Technology Guide

# Intel® Advanced Vector Extensions 512 (Intel® AVX-512) - Permuting Data Within and Between AVX Registers


## Author

Daniel Towner


## 1 Introduction

The Intel® Advanced Vector Extensions (Intel® AVX) family of instruction sets on Intel processors provides a rich variety of capabilities for supporting many different single instruction, multiple data (SIMD) instructions and data types. Like many other SIMD instruction sets, Intel AVX instructions are predominantly vertical, or map, instructions, where one or more SIMD values are converted to another SIMD value element-by-element. However, it can be desirable to reorder elements within or across SIMD values as well, and the Intel AVX families of instructions have many clever ways of achieving this. This document discusses the many different ways to perform permutations, describes the trade-offs, and shows how the techniques described can be used to implement versions of a few selected algorithms.

This document is part of the Network Transformation Experience Kits.

# Table of Contents



# Figures

## Tables



## Document Revision History

| Revision | Date | Description |
|---|---|---|
| 001 | October 2022 | Initial release. |

## 1.1 Terminology

Table 1. Terminology

| Abbreviation | Description |
|---|---|
| BLAS | Basic Linear Algebra Subprograms |
| GFNI | Galois Field New Instructions |
| Intel® AVX | Intel® Advanced Vector Extensions (Intel® AVX) |
| Intel® AVX2 | Intel® Advanced Vector Extensions 2 (Intel® AVX2) |
| Intel® AVX-512 | Intel® Advanced Vector Extensions 512 (Intel® AVX-512) |
| ISA | Instruction set architecture. A definition of a processor, its instructions, and its data storage. ISA is typically used to refer to a family of related instructions that implement a particular class of operations. |
| LLVM | LLVM is a set of compiler and toolchain technologies that can be used to develop a front end for any programming language and a back end for any instruction set architecture. |
| LSB | Least significant bit. |
| MSB | Most significant bit |
| SIMD | Single instruction, multiple data. A single instruction operates on registers that can contain more than one data element. As contrasted to Single Instruction, Single Data (SISD), where each instruction operates on registers that store exactly one data value. |
| VBMI | Vector Bit Manipulation Instructions |
| VNNI | Vector Neural Network Instructions |

## 1.2 Reference Documentation

Table 2. Reference Documents

| Reference | Source |
|---|---|
| Intel® AVX-512 - FP16 Instruction Set for Intel® Xeon® Processor Based Products Technology Guide | https://networkbuilders.intel.com/solutionslibrary/intel-avx-512-fp16-instruction-set-for-intel-xeon-processor-based-products-technology-guide |
| Intel® AVX-512 - Instruction Set for Packet Processing Technology Guide | https://networkbuilders.intel.com/solutionslibrary/intel-avx-512-instruction-set-for-packet-processing-technology-guide |
| Intel® AVX-512 - Packet Processing with Intel® AVX-512 Instruction Set Solution Brief | https://networkbuilders.intel.com/solutionslibrary/intel-avx-512-packet-processing-with-intel-avx-512-instruction-set-solution-brief |
| Intel® AVX-512 - Writing Packet Processing Software with Intel® AVX-512 Instruction Set Technology Guide | https://networkbuilders.intel.com/solutionslibrary/intel-avx-512-writing-packet-processing-software-with-intel-avx-512-instruction-set-technology-guide |
| Intel® AVX-512 – Accelerate Packet Processing Using Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Training Video | https://networkbuilders.intel.com/accelerate-packet-processing-using-intel-advanced-vector-extensions-512-intel-avx-512 |
| Intel® AVX-512-FP16 Architecture Specification | https://software.intel.com/content/www/us/en/develop/download/intel-avx512-fp16-architecture-specification.html |
| Intel® 64 and IA-32 Architectures Optimization Reference Manual | https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf |
| Intel® 64 and IA-32 Architectures Software Developer Manuals | https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html |
| Intel® Intrinsics Guide, an online reference guide | https://software.intel.com/sites/landingpage/IntrinsicsGuide/ |
| Galois Field New Instructions (GFNI) Technology Guide | https://networkbuilders.intel.com/solutionslibrary/galois-field-new-instructions-gfni-technology-guide |

# 2 Background

Permute instructions can be used in many ways, including:

- Transpositions of matrices in numeric processing (e.g., Basic Linear Algebra Subprograms, or BLAS)
- Reformatting multiple independent data sets to make them more suitable for SIMD processing (also known as array-of-struct and struct-of-array conversions)
- Horizontal reductions (e.g., determine the minimum value element in a SIMD value)
- In-register look-up tables

All of these examples depend in some critical way upon being able to move data within or across SIMD values, and consequently understanding the most efficient ways possible to achieve this can make a considerable impact on the execution performance.

Section 4 has worked examples that show ways to implement some of these functions.

## 2.1 Terminology

Before we discuss the classes of permute instructions, it is necessary to set out how AVX registers are organized to store different data elements since the layout of the register has many implications for the performance and utility of the instructions. Figure 1 shows how the data within a variety of registers may be organized. Throughout this document we adopted the convention that bits are numbered so that the least-significant bit is on the right and the most significant bit is on the left.

At the highest level, SIMD values may be SSE-style 128-bit, AVX/AXV2-style 256-bit, or Intel AVX-512 style 512-bit. All registers can be divided into 128-bit groups known as lanes. Lanes are an important concept since they have a direct impact on execution performance; any permute operation that takes place within a lane will be faster or more efficient than an operation that crosses lanes. Note that an SSE register is effectively an entire lane in its own right. Once again, the convention of putting the least significant bit on the right means that Lane 0 is the rightmost lane, and subsequent lanes are numbered ascendingly from right to left.

