Intel® Xeon Phi™ Coprocessor Vector Microarchitecture

ABSTRACT: This article explains the two key hardware features that dictate the performance of technical computing applications on Intel® Xeon Phi™ coprocessor: the vector processing unit and instruction set as implemented in this architecture. The vector processing unit (VPU) in Xeon Phi™ coprocessor provides data parallelism at a very fine grain, working on 512 bits of 16 single-precision floats or 32-bit integers at a time. The VPU implements a different instruction set architecture (ISA) with 218 new instructions compared to those implemented in the Xeon family of SIMD instruction sets.

==========

Physically, the VPU is an extension to the P54C core and communicates with the core to execute the VPU ISA implemented in Xeon Phi™ coprocessor. The VPU receives its instructions from the core arithmetic logic unit (ALU) and receives the data from the L1 cache by a dedicated 512-bit bus. The VPU has its own dependency logic and communicates with the core to stall when necessary.

The VPU is fully pipelined and can execute most instructions with 4-cycle latency with single-cycle throughput. It can read/write one vector per cycle from/to the vector register file or data cache. As mentioned, each vector can contain 16 single-precision floats or 32-bit integer elements; or eight 64-bit integer or double-precision floating point elements. The VPU can do one load and an operation in the same cycle. The VPU instructions are ternary-operand with two sources and a destination (which can also act as a source for fused multiply-and-add instructions). This configuration provides approximately a 20-percent gain in performance over traditional binary-operand SIMD instructions. The VPU instructions cannot generate exceptions owing to the simplified design, but they can set VXCSR flags to indicate exception conditions. A VPU instruction is considered retired when the core sends it to the VPU. If an error happens, the VPU sets VXCSR flags for overflow, underflow, or other exceptions. Each VPU underneath consists of 8 UALUs each containing 2 SP and 1 DP ALU with independent pipelines. Each UALU has access to a ROM containing a lookup table for transcendental lookup, constants that the UALU needs, etc.

Each VPU has 128 entry 512-bit vector registers divided up among the threads, thus getting 32 entries per thread. These are hard-partitioned. There are eight 16-bit mask registers per thread which are part of the vector register file. The mask registers act as a filter per element for the 16 elements and thus allows one to control which of the 16 32-bit elements are active during a computation. For double precision the mask bits are the bottom 8 bits.

Most of the VPU instructions are issued from the core through the U-pipe. Some of the instructions can be issued from the V-pipe and can be paired to be executed at the same time with instructions in the U-pipe VPU instructions.

The VPU Pipeline

Each VPU instruction passes through all five pipelines to completion:

- **Double Precision (DP) Pipeline**: used to execute float64 arithmetic, conversion from float64 to float32, and DP-compare instructions.
- **Single Precision (SP) Pipeline**: executes most of the instructions including 64-bit integer loads. This includes float32/int32 arithmetic and logical operations, shuffle/broadcast, loads including loadunpack, type conversions from float32/int32 pipelines, EMU transcendental instructions, int64 loads, int64/float64 logical, and other instructions.
- **Mask Pipeline**: executes mask instructions with one-cycle latencies.
- **Store Pipeline:** executes store instructions with one-cycle latencies
- **Scatter/Gather Pipeline:** read/write sparse data in memory into or out of the packed vector registers

Going between pipelines costs additional cycles in some cases. There are no bypasses between the SP and DP pipelines, so intermixing SP and DP code will cause performance penalties, but executing SP instructions consecutively results in good performance, as there are many bypasses built in the pipeline.

The *vector pipeline* is depicted in Figure 1. Once a vector instruction is decoded in stage D2 of the main pipeline, at E stage the VPU detects if there is any dependency stall. At the

VC1/VC2 stage the VPU does the shuffle and load conversion as needed. At V-V4 stages it does the 4-cycle multiply/add operations, followed by WB stage where it writes the vector/mask register contents back to cache as instructed.

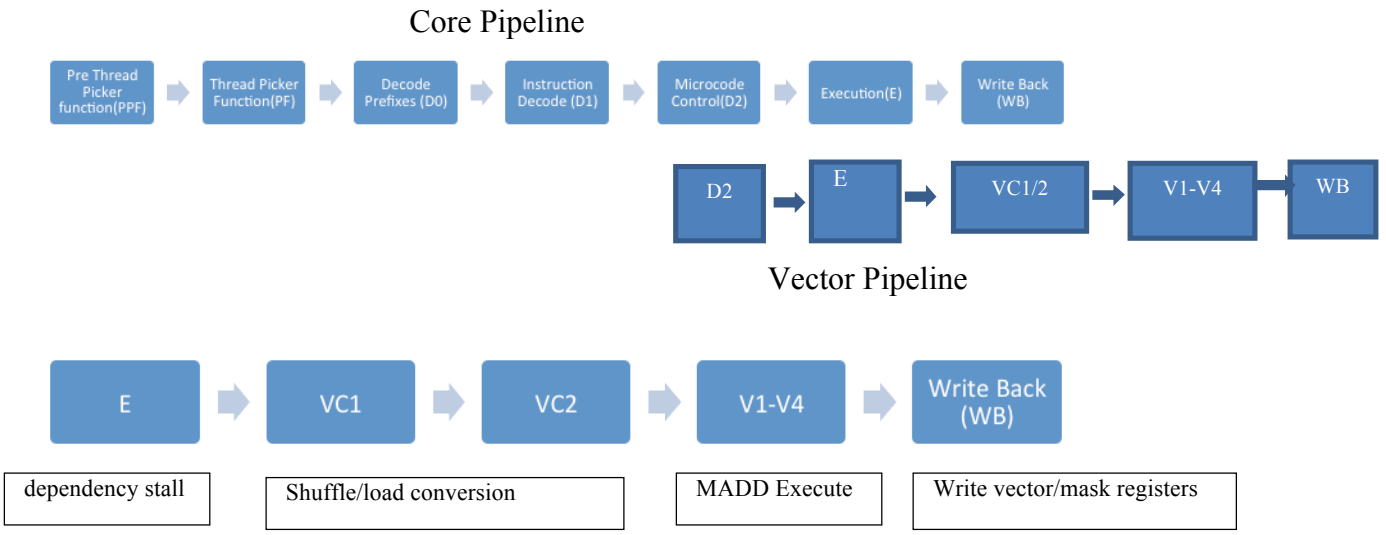

Figure 1a: The vector pipeline stages relative to the core pipeline

When there are four independent SP/DP instructions, the pipeline can throughput one instruction per cycle, with each instruction incurring a latency of four cycles.

