CHAPTER 3

Xeon Phi Vector Architecture and Instruction Set

Two key hardware features that dictate the performance of technical computing applications on Intel Xeon Phi are the vector processing unit and the instruction set implemented in this architecture. The vector processing unit (VPU) in Xeon Phi 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 novel instruction set architecture (ISA), with 218 new instructions compared with those implemented in the Xeon family of SIMD instruction sets.

Xeon Phi Vector Microarchitecture

Physically, the VPU is an extension to the P54C core and communicates with the core to execute the VPU ISA implemented in Xeon Phi. 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 four-cycle latency and single-cycle throughput. It can read/write one vector per cycle from/to the vector register file or data cache. 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 one operation in the same cycle. The VPU instructions are ternary operands 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. Owing to the simplified design, the VPU instructions cannot generate exceptions, but they can set MXCSR 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 MXCSR flags for overflow, underflow, or other exceptions. Each VPU underneath consists of eight master ALUs, each containing two single-precision (SP) and one double-precision (DP) ALU with independent pipelines, thus allowing sixteen SP and eight DP vector operations. Each master ALU has access to a *read-only memory* (ROM) containing a lookup table for transcendental lookup, constants that the master ALU needs, and so forth.

Each VPU has 128 entry 512-bit vector registers divided among the threads, thus providing 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 allow you to control which of the 16 32-bit elements are active during a computation. For double precision the mask bits are the bottom eight 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 one or more of the following 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, extended math unit (EMU) transcendental instructions, int64 loads, int64/float64 logical, and other instructions.
- *Mask pipeline*: Executes mask instructions with one-cycle latencies.
- *Store pipeline*: Executes the vector store operations.
- *Scatter/gather pipeline*: Executes the vector register read/writes from sparse memory locations.

It should be noted that going between pipelines costs additional cycles in some cases. For example, since there are no bypasses between the SP and DP pipelines, 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 execution pipeline* (or *vector pipeline*) is shown in Figure 3-1a. Once a vector instruction is decoded in stage D2 of the main pipeline, it enters the VPU execution 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 the V1-V4 stages it does the four-cycle multiply/add operations, followed by the WB stage, where it writes the vector/mask register contents back to the cache as instructed.

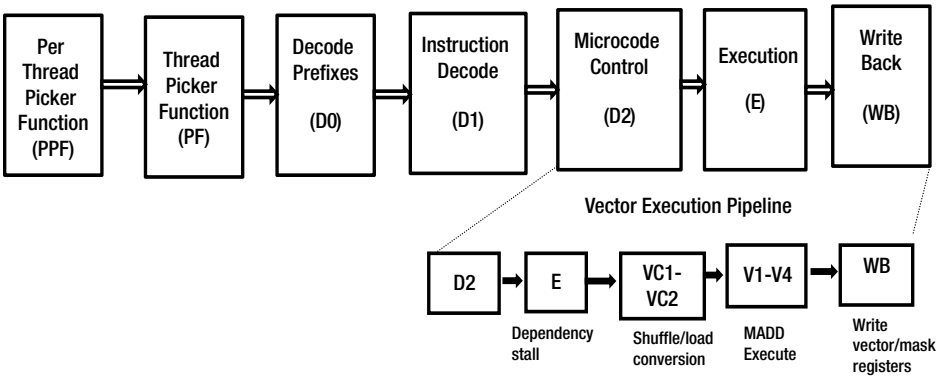

Figure 3-1a. *The vector pipeline stages relative to the core pipeline*

VPU Instruction Stalls

When there are two independent SP/SP instructions, as shown in Figure 3-1b, the pipeline can throughput one instruction per cycle, with each instruction execution incurring a latency of four or more cycles. The four-cycle minimum constitutes the best-case scenario, which involves a vector operation on registers without the need for any shuffle or writing back to memory, spent in the MADD computation unit shown as V1–V4 in Figure 3-1a.

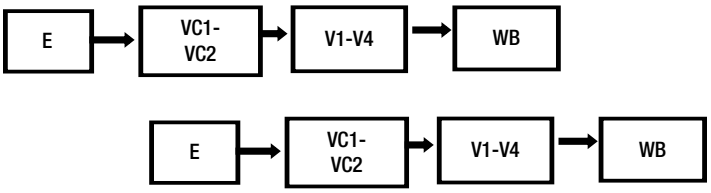

Figure 3-1b. *SP/SP-independent instruction pipeline throughput one per cycle*

If there are data dependencies—say for two consecutive SP instructions, as shown in Figure 3-1c—the second instruction will wait until data are produced at stage V4, where they will be passed over to the V1 stage of the second instruction using an internal bypass.