Registers may be subdivided further into individual elements. Such elements may be 8, 16, 32, or 64-bits in size. Those elements represent the smallest unit at which an operation may be applied to a SIMD value. For some permutation operations, elements are grouped to form small clusters of data that are worked on together. For example, some permute instructions operate on formats known as `f32x4' or 'i64x2', which represent four 32-bit values or two 64-bit values respectively, although it should be noted that such clusters are essentially a 128-bit lane by a different name.

In an Intel AVX-512 processor, instructions can access registers of 128-bit, 256-bit, or 512-bit. The different sizes of registers are overlaid on top of each other. For example, the 128-bit xmm register is the lowest part of a 256-bit ymm register, and that in turn is the lowest half of a 512-bit zmm register. If a zmm register is used in an SSE instruction, then the instruction only operates on the lowest 128-bits.

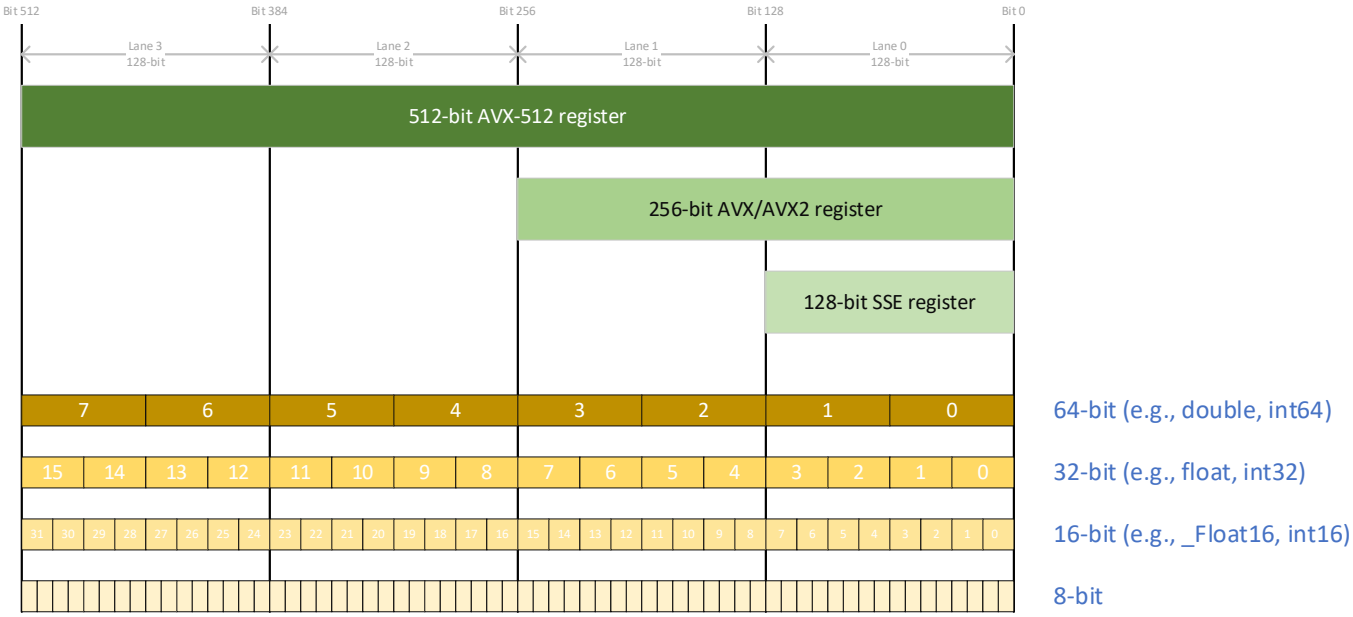


Figure 1. Layout of Various Sizes of SIMD Register and How Each Can Be Broken Down into Smaller Subgroups of Elements

Note that the terms shuffle or swizzle are also common names for the operation of permuting data, but we shall prefer the term permute in this document. Shuffle is used in the names of some older pre-Intel AVX-512 instructions, although there are some exceptions where shuffle issued more recently than this too.

## 2.2 Execution Performance

Permutes have a number of orthogonal properties that describe or influence their behavior and performance:

**In-lane or cross-lane** Permutes that operate within 128-bit lanes are faster, but less flexible, than their wider counterparts

**1-source or 2-source** Some permutes can read data from two source registers, thus allowing inter-register permutation.

**Static or dynamic** Some permutes encode their permutation pattern into the instruction itself or use an immediate that is supplied to the instruction. Encoding the pattern into the instruction at compile-time can remove the need for a separate index register but also will be less flexible in its capabilities. For example, immediate values that are encoded into an instruction have a limited number of bits, and hence can only configure a limited set of permutation indexes. In contrast, dynamic indexing is extremely flexible but is more expensive due to the need to load and use an extra index register.

**Granularity** The granularity of the units of data to which the indexes apply can vary, from entire 128-bit lanes and down to individual bytes in some of the more recent Intel AVX-512 instruction set variants. Typically, the large granularity permutes are cheaper to execute than variants that operate at the smaller granularities.

In all the cases above, the choice for each property impacts the performance of the corresponding permute. The cheapest permutes would be those that operate within a lane using a static index and address large elements, while at the expensive end of that extreme would be the cross-lane, multi-source dynamic byte permutes. When optimizing code, it is wise to try to use the cheapest possible permute to do the required job.

While there are many instructions dedicated to moving data across or within SIMD values, it is useful to be able to perform data movement in other ways too in order to improve the performance of the processor. This is because the processor is only capable of executing dedicated permute instructions on port 5 as shown in Table 3[1]. It can be useful to use non-permute instructions in ways that mimic permutes, so in a later section we also describe some of those instructions.