If there are data dependencies, say for two consecutive SP instructions, the second instruction will wait until data is produced at stage V4, where it will be passed over to the V1 stage of second instruction using internal bypass.

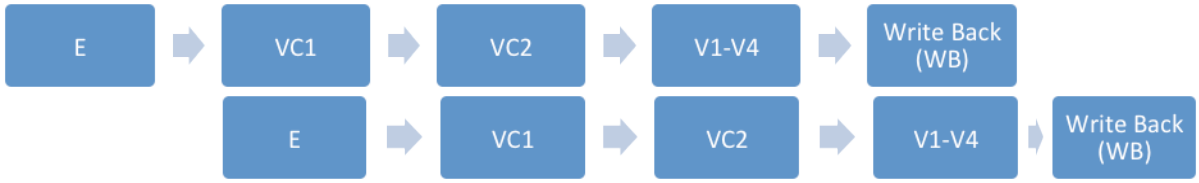

Figure 1b: SP/SP independent instruction pipeline throughput one per cycle

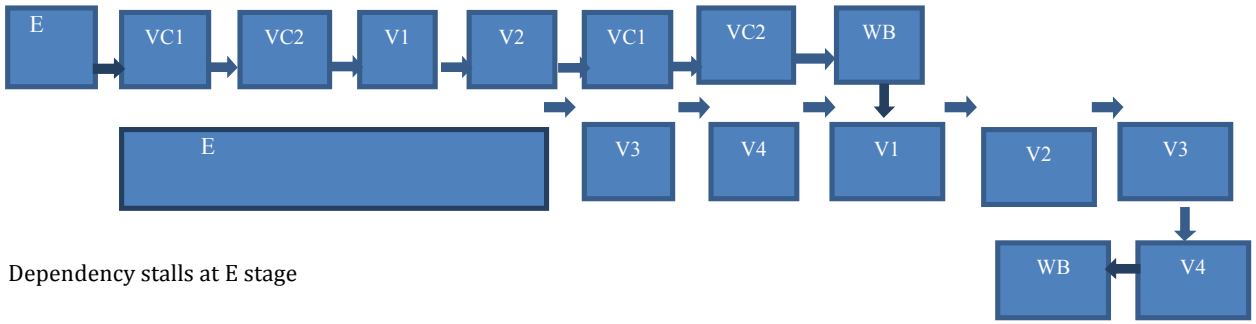

Figure 1c: SP/SP dependent instruction pipeline throughput one per cycle

For an SP instruction followed by a SP-dependent instruction, there is a forward path and, after the results of the first instruction are computed, the result is forwarded to the V1 stage of the second instruction, causing an additional 3-cycle delay, as for for the DP instructions.

For an SP instruction followed by another SP-dependent instruction with register swizzle, the dependent data has to be sent to the VC1 stage that does the swizzling, causing an additional 5-cycle delay.

If the PS instruction is followed by an EMU instruction, since the EMU instruction has to look up the transcendental lookup table, which is done in VC1 stage, then the data has to be

forwarded from V4 to VC1 as for the dependent swizzle case described before, likewise resulting in an additional 5-cycle dependency.

When an instruction in the DP pipeline is followed by a dependent instruction executing in a SP pipeline, the DP instruction has to complete the write before the dependent SP instruction can execute, as there are no bypasses between different pipelines. In this case the second instruction is stalled till the first instruction completes its write-back, incurring a 7-cycle delay.

Pairing Rule

Although all vector instructions can execute on the U-Pipe, some vector instructions can execute on the V-pipe as well. The following table describes these latter instructions. The details of these and other vector instructions are described in the Appendix A. Proper scheduling of these pairable instructions with instructions executing on U-pipe will provide performance boost.

Table 3-1. Instruction Pairing Rules between U- and V-Pipelines.

Type	Instruction Mnemonics
Vector Mask Instructions	JKNZ,JKZ,KAND,KANDN,KANDNR, KCONCATH,KCONCATL,KEXTRACT, KMERGE2L1H,KMERGE2L1L,KMOV, KNOT, KOR, KORTEST, KXNOR, KXOR
Vector Store Instructions	VMOVAPD,VMOVAPS,VMOVDQA32, VMOVDQA64, VMOVGPS, VMOVPGPS
Vector Packstore Instructions	VPACKSTOREHD,VPACKSTOREHPD, VPACKSTOREHPS,VPACKSTOREHQ, VPACKSTORELD,VPACKSTORELPD, VPACKSTORELPS,VPACKSTORELQ, VPACKSTOREHGPS, VPACKSTORELGPS
Vector Prefetch Instructions	VPREFETCH0,VPREFETCH1, VPREFETCH2,VPREFETCHE0, VPREFETCHE1,VPREFETCHE2, VPREFETCHENTA, VPREFETCHNTA
Scalar Instructions	CLEVICT0, CLEVICT1, BITINTERLEAVE11, BITINTERLEAVE21,TZCNT,TZCNTI, LZCNT, LZCNTI, POPCNT, QUADMASK

Vector Registers

The VPU state per thread is maintained in 32 512-bit general vector registers (zmm0-zmm31), 8 16-bit mask registers (K0-K7), and the status register VXCSR as shown in Figure 2.