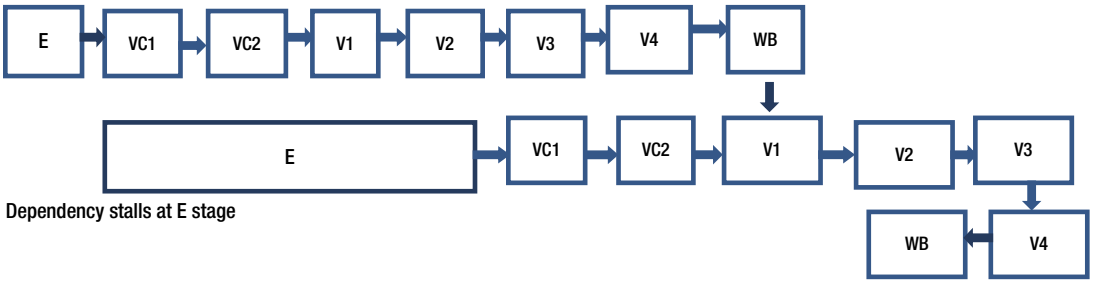

Figure 3-1c. *SP/SP-dependent instruction pipeline throughput one per cycle*

For an SP instruction followed by a dependent SP 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 three-cycle delay, as for the DP instructions.

For a SP instruction followed by another dependent SP instruction with register swizzle, the dependent data have to be sent to the VC1 stage, which does the swizzling, causing an additional five-cycle delay.

If the SP instruction is followed by an EMU instruction, because the EMU instruction has to look up the transcendental lookup table, which is done in VC1 stage, then the data have to be forwarded from V4 to VC1, as for the dependent swizzle case described, likewise resulting in an additional five-cycle dependency.

When an instruction in the DP pipeline is followed by a dependent instruction executing in an SP pipeline, the DP instruction has to complete the write before the dependent SP instruction can execute, because there are no bypasses between different pipelines. In this case the second instruction is stalled until the first instruction completes its write back, incurring a seven-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. Table 3-1 describes the latter instructions. The details of these and other vector instructions are described in 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`), eight 16-bit mask registers (K0-K7), and the status register (MXCSR), as shown in Figure 3-2.

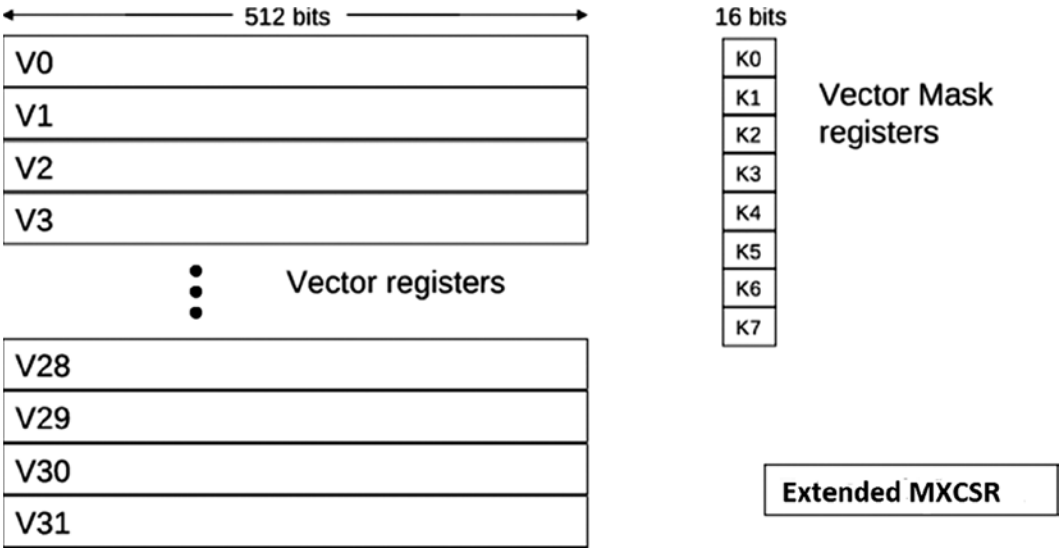

Figure 3-2. *Per-thread vector state registers*

Vector registers operate on 16 32-bit elements or eight 64-bit elements at a time. The MXCSR 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. 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 nonmasked 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 3-3.

