Qualcomm Technologies, Inc.

Qualcomm Hexagon v79 HVX Programmer Reference Manual

80-N2040-61 AB

January 16, 2025

Contents

1 Introduction

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

1.1 SIMD coprocessor

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

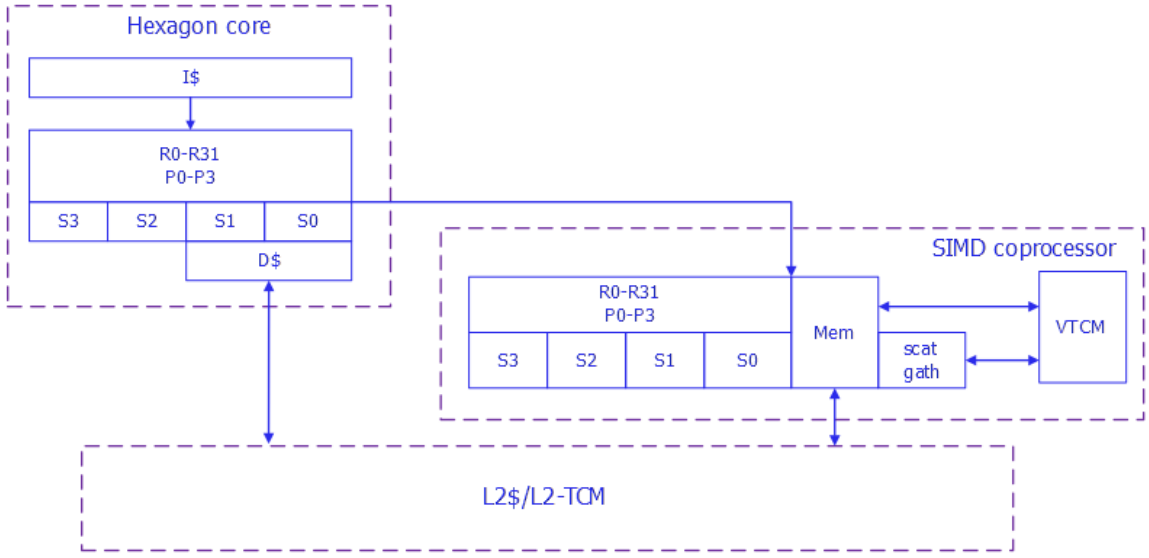


Figure1 Hexagon core with attached SIMD coprocessor

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

very long instruction word (VLIW) packet. HVX instructions can also use scalar source operands from the core.

1.2 HVX features

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

Vector length

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

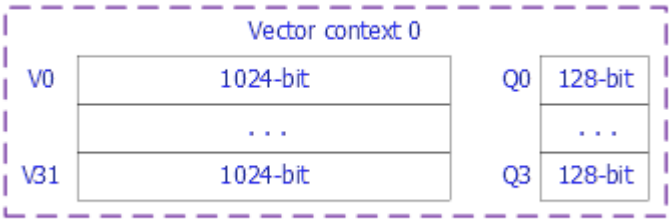


Figure2 Registers using 128 byte with a vector length of 1024 bits

Vector contexts

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

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

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

- SSR.XA = 5: HVX instructions use vector context 5, if it is available.
- SSR.XA = 6: HVX instructions use vector context 6, if it is available.
- SSR.XA = 7: HVX instructions use vector context 7, if it is available.

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

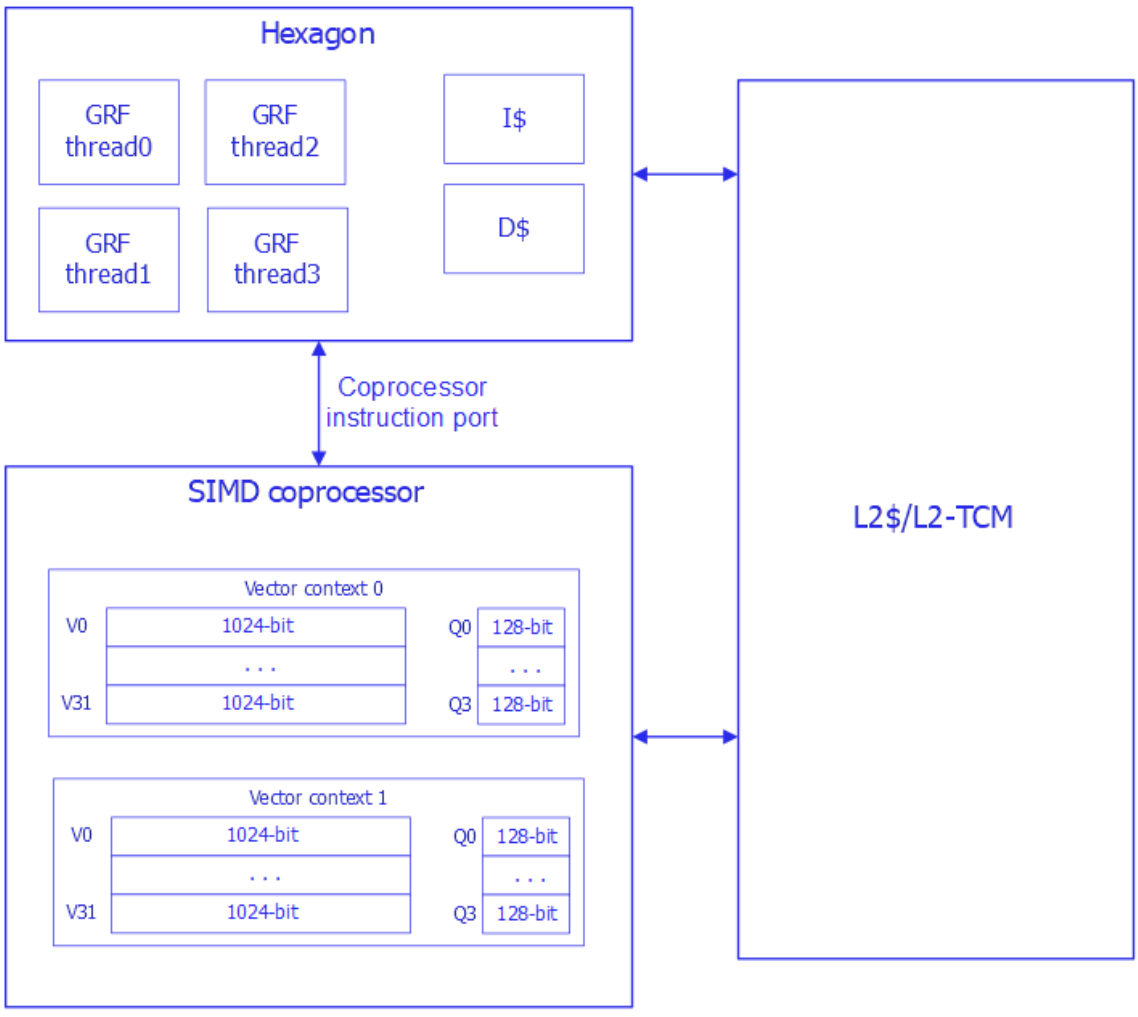


Figure3 Four hardware threads (two HVX-enabled threads and two scalar-only threads)

Memory access

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

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

Vector registers

HVX has two sets of registers:

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

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

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

Element ordering is little-endian with the lowest byte in the least-significant position, as shown in 1024-bit SIMD register.

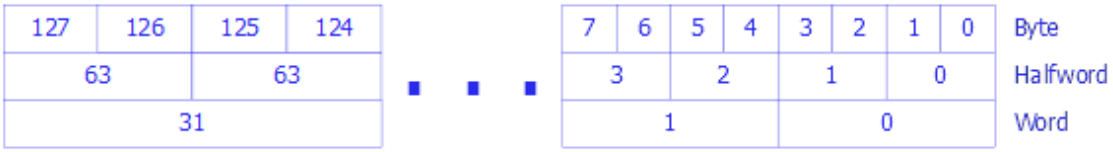


Figure4 1024-bit SIMD register

Vector compute instructions

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

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

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

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

- .b for signed byte
- .ub for unsigned byte
- .h for signed halfword
- .uh for unsigned halfword
- .w for signed word
- .uw for unsigned word
- .qf16 for 16-bit HVX floating point
- .qf32 for 32-bit HVX floating point
- .hf for half precision
- .sf for single precision For example:

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

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

2 HVX Revision history

HVX revision history

Revision	Date	Description
AA	December 2024	• Permute resource now allows shift resource instructions (See VLIW packing rules) • Qfloat is now referred to as HVX floating point – HVX floating point changed to achieve IEEE-754 compliance • New multiply by scalar register HVX floating point instructions – Vd.qf16=vmpy(Vu.hf,Rt.hf) – Vd.qf16=vmpy(Vu.qf16,Rt.hf) – Vd.qf32=vmpy(Vu.sf,Rt.sf) • Updated reversed vector pair wording • Added Dot-new vector operations description

3 Registers

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

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

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

3.1 Vector data registers

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

For example:

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

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

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

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

Reversed vector pairs

Reversed pairs are supported for vector pair register operands and input operands.

The example below demonstrates the destination and first source register pair having its register number flipped.

Original instructions

```
v7:6.b = vadd(v3:2.b, v4:5.b)       // Add vector pairs of bytes
v1:0.h = vadd(v13:12.h, v5:4.h)     // Add vector pairs of halfwords
```

Reversed instructions

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

This changes the instruction input operations. In the examples above, the original byte add computes:

```
V7.b = V3.b + V4.b
V6.b = V2.b + V5.b
```

The reversed byte add computes:

```
V6.b = V2.b + V4.b
V7.b = V3.b + V5.b
```

VRF to GRF transfers

VRF to GRF transfer instructions lists the Hexagon instructions that transfer values between the vector register file (VRF) and the general register file (GRF).

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

VRF to 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.

3.2 Vector predicate registers

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

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

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

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

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

4 Memory

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

4.1 Alignment

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

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

The intra-vector addressing bits are ignored.

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

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

4.2 HVX local memory: VTCM

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

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

- Guarantees no eviction (vs. L2 if the set is full)
- Faster than L2$ (does not have the overhead of cache management, like association)
- Reduces L2$ pressure
- Lower power than L2$

- Supports continuous read and write for every packet without contention

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

The following conditions are invalid for VTCM access:

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

4.3 Scatter and gather

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

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

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

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

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

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

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

Operations via scatter or gather usually perform better via scatter. The following conditions are

invalid for scatter or gather access:

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

4.4 Memory type

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

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

4.5 Nontemporal

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

4.6 Permissions

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

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

4.7 Ordering

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

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

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

4.8 Atomicity

Atomicity of types of memory accesses 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.

Atomicity of types of memory accesses

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

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

4.9 Maximizing performance of the vector memory system

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

Minimize VMEM access

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

Use aligned data

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

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.

L2FETCH

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

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

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

Access data contiguously

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

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

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

- Micro-TLB misses. A limited number of pages are remembered for fast translation. Containing data to a smaller number of pages helps translation performance.

Use nontemporal for final data

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

Scalar processing of vector data

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

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

Avoid scatter/gather stalls

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

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

Peak scatter/gather performance for v79

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

Operation	Addressing	Vector bandwidth (per packet)	Latency (packets)
Scatter	Random	1/6	30
Gather	Random	1/6	48

5 Vector instructions

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

Slot resource latency summarizes Hexagon slot, HVX resource, and instruction latency for the instruction categories.

5.1 VLIW packing rules

HVX provides the following resources for vector instruction execution:

- load
- store
- shift
- permute/shift
- two multiply

Each HVX instruction consumes some combination of these resources, as defined in vector instuction resource usage. VLIW packets cannot oversubscribe resources.

An instruction packet can contain up to four instructions, plus an end loop. The instructions inside the packet must obey the packet grouping rules described in vector instruction.

The permute resource allows shift resource instructions.

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

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.

Vector instruction resource usage

HVX execution resource usage 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.

HVX execution resource usage

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

Vector instruction

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

HVX instruction to Hexagon slots mapping

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

5.2 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.

Load-temp and load-current

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

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

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

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

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

New-value store

VMEM store instructions can store a newly generated value from a vector register in the same packet. The instructions do not consume a vector ALU resource as they do not read nor write the register file. This feature is expressed in assembly language by appending the suffix `.new` to the source register. The store must be in slot 0.

```
{
 V20.w = vmax(V0.w, V1.w)
 vmem(R1+#1)= V20.new    // Store V20 that was generated in the
current packet
}
```

Predicate stores

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

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

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

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

5.3 Scatter and gather

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

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

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

Sources for noncontiguous accesses: (Rt, Mu, Vv)

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

To form an HVX gather (memory to memory), vgather is paired with a vector store to specify the destination address. A scatter is specified with a single instruction. Ignoring element sizes,

Basic scatter and gather instructions describes the basic forms of scatter and gather instructions:

Basic scatter and gather instructions

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

5.4 Memory instruction slot combinations

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

Valid VMEM load/store and scatter/gather combinations 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.

Valid VMEM load/store and scatter/gather combinations

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

5.5 Special instructions

Histogram

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

Example:

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

5.6 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 to issue and complete a packet on every context cycle. The contexts are time interleaved to share the hardware such that using all contexts might be required to reach peak compute bandwidth.

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

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

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

The following example shows interlock cases:

```
V8 = vadd(V0,V0)
V0 = vadd(V8,V9)      // NO STALL
V1 = vmpy(V0,R0)      // STALL due to V0
V2 = vsub(V2,V1)      // NO STALL on V1
V5:4 = vunpack(V2)    // STALL due to V2
V2 = vadd(V0,V4)      // NO STALL on V4
```

Avoiding accumulator stalls

A 3 vector source using an accumulator (Vx) must be produced in the prior packet to avoid stalling.

The HVX_ACC_ORDER PMU event indicates stalling due to not following this rule.

The following example shows accumulator stall with 3 vector source instructions.

```
{
  V18.w += vrmpy(V0.b,V1.b)
}
{
  V24.w += vrmpy(V2.b, V3.b)     // Previous packet does not produces
V24
}
```

The following examples show non-stalling accumulator with 3 vector source instructions.

```
// Example 1:
{
  V24.w += vrmpy(V0.b,V1.b)
}
{
  V24.w += vrmpy(V2.b, V3.b)     // Previous packet produces V24
}

// Example 2:
{
  V24 = #0
}
{
  V24.w += vrmpy(V2.b, V3.b)     // Previous packet produces V24
}
```

5.7 Slot/resource/latency summary

HVX slot/resource/latency summary summarizes the Hexagon slot, HVX resource, and latency requirements for HVX instruction types.

HVX slot/resource/latency summary

		Core slots				Vector resources						Input latency
Category	**Variation**	**3**	**2**	**1**	**0**	**ld**	**mpy**	**mpy**	**shift**	**xlane**	**st**	
ALU	**1 vector**	any					any					1

	2 vectors	any				either pair						1
	Rt	either				either						1
Abs-diff	**1 vector**	either				either						2
	2 vector	either										2
Multiply	**by 8 bits; 1 vector**	either				either						2
	by 8 bits; 2 vector	either										2
	by 16 bits	either										2
Cross-lane	**1 vector**	any										2
	2 vectors	any										2
Shift or count	**1 vector**	any										2
load	**aligned**			either		any						-
	aligned; .tmp			either								-
	aligned; .cur			either		any						-
	unaligned											-
store	**aligned**					any						1
	aligned; .new											0
	unaligned											2
gather (needs .new store)	**1 vector**					any						1
	2 vector					either pair						1
scatter	**1 vector**					any						1
	2 vector					either pair						1
histogram (needs .tmp load)		any										2
extract												1

6 HVX floating point

V79 replaces the prior internal HVX floating point format for floating-point arithmetic. The new internal HVX floating-point format yields results that are identical to IEEE-754 round-to-even mode. The new format contains more bits than IEEE-754, which optionally produces results with greater range and accuracy.

Only the HVX vector registers use the HVX floating-point format. Memory maintains floating- point data in IEEE-754 format, and all loads/stores use the IEEE-754 format. A subset of HVX floating-point operations transform IEEE-754 floating-point data to HVX floating-point data. Subsequent HVX floating-point instructions can consume operands in the HVX floating-point without conversion to IEEE-754, which allows for performant and energy efficient code. The program does not need to continuously switch between formats. The program must convert the HVX floating-point results to IEEE-754 prior to storing to memory.

HVX floating-point achieves IEEE-754 compliance through normalization. The program can skip normalization when faster calculation is needed, and IEEE-754 compliance is not required.

6.1 Programming with HVX floating point

HVX floating-point contains two input types:

- qf32: single precision floating point
- qf16: half precision floating point

In Hexagon, IEEE-754 contains two input types:

- sf: single precision floating point
- hf: half precision floating point.

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

Handling the extended state of HVX floating-point

Only HVX floating-point source and destination instructions use HVX floating-point values. Instructions specify the HVX floating-point format with the `qf16` and `qf32` identifier. A source vector register drops the extended state of a HVX floating-point value when an instruction reads the source vector register without the `qf16` or `qf32` identifier. A destination vector register resets its extended state when an instruction writes to a vector register without the `qf16` or `qf32` identifier. When dropping the extended state, the floating-point value loses accuracy. The program can preserve the floating-point value by converting HVX floating-point values to IEEE- 754 values. Software must convert HVX floating-point values to IEEE-754 values before using as an input to stores, permutes, shifts, and any other operations that do not source the HVX floating- point format.

Examples of handling HVX floating-point value

Example of dropping extended state

```
V0.qf32 = vadd(V1.sf,V2.sf) // V0.qf32 holds extended state
vmem(R0) = V0               // Extended state dropped, incorrect
floating-point value stored
```

Example of resetting extended state

```
V0.qf32 = vadd(V1.sf,V2.sf) // V0.qf32 holds extended state
V0 = V0                     // Extended state reset, floating-point
value lost
```

Example of preserving floating-point value

```
V0.qf32 = vadd(V1.sf,V2.sf)  // V0.qf32 holds extended state
V0.sf = V0.qf32              // Extended state converted to IEEE-754
vmem(R0) = V0                // Value preserved and properly stored
to memory
```

Rules to achieve IEEE-754 compliance

Depending on the desired results, HVX floating-point operations have requirements on the input sources. The HVX floating-point values require normalization to achieve IEEE-754 compliance, while faster operations can skip normalization. The program normalizes HVX floating-point values before subsequent HVX floating-point operations, so the floating-point value does not lose precision.

The program also obtains results identical to IEEE-754 by converting all HVX floating-point results to IEEE-754 format before consumed in any subsequent operation. However, there are cases where this conversion is redundant, or the differences between IEEE-754 and HVX floating-point might not be a concern.

The table below describes when a HVX floating-point operation directly consumes a floating-point value as a source, when the floating-point values need normalization, and when the floating-point values must be converted to IEEE-754 before an operation.

HVX floating point values

Instruction inputs	**Inputs to add/subtract instructions**			**Inputs to multiplication instructions**						**Non-HVX floating point instructions**		
Source	**IEEE-754**	**HVX floating point format from multi**	**HVX floating point format from multi**	**sf**	**qf32 from multi**	**qf32 from adder**	**hf**	**qf16 from multi**	**qf16 from adder**	**IEEE 754**	**HVX floating point from multi**	**HVX floating point from adder**
Strict IEEE-754 compliance	Direct use	Convert to IEEE	Convert to IEEE	Normalize	Convert to IEEE then normalize	Convert to IEEE then normalize	Widening multiply then convert to IEEE	Convert to IEEE, widening multiply, then convert to IEEE	Convert to IEEE, widening multiply, then convert to IEEE	Direct use	Convert to IEEE	Convert to IEEE
IEEE-754 compliance[a]	Direct use	Direct use	Direct use	Normalize	Direct use	Normalize	Widening multiply	Direct use	Widening multiply	Direct use	Convert to IEEE	Convert to IEEE
Lossy subnormals[b]	Direct use	Direct use	Direct use	Direct use	Direct use	Normalize	Direct use	Direct use	Widening multiply	Direct use	Convert to IEEE	Convert to IEEE
Similar accuracy to prior HVX floating-point[c]	Direct use	Direct use	Direct use	Direct use	Direct use	Direct use	Direct use	Direct use	Direct use	Direct use	Convert to IEEE	Convert to IEEE

a. Excludes IEEE-754 overflows and lower precision subnormals due to larger dynamic range than IEEE-754. All subnormals have extra precision. Results that would result in infinity for IEEE-754 can be represented as a finite value in HVX floating-point.

b. Using IEEE-754 subnormals without normalization results in a loss of accuracy. This provides greater precision than a clamp of subnormals to 0. When the data set excludes subnormals, the behavior is the same as the IEEE-754 Compliance row.

c. Loss of 1 bit of accuracy compared to IEEE-754.

HVX floating point normalization

Unnormal inputs for multiplication operands naturally yield non-IEEE-754-compliant results. A loss of up to a half unit of least precision (ULP) of input precision can occur. For more performant code, skip normalization when IEEE-754 compliance is not a requirement.

Use the following sequences to normalize.

qf16 and hf normalization

For qf16 and hf operands, use a widening multiply to qf32, then convert back to qf16. Otherwise, a non-widening multiply has an input error on unnormal qf16 and hf inputs.

The following examples are Hexagon assembly code for a widening multiply.

Using two hf inputs:

```
{
   v0.hf = some IEEE-754 number
   v1.hf = some IEEE-754 number
}
{
   v3:2.qf32 = vmpy(v0.hf, v1.hf) // Widening multiply to qf32
}
{
   v4.hf = v3:2.qf32 // (Optional) Convert back to hf for strict
IEEE-754 compliance
}
```

Using two qf16 inputs:

```
{
   v0.qf16 = some qf16 number
   v1.qf16 = some qf16 number
}
{
   v3:2.qf32 = vmpy(v0.qf16, qf16) // Widening multiply to qf32
}
{
   v4.hf = v3:2.qf32 // (Optional) Convert back to hf for strict
IEEE-754 compliance
}
```

qf32 and sf normalization

Add a calculated zero value, defined as -0 with a minimum exponent (-255), to an unnormal number to create a normalized HVX floating-point value. Store the calculated zero in a vector register for reuse in future operations that require normalization.

The following examples are Hexagon assembly code to calculate -0 and normalize.

Using two sf inputs:

```
{
   v0.sf = some IEEE-754 number
   v1.sf = some IEEE-754 number
}
{
   v3:sf = 0x0 // IEEE-754 0
   v4.sf = 0x80000000 // IEEE-754 -0
}
{
   v3.qf32 = vmpy(v3.sf, v4.sf) // Create -0 with exponent at qf32
emin
}
{
   v0.qf32 = vadd(v3.qf32, v0.sf) // Normalize to HVX floating point
   v1.qf32 = vadd(v3.qf32, v1.sf) // Normalize to HVX floating point
}
{
   v2.qf32 = vmpy(v0.qf32, v1.qf32) // Multiply normalized HVX
floating point values
}
```

Using two qf32 inputs:

```
{
   v0.qf32 = some qf32 number
   v1.qf32 = some qf32 number
}
{
   v3:sf = 0x0        // IEEE-754 0
   v4.sf = 0x80000000 // IEEE-754 -0
}
{
   v3.qf32 = vmpy(v3.sf, v4.sf)     // Create -0 with exponent at
qf32 emin
}
{
```

```
   v0.qf32 = vadd(v3.qf32, v0.qf32) // Normalize to HVX floating
point
   v1.qf32 = vadd(v3.qf32, v1.qf32) // Normalize to HVX floating
point
}
{
   v2.qf32 = vmpy(v0.qf32, v1.qf32)    // Multiply normalized HVX
floating point values
}
```

6.2 HVX floating point behavior

The table below shows the characteristics of each floating point format alongside the IEEE-754 format.

HVX floating-point format with IEEE-754 format

Type	Precision	Maximum exponent	Minimum exponent
hf half precision IEEE 754	11	15	-14
qf16 single precision HVX format	11	15	-15
sf single precision IEEE 754	24	127	-126
qf32 single precision HVX format	24	255	-255

Represented values of HVX floating-point

Normal and unnormal numbers

A normal number is a value where the significand is within [-2, -1) for negative values and [1, 2) for positive values.

[1.0, 2.0) * 2^{exp} for positive values

[-2.0, -1.0) * 2^{exp} for negative values

An unnormal number is a value where the significand is out of the normal number range. An unnormal number has less than full precision compared to a normal number.

[-1.0, 1.0) * 2^{exp} for unnormal values

Unlike IEEE-754, an HVX floating-point unnormal number is not a special number. IEEE-754 unnormal, or subnormal, is a number at the minimum exponent whereas HVX floating-point unnormal can be at any exponent.

Exact and inexact numbers

An exact number is when HVX floating-point precisely represents an exact floating-point value.

An inexact number is when an exact value cannot be precisely represented. Only qf32 indicates an inexact number. Conversion from qf32 to IEEE-754 uses inexact to prevent double round errors.

All special values have an exact and inexact representation.

Sign of numbers

HVX floating-point values represent negative and positive numbers like IEEE-754. Both polarities are available for non-zero finites, zero, infinity, and Not a Number (NaN).

Special values

Zero

Zero values in HVX floating-point can have different exponent representation, ranging from the minimum exponent to the maximum exponent of the HVX floating-point exponent range.

HVX floating-point zero produces equivalent behavior to IEEE-754 zeros.

Infinity

Infinity results from a number too large to represent.

HVX floating-point infinity produces equivalent behavior to IEEE-754 infinities.

NaN

NaN describes a number that is undefined.

HVX floating-point NaN produces equivalent behavior to IEEE-754 NaN.

Rounding modes

HVX floating point only supports IEEE-754 round-to-even mode.

Round to nearest even

For any HVX floating point operation, the result rounds to the nearest value. When the result is exactly halfway between two representable values, the result rounds to the nearest even value.

For HVX floating point when a value overflows in the normal range, the exponent increments to keep the significand in [-2, 2). For IEEE-754, the exponent adjusts to keep the significand normalized.

When the exponent overflows, the result saturates to infinity

When the exponent underflows, the value results in an inexact zero. This is different from IEEE-754, where IEEE-754 gradually underflows and produces subnormals.

6.3 IEEE intrinsics

The user is encouraged to use the IEEE intrinsics for floating point operations. The IEEE intrinsics provide a standardized mechanism to operate on IEEE-754 data and eliminate the need to understand the underlying details of the HVX floating point format. The IEEE intrinsics round to the nearest even.

The compiler flags enable levels of compliance to the IEEE-754 specification. For details on the flags for IEEE-754 compliance, see the *Qualcomm Hexagon LLVM C/C++ Compiler User Guide* (80- VB419-8986).

IEEE absolute value

Single precision and half precision absolute value.

IEEE absolute value instruction intrinsics

Instruction syntax	Intrinsic
Vd.hf = vabs(Vu.hf)	HVX_Vector Q6_Vhf_vabs_Vhf(HVX_Vector Vu)
Vd.sf = vabs(Vu.sf)	HVX_Vector Q6_Vsf_vabs_Vsf(HVX_Vector Vu)

Instruction syntax	Intrinsic

IEEE addition/subtraction

Single precision and half precision addition/subtraction.

IEEE addition/subtraction instruction intrinsics

Instruction syntax	Intrinsic
Vd.hf = vadd(Vu.hf,Vv.hf)	HVX_Vector Q6_Vhf_vadd_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)
Vd.hf = vsub(Vu.hf,Vv.hf)	HVX_Vector Q6_Vhf_vsub_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)
Vdd.sf = vadd(Vu.hf,Vv.hf)	HVX_VectorPair Q6_Wsf_vadd_VbfVbf(HVX_Vector Vu, HVX_Vector Vv)
Vdd.sf = vsub(Vu.hf,Vv.hf)	HVX_VectorPair Q6_Wsf_vsub_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)
Vd.sf = vadd(Vu.sf,Vv.sf)	HVX_Vector Q6_Vsf_vadd_VsfVsf(HVX_Vector Vu, HVX_Vector Vv)
Vd.sf = vsub(Vu.sf,Vv.sf)	HVX_Vector Q6_Vsf_vsub_VsfVsf(HVX_Vector Vu, HVX_Vector Vv)

IEEE min/max/negate/copy

Min/max: IEEE compare the inputs and return the min or max value. If either operand is NaN the result is NaN.

Negate: IEEE single precision and half precision negation, only the sign bit is flipped. Copy: IEEE copy, no change in bits.

IEEE min/max/negation/move instruction intrinsics

Instruction syntax	Intrinsic
Vd.w = vfmv(Vu.w)	HVX_Vector Q6_Vw_vfmv_Vw(HVX_Vector Vu)
Vd.hf = vfmax(Vu.hf,Vv.hf)	HVX_Vector Q6_Vhf_vfmax_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)
Vd.hf = vfmin(Vu.hf,Vv.hf)	HVX_Vector Q6_Vhf_vfmin_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)
Vd.sf = vfmax(Vu.sf,Vv.sf)	HVX_Vector Q6_Vsf_vfmax_VsfVsf(HVX_Vector Vu, HVX_Vector Vv)
Vd.sf = vfmin(Vu.sf,Vv.sf)	HVX_Vector Q6_Vsf_vfmin_VsfVsf(HVX_Vector Vu, HVX_Vector Vv)
Vd.hf = vfneg(Vu.hf)	HVX_Vector Q6_Vhf_vfneg_Vhf(HVX_Vector Vu)
Vd.sf = vfneg(Vu.sf)	HVX_Vector Q6_Vsf_vfneg_Vsf(HVX_Vector Vu)

IEEE multiplication

IEEE single precision and half precision multiplication.

IEEE multiply instruction intrinsics

Instruction syntax	Intrinsic
Vd.hf= vmpy(Vu.hf,Vv.hf)	HVX_Vector Q6_Vhf_vmpy_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)
Vdd.sf = vmpy(Vu.hf,Vv.hf)	HVX_VectorPair Q6_Wsf_vmpy_VhfVhf(HVX_Vector Vu, HVX_ Vector Vv)
Vx.hf += vmpy(Vu.hf,Vv.hf)	HVX_Vector Q6_Vhf_vmpyacc_VhfVhfVhf(HVX_Vector Vx, HVX_ Vector Vu, HVX_Vector Vv)
Vxx.sf += vmpy(Vu.hf,Vv.hf)	HVX_VectorPair Q6_Wsf_vmpyacc_WsfVhfVhf(HVX_VectorPair Vxx, HVX_Vector Vu, HVX_Vector Vv)
Vd.sf = vmpy(Vu.sf,Vv.sf)	HVX_Vector Q6_Vsf_vmpy_VsfVsf(HVX_Vector Vu, HVX_Vector Vv)

IEEE fused multiplication

Half precision to single precision fused multiply reduce.

```
sf = Vu.hf[0]*Vv.hf[0] + Vu.hf[1]*Vv.hf[1]
```

The multiply operations and the addition operation round independently.

IEEE fused multiply instruction intrinsics

Instruction syntax	Intrinsic
Vd.sf = vdmpy(Vu.hf,Vv.hf)	HVX_Vector Q6_Vsf_vdmpy_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)
Vxx.sf += vdmpy(Vu.hf,Vv.hf)	HVX_Vector Q6_Vsf_vdmpyacc_VsfVhfVhf(HVX_Vector Vx, HVX_ Vector, Vu, HVX_Vector Vv)

IEEE converts

Convert IEEE single/half precision to byte/halfword/unsigned byte/unsigned half word. Convert byte/halfword/unsigned byte/unsigned halfword to IEEE single/half precision. All operations are round to nearest even.

IEEE convert instruction intrinsics

Instruction syntax	Intrinsic
Vd.b = vcvt(Vu.hf,Vv.hf)	HVX_Vector Q6_Vb_vcvt_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)
Vd.h = vcvt(Vu.hf)	HVX_Vector Q6_Vh_vcvt_Vhf(HVX_Vector Vu)

Instruction syntax	**Intrinsic**
Vd.hf = vcvt(Vu.h)	HVX_Vector Q6_Vhf_vcvt_Vh(HVX_Vector Vu)
Vd.hf = vcvt(Vu.sf,Vv.sf)	HVX_Vector Q6_Vhf_vcvt_VsfVsf(HVX_Vector Vu, HVX_Vector Vv)
Vd.hf = vcvt(Vu.uh)	HVX_Vector Q6_Vhf_vcvt_Vuh(HVX_Vector Vu)
Vd.ub = vcvt(Vu.hf,Vv.hf)	HVX_Vector Q6_Vub_vcvt_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)
Vd.uh = vcvt(Vu.hf)	HVX_Vector Q6_Vuh_vcvt_Vhf(HVX_Vector Vu)
Vdd.hf = vcvt(Vu.b)	HVX_VectorPair Q6_Whf_vcvt_Vb(HVX_Vector Vu)
Vdd.hf = vcvt(Vu.ub)	HVX_VectorPair Q6_Whf_vcvt_Vub(HVX_Vector Vu)
Vdd.sf = vcvt(Vu.hf)	HVX_VectorPair Q6_Wsf_vcvt_Vhf(HVX_Vector Vu)

7 HVX PMU events

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

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

Core PMU events describes the core PMU events.

HVX PMU events document the HVX events.

hvx v79 processor event symbols

Event	Symbol	Definition	Maskable
0x100	HVX_ACTIVE	VFIFO not empty	Not maskable
0x101	HVX_REG_ORDER	Stall cycles due to interlocks	Maskable
0x102	HVX_ACC_ORDER	Stall cycles due to accumulator not produced in previous context cycle.	Maskable
0x103	HVX_LD_L2_OUTSTANDING	Stall cycles due to load pending	Maskable
0x104	HVX_ST_L2_OUTSTANDING	Stall cycles due to store not yet allocated in L2	Maskable
0x105	HVX_VTCM_OUTSTANDING	Stall cycles due to VTCM transaction pending.	Maskable
0x106	HVX_SCATGATH_FULL	Scatter/gather: Network scoreboard not updated	Maskable
0x107	HVX_SCATGATH_IN_FULL	scatter/gather input buffer full	Maskable
0x108	HVX_ST_FULL	store buffer full	Maskable
0x10a	HVX_VOLTAGE_UNDER	Throttling: Voltage model would exceed undershoot threshold	Maskable
0x10b	HVX_POWER_OVER	Throttling: Sustained power exceeds budget	Maskable
0x10c	HVX_PKT_PARTIAL	Stall cycles due to multi-issue packet	Maskable
0x111	HVX_PKT	Increments by 2 per packet in 128-byte mode. Packets with HVX instructions	Maskable

Event	Symbol	Definition	Maskable
0x112	HVX_PKT_THREAD	Committed packets on a thread with the XE bit set, whether executed in Q6 or coprocessor	Not maskable
0x113	HVX_CORE_VFIFO_FULL_ STALL	Number of cycles a thread had to stall due to VFIFO	Not maskable
0x115	CYCLES_1_HVX_CONTEXTS_ RUNNING	Cycles 1 HVX context running	Not maskable
0x116	CYCLES_2_HVX_CONTEXTS_ RUNNING	Cycles 2 HVX contexts running concurrently	Not maskable
0x117	CYCLES_3_HVX_CONTEXTS_ RUNNING	Cycles 3 HVX contexts running concurrently	Not maskable
0x118	HVXLD_L2	L2 cacheable load access from HVX. Any load access from HVX that may cause a lookup in the L2 cache. Excludes cache ops, uncacheables, scalars	Maskable
0x119	HVXLD_L2_TCM	TCM load access for HVX. HVX load from the L2 tcm space	Maskable
0x11a	HVXLD_L2_MISS	L2 cacheable miss from HVX. Of the events qualified by 0xFB, the ones that resulted in a miss i.e. the 64-byte address was not previously allocated in the L2 tag array and data will be fetched from backing memory	Maskable
0x11b	HVXLD_L2_SECONDARY_ MISS	Of the events in 0xFB, the ones where the load could not be returned due to the immediatately prior access for the line being a pending load or pending L2Fetch	Maskable
0x11c	HVXST_L2_WR	Vector write in L2.	Maskable
0x11d	HVXST_SLD_CONFLICT	Lower priority with respect to scalar load to access L2 return data bus	Not maskable
0x11e	HVXST_VTCM_GATH_ CONFLICT	Lower priority with respect to gather	Not maskable
0x121	HVXST_L2_FULL	write fifo full.	Not maskable
0x122	HVXST_VTCM_FULL	write fifo full.	Not maskable
0x123	HVXST_L2	Vector store to L2.	Not maskable

Event	Symbol	Definition	Maskable
0x124	HVXST_L2_MISS	L2 cacheable miss from HVX store. Specifically the cases where the 128-byte-line address is not in the tag or a coalesce buffer.	Maskable
0x125	HVXST_L2TCM	TCM store access for HVX. HVX store to the L2 tcm space	Maskable
0x126	HVXST_VTCM	Vector store to VTCM.	Not maskable
0x127	HVXST_L2_SECODARY_MISS	L2 cacheable secondary miss from HVX store. Specifically the cases where the 128-byte-line address is not in the tag or a coalesce buffer.	Maskable
0x128	HVXPIPE_ALU	Executed simple ALU instruction	Not maskable
0x129	HVXPIPE_MPY	Executed multiply or abs-diff instruction	Not maskable
0x12a	HVXPIPE_SHIFT	Executed bit shift or bit count instruction	Not maskable
0x12b	HVXPIPE_PERM	Executed permute or cross-lane data movement instruction	Not maskable
0x12c	CYCLES_4_HVX_CONTEXTS_RUNNING	Cycles 4 HVX contexts running concurrently	Not maskable
0x190	HVX_VFIFO_EMPTY	Number of cycles a thread had empty VFIFO	Not maskable
0x191	CYCLES_5_HVX_CONTEXTS_RUNNING	Cycles 5 HVX contexts running concurrently	Not maskable
0x192	CYCLES_6_HVX_CONTEXTS_RUNNING	Cycles 6 HVX contexts running concurrently	Not maskable

8 HVX instruction set

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

8.1 Instruction symbols

Instruction syntax symbols lists the symbols used to specify the instruction syntax.

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

Instruction operand symbols lists the symbols used to specify instruction operands.

Symbol	Example	Meaning
#uN	R2 = #u16	Unsigned N-bit immediate value
#sN	R2 = add(R3,#s16)	Signed N-bit immediate value

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

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

The instruction behavior is specified using a superset of the C language. Instruction behavior symbols lists symbols not defined in C that are used to specify the instruction behavior.

Instruction behavior symbols

Symbol	Example	Meaning
usatN	$usat_{16}$(Rs)	Saturate a value to an unsigned N-bit
satN	sat16Rs)	Saturate a value to a signed N-bit number
sxt x->y	sxt32->64(Rs)	Sign-extend value from x to y bits
zxt x->y	zxt32->64(Rs)	Zero-extend value from x to y bits
>>>	Rss >>> offset	Logical right shift

8.2 New-value operands

VMEM store instructions can store a newly generated value from the same packet. They do not use a vector ALU resource because they neither read nor write to the register file. The new-value register specifies, through its encodings, which instruction in the packet has its destination register accessed as the new-value register. Instructions specify a new-value consumer with the `Os8.new` syntax. This 3-bit instruction field denotes the HVX instruction offset demonstrated in the table below:

HVX new-value behavior

Os8.new value	Behavior
0/1	Reserved
2/3	Producer is +1 HVX instruction ahead of consumer
4/5	Producer is +2 HVX instruction ahead of consumer
6/7	Producer is +3 HVX instruction ahead of consumer

“Ahead” is defined as the instruction encoded at a lower memory address than the consumer

instruction, not counting empty slots, constant extenders, or scalar instructions. The store is always slot 0.

For register pairs:

- bit 0: Refers to the register specified in the Vd field
- bit 1: Refers to the other register in the aligned pair

For dual independent registers:

- bit 0: Vx (0)
- bit 1: Vy (1)

Example:

```
P:000016B4|1C444305                V5.w=vadd(V3.w,V4.w);
P:000016B8|28124124                V4.cur=vmem(R18+#0x1);
P:000016BC|2822C024                vmem(R2+#0x0)=O4.new;
```

`O4.new` signifies the new registers is 2 words away (4 >> 1) to store the V5 register. A new-value consumer does consider a vector and predicate register producer as a register pair.

8.3 HVX instruction set architecture

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 4 instruction pipes. The instruction also takes 4 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 3 bits of each byte element in the 128-bit slice is used to select the 16-bit position, while the upper 5 bits select which vector register. The register file entry is then incremented by one.

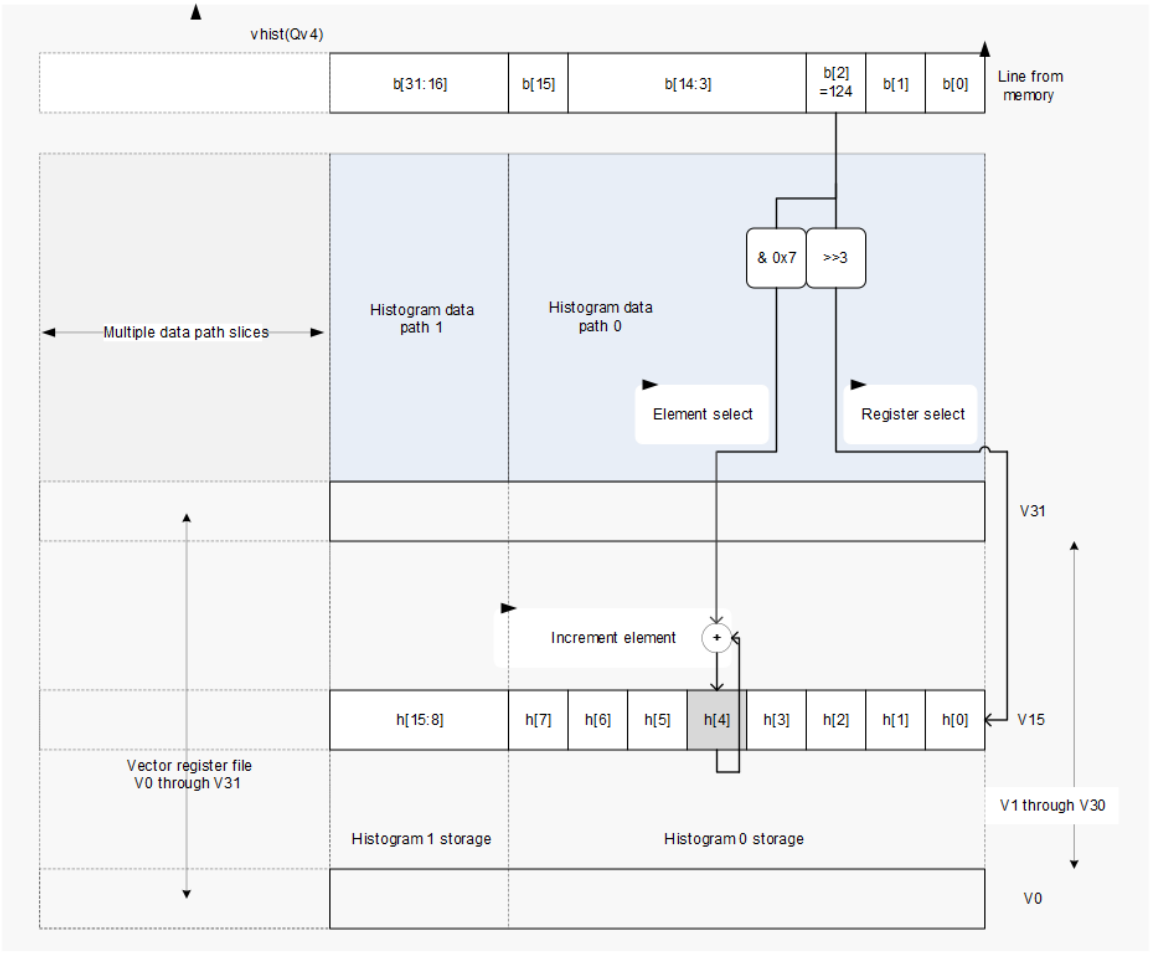
vhist is the only instruction that occupies all pipes and resources.

Before use, the vector register file must be cleared if a new histogram is to begin, otherwise the current state will be 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 which enables the selection of the elements used in the accumulation.

The following diagram shows a single 8-bit element in position 2 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 will affect 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 will affect 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.


vhist(Qv4)
b[31:16]
b[15]
b[14:3]
b[2] =124
b[1]
b[0]
Line from memory
& 0x7
>>3
Multiple data path slices
Histogram data path 1
Histogram data path 0
Element select
Register select
V31
Increment element
+
h[15:8]
h[7]
h[6]
h[5]
h[4]
h[3]
h[2]
h[1]
h[0]
V15
Vector register file V0 through V31
V1 through V30
Histogram 1 storage
Histogram 0 storage
V0

Histogram instructions

Syntax	Behavior

vhist	inputVec=Data from .tmp load;
	DECL_EXT_VREG(tmp);
	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;
	int vreg_idx = (128/16)*lane+(element);
	READ_EXT_VREG(regno,tmp,0);
	tmp.uh[vreg_idx]++;
	tmp_ext.uw[vreg_idx/8] = V_EXTENDED_WORDVAL;
	WRITE_EXT_VREG(regno,tmp,EXT_NEW);
	}
	}

Syntax	Behavior

vhist(Qv4)	`inputVec=Data from .tmp load;`
	`DECL_EXT_VREG(tmp);`
	`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;`
	`  int vreg_idx = (128/16)*lane+(element);`
	`  READ_EXT_VREG(regno,tmp,0);`
	`  if (QvV[128/8*lane+i]) {`
	`  tmp.uh[vreg_idx]++;`
	`  tmp_ext.uw[vreg_idx/8] = V_EXTENDED_WORDVAL;`
	`  }`
	`  WRITE_EXT_VREG(regno,tmp,EXT_NEW);`
	`  }`
	`}`

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

Encodings

Bitmap

Syntax	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
vhist	0	0	0	1	1	1	1	0	-	-	0	-	-	0	0	0	P	P	1	-	0	0	0	-	1	0	0	-	-	-	-	-
vhist(Qv4)	0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	0	P	P	1	-	-	0	0	-	1	0	0	-	-	-	-	-

Weighted Histogram

The vwhist instructions use all of the HVX core resources: the register file, V0-V31, and all 4 instruction pipes. The instruction also takes 4 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 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 vwhist256, the 32 vector registers are configured as multiple 128-entry histograms where each histogram bin has a width of 32 bits. Each histogram is 128 bits wide and 16 elements deep, 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 which 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 3 bits of each even byte element in the 128-bit slice is used to select the 16-bit position, while the upper 5 bits select which 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 5 bits select which 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.

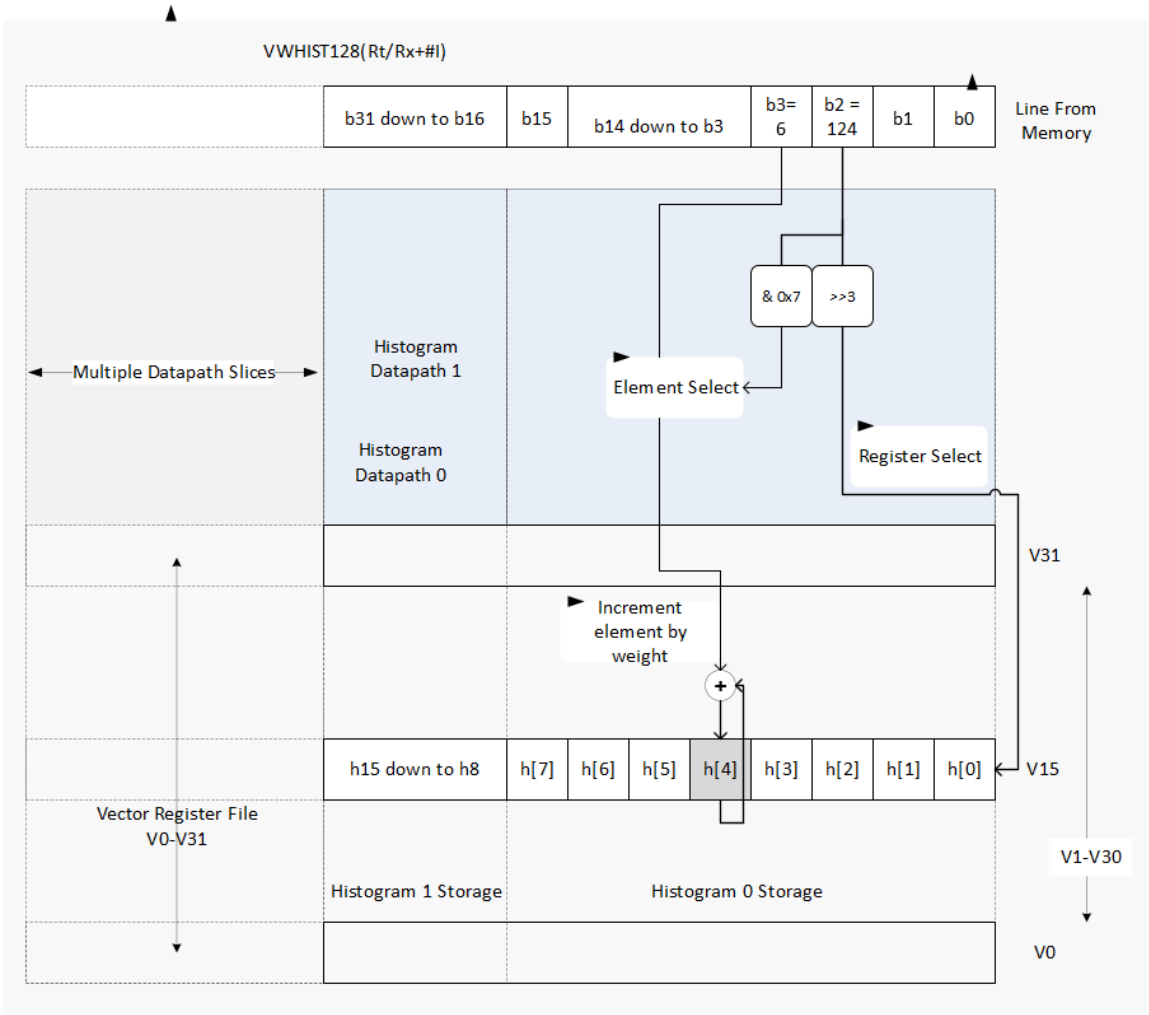
Before use, the vector register file must be cleared if a new histogram is to begin, otherwise the current state will be added to the histograms of the next data.

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

The following diagram shows a single 8-bit element in byte position 2 of the source data with corresponding weight in byte position 3. The value is 124, the register number assigned to this is 124 >> 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 will affect 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 will affect 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.