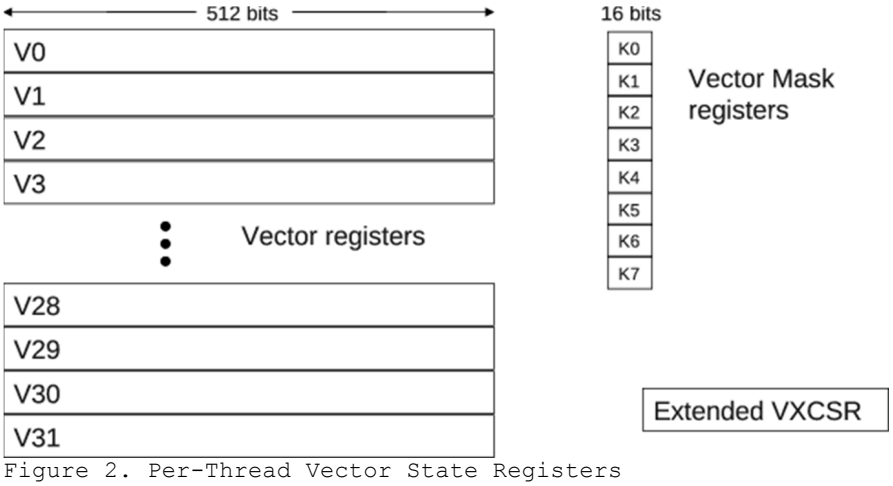

Figure 2. Per-Thread Vector State Registers

Vector registers operate on 16 32-bit elements or 8 64-bit elements at a time. The VXCSR maintains the status of each vector operation, which can be checked later for exceptions during floating-point execution.

The VPU reads and writes the data cache at a cache-line granularity of 512 bits through a dedicated 512-bit bus. Reads from the cache go through the load conversion, swizzling before getting to the ALU. The writes go through store conversion and alignment before going to the write-commit buffer in the data cache.

Vector Mask Registers

Vector mask registers control the update of vector registers inside the calculations. In a non-masked operation, such as a vector multiply, the destination register is completely overwritten by the results of the operation. Using write mask, however, one can make the update of the destination register element conditional on the bit content of a vector mask register, as shown in Figure 3.

This figure shows the effect of the write mask register on vector operations. Here the two vectors v2 and v3 are added and, depending on the mask register bit values, only the V1 register elements which correspond to 1 bit in the k1 registers get updated. The other values corresponding to bit values '0' remain unchanged, unlike implementations where these elements can get cleared. For some operations, such as the vector blend operation (VBLEND*), the mask can be used to select the element from one of the operands to be output.

There is a small set of Boolean operations that can be performed on the mask registers, such as Xor, OR, and AND—but not arithmetic operations such as + or x.

A write mask modifier can be used with all the vector instructions. If it is not specified, a default value of 0xFFFF is implied. There is a special mask register designated k0 which represents the default value of 0xFFFF and is not allowed to be specified as a write mask, since it is implied when no mask register is used. Though this mask register cannot be used for a write mask, it can be used for other mask register purposes, such as holding carry bits from integer arithmetic vector operations, comparison results, etc. One way to remember this restriction is to remember that any mask registers specified inside {} cannot be k0, but they can be used in other locations.

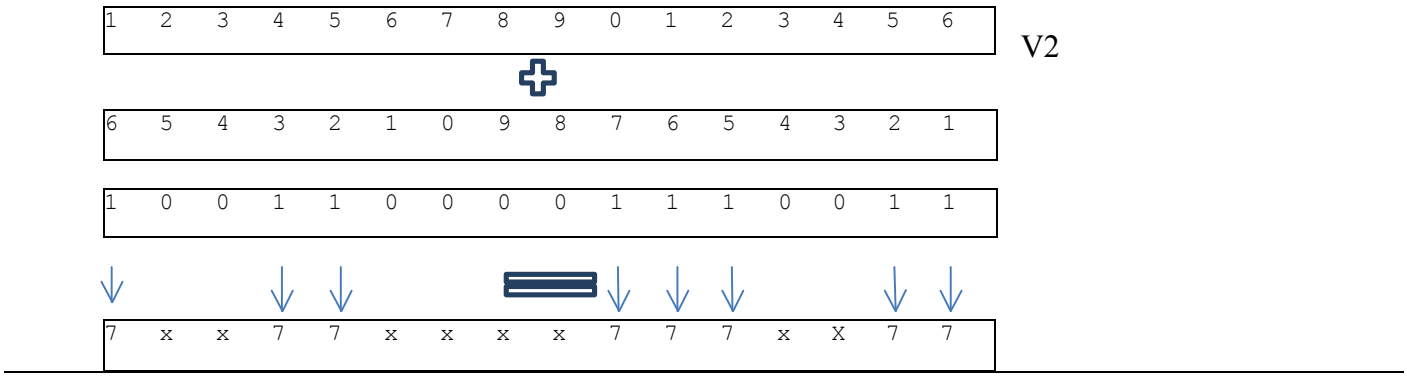

Figure 3: A write mask register updates only the elements of destination register v1 based on mask k1.

Extended Math Unit (EMU)

The VPU implements the single-precision transcendental functions needed by various technical computing applications in various computing domains. These instructions are computed using Quadratic Minimax Polynomial approximation and use a lookup table to provide a fast approximation to the transcendental functions. The EMU is a fully pipelined unit and the lookup table access happens in parallel with squarer computation. The hardware implements the following elementary transcendental functions: reciprocals, reciprocal square roots, base 2 exponential, and base 2 logarithms. There are three derived exponential functions dependent on these elementary functions: division using the reciprocal and multiplier; square root using the reciprocal square root and multiplier; and power using log 2, mult and exp2. Table 2 shows the latency/throughput of vector transcendental instructions in Xeon Phi™ microarchitecture.

Table 2. Latency and Throughput of Transcendental Functions

instructions	Latency (cycles)	Throughput (cycles)
Exp2	8	2
Log2	4	1
Recip	4	1
Rsqrt	4	1
Power	16	4
Sqrt	8	2
Div	8	2

Intel® Xeon Phi™ Vector Instruction Set Architecture (ISA)

The Vector ISA is designed to address technical computing and HPC applications. It supports native 32-bit float and integer and 64-bit float operations. The ISA syntax is composed of ternary instructions with two sources and one destination. There are also FMA (fused multiply and add) instructions, where each of the three registers acts as a source and one of them is also a destination. Although the designers of Xeon Phi™ microarchitecture had every intention to implement the ISA compatible with the Intel® Xeon® processor ISA, the longer vector length, various transcendental instructions, and other issues derailed the effort. However, the effort will continue in future Intel® MIC™ architecture instantiation to achieve ISA conformance with the Intel® Xeon® processor line.

