i].b[1]=usat$_8$(Vu.uh[i] + (1<<(shamt-1)) >> shamt);
	}

Vd.ub=vlsr(Vu.ub,Rt)	for (i = 0; i < VELEM(8); i++) {
	    Vd.b[i] = Vu.ub[i] >> (Rt & 0x7);
	}

Vd.uh=vasr(Vu.uw,Vv.uw,Rt)[:rnd]:sat	for (i = 0; i < VELEM(32); i++) {
	    shamt = Rt & 0xF;
	    Vd.uw[i].h[0]=usat$_{16}$(Vv.uw[i] + (1<<(shamt-1)) >> shamt);
	    Vd.uw[i].h[1]=usat$_{16}$(Vu.uw[i] + (1<<(shamt-1)) >> shamt);
	}

Vd.uh=vasr(Vu.w,Vv.w,Rt)[:rnd]:sat	for (i = 0; i < VELEM(32); i++) {
	    shamt = Rt & 0xF;
	    Vd.w[i].h[0]=usat$_{16}$(Vv.w[i] + (1<<(shamt-1)) >> shamt);
	    Vd.w[i].h[1]=usat$_{16}$(Vu.w[i] + (1<<(shamt-1)) >> shamt);
	}

Vd.uh=vlsr(Vu.uh,Rt)	for (i = 0; i < VELEM(16); i++) {
	    Vd.uh[i] = (Vu.uh[i] >> (Rt & (16-1)));
	}

Vd.uw=vlsr(Vu.uw,Rt)	for (i = 0; i < VELEM(32); i++) {
	    Vd.uw[i] = (Vu.uw[i] >> (Rt & (32-1)));
	}

Vd.w=vasl(Vu.w,Rt)	for (i = 0; i < VELEM(32); i++) {
	    Vd.w[i] = (Vu.w[i] << (Rt & (32-1)));
	}

Vd.w=vasl(Vu.w,Vv.w)	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]));
	}

Vd.w=vasr(Vu.w,Rt)	for (i = 0; i < VELEM(32); i++) {
	    Vd.w[i] = (Vu.w[i] >> (Rt & (32-1)));
	}

Vd.w=vasr(Vu.w,Vv.w)	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]));
	}

Vd.w=vlsr(Vu.w,Vv.w)	for (i = 0; i < VELEM(32); i++) {
	    Vd.uw[i] = (sxt$_{(5+1)->32}$(Vv.w[i])>0)?(Vu.uw[i]>>>sxt$_{(5+1)->32}$(Vv.w[i])):(Vu.uw[i]<<sxt$_{(5+1)->32}$(Vv.w[i]));
	}

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

Notes

- This instruction may not work correctly in Napali V1.
- Input scalar register Rt is limited to registers 0 through 7
- 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=vasl(Vu.h,Rt)`	`HVX_Vector Q6_Vh_vasl_VhR(HVX_Vector Vu, Word32 Rt)`
`Vd.h=vasl(Vu.h,Vv.h)`	`HVX_Vector Q6_Vh_vasl_VhVh(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.h=vasr(Vu.h,Rt)`	`HVX_Vector Q6_Vh_vasr_VhR(HVX_Vector Vu, Word32 Rt)`
`Vd.h=vasr(Vu.h,Vv.h)`	`HVX_Vector Q6_Vh_vasr_VhVh(HVX_Vector Vu, HVX_Vector Vv)`
`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.h=vlsr(Vu.h,Vv.h)`	`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