Weighted Histogram instructions

Syntax	Behavior
vwhist256	`input = Data from .tmp load;` `{` `  for (i = 0; i < VELEM(16); i++) {` `  bucket = input.h[i].ub[0];` `  weight = 0;` `  vindex = (bucket >> 3) & 0x1F;` ` elindex = ((i>>0) & (~7))	((bucket>>0) & 7);` ` DECL_EXT_VREG(tmp);` ` READ_EXT_VREG(vindex,tmp,0);` ` {` ` weight = input.h[i].ub[1];` ` }` ` {` ` tmp.uh[elindex] = (tmp.uh[elindex] + weight);` ` tmp_ext.uw[elindex/(8>>0)] = V_EXTENDED_WORDVAL;` ` }` ` 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 = 0;
        vindex = (bucket >> 3) & 0x1F;
        elindex = ((i>>0) & (~7)) |
((bucket>>0) & 7);
        DECL_EXT_VREG(tmp);
        READ_EXT_VREG(vindex,tmp,0);
        {
            weight = input.h[i].ub[1];
        }
        if (QvV[2*i]) {
            tmp.uh[elindex] = (tmp.
uh[elindex] + weight);
            tmp_ext.uw[elindex/(8>>0)] =
V_EXTENDED_WORDVAL;
        }
        WRITE_EXT_VREG(vindex,tmp,EXT_
NEW);
    }
```

Syntax	Behavior

vwhist256:sat	`input = Data from .tmp load;`
	`{`
	`  for (i = 0; i < VELEM(16); i++) {`
	`  bucket = input.h[i].ub[0];`
	`  weight = 0;`
	`  vindex = (bucket >> 3) & 0x1F;`
	`  elindex = ((i>>0) & (~7)) |`
	`((bucket>>0) & 7);`
	`  DECL_EXT_VREG(tmp);`
	`  READ_EXT_VREG(vindex,tmp,0);`
	`  {`
	`  weight = input.h[i].ub[1];`
	`  }`
	`  {`
	`  tmp.uh[elindex] = usat_`
	`16(tmp.uh[elindex] + weight);`
	`  tmp_ext.uw[elindex/(8>>0)] =`
	`V_EXTENDED_WORDVAL;`
	`  }`
	`  WRITE_EXT_VREG(vindex,tmp,EXT_`
	`NEW);`
	`  }`

Syntax	Behavior

vwhist256(Qv4):sat	input = Data from .tmp load;
	{
	for (i = 0; i < VELEM(16); i++) {
	bucket = input.h[i].ub[0];
	weight = 0;
	vindex = (bucket >> 3) & 0x1F;
	elindex = ((i>>0) & (~7)) |
	((bucket>>0) & 7);
	DECL_EXT_VREG(tmp);
	READ_EXT_VREG(vindex,tmp,0);
	{
	weight = input.h[i].ub[1];
	}
	if (QvV[2*i]) {
	tmp.uh[elindex] = usat_
	16(tmp.uh[elindex] + weight);
	tmp_ext.uw[elindex/(8>>0)] =
	V_EXTENDED_WORDVAL;
	}
	WRITE_EXT_VREG(vindex,tmp,EXT_
	NEW);
	}

Syntax	Behavior
vwhist128	

```
input = Data from .tmp load;
{
    for (i = 0; i < VELEM(16); i++) {
        bucket = input.h[i].ub[0];
        weight = 0;
        vindex = (bucket >> 3) & 0x1F;
        elindex = ((i>>1) & (~3)) |
((bucket>>1) & 3);
        DECL_EXT_VREG(tmp);
        READ_EXT_VREG(vindex,tmp,0);
        {
            weight = input.h[i].ub[1];
        }
        {
            tmp.uw[elindex] = (tmp.
uw[elindex] + weight);
            tmp_ext.uw[elindex/(8>>1)] =
V_EXTENDED_WORDVAL;
        }
        WRITE_EXT_VREG(vindex,tmp,EXT_
NEW);
    }
```

Syntax	Behavior
vwhist128(Qv4)	