Vector architecture supports a coherent memory model in which the Intel64 instructions and the vector instructions operate on the same address space.

One of the interesting features of vector architecture is the support for scatter/gather instructions to read/write sparse data in memory into or out of the packed vector registers, thus simplifying code generation for the sparse data manipulations so prevalent in technical computing applications.

The ISA supports the IEEE754-2008 floating-point instruction rounding mode requirements. It supports de-norms in DP FP operations, roundTiesToEven, round to 0, and round to + or - infinity. Knights Corner SP FP hardware achieves 0.5 ULP (unit in last place) for SP/DP FP add, sub, and multiply to conform to the IEEE 754-2008 standard.

Data Types

The VPU instructions support the following native data types:

- Packed 32-bit Integers (or dword),
- Packed 32-bit single precision FP values
- Packed 64-bit Integers (or qword)
- Packed 64-bit double precision FP values

The VPU instructions can be categorized into type less 32bit instructions (denoted w/ postfix “d”), type less 64bit instructions (denoted w/ postfix “q”), signed and unsigned int32 instructions (denoted w/ postfix “pi” and “pu”, respectively), signed int64 instructions (denoted w/ postfix “pq”), and fp32 and fp64 instructions (denoted w/ postfix “ps” and “pd”, respectively).

For arithmetic calculations, the VPU represents values internally using 32-bit or 64-bit two’s complement plus a sign bit (duplicate of the MSB) for signed integers, 32-bit or 64-bit plus a sign bit tied to zero for unsigned integers. This is to simplify the integer data path and not to have to implement multiple paths for the integer arithmetic. The VPU represents floating-point values internally using signed-magnitude with an exponent bias of 128 or 1024 to adhere to the IEEE basic single-precision or double precision format.

The VPU supports the up-conversion/down-conversion of the following data types to/from either 32-bit or 64-bit values in order to execute instructions in the SP ALU or the DP ALU.

Table 3 shows the data types that the VPU can convert to native representation for reading and writing from memory to work with 32/64 bit ALUs.

Table 3. VPU-Supported Memory Load Type Conversions

Memory Stored Data Type	Destination Register Data Type			
	float 32	float 64	int32/uint32	int64/uint64
float16	Yes	No	No	No
float32	Yes	No	No	No
sint8	Yes	No	Yes	No
uint8	Yes	No	Yes	No
int16	Yes	No	Yes	No
uint16	Yes	No	Yes	No
int32	Yes	No	Yes	No
uint32	Yes	No	Yes	No
float64	Yes	Yes	No	No
int64/uint64	No	No	No	Yes

Table 4: VPU supported Register Type Conversion

Register source Data Type	Destination Register Data Type			
	float 32	float 64	int32/uint32	int64/uint64
float16	Yes	No	No	No
float32	Yes	Yes	Yes	No
sint8	Yes	No	No	No
uint8	Yes	No	No	No
int16	Yes	No	No	No
uint16	Yes	No	No	No
int32	Yes	Yes	Yes	No
uint32	Yes	Yes	No	No
float64	Yes	No	Yes	No
int64/uint64	No	No	No	Yes

Vector Nomenclature

This section introduces nomenclature helpful for describing vector operations in detail.

Each vector register in the Xeon Phi™ coprocessor is 512 bits wide and can be considered to be divided into four lanes numbered 0-3 and each 128 bits long.

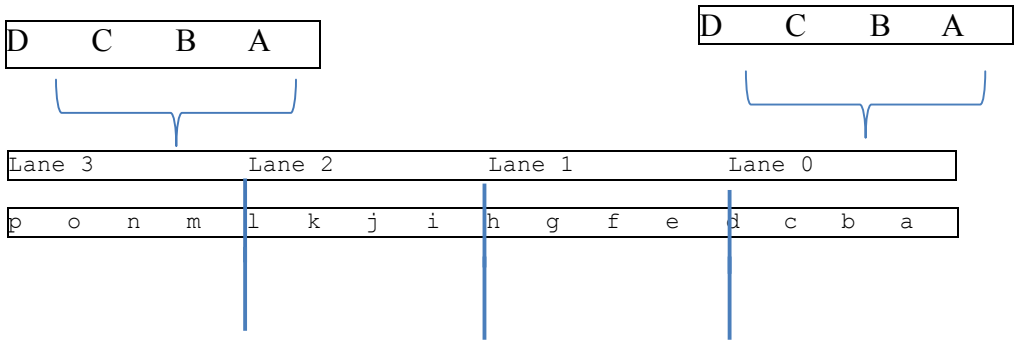

Figure 4: Vector registers nomenclature

There are four 32-bit elements in a 128-bit lane, identified by letters D...A regardless of which lane they belong to. All 16 elements in a vector are denoted by letters p…a , as shown in Figure 4.

The vectors are stored in the memory such that the lowest address is on the right-most side, and the terms are read right to left. For example, when loading a 32-bit full vector from memory address 0xC000, the first element 'a' will correspond to 32-bit memory content located at 0xC000, and the last element 'p' comes from memory at location 0xC03C.

Vector Instruction Syntax

Intel® Xeon Phi™ coprocessor uses three operand forms for its vector ISA. The basic form is as follows:

vop v0{mask}, v1, v2|mem {swizzle},

where *vop* indicates vector operator; *v0,v1,v2* various vector registers defined in the ISA; *mem* is a memory pointer; *{mask}* indicates an optional masking operation; and *{swizzle}* indicates an optional data element permutation/broadcast operation. The *mask* and *swizzle* operations are covered in detail later in the chapter.

Depending on the type of operation, various numbers of vectors from one to three may be referenced as input operands. *v0* in the above syntax is also the output operand of an instruction. The output may be masked with an optional *mask*, and the input operand may be modified by *swizzle* operations.