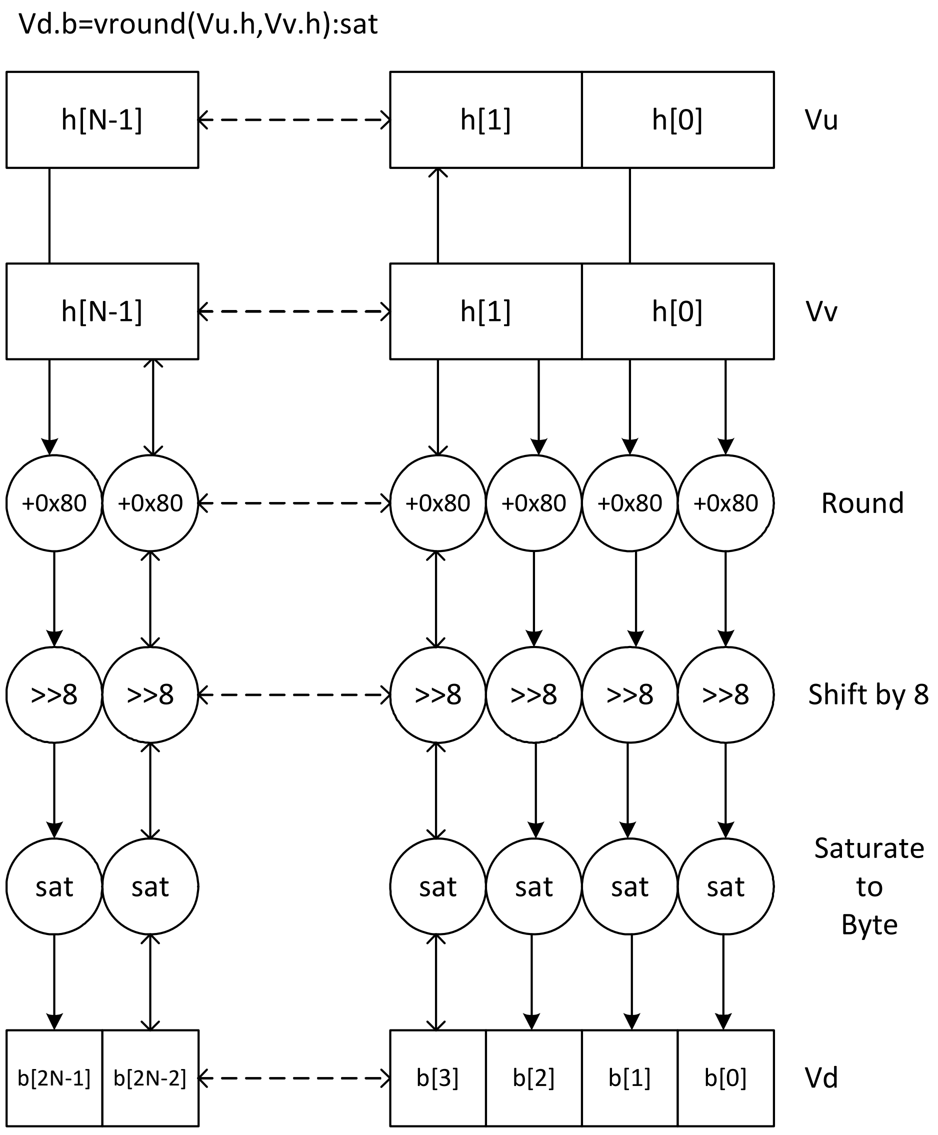
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS													t3			Parse			u5								d5					
0	0	0	1	1	0	0	0	**v**	**v**	**v**	**v**	**v**	**t**	**t**	**t**	**P**	**P**	0	**u**	**u**	**u**	**u**	**u**	0	0	0	**d**	**d**	**d**	**d**	**d**	Vd.b=vasr(Vu.h,Vv.h,Rt):sat
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

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	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
v2	Field to encode register v
v3	Field to encode register v
v5	Field to encode register v

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 8 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	Behavior

`Vd.b=vround(Vu.h,Vv.h):sat`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.uh[i].b[0]=sat`$_8$`((Vv.h[i] + 0x80) >> 8);`
	`    Vd.uh[i].b[1]=sat`$_8$`((Vu.h[i] + 0x80) >> 8);`
	`}`