```
input = Data from .tmp load;
{
    for (i = 0; i < VELEM(16); i++) {
        bucket = input.h[i].ub[0];
        weight = 0;
        vindex = (bucket >> 3) & 0x1F;
        elindex = ((i>>1) & (~3)) |
((bucket>>1) & 3);
        DECL_EXT_VREG(tmp);
        READ_EXT_VREG(vindex,tmp,0);
        {
            weight = input.h[i].ub[1];
        }
        if (QvV[2*i]) {
            tmp.uw[elindex] = (tmp.
uw[elindex] + weight);
            tmp_ext.uw[elindex/(8>>1)] =
V_EXTENDED_WORDVAL;
        }
        WRITE_EXT_VREG(vindex,tmp,EXT_
NEW);
    }
```

Syntax	Behavior

vwhist128(#u1)	input = Data from .tmp load;
	{
	for (i = 0; i < VELEM(16); i++) {
	bucket = input.h[i].ub[0];
	weight = 0;
	vindex = (bucket >> 3) & 0x1F;
	elindex = ((i>>1) & (~3)) |
	((bucket>>1) & 3);
	DECL_EXT_VREG(tmp);
	READ_EXT_VREG(vindex,tmp,0);
	if ((bucket & 1) == u) {
	weight = input.h[i].ub[1];
	}
	{
	tmp.uw[elindex] = (tmp.
	uw[elindex] + weight);
	tmp_ext.uw[elindex/(8>>1)] =
	V_EXTENDED_WORDVAL;
	}
	WRITE_EXT_VREG(vindex,tmp,EXT_
	NEW);
	}

Syntax	Behavior

vwhist128(Qv4,#u1)	input = Data from .tmp load;
	{
	for (i = 0; i < VELEM(16); i++) {
	bucket = input.h[i].ub[0];
	weight = 0;
	vindex = (bucket >> 3) & 0x1F;
	elindex = ((i>>1) & (~3)) |
	((bucket>>1) & 3);
	DECL_EXT_VREG(tmp);
	READ_EXT_VREG(vindex,tmp,0);
	if ((bucket & 1) == u) {
	weight = input.h[i].ub[1];
	}
	if (QvV[2*i]) {
	tmp.uw[elindex] = (tmp.
	uw[elindex] + weight);
	tmp_ext.uw[elindex/(8>>1)] =
	V_EXTENDED_WORDVAL;
	}
	WRITE_EXT_VREG(vindex,tmp,EXT_
	NEW);
	}

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

Syntax	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

Encodings

Bitmap

Syntax	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
vwhist256	0	0	0	1	1	1	1	0	-	-	0	-	-	0	0	0	P	P	1	-	0	0	1	0	1	0	0	-	-	-	-	-
vwhist256(Qv4)	0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	0	P	P	1	-	-	0	1	0	1	0	0	-	-	-	-	-
vwhist256:sat	0	0	0	1	1	1	1	0	-	-	0	-	-	0	0	0	P	P	1	-	0	0	1	1	1	0	0	-	-	-	-	-
vwhist256(Qv4):sat	0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	0	P	P	1	-	-	0	1	1	1	0	0	-	-	-	-	-
vwhist128	0	0	0	1	1	1	1	0	-	-	0	-	-	0	0	0	P	P	1	-	0	1	0	-	1	0	0	-	-	-	-	-
vwhist128(Qv4)	0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	0	P	P	1	-	-	1	0	-	1	0	0	-	-	-	-	-
vwhist128(#u1)	0	0	0	1	1	1	1	0	-	-	0	-	-	0	0	0	P	P	1	-	0	1	1	i	1	0	0	-	-	-	-	-
vwhist128(Qv4,#u1)	0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	0	P	P	1	-	-	1	1	i	1	0	0	-	-	-	-	-

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.

Predicate operations instructions

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

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

Note:

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

Encodings

Bitmap

Syntax	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
Qd4.h=vshuffe(Qs4.w,Qt4.w)	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.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	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	0	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	0	0	0	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	0	1	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	1	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	1	0	1	d	d

Intrinsics

Predicate operations intrinsics

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

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.

Combine instructions

Syntax	Behavior
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; }`
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; }`
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]; }`

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

Note:

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

Encodings

Bitmap

Syntax	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
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)	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
Vdd=vcombine(Vu,Vv)	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

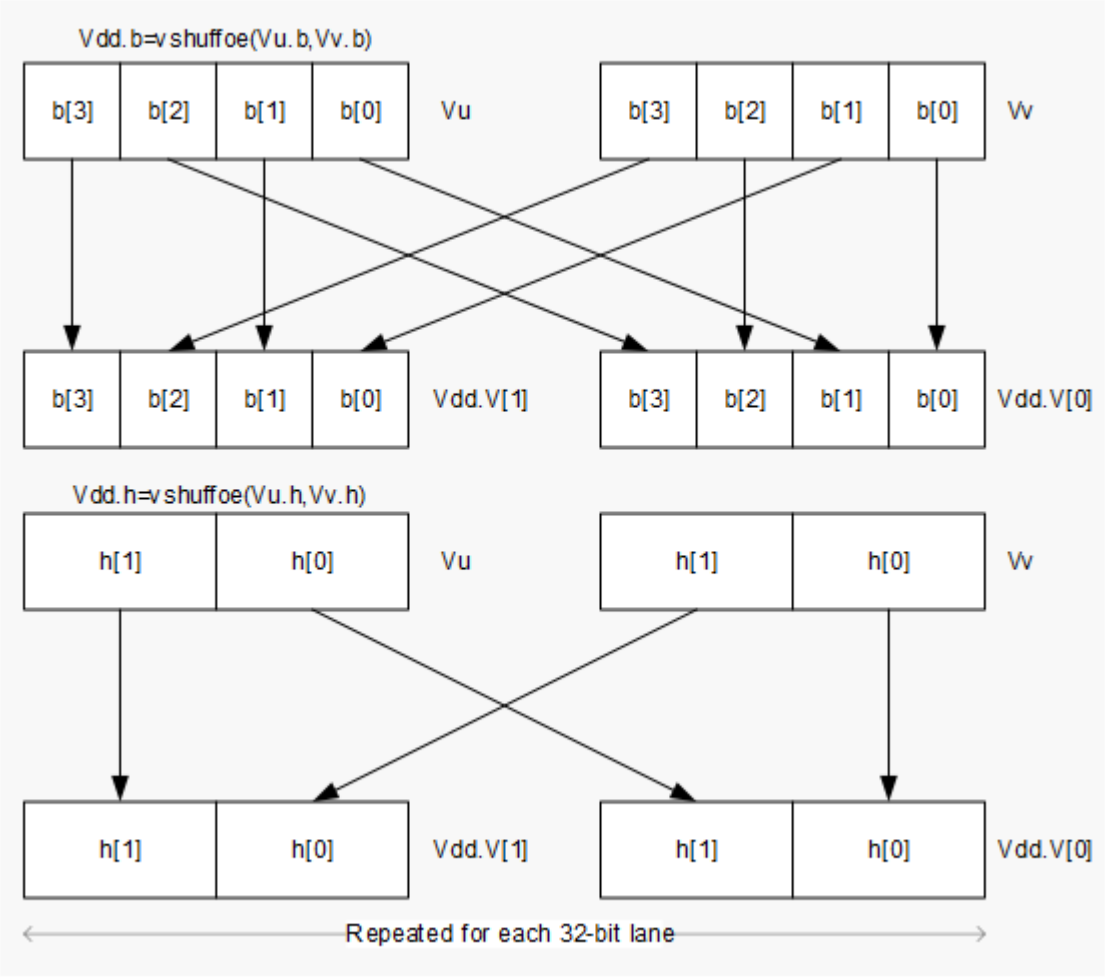
Intrinsics

Combine intrinsics

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

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.

In-lane shuffle instructions

Syntax	Behavior
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]; }`
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]; }`

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

Note:

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

Encodings

Bitmap

Syntax	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
Vdd.h=vshuffoe(Vu.h,Vv.h)	0	0	0	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.b=vshuffoe(Vu.b,Vv.b)	0	0	0	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

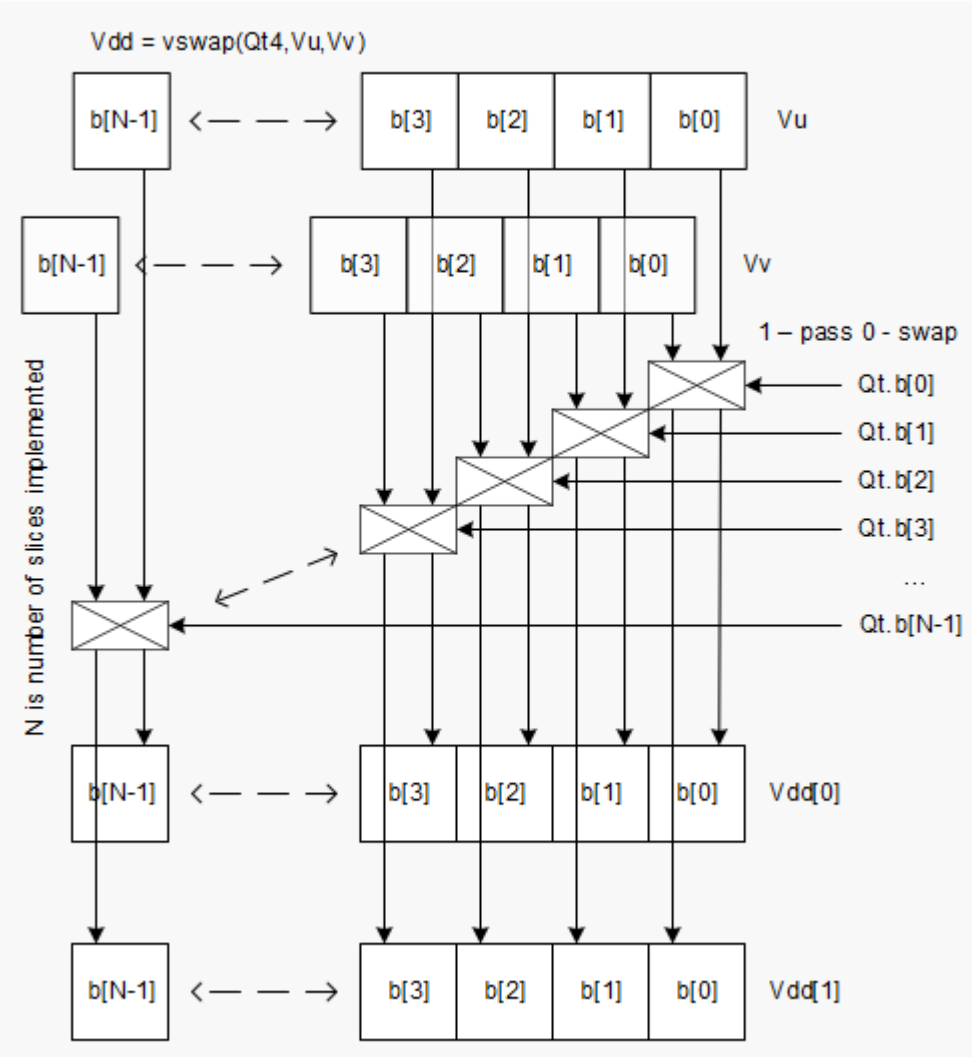
Intrinsics

In-lane shuffle intrinsics

Vdd.h=vshuffoe(Vu.h,Vv.h)	HVX_VectorPair Q6_Wh_vshuffoe_VhVh(HVX_Vector Vu, HVX_Vector Vv)
Vdd.b=vshuffoe(Vu.b,Vv.b)	HVX_VectorPair Q6_Wb_vshuffoe_VbVb(HVX_Vector Vu, HVX_Vector Vv)

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.



Swap instructions

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: HVX (slots 0,1,2,3)

Note:

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

Encodings

Bitmap

Syntax	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
Vdd=vswap(Qt4,Vu,Vv)	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

Intrinsics

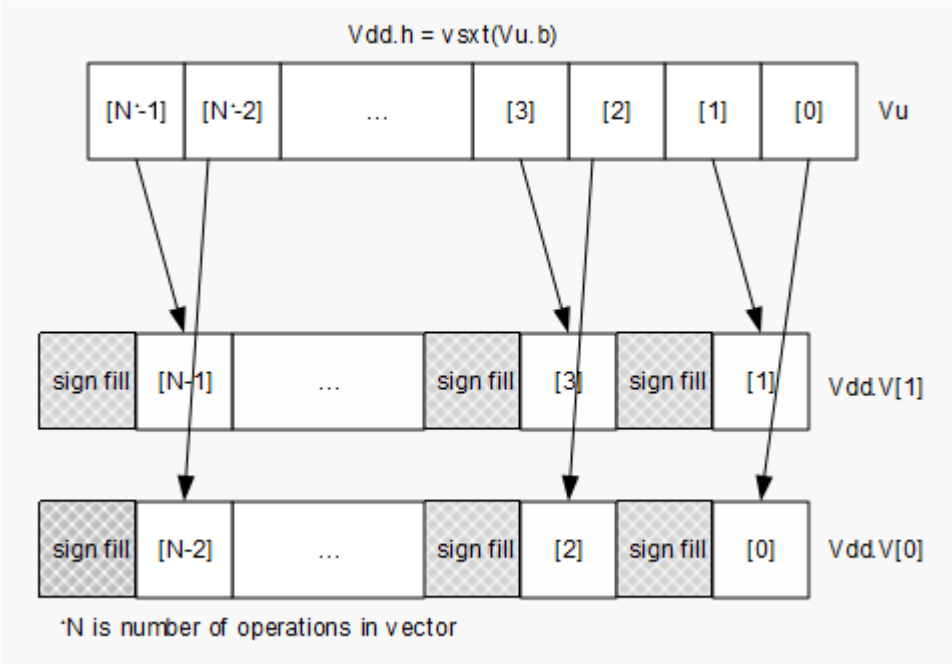
Swap intrinsics

Vdd=vswap(Qt4,Vu,Vv)	HVX_VectorPair Q6_W_vswap_QVV(HVX_VectorPred Qt, HVX_Vector Vu, HVX_Vector Vv)

Sign/Zero extension

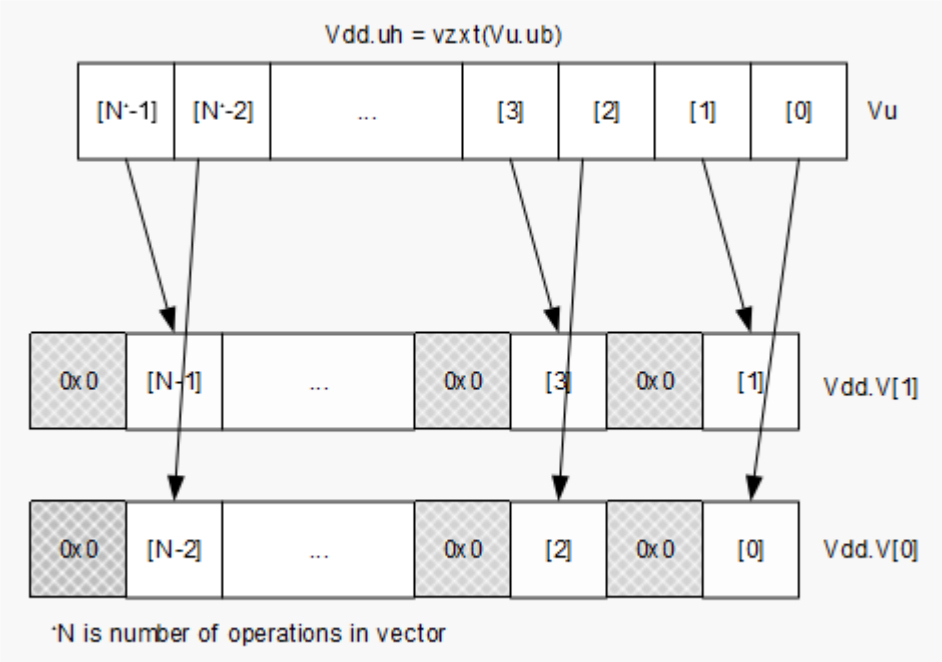
Perform sign extension on each even element in Vu, and place it in the even destination vector register Vdd[0]. Odd elements are sign-extended and placed in the odd destination vector register Vdd[1]. Bytes are converted to halfwords, and halfwords are converted to words.

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.



Sign/Zero extension instructions

Syntax	Behavior

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.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.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];`
	`}`

Syntax	Behavior

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];`
	`}`

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

Note:

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

Encodings

Bitmap

Syntax	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
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	0	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	0	1	1	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	0	d	d	d	d	d
Vdd.w=vsxt(Vu.h)	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

Intrinsics

Sign/Zero extension intrinsics

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

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.

Arithmetic instructions

Syntax	Behavior
Vdd.b=vadd(Vuu.b,Vvv.b)	`for (i = 0; i < VELEM(8); i++) { Vdd.v[0].b[i] = Vuu.v[0].b[i] + Vvv.v[0].b[i]; Vdd.v[1].b[i] = Vuu.v[1].b[i] + Vvv.v[1].b[i]; }`
Vdd.b=vsub(Vuu.b,Vvv.b)	`for (i = 0; i < VELEM(8); i++) { Vdd.v[0].b[i] = Vuu.v[0].b[i] - Vvv.v[0].b[i]; Vdd.v[1].b[i] = Vuu.v[1].b[i] - Vvv.v[1].b[i]; }`
Vdd.h=vadd(Vuu.h,Vvv.h)	`for (i = 0; i < VELEM(16); i++) { Vdd.v[0].h[i] = Vuu.v[0].h[i] + Vvv.v[0].h[i]; Vdd.v[1].h[i] = Vuu.v[1].h[i] + Vvv.v[1].h[i]; }`
Vdd.h=vsub(Vuu.h,Vvv.h)	`for (i = 0; i < VELEM(16); i++) { Vdd.v[0].h[i] = Vuu.v[0].h[i] - Vvv.v[0].h[i]; Vdd.v[1].h[i] = Vuu.v[1].h[i] - Vvv.v[1].h[i]; }`

Syntax	Behavior
Vdd.w=vadd(Vuu.w,Vvv.w)	`for (i = 0; i < VELEM(32); i++) { Vdd.v[0].w[i] = Vuu.v[0].w[i] + Vvv.v[0].w[i]; Vdd.v[1].w[i] = Vuu.v[1].w[i] + Vvv.v[1].w[i]; }`
Vdd.w=vsub(Vuu.w,Vvv.w)	`for (i = 0; i < VELEM(32); i++) { Vdd.v[0].w[i] = Vuu.v[0].w[i] - Vvv.v[0].w[i]; Vdd.v[1].w[i] = Vuu.v[1].w[i] - Vvv.v[1].w[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.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]); }`

Syntax	Behavior
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.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: HVX (slots 0,1,2,3)

Note:

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

Encodings

Bitmap

Syntax	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
Vdd.b=vadd(Vuu.b,Vvv.b)	0	u	u	0	b	0	v	v	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=vsub(Vuu.b,Vvv.b)	0	u	u	0	b	0	v	v	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.h=vadd(Vuu.h,Vvv.h)	0	u	u	0	h	0	v	v	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=vsub(Vuu.h,Vvv.h)	0	u	u	0	h	0	v	v	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.w=vadd(Vuu.w,Vvv.w)	0	u	u	0	w	0	v	v	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=vsub(Vuu.w,Vvv.w)	0	u	u	0	w	0	v	v	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.ub=vadd(Vuu.ub,Vvv.ub):sat	0	u	u	0	b	1	v	v	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=vsub(Vuu.ub,Vvv.ub):sat	0	u	u	0	b	1	v	v	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.uh=vadd(Vuu.uh,Vvv.uh):sat	0	u	u	0	h	1	v	v	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
Vdd.uh=vsub(Vuu.uh,Vvv.uh):sat	0	u	u	0	h	1	v	v	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.uw=vadd(Vuu.uw,Vvv.uw):sat	0	u	u	0	w	1	v	v	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
Vdd.uw=vsub(Vuu.uw,Vvv.uw):sat	0	u	u	0	w	1	v	v	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.b=vadd(Vuu.b,Vvv.b):sat	1	u	u	0	b	0	v	v	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
Vdd.b=vsub(Vuu.b,Vvv.b):sat	1	u	u	0	b	0	v	v	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.h=vadd(Vuu.h,Vvv.h):sat	1	u	u	0	h	0	v	v	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
Vdd.h=vsub(Vuu.h,Vvv.h):sat	1	u	u	0	h	0	v	v	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.w=vadd(Vuu.w,Vvv.w):sat	1	u	u	0	w	0	v	v	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
Vdd.w=vsub(Vuu.w,Vvv.w):sat	1	u	u	0	w	0	v	v	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

Intrinsics

Arithmetic intrinsics

Vdd.b=vadd(Vuu.b,Vvv.b)	HVX_VectorPair Q6_Wb_vadd_WbWb(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.h=vadd(Vuu.h,Vvv.h)	HVX_VectorPair Q6_Wh_vadd_WhWh(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.w=vadd(Vuu.w,Vvv.w)	HVX_VectorPair Q6_Ww_vadd_WwWw(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.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.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):sat	HVX_VectorPair Q6_Wb_vsub_WbWb_sat(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):sat	HVX_VectorPair Q6_Wh_vsub_WhWh_sat(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):sat	HVX_VectorPair Q6_Ww_vsub_WwWw_sat(HVX_VectorPair Vuu, HVX_VectorPair Vvv)

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.

Predicate operations instructions

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

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

Note:

- This instruction can use any HVX resource.

Encodings

Bitmap

Syntax	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
Qd4=not(Qs4)	0	0	0	1	1	1	1	0	-	-	0	-	-	-	1	1	P	P	0	-	-	-	s	s	0	0	0	0	1	0	d	d

Intrinsics

Predicate operations intrinsics

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

Byte-conditinal vector assign

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

Byte-conditinal vector assign instructions

Syntax	Behavior

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

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

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

Note:

- This instruction can use any HVX resource.

Encodings

Bitmap

Syntax	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
Vd=vand(Qv4,Vu)	0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	1	P	P	1	u	u	u	u	u	0	0	0	d	d	d	d	d
Vd=vand(!Qv4,Vu)	0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	1	P	P	1	u	u	u	u	u	0	0	1	d	d	d	d	d

Intrinsics

Byte-conditinal vector assign intrinsics

Vd=vand(Qv4,Vu)	HVX_Vector Q6_V_vand_QV(HVX_VectorPred Qv, HVX_Vector Vu)
Vd=vand(!Qv4,Vu)	HVX_Vector Q6_V_vand_QnV(HVX_VectorPred Qv, HVX_Vector Vu)

Syntax	Behavior

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.

Min/max instructions

Syntax	Behavior
Vd.sf=vmax(Vu.sf,Vv.sf)	`for (i = 0; i < VELEM(32); i++) { Vd.sf[i] = max(Vu.sf[i],Vv.sf[i]); }`
Vd.sf=vmin(Vu.sf,Vv.sf)	`for (i = 0; i < VELEM(32); i++) { Vd.sf[i] = min(Vu.sf[i],Vv.sf[i]); }`
Vd.hf=vmax(Vu.hf,Vv.hf)	`for (i = 0; i < VELEM(16); i++) { Vd.hf[i] = max(Vu.hf[i],Vv.hf[i]); }`
Vd.hf=vmin(Vu.hf,Vv.hf)	`for (i = 0; i < VELEM(16); i++) { Vd.hf[i] = min(Vu.hf[i],Vv.hf[i]); }`
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]; }`

Syntax	Behavior
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.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.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]; }`

Syntax	Behavior
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: HVX (slots 0,1,2,3)

Note:

- This instruction can use any HVX resource.

Encodings

Bitmap

Syntax	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
Vd.sf=vmax(Vu.sf,Vv.sf)	1	1	1	1	1	1	1	1	1	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	d	d	d	d	d
Vd.sf=vmin(Vu.sf,Vv.sf)	1	1	1	1	1	1	1	1	1	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	0	d	d	d	d	d
Vd.hf=vmax(Vu.hf,Vv.hf)	1	1	1	1	1	1	1	1	1	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	d	d	d	d	d
Vd.hf=vmin(Vu.hf,Vv.hf)	1	1	1	1	1	1	1	1	1	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	d	d	d	d	d
Vd.b=vmax(Vu.b,Vv.b)	1	1	1	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=vmin(Vu.b,Vv.b)	1	1	1	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.ub=vmax(Vu.ub,Vv.ub)	1	1	1	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=vmin(Vu.ub,Vv.ub)	1	1	1	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.uh=vmax(Vu.uh,Vv.uh)	1	1	1	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=vmin(Vu.uh,Vv.uh)	1	1	1	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.h=vmax(Vu.h,Vv.h)	1	1	1	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=vmin(Vu.h,Vv.h)	1	1	1	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.w=vmax(Vu.w,Vv.w)	1	1	1	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=vmin(Vu.w,Vv.w)	1	1	1	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

Intrinsics

Min/max intrinsics

Vd.sf=vmax(Vu.sf,Vv.sf)	HVX_Vector Q6_Vsf_vmax_VsfVsf(HVX_Vector Vu, HVX_Vector Vv)
Vd.sf=vmin(Vu.sf,Vv.sf)	HVX_Vector Q6_Vsf_vmin_VsfVsf(HVX_Vector Vu, HVX_Vector Vv)
Vd.hf=vmax(Vu.hf,Vv.hf)	HVX_Vector Q6_Vhf_vmax_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)
Vd.hf=vmin(Vu.hf,Vv.hf)	HVX_Vector Q6_Vhf_vmin_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)
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.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.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.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)

Absolute value

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

Absolute value instructions

Syntax	Behavior
Vd.b=vabs(Vu.b)	`for (i = 0; i < VELEM(8); i++) { Vd.b[i] = ABS(Vu.b[i]); }`
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)	`for (i = 0; i < VELEM(16); i++) { Vd.h[i] = ABS(Vu.h[i]); }`
Vd.h=vabs(Vu.h):sat	`for (i = 0; i < VELEM(16); i++) { Vd.h[i] = sat_16(ABS(Vu.h[i])); }`
Vd.w=vabs(Vu.w)	`for (i = 0; i < VELEM(32); i++) { Vd.w[i] = ABS(Vu.w[i]); }`
Vd.w=vabs(Vu.w):sat	`for (i = 0; i < VELEM(32); i++) { Vd.w[i] = sat_32(ABS(Vu.w[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)"`

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

Note:

- This instruction can use any HVX resource.

Encodings

Bitmap

Syntax	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
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	0	d	d	d	d	d
Vd.b=vabs(Vu.b):sat	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.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	0	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	0	1	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	0	d	d	d	d	d
Vd.w=vabs(Vu.w):sat	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

Intrinsics

Absolute value 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)

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.

Arithmetic instructions

Syntax	Behavior
Vd.b=vadd(Vu.b,Vv.b)	`for (i = 0; i < VELEM(8); i++) { Vd.b[i] = Vu.b[i] + Vv.b[i]; }`
Vd.b=vsub(Vu.b,Vv.b)	`for (i = 0; i < VELEM(8); i++) { Vd.b[i] = Vu.b[i] - Vv.b[i]; }`
Vd.h=vadd(Vu.h,Vv.h)	`for (i = 0; i < VELEM(16); i++) { Vd.h[i] = Vu.h[i] + Vv.h[i]; }`
Vd.h=vsub(Vu.h,Vv.h)	`for (i = 0; i < VELEM(16); i++) { Vd.h[i] = Vu.h[i] - Vv.h[i]; }`
Vd.w=vadd(Vu.w,Vv.w)	`for (i = 0; i < VELEM(32); i++) { Vd.w[i] = Vu.w[i] + Vv.w[i]; }`
Vd.w=vsub(Vu.w,Vv.w)	`for (i = 0; i < VELEM(32); i++) { Vd.w[i] = Vu.w[i] - Vv.w[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.ub):sat	`for (i = 0; i < VELEM(8); i++) { Vd.ub[i] = usat_8(Vu.ub[i]-Vv.ub[i]); }`

Syntax	Behavior
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]); }`
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.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]); }`

Syntax	Behavior
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.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=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.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=vsub(Vu.ub,Vv.b):sat	`for (i = 0; i < VELEM(8); i++) { Vd.ub[i] = usat_8(Vu.ub[i] - Vv. b[i]); }`

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

Note:

- This instruction can use any HVX resource.

Encodings

Bitmap

Syntax	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
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	0	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	0	1	d	d	d	d	d

Syntax	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
Vd.h=vadd(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	1	1	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	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	0	d	d	d	d	d
Vd.w=vsub(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	1	1	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	0	1	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	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	0	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	0	1	d	d	d	d	d
Vd.uw=vadd(Vu.uw,Vv.uw):sat	0	0	0	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=vsub(Vu.uw,Vv.uw):sat	0	0	0	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.b=vadd(Vu.b,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=vsub(Vu.b,Vv.b):sat	0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d
Vd.h=vadd(Vu.h,Vv.h):sat	0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d
Vd.h=vsub(Vu.h,Vv.h):sat	0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d
Vd.w=vadd(Vu.w,Vv.w):sat	0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	0	0	d	d	d	d	d
Vd.w=vsub(Vu.w,Vv.w):sat	0	0	0	1	1	1	1	1	0	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d
Vd.ub=vadd(Vu.ub,Vv.ub):sat	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.ub=vsub(Vu.ub,Vv.ub):sat	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

Intrinsics

Arithmetic intrinsics

Vd.b=vadd(Vu.b,Vv.b)	HVX_Vector Q6_Vb_vadd_VbVb(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.h=vadd(Vu.h,Vv.h)	HVX_Vector Q6_Vh_vadd_VhVh(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.w=vadd(Vu.w,Vv.w)	HVX_Vector Q6_Vw_vadd_VwVw(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.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.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.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):sat	HVX_Vector Q6_Vb_vsub_VbVb_sat(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):sat	HVX_Vector Q6_Vh_vsub_VhVh_sat(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):sat	HVX_Vector Q6_Vw_vsub_VwVw_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=vsub(Vu.ub,Vv.b):sat	HVX_Vector Q6_Vub_vsub_VubVb_sat(HVX_Vector Vu, HVX_Vector Vv)

Syntax	Behavior

Arithmetic with carry bit

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

Optionally saturate for word.

Arithmetic with carry bit instructions

Syntax	Behavior
Vd.w=vadd(Vu.w,Vv.w,Qs4):carry:sat	`for (i = 0; i < VELEM(32); i++) { Vd.w[i] = sat_32(Vu.w[i]+Vv.w[i]+QsV[i*4]); }`
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,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,Qe4=vadd(Vu.w,Vv.w):carry	`for (i = 0; i < VELEM(32); i++) { Vd.w[i] = Vu.w[i]+Vv.w[i]; QeV[4*i+4-1:4*i] = -carry_from(Vu.w[i],Vv.w[i],0); }`

Syntax	Behavior
Vd.w,Qe4=vsub(Vu.w,Vv.w):carry	`for (i = 0; i < VELEM(32); i++) { Vd.w[i] = Vu.w[i]+~Vv.w[i]+1; QeV[4*i+4-1:4*i] = -carry_from(Vu.w[i],~Vv.w[i],1); }`

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

Note:

- This instruction can use any HVX resource.

Encodings

Bitmap

Syntax	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
Vd.w=vadd(Vu.w,Vv.w,Qs4):carry:sat	0	0	0	1	1	0	0	1	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	s	s	d	d	d	d	d
Vd.w=vadd(Vu.w,Vv.w,Qx4):carry	0	0	0	1	1	0	0	1	0	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	x	x	d	d	d	d	d
Vd.w=vsub(Vu.w,Vv.w,Qx4):carry	0	0	0	1	1	0	0	1	0	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	x	x	d	d	d	d	d
Vd.w,Qe4=vadd(Vu.w,Vv.w):carry	0	0	0	1	1	0	0	1	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	e	e	d	d	d	d	d
Vd.w,Qe4=vsub(Vu.w,Vv.w):carry	0	0	0	1	1	0	0	1	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	e	e	d	d	d	d	d

Intrinsics

Arithmetic with carry bit intrinsics

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

Logical operations

Perform bitwise logical operations (and, or, xor) between the two vector registers. In the case of vnot, simply invert the input register.

Logical operations instructions

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

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

Note:

- This instruction can use any HVX resource.

Syntax	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

Encodings

Bitmap

Syntax	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
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	0	1	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	0	d	d	d	d	d
Vd=vxor(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=vnot(Vu)	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

Intrinsics

Logical operations intrinsics

Vd=vand(Vu,Vv)	HVX_Vector Q6_V_vand_VV(HVX_Vector Vu, HVX_Vector Vv)
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)
Vd=vnot(Vu)	HVX_Vector Q6_V_vnot_V(HVX_Vector Vu)

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.

Copy instructions

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

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

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

Note:

- This instruction can use any HVX resource.

Encodings

Bitmap

Syntax	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
if (Ps) Vd=Vu	0	0	0	1	1	0	1	0	0	0	0	-	-	-	-	-	P	P	0	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
Vd=Vu	0	0	0	1	1	1	1	0	-	-	0	-	-	0	1	1	P	P	1	u	u	u	u	u	1	1	1	d	d	d	d	d

Intrinsics

Copy intrinsics

Vd=Vu	HVX_Vector Q6_V_equals_V(HVX_Vector Vu)

Temporary Assignment

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

Temporary Assignment instructions

Syntax	Behavior
Vdd.tmp=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]; } for (i = 0; i < VELEM(32); i++) { Vdd.v[0].ext[i]=Vv.ext[i]; Vdd.v[1].ext[i]=Vu.ext[i]; }`
Vd.tmp=Vu	`for (i = 0; i < VELEM(32); i++) { Vd.w[i]=Vu.w[i]; Vd.ext[i]=Vu.ext[i]; }`

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

Encodings

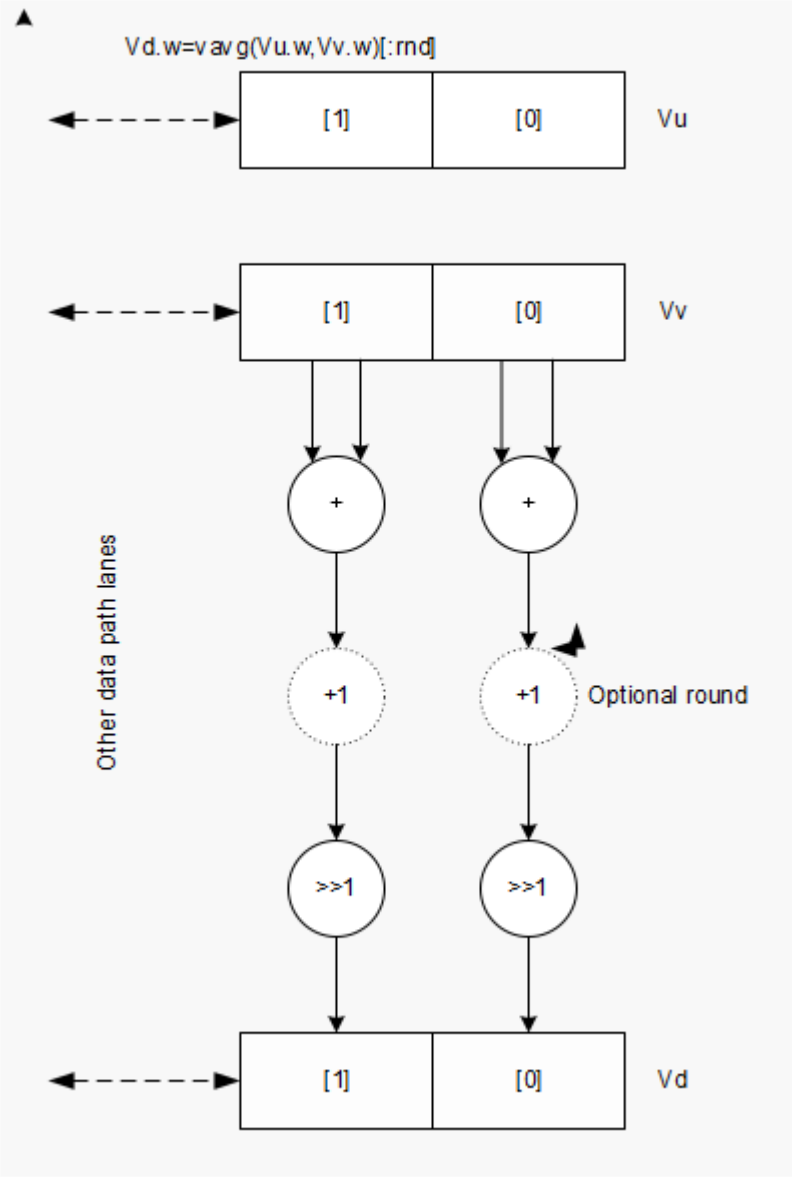
Bitmap

Syntax	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
Vdd.tmp=vcombine(Vu,Vv)	0	0	0	1	1	1	1	0	1	0	1	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d
Vd.tmp=Vu	0	0	0	1	1	1	1	0	-	-	0	-	-	-	0	1	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d

Average

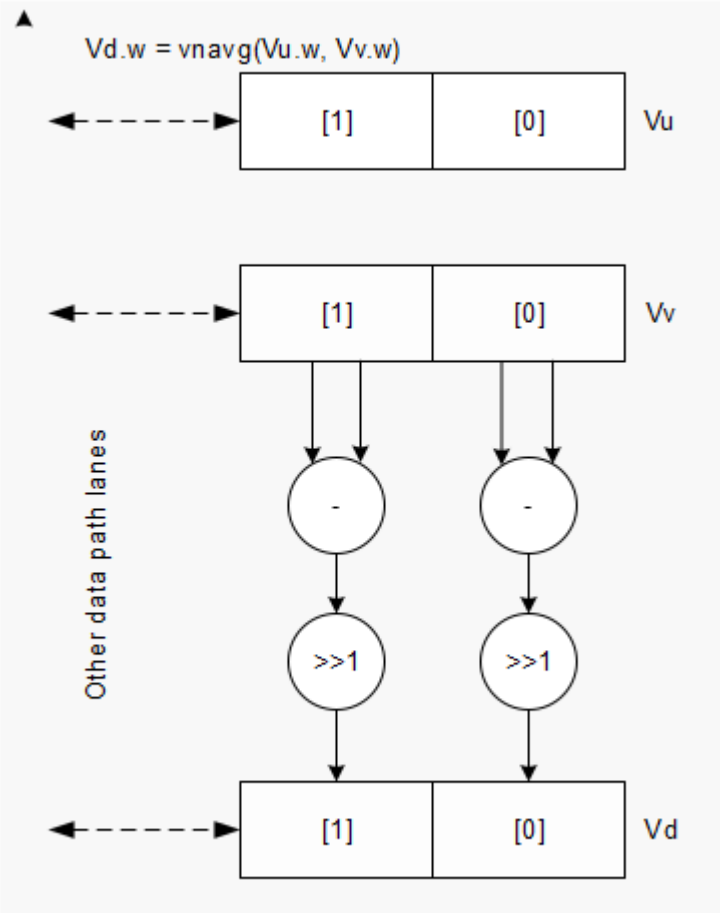
Add the elements of Vu to the respective elements of Vv, and shift the results right by 1 bit. The intermediate precision of the sum is larger than the input data precision. Optionally, a rounding constant 0x1 is added before shifting.

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



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

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



Average instructions

Syntax	Behavior
Vd.ub=vavg(Vu.ub,Vv.ub)	`for (i = 0; i < VELEM(8); i++) { Vd.ub[i] = (Vu.ub[i]+Vv.ub[i])/2; }`
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)	`for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = (Vu.uh[i]+Vv.uh[i])/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)	`for (i = 0; i < VELEM(32); i++) { Vd.uw[i] = (Vu.uw[i]+Vv.uw[i])/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.b=vavg(Vu.b,Vv.b)	`for (i = 0; i < VELEM(8); i++) { Vd.b[i] = (Vu.b[i]+Vv.b[i])/2; }`
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; }`

Syntax	Behavior
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.h=vavg(Vu.h,Vv.h)	`for (i = 0; i < VELEM(16); i++) { Vd.h[i] = (Vu.h[i]+Vv.h[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; }`
Vd.w=vavg(Vu.w,Vv.w)	`for (i = 0; i < VELEM(32); i++) { Vd.w[i] = (Vu.w[i]+Vv.w[i])/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; }`
Vd.b=vnavg(Vu.ub,Vv.ub)	`for (i = 0; i < VELEM(8); i++) { Vd.b[i] = (Vu.ub[i]-Vv.ub[i])/2; }`

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

Note:

- This instruction can use any HVX resource.

Encodings

Bitmap

Syntax	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
Vd.ub=vavg(Vu.ub,Vv.ub)	0	u	u	0	v	v	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):rnd	0	u	u	0	v	v	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.uh=vavg(Vu.uh,Vv.uh)	0	u	u	0	v	v	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):rnd	0	u	u	0	v	v	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.uw=vavg(Vu.uw,Vv.uw)	0	u	u	0	v	v	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):rnd	0	u	u	0	v	v	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.b=vavg(Vu.b,Vv.b)	0	u	u	0	v	v	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):rnd	0	u	u	0	v	v	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=vnavg(Vu.b,Vv.b)	0	u	u	0	v	v	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.h=vavg(Vu.h,Vv.h)	0	u	u	0	v	v	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.h=vavg(Vu.h,Vv.h):rnd	0	u	u	0	v	v	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.h=vnavg(Vu.h,Vv.h)	0	u	u	0	v	v	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.w=vavg(Vu.w,Vv.w)	0	u	u	0	v	v	1	1	1	1	1	0	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):rnd	0	u	u	0	v	v	1	1	1	1	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	1	1	0	d	d	d	d	d
Vd.w=vnavg(Vu.w,Vv.w)	0	u	u	0	v	v	1	1	1	1	1	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d	d
Vd.b=vnavg(Vu.ub,Vv.ub)	0	u	u	0	v	v	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

Intrinsics

Average intrinsics

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.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.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.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)
Vd.b=vnavg(Vu.ub,Vv.ub)	HVX_Vector Q6_Vb_vnavg_VubVub(HVX_Vector Vu, HVX_Vector Vv)

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 1 bit, halfwords 2 bits, and words 4 bits.

Optionally supports xor(^) with the destination, and(&) with the destination, and or(|) with the destination.

Compare vectors instructions

Syntax	Behavior
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.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.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.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); }`

Syntax	Behavior
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); }`
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.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.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); }`
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); }`
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.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); }`

Syntax	Behavior
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); }`
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); }`
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.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.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); }`
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); }`
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); }`

Syntax	Behavior
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.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); }`
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); }`
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.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.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); }`
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); }`

Syntax	Behavior
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.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.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); }`
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); }`
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.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.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); }`

Syntax	Behavior
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); }`
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.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.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); }`
Qd4=vcmp.eq(Vu.uw,Vv.uw)	`Assembler mapped to: "Qd4=vcmp.eq(Vu." "w" ",Vv." "w" ")"`
Qx4&=vcmp.eq(Vu.uw,Vv.uw)	`Assembler mapped to: "Qx4&=vcmp.eq(Vu." "w" ",Vv." "w" ")"`
Qx4|=vcmp.eq(Vu.uw,Vv.uw)	`Assembler mapped to: "Qx4|=vcmp.eq(Vu." "w" ",Vv." "w" ")"`
Qx4^=vcmp.eq(Vu.uw,Vv.uw)	`Assembler mapped to: "Qx4^=vcmp.eq(Vu." "w" ",Vv." "w" ")"`

Syntax	Behavior
Qd4=vcmp.eq(Vu.uh,Vv.uh)	`Assembler mapped to: "Qd4=vcmp.eq(Vu." "h" ",Vv." "h" ")"`
Qx4&=vcmp.eq(Vu.uh,Vv.uh)	`Assembler mapped to: "Qx4&=vcmp.eq(Vu." "h" ",Vv." "h" ")"`
Qx4|=vcmp.eq(Vu.uh,Vv.uh)	`Assembler mapped to: "Qx4|=vcmp.eq(Vu." "h" ",Vv." "h" ")"`
Qx4^=vcmp.eq(Vu.uh,Vv.uh)	`Assembler mapped to: "Qx4^=vcmp.eq(Vu." "h" ",Vv." "h" ")"`
Qd4=vcmp.eq(Vu.ub,Vv.ub)	`Assembler mapped to: "Qd4=vcmp.eq(Vu." "b" ",Vv." "b" ")"`
Qx4&=vcmp.eq(Vu.ub,Vv.ub)	`Assembler mapped to: "Qx4&=vcmp.eq(Vu." "b" ",Vv." "b" ")"`
Qx4|=vcmp.eq(Vu.ub,Vv.ub)	`Assembler mapped to: "Qx4|=vcmp.eq(Vu." "b" ",Vv." "b" ")"`
Qx4^=vcmp.eq(Vu.ub,Vv.ub)	`Assembler mapped to: "Qx4^=vcmp.eq(Vu." "b" ",Vv." "b" ")"`
Qx4&=vcmp.gt(Vu.sf,Vv.sf)	`for( i = 0; i < VWIDTH; i += 4) { VAL = (Vu.sf[i/4] > Vv.sf[i/4]) ? 0xF : 0; QxV[i+4-1:i] = QxV[i+4-1:i] & VAL; }`

Syntax	Behavior

Qx4^=vcmp.gt(Vu.sf,Vv.sf)	`for( i = 0; i < VWIDTH; i += 4) {`
	`  VAL = (Vu.sf[i/4] > Vv.sf[i/4]) ? 0xF : 0;`
	`  QxV[i+4-1:i] = QxV[i+4-1:i] ^ VAL;`
	`}`

Qx4|=vcmp.gt(Vu.sf,Vv.sf)	`for( i = 0; i < VWIDTH; i += 4) {`
	`  VAL = (Vu.sf[i/4] > Vv.sf[i/4]) ? 0xF : 0;`
	`  QxV[i+4-1:i] = QxV[i+4-1:i] | VAL;`
	`}`

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

Qx4&=vcmp.gt(Vu.hf,Vv.hf)	`for( i = 0; i < VWIDTH; i += 2) {`
	`  VAL = (Vu.hf[i/2] > Vv.hf[i/2]) ? 0x3 : 0;`
	`  QxV[i+2-1:i] = QxV[i+2-1:i] & VAL;`
	`}`

Qx4^=vcmp.gt(Vu.hf,Vv.hf)	`for( i = 0; i < VWIDTH; i += 2) {`
	`  VAL = (Vu.hf[i/2] > Vv.hf[i/2]) ? 0x3 : 0;`
	`  QxV[i+2-1:i] = QxV[i+2-1:i] ^ VAL;`
	`}`

Qx4|=vcmp.gt(Vu.hf,Vv.hf)	`for( i = 0; i < VWIDTH; i += 2) {`
	`  VAL = (Vu.hf[i/2] > Vv.hf[i/2]) ? 0x3 : 0;`
	`  QxV[i+2-1:i] = QxV[i+2-1:i] | VAL;`
	`}`

Syntax	Behavior
Qd4=vcmp.gt(Vu.hf,Vv.hf)	`for( i = 0; i < VWIDTH; i += 2) { VAL = (Vu.hf[i/2] > Vv.hf[i/2]) ? 0x3 : 0; QdV[i+2-1:i] = VAL; }`
Qx4&=vcmp.gt(Vu.bf,Vv.bf)	`for( i = 0; i < VWIDTH; i += 2) { VAL = (Vu.bf[i/2] > Vv.bf[i/2]) ? 0x3 : 0; QxV[i+2-1:i] = QxV[i+2-1:i] & VAL; }`
Qx4^=vcmp.gt(Vu.bf,Vv.bf)	`for( i = 0; i < VWIDTH; i += 2) { VAL = (Vu.bf[i/2] > Vv.bf[i/2]) ? 0x3 : 0; QxV[i+2-1:i] = QxV[i+2-1:i] ^ VAL; }`
Qx4|=vcmp.gt(Vu.bf,Vv.bf)	`for( i = 0; i < VWIDTH; i += 2) { VAL = (Vu.bf[i/2] > Vv.bf[i/2]) ? 0x3 : 0; QxV[i+2-1:i] = QxV[i+2-1:i] | VAL; }`
Qd4=vcmp.gt(Vu.bf,Vv.bf)	`for( i = 0; i < VWIDTH; i += 2) { VAL = (Vu.bf[i/2] > Vv.bf[i/2]) ? 0x3 : 0; QdV[i+2-1:i] = VAL; }`

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

Note:

- This instruction can use any HVX resource.

Encodings

Bitmap

Syntax	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
Qd4=vcmp.gt(Vu.uw,Vv.w)	0	0	0	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
Qx4&=vcmp.gt(Vu.uw,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	1	0	x	x
Qx4|=vcmp.gt(Vu.uw,Vv.w)	0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	0	1	1	0	x	x
Qx4^=vcmp.gt(Vu.uw,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	1	0	x	x
Qd4=vcmp.eq(Vu.uw,Vv.w)	0	0	0	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
Qx4&=vcmp.eq(Vu.uw,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	0	1	0	x	x
Qx4|=vcmp.eq(Vu.uw,Vv.w)	0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	0	0	1	0	x	x
Qx4^=vcmp.eq(Vu.uw,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	0	1	0	x	x
Qd4=vcmp.gt(Vu.uh,Vv.h)	0	0	0	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
Qx4&=vcmp.gt(Vu.uh,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	0	1	x	x
Qx4|=vcmp.gt(Vu.uh,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	0	1	x	x
Qx4^=vcmp.gt(Vu.uh,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	0	1	x	x
Qd4=vcmp.eq(Vu.uh,Vv.h)	0	0	0	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
Qx4&=vcmp.eq(Vu.uh,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	0	1	x	x
Qx4|=vcmp.eq(Vu.uh,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	0	1	x	x
Qx4^=vcmp.eq(Vu.uh,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	0	1	x	x
Qd4=vcmp.gt(Vu.ub,Vv.b)	0	0	0	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
Qx4&=vcmp.gt(Vu.ub,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	0	x	x
Qx4|=vcmp.gt(Vu.ub,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	0	x	x
Qx4^=vcmp.gt(Vu.ub,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	0	x	x
Qd4=vcmp.eq(Vu.ub,Vv.b)	0	0	0	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
Qx4&=vcmp.eq(Vu.ub,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	0	x	x
Qx4|=vcmp.eq(Vu.ub,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	0	x	x
Qx4^=vcmp.eq(Vu.ub,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	0	x	x
Qd4=vcmp.gt(Vu.uw,Vv.uw)	0	0	0	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
Qx4&=vcmp.gt(Vu.uw,Vv.uw)	0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	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	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	1	0	1	0	x	x
Qd4=vcmp.gt(Vu.uh,Vv.uh)	0	0	0	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
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	0	1	x	x

Syntax	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
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	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	0	1	x	x
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	0	d	d
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	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	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	0	x	x
Qx4&=vcmp.gt(Vu.sf,Vv.sf)	0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	0	0	1	0	x	x
Qx4^=vcmp.gt(Vu.sf,Vv.sf)	0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	0	1	0	x	x
Qx4|=vcmp.gt(Vu.sf,Vv.sf)	0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	1	0	0	x	x
Qd4=vcmp.gt(Vu.sf,Vv.sf)	0	0	0	1	1	1	1	1	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	1	0	0	d	d
Qx4&=vcmp.gt(Vu.hf,Vv.hf)	0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	0	0	1	1	x	x
Qx4^=vcmp.gt(Vu.hf,Vv.hf)	0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	0	1	1	x	x
Qx4|=vcmp.gt(Vu.hf,Vv.hf)	0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	1	0	1	x	x
Qd4=vcmp.gt(Vu.hf,Vv.hf)	0	0	0	1	1	1	1	1	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	1	0	1	d	d
Qx4&=vcmp.gt(Vu.bf,Vv.bf)	0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	0	1	0	0	x	x
Qx4^=vcmp.gt(Vu.bf,Vv.bf)	0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	1	0	0	x	x
Qx4|=vcmp.gt(Vu.bf,Vv.bf)	0	0	0	1	1	1	0	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	1	1	0	x	x
Qd4=vcmp.gt(Vu.bf,Vv.bf)	0	0	0	1	1	1	1	1	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	1	1	0	d	d

Intrinsics

Compare vectors intrinsics

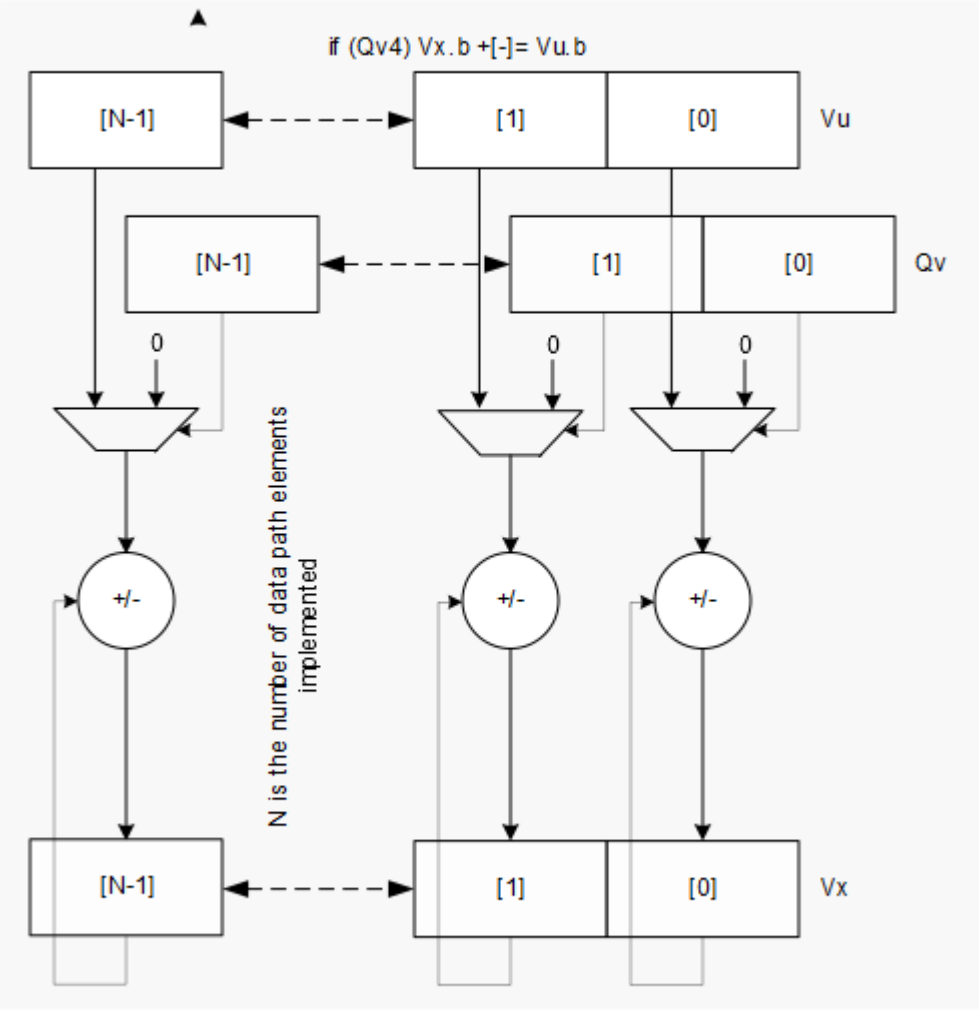
Qd4=vcmp.gt(Vu.w,Vv.w)	HVX_VectorPred Q6_Q_vcmp_gt_VwVw(HVX_Vector Vu, HVX_Vector Vv)
Qx4&=vcmp.gt(Vu.w,Vv.w)	HVX_VectorPred Q6_Q_vcmp_gtand_QVwVw(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)
Qx4|=vcmp.gt(Vu.w,Vv.w)	HVX_VectorPred Q6_Q_vcmp_gtor_QVwVw(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)
Qx4^=vcmp.gt(Vu.w,Vv.w)	HVX_VectorPred Q6_Q_vcmp_gtxacc_QVwVw(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)
Qd4=vcmp.eq(Vu.w,Vv.w)	HVX_VectorPred Q6_Q_vcmp_eq_VwVw(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.eq(Vu.w,Vv.w)	HVX_VectorPred Q6_Q_vcmp_eqor_QVwVw(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)
Qd4=vcmp.gt(Vu.h,Vv.h)	HVX_VectorPred Q6_Q_vcmp_gt_VhVh(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.h,Vv.h)	HVX_VectorPred Q6_Q_vcmp_gtor_QVhVh(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)
Qd4=vcmp.eq(Vu.h,Vv.h)	HVX_VectorPred Q6_Q_vcmp_eq_VhVh(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.h,Vv.h)	HVX_VectorPred Q6_Q_vcmp_eqor_QVhVh(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)
Qd4=vcmp.gt(Vu.b,Vv.b)	HVX_VectorPred Q6_Q_vcmp_gt_VbVb(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.b,Vv.b)	HVX_VectorPred Q6_Q_vcmp_gtor_QVbVb(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)
Qd4=vcmp.eq(Vu.b,Vv.b)	HVX_VectorPred Q6_Q_vcmp_eq_VbVb(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.b,Vv.b)	HVX_VectorPred Q6_Q_vcmp_eqor_QVbVb(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)
Qd4=vcmp.gt(Vu.uw,Vv.uw)	HVX_VectorPred Q6_Q_vcmp_gt_VuwVuw(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.uw,Vv.uw)	HVX_VectorPred Q6_Q_vcmp_gtor_QVuwVuw(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)
Qd4=vcmp.gt(Vu.uh,Vv.uh)	HVX_VectorPred Q6_Q_vcmp_gt_VuhVuh(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.uh,Vv.uh)	HVX_VectorPred Q6_Q_vcmp_gtor_QVuhVuh(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)
Qd4=vcmp.gt(Vu.ub,Vv.ub)	HVX_VectorPred Q6_Q_vcmp_gt_VubVub(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.ub,Vv.ub)	HVX_VectorPred Q6_Q_vcmp_gtor_QVubVub(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.sf,Vv.sf)	HVX_VectorPred Q6_Q_vcmp_gtand_QVsfVsf(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)
Qx4^=vcmp.gt(Vu.sf,Vv.sf)	HVX_VectorPred Q6_Q_vcmp_gtxacc_QVsfVsf(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)
Qx4|=vcmp.gt(Vu.sf,Vv.sf)	HVX_VectorPred Q6_Q_vcmp_gtor_QVsfVsf(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)
Qd4=vcmp.gt(Vu.sf,Vv.sf)	HVX_VectorPred Q6_Q_vcmp_gt_VsfVsf(HVX_Vector Vu, HVX_Vector Vv)
Qx4&=vcmp.gt(Vu.hf,Vv.hf)	HVX_VectorPred Q6_Q_vcmp_gtand_QVhfVhf(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)
Qx4^=vcmp.gt(Vu.hf,Vv.hf)	HVX_VectorPred Q6_Q_vcmp_gtxacc_QVhfVhf(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)
Qx4|=vcmp.gt(Vu.hf,Vv.hf)	HVX_VectorPred Q6_Q_vcmp_gtor_QVhfVhf(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)
Qd4=vcmp.gt(Vu.hf,Vv.hf)	HVX_VectorPred Q6_Q_vcmp_gt_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)
Qx4&=vcmp.gt(Vu.bf,Vv.bf)	HVX_VectorPred Q6_Q_vcmp_gtand_QVbfVbf(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)
Qx4^=vcmp.gt(Vu.bf,Vv.bf)	HVX_VectorPred Q6_Q_vcmp_gtxacc_QVbfVbf(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)
Qx4|=vcmp.gt(Vu.bf,Vv.bf)	HVX_VectorPred Q6_Q_vcmp_gtor_QVbfVbf(HVX_VectorPred Qx, HVX_Vector Vu, HVX_Vector Vv)
Qd4=vcmp.gt(Vu.bf,Vv.bf)	HVX_VectorPred Q6_Q_vcmp_gt_VbfVbf(HVX_Vector Vu, HVX_Vector Vv)

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.



Conditional accumulate instructions

Syntax	Behavior
if (Qv4) Vx.b+=Vu.b	`for (i = 0; i < VELEM(8); i++) { Vx.ub[i]=QvV.i ? Vx.ub[i]+Vu.ub[i] : Vx.ub[i]; }`
if (Qv4) Vx.b-=Vu.b	`for (i = 0; i < VELEM(8); i++) { Vx.ub[i]=QvV.i ? Vx.ub[i]-Vu.ub[i] : Vx.ub[i]; }`
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.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]+Vu.h[i],Vx.h[i]); }`
if (Qv4) Vx.h-=Vu.h	`for (i = 0; i < VELEM(16); i++) { Vx.h[i]=select_bytes(QvV,i,Vx.h[i]-Vu.h[i],Vx.h[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]); }`

Syntax	Behavior
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]+Vu.w[i],Vx.w[i]); }`
if (Qv4) Vx.w-=Vu.w	`for (i = 0; i < VELEM(32); i++) { Vx.w[i]=select_bytes(QvV,i,Vx.w[i]- Vu.w[i],Vx.w[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]); }`
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: HVX (slots 0,1,2,3)

Note:

- This instruction can use any HVX resource.

Encodings

Bitmap

Syntax	**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**
if (Qv4) Vx.b+=Vu.b	0	0	0	1	1	1	1	0	v	v	0	-	-	0	0	1	P	P	1	u	u	u	u	u	0	0	0	x	x	x	x	x
if (Qv4) Vx.b-=Vu.b	0	0	0	1	1	1	1	0	v	v	0	-	-	0	0	1	P	P	1	u	u	u	u	u	1	1	0	x	x	x	x	x
if (!Qv4) Vx.b+=Vu.b	0	0	0	1	1	1	1	0	v	v	0	-	-	0	0	1	P	P	1	u	u	u	u	u	0	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	0	P	P	1	u	u	u	u	u	0	0	1	x	x	x	x	x
if (Qv4) Vx.h+=Vu.h	0	0	0	1	1	1	1	0	v	v	0	-	-	0	0	1	P	P	1	u	u	u	u	u	0	0	1	x	x	x	x	x
if (Qv4) Vx.h-=Vu.h	0	0	0	1	1	1	1	0	v	v	0	-	-	0	0	1	P	P	1	u	u	u	u	u	1	1	1	x	x	x	x	x
if (!Qv4) Vx.h+=Vu.h	0	0	0	1	1	1	1	0	v	v	0	-	-	0	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	0	P	P	1	u	u	u	u	u	0	1	0	x	x	x	x	x
if (Qv4) Vx.w+=Vu.w	0	0	0	1	1	1	1	0	v	v	0	-	-	0	0	1	P	P	1	u	u	u	u	u	0	1	0	x	x	x	x	x
if (Qv4) Vx.w-=Vu.w	0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	0	P	P	1	u	u	u	u	u	0	0	0	x	x	x	x	x
if (!Qv4) Vx.w+=Vu.w	0	0	0	1	1	1	1	0	v	v	0	-	-	0	0	1	P	P	1	u	u	u	u	u	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	0	P	P	1	u	u	u	u	u	0	1	1	x	x	x	x	x

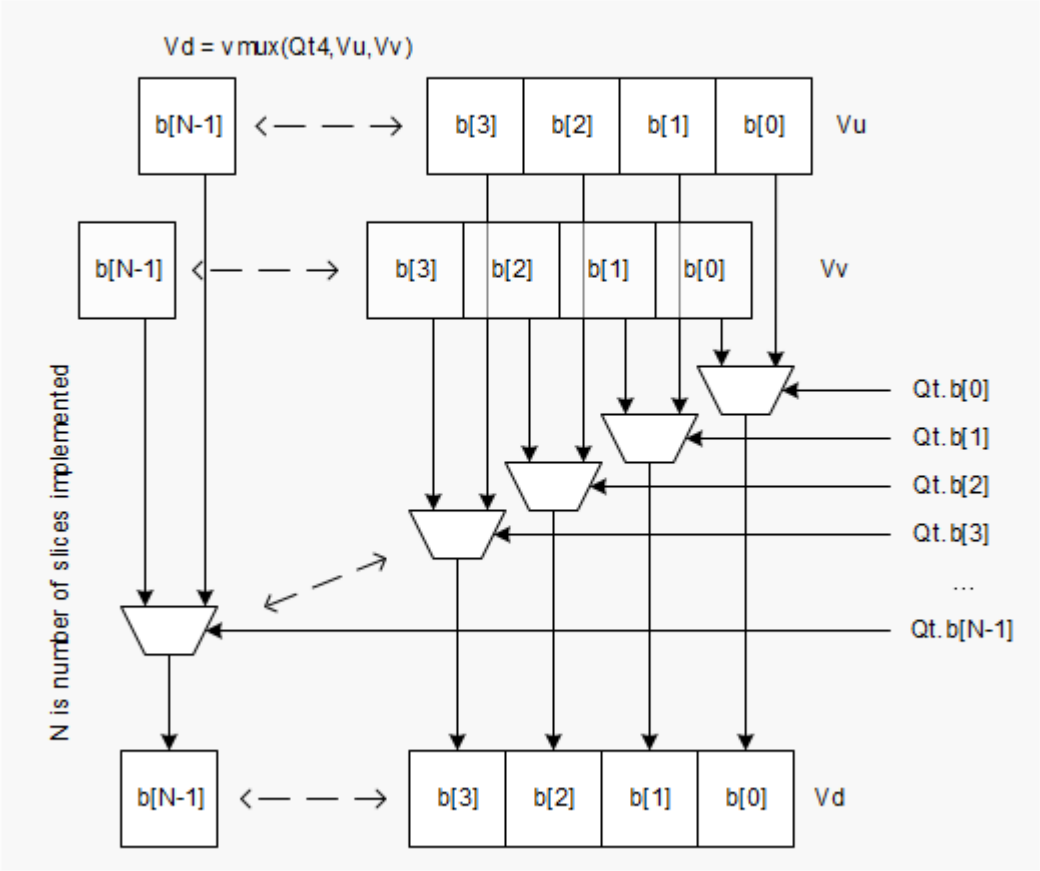
Intrinsics

Conditional accumulate intrinsics

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.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_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.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_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)
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)

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.



Mux select instructions

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: HVX (slots 0,1,2,3)

Note:

- This instruction can use any HVX resource.

Encodings

Bitmap

Syntax	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
Vd=vmux(Qt4,Vu,Vv)	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

Intrinsics

Mux select intrinsics

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

Syntax	Behavior

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.

Saturation instructions

Syntax	Behavior
Vd.w=vsatdw(Vu.w,Vv.w)	`for (i = 0; i < VELEM(32); i++) { Vd.w[i] = usat_32(Vu.w[i]:Vv.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.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.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: HVX (slots 0,1,2,3)

Note:

- This instruction can use any HVX resource.

Encodings

Bitmap

Syntax	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
Vd.w=vsatdw(Vu.w,Vv.w)	0	v	v	v	1	1	0	1	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	d	d	d	d	d
Vd.ub=vsat(Vu.h,Vv.h)	0	v	v	v	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.h=vsat(Vu.w,Vv.w)	0	v	v	v	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.uh=vsat(Vu.uw,Vv.uw)	0	v	v	v	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

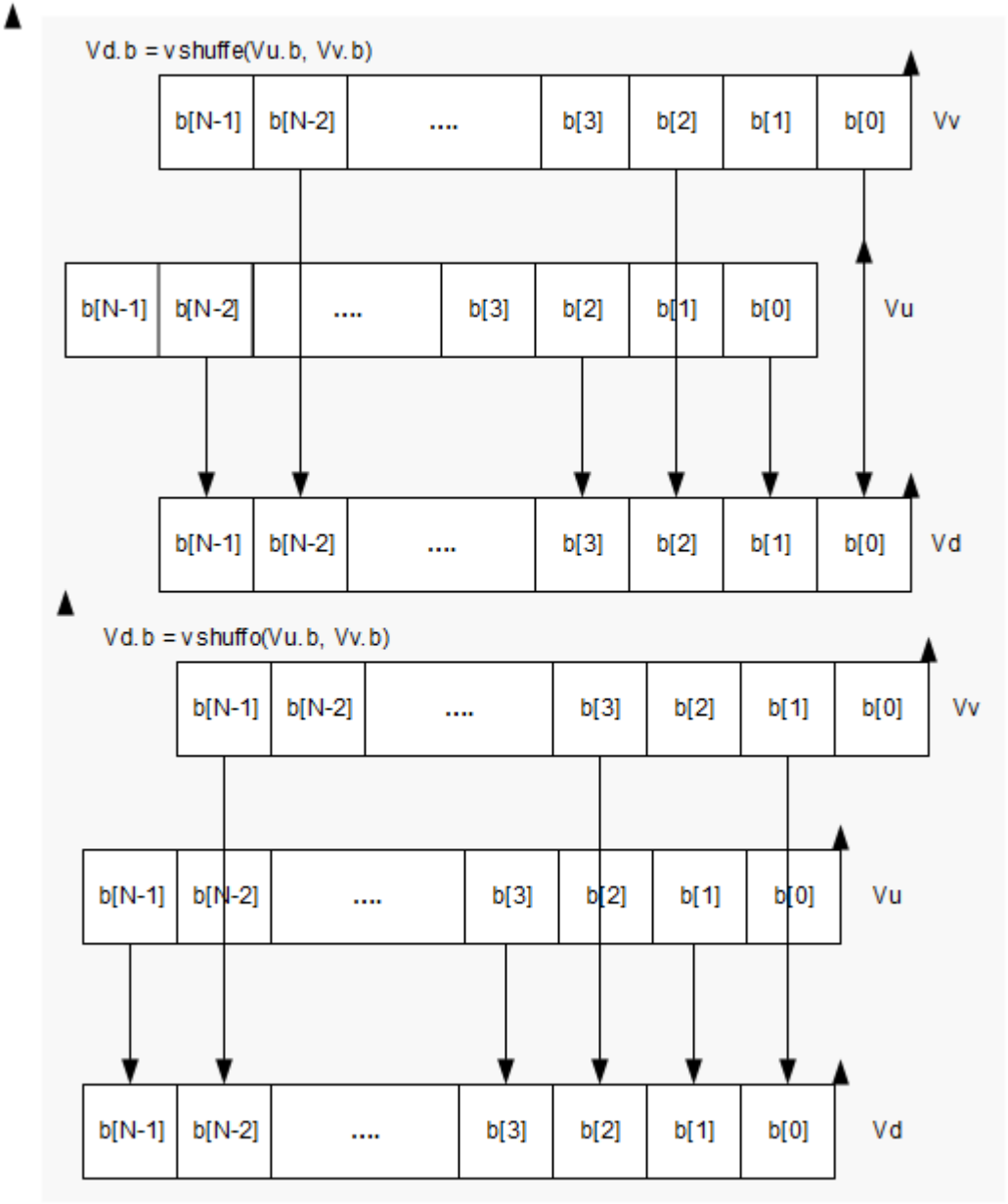
Intrinsics

Saturation intrinsics

Vd.w=vsatdw(Vu.w,Vv.w)	HVX_Vector Q6_Vw_vsatdw_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.h=vsat(Vu.w,Vv.w)	HVX_Vector Q6_Vh_vsat_VwVw(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)

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.

In-lane shuffle instructions

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: HVX (slots 0,1,2,3)

Note:

- This instruction can use any HVX resource.

Encodings

Bitmap

Syntax	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
Vd.b=vshuffe(Vu.b,Vv.b)	0	v	u	0	0	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=vshuffo(Vu.b,Vv.b)	0	v	u	0	0	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.h=vshuffe(Vu.h,Vv.h)	0	v	u	0	0	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=vshuffo(Vu.h,Vv.h)	0	v	u	0	0	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

Intrinsics

In-lane shuffle 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)

Clear vector register

Set a single or double vector register to 0.

Clear vector register instructions

Syntax	Behavior
Vd=#0	`Assembler mapped to: "Vd=vxor(V31,V31)"`
Vdd=#0	`Assembler mapped to: "Vdd.w=vsub(V31:30.w,V31:30.w)"`

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

Encodings

Bitmap

Syntax	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

Intrinsics

Clear vector register intrinsics

Vd=#0	HVX_Vector Q6_V_vzero()
Vdd=#0	HVX_VectorPair Q6_W_vzero()

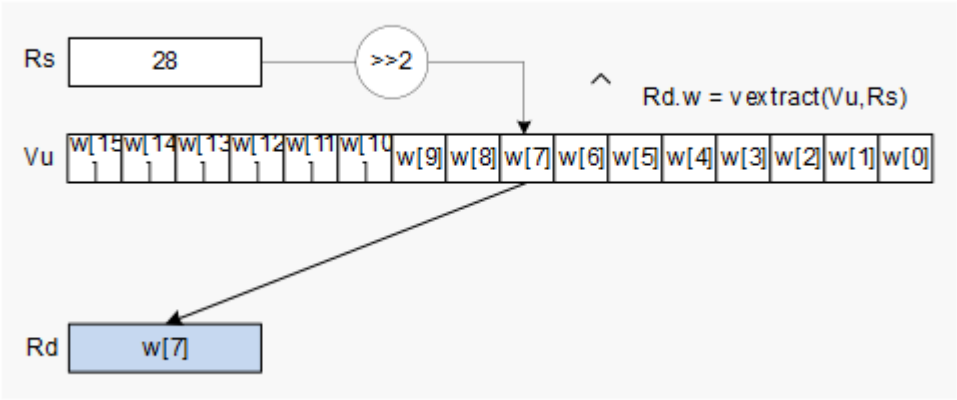
DEBUG

The HVX debug instruction subclass includes debugging instructions.

Extract vector element

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

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



Extract vector element instructions

Syntax	Behavior
Rd=vextract(Vu,Rs)	`Rd = Vu.uw[ (Rs & (VWIDTH-1)) >> 2];`
Rd.w=vextract(Vu,Rs)	`Assembler mapped to: "Rd=vextract(Vu,Rs)"`

Class: HVX (slots 0)

Note:

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

Encodings

Bitmap

Syntax	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
Rd=vextract(Vu,Rs)	1	u	u	0	1	0	0	1	0	0	0	0	s	s	s	s	s	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d

Intrinsics

Extract vector element intrinsics

Rd=vextract(Vu,Rs)	Word32 Q6_R_vextract_VR(HVX_Vector Vu, Word32 Rs)

GATHER-DOUBLE-RESOURCE

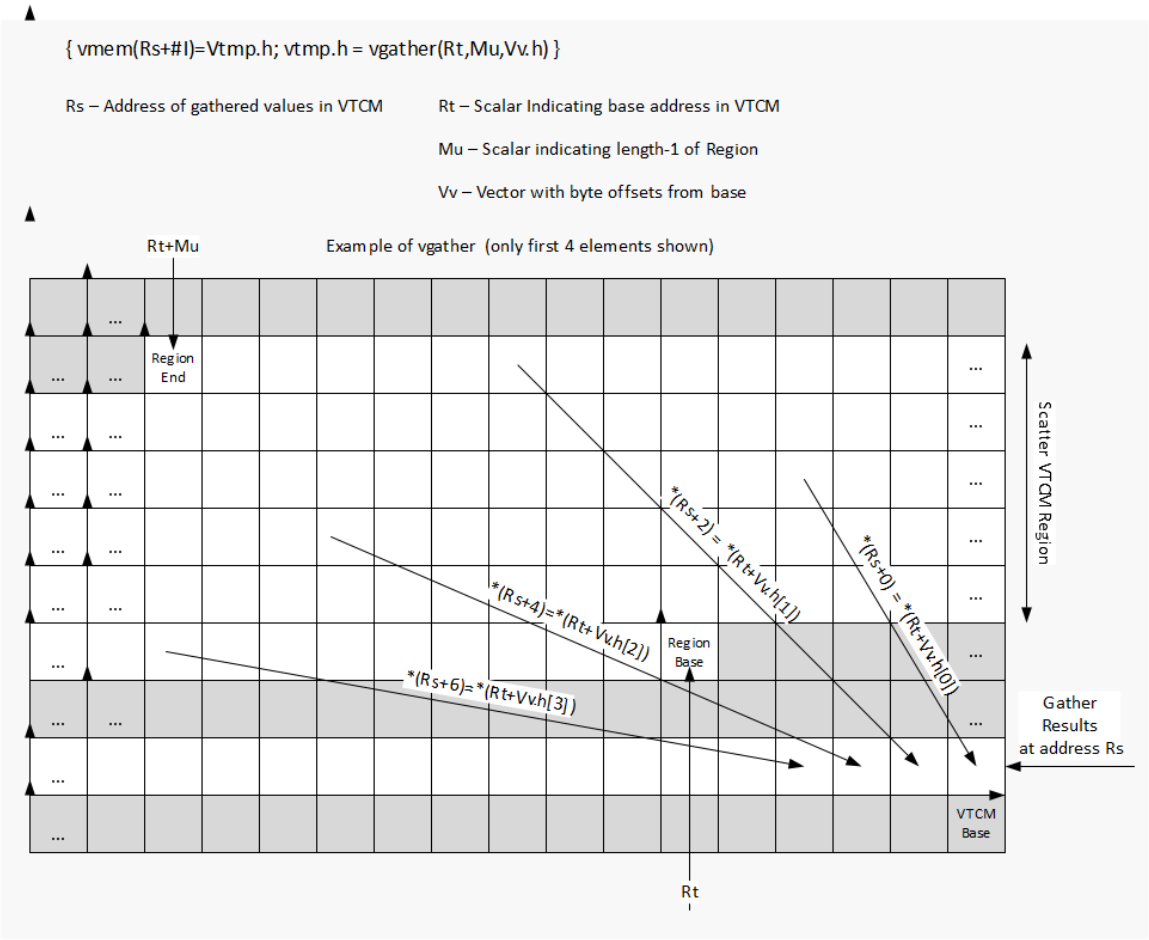
The HVX gather double resource instruction subclass includes instructions that perform gather operations into the vector TCM.

Vector gather

The following instructions perform gather operations in the vector TCM. Gather operations are effectively element copies from a large region in VTCM to a smaller vector-sized region. The larger region of memory is specified by two scalar registers: Rt 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, Vv, specifies byte offsets to this region. Elements of halfword granularity are copied from the address pointed to by Rt + Vv for each element in the vector to the corresponding element in the linear element pointed to by the accompanying store.

The offset vector, Vv, 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's simple dropped. Offsets must be positive otherwise they will be dropped. A vector predicate register can also be specified. If a the predicate is false, that byte will not be copied. This can be used to emulate a byte gather.

The gather instruction must be paired with a VMEM .new store that uses a tmp register source. Example: { VMEM(R0+#0) = Vtmp.new; Vtmp.h = vgather(R1,M0, V1:0.w); } gathers halfwords with halfword addresses and saves the results to the address pointed to by R0 of the VMEM instruction. If a vgather is not accompanied with a store, it will get dropped.



Vector gather instructions

Syntax	Behavior

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

Syntax	Behavior

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

Class: HVX (slots 0,1)

Note:

- This instruction can use any HVX resource.

Encodings

Bitmap

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

Intrinsics

Vector gather intrinsics

vtmp.h=vgather(Rt,Mu,Vvv.w).h	void Q6_vgather_ARMWw(HVX_Vector* A, HVX_Vector* Rb, Word32 Mu, HVX_VectorPair Vvv)
if (Qs4) vtmp.h=vgather(Rt,Mu,Vvv.w).h	void Q6_vgather_AQRMWw(HVX_Vector* A, HVX_VectorPred Qs, HVX_Vector* Rb, Word32 Mu, HVX_VectorPair Vvv)

GATHER

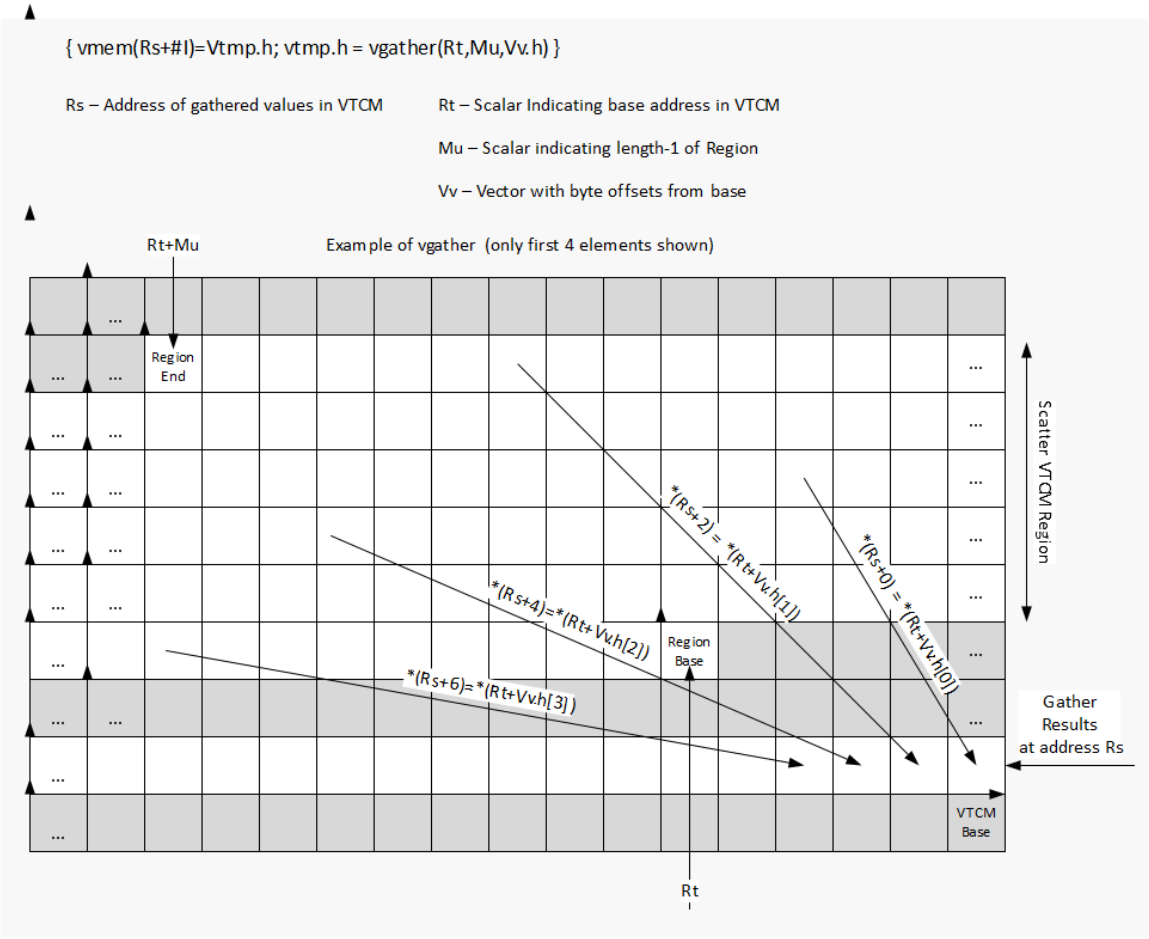
The HVX gather instruction subclass includes instructions that perform gather operations.

Vector gather

The following instructions perform gather operations in the vector TCM. Gather operations are effectively element copies from a large region in VTCM to a smaller vector-sized region. The larger region of memory is specified by two scalar registers: Rt 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, Vv, specifies byte offsets to this region. Elements of either halfword or word granularity are copied from the address pointed to by Rt + Vv for each element in the vector to the corresponding element in the linear element pointed to by the accompanying store.

The offset vector, Vv, 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's simple dropped. Offsets must be positive otherwise they will be dropped. A vector predicate register can also be specified. If a the predicate is false, that byte will not be copied. This can be used to emulate a byte gather.

The gather instruction must be paired with a VMEM .new store that uses a tmp register source. Example: { VMEM(R0+#0) = Vtmp.new; Vtmp.h = vgather(R1,M0, 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 will get dropped.



Vector gather instructions

Syntax	Behavior

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

Syntax	Behavior

vtmp.h=vgather(Rt,Mu,Vv.h).h	`MuV = MuV | (element_size-1);`
	`Rt = Rt & ~(element_size-1);`
	`for (i = 0; i < VELEM(16); i++) {`
	`  EA = Rt+Vv.uh[i];`
	`  if (Rt <= EA <= Rt + MuV) TEMP.uh[i] = *EA;`
	`}`

if (Qs4) vtmp.w=vgather(Rt,Mu,Vv.w).w	`MuV = MuV | (element_size-1);`
	`Rt = Rt & ~(element_size-1);`
	`for (i = 0; i < VELEM(32); i++) {`
	`  EA = Rt+Vv.uw[i];`
	`  if ( (Rt <= EA <= Rt + MuV) & QsV) TEMP.uw[i] = *EA;`
	`}`

if (Qs4) vtmp.h=vgather(Rt,Mu,Vv.h).h	`MuV = MuV | (element_size-1);`
	`Rt = Rt & ~(element_size-1);`
	`for (i = 0; i < VELEM(16); i++) {`
	`  EA = Rt+Vv.uh[i];`
	`  if ( (Rt <= EA <= Rt + MuV) & QsV) TEMP.uh[i] = *EA;`
	`}`

Class: HVX (slots 0,1)

Note:

- This instruction can use any HVX resource.

Encodings

Bitmap

Syntax	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
vtmp.w=vgather(Rt,Mu,Vv.w).w	0	0	1	0	0	0	1	1	0	0	0	t	t	t	t	t	P	P	u	-	-	0	0	0	-	-	-	v	v	v	v	v

Syntax	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
vtmp.h=vgather(Rt,Mu,Vv.h).h	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
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	0	-	s	s	v	v	v	v	v
if (Qs4) vtmp.h=vgather(Rt,Mu,Vv.h).h	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

Intrinsics

Vector gather intrinsics

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

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 simply 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.

Load - aligned instructions

Syntax	Behavior

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

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

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

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

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

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

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

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

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

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

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

Class: HVX (slots 0,1)

Note:

- This instruction can use any HVX resource.

- An optional “non-temporal” 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.

Encodings

Bitmap

Syntax	**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**
Vd=vmem(Rx++#s3)	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(Rt+#s4)	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(Rx++Mu)	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
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	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	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	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(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(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
Vd=vmem(Rx++#s3):nt	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(Rt+#s4):nt	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(Rx++Mu):nt	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
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	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	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	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(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

Syntax	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
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

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 simply 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.

Load - immediate use instructions

Syntax	Behavior

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

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

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

Syntax	Behavior
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(Rt+#s4)	`if (Pv[0]) { EA=Rt+s*VBYTES; Vd = *(EA&~(ALIGNMENT-1)); } 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++#s3)	`if (!Pv[0]) { EA=Rx; Vd = *(EA&~(ALIGNMENT-1)); Rx=Rx+s*VBYTES; } else { NOP; }`

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

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

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

Class: HVX (slots 0,1)

Note:

- This instruction can use any HVX resource.
- An optional “non-temporal” 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.

Encodings

Bitmap

Syntax	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
Vd.cur=vmem(Rx++#s3):1	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(Rt+#s4):1	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(Rx++Mu):1	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
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	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	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	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(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(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
Vd.cur=vmem(Rx++#s3):nt	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(Rt+#s4):nt	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(Rx++Mu):nt	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
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	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	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	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(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(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

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 simply 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.

Load - temporary immediate use instructions

Syntax	Behavior

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

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

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

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

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

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

Class: HVX (slots 0,1)

Note:

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

Syntax	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

Encodings

Bitmap

Syntax	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
Vd.tmp=vmem(Rx++#s3)	0	0	1	0	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(Rt++#s4)	0	0	1	0	1	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(Rx++Mu)	0	0	1	0	1	0	1	1	0	0	x	x	x	x	x	P	P	u	0	0	-	-	-	0	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	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	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	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(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(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
Vd.tmp=vmem(Rx++#s3):nt	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(Rt++#s4):nt	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(Rx++Mu):nt	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
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	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	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	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(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(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

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 synthesise a non-aligned access.

Note that this instruction uses both slot 0 and slot 1, allowing only 3 instructions at most to execute in a packet with vmemu in it.

Load - unaligned instructions

Syntax	Behavior

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

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

Vd=vmemu(Rx++Mu)	`EA=Rx;`
	`Vd = *EA;`
	`Rx=Rx+MuV;`

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

Class: HVX (slots 0)

Note:

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

Encodings

Bitmap

Syntax	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
Vd=vmemu(Rx++#s3)	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++#s4):t	0	1	0	0	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(Rx++Mu)	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	d	d	d

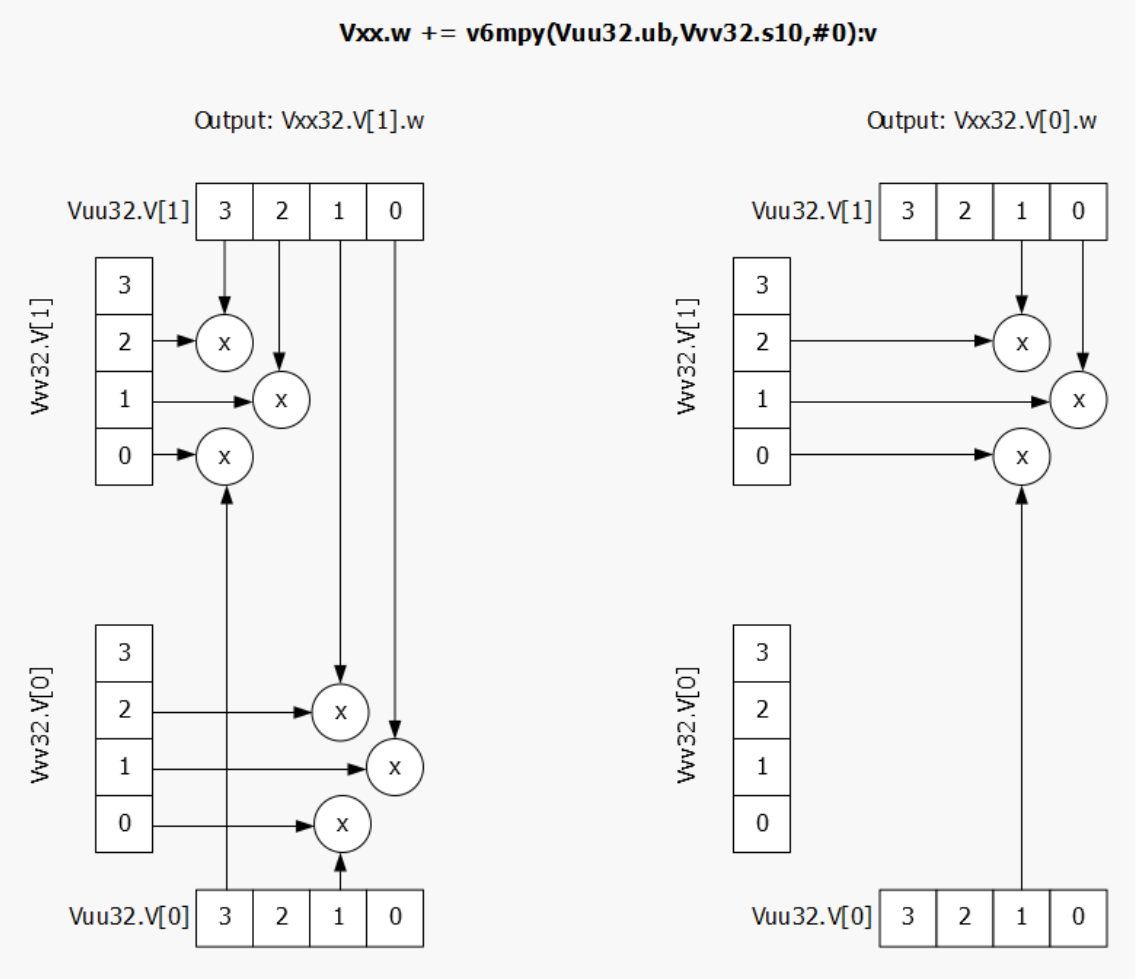
MPY-DOUBLE-RESOURCE

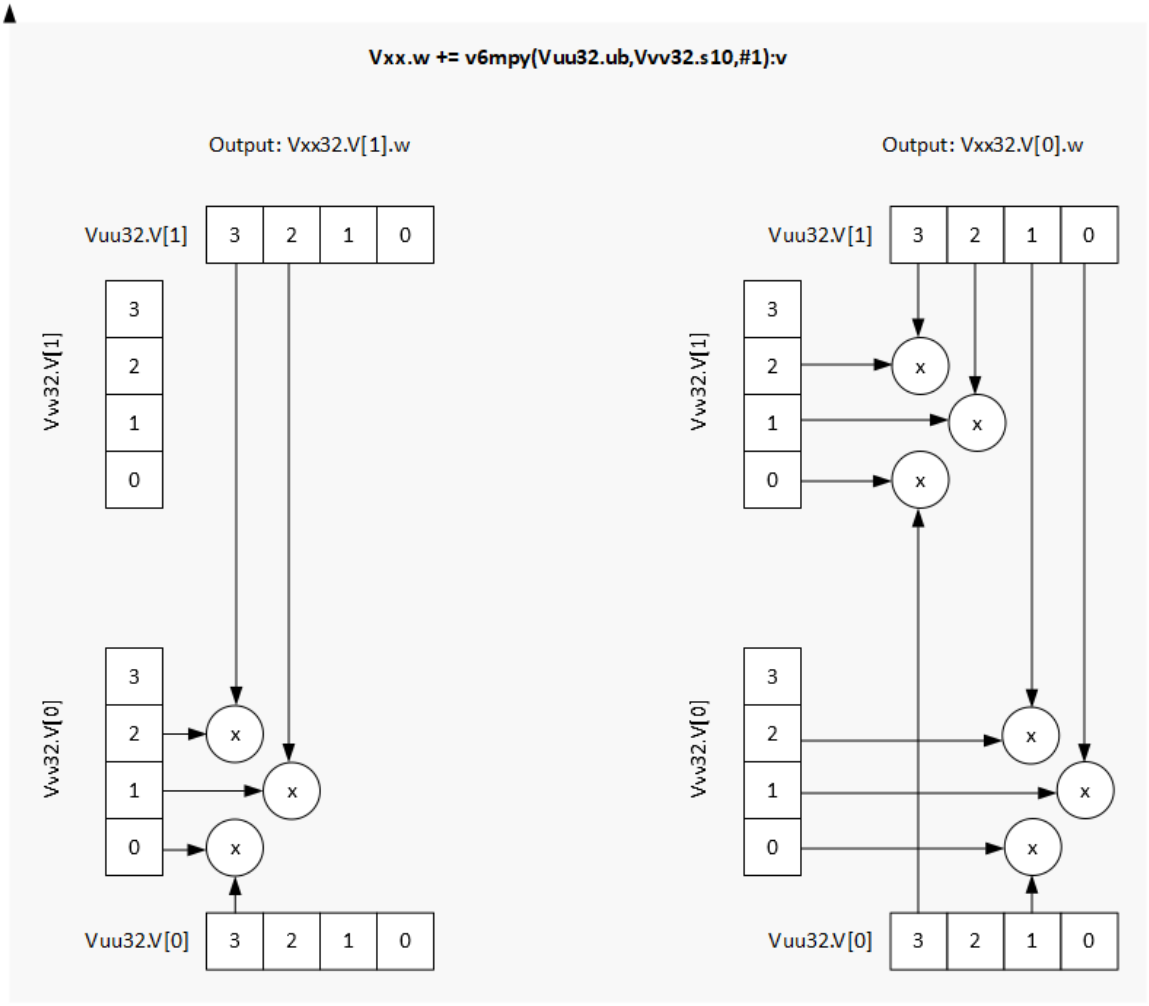
The HVX MPY double resource instruction subclass includes memory load instructions.

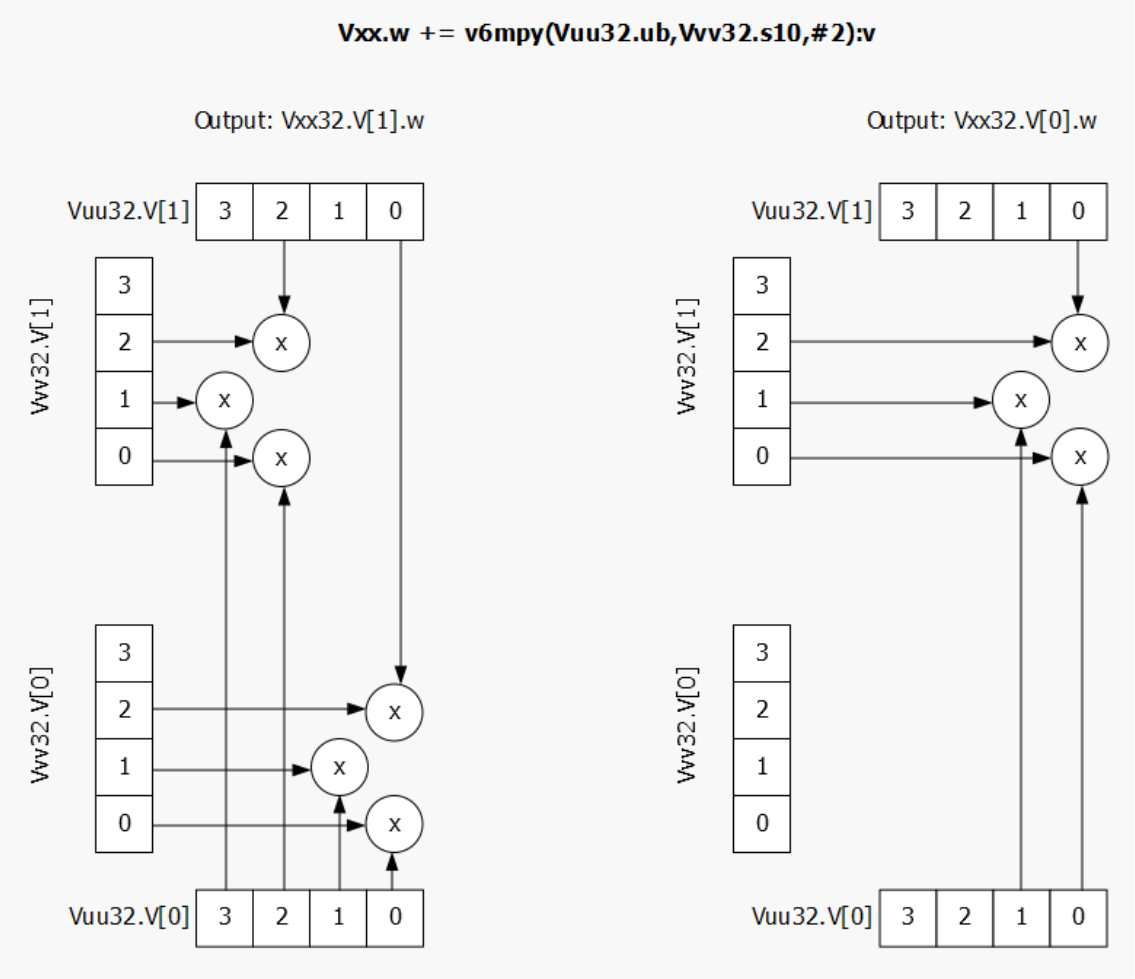
3x3 Multiply for 2x2 Tile

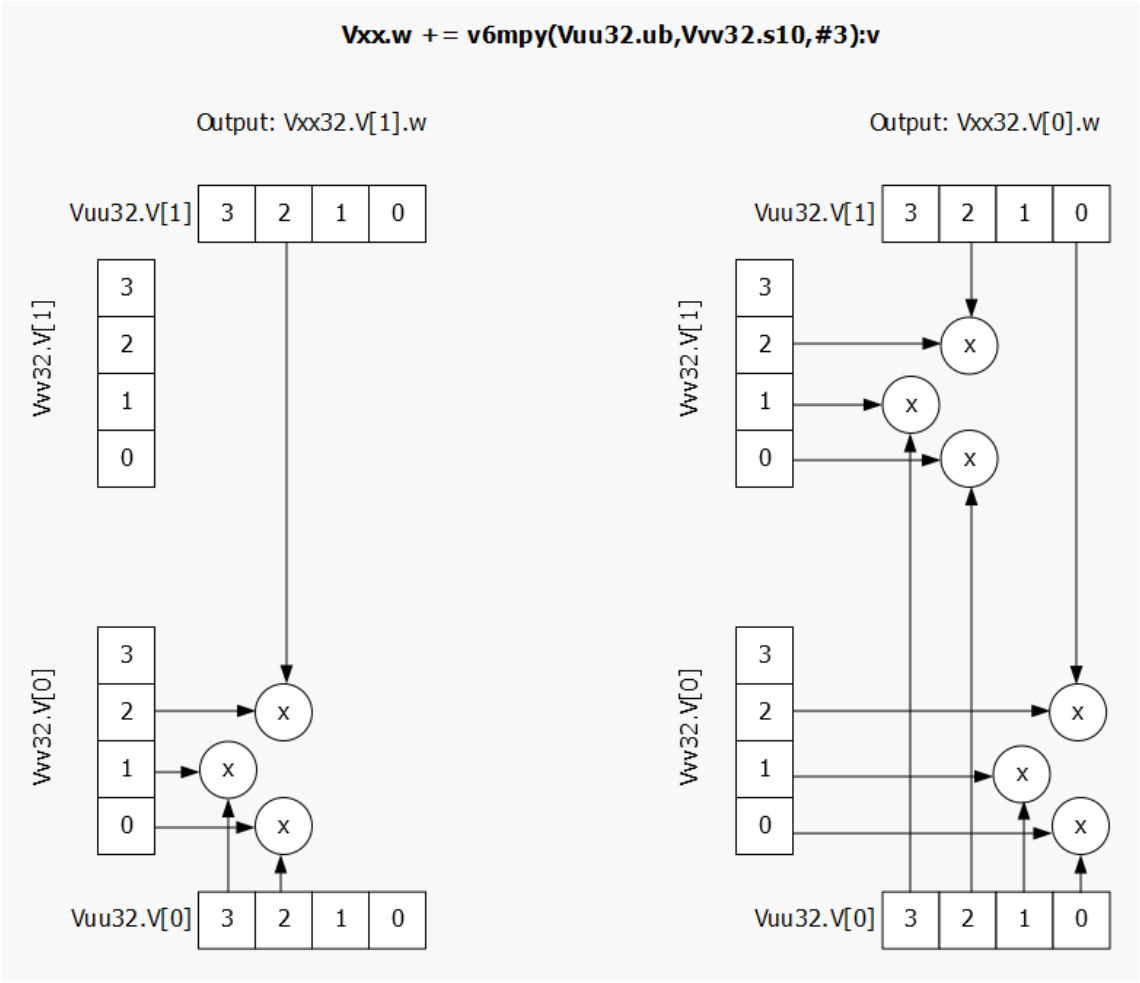
Multiply optimized for 3x3 filtering for a 2x2 tile format of 2 horizontal by 2 vertical bytes with 3 10-bit coefficients. Byte 4 of the inlane word in the coefficient vector specifies the upper two bits for each coefficient.

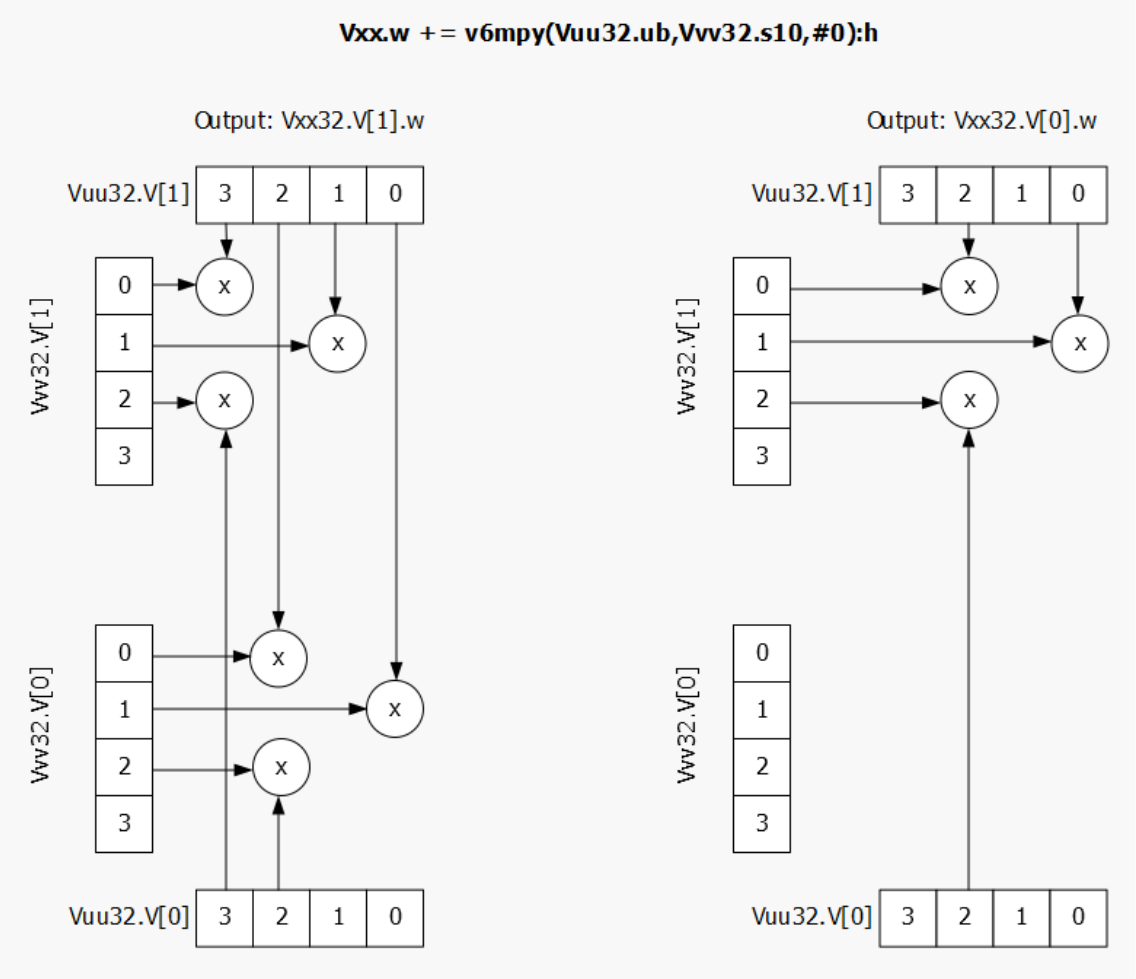
Accumulation is not show to simplify the diagrams, but all multiplies are assumed to reduce and accumulate with the corresponding output.

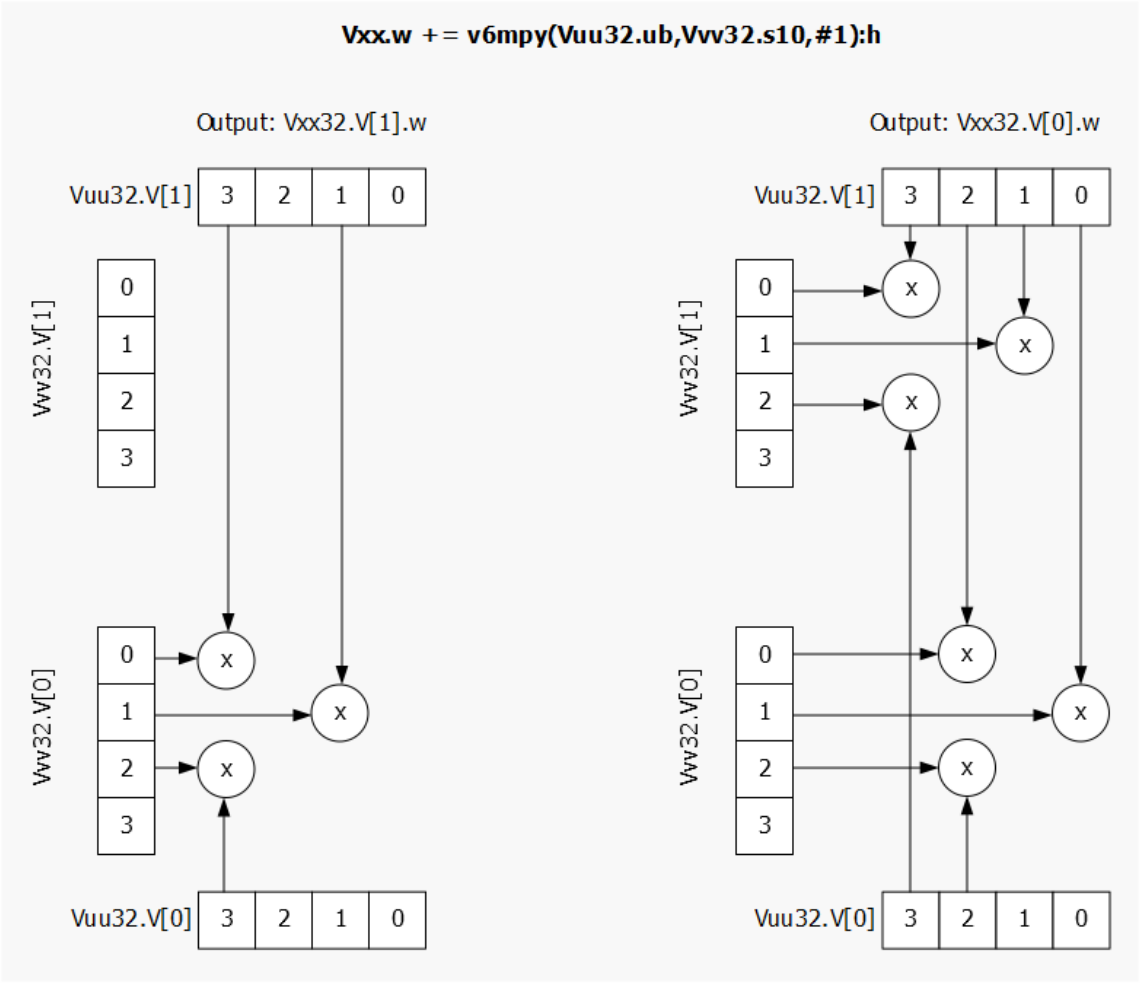


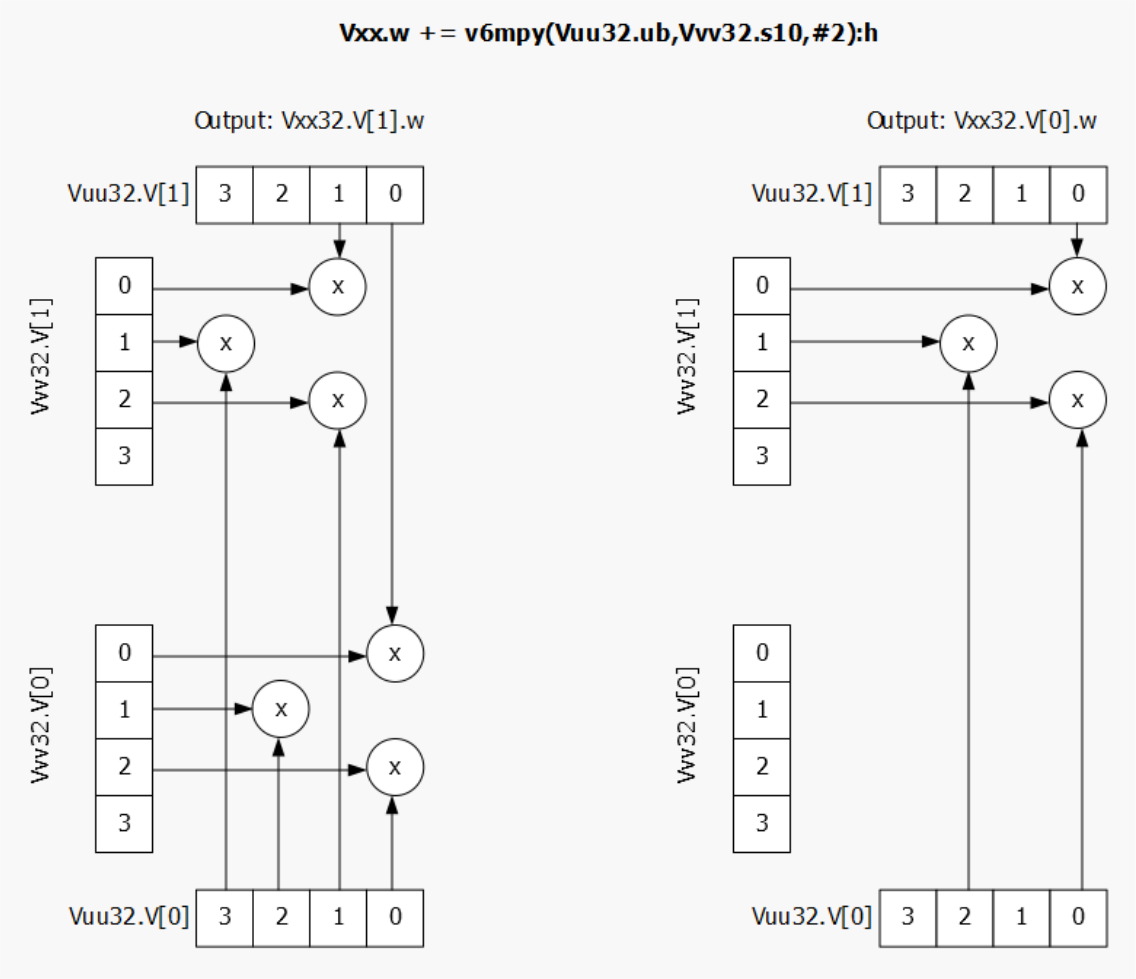

Vxx.w += v6mpy(Vuu32.ub,Vvv32.s10,#1):v
Output: Vxx32.V[1].w
Output: Vxx32.V[0].w
Vuu32.V[1]
Vuu32.V[0]
Vvv32.V[1]
Vvv32.V[0]
3
2
1
0
x

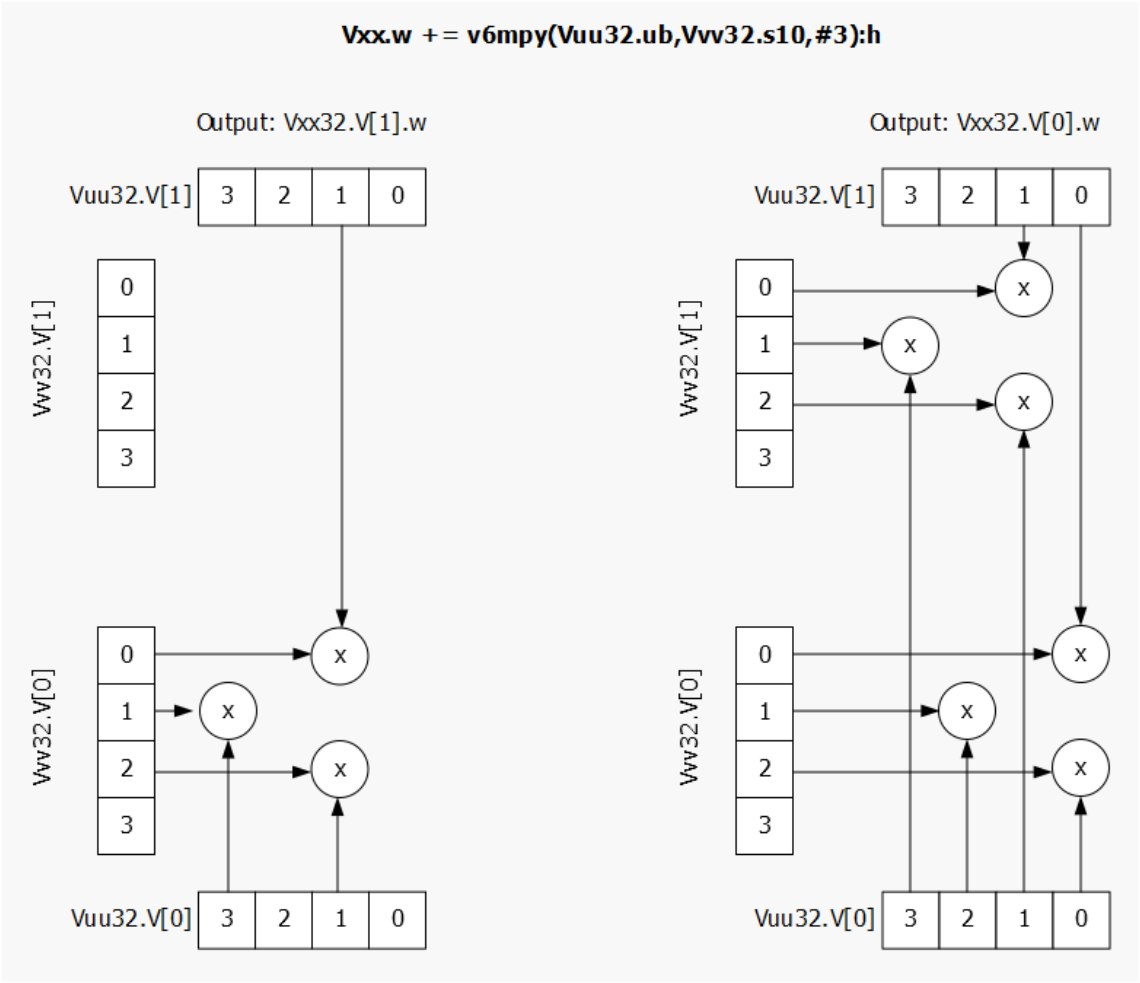

Vxx.w += v6mpy(Vuu32.ub,Vv32.s10,#2):v
Output: Vxx32.V[1].w
Output: Vxx32.V[0].w
Vuu32.V[1]
Vuu32.V[0]
Vv32.V[1]
Vv32.V[0]
3
2
1
0
x


Vxx.w += v6mpy(Vuu32.ub,Vvv32.s10,#3):v
Output: Vxx32.V[1].w
Output: Vxx32.V[0].w
Vuu32.V[1]
Vuu32.V[0]
Vvv32.V[1]
Vvv32.V[0]
3
2
1
0
x


Vxx.w += v6mpy(Vuu32.ub,Vv32.s10,#0):h
Output: Vxx32.V[1].w
Output: Vxx32.V[0].w
Vuu32.V[1]
Vuu32.V[0]
Vv32.V[1]
Vv32.V[0]
3
2
1
0
x


Vxx.w += v6mpy(Vuu32.ub,Vvv32.s10,#1):h
Output: Vxx32.V[1].w
Output: Vxx32.V[0].w
Vuu32.V[1]
Vuu32.V[0]
Vvv32.V[1]
Vvv32.V[0]
3
2
1
0
x


Vxx.w += v6mpy(Vuu32.ub,Vvv32.s10,#2):h
Output: Vxx32.V[1].w
Output: Vxx32.V[0].w
Vuu32.V[1]
Vuu32.V[0]
Vvv32.V[1]
Vvv32.V[0]
3 2 1 0
0 1 2 3
x


Vxx.w += v6mpy(Vuu32.ub,Vvv32.s10,#3):h
Output: Vxx32.V[1].w
Output: Vxx32.V[0].w
Vuu32.V[1]
Vuu32.V[0]
Vvv32.V[1]
Vvv32.V[0]
3
2
1
0
x

Syntax	Behavior

3x3 Multiply for 2x2 Tile instructions

Syntax	Behavior
Vxx.w+=v6mpy(Vuu.ub,Vvv.b,#u2):v	`for (i = 0; i < VELEM(32); i++) { c00=(((((Vvv.v[0].uw[i].ub[3] >> (2 * 0)) & 3) << 8) | Vvv.v[0].uw[i].ub[0]) << 6) >> 6; c01=(((((Vvv.v[0].uw[i].ub[3] >> (2 * 1)) & 3) << 8) | Vvv.v[0].uw[i].ub[1]) << 6) >> 6; c02=(((((Vvv.v[0].uw[i].ub[3] >> (2 * 2)) & 3) << 8) | Vvv.v[0].uw[i].ub[2]) << 6) >> 6; c10=(((((Vvv.v[1].uw[i].ub[3] >> (2 * 0)) & 3) << 8) | Vvv.v[1].uw[i].ub[0]) << 6) >> 6; c11=(((((Vvv.v[1].uw[i].ub[3] >> (2 * 1)) & 3) << 8) | Vvv.v[1].uw[i].ub[1]) << 6) >> 6; c12=(((((Vvv.v[1].uw[i].ub[3] >> (2 * 2)) & 3) << 8) | Vvv.v[1].uw[i].ub[2]) << 6) >> 6; if (u == 0) { Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c10); Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[2] * c11); Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[3] * c12); Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[1] * c00); Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[0] * c01); Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[1] * c02); Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[1] * c10); Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] * c11); Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[1] * c12); } else if (u == 1) { Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c00); Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[2] * c01); Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[3] * c02);`

Syntax	Behavior

Vxx.w+=v6mpy(Vuu.ub,Vvv.b,#u2):h	`for (i = 0; i < VELEM(32); i++) {`
	`  c00=(((((Vvv.v[0].uw[i].ub[3] >> (2 * 0)) & 3) << 8) | Vvv.v[0].uw[i].ub[0]) << 6) >> 6;`
	`  c01=(((((Vvv.v[0].uw[i].ub[3] >> (2 * 1)) & 3) << 8) | Vvv.v[0].uw[i].ub[1]) << 6) >> 6;`
	`  c02=(((((Vvv.v[0].uw[i].ub[3] >> (2 * 2)) & 3) << 8) | Vvv.v[0].uw[i].ub[2]) << 6) >> 6;`
	`  c10=(((((Vvv.v[1].uw[i].ub[3] >> (2 * 0)) & 3) << 8) | Vvv.v[1].uw[i].ub[0]) << 6) >> 6;`
	`  c11=(((((Vvv.v[1].uw[i].ub[3] >> (2 * 1)) & 3) << 8) | Vvv.v[1].uw[i].ub[1]) << 6) >> 6;`
	`  c12=(((((Vvv.v[1].uw[i].ub[3] >> (2 * 2)) & 3) << 8) | Vvv.v[1].uw[i].ub[2]) << 6) >> 6;`
	`  if (u == 0) {`
	`  Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[3] * c10);`
	`  Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[1] * c11);`
	`  Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c12);`
	`  Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[2] * c00);`
	`  Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[0] * c01);`
	`  Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[2] * c02);`
	`  Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[2] * c10);`
	`  Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] * c11);`
	`  Vxx.v[0].w[i] += (Vuu.v[0].uw[i].ub[2] * c12);`
	`  } else if (u == 1) {`
	`  Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[3] * c00);`
	`  Vxx.v[1].w[i] += (Vuu.v[1].uw[i].ub[1] * c01);`
	`  Vxx.v[1].w[i] += (Vuu.v[0].uw[i].ub[3] * c02);`
	`  Vxx.v[0].w[i] += (Vuu.v[1].uw[i].ub[3] * c10);`

Syntax	Behavior

Vdd.w=v6mpy(Vuu.ub,Vvv.b,#u2):v	for (i = 0; i < VELEM(32); i++) {
	c00=(((((Vvv.v[0].uw[i].ub[3] >> (2 * 0)) & 3) << 8) | Vvv.v[0].uw[i].ub[0]) << 6) >> 6;
	c01=(((((Vvv.v[0].uw[i].ub[3] >> (2 * 1)) & 3) << 8) | Vvv.v[0].uw[i].ub[1]) << 6) >> 6;
	c02=(((((Vvv.v[0].uw[i].ub[3] >> (2 * 2)) & 3) << 8) | Vvv.v[0].uw[i].ub[2]) << 6) >> 6;
	c10=(((((Vvv.v[1].uw[i].ub[3] >> (2 * 0)) & 3) << 8) | Vvv.v[1].uw[i].ub[0]) << 6) >> 6;
	c11=(((((Vvv.v[1].uw[i].ub[3] >> (2 * 1)) & 3) << 8) | Vvv.v[1].uw[i].ub[1]) << 6) >> 6;
	c12=(((((Vvv.v[1].uw[i].ub[3] >> (2 * 2)) & 3) << 8) | Vvv.v[1].uw[i].ub[2]) << 6) >> 6;
	if (u == 0) {
	Vdd.v[1].w[i] = (Vuu.v[0].uw[i].ub[3] * c10);
	Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[2] * c11);
	Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[3] * c12);
	Vdd.v[1].w[i] += (Vuu.v[0].uw[i].ub[1] * c00);
	Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[0] * c01);
	Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[1] * c02);
	Vdd.v[0].w[i] = (Vuu.v[0].uw[i].ub[1] * c10);
	Vdd.v[0].w[i] += (Vuu.v[1].uw[i].ub[0] * c11);
	Vdd.v[0].w[i] += (Vuu.v[1].uw[i].ub[1] * c12);
	} else if (u == 1) {
	Vdd.v[1].w[i] = (Vuu.v[0].uw[i].ub[3] * c00);
	Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[2] * c01);
	Vdd.v[1].w[i] += (Vuu.v[1].uw[i].ub[3] * c02);
	Vdd.v[0].w[i] = (Vuu.v[0].uw[i].ub[3] * c10);

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

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

Syntax	Behavior

Class: HVX (slots 2,3)

Note:

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

Encodings

Bitmap

Syntax	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
Vxx.w+=v6mpy(Vuu.ub,Vvv.b,#u2):v0	0	0	1	0	1	0	1	0	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	i	i	x	x	x	x	x
Vxx.w+=v6mpy(Vuu.ub,Vvv.b,#u2):h0	0	0	1	0	1	0	1	0	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	i	i	x	x	x	x	x
Vdd.w=v6mpy(Vuu.ub,Vvv.b,#u2):v0	0	0	1	0	1	0	1	0	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	i	i	d	d	d	d	d
Vdd.w=v6mpy(Vuu.ub,Vvv.b,#u2):h0	0	0	1	0	1	0	1	0	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	i	i	d	d	d	d	d

Intrinsics

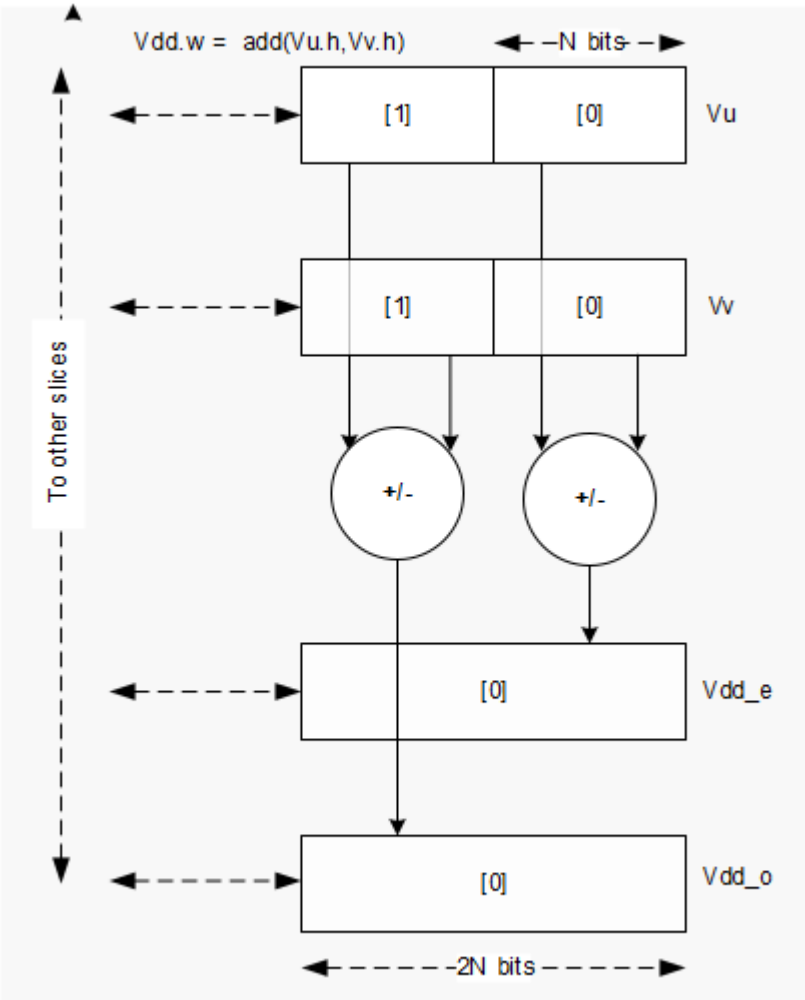
3x3 Multiply for 2x2 Tile intrinsics

Vxx.w+=v6mpy(Vuu.ub,Vvv.b,#u2):v0	HVX_VectorPair Q6_Ww_v6mpyacc_WwWubWbI_v(HVX_VectorPair Vxx, HVX_VectorPair Vuu, HVX_VectorPair Vvv, Word32 Iu2)
Vxx.w+=v6mpy(Vuu.ub,Vvv.b,#u2):h0	HVX_VectorPair Q6_Ww_v6mpyacc_WwWubWbI_h(HVX_VectorPair Vxx, HVX_VectorPair Vuu, HVX_VectorPair Vvv, Word32 Iu2)
Vdd.w=v6mpy(Vuu.ub,Vvv.b,#u2):v0	HVX_VectorPair Q6_Ww_v6mpy_WubWbI_v(HVX_VectorPair Vuu, HVX_VectorPair Vvv, Word32 Iu2)
Vdd.w=v6mpy(Vuu.ub,Vvv.b,#u2):h0	HVX_VectorPair Q6_Ww_v6mpy_WubWbI_h(HVX_VectorPair Vuu, HVX_VectorPair Vvv, Word32 Iu2)

Arithmetic widening

Add or subtract the elements of vector registers Vu and Vv. The resulting elements are double the width of the input size in order 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.



Arithmetic widening instructions

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=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=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]; }`

Syntax	Behavior
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.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]; }`
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]; }`

Class: HVX (slots 2,3)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

Syntax	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
Vdd.h=vadd(Vu.ub,Vv.ub)	0	u	u	0	v	v	v	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=vsub(Vu.ub,Vv.ub)	0	u	u	0	v	v	v	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.w=vadd(Vu.h,Vv.h)	0	u	u	0	v	v	v	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=vsub(Vu.h,Vv.h)	0	u	u	0	v	v	v	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=vadd(Vu.uh,Vv.uh)	0	u	u	0	v	v	v	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=vsub(Vu.uh,Vv.uh)	0	u	u	0	v	v	v	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
Vxx.w+=vadd(Vu.h,Vv.h)	0	u	u	0	v	v	v	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.uh,Vv.uh)	0	u	u	0	v	v	v	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.h+=vadd(Vu.ub,Vv.ub)	0	u	u	0	v	v	v	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

Intrinsics

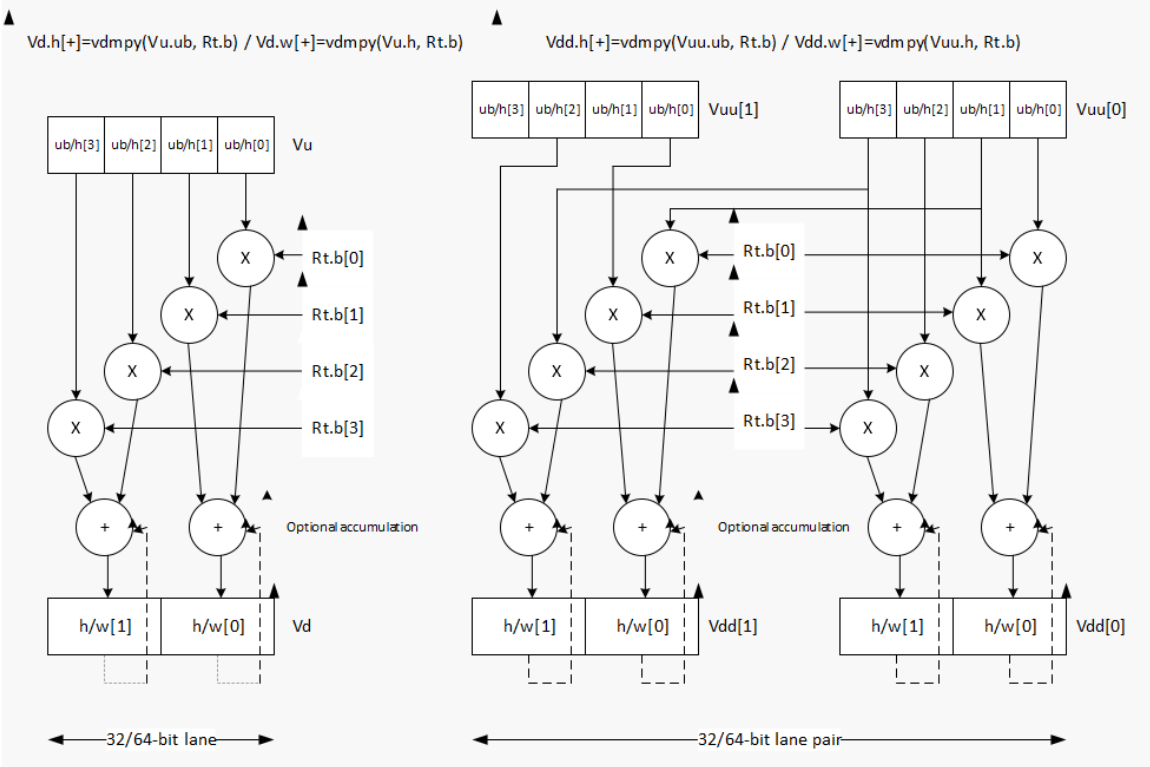
Arithmetic widening intrinsics

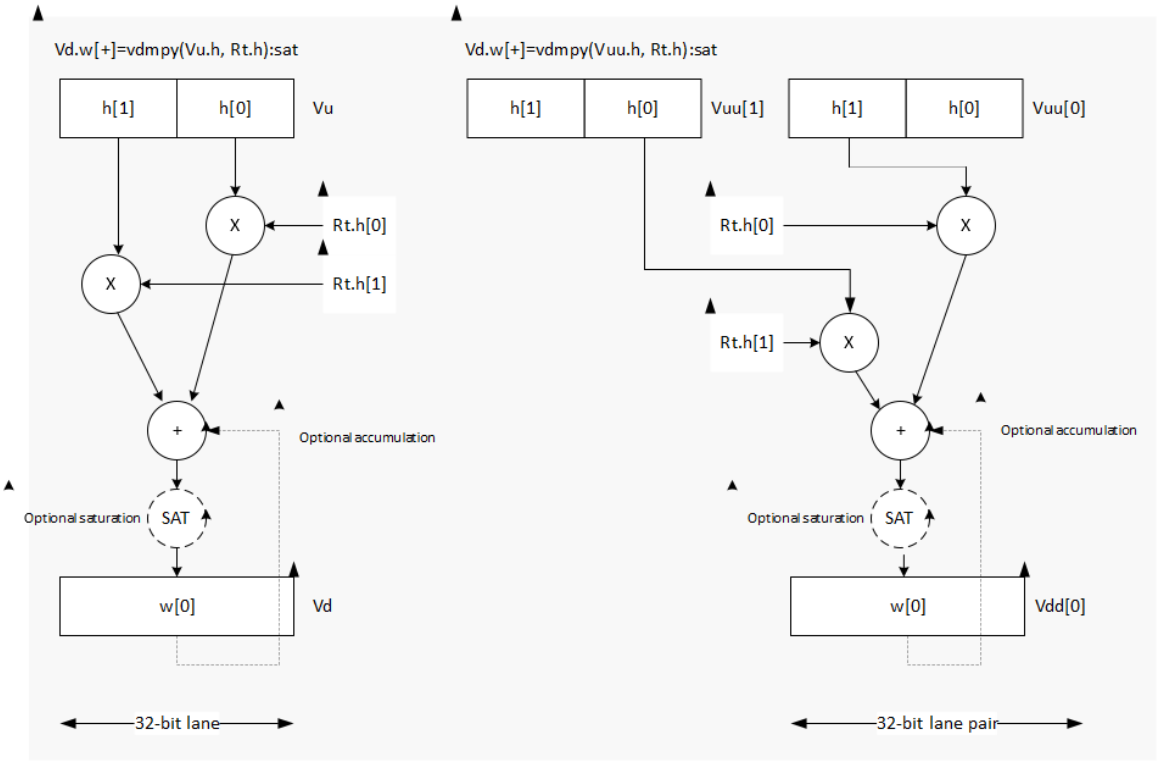
Vdd.h=vadd(Vu.ub,Vv.ub)	HVX_VectorPair Q6_Wh_vadd_VubVub(HVX_Vector Vu, HVX_Vector Vv)
Vdd.h=vsub(Vu.ub,Vv.ub)	HVX_VectorPair Q6_Wh_vsub_VubVub(HVX_Vector Vu, HVX_Vector Vv)
Vdd.w=vadd(Vu.h,Vv.h)	HVX_VectorPair Q6_Ww_vadd_VhVh(HVX_Vector Vu, HVX_Vector Vv)
Vdd.w=vsub(Vu.h,Vv.h)	HVX_VectorPair Q6_Ww_vsub_VhVh(HVX_Vector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

Multiply with 2-wide reduction

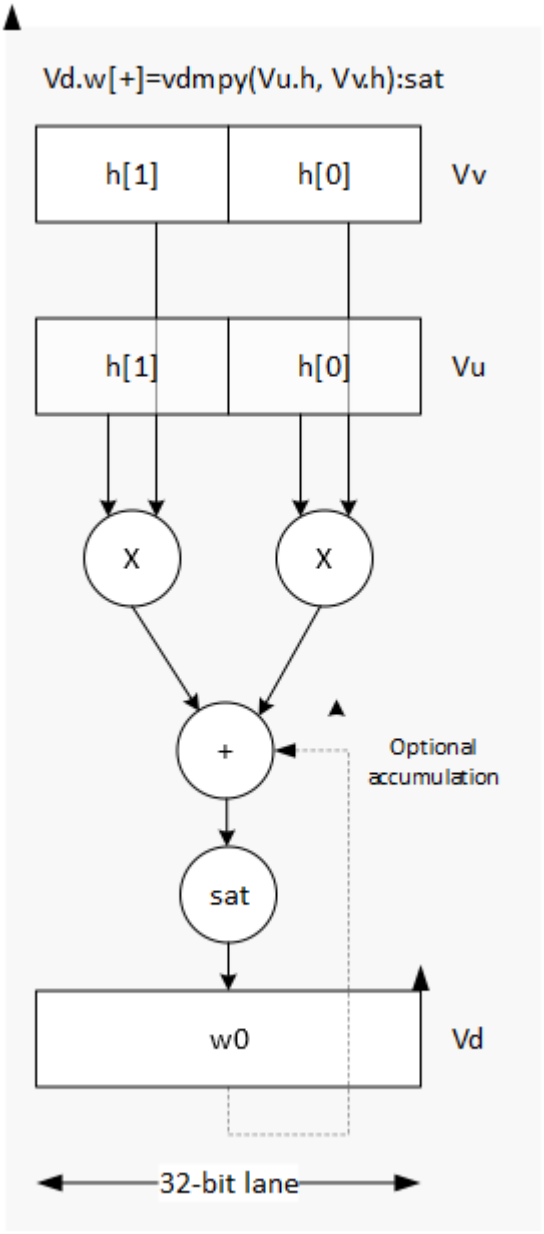
Multiply elements from Vuu by the corresponding elements in the scalar register Rt. The products are added in pairs to yield a by-2 reduction. The products can optionally be accumulated with Vxx, with optional saturation after summation.

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





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


Vd.w[+]=vdmpy(Vu.h, Vv.h):sat
h[1]
h[0]
Vv
h[1]
h[0]
Vu
X
X
+
Optional
accumulation
sat
w0
Vd
32-bit lane

Multiply with 2-wide reduction instructions

Syntax	Behavior
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]); }`
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]); }`
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); }`

Syntax	Behavior
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); }`
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); }`
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); }`

Class: HVX (slots 2,3)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

Syntax	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
Vdd.h=vdmpy(Vuu.ub,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	1	d	d	d	d	d
Vxx.h+=vdmpy(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	1	1	1	x	x	x	x	x
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	0	1	1	d	d	d	d	d
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	0	x	x	x	x	x
Vd.w=vdmpy(Vuu.h,Rt.uh,#0):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
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	0	1	x	x	x	x	x

Intrinsics

Multiply with 2-wide reduction intrinsics

Vdd.h=vdmpy(Vuu.ub,Rt.b)	HVX_VectorPair Q6_Wh_vdmpy_WubRb(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)
Vd.w=vdmpy(Vuu.h,Rt.h):sat	HVX_Vector Q6_Vw_vdmpy_WhRh_sat(HVX_VectorPair Vuu, Word32 Rt)
Vx.w+=vdmpy(Vuu.h,Rt.h):sat	HVX_Vector Q6_Vw_vdmpyacc_VwWhRh_sat(HVX_Vector Vx, 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)
Vx.w+=vdmpy(Vuu.h,Rt.uh,#1):sat	HVX_Vector Q6_Vw_vdmpyacc_VwWhRuh_sat(HVX_Vector Vx, HVX_VectorPair Vuu, Word32 Rt)

Lookup table for piecewise from 64-bit scalar

The vlut4 instruction implements a 4 entry lookup table that is specified in scalar register pair, Rtt.

Lookup table for piecewise from 64-bit scalar instructions

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: HVX (slots 2)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

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

Intrinsics

Lookup table for piecewise from 64-bit scalar intrinsics

Vd.h=vlut4(Vu.uh,Rtt.h)	HVX_Vector Q6_Vh_vlut4_VuhPh(HVX_Vector Vu, Word64 Rtt)

Multiply with piecewise add/sub from 64-bit scalar

Instructions to help nonlinear function calculations

Multiply with piecewise add/sub from 64-bit scalar instructions

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: HVX (slots 2)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

Syntax	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
Vx.h=vmpa(Vx.h,Vu.h,Rtt.h):sat	0	0	0	1	1	0	0	sat	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.uh,Rtt.uh):sat	0	0	0	1	1	0	1	sat	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=vmps(Vx.h,Vu.uh,Rtt.uh):sat	0	0	0	1	1	1	0	sat	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

Intrinsics

Multiply with piecewise add/sub from 64-bit scalar intrinsics

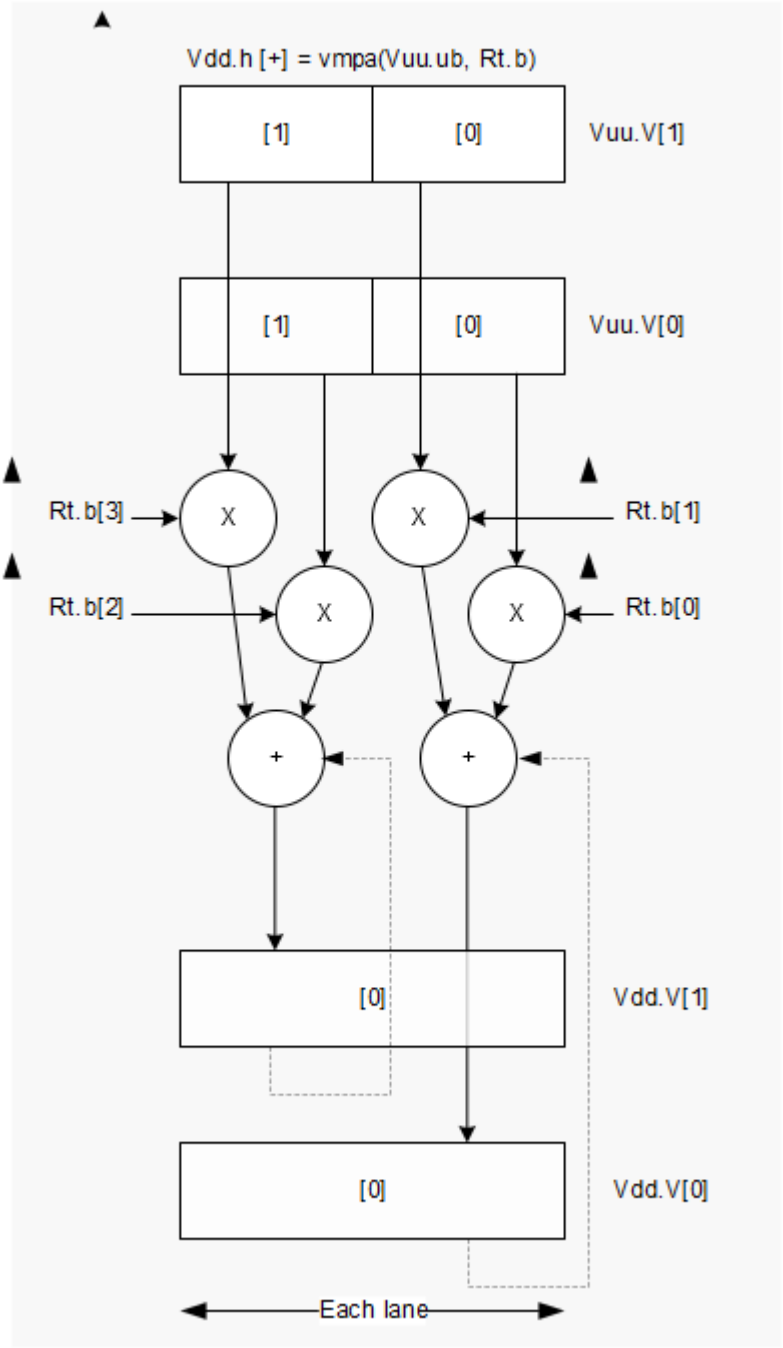
Vx.h=vmpa(Vx.h,Vu.h,Rtt.h):sat	HVX_Vector Q6_Vh_vmpa_VhVhVhPh_sat(HVX_Vector Vx, HVX_Vector Vu, Word64 Rtt)
Vx.h=vmpa(Vx.h,Vu.uh,Rtt.uh):sat	HVX_Vector Q6_Vh_vmpa_VhVhVuhPuh_sat(HVX_Vector Vx, HVX_Vector Vu, Word64 Rtt)
Vx.h=vmps(Vx.h,Vu.uh,Rtt.uh):sat	HVX_Vector Q6_Vh_vmps_VhVhVuhPuh_sat(HVX_Vector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

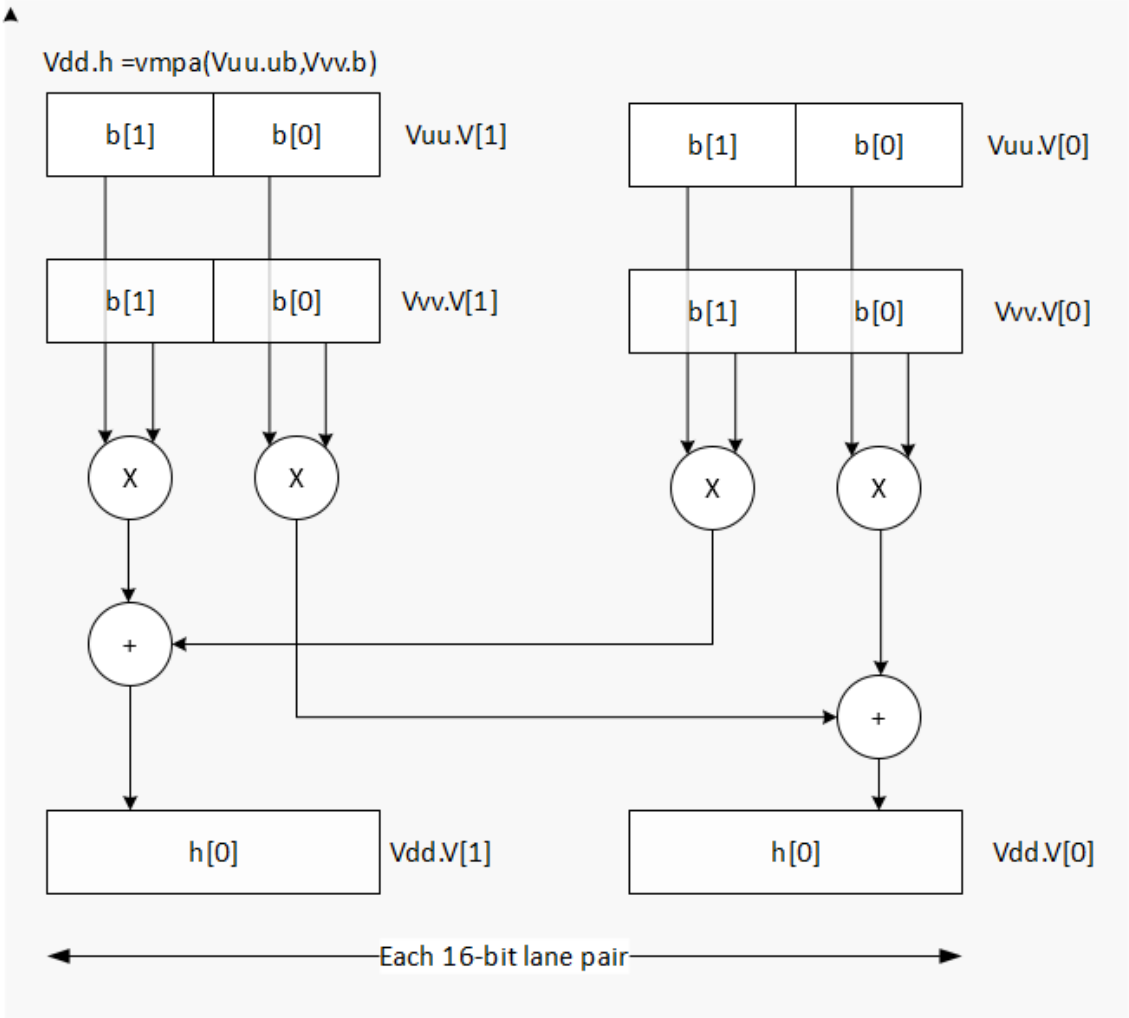
Multiply-add

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


Vdd.h =vmpa(Vuu.ub,Vvv.b)
b[1]
b[0]
Vuu.V[1]
b[1]
b[0]
Vuu.V[0]
b[1]
b[0]
Vvv.V[1]
b[1]
b[0]
Vvv.V[0]
X
X
X
X
+
+
h[0]
Vdd.V[1]
h[0]
Vdd.V[0]
Each 16-bit lane pair

Multiply-add instructions

Syntax	Behavior
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]); }`
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.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]); }`
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]); }`

Syntax	Behavior
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]); }`
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]); }`
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]); }`
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]); }`

Syntax	Behavior
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.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: HVX (slots 2,3)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

Syntax	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
Vdd.h=vmpa(Vuu.ub,Vvv.b)	1	0	0	0	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.ub)	1	0	0	0	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,Rt.b)	1	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
Vxx.h+=vmpa(Vuu.ub,Rt.b)	1	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
Vdd.h=vmpa(Vuu.ub,Rt.ub)	1	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
Vxx.h+=vmpa(Vuu.ub,Rt.ub)	1	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
Vdd.w=vmpa(Vuu.uh,Rt.b)	1	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
Vxx.w+=vmpa(Vuu.uh,Rt.b)	1	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

Syntax	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
Vdd.w=vmpa(Vuu.uh,Rt.b)	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
Vxx.w+=vmpa(Vuu.uh,Rt.b)	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

Intrinsics

Multiply-add intrinsics

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.h=vmpa(Vuu.ub,Rt.b)	HVX_VectorPair Q6_Wh_vmpa_WubRb(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)
Vdd.h=vmpa(Vuu.ub,Rt.ub)	HVX_VectorPair Q6_Wh_vmpa_WubRub(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)
Vdd.w=vmpa(Vuu.h,Rt.b)	HVX_VectorPair Q6_Ww_vmpa_WhRb(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)
Vdd.w=vmpa(Vuu.uh,Rt.b)	HVX_VectorPair Q6_Ww_vmpa_WuhRb(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)

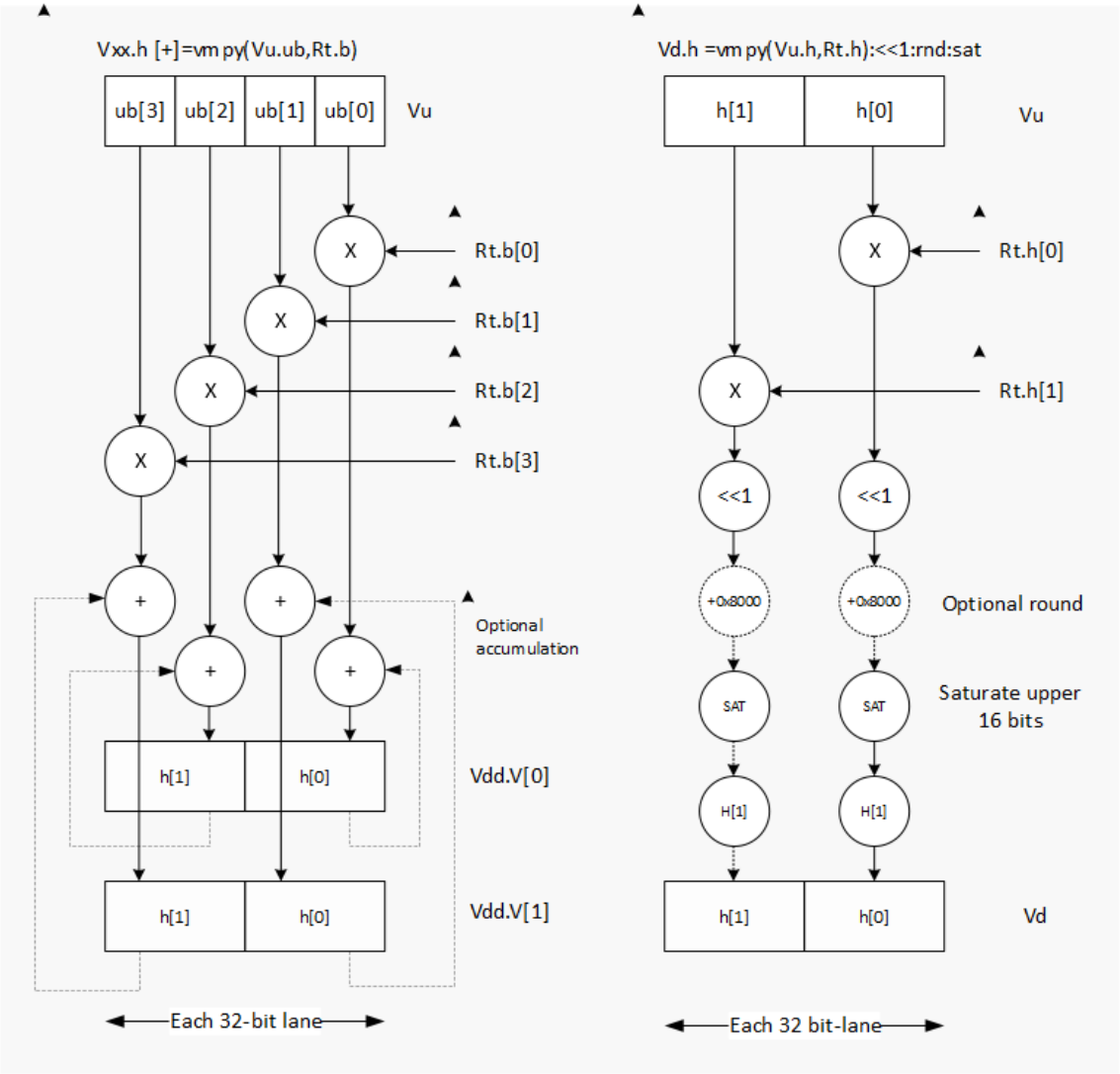
Multiply double resource vector by scalar

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 1, 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,Rt.b)
ub[3]
ub[2]
ub[1]
ub[0]
Vu
X
Rt.b[0]
Rt.b[1]
Rt.b[2]
Rt.b[3]
+
Optional accumulation
h[1]
h[0]
Vdd.V[0]
Vdd.V[1]
Each 32-bit lane
Vd.h =vmpy(Vu.h,Rt.h):<<1:rnd:sat
h[1]
h[0]
Vu
X
Rt.h[0]
Rt.h[1]
<<1
+0x8000
Optional round
SAT
Saturate upper 16 bits
H[1]
Vd
Each 32 bit-lane

Multiply double resource vector by scalar instructions

Syntax	Behavior
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]); }`
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]); }`
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]); }`
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]); }`
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]); }`

Syntax	Behavior
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]); }`
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])); }`
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]); }`
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]); }`

Class: HVX (slots 2,3)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

Syntax	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
Vdd.uh=vmpy(Vu.ub,Rt.ub)	0	0	0	1	1	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
Vxx.uh+=vmpy(Vu.ub,Rt.ub)	0	0	0	1	1	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
Vdd.h=vmpy(Vu.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	0	1	d	d	d	d	d
Vxx.h+=vmpy(Vu.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	0	1	x	x	x	x	x
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	0	d	d	d	d	d
Vxx.w+=vmpy(Vu.h,Rt.h)	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):sat	0	0	0	1	1	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
Vdd.uw=vmpy(Vu.uh,Rt.uh)	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
Vxx.uw+=vmpy(Vu.uh,Rt.uh)	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

Intrinsics

Multiply double resource vector by scalar intrinsics

Vdd.uh=vmpy(Vu.ub,Rt.ub)	HVX_VectorPair Q6_Wuh_vmpy_VubRub(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)
Vdd.h=vmpy(Vu.ub,Rt.b)	HVX_VectorPair Q6_Wh_vmpy_VubRb(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)
Vdd.w=vmpy(Vu.h,Rt.h)	HVX_VectorPair Q6_Ww_vmpy_VhRh(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)
Vdd.uw=vmpy(Vu.uh,Rt.uh)	HVX_VectorPair Q6_Wuw_vmpy_VuhRuh(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)

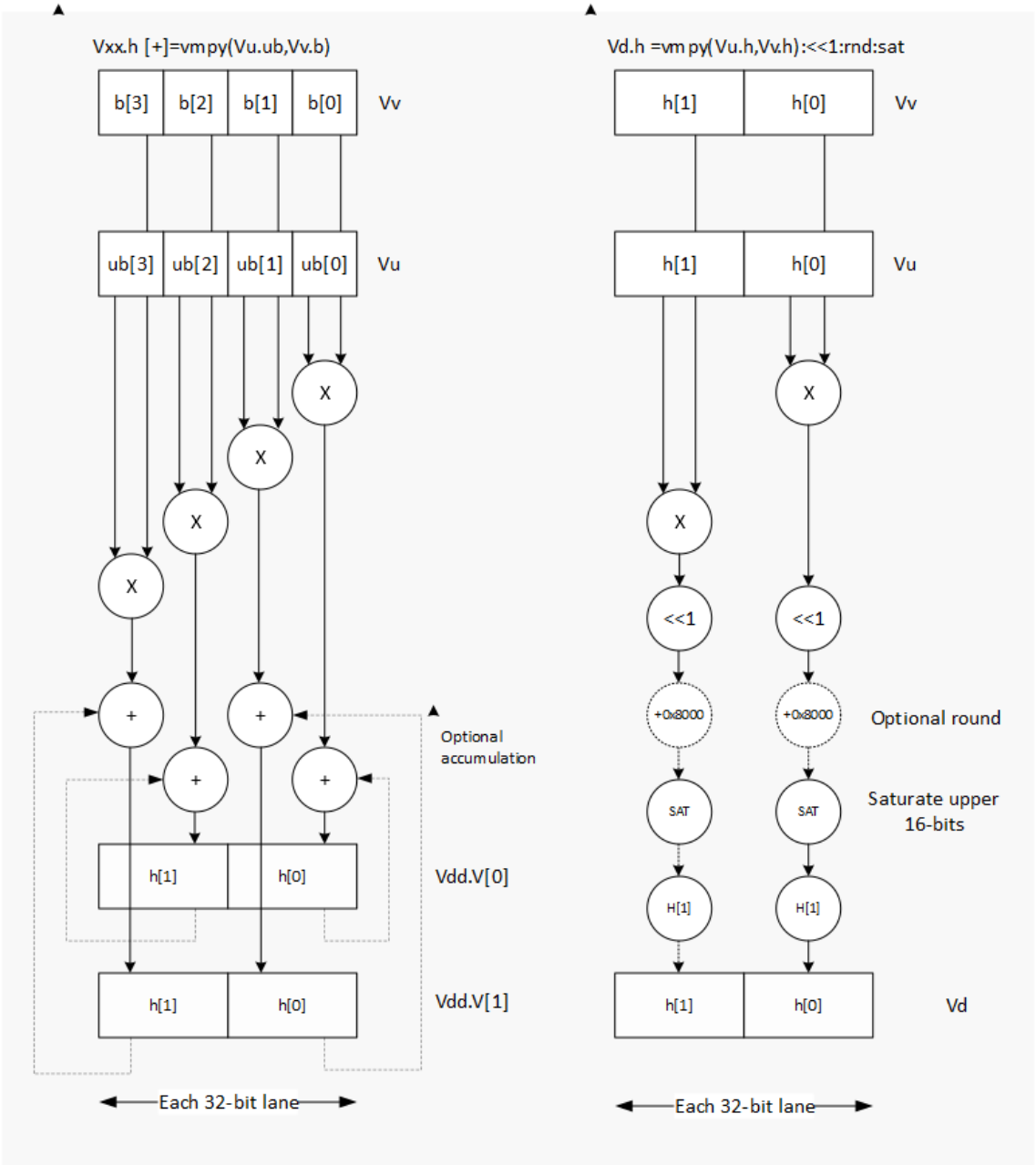
Multiply double resource vector by vector

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 1, saturating the product to 32 bits, and placing the upper 16 bits in the output.


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

Multiply double resource vector by vector instructions

Syntax	Behavior
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]); }`
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]); }`
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]); }`
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]); }`
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]); }`

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]); }`
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]); }`
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]); }`
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]); }`
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]); }`

Syntax	Behavior
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.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: HVX (slots 2,3)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

Syntax	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
Vdd.h=vmpy(Vu.b,Vv.b)	1	1	0	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
Vxx.h+=vmpy(Vu.b,Vv.b)	1	1	0	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
Vdd.uh=vmpy(Vu.ub,Vv.ub)	1	1	0	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
Vxx.uh+=vmpy(Vu.ub,Vv.ub)	1	1	0	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
Vdd.h=vmpy(Vu.ub,Vv.b)	1	1	0	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
Vxx.h+=vmpy(Vu.ub,Vv.b)	1	1	0	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
Vdd.w=vmpy(Vu.h,Vv.h)	1	1	0	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
Vxx.w+=vmpy(Vu.h,Vv.h)	1	1	0	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
Vdd.uw=vmpy(Vu.uh,Vv.uh)	1	1	0	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
Vxx.uw+=vmpy(Vu.uh,Vv.uh)	1	1	0	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
Vdd.w=vmpy(Vu.h,Vv.uh)	1	1	0	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
Vxx.w+=vmpy(Vu.h,Vv.uh)	1	1	0	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

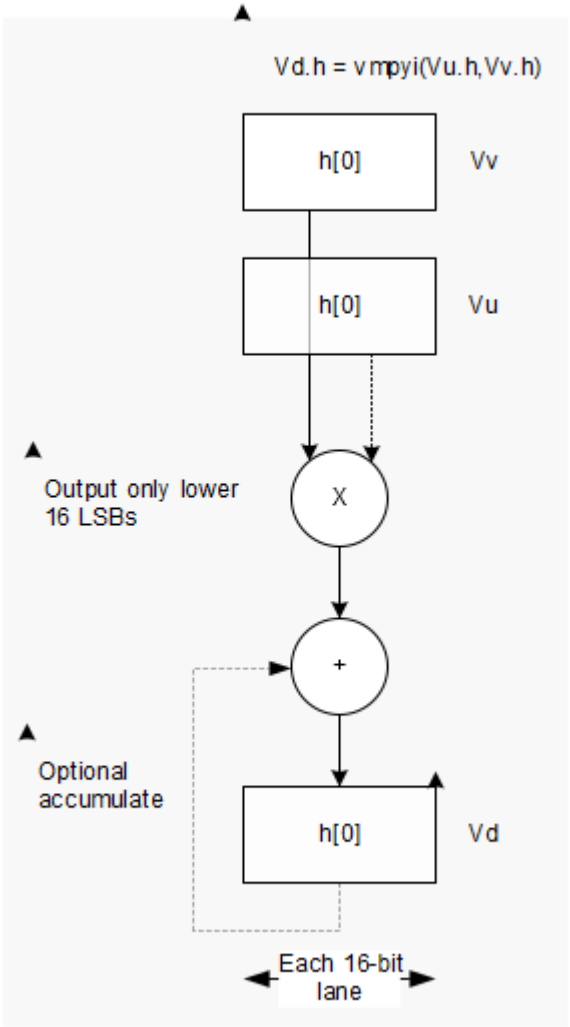
Intrinsics

Multiply double resource vector by vector intrinsics

Vdd.h=vmpy(Vu.b,Vv.b)	HVX_VectorPair Q6_Wh_vmpy_VbVb(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)
Vdd.uh=vmpy(Vu.ub,Vv.ub)	HVX_VectorPair Q6_Wuh_vmpy_VubVub(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)
Vdd.h=vmpy(Vu.ub,Vv.b)	HVX_VectorPair Q6_Wh_vmpy_VubVb(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)
Vdd.w=vmpy(Vu.h,Vv.h)	HVX_VectorPair Q6_Ww_vmpy_VhVh(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)
Vdd.uw=vmpy(Vu.uh,Vv.uh)	HVX_VectorPair Q6_Wuw_vmpy_VuhVuh(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)
Vdd.w=vmpy(Vu.h,Vv.uh)	HVX_VectorPair Q6_Ww_vmpy_VhVuh(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)

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.



Integer multiply - vector by vector instructions

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: HVX (slots 2,3)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

Syntax	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
Vd.h=vmpyi(Vu.h,Vv.h)	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
Vx.h+=vmpyi(Vu.h,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	0	0	x	x	x	x	x

Intrinsics

Integer multiply - vector by vector 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)

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.

Integer Multiply (32x16) instructions

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: HVX (slots 2,3)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

Syntax	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
Vd.w=vmpyie(Vu.w,Vv.uh)	0	0	0	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=vmpyio(Vu.w,Vv.h)	0	0	0	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

Syntax	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
Vx.w+=vmpyie(Vu.w,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	0	0	0	x	x	x	x	x
Vx.w+=vmpyie(Vu.w,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	1	0	1	x	x	x	x	x

Intrinsics

Integer Multiply (32x16) 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)

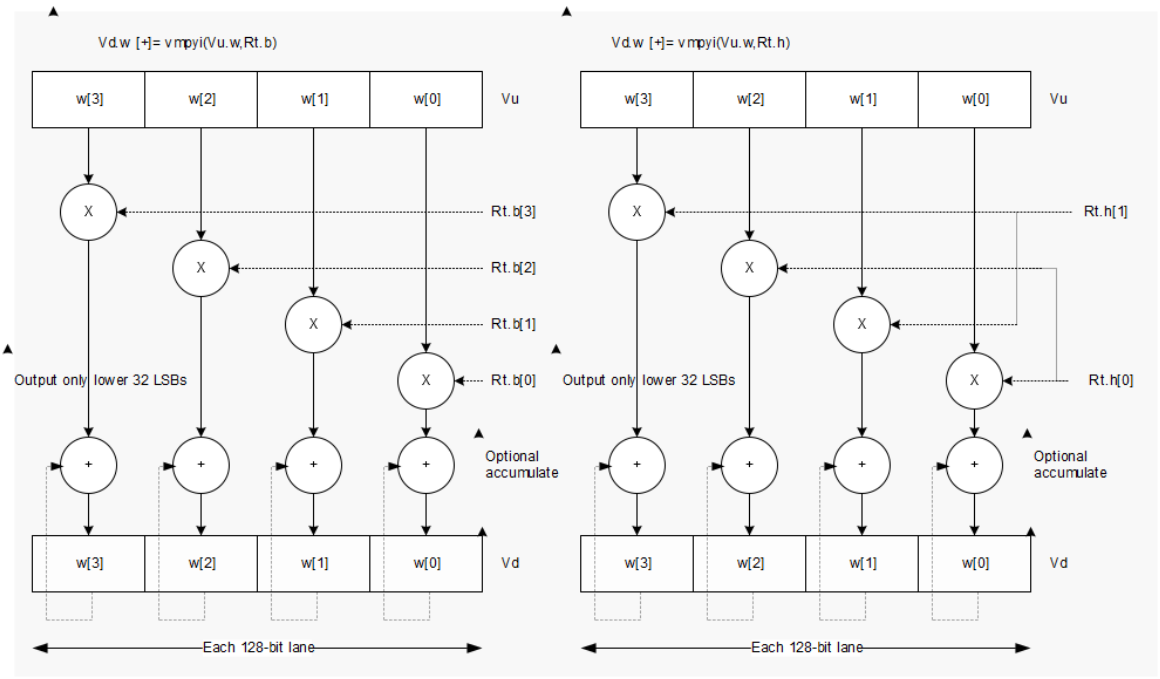
Integer multiply accumulate even/odd

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.



Integer multiply accumulate even/odd instructions

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: HVX (slots 2,3)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

Syntax	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	8	7	6	5	4	3	2	1	0
Vd.w=vmpyi(Vu.w,Rt.h)	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			
Vx.w+=vmpyi(Vu.w,Rt.h)	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			

Intrinsics

Integer multiply accumulate even/odd 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)

HVX floating point multiply - half-precision

HVX floating point half-precision (16-bit) multiply.

Multiplies are vector element by vector element or vector element by scalar value.

A scalar input (Rt) represents 2 half-precision floating point values that is multiplied to each half-precision vector element in Vu.

HVX floating point multiply - half-precision instructions

Syntax	Behavior

Vd.qf16=vmpy(Vu.qf16,Vv.qf16)	`for (i = 0; i < VELEM(16); i++) {`
	`  u = parse_qf_to_unfloat(Vu.qf16[i]);`
	`  v = parse_qf_to_unfloat(Vv.qf16[i]);`
	`  Vd.exp = u.exp + v.exp;`
	`  Vd.sig = u.sig * v.sig;`
	`  if(Vd == 0) Vd.exp = E_MIN_EXTQF16;`
	`}`

Vd.qf16=vmpy(Vu.hf,Vv.hf)	`for (i = 0; i < VELEM(16); i++) {`
	`  u = Vu.hf[i];`
	`  v = Vv.hf[i];`
	`  Vd.exp = u.exp + v.exp;`
	`  Vd.sig = u.sig * v.sig;`
	`  if(Vd == 0) Vd.exp = E_MIN_EXTQF16;`
	`}`

Vd.qf16=vmpy(Vu.qf16,Vv.hf)	`for (i = 0; i < VELEM(16); i++) {`
	`  u = parse_qf_to_unfloat(Vu.qf16[i]);`
	`  v = Vv.hf[i];`
	`  Vd.exp = u.exp + v.exp;`
	`  Vd.sig = u.sig * v.sig;`
	`  if(Vd == 0) Vd.exp = E_MIN_EXTQF16;`
	`}`

Vd.qf16=vmpy(Vu.qf16,Rt.hf)	`for (i = 0; i < VELEM(16); i++) {`
	`  u = parse_qf_to_unfloat(Vu.qf16[i]);`
	`  v = (Rt >> ((i%2)*16)) & 0xFFFF;`
	`  Vd.exp = u.exp + v.exp;`
	`  Vd.sig = u.sig * v.sig;`
	`  if(Vd == 0) Vd.exp = E_MIN_EXTQF16;`
	`}`

Syntax	Behavior

Vd.qf16=vmpy(Vu.hf,Rt.hf)	`for (i = 0; i < VELEM(16); i++) {`
	`  u = Vu.hf[i];`
	`  v = (Rt >> ((i%2)*16)) & 0xFFFF;`
	`  Vd.exp = u.exp + v.exp;`
	`  Vd.sig = u.sig * v.sig;`
	`  if(Vd == 0) Vd.exp = E_MIN_EXTQF16;`
	`}`

Class: HVX (slots 2,3)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

Syntax	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
Vd.qf16=vmpy(Vu.qf16,Vv.qf16)	1	1	1	0	1	0	1	1	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	d	d	d	d	d
Vd.qf16=vmpy(Vu.hf,Vv.hf)	1	1	1	0	1	0	1	1	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	d	d	d	d	d
Vd.qf16=vmpy(Vu.qf16,Vv.hf)	1	1	1	0	1	0	1	1	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	1	d	d	d	d	d
Vd.qf16=vmpy(Vu.qf16,Rt.hf)	1	1	1	0	1	0	1	0	0	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	0	1	0	d	d	d	d	d
Vd.qf16=vmpy(Vu.hf,Rt.hf)	1	1	1	0	1	0	0	1	0	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	0	1	1	d	d	d	d	d

Intrinsics

HVX floating point multiply - half-precision intrinsics

Vd.qf16=vmpy(Vu.qf16,Vv.qf16)	HVX_Vector Q6_Vqf16_vmpy_Vqf16Vqf16(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf16=vmpy(Vu.hf,Vv.hf)	HVX_Vector Q6_Vqf16_vmpy_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf16=vmpy(Vu.qf16,Vv.hf)	HVX_Vector Q6_Vqf16_vmpy_Vqf16Vhf(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf16=vmpy(Vu.qf16,Rt.hf)	HVX_Vector Q6_Vqf16_vmpy_Vqf16Rhf(HVX_Vector Vu, Word32 Rt)
Vd.qf16=vmpy(Vu.hf,Rt.hf)	HVX_Vector Q6_Vqf16_vmpy_VhfRhf(HVX_Vector Vu, Word32 Rt)

HVX floating point multiply - single-precision

HVX floating point single-precision (32-bit) multiply.

Multiplies are vector element by vector element or vector element by scalar value.

A scalar input (Rt) is a single-precision floating point value that is multiplied to each single-precision vector element in Vu.

HVX floating point multiply - single-precision instructions

Syntax	Behavior

Vd.qf32=vmpy(Vu.qf32,Vv.qf32)	`for (i = 0; i < VELEM(32); i++) {`
	`  u = parse_qf_to_unfloat(Vu.qf32[i]);`
	`  v = parse_qf_to_unfloat(Vv.qf32[i]);`
	`  Vd.exp = u.exp + v.exp;`
	`  Vd.sig = u.sig * v.sig;`
	`  if(Vd == 0) Vd.exp = E_MIN_EXTQF32;`
	`}`

Syntax	Behavior

Vd.qf32=vmpy(Vu.sf,Vv.sf)	`for (i = 0; i < VELEM(32); i++) {`
	`  u = Vu.sf[i];`
	`  v = Vv.sf[i];`
	`  Vd.exp = u.exp + v.exp;`
	`  Vd.sig = u.sig * v.sig;`
	`  if(Vd == 0) Vd.exp = E_MIN_EXTQF32;`
	`}`

Vd.qf32=vmpy(Vu.sf,Rt.sf)	`for (i = 0; i < VELEM(32); i++) {`
	`  u = Vu.sf[i];`
	`  v = Rt;`
	`  Vd.exp = u.exp + v.exp;`
	`  Vd.sig = u.sig * v.sig;`
	`  if(Vd == 0) Vd.exp = E_MIN_EXTQF32;`
	`}`

Class: HVX (slots 2,3)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

Syntax	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
Vd.qf32=vmpy(Vu.qf32,Vv.qf32)	1	1	0	1	0	1	0	1	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	d	d	d	d	d
Vd.qf32=vmpy(Vu.sf,Vv.sf)	1	1	0	1	0	1	0	1	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	d	d	d	d	d
Vd.qf32=vmpy(Vu.sf,Rt.sf)	1	1	0	1	0	1	0	1	0	0	0	t	t	t	t	t	P	P	1	u	u	u	u	u	0	0	1	d	d	d	d	d

Intrinsics

HVX floating point multiply - single-precision intrinsics

Vd.qf32=vmpy(Vu.qf32,Vv.qf32)	HVX_Vector Q6_Vqf32_vmpy_Vqf32Vqf32(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf32=vmpy(Vu.sf,Vv.sf)	HVX_Vector Q6_Vqf32_vmpy_VsfVsf(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf32=vmpy(Vu.sf,Rt.sf)	HVX_Vector Q6_Vqf32_vmpy_VsfRsf(HVX_Vector Vu, Word32 Rt)

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.</p>

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.</p> The rounding fractional multiply:

```
vectorize( sat_32(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( sat_32(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 32bits of products are in V2 and the upper 32bits in
V3. If only vmpye is performed the result will be a
48bit product of 32signed x 16bit unsigned asserted into the upper
48bits of Vdd. If vmpyo only is performed assuming
Vxx = #0, the result will be a 32signed x 16signed product asserted
into the upper 48bits of Vxx.
```

Multiply (32x16) instructions

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:sat	`for (i = 0; i < VELEM(32); i++) { Vd.w[i] = sat_32((((Vu.w[i] * Vv.w[i].h[1]) >> 14) + 0) >> 1)); }`
Vd.w=vmpyo(Vu.w,Vv.h):<<1:rnd:sat	`for (i = 0; i < VELEM(32); i++) { Vd.w[i] = sat_32((((Vu.w[i] * Vv.w[i].h[1]) >> 14) + 1) >> 1)); }`

Syntax	Behavior
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; }`
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; }`
Vx.w+=vmpyo(Vu.w,Vv.h):<<1:sat:shift	`for (i = 0; i < VELEM(32); i++) { Vx.w[i] = sat_32((((Vx.w[i] + (Vu.w[i] * Vv.w[i].h[1])) >> 14) + 0) >> 1)); }`
Vx.w+=vmpyo(Vu.w,Vv.h):<<1:rnd:sat:shift	`for (i = 0; i < VELEM(32); i++) { Vx.w[i] = sat_32((((Vx.w[i] + (Vu.w[i] * Vv.w[i].h[1])) >> 14) + 1) >> 1)); }`

Class: HVX (slots 2,3)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

Syntax	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
Vd.w=vmpye(Vu.w,Vv.uh)	0	v	v	w	0	v	v	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=vmpyo(Vu.w,Vv.h):<<1:sat	0	v	v	w	0	v	v	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:rnd:sat	0	v	v	w	0	v	v	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
Vdd=vmpye(Vu.w,Vv.uh)	1	v	v	w	0	v	v	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
Vxx+=vmpyo(Vu.w,Vv.h)	1	v	v	w	0	v	v	1	1	1	1	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+=vmpyo(Vu.w,Vv.h):<<1:sat:shift	1	v	v	w	0	v	v	1	1	1	1	0	1	0	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:rnd:sat:shift	1	v	v	w	0	v	v	1	1	1	1	0	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	x	x	x	x	x

Intrinsics

Multiply (32x16) 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:sat	HVX_Vector Q6_Vw_vmpyo_VwVh_s1_sat(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)
Vdd=vmpye(Vu.w,Vv.uh)	HVX_VectorPair Q6_W_vmpye_VwVuh(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)
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)
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)

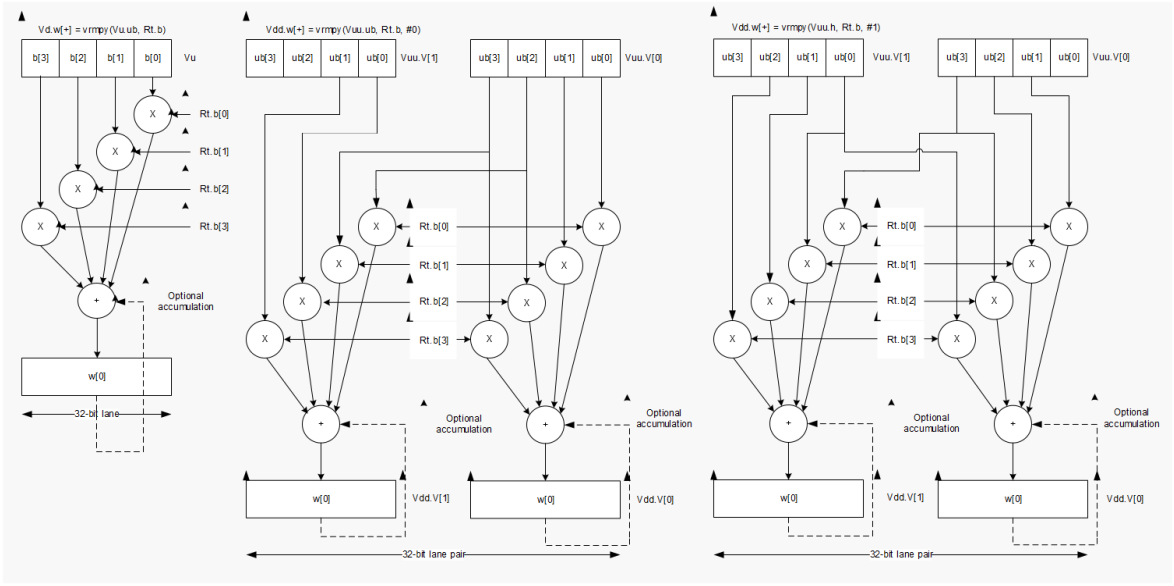
Multiply bytes with 4-wide reduction - vector by scalar

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

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

The operation has two forms: the first performs simple dot product of 4 elements into a single result. The second form takes a 1 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 2 elements and the upper 2 data elements taken from the even register of Vuu. For #u = 1, the even destination takes coefficients rotated by -1 and data element 0

from the odd register of Vuu. The odd destination uses coefficients rotated by -1 and takes data element 3 from the even register of Vuu.



Multiply bytes with 4-wide reduction - vector by scalar instructions

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]); }`
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
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.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]); }`

Class: HVX (slots 2,3)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

Syntax	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
Vdd.uw=vrmpy(Vuu.ub,Rt.ub,#u1)	0	0	1	0	1	0	1	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
Vxx.uw+=vrmpy(Vuu.ub,Rt.ub,#u1)	0	0	1	0	1	0	1	0	1	1	1	t	t	t	t	t	P	P	1	u	u	u	u	u	1	1	i	x	x	x	x	x
Vdd.w=vrmpy(Vuu.ub,Rt.b,#u1)	0	0	1	0	1	0	1	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
Vxx.w+=vrmpy(Vuu.ub,Rt.b,#u1)	0	0	1	0	1	0	1	0	1	0	1	t	t	t	t	t	P	P	1	u	u	u	u	u	1	0	i	x	x	x	x	x

Intrinsics

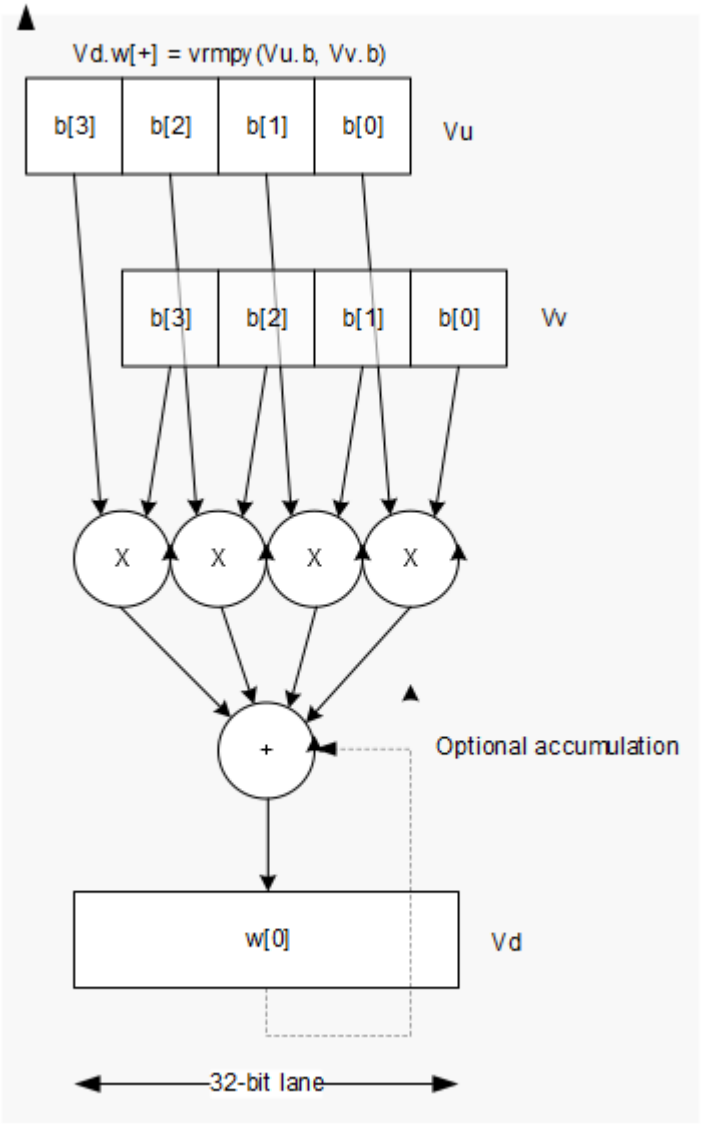
Multiply bytes with 4-wide reduction - vector by scalar intrinsics

Vdd.uw=vrmpy(Vuu.ub,Rt.ub,#u1)	HVX_VectorPair Q6_Wuw_vrmpy_WubRubI(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)
Vdd.w=vrmpy(Vuu.ub,Rt.b,#u1)	HVX_VectorPair Q6_Ww_vrmpy_WubRbI(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)

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

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

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



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

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: HVX (slots 2,3)

Note:

- This instruction uses a HVX multiply resource.

Encodings

Bitmap

Syntax	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
Vx.uw+=vrmpy(Vu.ub,Vv.ub)	0	0	0	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.w+=vrmpy(Vu.b,Vv.b)	0	0	0	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.ub,Vv.b)	0	0	0	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

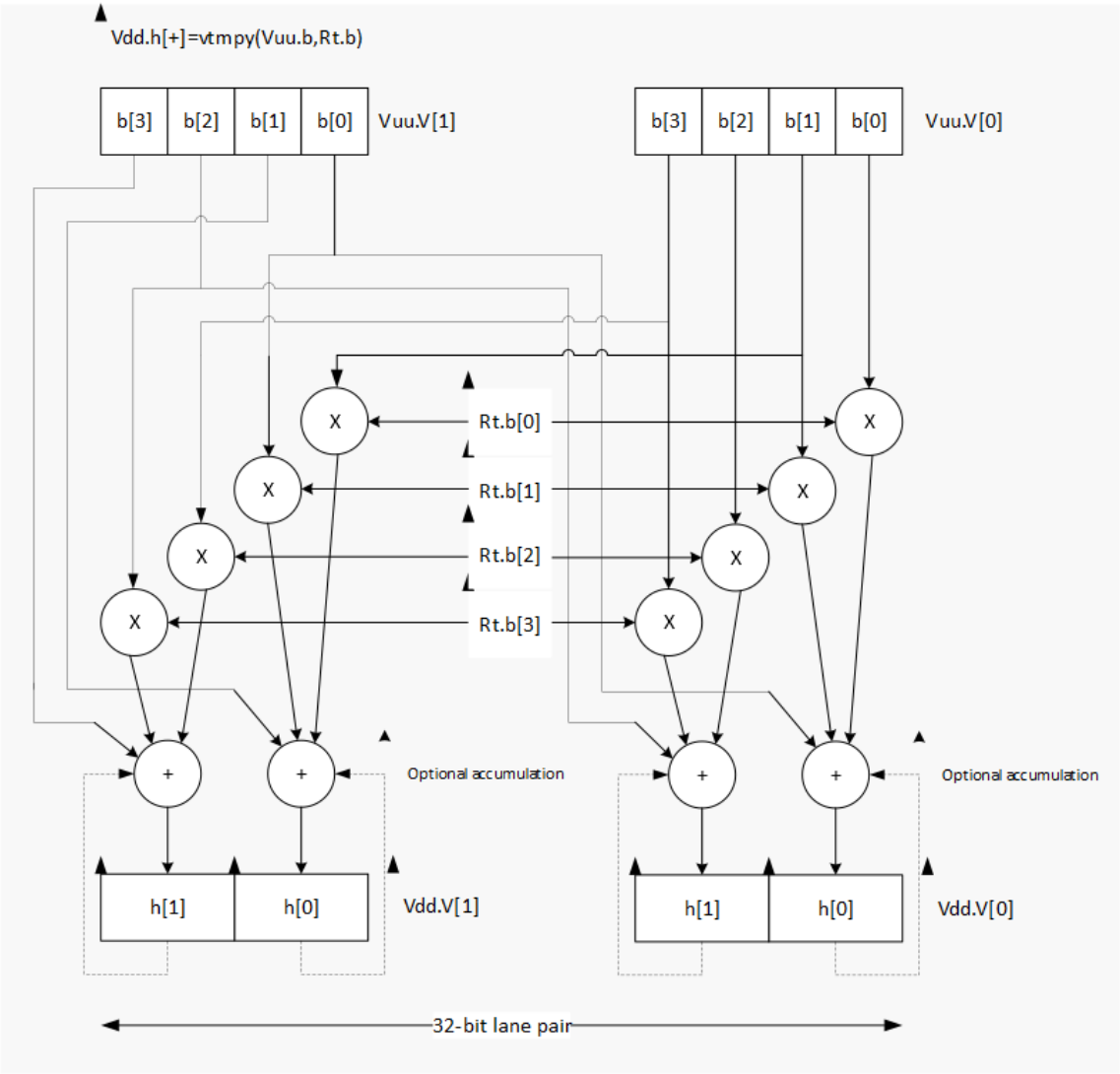
Intrinsics

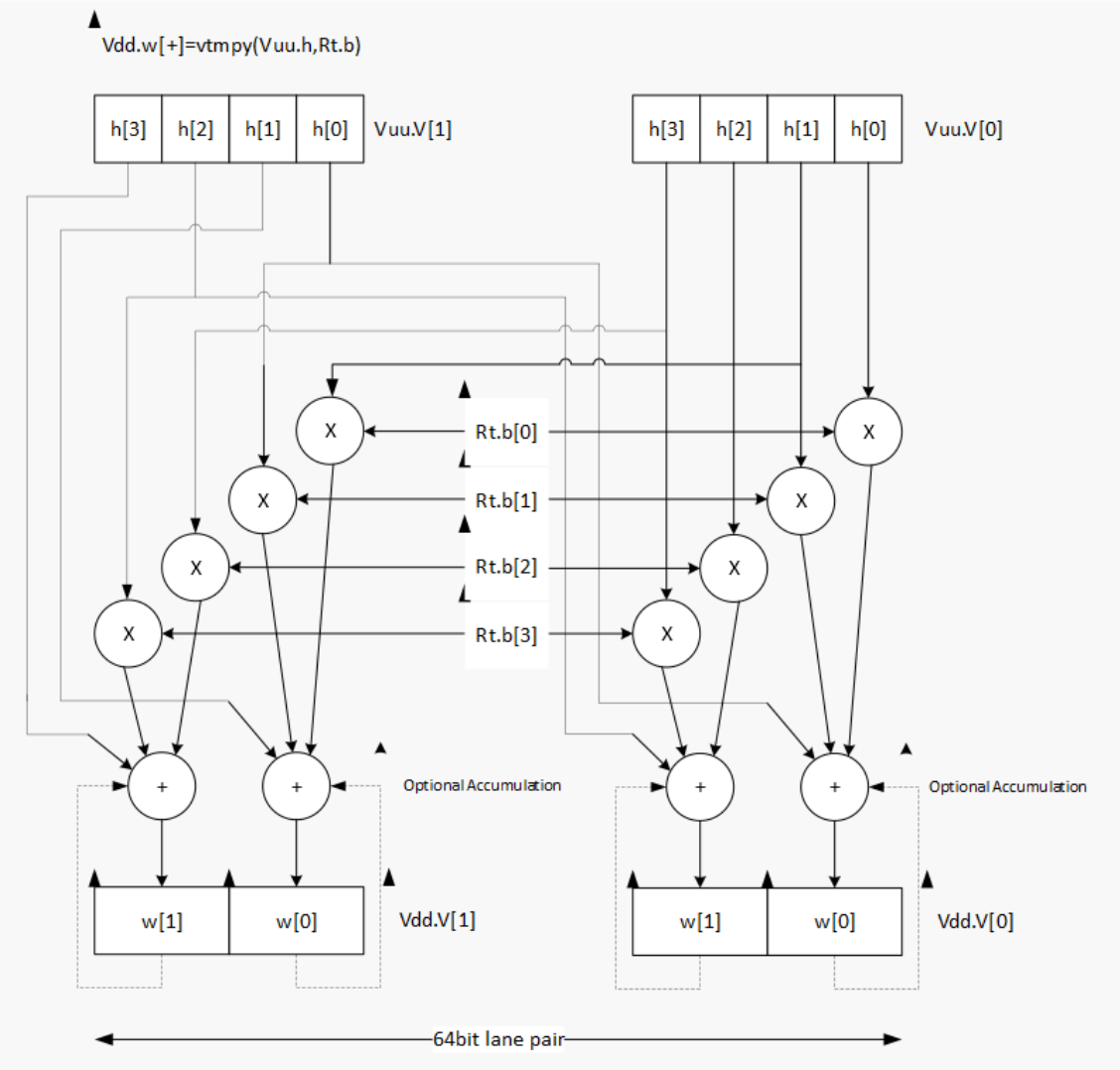
Multiply by byte with accumulate and 4-wide reduction - vector by vector intrinsics

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)

Multiply with 3-wide reduction

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




Vdd.w[+]=vtmpy(Vuu.h,Rt.b)
h[3]
h[2]
h[1]
h[0]
Vuu.V[1]
h[3]
h[2]
h[1]
h[0]
Vuu.V[0]
X
Rt.b[0]
Rt.b[1]
Rt.b[2]
Rt.b[3]
+
Optional Accumulation
w[1]
w[0]
Vdd.V[1]
Vdd.V[0]
64bit lane pair

Multiply with 3-wide reduction instructions

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]; }`
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]; }`

Syntax	Behavior
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]; }`
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]; }`

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.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: HVX (slots 2,3)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

Syntax	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
Vdd.h=vtmpy(Vuu.b,Rt.b)	0	u	u	0	1	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
Vxx.h+=vtmpy(Vuu.b,Rt.b)	0	u	u	0	1	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
Vdd.h=vtmpy(Vuu.ub,Rt.b)	0	u	u	0	1	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
Vxx.h+=vtmpy(Vuu.ub,Rt.b)	0	u	u	0	1	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
Vdd.w=vtmpy(Vuu.h,Rt.b)	0	u	u	0	1	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
Vxx.w+=vtmpy(Vuu.h,Rt.b)	0	u	u	0	1	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

Intrinsics

Multiply with 3-wide reduction intrinsics

Vdd.h=vtmpy(Vuu.b,Rt.b)	HVX_VectorPair Q6_Wh_vtmpy_WbRb(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)
Vdd.h=vtmpy(Vuu.ub,Rt.b)	HVX_VectorPair Q6_Wh_vtmpy_WubRb(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)
Vdd.w=vtmpy(Vuu.h,Rt.b)	HVX_VectorPair Q6_Ww_vtmpy_WhRb(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)

HVX floating point multiply - widening

Widening multiply from half-precision (16-bit) to HVX floating point single-precision (32-bit). Multiplies 64 half-precision inputs and produces 64 single-precision results.

HVX floating point multiply - widening instructions

Syntax	Behavior
Vdd.qf32=vmpy(Vu.qf16,Vv.qf16)	`for (i = 0; i < VELEM(32); i++) { u0 = parse_qf_to_unfloat(Vu.qf16[2*i]); u1 = parse_qf_to_unfloat(Vu.qf16[2*i+1]); v0 = parse_qf_to_unfloat(Vv.qf16[2*i]); v1 = parse_qf_to_unfloat(Vv.qf16[2*i+1]); Vdd.v[0].exp = u0.exp + v0.exp; Vdd.v[0].sig = u0.sig * v0.sig; if(Vdd.v[0] == 0) Vdd.v[0].exp = E_MIN_EXTQF32 Vdd.v[1].exp = u1.exp + v1.exp; Vdd.v[1].sig = u1.sig * v1.sig; if(Vdd.v[1] == 0) Vdd.v[1].exp = E_MIN_EXTQF32; }`
Vdd.qf32=vmpy(Vu.hf,Vv.hf)	`for (i = 0; i < VELEM(32); i++) { u0 = Vu.w[i] & 0xFFFF; u1 = (Vu.w[i]>>16) & 0xFFFF; v0 = Vv.w[i] & 0xFFFF; v1 = (Vv.w[i]>>16) & 0xFFFF; Vdd.v[0].exp = u0.exp + v0.exp; Vdd.v[0].sig = u0.sig * v0.sig; if(Vdd.v[0] == 0) Vdd.v[0].exp = E_MIN_EXTQF32 Vdd.v[1].exp = u1.exp + v1.exp; Vdd.v[1].sig = u1.sig * v1.sig; if(Vdd.v[1] == 0) Vdd.v[1].exp = E_MIN_EXTQF32; }`

Syntax	Behavior

Vdd.qf32=vmpy(Vu.qf16,Vv.hf)	`for (i = 0; i < VELEM(32); i++) {`
	`  u0 = parse_qf_to_unfloat(Vu.qf16[2*i]);`
	`  u1 = parse_qf_to_unfloat(Vu.qf16[2*i+1]);`
	`  v0 = Vv.w[i] & 0xFFFF;`
	`  v1 = (Vv.w[i]>>16) & 0xFFFF;`
	`  Vdd.v[0].exp = u0.exp + v0.exp;`
	`  Vdd.v[0].sig = u0.sig * v0.sig;`
	`  if(Vdd.v[0] == 0) Vdd.v[0].exp = E_MIN_EXTQF32 Vdd.v[1].exp = u1.exp + v1.exp;`
	`  Vdd.v[1].sig = u1.sig * v1.sig;`
	`  if(Vdd.v[1] == 0) Vdd.v[1].exp = E_MIN_EXTQF32;`
	`}`

Class: HVX (slots 2,3)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

Syntax	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
Vdd.qf32=vmpy(Vu.qf16,Vv.qf16)	0	0	0	1	1	0	1	1	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	0	d	d	d	d	d
Vdd.qf32=vmpy(Vu.qf16,Vv.hf)	0	0	0	1	1	0	1	1	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	d	d	d	d	d
Vdd.qf32=vmpy(Vu.qf16,Vv.hf)	0	0	0	1	1	0	1	1	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	d	d	d	d	d

Intrinsics

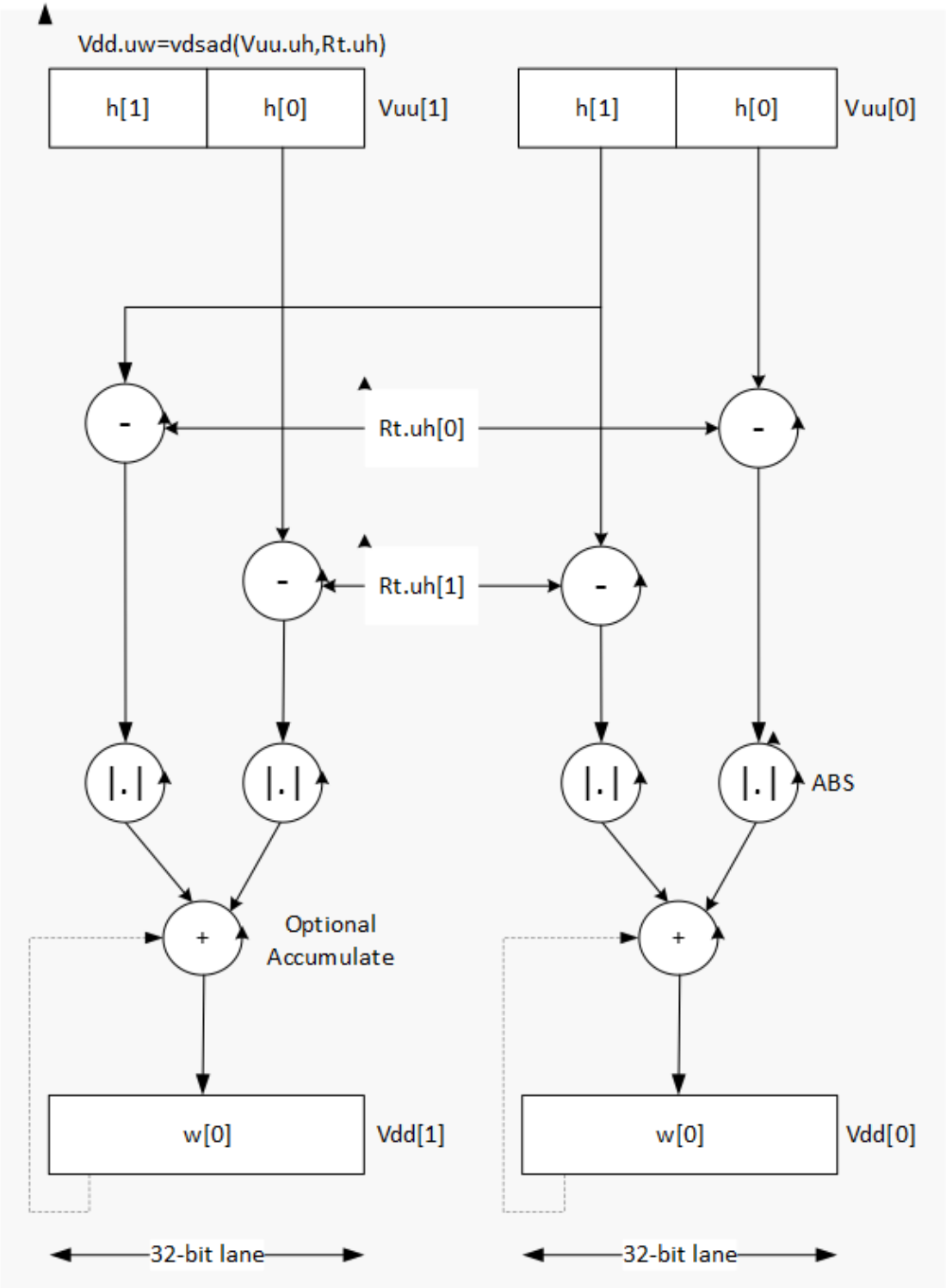
HVX floating point multiply - widening intrinsics

Vdd.qf32=vmpy(Vu.qf16,Vv.qf16)	HVX_VectorPair Q6_Wqf32_vmpy_Vqf16Vqf16(HVX_Vector Vu, HVX_Vector Vv)
Vdd.qf32=vmpy(Vu.hf,Vv.hf)	HVX_VectorPair Q6_Wqf32_vmpy_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)
Vdd.qf32=vmpy(Vu.qf16,Vv.hf)	HVX_VectorPair Q6_Wqf32_vmpy_Vqf16Vhf(HVX_Vector Vu, HVX_Vector Vv)

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]
32-bit lane
32-bit lane

Sum of reduction of absolute differences halfwords instructions

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[1].uw[i].uh[0] - Rt.uh[1]); Vxx.v[1].uw[i] += ABS(Vuu.v[1].uw[i].uh[1] - Rt.uh[0]); }`

Class: HVX (slots 2,3)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

Syntax	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
Vdd.uw=vdsad(Vuu.uh,Rt.uh)	0	0	0	1	1	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
Vxx.uw+=vdsad(Vuu.uh,Rt.uh)	0	0	0	1	1	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

Intrinsics

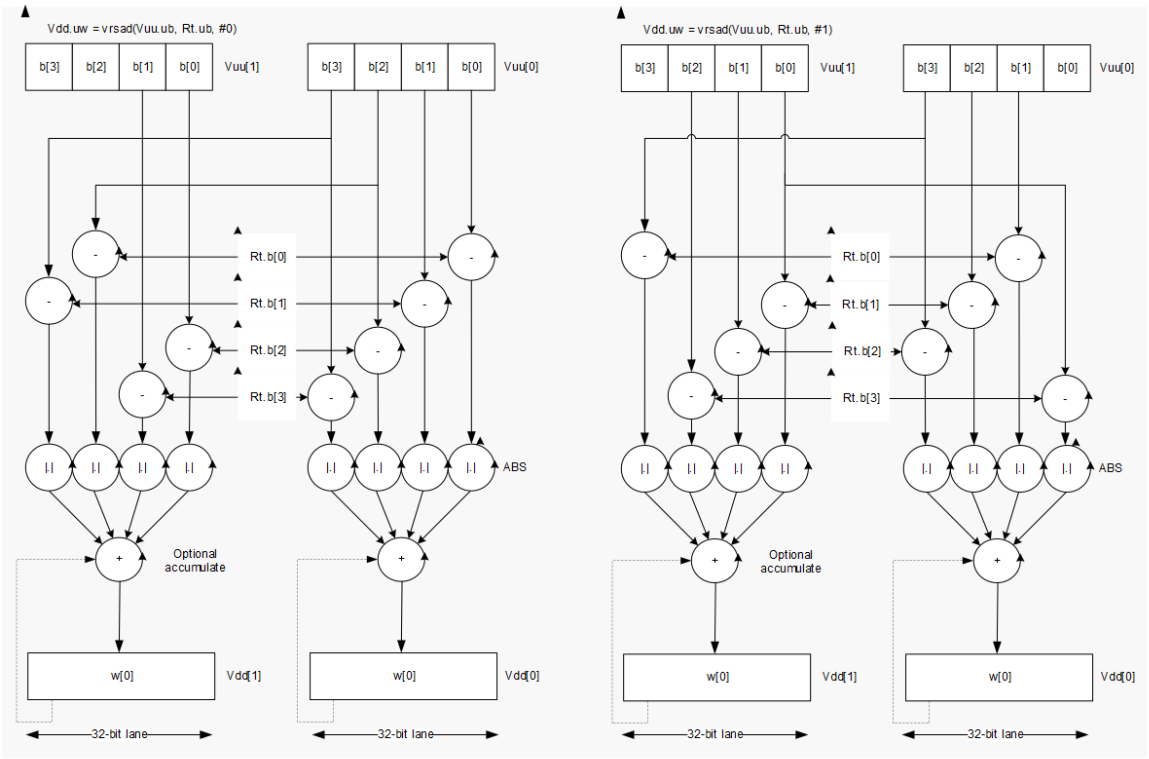
Sum of reduction of absolute differences halfwords intrinsics

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)

Sum of absolute differences byte

Take groups of 4 bytes from the vector register source Vuu, subtract the bytes from the scalar register Rt, and take 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.



Sum of absolute differences byte instructions

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: HVX (slots 2,3)

Note:

- This instruction uses both HVX multiply resources.

Encodings

Bitmap

Syntax	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
Vdd.uw=vrsad(Vuu.ub,Rt.ub,#u1)	0	0	0	1	1	0	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
Vxx.uw+=vrsad(Vuu.ub,Rt.ub,#u1)	0	0	0	1	1	0	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

Intrinsics

Sum of absolute differences byte 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)

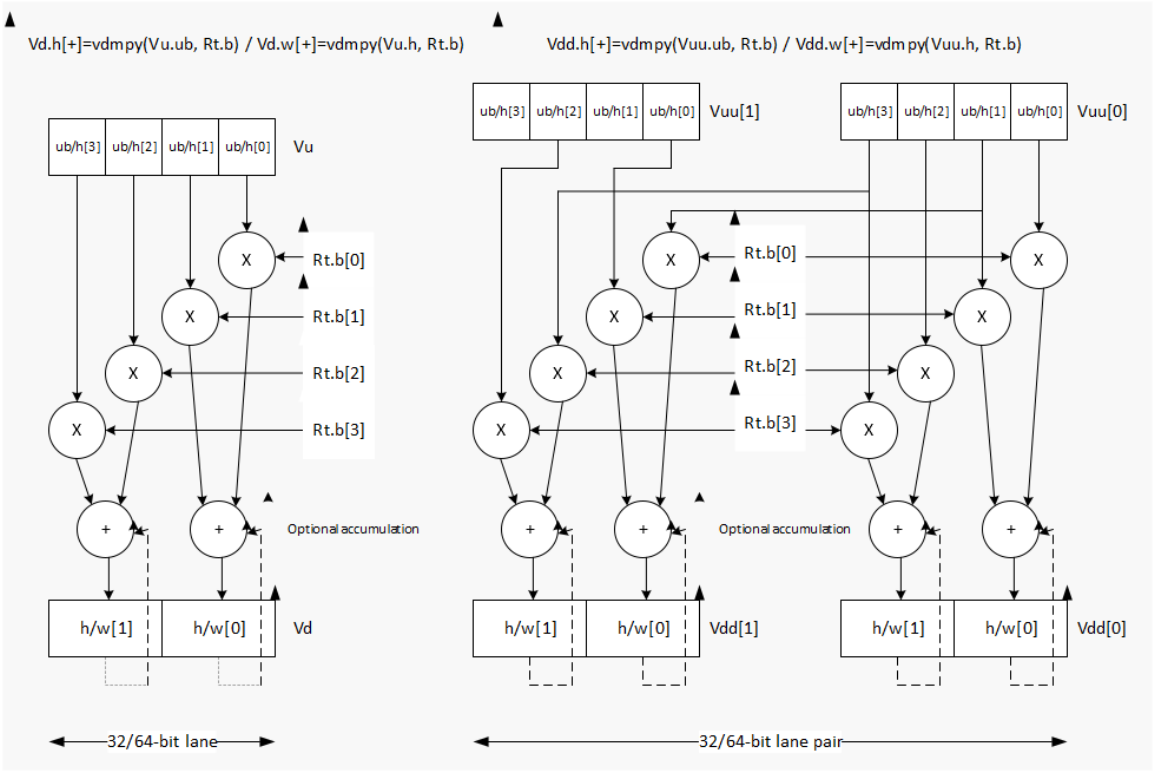
MPY-RESOURCE

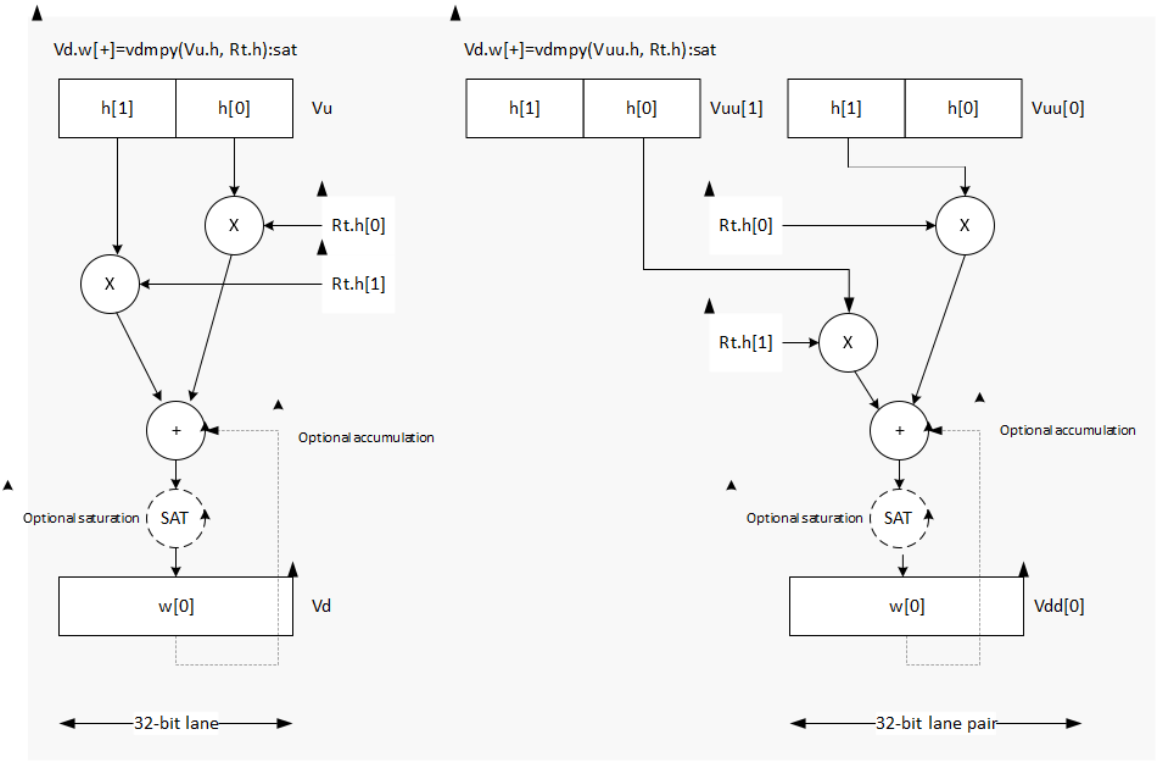
The HVX MPY resource instruction subclass includes instructions that use a single HVX multiply resource.

Multiply with 2-wide reduction

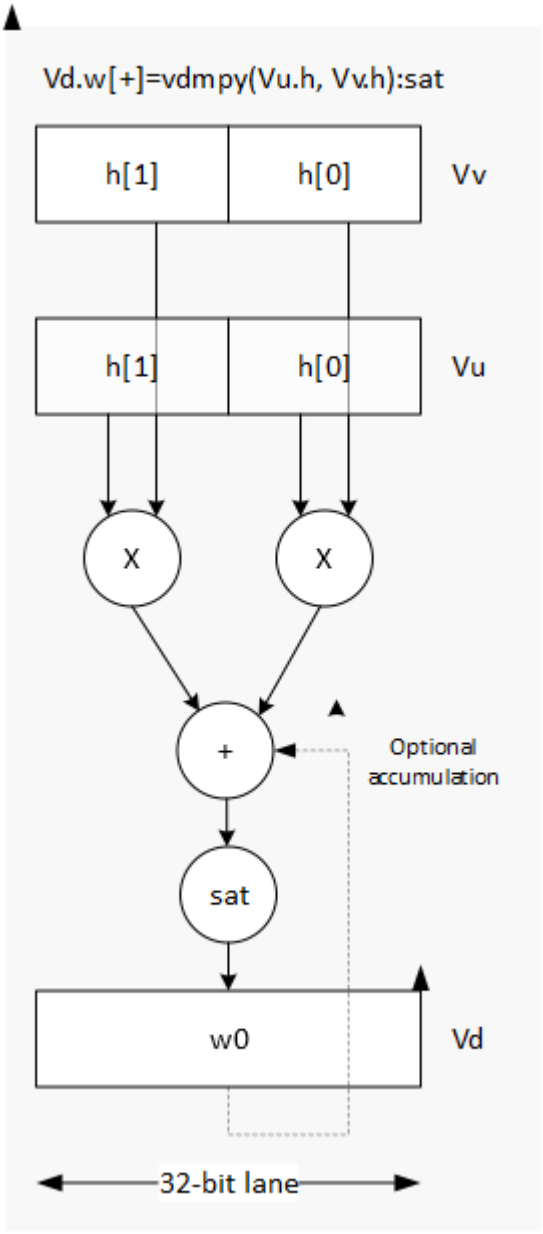
Multiply elements from Vu by the corresponding elements in the scalar register Rt. The products are added in pairs to yield a by-2 reduction. The products can optionally be accumulated with Vx, with optional saturation after summation.

Supports multiplication of unsigned bytes by bytes, halfwords by signed bytes, and halfwords by halfwords.





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


Vd.w[+]=vdmpy(Vu.h, Vv.h):sat
h[1]
h[0]
Vv
h[1]
h[0]
Vu
X
X
+
Optional
accumulation
sat
w0
Vd
32-bit lane

Multiply with 2-wide reduction instructions

Syntax	Behavior
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); }`
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); }`
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); }`
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); }`
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); }`

Class: HVX (slots 2,3)

Note:

- This instruction uses a HVX multiply resource.

Encodings

Bitmap

Syntax	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
Vd.w=vdmpy(Vu.h,Vv.h):sat	0	u	u	0	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,Rt.h):sat	0	u	u	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
Vx.w+=vdmpy(Vu.h,Rt.h):sat	0	u	u	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
Vd.w=vdmpy(Vu.h,Rt.uh):sat	0	u	u	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
Vx.w+=vdmpy(Vu.h,Rt.uh):sat	0	u	u	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

Intrinsics

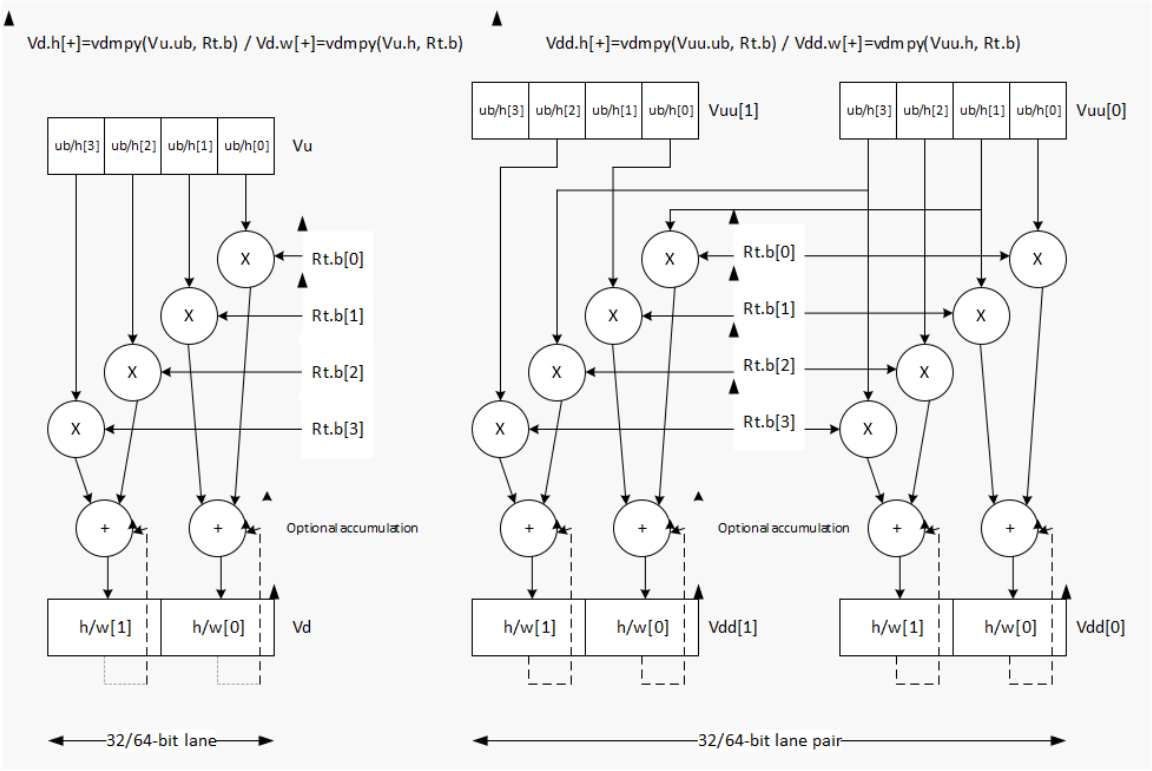
Multiply with 2-wide reduction intrinsics

Vd.w=vdmpy(Vu.h,Vv.h):sat	HVX_Vector Q6_Vw_vdmpy_VhVh_sat(HVX_Vector Vu, HVX_Vector Vv)
Vd.w=vdmpy(Vu.h,Rt.h):sat	HVX_Vector Q6_Vw_vdmpy_VhRh_sat(HVX_Vector Vu, 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)
Vd.w=vdmpy(Vu.h,Rt.uh):sat	HVX_Vector Q6_Vw_vdmpy_VhRuh_sat(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)

Multiply by byte with 2-wide reduction

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

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



Multiply by byte with 2-wide reduction instructions

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]); }`

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]);`
	`}`

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]);`
	`}`

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: HVX (slots 2,3)

Note:

- This instruction uses a HVX multiply resource.

Encodings

Bitmap

Syntax	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
Vd.h=vdmpy(Vu.ub,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
Vx.h+=vdmpy(Vu.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	1	1	0	x	x	x	x	x
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	0	1	0	d	d	d	d	d
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	0	1	1	x	x	x	x	x

Intrinsics

Multiply by byte with 2-wide reduction intrinsics

Vd.h=vdmpy(Vu.ub,Rt.b)	HVX_Vector Q6_Vh_vdmpy_VubRb(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)
Vd.w=vdmpy(Vu.h,Rt.b)	HVX_Vector Q6_Vw_vdmpy_VhRb(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)

Read and set extended bits of vector

The vgetqfext instructions copy the byte elements of scalar register Rt into the destination vector register Vd, under the control of the source vector register's extended bits (Vu.x). Instead of a direct write, the destination can also be or'd with the result. If the corresponding bit i of Vu.x is set, the contents of byte[i % 4] are written or or'ed into Vd or Vx. If Rt contains 0x01010101 then Vu.x can effectively be expanded into Vd or Vx, 1 bit per byte. The destination vector register can hold up to 8 vector register's extended bits.

The vsetqfext instruction copies bits into the destination vector register's 4 extended bits (Vd.x), under the control of the scalar register Rt and the input vector register Vu. If the corresponding byte i of Vu matches any of the bits in Rt byte[i%4] the destination extended bit is set to 1 or 0. If Rt contains 0x01010101 then the extended bits can effectively be filled with the lsb's of Vu, 1 bit per byte. A source vector register can hold up to 8 vector register's extended bits.

These instructions support the operating system to save/restore the register state.

Read and set extended bits of vector instructions

Syntax	Behavior
Vx|=vgetqfext(Vu.x,Rt)	`for (i = 0; i < VELEM(8); i++) { Vx.ub[i] |= Vu.ext[i/4] >> ((i) % 4) ? Rt.ub[i % 4] : 0; }`
Vd=vgetqfext(Vu.x,Rt)	`for (i = 0; i < VELEM(8); i++) { Vd.ub[i] = Vu.ext[i/4] >> ((i) % 4) ? Rt.ub[i % 4] : 0; }`

Syntax	Behavior
Vd.x=vsetqfext(Vu,Rt)	`for (i = 0; i < VELEM(8); i++) { Vd.ext[i/4] |= (((((Vu.ub[i] & Rt.ub[i % 4]) != 0) ? 1 : 0 & (1)) << ((i) % 4))); }`

Class: HVX (slots 2,3)

Note:

- This instruction uses a HVX multiply resource.

Encodings

Bitmap

Syntax	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
Vx|=vgetqfext(Vu.x,Rt)	0	0	0	1	1	0	0	1	1	1	0	t	t	t	t	t	P	P	0	u	u	u	u	u	1	1	0	x	x	x	x	x
Vd=vgetqfext(Vu.x,Rt)	0	0	0	1	1	0	0	1	1	1	0	t	t	t	t	t	P	P	0	u	u	u	u	u	1	1	1	d	d	d	d	d
Vd.x=vsetqfext(Vu,Rt)	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	1	1	d	d	d	d	d

Intrinsics

Read and set extended bits of vector intrinsics

Vd.x=vsetqfext(Vu,Rt)	HVX_Vector Q6_V_vsetqfext_VR(HVX_Vector Vu, Word32 Rt)

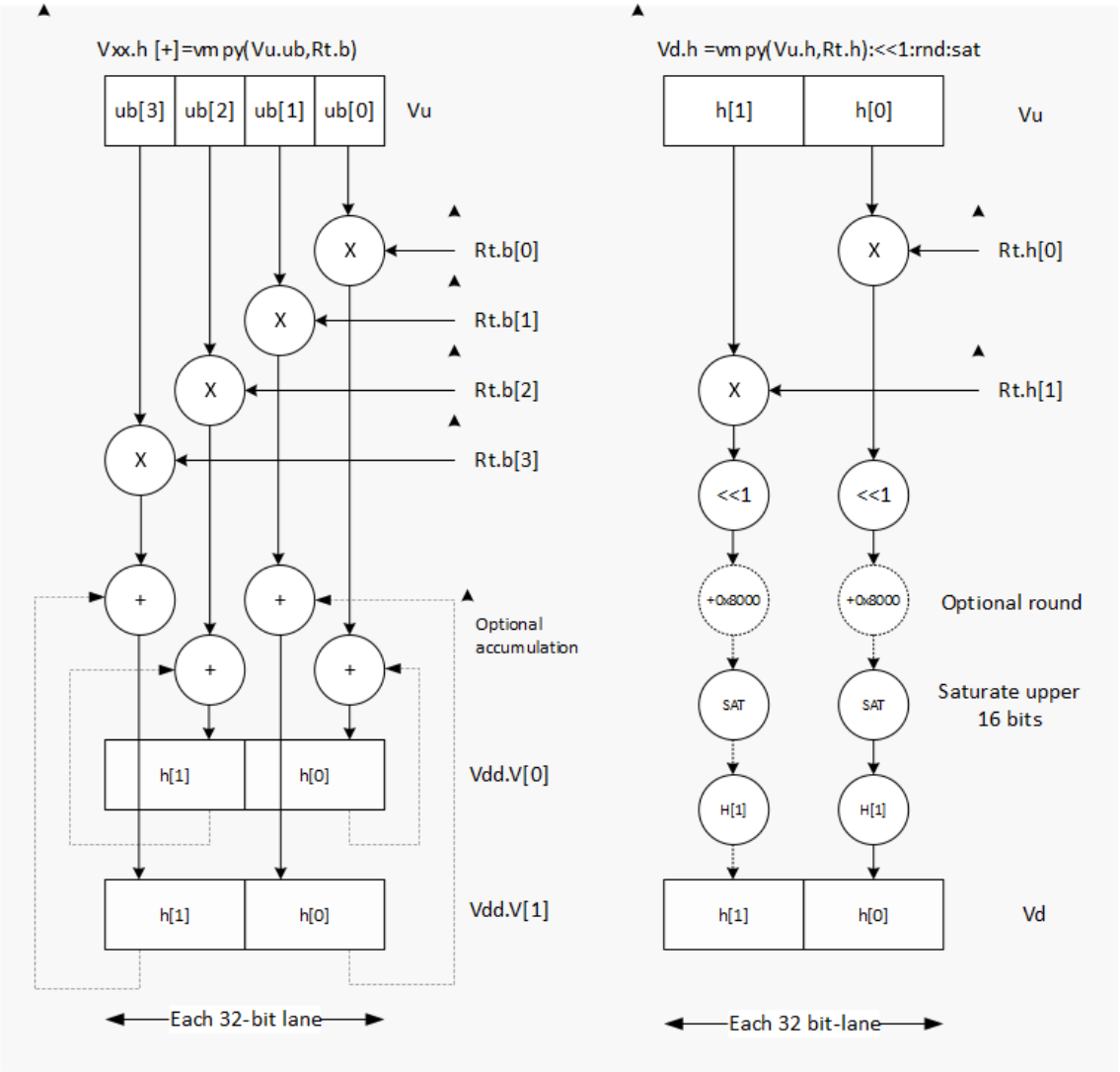
Multiply vector by scalar

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 1, 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,Rt.b)
ub[3]
ub[2]
ub[1]
ub[0]
Vu
X
Rt.b[0]
Rt.b[1]
Rt.b[2]
Rt.b[3]
+
Optional accumulation
h[1]
h[0]
Vdd.V[0]
Vdd.V[1]
Each 32-bit lane
Vd.h =vmpy(Vu.h,Rt.h):<<1:rnd:sat
h[1]
h[0]
Vu
X
Rt.h[0]
Rt.h[1]
<<1
+0x8000
Optional round
SAT
Saturate upper 16 bits
H[1]
Vd
Each 32 bit-lane

Multiply vector by scalar instructions

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]); }`
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]); }`

Class: HVX (slots 2,3)

Note:

- This instruction uses a HVX multiply resource.

Encodings

Bitmap

Syntax	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
Vd.h=vmpy(Vu.h,Rt.h):<<1:sat	0	1	0	1	0	1	1	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:rnd:sat	0	1	0	1	0	1	1	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

Intrinsics

Multiply vector by scalar intrinsics

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

Multiply vector by vector

Multiply groups of elements in the vector Vu by the corresponding elements in the vector register Vv. Optionally rnd, saturate, or shift

Multiply vector by vector instructions

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]); }`
Vd.uh=vmpy(Vu.uh,Vv.uh):>>16	`for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = (Vu.uh[i] * Vv.uh[i]).uh[1]; }`

Class: HVX (slots 2,3)

Note:

- This instruction uses a HVX multiply resource.

Encodings

Bitmap

Syntax	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
Vd.h=vmpy(Vu.h,Vv.h):<<1:rnd:sat	0	0	0	1	1	1	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.uh=vmpy(Vu.uh,Vv.uh):>>16	0	0	0	1	1	1	0	0	1	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	d	d	d	d	d

Intrinsics

Multiply vector by vector 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)
Vd.uh=vmpy(Vu.uh,Vv.uh):>>16	HVX_Vector Q6_Vuh_vmpy_VuhVuh_rs16(HVX_Vector Vu, HVX_Vector Vv)

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.

Multiply half of the elements (16x16) instructions

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: HVX (slots 2,3)

Note:

- This instruction uses a HVX multiply resource.

Encodings

Bitmap

Syntax	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
Vd.w=vmpyieo(Vu.h,Vv.h)	0	0	u	u	u	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

Intrinsics

Multiply half of the elements (16x16) intrinsics

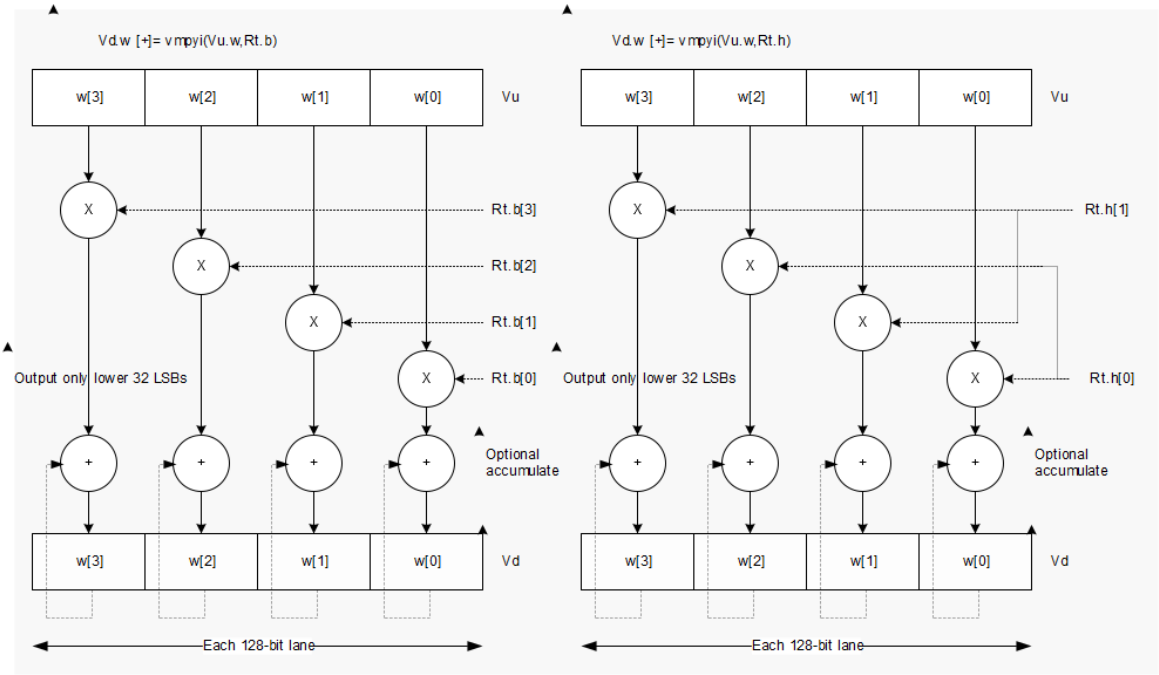
Vd.w=vmpyieo(Vu.h,Vv.h)	HVX_Vector Q6_Vw_vmpyieo_VhVh(HVX_Vector Vu, HVX_Vector Vv)

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.



Integer multiply by byte instructions

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]); }`
Vx.h+=vmpyi(Vu.h,Rt.b)	`for (i = 0; i < VELEM(16); i++) { Vx.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]); }`
Vx.w+=vmpyi(Vu.w,Rt.b)	`for (i = 0; i < VELEM(32); i++) { Vx.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.w+=vmpyi(Vu.w,Rt.ub)	`for (i = 0; i < VELEM(32); i++) { Vx.w[i] += (Vu.w[i] * Rt.ub[i % 4]); }`

Class: HVX (slots 2,3)

Note:

- This instruction uses a HVX multiply resource.

Encodings

Bitmap

Syntax	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
Vd.h=vmpyi(Vu.h,Rt.b)	1	1	0	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
Vx.h+=vmpyi(Vu.h,Rt.b)	1	1	0	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
Vd.w=vmpyi(Vu.w,Rt.b)	1	1	0	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
Vx.w+=vmpyi(Vu.w,Rt.b)	1	1	0	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
Vd.w=vmpyi(Vu.w,Rt.ub)	1	1	0	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
Vx.w+=vmpyi(Vu.w,Rt.ub)	1	1	0	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

Intrinsics

Integer multiply by byte intrinsics

Vd.h=vmpyi(Vu.h,Rt.b)	HVX_Vector Q6_Vh_vmpyi_VhRb(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)
Vd.w=vmpyi(Vu.w,Rt.b)	HVX_Vector Q6_Vw_vmpyi_VwRb(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)
Vd.w=vmpyi(Vu.w,Rt.ub)	HVX_Vector Q6_Vw_vmpyi_VwRub(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)

Syntax	Behavior

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.

Multiply half of the elements with scalar (16 x 16) instructions

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: HVX (slots 2,3)

Note:

- This instruction uses a HVX multiply resource.

Encodings

Bitmap

Syntax	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
Vd.uw=vmpye(Vu.uh,Rt.uh)	0	u	u	0	1	1	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
Vx.uw+=vmpye(Vu.uh,Rt.uh)	0	u	u	0	1	1	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

Intrinsics

Multiply half of the elements with scalar (16 x 16) intrinsics

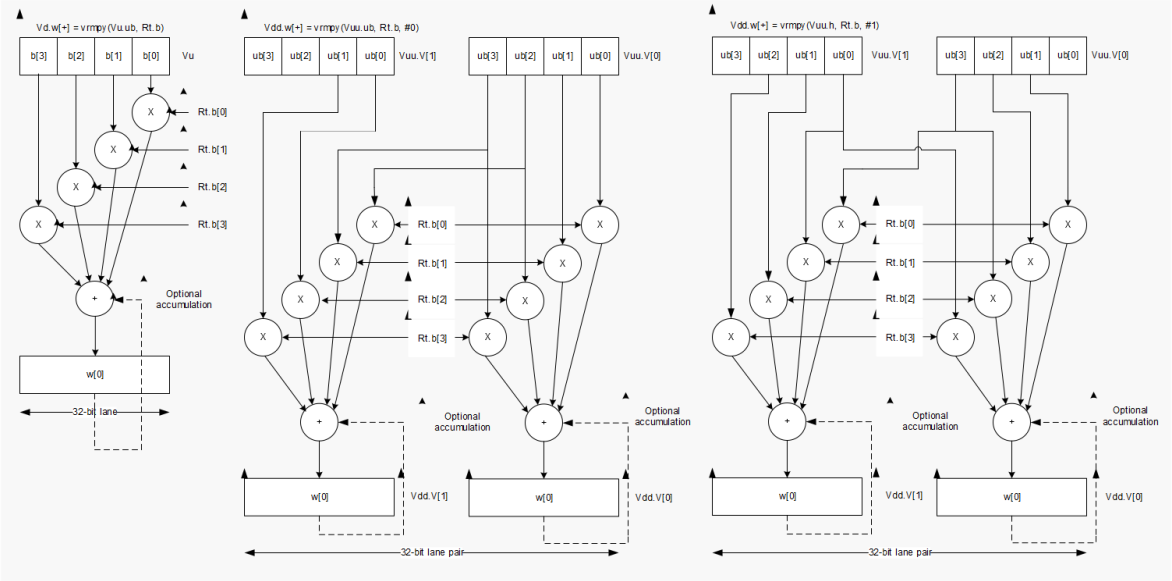
Vd.uw=vmpye(Vu.uh,Rt.uh)	HVX_Vector Q6_Vuw_vmpye_VuhRuh(HVX_Vector Vu, Word32 Rt)
Vx.uw+=vmpye(Vu.uh,Rt.uh)	HVX_Vector Q6_Vuw_vmpyeacc_VuwVuhRuh(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)

Multiply bytes with 4-wide reduction - vector by scalar

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

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

The operation has two forms: the first performs simple dot product of 4 elements into a single result. The second form takes a 1 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 2 elements and the upper 2 data elements taken from the even register of Vuu. For #u = 1, the even destination takes coefficients rotated by -1 and data element 0 from the odd register of Vuu. The odd destination uses coefficients rotated by -1 and takes data element 3 from the even register of Vuu.



Multiply bytes with 4-wide reduction - vector by scalar instructions

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]); }`
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]); }`
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]); }`

Syntax	Behavior

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: HVX (slots 2,3)

Note:

- This instruction uses a HVX multiply resource.

Encodings

Bitmap

Syntax	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
Vd.uw=vrmpy(Vu.ub,Rt.ub)	0	v	v	v	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
Vx.uw+=vrmpy(Vu.ub,Rt.ub)	0	v	v	v	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
Vd.w=vrmpy(Vu.ub,Rt.b)	1	v	v	v	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
Vx.w+=vrmpy(Vu.ub,Rt.b)	0	v	v	v	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

Intrinsics

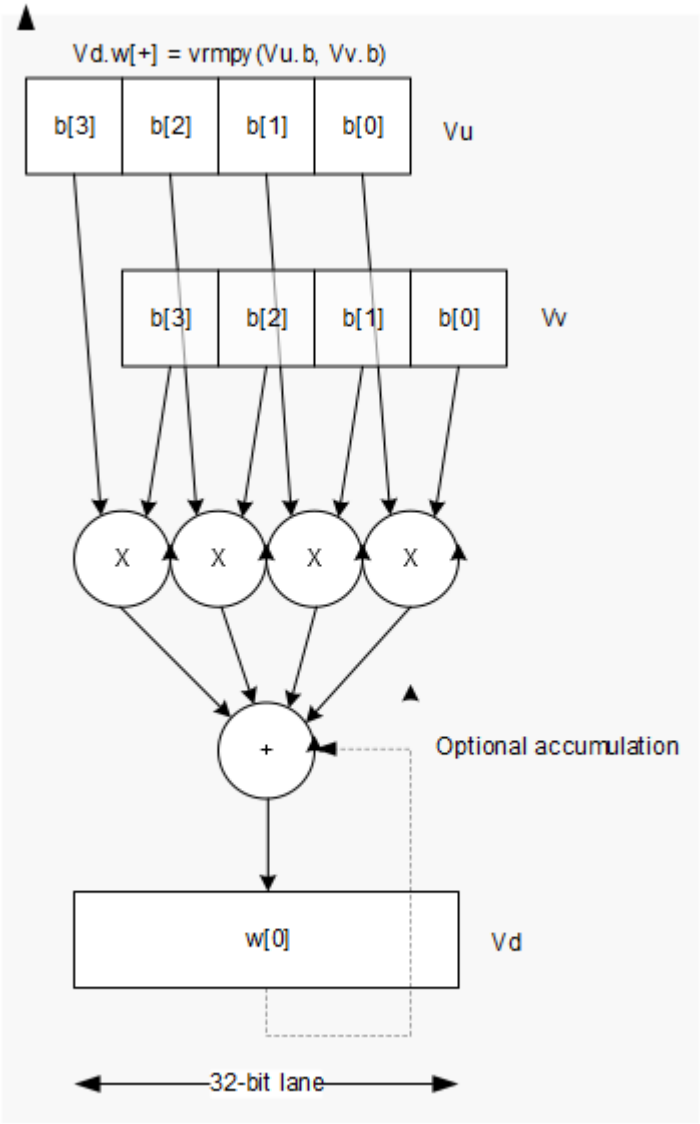
Multiply bytes with 4-wide reduction - vector by scalar intrinsics

Vd.uw=vrmpy(Vu.ub,Rt.ub)	HVX_Vector Q6_Vuw_vrmpy_VubRub(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)
Vd.w=vrmpy(Vu.ub,Rt.b)	HVX_Vector Q6_Vw_vrmpy_VubRb(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)

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

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

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


Vd.w[+] = vrmpy(Vu.b, Vv.b)
b[3]
b[2]
b[1]
b[0]
Vu
b[3]
b[2]
b[1]
b[0]
Vv
X
X
X
X
+
Optional accumulation
w[0]
Vd
32-bit lane

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

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: HVX (slots 2,3)

Note:

- This instruction uses a HVX multiply resource.

Encodings

Bitmap

Syntax	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
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	0	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	0	1	d	d	d	d	d
Vd.w=vrmpy(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	0	1	0	d	d	d	d	d

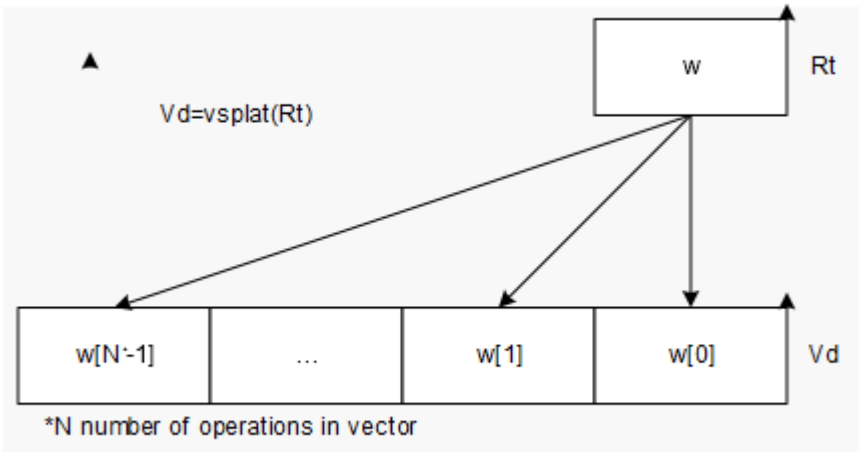
Intrinsics

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

Splat from scalar

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



Splat from scalar instructions

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

Class: HVX (slots 2,3)

Note:

- This instruction uses a HVX multiply resource.

Encodings

Bitmap

Syntax	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
Vd=vsplat(Rt)	0	0	0	1	1	0	0	1	1	0	1	t	t	t	t	t	P	P	0	-	-	-	-	0	0	0	1	d	d	d	d	d
Vd.h=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.b=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

Intrinsics

Splat from scalar intrinsics

Vd=vsplat(Rt)	HVX_Vector Q6_V_vsplat_R(Word32 Rt)
Vd.h=vsplat(Rt)	HVX_Vector Q6_Vh_vsplat_R(Word32 Rt)
Vd.b=vsplat(Rt)	HVX_Vector Q6_Vb_vsplat_R(Word32 Rt)

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, 1 bit per byte.

Vector to predicate transfer instructions

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: HVX (slots 2,3)

Note:

- This instruction uses a HVX multiply resource.

Encodings

Bitmap

Syntax	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
Qd4=vand(Vu,Rt)	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
Qx4|=vand(Vu,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	0	-	-	-	x	x

Intrinsics

Vector to predicate transfer 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)

Syntax	Behavior

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'ed into Vd or Vx.

If Rt contains 0x01010101 then Qt can effectively be expanded into Vd or Vx, 1 bit per byte.

Predicate to vector transfer instructions

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; }`
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: HVX (slots 2,3)

Note:

- This instruction uses a HVX multiply resource.

Encodings

Bitmap

Syntax	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Vd=vand(Qu4,Rt)	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
Vx|=vand(Qu4,Rt)	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
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
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

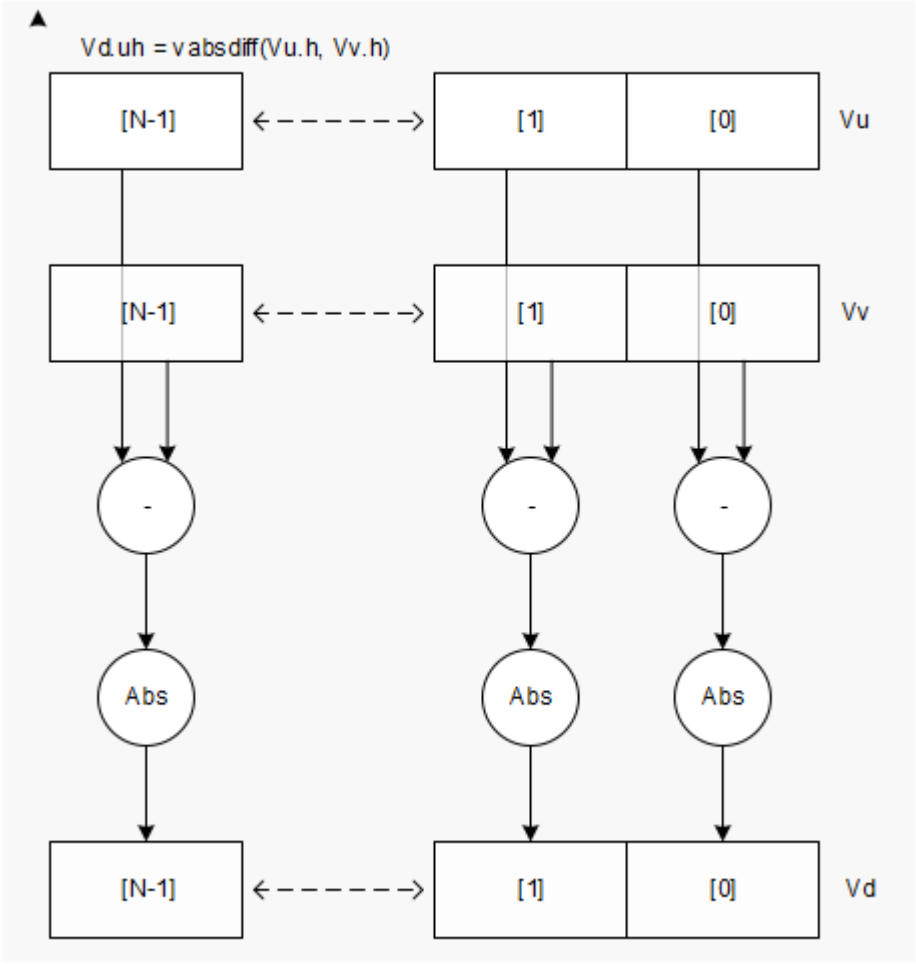
Intrinsics

Predicate to vector transfer intrinsics

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

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.

Absolute value of difference instructions

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.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.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.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: HVX (slots 2,3)

Note:

- This instruction uses a HVX multiply resource.

Encodings

Bitmap

Syntax	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
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	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	0	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	0	1	d	d	d	d	d

Syntax	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
Vd.uw=vabsdiff(Vu.w,Vv.w)	0	0	0	1	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

Intrinsics

Absolute value of difference intrinsics

Vd.ub=vabsdiff(Vu.ub,Vv.ub)	HVX_Vector Q6_Vub_vabsdiff_VubVub(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.uh=vabsdiff(Vu.h,Vv.h)	HVX_Vector Q6_Vuh_vabsdiff_VhVh(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)

Insert element

Insert a 32-bit element in Rt into the destination vector register Vx, at the word element 0.

Insert element instructions

Syntax	Behavior
Vx.w=vinsert(Rt)	`Vx.uw[0] = Rt;`

Class: HVX (slots 2,3)

Note:

- This instruction uses a HVX multiply resource.

Encodings

Bitmap

Syntax	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
Vx.w=vinsert(Rt)	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

Intrinsics

Insert element intrinsics

Vx.w=vinsert(Rt)	HVX_Vector Q6_Vw_vinsert_VwR(HVX_Vector Vx, Word32 Rt)

PERMUTE-RESOURCE

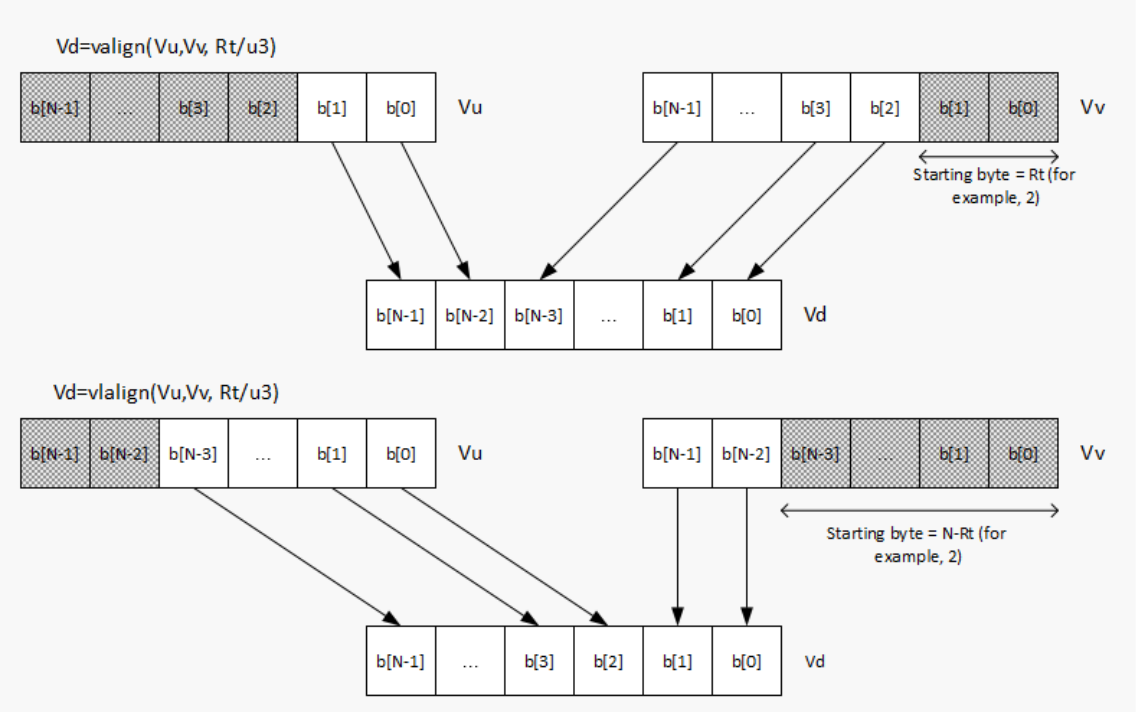
The HVX permute-resource instruction subclass includes instructions that use the HVX permute resource.

Byte alignment

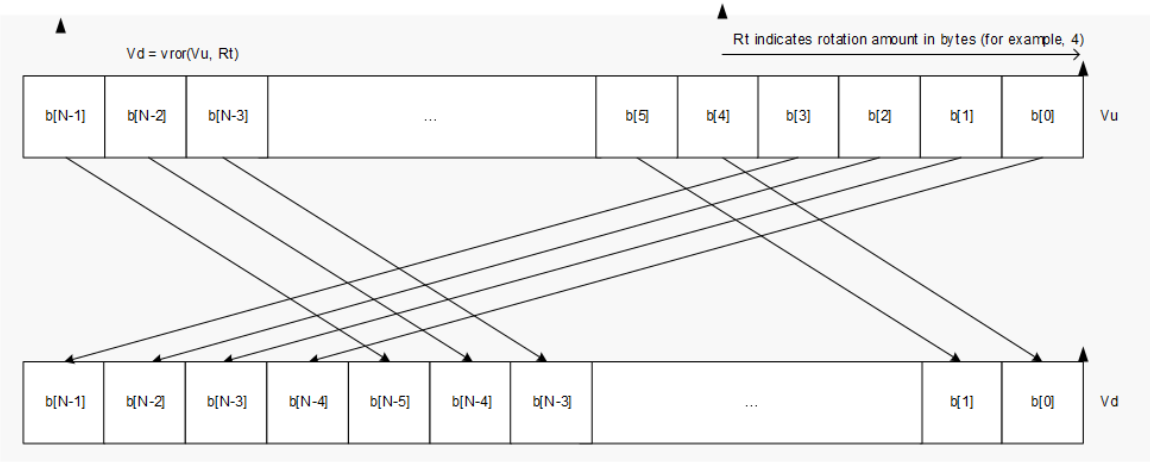
Select a continuous group of bytes the size of a vector register from vector registers Vu and Vv. The starting location is provided by the lower bits of Rt (modulo the vector length) or by a 3-bit immediate value.

There are two forms of the operation, The first, valign, uses the Rt or immediate input directly to specify the beginning of the block. The second, vlalign, uses the inverse of the input value by subtracting it from the vector length.

The operation can be used to implement a non-aligned vector load, using two aligned loads (above and below the pointer) and a valign where the pointer is used as the control input.



Perform a right rotate vector operation on vector register Vu, by the number of bytes specified by the lower bits of Rt. The result is written into Vd. Byte[i] moves to Byte[(i+N-R)%N], where R is the right rotate amount in bytes, and N is the vector register size in bytes.



Byte alignment instructions

Syntax	Behavior

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,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=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];`
	`}`

Syntax	Behavior

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=vror(Vu,Rt)	`for (k=0;k<VWIDTH;k++) {`
	`  Vd.ub[k] = Vu.ub[(k+Rt)&(VWIDTH-1)];`
	`}`

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

Note:

- This instruction uses the HVX permute/shift resource.
- Input scalar register Rt is limited to registers 0 through 7

Encodings

Bitmap

Syntax	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
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	0	d	d	d	d	d
Vd=vlalign(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=valign(Vu,Vv,#u3)	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=vlalign(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=vror(Vu,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	0	0	1	d	d	d	d	d

Intrinsics

Byte alignment intrinsics

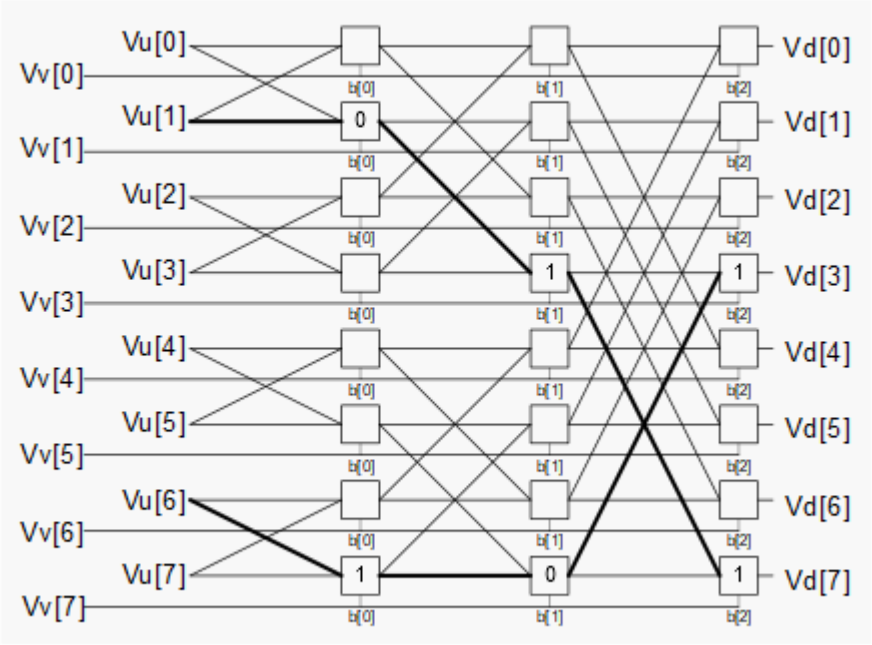
Vd=valign(Vu,Vv,Rt)	HVX_Vector Q6_V_valign_VVR(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd=vlalign(Vu,Vv,Rt)	HVX_Vector Q6_V_vlalign_VVR(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd=valign(Vu,Vv,#u3)	HVX_Vector Q6_V_valign_VVI(HVX_Vector Vu, HVX_Vector Vv, Word32 Iu3)
Vd=vlalign(Vu,Vv,#u3)	HVX_Vector Q6_V_vlalign_VVI(HVX_Vector Vu, HVX_Vector Vv, Word32 Iu3)
Vd=vror(Vu,Rt)	HVX_Vector Q6_V_vror_VR(HVX_Vector Vu, Word32 Rt)

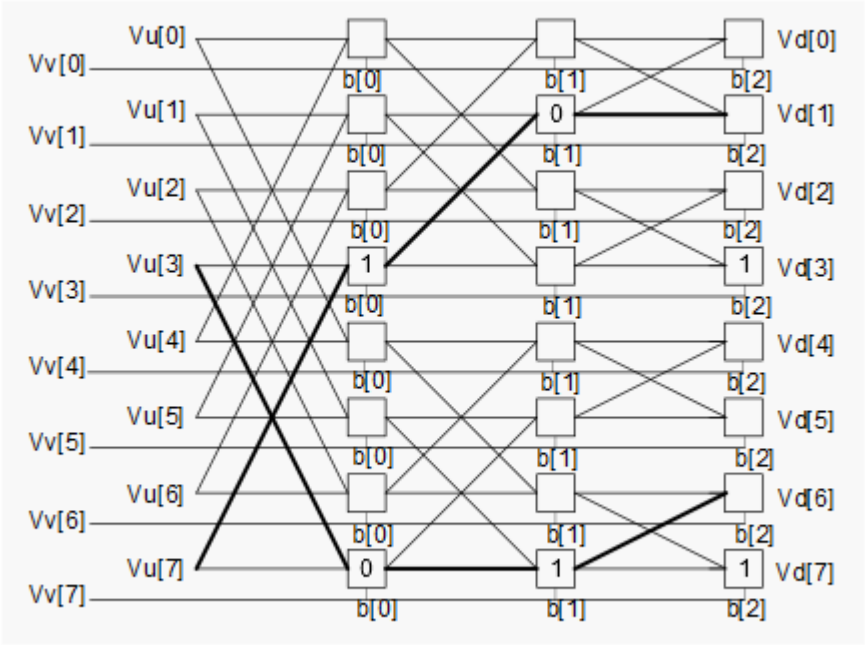
General permute network

Perform permutation and re-arrangement of the 64 input bytes, which is the width of a data slice. The input data is passed through a network of switch boxes, these are able to take two inputs and based on the two controls can pass through, swap, replicate the first input, or replicate the second input. Though the functionality is powerful the algorithms to compute the controls are complex.

The input vector of bytes is passed through six levels of switches 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 3), the bit at that position would look at the corresponding bit (bit 3) 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 8-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:0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,52,54,56,58,60,62,63,61,5

controls =
{0x00,0x02,0x05,0x07,0x0A,0x08,0x0F,0x0D,0x14,0x16,0x11,0x13,0x1E,0x1C,0x1B,0x19,0x28,0x2A,0x2D,0x2F,0x

Similarly, here is a function that replicates every 4th element:
0,0,0,0,4,4,4,4,8,8,8,8,12,12,12,12,16,16,16,16,20,20,20,20,24,24,24,24,28,28,28,28,32,32,32,32,36,36,36,36,40,4

Valid controls =
{0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x00,0x01,0x02,0x03,0x0

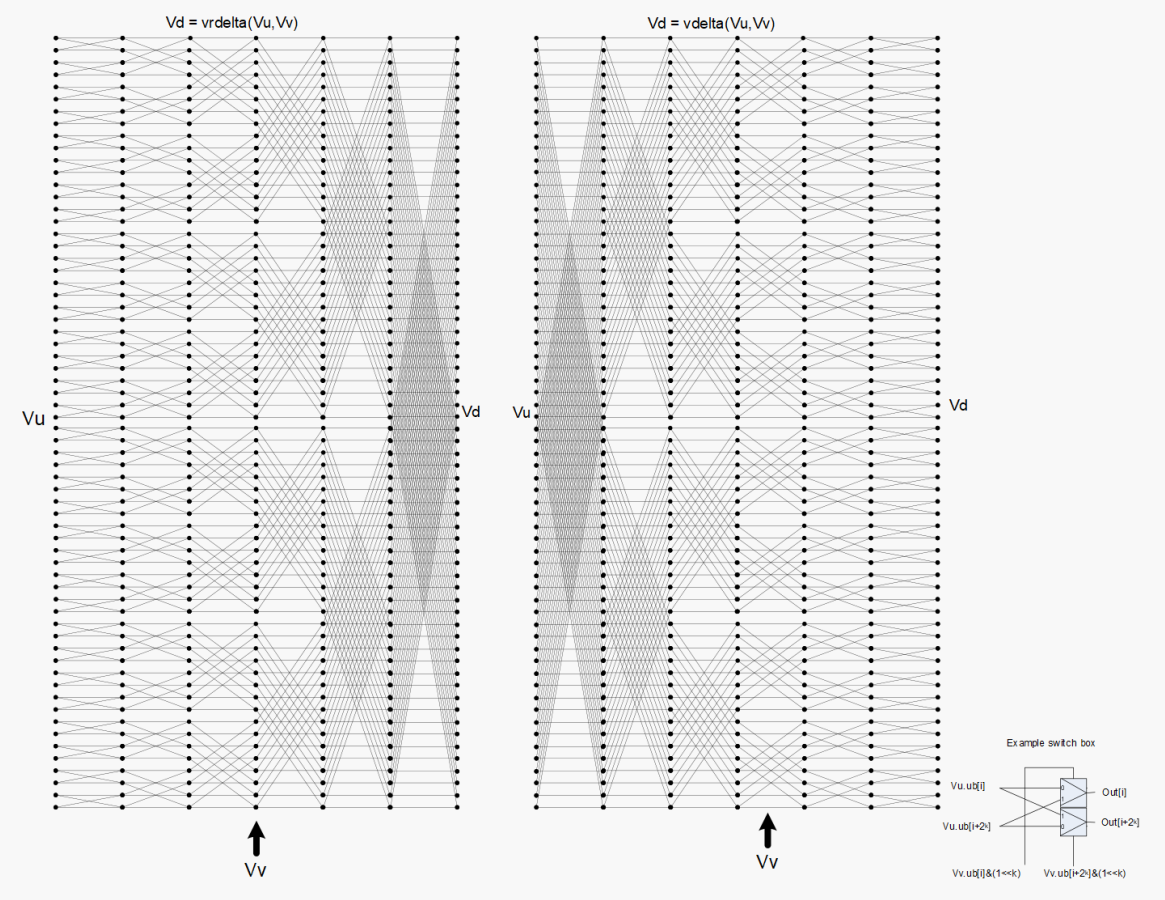
The other general form of permute is a Benes Network, which requires a vrdelta immediately followed by a vdelta operation. This form is non-blocking: any possible permute, however random, can be accommodated, though it has to be a permutation, each input must have a position in the output. Replication 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}



General permute network instructions

Syntax	Behavior
Vd=vdelta(Vu,Vv)	`for (offset=VWIDTH; (offset>>=1)>0; ) { for (k = 0; k<VWIDTH; k++) { Vd.ub[k] = (Vv.ub[k]&offset) ? Vu.ub[k^offset] : Vu.ub[k]; } for (k = 0; k<VWIDTH; k++) { Vu.ub[k] = Vd.ub[k]; } }`

Syntax	Behavior
Vd=vrdelta(Vu,Vv)	`for (offset=1; offset<VWIDTH; offset<<=1) { for (k = 0; k<VWIDTH; k++) { Vd.ub[k] = (Vv.ub[k]&offset) ? Vu.ub[k^offset] : Vu.ub[k]; } for (k = 0; k<VWIDTH; k++) { Vu.ub[k] = Vd.ub[k]; } }`

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

Note:

- This instruction uses the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
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	0	1	d	d	d	d	d
Vd=vrdelta(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

Intrinsics

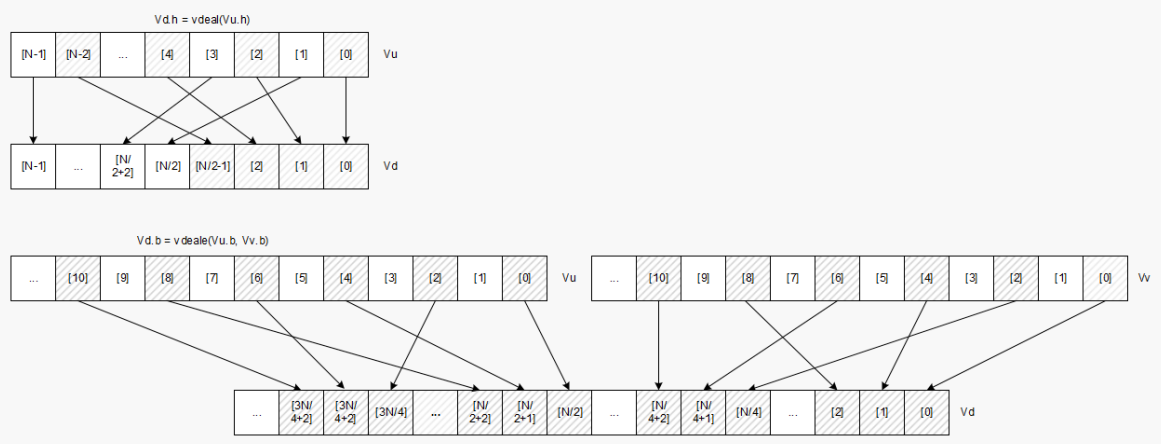
General permute network intrinsics

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)

Shuffle - Deal

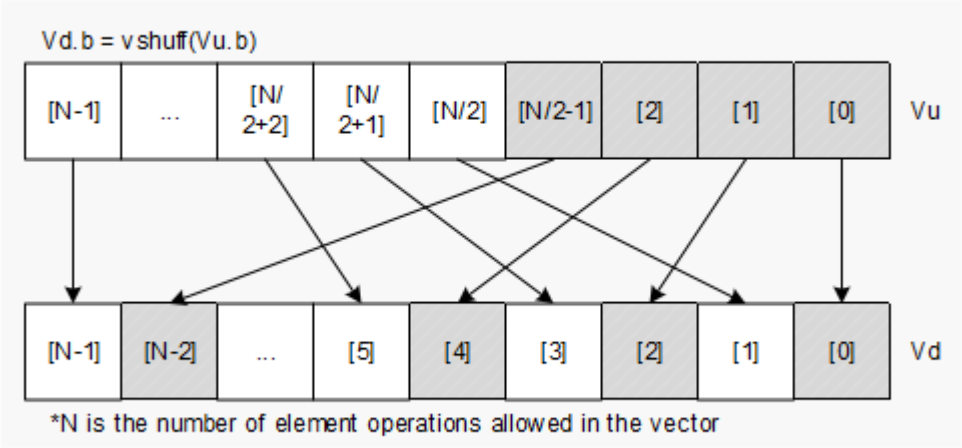
Deal or deinterleave the elements into the destination register Vd. Even elements of Vu are placed in the lower half of Vd, and odd elements are placed in the upper half.

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.



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.



Shuffle - Deal instructions

Syntax	Behavior
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.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]; }`

Syntax	Behavior

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];`
	`}`

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];`
	`}`

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

Note:

- This instruction uses the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
Vd.h=vdeal(Vu.h,Rt)	0	1	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.b=vdeal(Vu.b,Rt)	0	1	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=vdeale(Vu.b,Vv.b)	0	1	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.h=vshuff(Vu.h,Rt)	0	1	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.b=vshuff(Vu.b,Rt)	0	1	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

Intrinsics

Shuffle - Deal intrinsics

Vd.h=vdeal(Vu.h)	HVX_Vector Q6_Vh_vdeal_Vh(HVX_Vector Vu)
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.h=vshuff(Vu.h)	HVX_Vector Q6_Vh_vshuff_Vh(HVX_Vector Vu)
Vd.b=vshuff(Vu.b)	HVX_Vector Q6_Vb_vshuff_Vb(HVX_Vector Vu)

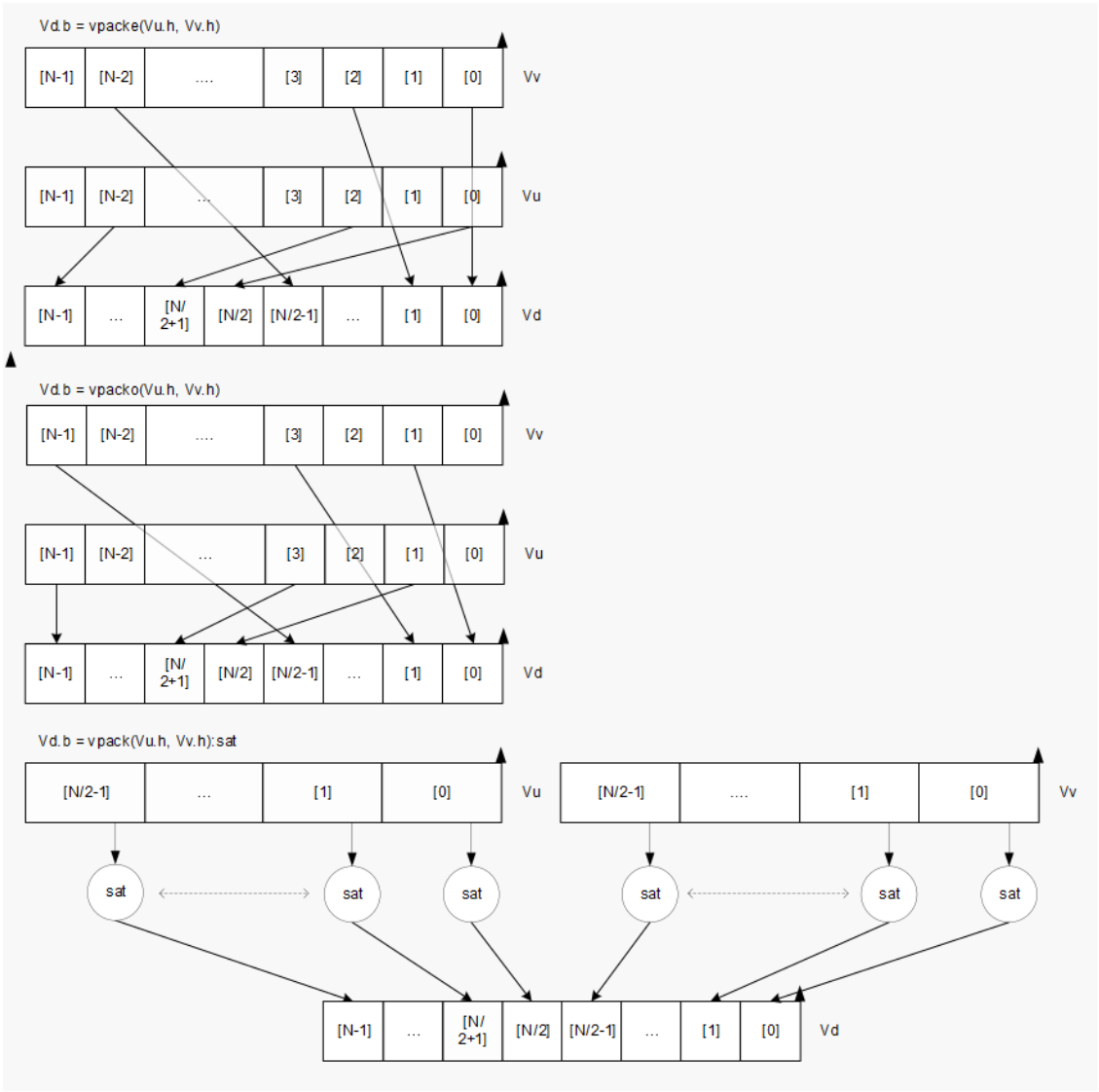
Pack

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.



Pack instructions

Syntax	Behavior
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]; }`

Syntax	Behavior
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.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=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.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.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]); }`
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]); }`

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

Note:

- This instruction uses the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
Vd.b=vpacke(Vu.h,Vv.h)	0	u	u	0	v	v	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
Vd.h=vpacke(Vu.w,Vv.w)	0	u	u	0	v	v	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
Vd.b=vpacko(Vu.h,Vv.h)	0	u	u	0	v	v	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
Vd.h=vpacko(Vu.w,Vv.w)	0	u	u	0	v	v	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
Vd.ub=vpack(Vu.h,Vv.h):sat	0	u	u	0	v	v	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
Vd.b=vpack(Vu.h,Vv.h):sat	0	u	u	0	v	v	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
Vd.uh=vpack(Vu.w,Vv.w):sat	0	u	u	0	v	v	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
Vd.h=vpack(Vu.w,Vv.w):sat	0	u	u	0	v	v	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

Intrinsics

Pack intrinsics

Vd.b=vpacke(Vu.h,Vv.h)	HVX_Vector Q6_Vb_vpacke_VhVh(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.b=vpacko(Vu.h,Vv.h)	HVX_Vector Q6_Vb_vpacko_VhVh(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.b=vpack(Vu.h,Vv.h):sat	HVX_Vector Q6_Vb_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)
Vd.h=vpack(Vu.w,Vv.w):sat	HVX_Vector Q6_Vh_vpack_VwVw_sat(HVX_Vector Vu, HVX_Vector Vv)

Set predicate

Set a vector predicate register with a sequence of 1's based on the lower bits of the scalar register Rt.

Rt = 0x11 : Qd4 = 0——00111111111111111b

Rt = 0x07 : Qd4 = 0——00000000000001111111b

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.

Set predicate instructions

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: HVX (slots 0,1,2,3)

Note:

- This instruction uses the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
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	-	0	1	d	d
Qd4=vsetq2(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

Intrinsics

Set predicate intrinsics

Qd4=vsetq(Rt)	HVX_VectorPred Q6_Q_vsetq_R(Word32 Rt)
Qd4=vsetq2(Rt)	HVX_VectorPred Q6_Q_vsetq2_R(Word32 Rt)

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).

8bit elements. </p>

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 5 bits of the input byte for the respective output byte. A control input register, Rt, contains match and select bits. The lower 3 bits of Rt must match the upper 3 bits of the input byte index in order for the table entry to be written to or Or'ed 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 as below

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

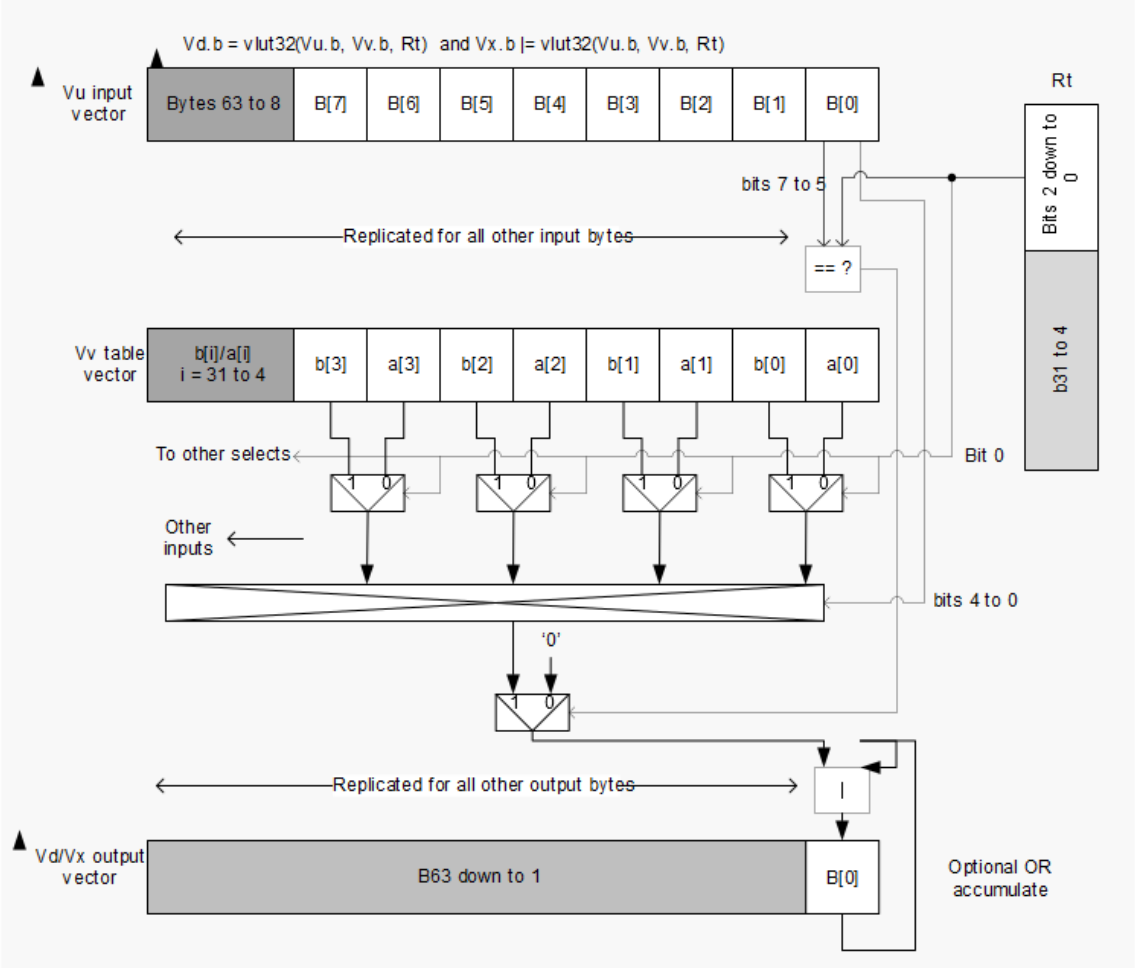
in order to prepare it for use with the vlut instruction in 64B mode it must be shuffled in blocks of 32 bytes

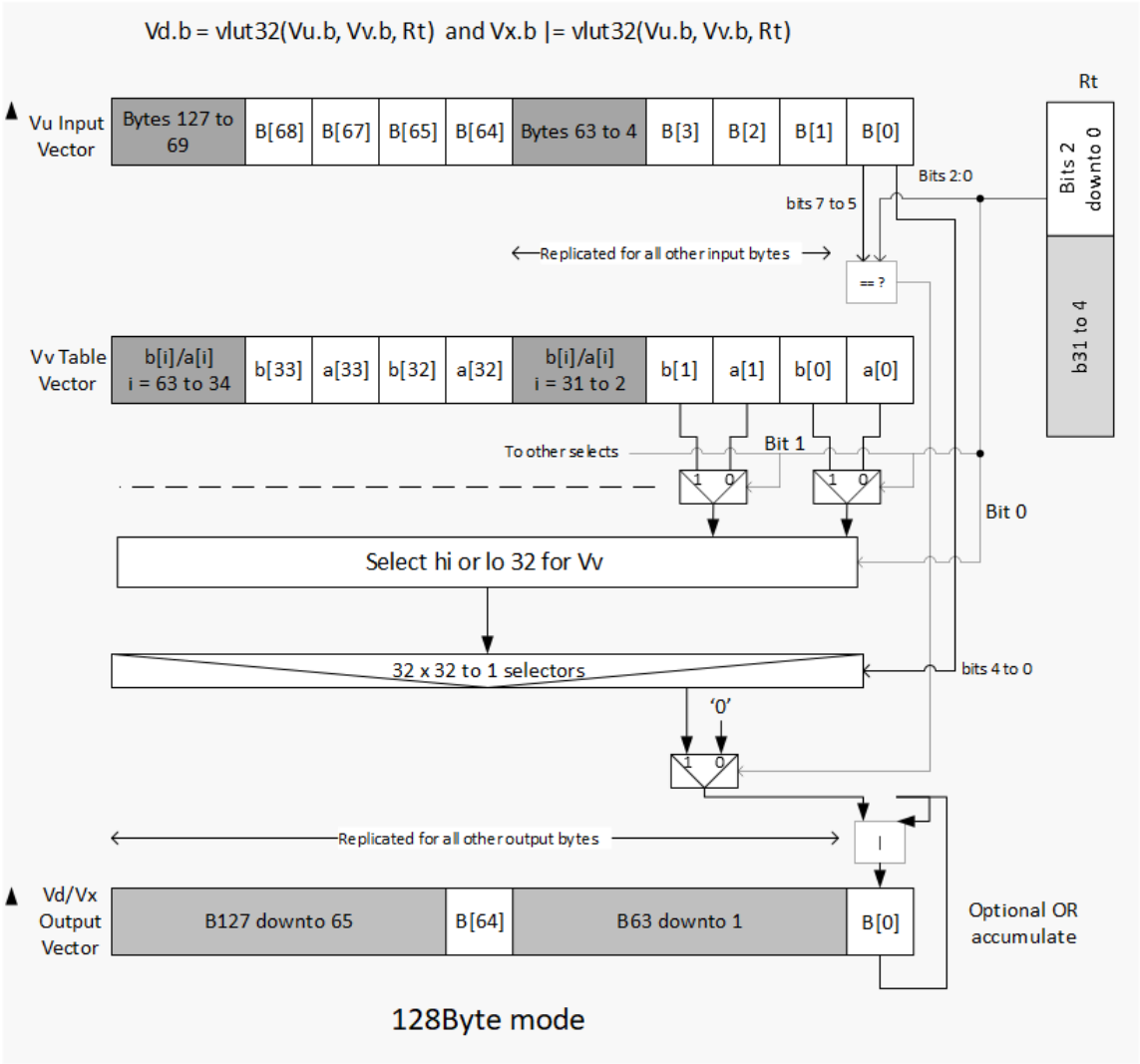
```
 63, 31, 62, 30,......36,  4, 35,  3, 34,  2, 33,  1, 32,  0 Rt=0,
Rt=1
127, 95,126, 94,.....100, 68, 99, 67, 98, 66, 97, 65, 96, 64 Rt=2,
Rt=3