Each Intel® Xeon Phi™ instruction can operate on from one to three operands to support the ISA. Syntax for various operand sizes are described below:

1. One-input instructions, such as the vector converter instructions:
   v0 <= vop (v1|mem)

where vop is the vector operator; v1 are vector registers; and 'mem' represents a memory reference. The memory reference conforms to standard Intel-64 ISA and can be direct or indirect addressing—with offset, scale, and other modifiers to calculate the address.
An example is *vcvtpu2ps*, which instructs a vector (*'vcvt' part of the instruction mnemonics*) of unsigned integers (*'pu'*) to convert to a (*'2'*) vector of floats (*'ps'*).

2. Two-input instructions, such as vector add operations:
v0 <= vop (v1, v2|mem)
where the operator *vop* works on *v1* and *v2* or *mem* and writes the output to *v0*. The swizzle/broadcast modifiers may be added to *v2/mem*, and the mask operator can be used to select the output of the vector operation to update the v0 register.
An example is *vaddps* , an instruction to add two vectors of floating point data.
Three-input instructions such as fused multiply operations that work on three inputs:
v0 <= vop (v0,v1,v2/mem)
where the operator works on all three vector register data v0, v1, and v2 and writes the result of operation to one of the registers v0.

Xeon Phi™ Vector ISA by Categories

The Xeon Phi™ vector ISA can be broadly categorized into these categories: Mask Operations, Swizzle/Shuffle/Broadcast/Convert Instructions, and Shift Operations.

Mask Operations

The Xeon Phi™ ISA supports optional mask registers that allow one to specify individual elements of a vector register to be worked on. The mask register is specified by curly braces {}. An example of their usage is

vop v1[{k1}], v2, v3|mem

In this instruction, the bits in the 16-bit vector k1 determine which elements of the vector v1 will be written to by this operation. Here k1 is working as a write mask. If the mask bit corresponding to an element is zero, the corresponding element of v1 will remain unchanged; otherwise will be overwritten by the corresponding element of the output of the computation. The square bracket indicates optional arguments.

There are 16 Mask instructions: K*, JKXZ, JKNZ. Mask register k0 has all bits 1. This is a default mask register for all the instructions that do not have their mask specified. Please refer to http://software.intel.com/sites/default/files/forum/278102/327364001en.pdf for details of these operations. The behavior of mask operations is described in the section "Vector Registers" above.

Swizzle/Shuffle/Broadcast/Convert Instructions

Swizzle

Swizzle is an operation to perform data element re-arrangement/permutations of the source operands before executing it. A swizzle modifies the source operand by creating a copy of the input and generating a multiplexed data pattern using the temporary value and feeding it to the ALU as a source for the operation. The temporary value is discarded after the operation is done, thus keeping the original source operand intact.

In the instruction syntax, swizzles are optional arguments to instructions such as the mask operations described in the preceding section.

There are some restrictions on types of swizzles that may be used, depending on the micro-architectural support for the instruction. The instruction behavior is tabulated in Table 3-5.

The swizzle command can be represented as follows:
vectorop v0, v1, v2/mem{swizzle}, where v0,v1 and v2 represents vector registers.

The swizzle operations are element-wise and limited to 8 types. These operations are limited to permuting within 4-element sets of a 32-bit or 64-bit element.

For register source swizzle, the supported swizzle operations are described in Table 5 below. In this table, {dcba} denotes the 32-bit elements that form one 128-bit block in the source (with 'a' least significant and 'd' most significant). {aaaa} means that the least significant

element of each lane of a source register with shuffle modifier is replicated to all four elements of the same lane. When the source is a register, this functionality is same for both integer and floating-point instructions. The first few swap patterns in the table ({cdab}, {badc}, {dacb}) are used to shuffle elements within a lane for arithmetic manipulations such as cross-product, horizontal add, etc. The last four patterns "repeat element" are useful in many operations, such as scalar-vector arithmetic.

Table 5. Supported Swizzle Operations with Vector instructions

Function: 4x32 bits/4x64 bits	Usage {swizzle}
No swizzle	No swizzle modifier (default) or {dcba}
Swap inner pairs	{cdab}
Swap with two away	{badc}
Cross product swizzle	{dacb}
Broadcast 'a' element across 4-element packets	{aaaa}
Broadcast 'b' element across 4-element packets	{bbbb}
Broadcast 'c' element across 4-element packts	{cccc}
Broadcast 'd' element across 4-element packets	{dddd}

Examples: Figure 4 illustrates the swizzle operation for register/register vector operation *vorpi v0,v1,v2{aaaa}*. It shows the replication of the least significant element across all lanes due to the swizzle operation {aaaa}.

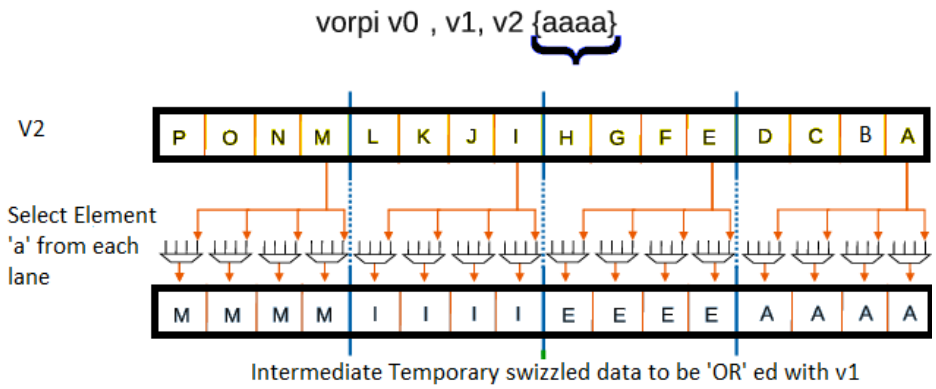

Figure 4. Register – register swizzle operations.

Register/Memory Swizzle

The register memory swizzle operations are available for all implicit loads. These operations perform data replication through broadcast operation or data conversion.

Data Broadcasts

If the input data is a memory pointer instead of a vector register, the swizzle operator works as a broadcast operator. That is, it can read specific elements from memory and replicate or broadcast them to the entire length of the vector register. This can be useful, for example, for vector expansion of a scalar.