`Vd.h=vround(Vu.w,Vv.w):sat`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.uw[i].h[0]=sat`$_{16}$`((Vv.w[i] + 0x8000) >>`
	`16);`
	`    Vd.uw[i].h[1]=sat`$_{16}$`((Vu.w[i] + 0x8000) >>`
	`16);`
	`}`

`Vd.ub=vround(Vu.h,Vv.h):sat`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.uh[i].b[0]=usat`$_8$`((Vv.h[i] + 0x80) >> 8);`
	`    Vd.uh[i].b[1]=usat`$_8$`((Vu.h[i] + 0x80) >> 8);`
	`}`

`Vd.ub=vround(Vu.uh,Vv.uh):sat`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.uh[i].b[0]=usat`$_8$`((Vv.uh[i] + 0x80) >> 8);`
	`    Vd.uh[i].b[1]=usat`$_8$`((Vu.uh[i] + 0x80) >> 8);`
	`}`

`Vd.uh=vround(Vu.uw,Vv.uw):sat`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.uw[i].h[0]=usat`$_{16}$`((Vv.uw[i] + 0x8000) >>`
	`16);`
	`    Vd.uw[i].h[1]=usat`$_{16}$`((Vu.uw[i] + 0x8000) >>`
	`16);`
	`}`

`Vd.uh=vround(Vu.w,Vv.w):sat`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.uw[i].h[0]=usat`$_{16}$`((Vv.w[i] + 0x8000) >>`
	`16);`
	`    Vd.uw[i].h[1]=usat`$_{16}$`((Vu.w[i] + 0x8000) >>`
	`16);`
	`}`

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

Notes

- This instruction uses the HVX shift resource.

Intrinsics

`Vd.b=vround(Vu.h,Vv.h):sat`	`HVX_Vector Q6_Vb_vround_VhVh_sat(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.h=vround(Vu.w,Vv.w):sat`	`HVX_Vector Q6_Vh_vround_VwVw_sat(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.ub=vround(Vu.h,Vv.h):sat`	`HVX_Vector Q6_Vub_vround_VhVh_sat(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.ub=vround(Vu.uh,Vv.uh):sat`	`HVX_Vector Q6_Vub_vround_VuhVuh_sat(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.uh=vround(Vu.uw,Vv.uw):sat`	`HVX_Vector Q6_Vuh_vround_VuwVuw_sat(HVX_Vector Vu, HVX_Vector Vv)`
`Vd.uh=vround(Vu.w,Vv.w):sat`	`HVX_Vector Q6_Vuh_vround_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	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

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 (vcl0) 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 non-zero 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

`Vd.h=vadd(vclb(Vu.h),Vv.h)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.h[i] =`
	`max(count_leading_ones(~Vu.h[i]),count_leading_o`
	`nes(Vu.h[i])) + Vv.h[i];`
	`}`

`Vd.h=vnormamt(Vu.h)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.h[i]=max(count_leading_ones(~Vu.h[i]),cou`
	`nt_leading_ones(Vu.h[i]))-1;`
	`;`
	`}`

`Vd.h=vpopcount(Vu.h)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.uh[i]=count_ones(Vu.uh[i]);`
	`}`

`Vd.uh=vcl0(Vu.uh)`	`for (i = 0; i < VELEM(16); i++) {`
	`    Vd.uh[i]=count_leading_ones(~Vu.uh[i]);`
	`}`

`Vd.uw=vcl0(Vu.uw)`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.uw[i]=count_leading_ones(~Vu.uw[i]);`
	`}`

`Vd.w=vadd(vclb(Vu.w),Vv.w)`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.w[i] =`
	`max(count_leading_ones(~Vu.w[i]),count_leading_o`
	`nes(Vu.w[i])) + Vv.w[i];`
	`}`

`Vd.w=vnormamt(Vu.w)`	`for (i = 0; i < VELEM(32); i++) {`
	`    Vd.w[i]=max(count_leading_ones(~Vu.w[i]),cou`
	`nt_leading_ones(Vu.w[i]))-1;`
	`;`
	`}`