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

Figure 3-3 shows the effect of the write mask register on SP vector operations. Here the two vectors `v2` and `v3` are added and, depending on the mask register bit values, only the `V1` register element, which corresponds to 1 bit in the `k1` register, gets 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 ×.

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, because it is implied when no mask register is used. Although 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, and so forth. One way to remember this restriction is to recall that any mask registers specified inside a braces {} mask specifier cannot be `k0`, but they can be used in other locations.

Extended Math Unit

The VPU implements the SP 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 can execute hardware transcendental instructions within one or two cycles. 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 3-2 shows the latency and throughput of vector transcendental instructions in Xeon Phi.

Table 3-2. *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

Xeon Phi Vector Instruction Set Architecture

The Vector ISA is designed to address technical computing and high-performance computing (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 architecture had every intention to implement an ISA compatible with the Intel Xeon processor ISA, the longer vector length, various transcendental instructions, and other issues prevented implementation at this time.

Vector architecture supports a coherent memory model in which the Intel-64 instructions and the vector instructions operate on the same address space.

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

The ISA supports the proposed standard IEEE 754-2008 floating-point instruction rounding mode requirements. It supports denorms in DP floating point operations, round TiesToEven, round to 0, and round to + or - infinity. Xeon Phi floating point hardware achieves 0.5 ULP (unit in last place) for SP/DP floating point FP add, subtract, 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 typeless 32-bit instructions (denoted with the postfix “d”), type less 64-bit instructions (denoted with the postfix “q”), signed and unsigned int32 instructions (denoted with the postfix “pi” and “pu,” respectively), signed int64 instructions (denoted with the postfix “pq”), and fp32 and fp64 instructions (denoted with the 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 most significant bit (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 to avoid implementing 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 SP or DP format.

The VPU supports the up-conversion/down-conversion of the data types listed in Table 3-3 to/from either 32-bit or 64-bit values to execute instructions in the SP ALU or the DP ALU. These are the data types that the VPU can convert to native representation for reading and writing from memory to work with 32- or 64-bit ALUs.

Table 3-3. *VPU-Supported Memory Load Type Conversions*

Memory Stored Data Type	Destination Register Data Type			
	float32	**float64**	**int32/uint32**	**int64/uint64**
float16	Yes	No	No	No
float32	Yes	Yes	Yes	No
sint8	Yes	No	Yes	No
uint8	Yes	No	Yes	No
int16	Yes	No	Yes	No
uint16	Yes	No	Yes	No
int32	Yes	Yes	Yes	No
uint32	Yes	Yes	Yes	No
float64	Yes	Yes	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 Xeon Phi is 512 bits wide and can be considered as being divided into four lanes numbered 0–3 with each being 128 bits long, as shown in Figure 3-4.

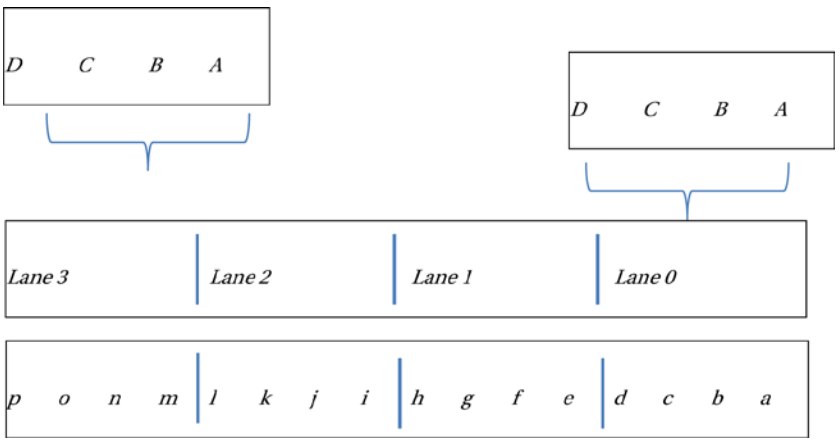

Figure 3-4. *Vector registers nomenclature*

There are four 32-bit elements in a 128-bit lane, identified by letters *D* through *A* regardless of which lane they belong to. All 16 elements in a vector are denoted by letters *p* through *a*, as shown in Figure 3-4.