The data broadcast operation allows one to perform data replication without having to load all of the 64-byte vector width from memory hierarchy, thus reducing memory traffic. In this operation, a subset of data is loaded and replicated the desired number of times to fill the 64-byte vector width. The three predefined swizzle modes for data broadcasts are: {1 to 16}, {4 to 16}, and {16 to 16}:

- In {1 to 16} broadcast/swizzle pattern, one 32- bit element pointed to by the memory pointer is read from memory and replicated 15 times, which together with the single element read in form memory creates 16 element entries.
- For {4 to 16} broadcast, the first four elements pointed to by the memory pointer are read from memory and replicated three more times to create 16 element entries.
- {16 by 16} broadcasts are implied when no conversions are specified on memory reads and all 16 elements load from the memory into the registers. {16 by 16} is the default pattern in which no replication happens.

Data Conversions

Data conversion allows the use of a swizzle field to convert various data formats in the memory to either 32-bit signed or unsigned integers, or 32-bit floating point data types supported by native Xeon Phi™ coprocessor operations on vector elements. The memory data types supported are 16-bit floats (float16), signed and unsigned 8-bit integers (sint8, uint8), and signed and unsigned 16-bit integers (sint16 and uint16). There is no load conversion support for 64-bit data types.

Intel® Xeon Phi™ coprocessor allows data transformations like swizzle/data conversions etc. on only one operand at most. For instructions which takes more than one operands, the other operands are used unmodified. The ISA does not allow the swizzling and data conversion at the same time. However the data conversion and broadcast can be combined together when doing vector loads.

The vector registers are treated such that they contain either all 32-bit or all 64-bit data—not a mix of 32 and 64 bits. This means that data types other than 32- and 64-bit–such as float16—will have to be mapped to SP or DP floats before computations can be performed on them. Such mapping may result in different results than those obtained by working directly on float16 numbers—violating the commutative or associative rules of the underlying arithmetic. The allowed data conversions are listed in Table 3-3.

Shuffles

The shuffle instructions permute 32-bit blocks of vectors read from memory or vector registers using index bits in the immediate field. No swizzle, broadcast, or conversion is performed by this instruction. Unlike swizzle instruction, which is limited to eight predefined data patterns, the shuffle operation can take arbitrary data patterns. For example, shuffle can generate the pattern dddc, whereas swizzlecannot.

There are two supported shuffle instructions:

1. vpshufd zmm1{k}, zmm2/mem, imm8

This instruction shuffles 32-bit blocks of the vector read from the memory of zmm2 using index bits in imm8. The results are written to zmm1 after applying appropriate masking using mask bits in k.

2. vpermf32x4 zmm1{k}, zmm2/mem, imm8

This instruction differs from the previous one in that it shuffles 128-bit lanes instead of 32-bit blocks within a lane. That is, this instruction is an interlane shuffle, as opposed to an intra-lane shuffle. These two shuffles can be combined consecutively to shuffle inter lane and then within the lanes.

Shift Operation

```
      _mm512_sllv_epi32
      /*    Performs an element-by-element shift of the int32 vector, shifting by the
       *    number of bits given by the corresponding int32 element of last vector.
       *    If the shift count is greater than 31, then for logical shifts the result
       *    is zero, and for arithmetic right-shifts the result is all ones or all
       *    zeroes, depending on the original sign bit.
       *
       *    sllv   logical shift left
       *    srlv   logical shift right
       *    srav   arithmetic shift right
       */

extern __m512i __ICL_INTRINCC _mm512_sllv_epi32(__m512i, __m512i);
extern __m512i __ICL_INTRINCC _mm512_mask_sllv_epi32(__m512i, __mmask16,
                          __m512i,__m512i);

extern __m512i __ICL_INTRINCC _mm512_srav_epi32(__m512i, __m512i);
extern __m512i __ICL_INTRINCC _mm512_mask_srav_epi32(__m512i, __mmask16,
                          __m512i, __m512i);

extern __m512i __ICL_INTRINCC _mm512_srlv_epi32(__m512i, __m512i);
extern __m512i __ICL_INTRINCC _mm512_mask_srlv_epi32(__m512i, __mmask16,
                          __m512i, __m512i);
```

```
/*
 * Shift int32 vector by the full immediate count.
 *
 *    Performs an element-by-element shift of the int32 vector, shifting
 *    by the number of bits given by the count.  If the count is greater than
 *    31, then for logical shifts the result is zero, and for arithmetic
 *    right-shifts the result is all ones or all zeroes, depending on the
 *    original sign bit.
 *
 *    slli   logical shift left
 *    srli   logical shift right
 *    srai   arithmetic shift right
```

Sample Code for Swizzle and Shuffle Instructions

Although we shall be discussing how to program for a Xeon Phi coprocessor, here I am providing some code example that you can scan through now for understanding the concepts of swizzle and shuffle. You can try the code segment out when you are setup with Xeon coprocessor hardware and tools.

The following code fragment shows a simple C++ program using the C++ vector class Is32vec16 provided by the Intel® Xeon Phi™ coprocessor compiler. In order to build and run this code, I went through the steps shown in the compiling and running sample below. The source code is designed to test the shuffle instruction behavior using the C++ vector library and compiler intrinsics (these are functions you can call from C++ routines, which usually map into one assembly instruction).

Compiling and running sample shuftest.cpp on Intel® Xeon Phi™

//compiled the code

- *icpc -mmic shuftest.cpp -o shuftest*

//copied output to Xeon Phi

- *scp shuftest mic0:/tmp*

//Executed the binary

```
ssh mic0 "/tmp/shuftest"
```

Source Code for shuftest.cpp

```
//-------------------------
//-- Program shuftest.cpp
//-- Author: Reza Rahman
////-------------------------

#define MICVEC_DEFINE_OUTPUT_OPERATORS
#include <iostream>
#include <micvec.h>

int main()
{
  _MM_PERM_ENUM p32;

  __declspec(align(64)) Is32vec16  inputData(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);
  __declspec(align(64)) Is32vec16  outputData;

  std::cout << "input = "  << inputData;

  // swizzle input data and print
  //
  std::cout << "\nswizzle data for pattern 'cdab' \n" << inputData.cdab();

  // swizzle input data and print
  std::cout << "\n Intra lane shuffle data for pattern 'aaaa' \n";

  p32 = _MM_PERM_AAAA;

  //shuffle intra lane data
  outputData = Is32vec16(_mm512_shuffle_epi32(__m512i(inputData), p32));
  std::cout << outputData << "\n";

  std::cout << " Inter lane shuffle data for pattern 'aabc' \n";
```

```
  p32 = _MM_PERM_AABC;
  //shuffle inter lane data
  outputData = Is32vec16(_mm512_permute4f128_epi32(__m512i(inputData), p32));
  std::cout << outputData << "\n";
}
```

Below is the output from the run of the sample code. The swizzle form 'cdab' swaps inner pairs of each lane; the intra-lane shuffle with the pattern 'aaaa' on the same inputData replicates element A to each element of each lane; and, finally, the interlane shufflewith the data pattern 'aabc' re-organizes the lanes.