`Vd=vnormamth(Vu)`	`Assembler mapped to: "Vd.h=vnormamt(Vu.h)"`
`Vd=vnormamtw(Vu)`	`Assembler mapped to: "Vd.w=vnormamt(Vu.w)"`

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)
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
`v5`	Field to encode register v

5.15 HVX/STORE

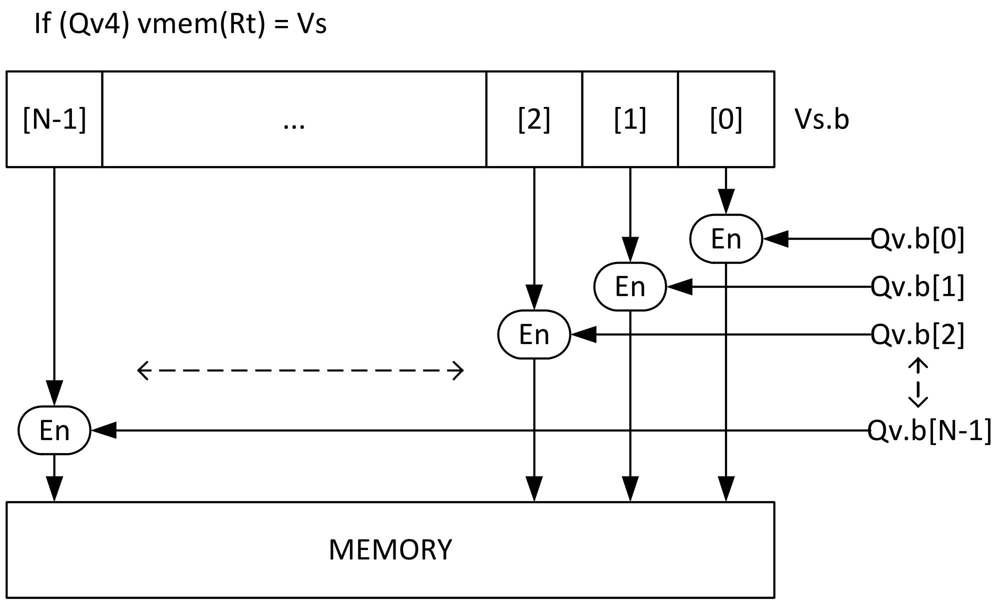
The HVX/STORE instruction subclass includes memory store instructions.

Store - byte-enabled aligned

Of the bytes in vector register Vs, store to memory only the ones 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
`if ([!]Qv4) vmem(Rt):nt=Vs`	`Assembler mapped to: "if ([!]Qv4) vmem(Rt+#0):nt=Vs"`
`if ([!]Qv4) vmem(Rt)=Vs`	`Assembler mapped to: "if ([!]Qv4) vmem(Rt+#0)=Vs"`

`if ([!]Qv4) vmem(Rt+#s4):nt=Vs`	`EA=Rt+#s*VBYTES;`
	`*(EA&~(ALIGNMENT-1)) = Vs;`

`if ([!]Qv4) vmem(Rt+#s4)=Vs`	`EA=Rt+#s*VBYTES;`
	`*(EA&~(ALIGNMENT-1)) = Vs;`

`if ([!]Qv4) vmem(Rx++#s3):nt=Vs`	`EA=Rx;`
	`*(EA&~(ALIGNMENT-1)) = Vs;`
	`Rx=Rx+#s*VBYTES;`

`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 "nontemporal" hint to the micro-architecture 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	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
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