Table 3. Dispatch Port and Execution Stacks of the Ice Lake Client Microarchitecture

| Port 0 | Port 1[1] | Port 2 | Port 3 | Port 4 | Port 5 | Port 6 | Port 7 | Port 8 | Port 9 |
|---|---|---|---|---|---|---|---|---|---|
| INT ALU<br>LEA<br>INT Shift<br>Jump1 | INT ALU<br>LEA<br>INT Mul<br>INT Div | Load | Load | Store Data | INT ALU<br>LEA<br>INT MUL Hi | INT ALU<br>LEA<br>INT Shift<br>Jump2 | Store Address | Store Address | Store Data |
| FMA<br>Vec ALU<br>Vec Shift<br>FP Div | FMA*<br>Vec ALU*<br>Vec Shift*<br>Vec Shuffle* | | | | Vec ALU<br>Vec Shuffle | | | | |

## 2.3 Element Data Type Support

As shown in Figure 1 above, SIMD values may be divided into smaller groups of elements. In the earlier SSE and Intel AVX instruction sets, most instructions operated primarily on 32-bit and 64-bit elements, in both integer and floating-point forms. At the time, these data elements were the most common and consequently Intel AVX permutation support was skewed towards those data types in preference to other less common sizes, for both vertical and horizontal operations. There was some support for the less common 8-bit operations, and very little support for 16-bit operations. Note that the 8-bit operations could be used readily to implement 16-bit permutations where necessary, although with some slight inconvenience (e.g., a 16-bit dynamic permute requires the individual 8-bit permute elements to be generated).

With the evolution of ISAs from SSE through Intel AVX2 and into Intel AVX-512, other data sizes become more common and these instruction sets started to gain additional 8- and 16-bit capabilities for all types of operation. These instructions were still slightly disadvantaged as they often had lower execution performance, but at least they provided ways of handling these operations more efficiently than synthesized code sequences. However, support for some of the more exotic instructions, such as compress and expand, still lacked 16-bit and 8-bit support.

In the 3rd Gen Intel® Xeon® Scalable processor, a major upgrade of many instructions occurred. 8- and 16-bit permutations were finally handled with virtually the same ease as the other data types. For example, the compress and expand instructions that we describe in Section 3.4 gained variants that operated at 8- and 16-bit granularity.

[1] Note that on 3rd Gen Intel® Xeon® Scalable processors (codenamed Ice Lake) and 4th Gen Intel® Xeon® Scalable processors (codenamed Sapphire Rapids), port 1 also can execute some shuffle instructions, but not the full range of general purpose permutes described in this document, so we do not consider that dimension further in this document.

At the time of writing, 3rd Gen and 4th Gen Intel Xeon Scalable processors support essentially every type of permute at every element granularity. There may still be some performance differences (especially for small element types) but there is no lack of instructions available to perform every possible instruction that could be required.

In a few places an instruction is available for only some data types of a particular size and not for others. For example, the recently introduced AVX512-FP16 instruction set, described in Intel® AVX-512 - FP16 Instruction Set for Intel® Xeon® Processor Based Products Technology Guide, provides support for half-precision floating-point operations but does not provide any way to permute such values. However, since the FP16 data format uses 16-bits for storage, we can implement variants of those instructions using the existing epi16 permute support, as illustrated in Figure 2. In that example, the incoming FP16 data is cast to epi16, the compress instruction performed on it, and then the value cast back to FP16 on completion. The cast instructions cost nothing to implement and are only there to tell the compiler the intent.

```
__m512h compress_ph(__mmask32 mask, __m512h value)
{
  const auto asInt16 = _mm512_castph_si512(value);
  const auto comp = _mm512_mask_compress_epi16(mask, asInt16);
  return _mm512_castsi512_ph(comp);
}
```

Figure 2. Function to Implement the 16-bit Compress Operation on FP16 Vector Elements

# 3 AVX Permutation Instructions

In this major section we describe each of the common classes of permute instructions, starting with general purpose instructions that can essentially implement any permute, and working through some of the more unusual specialized instructions that can be used to perform particular types of permute more efficiently or faster.

## 3.1 General Purpose Permutes

The general purpose permutes are the most flexible way to perform almost any type of horizontal movement of data within or across multiple AVX SIMD values. Practically any type of desired permutation of data could be implemented using the instructions described in this section, with the only exception being the compress and expand instructions from Section 3.4. All permutes operate on the general principle shown in Figure 3. A set of elements from some input SIMD value is reordered using an index to form a new SIMD value. Individual elements may be duplicated, moved, or omitted entirely from the output result.

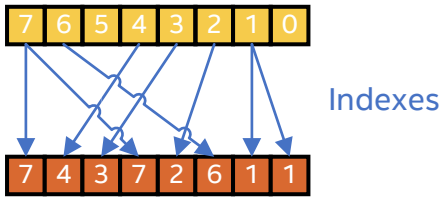


Figure 3. General Operation of a Permute

Later in this document we describe many specialized permute instructions. The general purpose permute instructions are sufficiently powerful that they can often behave in the same way as the specialized permutes, but there are some disadvantages to using a general permute in such a way. One reason is that the more powerful general purpose permutes often require an index register to be used to store the desired permutation. Using a register to store the index increases register pressure (i.e., how many registers are needed to store all in-use values), which can limit some optimizations, and also requires an extra instruction to load the desired index values into the register. Another reason for preferring a specialized instruction is that it may use dedicated hardware to make the operation more efficient. For example, if a single value has to be broadcast from memory into all elements of a register, one way to achieve this is by loading a value from memory into a register in one instruction, and then broadcasting it in the register using a general purpose permute in another instruction. However, the load execution unit in all recent Intel processors has a special broadcast unit built into the memory load unit, which means that a single instruction can perform the work of both the load and the broadcast more cheaply.