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 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 or broadcast operation. The `mask` and `swizzle` operations are covered in detail later in this 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` operates on input `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.

3. Three-input instructions, such as fused multiply operations that work on three inputs:

   ```
   v0 <= vop (v0,v1,v2/mem),
   ```

   Where the operator operates on all three input 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 grouped into the following categories.:

Mask Operations

The Xeon Phi ISA supports optional mask registers that allow you 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. For 64-bit data types, the last eight bits of the mask register are used as mask bits. 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 it 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. The mask register `k0` has all bits 1. This is a default mask register for all the instructions that do not have their mask specified. The behavior of mask operations was described in the section of this chapter, "Vector Registers."

Swizzle, Shuffle, Broadcast, and Convert Instructions

These instructions allow you to permute and replicate input data before being operated on by the instructions. The following subsections will look at these broad categories of instructions in detail.

Swizzle

Swizzle is an operation to perform data element rearrangement or permutations of the source operands before operating on the data element. 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 microarchitectural support for the instruction. The instruction behavior is tabulated in Table 3-4.

Table 3-4. *Supported Swizzle Operations with Vector Instructions*

Function: 4 × 32 bits/4 × 64 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 packets	`{cccc}`
Broadcast 'd' element across 4-element packets	`{dddd}`

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 eight types. These operations are limited to permuting within four-element sets of a 32-bit or 64-bit element.

For register source swizzle, the supported swizzle operations are described in Table 3-4, where `{dcba}` denotes the 32-bit elements that form one 128-bit block in the source (with *a* being least significant and *d* being 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 the 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, and so forth. The last four patterns' "repeat element" are useful in many operations, such as scalar-vector arithmetic.

Figure 3-5 illustrates the swizzle operation for the 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}`. Intel compiler intrinsics define `__MM_SWIZZLE_ENUM` to express these permutations.

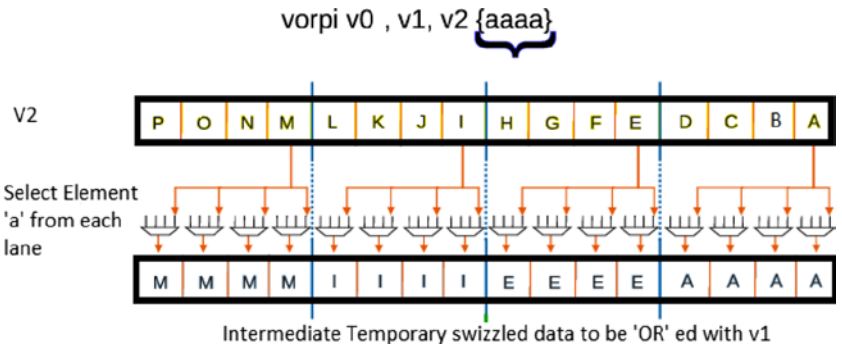

Figure 3-5. *Register/register swizzle operations*

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 you 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 from 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 operations on vector elements. The memory data types supported are 16-bit floats (float16), signed and unsigned eight-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 allows data transformations such as swizzle or data conversions on only one operand at most. For instructions that take more than one operand, 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 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 can generate different results than those obtained by working directly on float16 numbers, violating the commutative or associative rules of the underlying arithmetic. 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, as listed in Table 3-4, the shuffle operation can take arbitrary data patterns. For example, shuffle can generate the pattern `dddc`, whereas swizzle cannot.

There are two supported shuffle instructions:

```
vpshufd zmm1{k}, zmm2/mem, imm8
```

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

```
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, it is an interlane shuffle as opposed to an intralane shuffle. These two shuffles can be combined consecutively to shuffle interlane and then within the lanes. One can use Intel compiler intrinsics enumeration `__MM_SWIZZLE_ENUM` to express these permutations.

Shift Operation

Intel Xeon Phi supports various shift operations on 32-bit integer vectors, as follow.

Logical Shifts