`if ([!]Pv) vmem(Rt+#s4):nt=Os8.new`	`if ([!]Pv[0]) {`
	`    EA=Rt+#s*VBYTES;`
	`    *(EA&~(ALIGNMENT-1)) = OsN.new;`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) vmem(Rt+#s4)=Os8.new`	`if ([!]Pv[0]) {`
	`    EA=Rt+#s*VBYTES;`
	`    *(EA&~(ALIGNMENT-1)) = OsN.new;`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) vmem(Rx++#s3):nt=Os8.new`	`if ([!]Pv[0]) {`
	`    EA=Rx;`
	`    *(EA&~(ALIGNMENT-1)) = OsN.new;`
	`    Rx=Rx+#s*VBYTES;`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) vmem(Rx++#s3)=Os8.new`	`if ([!]Pv[0]) {`
	`    EA=Rx;`
	`    *(EA&~(ALIGNMENT-1)) = OsN.new;`
	`    Rx=Rx+#s*VBYTES;`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) vmem(Rx++Mu):nt=Os8.new`	`if ([!]Pv[0]) {`
	`    EA=Rx;`
	`    *(EA&~(ALIGNMENT-1)) = OsN.new;`
	`    Rx=Rx+MuV;`
	`} else {`
	`    NOP;`
	`}`

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 "nontemporal" hint to the micro-architecture 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
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

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		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 simply 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
`if ([!]Pv) vmem(Rt):nt=Vs`	`Assembler mapped to: "if ([!]Pv) vmem(Rt+#0):nt=Vs"`
`if ([!]Pv) vmem(Rt)=Vs`	`Assembler mapped to: "if ([!]Pv) vmem(Rt+#0)=Vs"`

`if ([!]Pv) vmem(Rt+#s4):nt=Vs`	`if ([!]Pv[0]) {`
	`    EA=Rt+#s*VBYTES;`
	`    *(EA&~(ALIGNMENT-1)) = Vs;`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) vmem(Rt+#s4)=Vs`	`if ([!]Pv[0]) {`
	`    EA=Rt+#s*VBYTES;`
	`    *(EA&~(ALIGNMENT-1)) = Vs;`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) vmem(Rx++#s3):nt=Vs`	`if ([!]Pv[0]) {`
	`    EA=Rx;`
	`    *(EA&~(ALIGNMENT-1)) = Vs;`
	`    Rx=Rx+#s*VBYTES;`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) vmem(Rx++#s3)=Vs`	`if ([!]Pv[0]) {`
	`    EA=Rx;`
	`    *(EA&~(ALIGNMENT-1)) = Vs;`
	`    Rx=Rx+#s*VBYTES;`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) vmem(Rx++Mu):nt=Vs`	`if ([!]Pv[0]) {`
	`    EA=Rx;`
	`    *(EA&~(ALIGNMENT-1)) = Vs;`
	`    Rx=Rx+MuV;`
	`} else {`
	`    NOP;`
	`}`

Syntax	Behavior

`if ([!]Pv) vmem(Rx++Mu)=Vs`	`if ([!]Pv[0]) {`
	`    EA=Rx;`
	`    *(EA&~(ALIGNMENT-1)) = Vs;`
	`    Rx=Rx+MuV;`
	`} else {`
	`    NOP;`
	`}`

`vmem(Rt):nt=Vs`	`Assembler mapped to: "vmem(Rt+#0):nt=Vs"`
`vmem(Rt)=Vs`	`Assembler mapped to: "vmem(Rt+#0)=Vs"`

`vmem(Rt+#s4):nt=Vs`	`EA=Rt+#s*VBYTES;`
	`*(EA&~(ALIGNMENT-1)) = Vs;`

`vmem(Rt+#s4)=Vs`	`EA=Rt+#s*VBYTES;`
	`*(EA&~(ALIGNMENT-1)) = Vs;`

`vmem(Rx++#s3):nt=Vs`	`EA=Rx;`
	`*(EA&~(ALIGNMENT-1)) = Vs;`
	`Rx=Rx+#s*VBYTES;`

`vmem(Rx++#s3)=Vs`	`EA=Rx;`
	`*(EA&~(ALIGNMENT-1)) = Vs;`
	`Rx=Rx+#s*VBYTES;`

`vmem(Rx++Mu):nt=Vs`	`EA=Rx;`
	`*(EA&~(ALIGNMENT-1)) = Vs;`
	`Rx=Rx+MuV;`

`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 "nontemporal" hint to the micro-architecture 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	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	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	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	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	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	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	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

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	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	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	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	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	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	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	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
0	0	1	0	1	0	1	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	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	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	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

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. Care should be taken to use aligned memory operations and combinations of permute operations, when possible.