The remainder of this section describes the main variants of permute, working from the simplest and cheapest, through to the most comprehensive and expensive.

### 3.1.1 In-Lane General-Purpose Permute

The in-lane permutes can only move data within a single 128-bit lane of data, and historically they all derive from the original instructions found in the 128-bit SSE-family of instructions. The 256-bit and 512-bit variants that were introduced in Intel AVX and Intel AVX-512 are implemented by duplicating the underlying 128-bit lane function across every lane in the wider registers. An illustration of their general behavior for 256-bit registers is shown in Figure 4. On the left, the permute is operating on 32-bit data elements. Because each lane has only four such elements and they can only permute to four possible locations within the output lane, the desired index can be expressed in a single 8-bit immediate. Because there is only one immediate value however, all the lanes must perform the same permutation. In contrast, the right-hand example of that figure shows how a byte-level permute is implemented. In this, every byte can be moved somewhere else, but because the indexing is more complicated, the index can only come from a second index register. Because a separate index register is used it does mean that each lane can use a different permutation index.

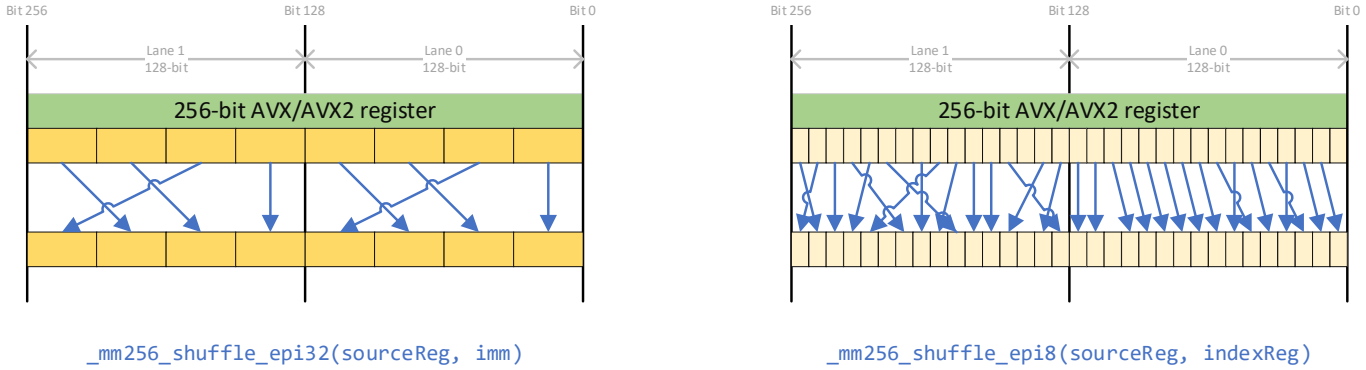


Figure 4. Operation of Two Different Types of In-Lane Permute

Note that the limited size of the immediate value means that it is impossible to encode 8- or 16-bit permutes using an immediate. The only way to permute at this granularity is by using the shuffle_epi8 instruction with a dynamic index. In the case of 16-bit permutes, adjacent 8-bit indexes must be configured to move pairs of bytes, rather than having a single index representing the entire 16-bit index.

One note of warning is that the index register used for `_mm[256]_shuffle_epi8` has a slight difference in behavior compared with many of the other general-purpose indexed permutes described in this document. Normally a permute would use the least number of bits required to form the index (e.g., an in-lane shuffle would require four bits to store any possible index position for a byte within a lane). Any extra bits would be ignored by most of the AVX permute instructions, forming what we shall call a dirty index (see Section 3.1.5 for more details). In the case of the shuffle_epi8 instructions however, the MSB of each index is used to decide whether to insert a zero value into the output position, as illustrated in Figure 5. Note how the hexadecimal index uses the lowest 4-bits as the output index, but if the MSB of the index is set, a zero value (gray in this diagram) will be inserted instead.

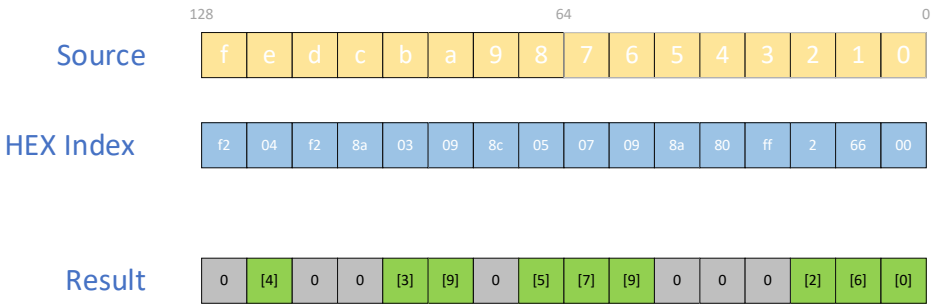


Figure 5. How In-Lane Dynamic Shuffle Instruction Permits a Conditional 0 to be Inserted Instead of an Index Element

Note that the in-lane permutes are named shuffles because they come from the original SSE instruction set that used that terminology, and the same naming convention has been used for their 256-bit and 512-bit in-lane counterparts.

All variants of in-lane permutes are cheap and can operate in 1 cycle, with a throughput of 1 instruction per cycle.

### 3.1.2 Single Source Cross-Lane General-Purpose Permutes

Full register width, cross-lane permutes operate as illustrated in Figure 6. Cross-lane permutes allow any input element to be copied to any output element position. Because of the large set of possible indexes, an immediate cannot be used and the index is supplied in another register.