same ordering for bytes 128-255       Rt=4, 5, 6, 7
```

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 will give the same results. In the case of 128B mode bit 1 of Rt selects whether to use the odd or even packed table and bit 0 chooses the high of low 32 elements of that hi or low table.





16bit elements. </p>

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 4 bits for the even destination register Vdd, the odd byte entries select table entries into the odd vector destination register Vdd. 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 4 bits of Rt must match the upper 4 bits of the input bytes in order for the table entry to be written to or Or'ed 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 4 of Rt. Bit 1 of Rt selects odd or even hwords and bit 0 selects the lower or upper 16 entries in the Vv register.

For larger than 32-element tables in the hword case (for example 256 entries), the user must access the main lookup table in 8 different 32 hword sections. If a 256H table is stored naturally in memory it would look as below

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

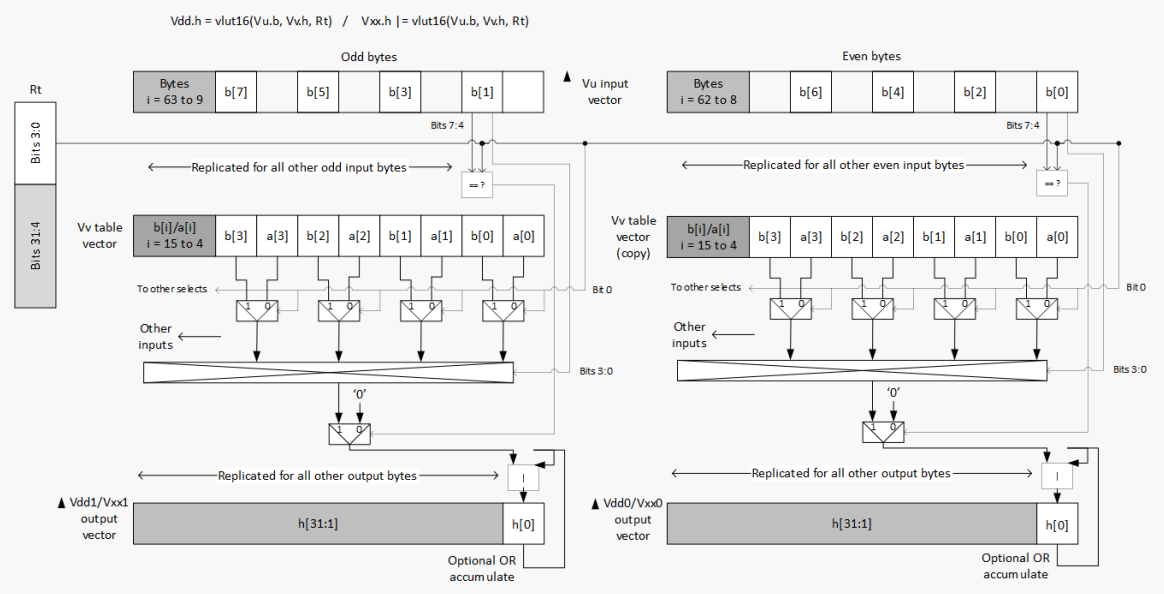
in order to prepare it for use with the vlut instruction in 64B mode it must be shuffled in blocks of 16 hwords, the LSB of Rt is used to choose the even or odd 16 entry hword tables in Vv.

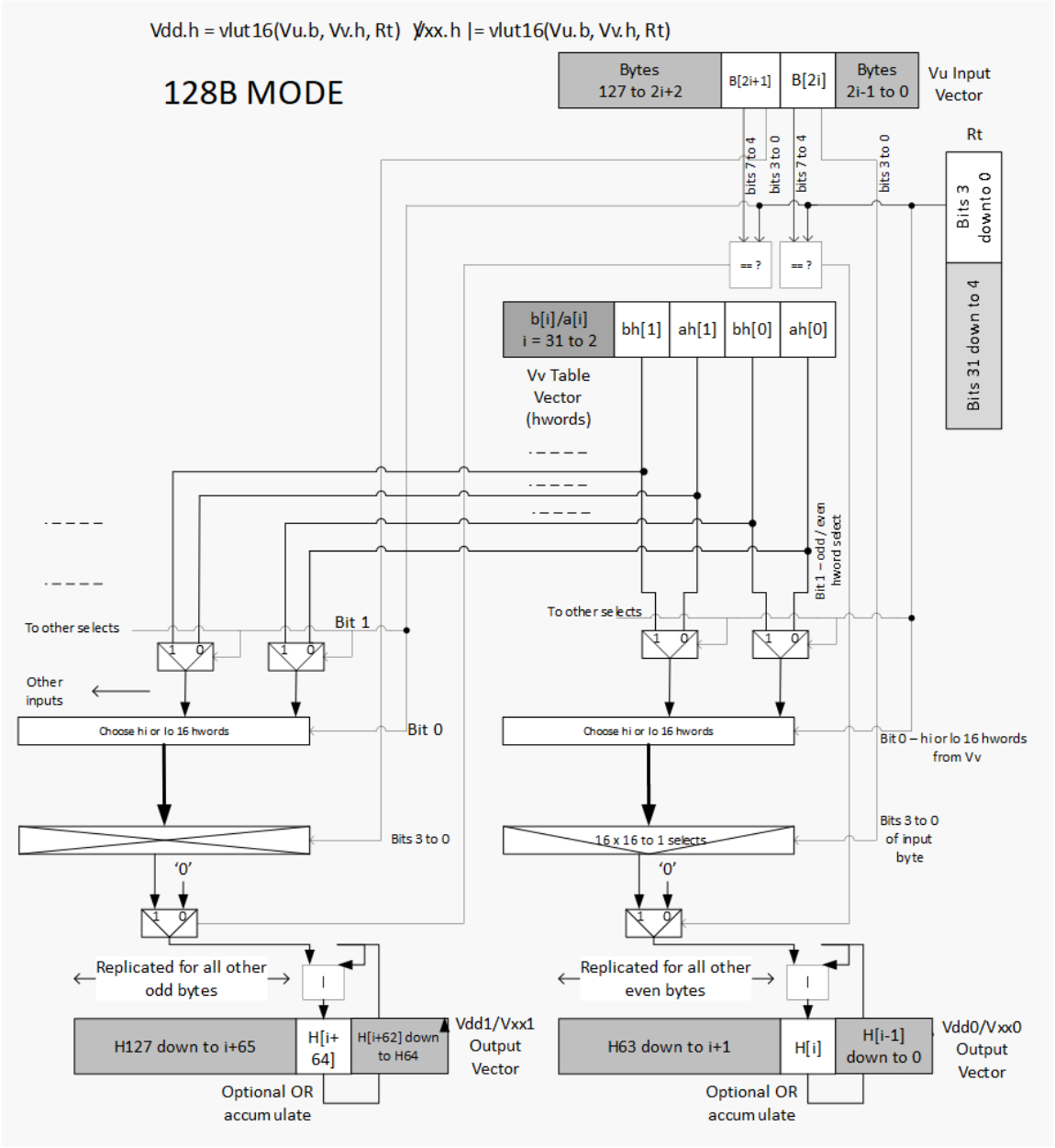
```
 31, 15, 30, 14,......20,  4, 19,  3, 18,  2, 17,  1, 16,  0 Rt=0,
Rt=1
 63, 47, 62, 46,..... 52, 36, 51, 35, 50, 34, 49, 33, 48, 32 Rt=2,
Rt=3