Note that this instruction uses both slot 0 and slot 1, allowing only three instructions at most to execute in a packet with vmemu in it.

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
`if ([!]Pv) vmemu(Rt)=Vs`	`Assembler mapped to: "if ([!]Pv) vmemu(Rt+#0)=Vs"`

`if ([!]Pv) vmemu(Rt+#s4)=Vs`	`if ([!]Pv[0]) {`
	`    EA=Rt+#s*VBYTES;`
	`    *EA = Vs;`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) vmemu(Rx++#s3)=Vs`	`if ([!]Pv[0]) {`
	`    EA=Rx;`
	`    *EA = Vs;`
	`    Rx=Rx+#s*VBYTES;`
	`} else {`
	`    NOP;`
	`}`

`if ([!]Pv) vmemu(Rx++Mu)=Vs`	`if ([!]Pv[0]) {`
	`    EA=Rx;`
	`    *EA = Vs;`
	`    Rx=Rx+MuV;`
	`} else {`
	`    NOP;`
	`}`

`vmemu(Rt)=Vs`	`Assembler mapped to: "vmemu(Rt+#0)=Vs"`

`vmemu(Rt+#s4)=Vs`	`EA=Rt+#s*VBYTES;`
	`*EA = Vs;`

`vmemu(Rx++#s3)=Vs`	`EA=Rx;`
	`*EA = Vs;`
	`Rx=Rx+#s*VBYTES;`

`vmemu(Rx++Mu)=Vs`	`EA=Rx;`
	`*EA = Vs;`
	`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											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

Scatter release

Specialized store that follows outstanding scatters or gathers to make sure they complete. When written to VTCM space, no data is stored.

A VMEM load from the same address (Rt) causes a stalling synchronization until the scatter release operation completes.

The EA of the store release must be in the VTCM, otherwise it is dropped

Syntax	**Behavior**

`vmem(Rt+#s4):scatter_release`	`EA=Rt+#s*VBYTES;`
	`char* addr = EA&~(ALIGNMENT-1);`
	`Zero Byte Store Release (Non-blocking Sync);`

`vmem(Rx++#s3):scatter_release`	`EA=Rx;`
	`char* addr = EA&~(ALIGNMENT-1);`
	`Zero Byte Store Release (Non-blocking Sync);`
	`Rx=Rx+#s*VBYTES;`

`vmem(Rx++Mu):scatter_release`	`EA=Rx;`
	`char* addr = EA&~(ALIGNMENT-1);`
	`Zero Byte Store Release (Non-blocking Sync);`
	`Rx=Rx+MuV;`

Class: COPROC_VMEM (slots 0)

Notes

- This instruction can use any HVX 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																
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	-	1	-	-	-	vmem(Rt+#s4):scatter_release
ICLASS									NT		x5					Parse																
0	0	1	0	1	0	0	1	0	0	1	x	x	x	x	x	P	P	-	-	-	i	i	i	0	0	1	-	1	-	-	-	vmem(Rx++#s3):scatter_release
ICLASS									NT		x5					Parse		u1														
0	0	1	0	1	0	1	1	0	0	1	x	x	x	x	x	P	P	u	-	-	-	-	-	0	0	1	-	1	-	-	-	vmem(Rx++Mu):scatter_release

Field name	**Description**
`ICLASS`	Instruction Class
`NT`	NonTemporal
`Parse`	Packet/Loop parse bits

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

Instruction Index