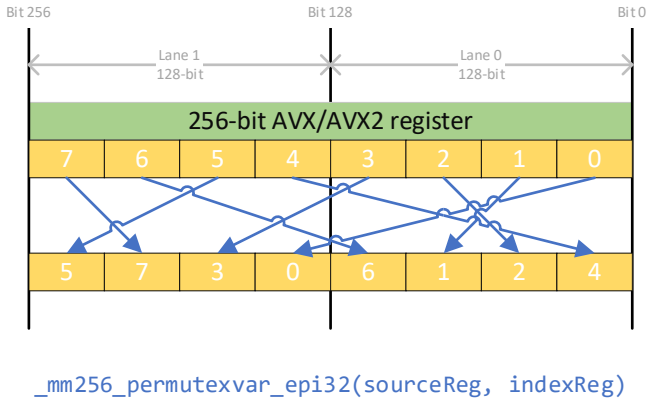


Figure 6. A Full-Register Single-Source Cross-Lane Permute

Cross-lane permutes typically have a latency of 3 cycles and a throughput of 1 instruction per cycle.

The Intel AVX-512 foundation instruction set has support for 32-bit and 64-bit elements cross-lane permutes. 8-bit and 16-bit support for cross-lane permutes was only added in the VBMI ISA available in 3rd Gen Intel Xeon Scalable processors and onwards. Note also that while 8-bit element permutes have the same performance as their 32- and 64-bit counterparts, 16-bit cross-lane permute is more expensive and should be avoided where possible or replaced with an 8-bit variant instead.

### 3.1.3 Dual Source Cross-Lane General-Purpose Permutes

Dual source permutes allow two source registers to be used as though they were indexed as a single register of twice the width. This is illustrated in Figure 7, where the first source register (on the right, following our LSB-on-the-right convention) provides indexes [0..7] and the left-hand register provides indexes [8..15]. The output register can therefore not only choose which element from the register to use in any given output position, but it may even select from which register the element should be taken. This is a powerful instruction for performing any permute where data from several sources needs to be mixed together (e.g., transpose).

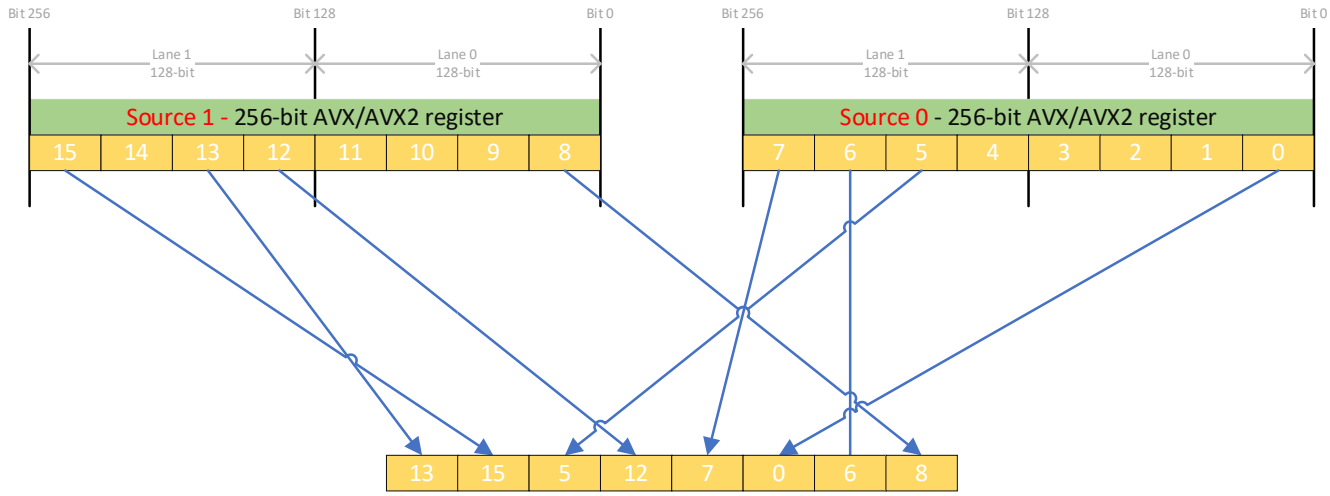


Figure 7. A Dual-Source Cross-Lane Permute

The 32-bit and 64-bit variants available in the Intel AVX-512 foundation ISA have the same performance as their single-source counterparts: 3 cycle latency and a throughput of 1.

8-bit and 16-bit dual source permutes were introduced in the VBMI ISA added in 3rd Gen Intel Xeon Scalable processors. They are both slightly more expensive than their single-source counterparts.

### 3.1.4 Bit-Level Byte Shuffle (VBMI)

The general-purpose instructions discussed so far operate at an element granularity that is no less than the byte, but it can sometimes be useful to be able to permute data at the bit-level. This would be an expensive operation if applied at the full-register size (i.e., moving a set of bits from any one location in an Intel AVX-512 register to anywhere else would require very expensive hardware), and also defining the index would be very tricky. However, by imposing a limitation that data movement is restricted to within a half-lane of 64-bits, it becomes possible to define a special purpose bit-level permutation, and this was introduced to the 3rd Gen Intel Xeon Scalable processors VBMI ISA.

The basic operation of the VBMI multishift byte instruction is shown in Figure 8. The input is a set of 64-bit elements. A source index register is supplied that is broken down into 8-bit groups, and within each 8-bit group is a single index value that represents the bit-level index of the 8-bit block to read into that byte output. For example, the first byte of the output comes from bits [1,9) of the source value, and the second byte of the output reads bits [11,19). Outputs are written in bytes, but each byte may come from completely arbitrary bit positions from the input.