Output from shuftest run on Intel® Xeon Phi™

input = {15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0}

swizzle data for pattern 'cdab'
{14, 15, 12, 13, 10, 11, 8, 9, 6, 7, 4, 5, 2, 3, 0, 1}

Intra lane shuffle data for pattern 'aaaa'
{15, 15, 15, 15, 11, 11, 11, 11, 7, 7, 7, 7, 3, 3, 3, 3}

Inter lane shuffle data for pattern 'aabc'
{7, 6, 5, 4, 11, 10, 9, 8, 15, 14, 13, 12, 15, 14, 13, 12}

There 21 swizzle/shuffle/broadcast and 20 conversion instructions in this category. Please refer to http://software.intel.com/sites/default/files/forum/278102/327364001en.pdf for instruction set details. This document also covers fused multiply-add on 512-bit vectors.

Arithmetic and Logic Operations

There are 55 Arithmetic instructions coded as V*PS for SP FP, V*PD for DP arithmetic, VP*D for int32, and VP*Q for int64. These instructions include 9 MAX/MIN instructions: V*MAX*, V*MIN* plus 4 hardware-implemented EMU transcendental instructions (VEXP233PS, VLOG2PS, VRECIP23PS, VRSQRT23PS). The hardware supports SP/DP denorms, and there is no performance penalty working on the denorms. So it does not assert DAZ (denormals are zero) and FTZ (flush to zero support). For logical operations, the ISA contains 7 compare instructions: V*CMP* which compares vector elements and sets the vector masks. There are also 15 boolean instructions.

Fused Multiply-Add

Intel® Xeon Phi™ coprocessor supports IEEE754-2008-compliant fused multiply-add/subtract (FMA/FMS) instructions which are accurate to 0.5 ulp (units in last place). The computation performed by a ternary FMA operation can be semantically represented by

v1 = v1 vop1 v2 vop2 v3
where vop1 can be set to multiply operation (x) and vop2 set to addition operation (+).

Other forms of these operations are possible by using implicit memory load or by using modifiers to broadcast, swizzles, and other conversions to source v3. In order to simplify the coding effort for the programmers, the ISA contains a series of FMA/FMS (fused multiply-add/subtract) operations that can be numbered with three digits to signify the association of the source vectors with specific operations without remembering the rules which allow for specific sources to be tied to specific modifiers. For example, a basic single-precision vector FMA can have three mnemonics associated with it which are interpreted based on the three digits embedded in mnemonics, as follows:

vfmadd132ps v1,v2,v3 => v1 = v1xv3 + v2
vfmadd213ps v1,v2,v3 => v1 = v2xv1 + v3
vfmadd231ps v1,v2,v3 => v1 = v2xv3 + v1

Memory load modifiers such as broadcast, conversion, are applicable only to v3, but by using the various mnemonics programmers can apply the shuffle operators to the appropriate source vector of interest.

Another FMA mnemonic, vfmadd233ps, allows one to do a scale and bias transformation in one instruction which could be useful in image processing applications.

A vfmadd233ps v1,v2,v3 generates code equivalent to the following:

v1[3..0] = v2[3..0] x v3[1] + v3[0]
v1[7..4] = v2[7..4] x v3[5] + v3[4]
v1[11..8] = v2[11..8] x v3[9] + v3[8]
v1[15..12] = v2[15..12] x v3[13] + v3[12]

The Xeon Phi™ coprocessor also introduces a vector version of carry-propagate instructions. These instructions can be combined to support wider integer arithmetic than the hardware default. In order to support this operation on vector elements, a carry-out flag can be generated for each individual vector element and a carry-in flag needs to be added to each element for the propagate operations. The VPU uses vector mask registers for both the carry-out bit vectors and carry-in bit vectors.

Data Access Operations (Load, Store, Prefetch, and Gather/Scatter)

Data access operations are implemented by instructions to control data load, store and prefetch from memory subsystem in Intel® Xeon Phi™ coprocessor. Please refer to http://software.intel.com/sites/default/files/forum/278102/327364001en.pdf for details of these instructions

Masked load store operations can be used to select the elements to be read/stored to/from a vector register by using mask bits, as described in {mask} operations earlier in this article. We have also discussed broadcast load instructions as part of the swizzle operations. There 22 load/store instructions—V*LOADUNPACK*, V*PACKSTORE* operations and 19 scatter/gather instructions—that implement the semantics for the various scatter/gather operations that are required by the many technical computing applications supported by this ISZ. The mnemonics for these instructions are V*GATHER*, V*SCATTER*. In addition, the ISA supports 8 prefetch instructions V*PREFETCH* to help prefetch data to various cache levels, to reduce data access latency when needed.

All vector instructions that access memory can take an optional cache line eviction hint (EH). The hint can be added to prefetch as well as memory load instructions.