```
vpslld zmm1{k1}, Si32(zmm2/mem), imm8
```

This immediate shift version of the instruction uses the `imm8` value to perform the shift. This instruction performs an element-by-element logical shift of the result of the swizzle, broadcast, or conversion of the input data `zmm2/mem` (indicated by the `Si32` operation in the instruction mnemonic) by shift count given by the immediate value `imm8` and stores the result in `zmm1` using write mask `{k1}`. If the shift count is more than 31, the result is set to all zeros. The write mask dictates which of the elements of the output registers will be written to. The elements in the destination register `zmm1`, for which the corresponding bits in `{k1}` are clear, retain their original values.

The conjugate operation, the right shift immediate operation, is indicated by `vpsrld`, where the `r` replacing `l` in `vpslld` indicates the right shift operation. The logical shift right shifts a 0-bit in the MSB for each shift count. Similar to the left shift, if the number of shifts is greater than 31, all bits are set to zero.

```
vpsllvd zmm1{k1}, zmm2, Si32(zmm3/mem)
```

This version of the instruction uses a vector `Si32 (zmm3/meme)` to indicate the desired shift amount. This instruction performs an element-by-element logical shift of the 32-bit integer vector `zmm2` by the `int32` data computed by the swizzle, broadcast, or conversion of the `zmm3/mem` and stores the result in `zmm1` using write mask `{k1}`. If the shift count is more than 31, the result is set to all zeros. The write mask dictates which of the elements of the output registers will be written to. The elements in the destination register `zmm1`, for which the corresponding bits in `{k1}` are clear, retain their original values.

In the conjugate operation, the right shift vector operation is indicated by `vpsrlvd`, where the `r` replacing `l` in `vpsllvd` indicates the right shift operations. The logical shift right shifts a 0 bit in the MSB for each shift count. Similar to left shift, if the number of shifts is greater than 31, all bits are set to zero.

Arithmetic Shifts

Arithmetic shift Int32 vector right immediate:

```
vpsrad zmm1{k1}, Si32(zmm2/mem), imm8
```

This immediate arithmetic shift version of the instruction uses the `imm8` value to perform the shift. This instruction performs an element-by-element arithmetic right shift of the result of the swizzle, broadcast, or conversion of the input data `zmm2/mem` (indicated by the `Si32` operation in the instruction mnemonic) by the shift count given by the immediate value `imm8` and stores the result in `zmm1` using write mask `{k1}`. The arithmetic shift keeps the sign bit unchanged after each shift count and shifts the results into MSB bits. If the shift count is more than 31, the result is set to the original sign bit for all destination elements. The write mask dictates which of the elements of the output registers will be written to. The elements in the destination register `zmm1`, for which the corresponding bits in `{k1}` are clear, retain their original values.

Arithmetic shift Int32 vector right:

```
vpsravd zmm1{k1}, zmm2, Si32(zmm3/mem)
```

This version of the instruction uses a vector `Si32 (zmm3/mem)` to indicate desired arithmetic right shift count. This instruction performs an element-by-element arithmetic right shift of the 32-bit integer vector `zmm2` by the `int32` data computed by the swizzle, broadcast, or conversion of the `zmm3/mem` and stores the results in `zmm1` using write mask `{k1}`.

The arithmetic shift keeps the sign bit unchanged after each shift count and shifts the results into MSB bits. If the shift count is more than 31, the result is set to the original sign bit for all destination elements. The write mask dictates which of the elements of the output registers will be written to. The elements in the destination register `zmm1`, for which the corresponding bits in `{k1}` are clear, retain their original values.

Sample Code for Swizzle and Shuffle Instructions

Although I will be explaining how to program for a Xeon Phi coprocessor, here I am providing some code examples you can scan through to help you understand the concepts of swizzle and shuffle. You can try the code segment out when you are set up with Xeon Phi hardware and tools.

The code fragment in Listing 3-1 shows a simple C++ program using the C++ vector class `Is32vec16` provided with the Intel Xeon Phi compiler. In order to build and run this code, I went through the steps shown in the top of Listing 3-1. The middle section of Listing 3-1 shows the source code 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).

The bottom section of Listing 3-1 shows the output from the run of the sample code. The swizzle form `cdab` swaps inner pairs of each lane; the intralane shuffle with the pattern `aaaa` on the same input data replicates element `A` to each element of each lane; and, finally, the interlane shuffle with the data pattern `aabc` reorganizes the lanes.

Listing 3-1. Simple C++ Program

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";
}
```