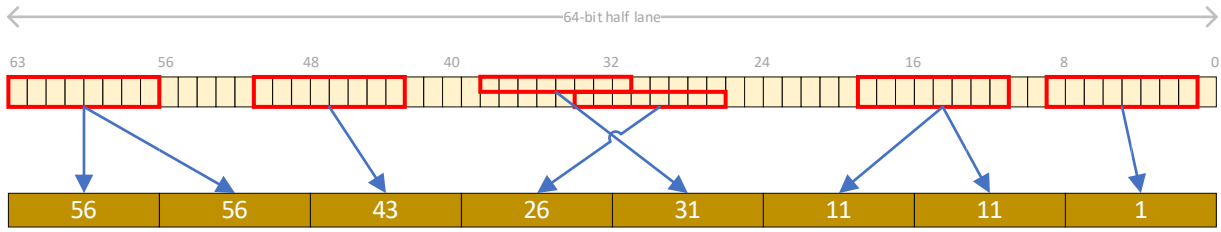


Figure 8. How `_mmX_multishift_epi64_epi8` Works for Each 64-bit Half-Lane

The multishift byte instruction performs the same basic operation for every 64-bit block within the source register (i.e., for 128-bit registers, it would perform two such permutes, for 256-bit it would perform four such permutes, and so on).

The instruction takes 3 cycles and has a throughput of 1 on 3rd Gen Intel Xeon Scalable processors.

### 3.1.5 Clean and Dirty Indexing

All of the dynamic permute instructions described in this section require the indexes to be specified using a separate SIMD register, but the instructions vary a little in how they interpret the contents of that register. Mostly, each element is indexed by the least number of bits required to represent the index. An example is shown in Figure 9, which illustrates the behavior of the

`_mm_permutevar_ps` instruction. There are four possible input elements in the source SIMD value, so two index bits are required. Since each index is represented by the same number of bits as the elements being permuted (i.e., 32-bits in this case), it follows that the upper bits of each index element contain bits that are not used by the index operation itself. Most dynamic permute instructions allow those bits to be completely arbitrary and they only use the lowest bits that are actually required; we shall call this a dirty index. Dirty indexes are useful when computing index values, rather than preloading them. For example, if the index is computed by shifting or rotating values within an index to put the required index bits into the lowest bit position, there is no need to do further work to mask out the unwanted higher bits since the instruction will ignore them anyway.

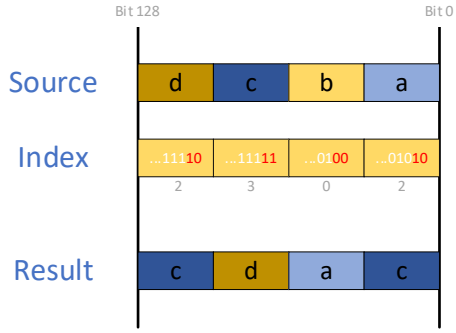


Figure 9. How Dirty Indexing Works in `_mm_permutevar_ps`

In contrast, a few of the general purpose permute instructions (e.g., `_mm_shuffle_epi8`) require what we shall call clean indexes. In those indexes the lowest bits provide the index, but the other bits also convey some meaning to the instruction (see Section 3.1.1 for more details). This can be extremely useful in some circumstances, but it must be accounted for when computing indexes using other instruction sequences, since bits may be left behind in places other than the lowest actual index bits and these change the behavior of the instruction. In such a case, care must be taken to mask out the unwanted bits.

### 3.1.6 Summary of Relative Costs for Different Permutes

In Table 4 we show some example costs for different general-purpose instructions running on a 3rd Gen Intel Xeon Scalable processor. These illustrate how it is best to use in-lane permutes where possible, and for cross-lane permutes to try to use the larger granularities.

For true 16-bit permutes, the instructions should be used since there is no easy alternative, but, where the permutation pattern is known at compile-time, it is cheaper to use the 8-bit permute instead, with pairs of indexes to represent a 16-bit permute.

Note that future processors may have different permutation costs.

Table 4. Latency and Throughput Costs for a Variety of General Purpose Permute Instructions

| Element Granularity | In-Lane | Single-Source Cross-Lane | Dual-Source Cross-Lane |
|---|---|---|---|
| 8-bit | 1/1 | 3/1 (VBMI only) | 4/2 (VBMI only) |
| 16-bit | n/a | 4/1 (VBMI only) | 7/2 (VBMI only) |
| 32-bit | 1/1 | 3/1 | 3/1 |
| 64-bit | 1/1 | 3/1 | 3/1 |

## 3.2 Fixed Purpose Permutes

There are a few common types of permutations or shuffles[2] – broadcasting, duplicating packing and unpacking – that are directly supported using specific instructions since this typically confers some performance advantages.

### 3.2.1 Broadcast

A broadcast operation takes small group of elements – either a single element or a few contiguous elements – and puts duplicated copies of that element group across an entire SIMD register value. For example, Figure 10 illustrates how a single 16-bit element has been broadcast from the lowest position of one SIMD data value into all positions in a different SIMD data value.

[2] Historically many of these fixed-purpose permutes can be found in pre-Intel AVX-512 instruction sets where they were called shuffles instead. Where they are also available in Intel AVX-512 instruction sets the term shuffle is still used to show how they relate to their pre-Intel AVX-512 counterparts.