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 1 of Rt is used to choose between the even or odd 32 hwords in Vv. Bit 0 accesses the hi or lo 16 half words of the odd or even set.

```
 63, 31, 62, 30,........36,  4, 35,  3, 34,  2, 33,  1, 32,  0 Rt=0,1
Rt=2,3

same ordering for bytes 128-255     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.


Vdd.h = vlut16(Vu.b, Vv.h, Rt) / Vxx.h |= vlut16(Vu.b, Vv.h, Rt)
Odd bytes
Even bytes
Rt
Bits 3:0
Bits 31:4
Vu input vector
Bytes i = 63 to 9
b[7]
b[5]
b[3]
b[1]
Bytes i = 62 to 8
b[6]
b[4]
b[2]
b[0]
Bits 7:4
Replicated for all other odd input bytes
Replicated for all other even input bytes
==?
Vv table vector
b[i]/a[i] i = 15 to 4
b[3]
a[3]
b[2]
a[2]
b[1]
a[1]
b[0]
a[0]
Vv table vector (copy)
To other selects
Bit 0
Other inputs
Bits 3:0
'0'
1
0
Replicated for all other output bytes
Vdd1/Vxx1 output vector
Vdd0/Vxx0 output vector
h[31:1]
h[0]
Optional OR accumulate



vluts with the nomatch extension no not look at the upper bits and always produce a result. These are for small lookup tables.

Vector in-lane lookup table instructions

Syntax	Behavior
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,#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):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: HVX (slots 0,1,2,3)

Note:

- This instruction uses the HVX permute/shift resource.
- Input scalar register Rt is limited to registers 0 through 7

Encodings

Bitmap

Syntax	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
Vd.b=vlut32(Vu.b,Vv.b,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	0	0	1	d	d	d	d	d
Vd.b=vlut32(Vu.b,Vv.b,#u3)	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,Rt):nomatch	0	0	0	1	1	0	0	1	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d	d

Intrinsics

Vector in-lane lookup table intrinsics

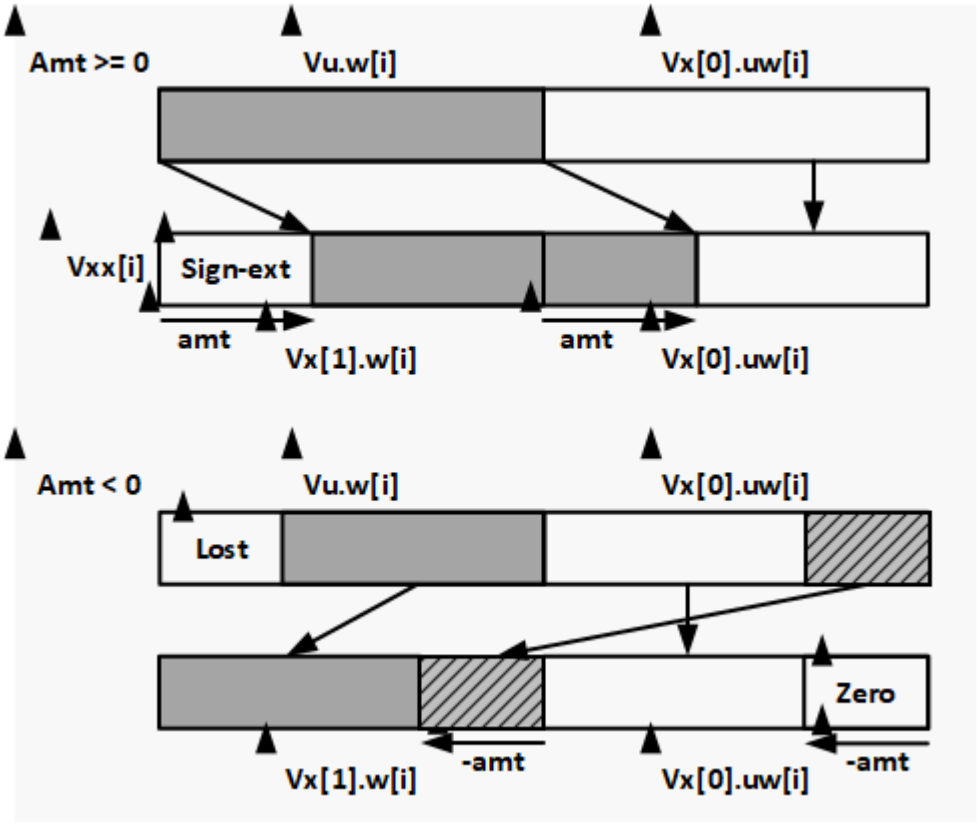
Vd.b=vlut32(Vu.b,Vv.b,Rt)	HVX_Vector Q6_Vb_vlut32_VbVbR(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vd.b=vlut32(Vu.b,Vv.b,#u3)	HVX_Vector Q6_Vb_vlut32_VbVbI(HVX_Vector Vu, HVX_Vector Vv, Word32 Iu3)
Vd.b=vlut32(Vu.b,Vv.b,Rt):nomatch	HVX_Vector Q6_Vb_vlut32_VbVbR_nomatch(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)

PERMUTE-SHIFT-RESOURCE

The HVX permute shift resource instruction subclass includes instructions that use both the HVX permute and shift resources.

Vector ASR overlay

The primary use of this instruction is to complete a 64b bi-dir ASR by shifting the high-word source(Vu.w[i]) and merging with the dest register. This assumes a rotate on the low-word source was already performed and placed in the low-word of the dest register. However, this instruction could also be used to concatenate LSB portions of the source and dest registers and placed into the high or low word of the destination depending on the shift amount.



Vector ASR overlay instructions

Syntax	Behavior
Vxx.w=vasrinto(Vu.w,Vv.w)	`for (i = 0; i < VELEM(32); i++) { shift = (Vu.w[i] << 32); mask = (((Vxx.v[0].w[i]) << 32) | Vxx.v[0].w[i]); lomask = (((1) << 32) - 1); count = -(0x40 & Vv.w[i]) + (Vv.w[i] & 0x3f); result = (count == -0x40) ? 0 : (((count < 0) ? ((shift << -(count)) | (mask & (lomask << -(count)))) : ((shift >> count) | (mask & (lomask >> count))))); Vxx.v[1].w[i] = ((result >> 32) & 0xffffffff); Vxx.v[0].w[i] = (result & 0xffffffff); }`

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

Note:

- This instruction uses the HVX shift or the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
Vxx.w=vasrinto(Vu.w,Vv.w)	0	0	0	1	1	0	1	0	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	x	x	x	x	x

Intrinsics

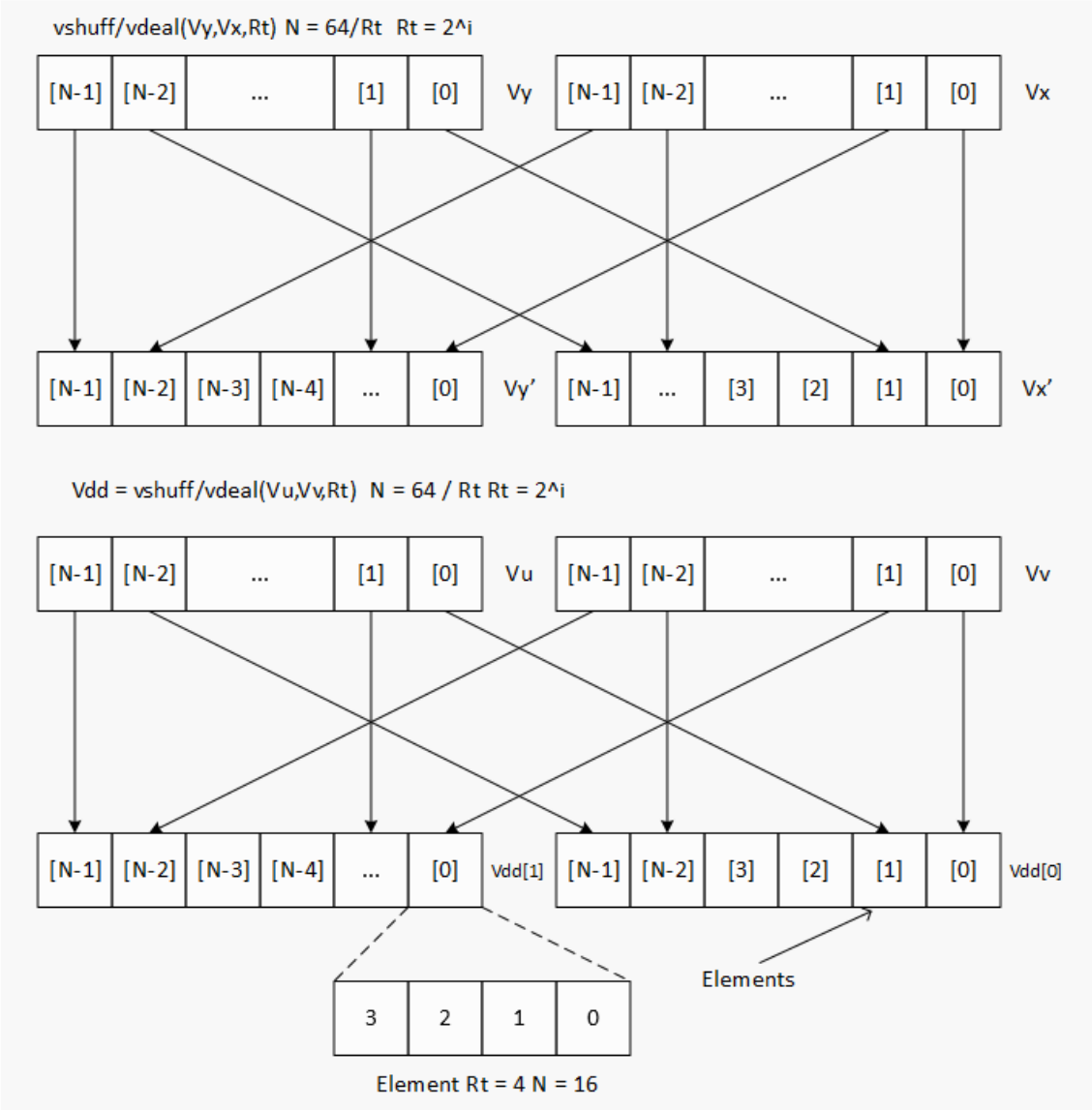
Vector ASR overlay intrinsics

Vxx.w=vasrinto(Vu.w,Vv.w)	HVX_VectorPair Q6_Ww_vasrinto_WwVwVw(HVX_VectorPair Vxx, HVX_Vector Vu, HVX_Vector Vv)

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 1 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, if Rt = 4 this indicates that each element contains 4 bytes. The matrix of 4 of these elements, 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.

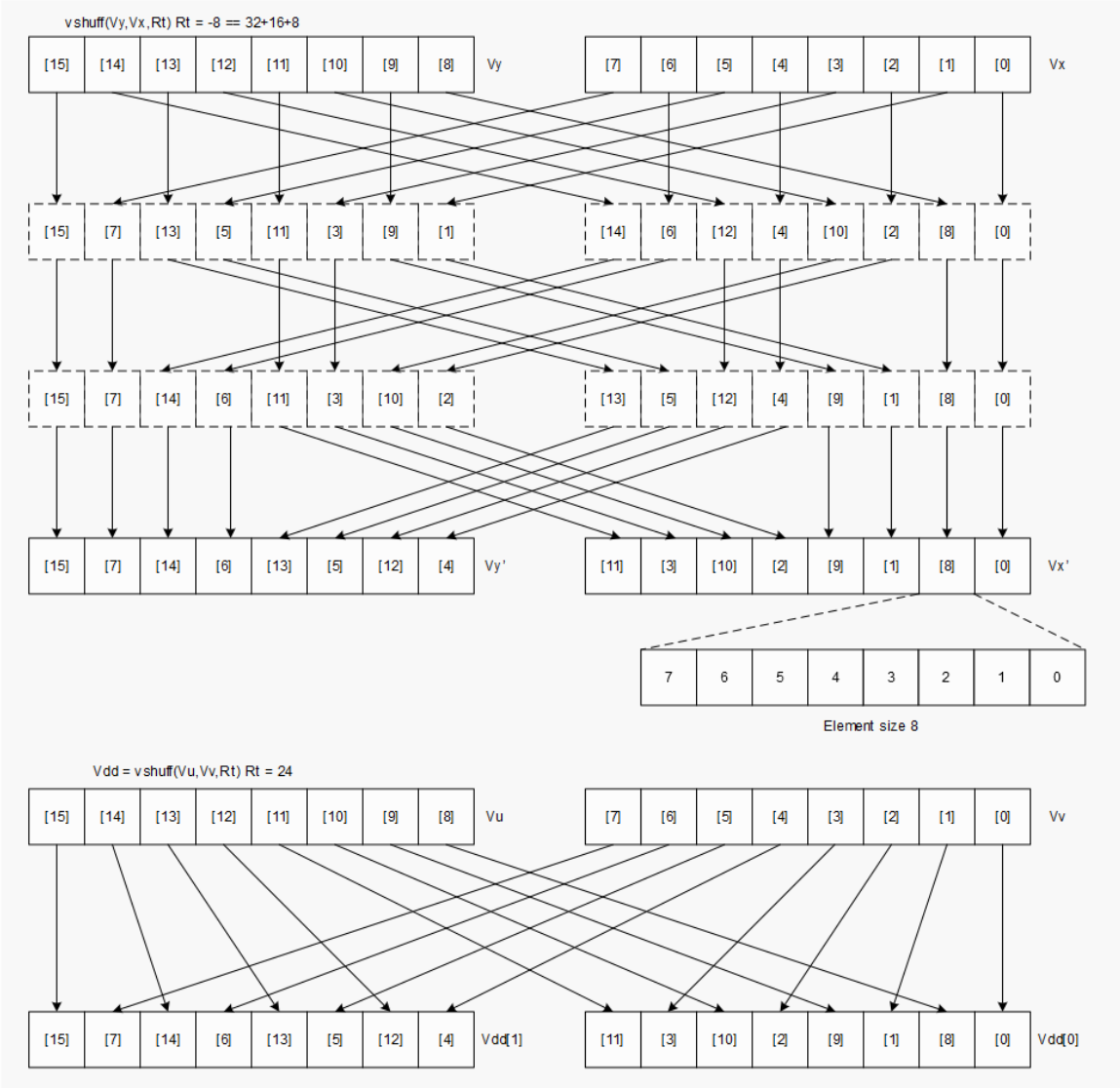
Examples for Rt = 1,2,4,8,16,32 are shown below. In these cases vdeal and vshuff perform the same operation. The diagram is valid for vshuff and vdeal.

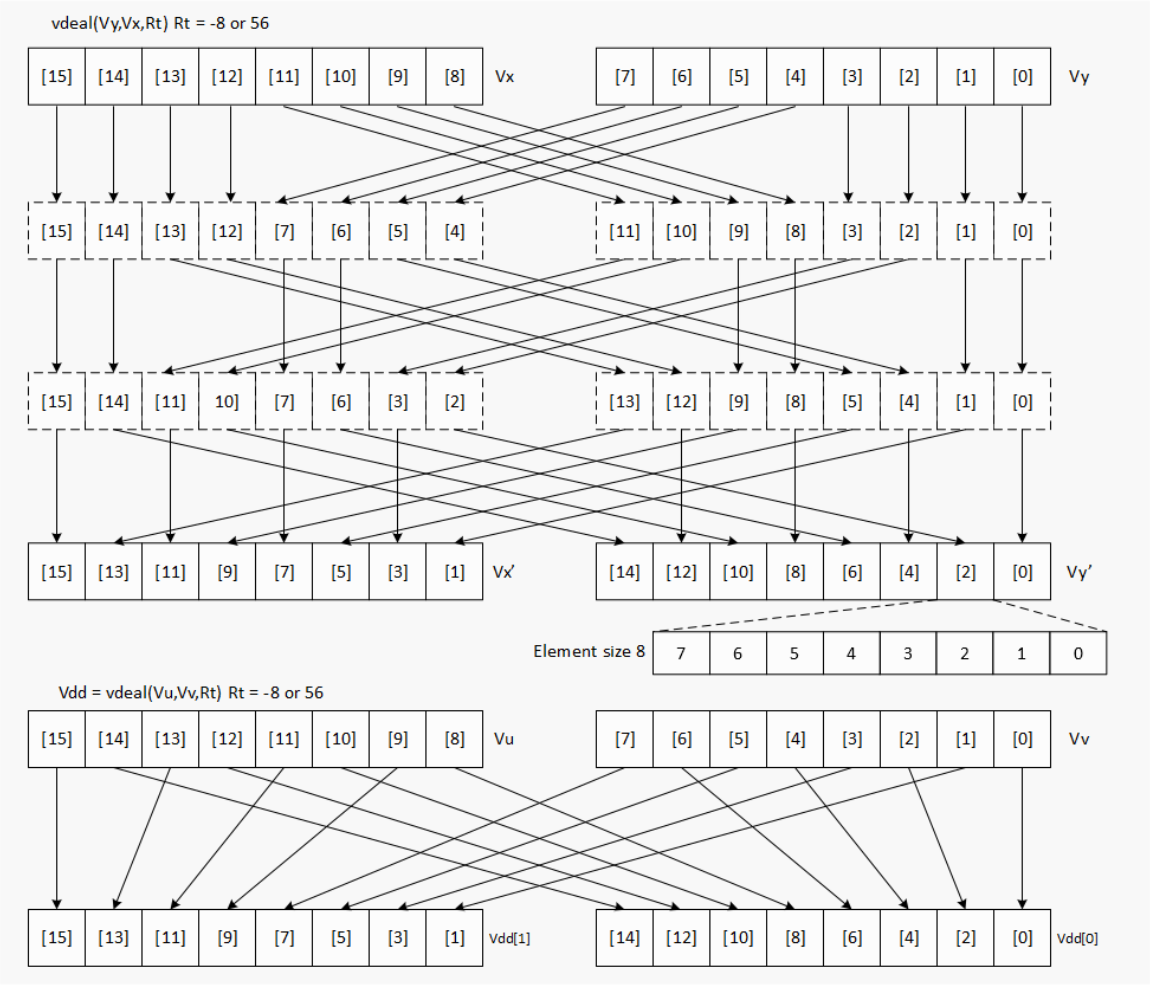


When a value of Rt other than 1,2,4,8,16,32 is used, the effect is a compound hierarchical transpose. For example, if the value 23 is used, 23 = 1+2+4+16. This indicates that the transformation is the same as performing the vshuff instruction with Rt=1, then Rt=2 on that result, then Rt = 4 on its result, 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 8, so 8 is the smallest element size. With a -ve value of Rt, all the upper bits of the value Rt are set.

For example, with Rt=-8 this is the same as 32+16+8. The diagram below shows the effect of this transform for both vshuff and vdeal.





If in addition to this family of transformations a block size is defined B, and the element size is defined as E, then if Rt = B - E, the resulting transformation will be a set of B contiguous blocks, each containing perfectly shuffled or dealt elements of element size E. Each block B will contain 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.

Vector shuffle and deal cross-lane instructions

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

Syntax	Behavior
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]); } } } }`
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]); } } } }`

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]);`
	`  }`
	`  }`
	`  }`
	`}`

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

Note:

- Input scalar register Rt is limited to registers 0 through 7
- This instruction uses the HVX shift or the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
vshuff(Vy,Vx,Rt)	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
Vdd=vshuff(Vu,Vv,Rt)	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
vdeal(Vy,Vx,Rt)	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
Vdd=vdeal(Vu,Vv,Rt)	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

Intrinsics

Vector shuffle and deal cross-lane intrinsics

Vdd=vshuff(Vu,Vv,Rt)	HVX_VectorPair Q6_W_vshuff_VVR(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
Vdd=vdeal(Vu,Vv,Rt)	HVX_VectorPair Q6_W_vdeal_VVR(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)

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).

8bit elements. </p>

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 5 bits of the input byte for the respective output byte. A control input register, Rt, contains match and select bits. The lower 3 bits of Rt must match the upper 3 bits of the input byte index in order for the table entry to be written to or Or'ed 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 as below

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

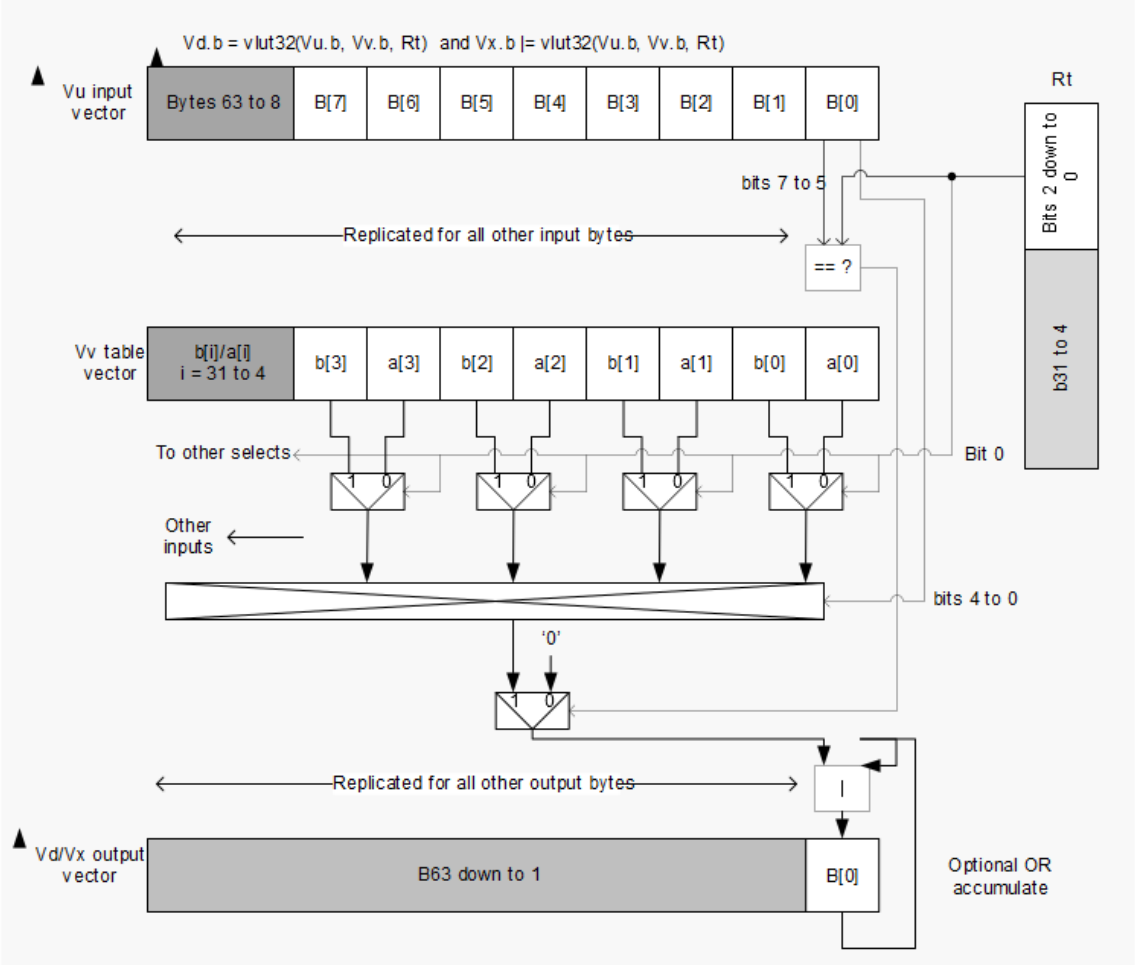
in order to prepare it for use with the vlut instruction in 64B mode it must be shuffled in blocks of 32 bytes

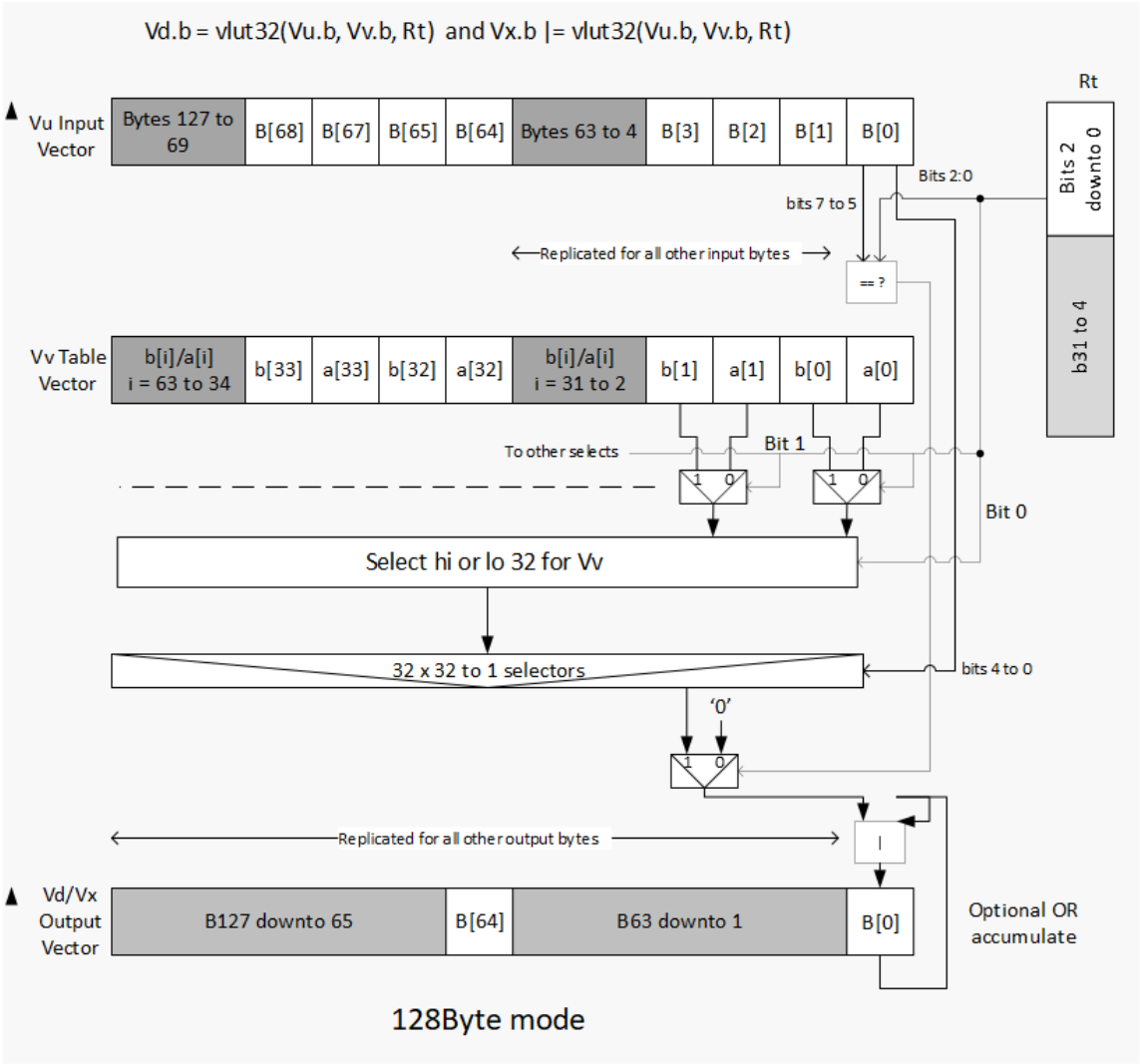
```
 63, 31, 62, 30,......36,  4, 35,  3, 34,  2, 33,  1, 32,  0 Rt=0,
Rt=1
127, 95,126, 94,.....100, 68, 99, 67, 98, 66, 97, 65, 96, 64 Rt=2,
Rt=3