Output from `shuftest.cpp` **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}
```

Arithmetic and Logic Operations

There are 55 arithmetic instructions coded as `V*PS` for SP arithmetic, `V*PD` for DP arithmetic, `VP*D` for int32, and `VP*Q` for int64. These instructions include nine MAX/MIN instructions: `V*MAX*`, `V*MIN*`. There are four hardware-implemented (EMU) transcendental instructions (`VEXP223PS`, `VLOG2PS`, `VRCP23PS`, and `VRSQRT23PS`). The hardware supports SP and DP floating point 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 seven compare instructions—`V*CMP*`—which compare vector elements and set the vector masks. There are also 15 Boolean instructions to implement logical operations.

Fused Multiply-Add

Intel Xeon Phi supports IEEE 754-2008–compliant fused multiply-add/subtract (FMA/FMS) instructions, which are accurate to 0.5 ULP. 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 (×) 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 operations that can be numbered with three digits to signify the association of the source vectors with specific operations without remembering the rules that allow for specific sources to be tied to specific modifiers. For example, a basic SP vector FMA can have three mnemonics associated with it that 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 it can apply the shuffle operators to the appropriate source vector of interest.

Another FMA mnemonic, vfmadd233ps, allows you to do a scale-and-bias transformation in one instruction, which could be useful in image-processing applications. This instruction uses four element sets—0–3, 4–7, 8–11, and 12–15—of source vector v2 and uses v3 elements as scale and bias to generate the results in vector v1; 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]
```

Xeon Phi 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)

The data access instructions control data load, store, and prefetch from memory subsystem in the Intel Xeon Phi coprocessor.

Masked load store operations can be used to select the elements to be read or stored to or from a vector register by using mask bits, as described earlier in the “Mask Operations” section. I also discussed broadcast load instructions as part of the swizzle operations in the “Swizzle” section. There 22 load or store instructions—V*LOADUNPACK*, V*PACKSTORE* operations—and 19 scatter or gather instructions implement the semantics for the various scatter or gather operations that are required by the many technical computing applications supported by this ISA. The mnemonics for these instructions are V*GATHER*, V*SCATTER*. In addition, the ISA supports eight consecutive memory prefetch instructions V*PREFETCH* and six scattered memory gather or scatter prefetch instructions to help prefetch data reach various cache levels and 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.

Memory 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 four-byte (32-bit) SP floats, the referenced data

elements must be 16x4=64 [number of elements times the size of (float)] byte aligned. The Intel Xeon Phi memory alignment rules for vector operations are shown in Table 3-5.

Table 3-5. *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. You can use pack and unpack instructions to handle the case where the memory data have 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 you 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 to indicate that the data are 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) rather than “most recently used” (MRU) 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 are not reused; you unnecessarily take up the memory BW for reading nontemporal data. Intel Xeon Phi supports instructions that do not read in data if the data are a streaming store. These instructions, `VMOVNRAP*`, `VMOVNRNGOAP*`, allow you to indicate that the data need to be written without reading the data first. In Xeon Phi the `VMOVNRAPS/VMOVNRAPD` 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. A memory-fencing operation should be used in conjunction with this operation if multiple threads are reading and writing to the same location.[1]

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 reside in consecutive locations. Rather, some may reside sparsely on the memory virtual address space. You 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 rules given in Table 3-5. Gather instructions have a syntax of `vgatherd*` to gather SP, DP, 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 hardware implements a hardware prefetcher..In addition, the ISA supports a software prefetch to L1 and L2 data caches. The `vprefetch*` instructions are implemented for performing these operations. Intel Xeon Phi 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 are 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 or 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.

Summary

This chapter looked at an important component of Xeon Phi architecture: the vector unit. Used properly, the vector unit allows the coprocessor to achieve teraflops double-precision performance. In order to achieve maximum application performance, you must understand how the vector units are organized and restructure their source as necessary to allow the compiler to generate efficient code and achieve high performance on the coprocessor. This chapter also examined vector pipelines and what might cause them to stall and hurt performance.

[1]Memory Fencing Instructions: Since Intel Xeon Phi cores are `inorder` machines, where instructions are executed in the order they appear in the program, there are no memory fence instructions in these cores—unlike Intel Xeon cores' `mfence` instructions. However, memory access ordering will be needed between threads for consistency. So, in order to provide memory-access ordering on Xeon Phi, you need to impose explicit ordering by using instructions, such as `lock` or `xchg` instructions, to create similar semantics.