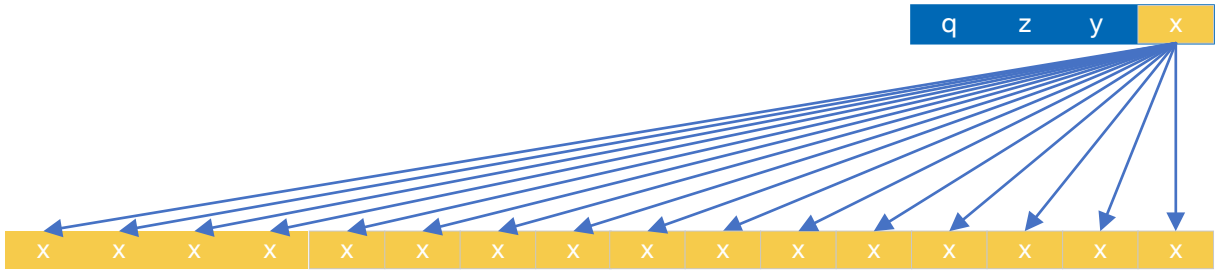


Figure 10. Broadcast of a Single 16-bit Element into a 512-bit SIMD Value

A broadcast from memory is very common and consequently it has been given dedicated hardware support for 32-bit elements and above to enable it to happen as part of a normal load operation and with the same throughput and latency as a normal load. It is therefore advantageous to try to optimize code to broadcast data from memory rather than a register wherever possible. 8- and 16-bit broadcasts cannot be broadcast directly from memory and require a separate broadcast-from-register instruction to be used instead. Figure 11 shows how a group of 4x32-bit integer values stored in memory are loaded and broadcast into every such group within a larger 512-bit SIMD value (this is equivalent to a 128-bit lane broadcast). Note that it can sometimes be cheaper to use the load unit's embedded hardware and perform multiple broadcast loads than it would be to perform a single load followed by several permutes of the loaded value.

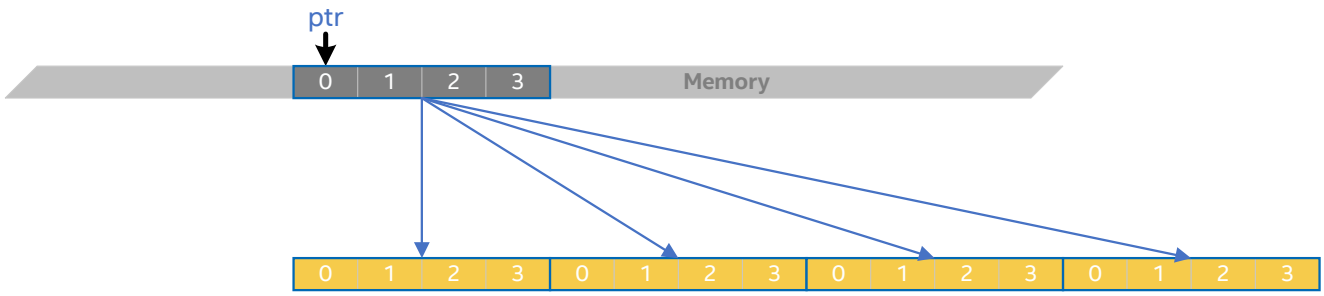


Figure 11. Broadcast of Four 32-bit Elements from Memory into a 512-bit SIMD Value[3]

A broadcast from a register is effectively a cross-lane permute and has the same performance as a general permute that would do the same thing (i.e., 3 cycle latency and a throughput of 1). The advantage of the broadcast instruction is that it does not require an index register to be configured first. Only the lowest element or group of elements may be broadcast. Broadcasting a different element of a register either requires a general purpose permute to be set up or it should be arranged that the value is in memory instead.

In Intel AVX-512 broadcasts of 32-bit elements or wider may be embedded into the instruction itself using a special assembly instruction syntax. These may be handled more efficiently than separate broadcast-and-use instructions since they require fewer registers and increase the possibility of instruction micro-fusion in the processor (see section 18.9 of the Intel® 64 and IA-32 Architectures Optimization Reference Manual). For compiled code, you can use separate broadcast intrinsics followed by an instruction that uses the broadcast value, and the compiler handles the embedding of the broadcast into the instruction by itself.

### 3.2.2 Duplicate Low/High

The duplicate instructions allow one element from each pair of elements in a SIMD value to be duplicated into both elements in the output pair. This is shown in Figure 12, where on the left the lower element in each pair is duplicated, and on the right the upper element of each pair is duplicated.

[3] Convention has changed for this diagram since memory is often depicted as being numbered left to right, and SIMD values as right-to-left. In an attempt to make this diagram simpler, the SIMD order has been reversed to match memory convention.

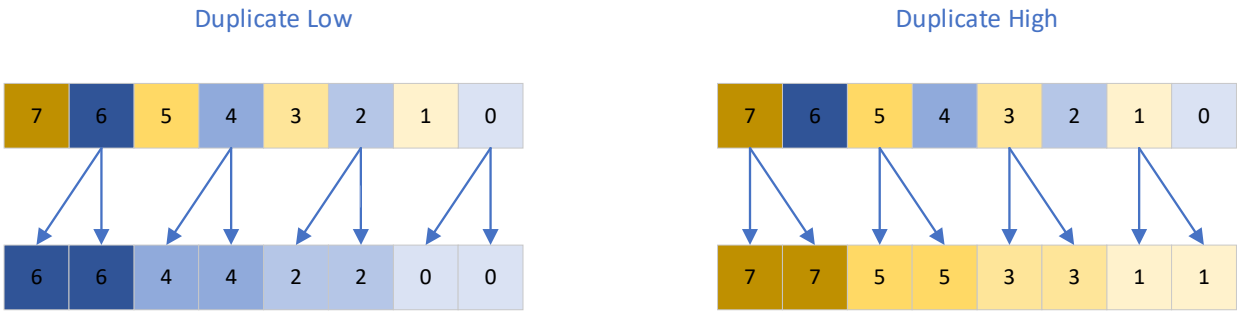


Figure 12. How Duplicate Low/High Instructions Work

There are only three general forms of duplicate:

| | |
|---|---|
| `_mXXX_moveldup_ps` | Duplicate the lower 32-bit values corresponding to a floating-point element |
| `_mXXX_movehdup_ps` | Duplicate the higher 32-bit values corresponding to a floating-point element |
| `_mXXX_movedup_pd` | Duplicate the even elements corresponding to a 64-bit floating-point value |

Note that there is no 64-bit duplicate high instruction. Other 32-bit and 64-bit data types can be duplicated by casting to FP32/FP64 element types (zero cost), performing the duplication, and casting back again. There are no duplicates for 8- or 16-bit elements.