same ordering for bytes 128-255       Rt=4, 5, 6, 7
```

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 will give the same results. In the case of 128B mode bit 1 of Rt selects whether to use the odd or even packed table and bit 0 chooses the high of low 32 elements of that hi or low table.





16bit elements. </p>

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 4 bits for the even destination register Vdd, the odd byte entries select table entries into the odd vector destination register Vdd. 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 4 bits of Rt must match the upper 4 bits of the input bytes in order for the table entry to be written to or Or'ed 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 4 of Rt. Bit 1 of Rt selects odd or even hwords and bit 0 selects the lower or upper 16 entries in the Vv register.

For larger than 32-element tables in the hword case (for example 256 entries), the user must access the main lookup table in 8 different 32 hword sections. If a 256H table is stored naturally in memory it would look as below

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

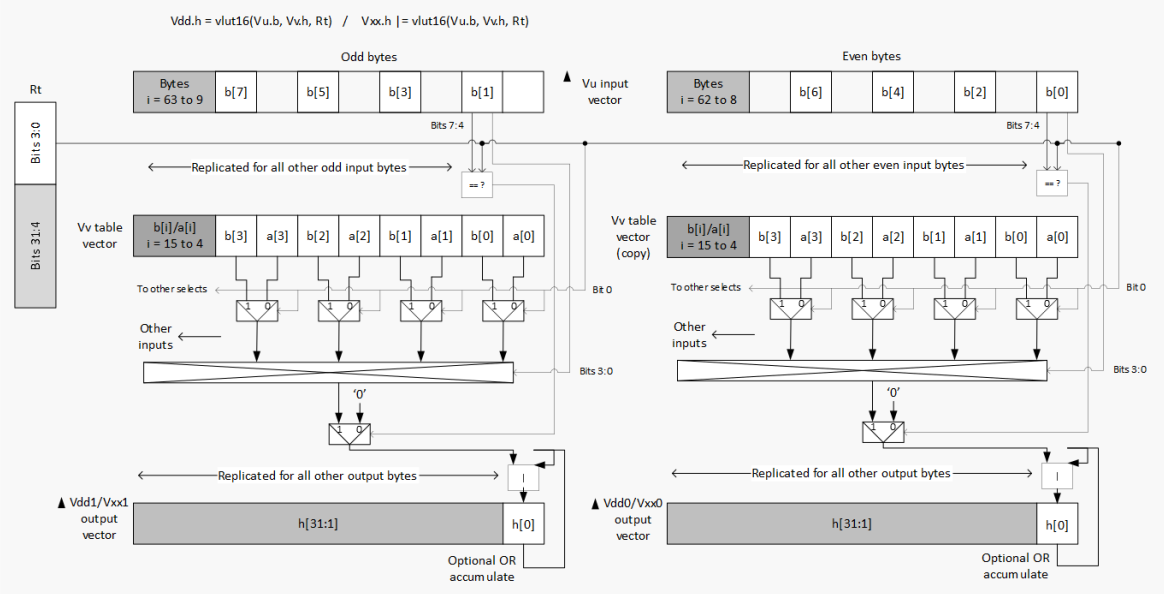
in order to prepare it for use with the vlut instruction in 64B mode it must be shuffled in blocks of 16 hwords, the LSB of Rt is used to choose the even or odd 16 entry hword tables in Vv.

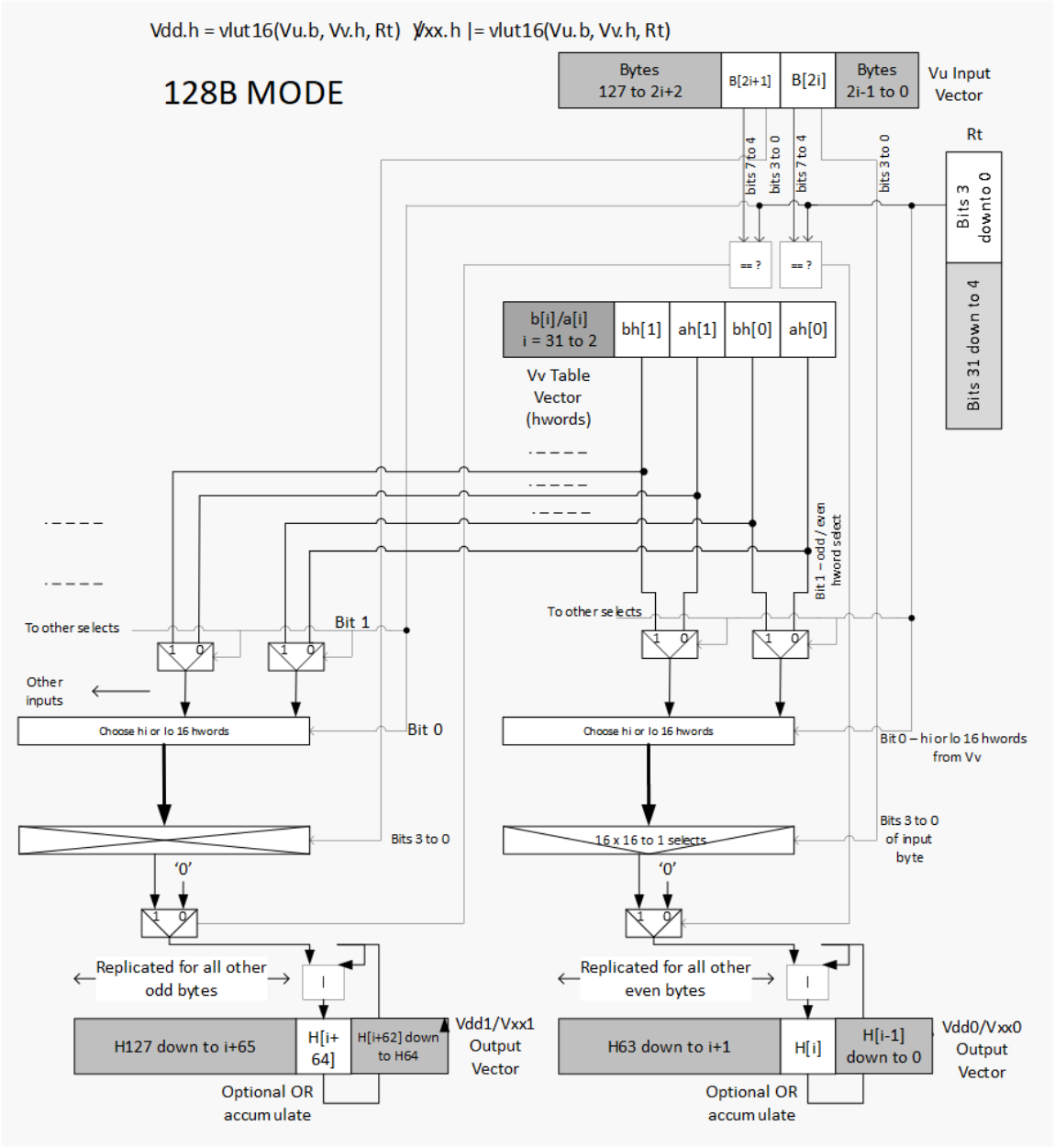
```
 31, 15, 30, 14,......20,  4, 19,  3, 18,  2, 17,  1, 16,  0 Rt=0,
Rt=1
 63, 47, 62, 46,..... 52, 36, 51, 35, 50, 34, 49, 33, 48, 32 Rt=2,
Rt=3