Alignment: All memory-based operations must be on properly aligned addresses. Each source-of-memory operand must have an address that is aligned to the number of bytes accessed by the operand. Otherwise a #GP (General Protection) fault will occur. The alignment requirement is dictated by the number of data elements and the type of the data element. For example, if a vector operation needs to access 16 elements of 4-byte (32 bit) single-precision floats, the referenced data elements must be 16x4=64 [number of elements x size of (float)] byte aligned. The Intel® Xeon Phi™ coprocessor memory alignment rules for vector operations are shown the Table 5 below:

Table 5 - Memory Alignment Rules for Vector Instructions

Memory Storage Form	Number of Load/Store Elements	Needed Memory Alignment (bytes)
4 Bytes (float, int32, uint32)	1 (1 to 16 broadcast)	4
	4 (4 to 16)	16
	16 (16 to 16)	64
2 Bytes (float16, sint16, uint16)	1 (1 to 16 broadcast)	2
	4 (4 to 16)	8
	16 (16 to 16)	32
1 Byte (sint8, uint8)	1 (1 to 16 broadcast)	1
	4 (4 to 16)	4
	16 (16 to 16)	16

Pack/Unpack: The normal vector instructions read 64 bytes of data and overwrite the destination based on the mask register. The unpack instructions keep the serial ordering of the source and write them sparsely to the destination. One can use pack/unpack instructions to handle the case where the memory data has to be compressed or expanded as they are written to memory or read from memory into a register. The mask register dictates how the memory has to be expanded to fill the 64-byte form of the compressed memory data. Examples include the vloadunpackh*/vloadunpackl* instruction pairs. These instructions allow one to relax the memory alignment requirements by requiring alignment to the memory storage form only. As long as the

address to load from is aligned to a boundary for memory storage form, then executing a pair of vloadunpackl* and vloadunpackh* will load all 16 elements with default mask.

Non-temporal data: Cache helps improve application performance by making use of the locality of data being accessed by a program. However, for certain applications, such as streaming data apps, this model is broken and cache is polluted by taking up space for non-reusable data. To allow programmers or compiler developers to support such semantics of non-temporal memory, all memory operands in this ISA have an optional attribute called the eviction hint (EH hint) to indicate that the data is non-temporal. That is, EH indicates that the data may not be reused in time. This is a hint and the coprocessor can ignore it. The hint forces the latest data loaded by this instruction to become "least recently used" (LRU) in the LRU/MRU cache policy enforced by the cache subsystem.

Streaming Stores: In general, in order to write to a cache line, the Xeon Phi™ coprocessor needs to read in a cache line before writing to it. This is known as read for ownership (RFO). One problem with this implementation is that the written data is not reused; we unnecessarily take up the BW for reading non-temporal data. The Intel® Xeon Phi™ coprocessor supports instructions that do not read in data if the data is a streaming store. These instructions, VMOVNRAP*, VMOVNRNGOAP* allow one to indicate that the data needs to be written without reading the data first. In the Xeon Phi ISA the VMOVNRAPS/VMOVNRPD instructions are able to optimize the memory BW in case of a cache miss by not going through the unnecessary read step.

The VMOVNRNGOAP* instructions are useful when the programmer tolerates weak write-ordering of the application data—that is, the stores performed by these instructions are not globally ordered. This means that the subsequent write by the same thread can be observed before the VMOVNRNGOAP instructions are executed. A memory-fencing operation should be used in conjunction with this operation if multiple threads are reading and writing to the same location.

Scatter/Gather: The Intel® Xeon Phi™ ISA implements scatter and gather instructions to enable vectorization of algorithms working with a sparse data layout. Vector scatters are store operations in which the data elements of a vector do not all reside on consecutive locations. Rather, some may reside sparsely on the memory virtual address space. One can still use write mask to select the data elements to be written to, and every one of these elements that are not write masked must obey the memory alignment rule given in Table 3-6. Gather instructions have a syntax of vgatherd* to gather single-precision, double-precision, int32, or int64 elements in a vector register using signed dword indices for source elements. These instructions can gather up to 16 32-bit elements in up to 16 different cache lines. The number of elements gathered will depend on the number of bits set in the mask register provided as source to the instruction.

Prefetch Instructions: The Intel® Xeon Phi™ coprocessor implements a hardware prefetcher, as discussed in chapter 1. In addition, the ISA supports a software prefetch to L1 and L2 data caches. The vprefetch* instructions are implemented for performing these operations. The Intel® Xeon Phi™ coprocessor also implements gather prefetch instructions vgatherpf*. These instructions are critical where the hardware prefetch is not able to bring in necessary data to the cache lines, causing cache-line misses and hence increased instruction retirement stalls. Prefetch instruction can be used to fetch data to L1 or L2 cache lines. Since this hardware implements an inclusive cache mechanism, L1 data prefetched is also present in L2 cache—but not vice versa.

For prefetch instructions, if the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetch instructions can specify invalid addresses without causing a GP fault because of their speculative nature.

Gather/scatter prefetches can be used to reduce the data-access latency of sparse vector elements. The gather/scatter prefetches set the access bits in the related TLB page entry. Scatter prefetches do not set dirty bits.

==========

This article is based on material found in the book *Intel® Xeon Phi™ Coprocesser Micro Architecture and Tools.* Visit the Intel Press web site to learn more about this book: http://noggin.intel.com/intelpress/categories/books/intel%C2%AE-xeon-phi%E2%84%A2-coprocesser-micro-architecture-and-tools

Also see our Recommended Reading List for related topics: www.intel.com/technology/rr

About the Author

Reza Rahman is a Sr. Staff Engineer at Intel Software and Services Group. Reza lead the worldwide technical enabling team for Intel Xeon Phi™ product through Intel software engineering team. He played a key role during the inception of Intel MIC product line by presenting value of such architecture for technical

computing and leading a team of software engineers to work with 100s of customers outside Intel Corporation to optimize code on Intel® Xeon Phi™. He worked internally with hardware architects and Intel compiler and tools team to optimize and add features to improve performance of Intel MIC software and hardware components to meet the need of technical computing customers. He has been with Intel for 19 years. During his first seven years he was at Intel Labs working on developing audio/video technologies and device drivers. Rest of the time at Intel he has been involved in optimizing technical computing applications as part of software enabling team. He is also believes in standardization process allowing software and hardware solutions to interoperate and has been involved in various industry standardization group like World Wide Web consortium (W3C).

Reza holds a Masters in computer Science from Texas A&M university and Bachelors in Electrical Engineering from Bangladesh University of engineering and Technology.

==========