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 1 of Rt is used to choose between the even or odd 32 hwords in Vv. Bit 0 accesses the hi or lo 16 half words of the odd or even set.

```
 63, 31, 62, 30,........36,  4, 35,  3, 34,  2, 33,  1, 32,  0 Rt=0,1
Rt=2,3

same ordering for bytes 128-255     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.


Vdd.h = vlut16(Vu.b, Vv.h, Rt) / Vxx.h |= vlut16(Vu.b, Vv.h, Rt)
Odd bytes
Even bytes
Rt
Bits 3:0
Bits 31:4
Vu input vector
Bytes i = 63 to 9
Bytes i = 62 to 8
b[7]
b[5]
b[3]
b[1]
b[6]
b[4]
b[2]
b[0]
Bits 7:4
Replicated for all other odd input bytes
Replicated for all other even input bytes
==?
Vv table vector
Vv table vector (copy)
b[i]/a[i] i = 15 to 4
b[3]
a[3]
b[2]
a[2]
b[1]
a[1]
b[0]
a[0]
To other selects
Bit 0
Other inputs
Bits 3:0
'0'
Replicated for all other output bytes
Vdd1/Vxx1 output vector
Vdd0/Vxx0 output vector
h[31:1]
h[0]
Optional OR accumulate



vluts with the nomatch extension no not look at the upper bits and always produce a result. These are for small lookup tables.

Vector in-lane lookup table instructions

Syntax	Behavior
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; }`
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; }`
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; }`

Syntax	Behavior
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; }`
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; }`
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; }`

Syntax	Behavior

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];`
	`}`

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

Note:

- Input scalar register Rt is limited to registers 0 through 7
- This instruction uses the HVX shift or the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
Vx.b|=vlut32(Vu.b,Vv.b,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	1	x	x	x	x	x
Vdd.h=vlut16(Vu.b,Vv.h,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	1	0	d	d	d	d	d
Vxx.h|=vlut16(Vu.b,Vv.h,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	1	1	x	x	x	x	x
Vx.b|=vlut32(Vu.b,Vv.b,#u3)	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
Vdd.h=vlut16(Vu.b,Vv.h,#u3)	0	0	0	1	1	1	0	0	1	1	1	v	v	v	v	v	P	P	0	u	u	u	u	u	i	i	i	d	d	d	d	d
Vxx.h|=vlut16(Vu.b,Vv.h,#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
Vdd.h=vlut16(Vu.b,Vv.h,Rt):nomatch	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

Intrinsics

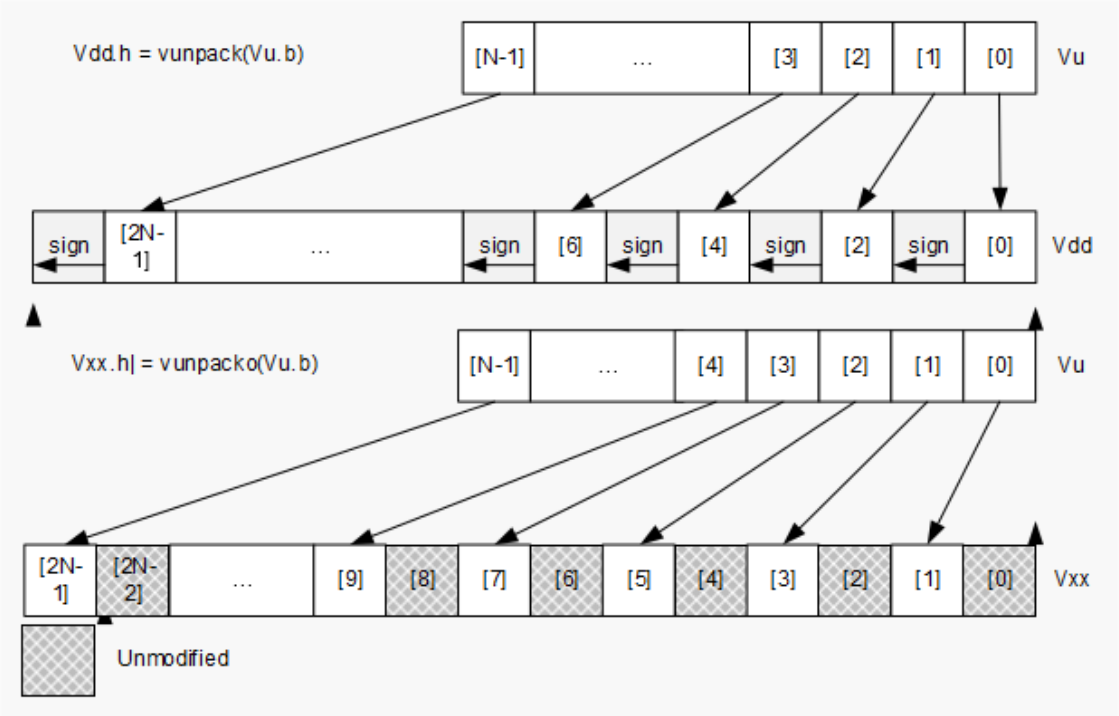
Vector in-lane lookup table intrinsics

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)
Vdd.h=vlut16(Vu.b,Vv.h,Rt)	HVX_VectorPair Q6_Wh_vlut16_VbVhR(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)
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)
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)
Vdd.h=vlut16(Vu.b,Vv.h,#u3)	HVX_VectorPair Q6_Wh_vlut16_VbVhI(HVX_Vector Vu, HVX_Vector Vv, Word32 Iu3)
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)
Vdd.h=vlut16(Vu.b,Vv.h,Rt):nomatch	HVX_VectorPair Q6_Wh_vlut16_VbVhR_nomatch(HVX_Vector Vu, HVX_Vector Vv, Word32 Rt)

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.



Unpack instructions

Syntax	Behavior
Vdd.uh=vunpack(Vu.ub)	`for (i = 0; i < VELEM(8); i++) { Vdd.uh[i] = Vu.ub[i]; }`

Syntax	Behavior
Vdd.h=vunpack(Vu.b)	`for (i = 0; i < VELEM(8); i++) { Vdd.h[i] = Vu.b[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: HVX (slots 0,1,2,3)

Note:

- This instruction uses the HVX shift or the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
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	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	0	d	d	d	d	d

Syntax	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
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	0	1	d	d	d	d	d
Vdd.w=vunpack(Vu.uh)	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
Vxx.h|=vunpacko(Vu.b)	0	0	0	1	1	1	1	0	-	-	0	-	-	0	0	0	P	P	1	u	u	u	u	u	0	0	0	x	x	x	x	x
Vxx.w|=vunpacko(Vu.h)	0	0	0	1	1	1	1	0	-	-	0	-	-	0	0	0	P	P	1	u	u	u	u	u	0	0	1	x	x	x	x	x

Intrinsics

Unpack intrinsics

Vdd.uh=vunpack(Vu.ub)	HVX_VectorPair Q6_Wuh_vunpack_Vub(HVX_Vector Vu)
Vdd.h=vunpack(Vu.b)	HVX_VectorPair Q6_Wh_vunpack_Vb(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)

SCATTER-DOUBLE-RESOURCE

The HVX scatter double resource instruction subclass includes instructions that perform scatter operations to the vector TCM.

Vector scatter

The following 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: Rt 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.

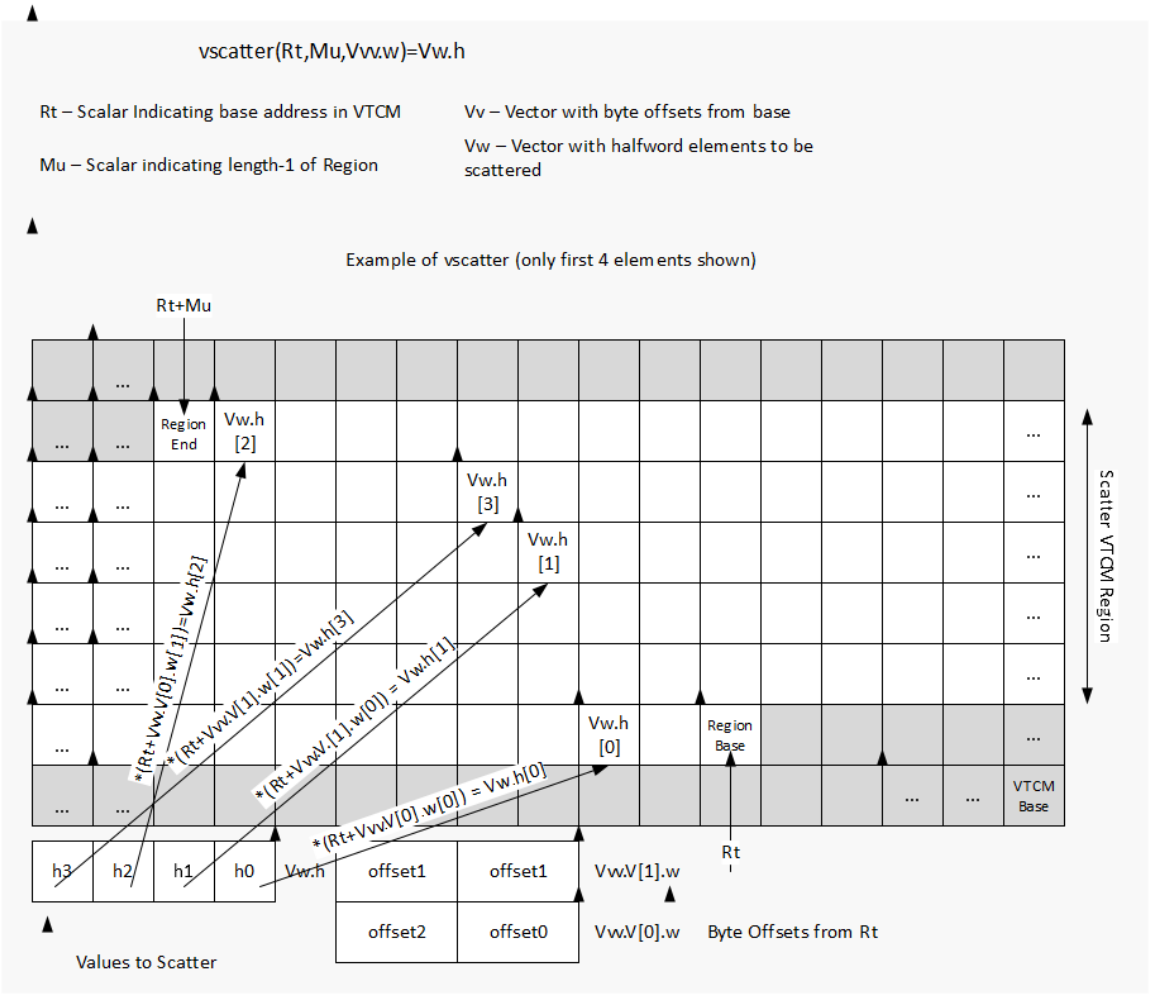
A vector register, Vvv, specifies byte offsets in this region. Elements of either halfword or word granularity, specified by Vw32, are sent to addresses pointed to by Rt + Vvv for each element. In the memory, the element is either write to memory or accumulated with the memory (scatter-accumulate).

If multiple values are written to the same memory location, ordering is not guaranteed. This applies to a single scatter or multiple scatters.

The offset vector, Vvv, 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 scatter region's end, it's simple dropped. Offsets must be positive otherwise they will be dropped.

All vectors registers can be used immediately after the scatter operation.



Vector scatter instructions

Syntax	Behavior

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

if (Qs4) vscatter(Rt,Mu,Vvv.w).h=Vw32	`MuV = MuV | (element_size-1);`
	`Rt = Rt & ~(element_size-1);`
	`for (i = 0; i < VELEM(32); i++) {`
	`  for(j = 0; j < 2; j++) {`
	`  EA = Rt+Vvv.v[j].uw[i];`
	`  if ( (Rt <= EA <= Rt + MuV) & QsV) *EA = VwV.w[i].uh[j];`
	`  }`
	`}`

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

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

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

Class: HVX (slots 0)

Note:

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

Encodings

Bitmap

Syntax	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
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	0	1	0	w	w	w	w	w
if (Qs4) 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	0	s	s	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

Intrinsics

Vector scatter intrinsics

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

SCATTER

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

Vector scatter

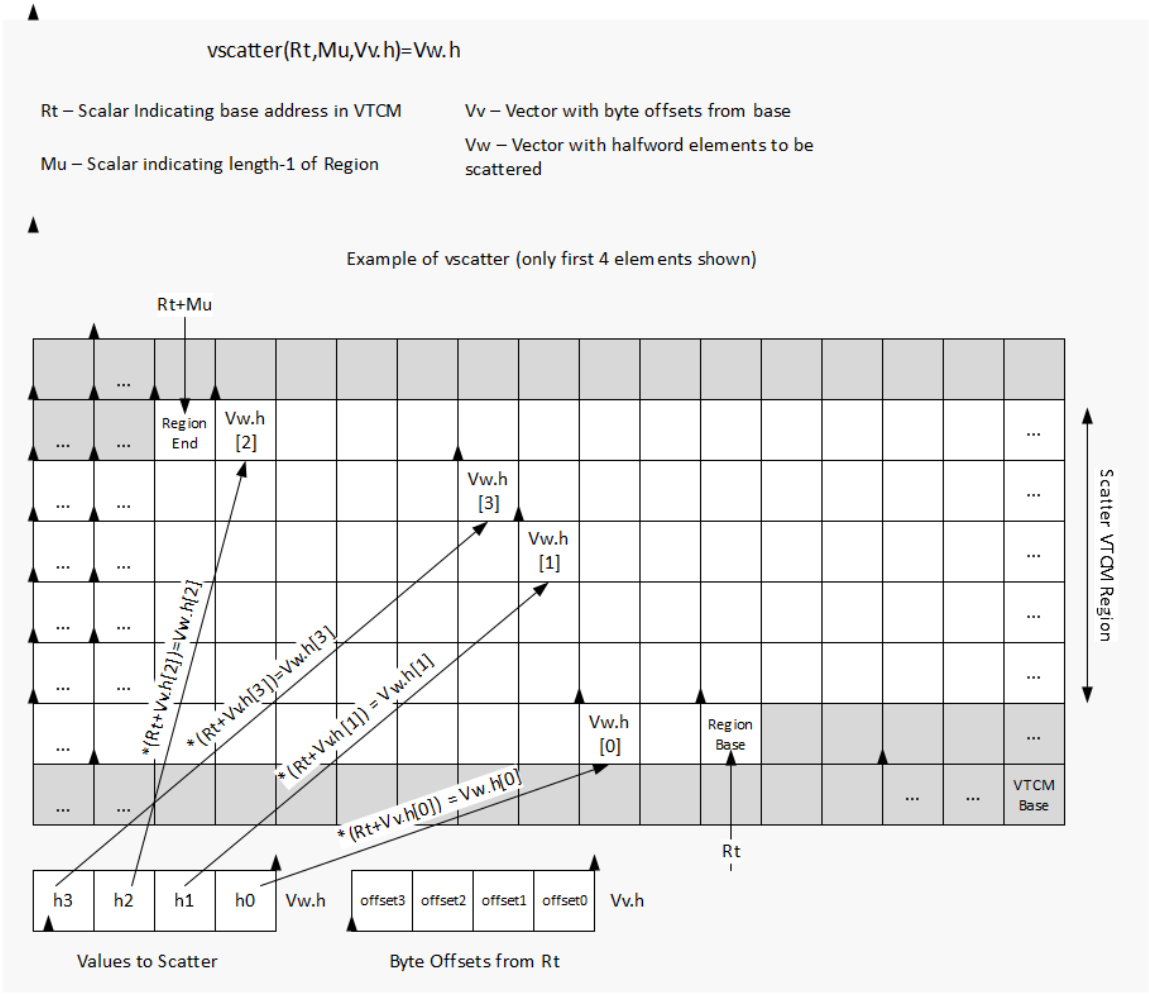
The following instructions perform scatter operations to the vector TCM. Scatter operations copy values from the register file to a region in VTCM. This region of memory is specified by two scalar registers: Rt is the base and Mu2 specified the length-1 of the region in bytes. This region must reside in VTCM and cannot cross a page boundary.

A vector register, Vv, specifies byte offsets in this region. Elements of either halfword or word granularity, specified by Vw32, are sent to addresses pointed to by Rt + Vv 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, Vv, 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's simple dropped. Offsets must be positive otherwise they will be dropped.

All vectors registers can be used immediately after the scatter operation.



Vector scatter instructions

Syntax	Behavior

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

Syntax	Behavior

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

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

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

if (Qs4) vscatter(Rt,Mu,Vv.w).w=Vw32	`MuV = MuV | (element_size-1);`
	`Rt = Rt & ~(element_size-1);`
	`for (i = 0; i < VELEM(32); i++) {`
	`  EA = Rt+Vv.uw[i];`
	`  if ( (Rt <= EA <= Rt + MuV) & QsV) *EA = VwV.uw[i];`
	`}`

Syntax	Behavior

if (Qs4) vscatter(Rt,Mu,Vv.h).h=Vw32	`MuV = MuV | (element_size-1);`
	`Rt = Rt & ~(element_size-1);`
	`for (i = 0; i < VELEM(16); i++) {`
	`EA = Rt+Vv.uh[i];`
	`if ( (Rt <= EA <= Rt + MuV) & QsV)`
	`*EA = VwV.uh[i];`
	`}`

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

Class: HVX (slots 0)

Note:

- This instruction can use any HVX resource.

Encodings

Bitmap

Syntax	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
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	0	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	0	0	1	w	w	w	w	w
vscatter(Rt,Mu,Vv.w).w+=Vw32	0	0	1	0	1	1	1	2	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.h).h+=Vw32	0	0	1	0	1	1	1	2	0	0	1	t	t	t	t	t	P	P	u	v	v	v	v	v	1	0	1	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	0	s	s	w	w	w	w	w
if (Qs4) vscatter(Rt,Mu,Vv.h).h=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

Intrinsics

Vector scatter intrinsics

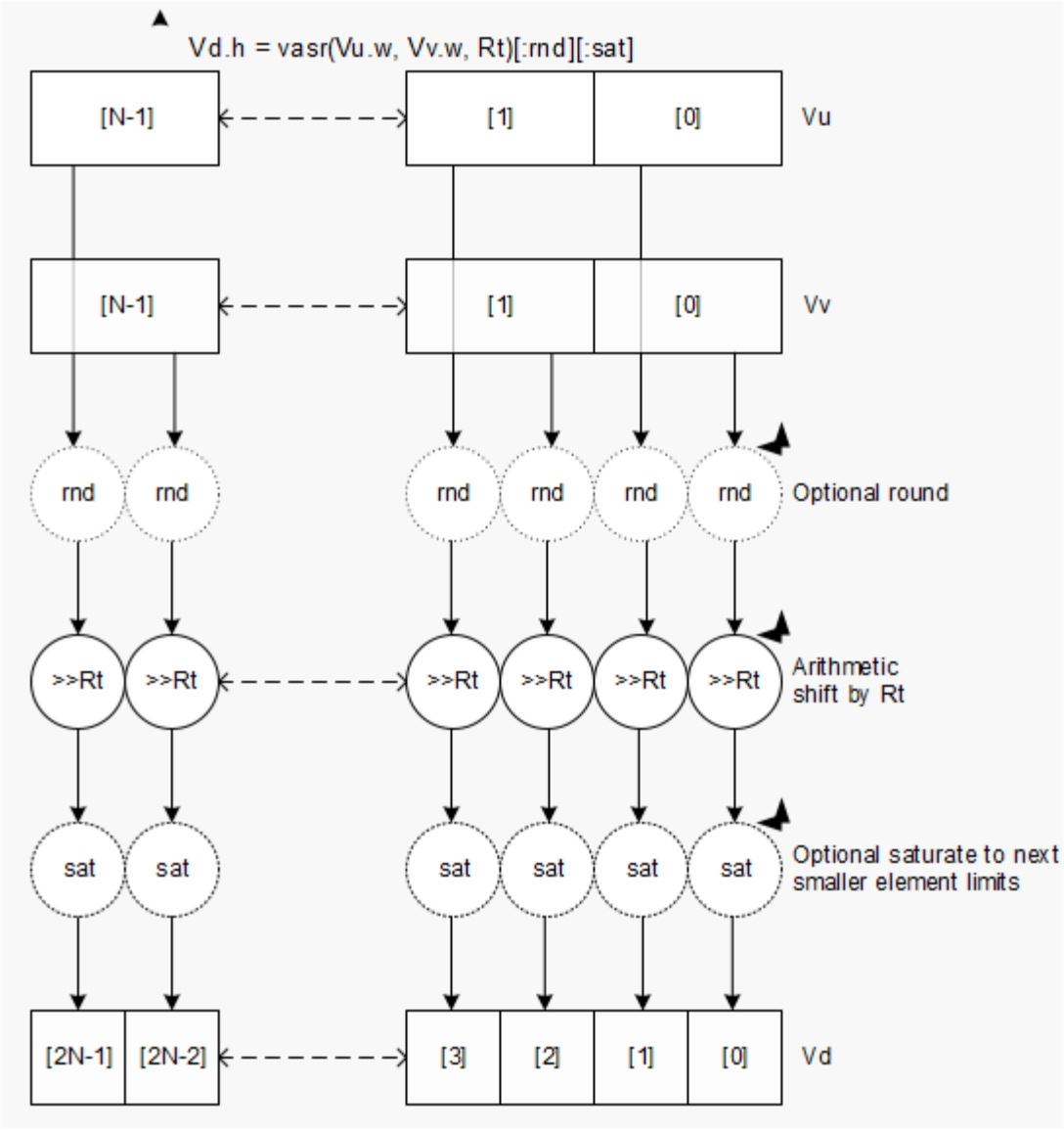
vscatter(Rt,Mu,Vv.w).w=Vw32	void Q6_vscatter_RMVwV(HVX_Vector* Rb, Word32 Mu, HVX_Vector Vv, HVX_Vector Vw)
vscatter(Rt,Mu,Vv.h).h=Vw32	void Q6_vscatter_RMVhV(HVX_Vector* Rb, Word32 Mu, HVX_Vector Vv, HVX_Vector Vw)
vscatter(Rt,Mu,Vv.w).w+=Vw32	void Q6_vscatteracc_RMVwV(HVX_Vector* Rb, Word32 Mu, HVX_Vector Vv, HVX_Vector Vw)
vscatter(Rt,Mu,Vv.h).h+=Vw32	void Q6_vscatteracc_RMVhV(HVX_Vector* Rb, Word32 Mu, HVX_Vector Vv, HVX_Vector Vw)
if (Qs4) vscatter(Rt,Mu,Vv.w).w=Vw32	void Q6_vscatter_QRMVwV(HVX_VectorPred Qs, HVX_Vector* Rb, Word32 Mu, HVX_Vector Vv, HVX_Vector Vw)
if (Qs4) vscatter(Rt,Mu,Vv.h).h=Vw32	void Q6_vscatter_QRMVhV(HVX_VectorPred Qs, HVX_Vector* Rb, Word32 Mu, HVX_Vector Vv, HVX_Vector Vw)

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.



Narrowing Shift instructions

Syntax	Behavior
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.uw,Vv.uw,Rt):sat	`for (i = 0; i < VELEM(32); i++) { shamt = Rt & 0xF; Vd.uw[i].h[0]=usat_16(Vv.uw[i] >> shamt); Vd.uw[i].h[1]=usat_16(Vu.uw[i] >> shamt); }`
Vd.ub=vasr(Vu.uh,Vv.uh,Rt):sat	`for (i = 0; i < VELEM(16); i++) { shamt = Rt & 0x7; Vd.uh[i].b[0]=usat_8(Vv.uh[i] >> shamt); Vd.uh[i].b[1]=usat_8(Vu.uh[i] >> 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); }`

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

Syntax	Behavior
Vd.uh=vasr(Vu.w,Vv.w,Rt):sat	`for (i = 0; i < VELEM(32); i++) { shamt = Rt & 0xF; Vd.w[i].h[0]=usat_16(Vv.w[i] >> shamt); Vd.w[i].h[1]=usat_16(Vu.w[i] >> shamt); }`
Vd.ub=vasr(Vu.h,Vv.h,Rt):sat	`for (i = 0; i < VELEM(16); i++) { shamt = Rt & 0x7; Vd.h[i].b[0]=usat_8(Vv.h[i] >> shamt); Vd.h[i].b[1]=usat_8(Vu.h[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.b=vasr(Vu.h,Vv.h,Rt):sat	`for (i = 0; i < VELEM(16); i++) { shamt = Rt & 0x7; Vd.h[i].b[0]=sat_8(Vv.h[i] >> shamt); Vd.h[i].b[1]=sat_8(Vu.h[i] >> shamt); }`

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);`
	`}`

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

Note:

- Input scalar register Rt is limited to registers 0 through 7
- This instruction uses the HVX shift or the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
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	0	1	d	d	d	d	d
Vd.uh=vasr(Vu.uw,Vv.uw,Rt):sat	0	0	0	1	1	0	0	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
Vd.ub=vasr(Vu.uh,Vv.uh,Rt):sat	0	0	0	1	1	0	1	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):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	1	1	d	d	d	d	d
Vd.h=vasr(Vu.w,Vv.w,Rt)	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	d	d	d	d
Vd.h=vasr(Vu.w,Vv.w,Rt):sat	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	d	d	d	d
Vd.h=vasr(Vu.w,Vv.w,Rt):rnd:sat	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	d	d	d	d
Vd.uh=vasr(Vu.w,Vv.w,Rt):rnd:sat	0	0	0	1	1	0	1	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):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.ub=vasr(Vu.h,Vv.h,Rt):sat	0	0	0	1	1	1	0	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):rnd:sat	0	0	0	1	1	1	0	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.b=vasr(Vu.h,Vv.h,Rt):sat	0	0	0	1	1	1	1	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):rnd:sat	0	0	0	1	1	1	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

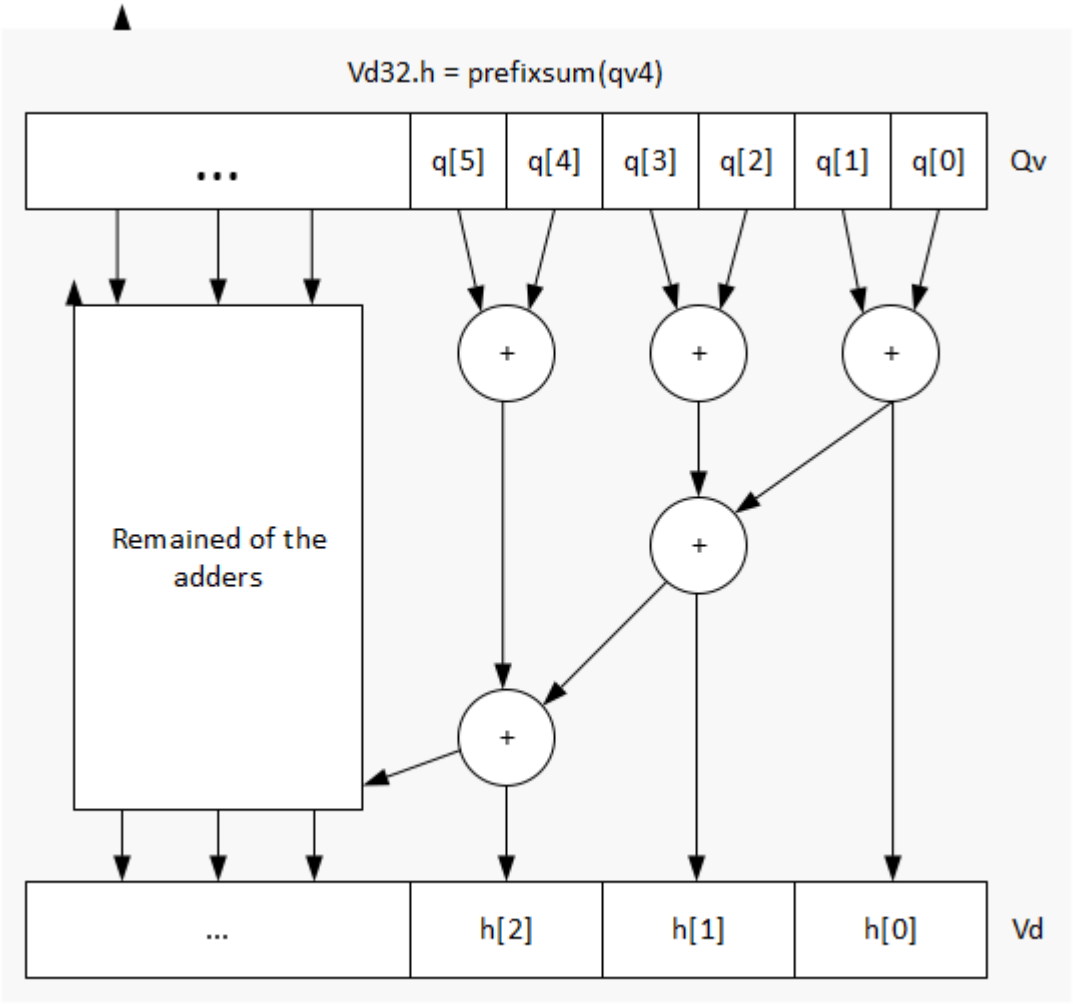
Intrinsics

Narrowing Shift intrinsics

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.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=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.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):sat	HVX_Vector Q6_Vh_vasr_VwVwR_sat(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.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.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.h,Vv.h,Rt):rnd:sat	HVX_Vector Q6_Vub_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.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)

Compute contiguous offsets for valid positions

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



Compute contiguous offsets for valid positions instructions

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: HVX (slots 0,1,2,3)

Note:

- This instruction only uses the shift resource and cannot use the permute/shift resource

Encodings

Bitmap

Syntax	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
Vd.b=prefixsum(Qv4)	0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	1	P	P	1	-	-	0	0	0	0	1	0	d	d	d	d	d
Vd.h=prefixsum(Qv4)	0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	1	P	P	1	-	-	0	0	1	0	1	0	d	d	d	d	d
Vd.w=prefixsum(Qv4)	0	0	0	1	1	1	1	0	v	v	0	-	-	0	1	1	P	P	1	-	-	0	1	0	0	1	0	d	d	d	d	d

Intrinsics

Compute contiguous offsets for valid positions 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)

HVX floating point add and subtract - half-precision

HVX floating point half-precision (16-bit) add and subtract.

HVX floating point add and subtract - half-precision instructions

Syntax	Behavior
Vd.qf16=vadd(Vu.qf16,Vv.qf16)	`for (i = 0; i < VELEM(16); i++) { u = parse_qf_to_unfloat(Vu.qf16[i]); v = parse_qf_to_unfloat(Vv.qf16[i]); exp = u.exp > v.exp ? u.exp : v.exp; Vd.qf16 = rnd_sat(ldexp(u.sig,u.exp-exp)+ldexp(v.sig,v.exp-exp),exp); }`
Vd.qf16=vadd(Vu.hf,Vv.hf)	`for (i = 0; i < VELEM(16); i++) { u = Vu.hf[i]; v = Vv.hf[i]; exp = u.exp > v.exp ? u.exp : v.exp; Vd.qf16 = rnd_sat(ldexp(u.sig,u.exp-exp)+ldexp(v.sig,v.exp-exp),exp); }`
Vd.qf16=vadd(Vu.qf16,Vv.hf)	`for (i = 0; i < VELEM(16); i++) { u = parse_qf_to_unfloat(Vu.qf16[i]); v = Vv.hf[i]; exp = u.exp > v.exp ? u.exp : v.exp; Vd.qf16 = rnd_sat(ldexp(u.sig,u.exp-exp)+ldexp(v.sig,v.exp-exp),exp); }`

Syntax	Behavior
Vd.qf16=vsub(Vu.qf16,Vv.qf16)	`for (i = 0; i < VELEM(16); i++) { u = parse_qf_to_unfloat(Vu.qf16[i]); v = parse_qf_to_unfloat(Vv.qf16[i]); v.sign = !v.sign; exp = u.exp > v.exp ? u.exp : v.exp; Vd.qf16 = rnd_sat(ldexp(u.sig,u.exp-exp)+ldexp(v.sig,v.exp-exp),exp); }`
Vd.qf16=vsub(Vu.hf,Vv.hf)	`for (i = 0; i < VELEM(16); i++) { u = Vu.hf[i]; v = Vv.hf[i]; v.sign = !v.sign; exp = u.exp > v.exp ? u.exp : v.exp; Vd.qf16 = rnd_sat(ldexp(u.sig,u.exp-exp)+ldexp(v.sig,v.exp-exp),exp); }`
Vd.qf16=vsub(Vu.qf16,Vv.hf)	`for (i = 0; i < VELEM(16); i++) { u = parse_qf_to_unfloat(Vu.qf16[i]); v = Vv.hf[i]; v.sign = !v.sign; exp = u.exp > v.exp ? u.exp : v.exp; Vd.qf16 = rnd_sat(ldexp(u.sig,u.exp-exp)+ldexp(v.sig,v.exp-exp),exp); }`

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

Note:

- This instruction uses the HVX shift or the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
Vd.qf16=vadd(Vu.qf16,Vv.qf16)	0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	0	d	d	d	d	d
Vd.qf16=vadd(Vu.hf,Vv.hf)	0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	d	d	d	d	d
Vd.qf16=vadd(Vu.qf16,Vv.hf)	0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	d	d	d	d	d
Vd.qf16=vsub(Vu.qf16,Vv.qf16)	0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	1	d	d	d	d	d
Vd.qf16=vsub(Vu.hf,Vv.hf)	0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	0	d	d	d	d	d
Vd.qf16=vsub(Vu.qf16,Vv.hf)	0	0	0	1	1	1	1	1	0	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	d	d	d	d	d

Intrinsics

HVX floating point add and subtract - half-precision intrinsics

Vd.qf16=vadd(Vu.qf16,Vv.qf16)	HVX_Vector Q6_Vqf16_vadd_Vqf16Vqf16(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf16=vadd(Vu.hf,Vv.hf)	HVX_Vector Q6_Vqf16_vadd_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf16=vadd(Vu.qf16,Vv.hf)	HVX_Vector Q6_Vqf16_vadd_Vqf16Vhf(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf16=vsub(Vu.qf16,Vv.qf16)	HVX_Vector Q6_Vqf16_vsub_Vqf16Vqf16(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf16=vsub(Vu.hf,Vv.hf)	HVX_Vector Q6_Vqf16_vsub_VhfVhf(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf16=vsub(Vu.qf16,Vv.hf)	HVX_Vector Q6_Vqf16_vsub_Vqf16Vhf(HVX_Vector Vu, HVX_Vector Vv)

HVX floating point add and subtract - single-precision

HVX floating point single-precision (32-bit) add and subtract.

HVX floating point add and subtract - single-precision instructions

Syntax	Behavior
Vd.qf32=vadd(Vu.qf32,Vv.qf32)	`for (i = 0; i < VELEM(32); i++) { u = parse_qf_to_unfloat(Vu.qf32[i]); v = parse_qf_to_unfloat(Vv.qf32[i]); exp = u.exp > v.exp ? u.exp : v.exp; Vd.qf32 = rnd_sat(ldexp(u.sig,u.exp-exp)+ldexp(v.sig,v.exp-exp),exp); }`
Vd.qf32=vadd(Vu.sf,Vv.sf)	`for (i = 0; i < VELEM(32); i++) { u = Vu.sf[i]; v = Vv.sf[i]; exp = u.exp > v.exp ? u.exp : v.exp; Vd.qf32 = rnd_sat(ldexp(u.sig,u.exp-exp)+ldexp(v.sig,v.exp-exp),exp); }`
Vd.qf32=vadd(Vu.qf32,Vv.sf)	`for (i = 0; i < VELEM(32); i++) { u = parse_qf_to_unfloat(Vu.qf32[i]); v = Vv.sf[i]; exp = u.exp > v.exp ? u.exp : v.exp; Vd.qf32 = rnd_sat(ldexp(u.sig,u.exp-exp)+ldexp(v.sig,v.exp-exp),exp); }`
Vd.qf32=vsub(Vu.qf32,Vv.qf32)	`for (i = 0; i < VELEM(32); i++) { u = parse_qf_to_unfloat(Vu.qf32[i]); v = parse_qf_to_unfloat(Vv.qf32[i]); v.sign = !v.sign; exp = u.exp > v.exp ? u.exp : v.exp; Vd.qf32 = rnd_sat(ldexp(u.sig,u.exp-exp)+ldexp(v.sig,v.exp-exp),exp); }`

Syntax	Behavior
Vd.qf32=vsub(Vu.sf,Vv.sf)	`for (i = 0; i < VELEM(32); i++) { u = Vu.sf[i]; v = Vv.sf[i]; v.sign = !v.sign; exp = u.exp > v.exp ? u.exp : v.exp; Vd.qf32 = rnd_sat(ldexp(u.sig,u.exp-exp)+ldexp(v.sig,v.exp-exp),exp); }`
Vd.qf32=vsub(Vu.qf32,Vv.sf)	`for (i = 0; i < VELEM(32); i++) { u = parse_qf_to_unfloat(Vu.qf32[i]); v = Vv.sf[i]; v.sign = !v.sign; exp = u.exp > v.exp ? u.exp : v.exp; Vd.qf32 = rnd_sat(ldexp(u.sig,u.exp-exp)+ldexp(v.sig,v.exp-exp),exp); }`

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

Note:

- This instruction uses the HVX shift or the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
Vd.qf32=vadd(Vu.qf32,Vv.qf32)	0	0	0	1	1	0	1	0	1	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	d	d	d	d	d
Vd.qf32=vadd(Vu.sf,Vv.sf)	0	0	0	1	1	0	1	0	1	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	1	d	d	d	d	d
Vd.qf32=vadd(Vu.qf32,Vv.sf)	0	0	0	1	1	0	1	0	1	1	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	0	d	d	d	d	d
Vd.qf32=vsub(Vu.qf32,Vv.qf32)	0	0	0	1	1	0	1	0	1	1	0	1	v	v	v	v	v	P	P	1	u	u	u	u	u	0	1	1	d	d	d	d	d
Vd.qf32=vsub(Vu.sf,Vv.sf)	0	0	0	1	1	0	1	0	1	1	1	1	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	0	d	d	d	d	d
Vd.qf32=vsub(Vu.qf32,Vv.sf)	0	0	0	1	1	0	1	0	1	1	1	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	0	1	d	d	d	d	d

Intrinsics

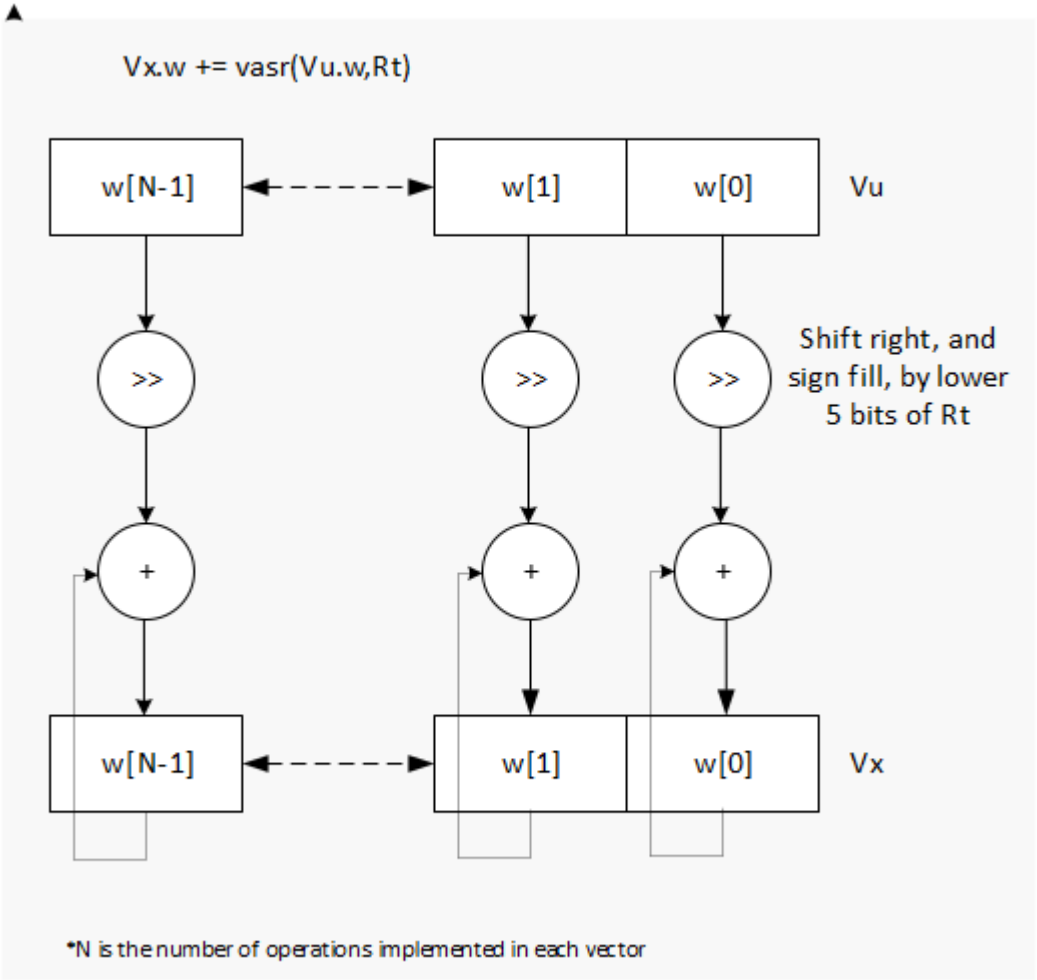
HVX floating point add and subtract - single-precision intrinsics

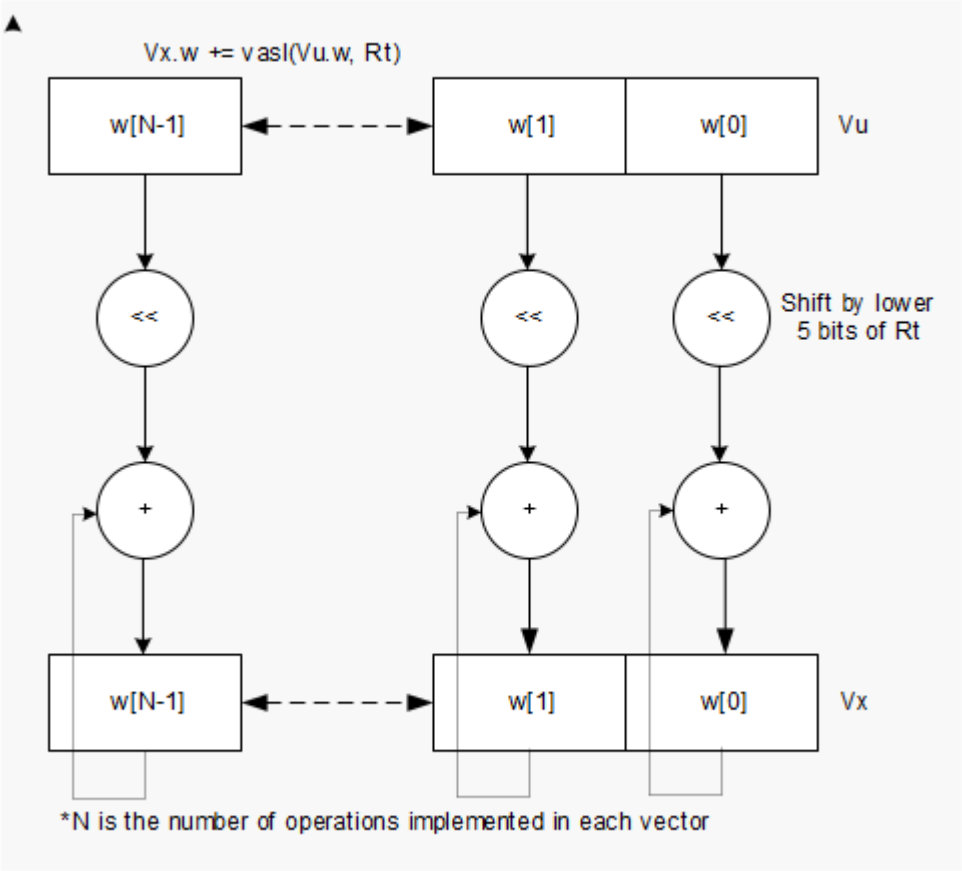
Vd.qf32=vadd(Vu.qf32,Vv.qf32)	HVX_Vector Q6_Vqf32_vadd_Vqf32Vqf32(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf32=vadd(Vu.sf,Vv.sf)	HVX_Vector Q6_Vqf32_vadd_VsfVsf(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf32=vadd(Vu.qf32,Vv.sf)	HVX_Vector Q6_Vqf32_vadd_Vqf32Vsf(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf32=vsub(Vu.qf32,Vv.qf32)	HVX_Vector Q6_Vqf32_vsub_Vqf32Vqf32(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf32=vsub(Vu.sf,Vv.sf)	HVX_Vector Q6_Vqf32_vsub_VsfVsf(HVX_Vector Vu, HVX_Vector Vv)
Vd.qf32=vsub(Vu.qf32,Vv.sf)	HVX_Vector Q6_Vqf32_vsub_Vqf32Vsf(HVX_Vector Vu, HVX_Vector Vv)

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 then added to the destination vector register Vx. For signed word shifts the lower 5 bits of Rt specify the shift amount.

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





Shift and add instructions

Syntax	Behavior
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))); }`

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))); }`

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

Note:

- This instruction uses the HVX shift or the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
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	0	1	0	x	x	x	x	x
Vx.w+=vasr(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.h+=vasl(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+=vasr(Vu.h,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

Intrinsics

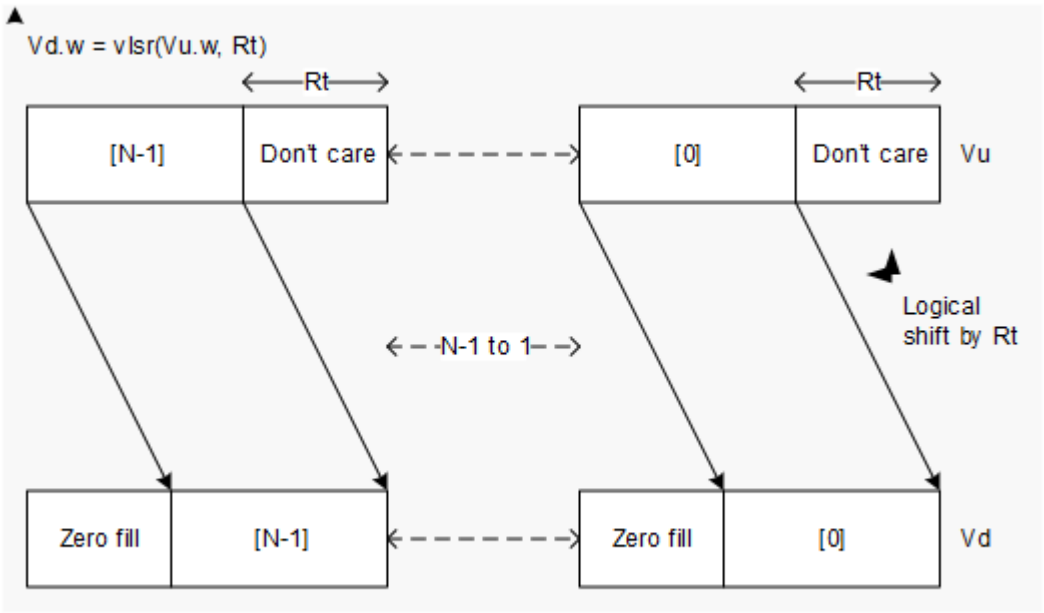
Shift and add intrinsics

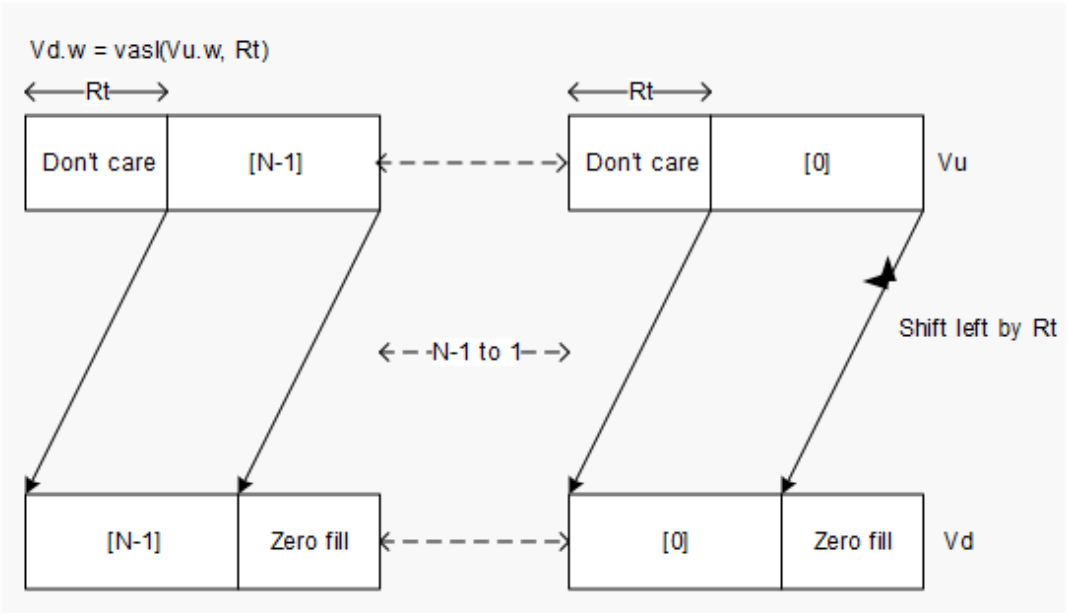
Vx.w+=vasl(Vu.w,Rt)	HVX_Vector Q6_Vw_vaslacc_VwVwR(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)
Vx.w+=vasr(Vu.w,Rt)	HVX_Vector Q6_Vw_vasracc_VwVwR(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)
Vx.h+=vasl(Vu.h,Rt)	HVX_Vector Q6_Vh_vaslacc_VhVhR(HVX_Vector Vx, HVX_Vector Vu, Word32 Rt)
Vx.h+=vasr(Vu.h,Rt)	HVX_Vector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

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 4 bits are used, while for word shifts the lower 5 bits are used.

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





Shift instructions

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

Syntax	Behavior
Vd.w=vasr(Vu.w,Vv.w)	`for (i = 0; i < VELEM(32); i++) { Vd.w[i] = (sxt_(5+1)_to_32(Vv.w[i])> 0)?(Vu.w[i]>>sxt_(5+1)_to_32(Vv.w[i])): (Vu.w[i]<<sxt_(5+1)_to_32(Vv.w[i])); }`
Vd.w=vasl(Vu.w,Vv.w)	`for (i = 0; i < VELEM(32); i++) { Vd.w[i] = (sxt_(5+1)_to_32(Vv.w[i])> 0)?(Vu.w[i]<<sxt_(5+1)_to_32(Vv.w[i])): (Vu.w[i]>>sxt_(5+1)_to_32(Vv.w[i])); }`
Vd.w=vlsr(Vu.w,Vv.w)	`for (i = 0; i < VELEM(32); i++) { Vd.uw[i] = (sxt_(5+1)_to_32(Vv.w[i])> 0)?(Vu.uw[i]>>>sxt_(5+1)_to_32(Vv.w[i])): (Vu.uw[i]<<sxt_(5+1)_to_32(Vv.w[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=vasl(Vu.h,Rt)	`for (i = 0; i < VELEM(16); i++) { Vd.h[i] = (Vu.h[i] << (Rt & (16-1))); }`
Vd.uh=vlsr(Vu.uh,Rt)	`for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = (Vu.uh[i] >> (Rt & (16-1))); }`

Syntax	Behavior
Vd.h=vasr(Vu.h,Vv.h)	`for (i = 0; i < VELEM(16); i++) { Vd.h[i] = (sxt_(4+1)_to_16(Vv.h[i])> 0)?(Vu.h[i]>>sxt_(4+1)_to_16(Vv.h[i])): (Vu.h[i]<<sxt_(4+1)_to_16(Vv.h[i])); }`
Vd.h=vasl(Vu.h,Vv.h)	`for (i = 0; i < VELEM(16); i++) { Vd.h[i] = (sxt_(4+1)_to_16(Vv.h[i])> 0)?(Vu.h[i]<<sxt_(4+1)_to_16(Vv.h[i])): (Vu.h[i]>>sxt_(4+1)_to_16(Vv.h[i])); }`
Vd.h=vlsr(Vu.h,Vv.h)	`for (i = 0; i < VELEM(16); i++) { Vd.uh[i] = (sxt_(4+1)_to_16(Vv.h[i])> 0)?(Vu.uh[i]>>>sxt_(4+1)_to_16(Vv.h[i])): (Vu.uh[i]<<sxt_(4+1)_to_16(Vv.h[i])); }`
Vd.ub=vlsr(Vu.ub,Rt)	`for (i = 0; i < VELEM(8); i++) { Vd.b[i] = Vu.ub[i] >> (Rt & 0x7); }`

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

Note:

- This instruction uses the HVX shift or the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
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	0	1	d	d	d	d	d
Vd.w=vasl(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	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	0	1	d	d	d	d	d

Syntax	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
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	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	0	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	0	1	d	d	d	d	d
Vd.h=vasr(Vu.h,Rt)	0	0	1	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=vasl(Vu.h,Rt)	0	0	1	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.uh=vlsr(Vu.uh,Rt)	0	0	1	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.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	0	1	1	d	d	d	d	d
Vd.h=vasl(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	1	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	0	d	d	d	d	d
Vd.ub=vlsr(Vu.ub,Rt)	0	0	1	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

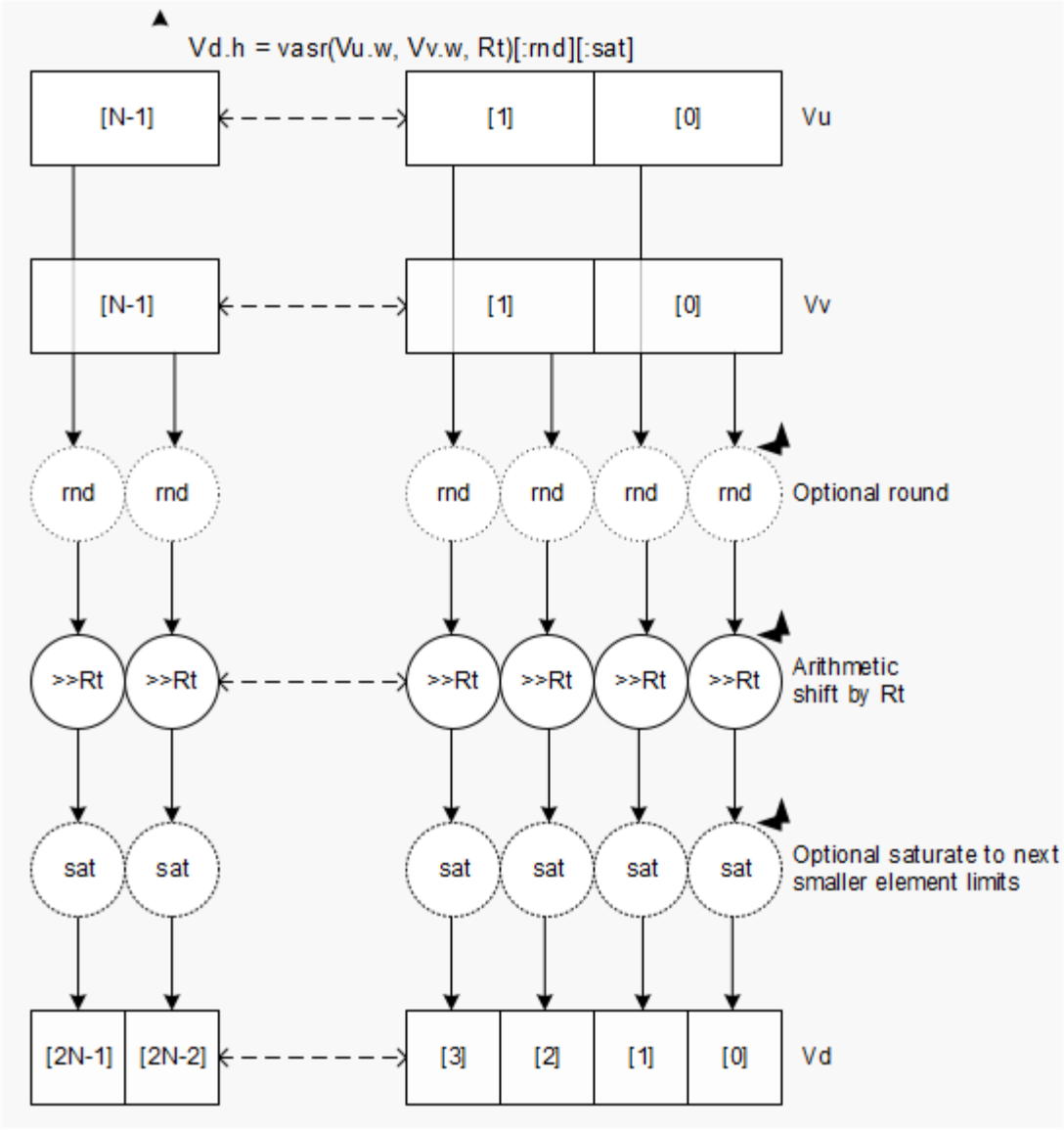
Intrinsics

Shift intrinsics

Vd.w=vasr(Vu.w,Rt)	HVX_Vector Q6_Vw_vasr_VwR(HVX_Vector Vu, Word32 Rt)
Vd.w=vasl(Vu.w,Rt)	HVX_Vector Q6_Vw_vasl_VwR(HVX_Vector Vu, Word32 Rt)
Vd.uw=vlsr(Vu.uw,Rt)	HVX_Vector Q6_Vuw_vlsr_VuwR(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=vasl(Vu.w,Vv.w)	HVX_Vector Q6_Vw_vasl_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)
Vd.h=vasr(Vu.h,Rt)	HVX_Vector Q6_Vh_vasr_VhR(HVX_Vector Vu, Word32 Rt)
Vd.h=vasl(Vu.h,Rt)	HVX_Vector Q6_Vh_vasl_VhR(HVX_Vector Vu, Word32 Rt)
Vd.uh=vlsr(Vu.uh,Rt)	HVX_Vector Q6_Vuh_vlsr_VuhR(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=vasl(Vu.h,Vv.h)	HVX_Vector Q6_Vh_vasl_VhVh(HVX_Vector Vu, HVX_Vector Vv)
Vd.h=vlsr(Vu.h,Vv.h)	HVX_Vector Q6_Vh_vlsr_VhVh(HVX_Vector Vu, HVX_Vector Vv)
Vd.ub=vlsr(Vu.ub,Rt)	HVX_Vector Q6_Vub_vlsr_VubR(HVX_Vector Vu, Word32 Rt)

Narrowing Shift by Vector

Arithmetically shift-right the elements in vector register pair Vuu by the lower bits of the elements in vector register Vv. 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 Vu+1, and each odd element in Vd comes from the vector register Vu.



Narrowing Shift by Vector instructions

Syntax	Behavior
Vd.uh=vasr(Vuu.w,Vv.uh):sat	`for (i = 0; i < VELEM(32); i++) { shamt = Vv.uh[2*i+0] & 0xF; Vd.w[i].h[0]=usat_16(Vuu.v[0].w[i] >> shamt); shamt = Vv.uh[2*i+1] & 0xF; Vd.w[i].h[1]=usat_16(Vuu.v[1].w[i] >> shamt); }`
Vd.uh=vasr(Vuu.w,Vv.uh):rnd:sat	`for (i = 0; i < VELEM(32); i++) { shamt = Vv.uh[2*i+0] & 0xF; Vd.w[i].h[0]=usat_16(Vuu.v[0].w[i] + (1<<(shamt-1)) >> shamt); shamt = Vv.uh[2*i+1] & 0xF; Vd.w[i].h[1]=usat_16(Vuu.v[1].w[i] + (1<<(shamt-1)) >> shamt); }`
Vd.ub=vasr(Vuu.uh,Vv.ub):sat	`for (i = 0; i < VELEM(16); i++) { shamt = Vv.ub[2*i+0] & 0x7; Vd.uh[i].b[0]=usat_8(Vuu.v[0].uh[i] >> shamt); shamt = Vv.ub[2*i+1] & 0x7; Vd.uh[i].b[1]=usat_8(Vuu.v[1].uh[i] >> shamt); }`
Vd.ub=vasr(Vuu.uh,Vv.ub):rnd:sat	`for (i = 0; i < VELEM(16); i++) { shamt = Vv.ub[2*i+0] & 0x7; Vd.uh[i].b[0]=usat_8(Vuu.v[0].uh[i] + (1<<(shamt-1)) >> shamt); shamt = Vv.ub[2*i+1] & 0x7; Vd.uh[i].b[1]=usat_8(Vuu.v[1].uh[i] + (1<<(shamt-1)) >> shamt); }`

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

Note:

- This instruction cannot be paired with a HVX permute instruction
- This instruction only uses the shift resource and cannot use the permute/shift resource
- If a packet contains this instruction and a HVX ALU op then the ALU OP must be unary.

Encodings

Bitmap

Syntax	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
Vd.uh=vasr(Vuu.w,Vv.uh):sat	0	0	v	0	v	1	1	0	1	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	0	d	d	d	d
Vd.uh=vasr(Vuu.w,Vv.uh):rnd:sat	0	0	v	0	v	1	1	0	1	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	0	1	d	d	d	d
Vd.ub=vasr(Vuu.uh,Vv.ub):sat	0	0	u	0	v	1	1	0	1	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	0	d	d	d	d
Vd.ub=vasr(Vuu.uh,Vv.ub):rnd:sat	0	0	u	0	v	1	1	0	1	0	0	0	v	v	v	v	v	P	P	0	u	u	u	u	u	0	1	1	d	d	d	d

Intrinsics

Narrowing Shift by Vector intrinsics

Vd.uh=vasr(Vuu.w,Vv.uh):sat	HVX_Vector Q6_Vuh_vasr_WwVuh_sat(HVX_VectorPair Vuu, HVX_Vector Vv)
Vd.uh=vasr(Vuu.w,Vv.uh):rnd:sat	HVX_Vector Q6_Vuh_vasr_WwVuh_rnd_sat(HVX_VectorPair Vuu, HVX_Vector Vv)
Vd.ub=vasr(Vuu.uh,Vv.ub):sat	HVX_Vector Q6_Vub_vasr_WuhVub_sat(HVX_VectorPair Vuu, HVX_Vector Vv)
Vd.ub=vasr(Vuu.uh,Vv.ub):rnd:sat	HVX_Vector Q6_Vub_vasr_WuhVub_rnd_sat(HVX_VectorPair Vuu, HVX_Vector Vv)

Convert IEEE floating point to integer

Convert from sf/hf to w/h through the round to zero rounding mode, where the floating-point fractional is truncated.

Convert IEEE floating point to integer instructions

Syntax	Behavior
Vd.w=Vu.sf	`for (i = 0; i < VELEM(32); i++) { Vd.w[i] = conv_w_sf(Vu.sf[i]); }`
Vd.h=Vu.hf	`for (i = 0; i < VELEM(16); i++) { Vd.h[i] = conv_h_hf(Vu.hf[i]); }`

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

Note:

- This instruction uses the HVX shift or the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
Vd.w=Vu.sf	0	0	0	1	1	1	1	0	-	-	0	-	-	1	0	1	P	P	1	u	u	u	u	u	0	0	1	d	d	d	d	d
Vd.h=Vu.hf	0	0	0	1	1	1	1	0	-	-	0	-	-	1	0	1	P	P	1	u	u	u	u	u	0	1	0	d	d	d	d	d

Intrinsics

Convert IEEE floating point to integer intrinsics

Vd.w=Vu.sf	HVX_Vector Q6_Vw_equals_Vsf(HVX_Vector Vu)
Vd.h=Vu.hf	HVX_Vector Q6_Vh_equals_Vhf(HVX_Vector Vu)

Convert to IEEE floating point

Convert from qf32/w/qf16/h to sf/hf

Convert to IEEE floating point instructions

Syntax	Behavior
Vd.sf=Vu.qf32	`for (i = 0; i < VELEM(32); i++) { Vd.sf[i] = Vu.qf32[i]; }`
Vd.hf=Vu.qf16	`for (i = 0; i < VELEM(16); i++) { Vd.hf[i] = Vu.qf16[i]; }`
Vd.hf=Vuu.qf32	`for (i = 0; i < VELEM(32); i++) { Vd.hf[2*i] = Vuu.v[0].qf32[i]; Vd.hf[2*i+1] = Vuu.v[1].qf32[i]; }`
Vd.sf=Vu.w	`for (i = 0; i < VELEM(32); i++) { Vd.sf[i] = conv_sf_w(Vu.w[i]); }`
Vd.hf=Vu.h	`for (i = 0; i < VELEM(16); i++) { Vd.hf[i] = conv_hf_h(Vu.h[i]); }`

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

Note:

- This instruction uses the HVX shift or the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
Vd.sf=Vu.qf32	0	0	0	1	1	1	1	0	-	-	0	-	-	1	0	0	P	P	1	u	u	u	u	u	0	0	0	d	d	d	d	d
Vd.hf=Vu.qf16	0	0	0	1	1	1	1	0	-	-	0	-	-	1	0	0	P	P	1	u	u	u	u	u	0	1	1	d	d	d	d	d
Vd.hf=Vuu.qf32	0	0	0	1	1	1	1	0	-	-	0	-	-	1	0	0	P	P	1	u	u	u	u	u	1	1	0	d	d	d	d	d
Vd.sf=Vu.w	0	0	0	1	1	1	1	0	-	-	0	-	-	1	0	1	P	P	1	u	u	u	u	u	0	1	1	d	d	d	d	d
Vd.hf=Vu.h	0	0	0	1	1	1	1	0	-	-	0	-	-	1	0	1	P	P	1	u	u	u	u	u	1	0	0	d	d	d	d	d

Intrinsics

Convert to IEEE floating point intrinsics

Vd.sf=Vu.qf32	HVX_Vector Q6_Vsf_equals_Vqf32(HVX_Vector Vu)
Vd.hf=Vu.qf16	HVX_Vector Q6_Vhf_equals_Vqf16(HVX_Vector Vu)
Vd.hf=Vuu.qf32	HVX_Vector Q6_Vhf_equals_Wqf32(HVX_VectorPair Vuu)
Vd.sf=Vu.w	HVX_Vector Q6_Vsf_equals_Vw(HVX_Vector Vu)
Vd.hf=Vu.h	HVX_Vector Q6_Vhf_equals_Vh(HVX_Vector Vu)

Merge vector with extended bits

Combines word of Vv with extended bits of Vu into destination register.

This instruction supports the operating system to save/restore the register state.

Merge vector with extended bits instructions

Syntax	Behavior

Vd=vmerge(Vu.x,Vv.w)	`for (i = 0; i < VELEM(32); i++) {`
	`  Vd.uw[i] = Vv.uw[i];`
	`  Vd.ext[i]=Vu.ext[i];`
	`}`

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

Note:

- This instruction uses the HVX shift or the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
Vd=vmerge(Vu.x,Vv.w)	0	x	x	0	w	1	1	1	1	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	d	d	d	d	d

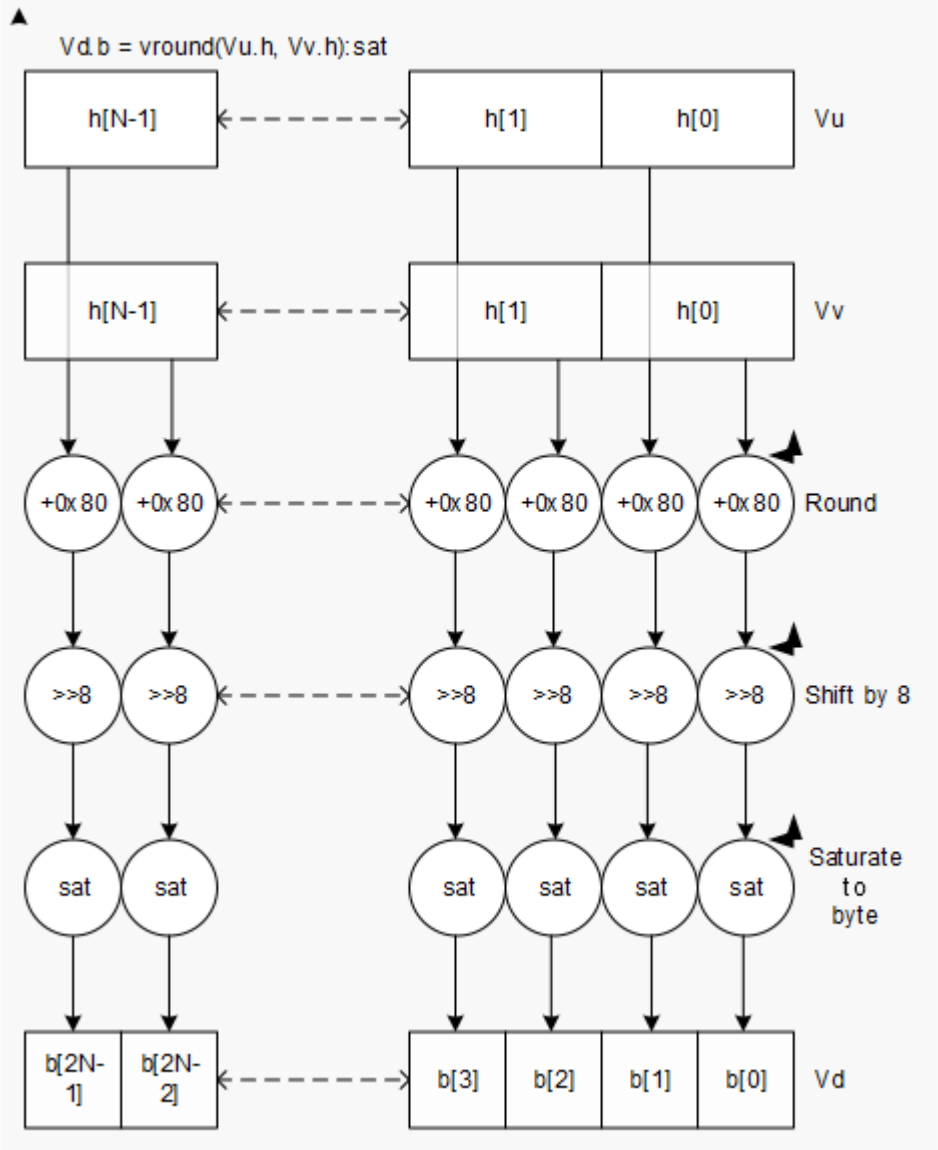
Intrinsics

Merge vector with extended bits intrinsics

Vd=vmerge(Vu.x,Vv.w)	HVX_Vector Q6_V_vmerge_VVw(HVX_Vector Vu, HVX_Vector Vv)

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 8, 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.



Round to next smaller element size instructions

Syntax	Behavior
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.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); }`
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.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.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); }`

Syntax	Behavior

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);`
	`}`

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

Note:

- This instruction uses the HVX shift or the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
Vd.h=vround(Vu.w,Vv.w):sat	0	v	u	0	v	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.uh=vround(Vu.w,Vv.w):sat	0	v	u	0	v	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.uw,Vv.uw):sat	0	v	u	0	v	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.b=vround(Vu.h,Vv.h):sat	0	v	u	0	v	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.ub=vround(Vu.h,Vv.h):sat	0	v	u	0	v	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.uh,Vv.uh):sat	0	v	u	0	v	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

Intrinsics

Round to next smaller element size intrinsics

Vd.h=vround(Vu.w,Vv.w):sat	HVX_Vector Q6_Vh_vround_VwVw_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)
Vd.uh=vround(Vu.uw,Vv.uw):sat	HVX_Vector Q6_Vuh_vround_VuwVuw_sat(HVX_Vector Vu, HVX_Vector Vv)
Vd.b=vround(Vu.h,Vv.h):sat	HVX_Vector Q6_Vb_vround_VhVh_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)

Vector rotate right word

Rotate right each element of Vu.w by the unsigned amount specified by bits 4:0 of corresponding element of Vv.w, place the result in respective elements of Vd.w

Vector rotate right word instructions

Syntax	Behavior
Vd.uw=vrotr(Vu.uw,Vv.uw)	`for (i = 0; i < VELEM(32); i++) { Vd.uw[i] = ((Vu.uw[i] >> (Vv.uw[i] & 0x1f)) | (Vu.uw[i] << (32 - (Vv.uw[i] & 0x1f)))); }`

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

Note:

- This instruction uses the HVX shift or the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
Vd.uw=vrotr(Vu.uw,Vv.uw)	0	u	u	0	v	v	1	1	0	1	0	1	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	1	1	1	d	d	d	d	d

Intrinsics

Vector rotate right word intrinsics

Vd.uw=vrotr(Vu.uw,Vv.uw)	HVX_Vector Q6_Vuw_vrotr_VuwVuw(HVX_Vector Vu, HVX_Vector Vv)

Syntax	Behavior

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

Bit counting instructions

Syntax	Behavior
Vd.uw=vcl0(Vu.uw)	`for (i = 0; i < VELEM(32); i++) { Vd.uw[i]=count_leading_ones(~Vu.uw[i]); }`
Vd.uh=vcl0(Vu.uh)	`for (i = 0; i < VELEM(16); i++) { Vd.uh[i]=count_leading_ones(~Vu.uh[i]); }`
Vd.w=vnormamt(Vu.w)	`for (i = 0; i < VELEM(32); i++) { Vd.w[i]=max(count_leading_ones(~Vu.w[i]),count_leading_ones(Vu.w[i]))-1; }`
Vd.h=vnormamt(Vu.h)	`for (i = 0; i < VELEM(16); i++) { Vd.h[i]=max(count_leading_ones(~Vu.h[i]),count_leading_ones(Vu.h[i]))-1; }`

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

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

Note:

- This instruction uses the HVX shift or the HVX permute/shift resource.

Encodings

Bitmap

Syntax	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
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	0	1	d	d	d	d	d
Vd.uh=vcl0(Vu.uh)	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.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	0	d	d	d	d	d
Vd.h=vnormamt(Vu.h)	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.w=vadd(vclb(Vu.w),Vv.w)	0	0	0	1	1	1	1	0	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.h=vadd(vclb(Vu.h),Vv.h)	0	0	0	1	1	1	1	0	0	0	0	v	v	v	v	v	P	P	1	u	u	u	u	u	0	0	0	d	d	d	d	d

Syntax	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
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	0	d	d	d	d	d

Intrinsics

Bit counting intrinsics

Vd.uw=vcl0(Vu.uw)	HVX_Vector Q6_Vuw_vcl0_Vuw(HVX_Vector Vu)
Vd.uh=vcl0(Vu.uh)	HVX_Vector Q6_Vuh_vcl0_Vuh(HVX_Vector Vu)
Vd.w=vnormamt(Vu.w)	HVX_Vector Q6_Vw_vnormamt_Vw(HVX_Vector Vu)
Vd.h=vnormamt(Vu.h)	HVX_Vector Q6_Vh_vnormamt_Vh(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.h=vadd(vclb(Vu.h),Vv.h)	HVX_Vector Q6_Vh_vadd_vclb_VhVh(HVX_Vector Vu, HVX_Vector Vv)
Vd.h=vpopcount(Vu.h)	HVX_Vector Q6_Vh_vpopcount_Vh(HVX_Vector Vu)

STORE

The HVX store instruction subclass includes memory store instructions.

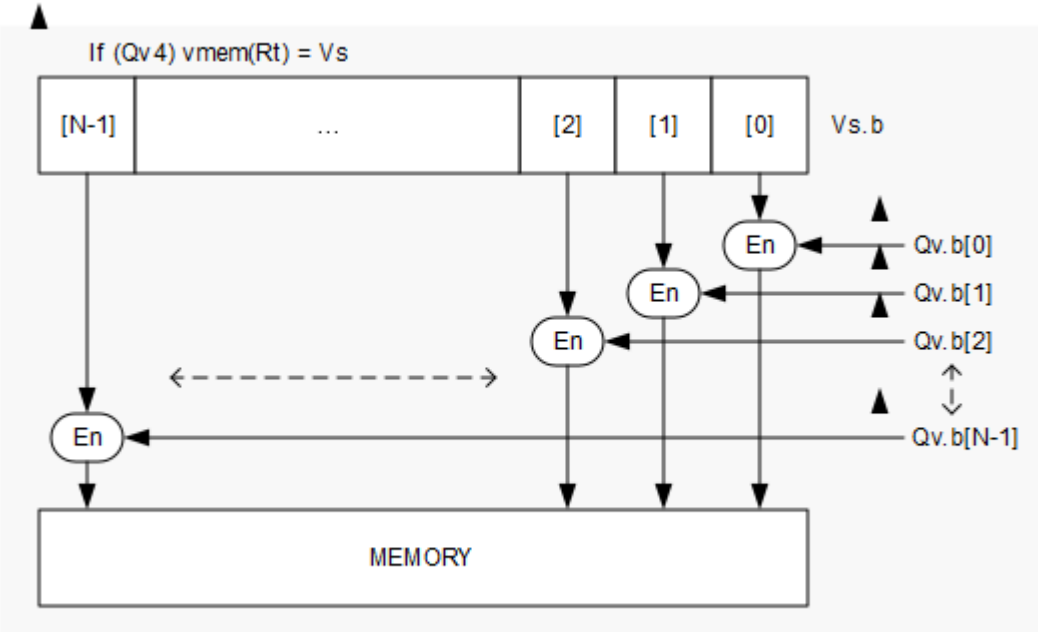
Store - byte-enabled aligned

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 will be stored to memory, but the post-increment of the pointer in Rt will occur.

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



Store - byte-enabled aligned instructions

Syntax	Behavior

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

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

if (Qv4) vmem(Rx++Mu)=Vs	`EA=Rx;`
	`*(EA&~(ALIGNMENT-1)) = Vs;`
	`Rx=Rx+MuV;`

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

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

if (!Qv4) vmem(Rx++Mu)=Vs	`EA=Rx;`
	`*(EA&~(ALIGNMENT-1)) = Vs;`
	`Rx=Rx+MuV;`

if (Qv4) vmem(Rt)=Vs	`Assembler mapped to: "if (Qv4) vmem(Rt+0)=Vs"`
if (!Qv4) vmem(Rt)=Vs	`Assembler mapped to: "if (!Qv4) vmem(Rt+0)=Vs"`

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

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

Syntax	Behavior

if (Qv4) vmem(Rx++Mu):nt=Vs	`EA=Rx;`
	`*(EA&~(ALIGNMENT-1)) = Vs;`
	`Rx=Rx+MuV;`

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

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

if (!Qv4) vmem(Rx++Mu):nt=Vs	`EA=Rx;`
	`*(EA&~(ALIGNMENT-1)) = Vs;`
	`Rx=Rx+MuV;`

if (Qv4) vmem(Rt):nt=Vs	`Assembler mapped to: "if (Qv4) vmem(Rt+0):nt=Vs"`
if (!Qv4) vmem(Rt):nt=Vs	`Assembler mapped to: "if (!Qv4) vmem(Rt+0):nt=Vs"`

Class: HVX (slots 0)

Note:

- This instruction can use any HVX resource.
- An optional “non-temporal” 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.

Encodings

Bitmap

Syntax	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
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	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	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	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(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(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++#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	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	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	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(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(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

Intrinsics

Store - byte-enabled aligned intrinsics

if (Qv4) vmem(Rt+#s4)=Vs	void Q6_vmem_QRIV(HVX_VectorPred Qv, HVX_Vector* A, HVX_Vector Vs)
if (!Qv4) vmem(Rt+#s4)=Vs	void Q6_vmem_QnRIV(HVX_VectorPred Qv, HVX_Vector* A, HVX_Vector Vs)
if (Qv4) vmem(Rt+#s4):nt=Vs	void Q6_vmem_QRIV_nt(HVX_VectorPred Qv, HVX_Vector* A, HVX_Vector Vs)
if (!Qv4) vmem(Rt+#s4):nt=Vs	void Q6_vmem_QnRIV_nt(HVX_VectorPred Qv, HVX_Vector* A, HVX_Vector Vs)

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.

Store - new instructions

Syntax	Behavior

vmem(Rx++#s3)=Os8.new	`EA=Rx;`
	`*(EA&~(ALIGNMENT-1)) = OsN.new;`
	`Rx=Rx+s*VBYTES;`

vmem(Rt+#s4)=Os8.new	`EA=Rt+s*VBYTES;` `*(EA&~(ALIGNMENT-1)) = OsN.new;`

vmem(Rx++Mu)=Os8.new	`EA=Rx;`
	`*(EA&~(ALIGNMENT-1)) = OsN.new;`
	`Rx=Rx+MuV;`

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

Syntax	Behavior
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++Mu)=Os8.new	`if (Pv[0]) { EA=Rx; *(EA&~(ALIGNMENT-1)) = OsN.new; Rx=Rx+MuV; } 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(Rt+#s4)=Os8.new	`if (!Pv[0]) { EA=Rt+s*VBYTES; *(EA&~(ALIGNMENT-1)) = OsN.new; } 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)=Os8.new	`Assembler mapped to: "vmem(Rt+0)=Os8.new"`
vmem(Rx++#s3):nt=Os8.new	`EA=Rx; *(EA&~(ALIGNMENT-1)) = OsN.new; Rx=Rx+s*VBYTES;`
vmem(Rt+#s4):nt=Os8.new	`EA=Rt+s*VBYTES; *(EA&~(ALIGNMENT-1)) = OsN.new;`
vmem(Rx++Mu):nt=Os8.new	`EA=Rx; *(EA&~(ALIGNMENT-1)) = OsN.new; Rx=Rx+MuV;`
if (Pv) vmem(Rx++#s3):nt=Os8.new	`if (Pv[0]) { EA=Rx; *(EA&~(ALIGNMENT-1)) = OsN.new; Rx=Rx+s*VBYTES; } else { 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(Rx++Mu):nt=Os8.new	`if (Pv[0]) { EA=Rx; *(EA&~(ALIGNMENT-1)) = OsN.new; Rx=Rx+MuV; } 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(Rt+#s4):nt=Os8.new	`if (!Pv[0]) { EA=Rt+s*VBYTES; *(EA&~(ALIGNMENT-1)) = OsN.new; } else { NOP; }`

Syntax	Behavior
if (!Pv) vmem(Rx++Mu):nt=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"`

Class: HVX (slots 0)

Note:

- This instruction can use any HVX resource.
- An optional “non-temporal” 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.

Encodings

Bitmap

Syntax	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
vmem(Rx++#s3)=Os8.new	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(Rt+#s4)=Os8.new	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(Rx++Mu)=Os8.new	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
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	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	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	0	0	0	s	s	s

Syntax	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
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(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(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
vmem(Rx++#s3):nt=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(Rt+#s4):nt=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(Rx++Mu):nt=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
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	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	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	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(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(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

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

Store - aligned instructions

Syntax	Behavior

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

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

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

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

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

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

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

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

Class: HVX (slots 0)

Note:

- This instruction can use any HVX resource.

- An optional “non-temporal” 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.

Encodings

Bitmap

Syntax	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
vmem(Rx++#s3)=Vs	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(Rt+#s4)=Vs	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(Rx++Mu)=Vs	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
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	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	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	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(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(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
vmem(Rx++#s3):nt=Vs	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(Rt+#s4):nt=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(Rx++Mu):nt=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
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	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	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	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

Syntax	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
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(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

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 3 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.

Store - unaligned instructions

Syntax	Behavior
if (Pv) vmemu(Rt)=Vs	`Assembler mapped to: "if (Pv) vmemu(Rt+0)=Vs"`
if (!Pv) vmemu(Rt)=Vs	`Assembler mapped to: "if (!Pv) vmemu(Rt+0)=Vs"`

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

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

Syntax	Behavior

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

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

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

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

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

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

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

Note:

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

Syntax	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

Encodings

Bitmap

Syntax	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
vmemu(Rx++#s3)=Vs	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(Rt+#s4)=Vs	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(Rx++Mu)=Vs	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
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	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	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	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(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(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

LEGAL INFORMATION

Your access to and use of this material, along with any documents, software, specifications, reference board files, drawings, diagnostics and other information contained herein (collectively this "Material"), is subject to your (including the corporation or other legal entity you represent, collectively "You" or "Your") acceptance of the terms and conditions ("Terms of Use") set forth below. If You do not agree to these Terms of Use, you may not use this Material and shall immediately destroy any copy thereof.

1) Legal Notice.

This Material is being made available to You solely for Your internal use with those products and service offerings of Qualcomm Technologies, Inc. ("**Qualcomm Technologies**"), its affiliates and/or licensors described in this Material, and shall not be used for any other purposes. If this Material is marked as "**Qualcomm Internal Use Only**", no license is granted to You herein, and You must immediately (a) destroy or return this Material to Qualcomm Technologies, and (b) report Your receipt of this Material to qualcomm.support@qti.qualcomm.com. This Material may not be altered, edited, or modified in any way without Qualcomm Technologies' prior written approval, nor may it be used for any machine learning or artificial intelligence development purpose which results, whether directly or indirectly, in the creation or development of an automated device, program, tool, algorithm, process, methodology, product and/or other output. Unauthorized use or disclosure of this Material or the information contained herein is strictly prohibited, and You agree to indemnify Qualcomm Technologies, its affiliates and licensors for any damages or losses suffered by Qualcomm Technologies, its affiliates and/or licensors for any such unauthorized uses or disclosures of this Material, in whole or part.

Qualcomm Technologies, its affiliates and/or licensors retain all rights and ownership in and to this Material. No license to any trademark, patent, copyright, mask work protection right or any other intellectual property right is either granted or implied by this Material or any information disclosed herein, including, but not limited to, any license to make, use, import or sell any product, service or technology offering embodying any of the information in this Material.

THIS MATERIAL IS BEING PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, WHETHER EXPRESSED, IMPLIED, STATUTORY OR OTHERWISE. TO THE MAXIMUM EXTENT PERMITTED BY LAW, QUALCOMM TECHNOLOGIES, ITS AFFILIATES AND/OR LICENSORS SPECIFICALLY DISCLAIM ALL WARRANTIES OF TITLE, MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, SATISFACTORY QUALITY, COMPLETENESS OR ACCURACY, AND ALL WARRANTIES ARISING OUT OF TRADE USAGE OR OUT OF A COURSE OF DEALING OR COURSE OF PERFORMANCE. MOREOVER, NEITHER QUALCOMM TECHNOLOGIES, NOR ANY OF ITS AFFILIATES AND/OR LICENSORS, SHALL BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY EXPENSES, LOSSES, USE, OR ACTIONS HOWSOEVER INCURRED OR UNDERTAKEN BY YOU IN RELIANCE ON THIS MATERIAL.

Certain product kits, tools and other items referenced in this Material may require You to accept additional terms and conditions before accessing or using those items.

Technical data specified in this Material may be subject to U.S. and other applicable export control laws. Transmission contrary to U.S. and any other applicable law is strictly prohibited.

Nothing in this Material is an offer to sell any of the components or devices referenced herein.

This Material is subject to change without further notification.

In the event of a conflict between these Terms of Use and the *Website Terms of Use* on www.qualcomm.com, the *Qualcomm Privacy Policy* referenced on www.qualcomm.com, or other legal statements or notices found on prior pages of the Material, these Terms of Use will control. In the event of a conflict between these Terms of Use and any other agreement (written or click-through, including, without limitation any non-disclosure agreement) executed by You and Qualcomm Technologies or a Qualcomm Technologies affiliate and/or licensor with respect to Your access to and use of this Material, the other agreement will control.

These Terms of Use shall be governed by and construed and enforced in accordance with the laws of the State of California, excluding the U.N. Convention on International Sale of Goods, without regard to conflict of laws principles. Any dispute, claim or controversy arising out of or relating to these Terms of Use, or the breach or validity hereof, shall be adjudicated only by a court of competent jurisdiction in the county of San Diego, State of California, and You hereby consent to the personal jurisdiction of such courts for that purpose.

2) Trademark and Product Attribution Statements.

Qualcomm is a trademark or registered trademark of Qualcomm Incorporated. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the U.S. and/or elsewhere. The Bluetooth® word mark is a registered trademark owned by Bluetooth SIG, Inc. Other product and brand names referenced in this Material may be trademarks or registered trademarks of their respective owners.

Snapdragon and Qualcomm branded products referenced in this Material are products of Qualcomm Technologies, Inc. and/or its subsidiaries. Qualcomm patented technologies are licensed by Qualcomm Incorporated.