Because duplication is an in-lane operation it has latency of 1 cycle and a throughput of 1.

### 3.2.3 Insert/Extract

Insert and extract operations are conceptually very simple; one or a few contiguous elements from a position in one register are moved to a different position in another register. However, these basic operations are implemented using several different building blocks that achieve smaller pieces of this behavior, and often the compiler is used to synthesize the necessary code sequence. In this section we describe these building blocks, how they can be combined, and why there are often faster alternatives to the obvious implementation.

#### 3.2.3.1 Scalar to/from Lane

The simplest type of insert and extract operations work on individual elements. For example, a 16-bit data element might be inserted into a SIMD value at a given position, or a 16-bit value could be extracted from a given position in a SIMD value. In such cases, the 16-bit element would be represented in a scalar register, and the SIMD value in these cases would be limited to a single lane of data. This is illustrated in Figure 13, where a scalar register is inserted into a SIMD 128-bit lane on the left, and a scalar register extracted from a 128-bit SIMD lane on the right.

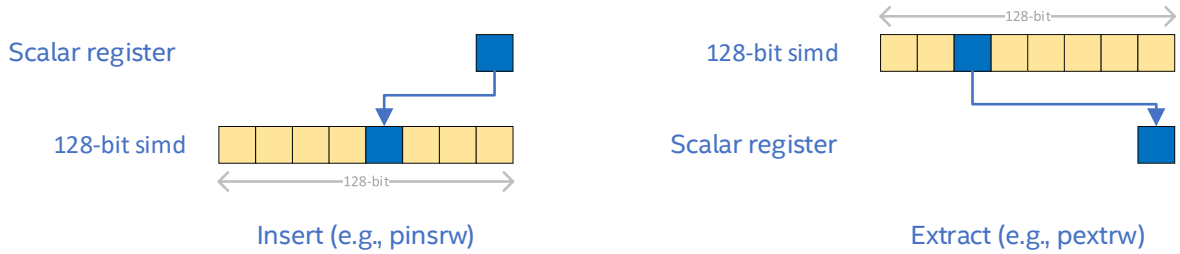


Figure 13. Behavior of Scalar/SIMD Value Insert and Extract Operations

Note that there is no instruction that allows a scalar register to be inserted or extracted using a 256-bit or 512-bit SIMD register. Such operations must be synthesized by you or the compiler, as described in the following sections.

#### 3.2.3.2 Lane to/from register

The next step up in the toolkit of insert and extract operations is to allow a 128-bit lane of data to be inserted or extracted from another register, as shown in Figure 14. The instructions, which do go by many different names, are all essentially doing the same thing. For example, `_mm256_insertf128_ps`, `_mm256_insertf32x4`, `_mm256_insertf64x2`, and so on. The reason for having these variants is to allow the mask behavior to be changed. By specifying the granularity within each lane for the insert or extract operation, individual elements can be zeroed or copied using a bit mask.

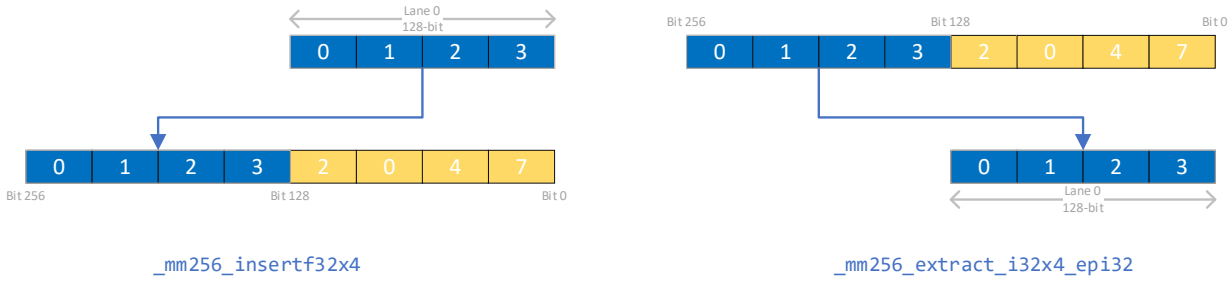


Figure 14. Behavior of Inter-Lane Insert and Extract Operations

The ability to pass a bit mask to the insert operations makes for an interesting possibility; a permute from 3 different sources. For example, `_mm512_mask_insertf32x4` allows a single 128-bit lane to be inserted into another register, and then mask combined with a third register.

Intel AVX-512 also allows a double-lane of 256-bits to be inserted and extracted using, for example, `_mm512_insert_f32x8_ps`.

### 3.2.3.3 Compiler Sequences for Scalar to/from Register

As noted above, there are no instructions for inserting scalar values into SIMD registers wider than a single lane or extracting a scalar value from a SIMD register wider than a single lane. There are C/C++ intrinsics to do this job but they are synthesized by the compiler.

For example, consider the `_mm256_insert_epi16` intrinsic. This can insert a scalar register anywhere within a parent 256-bit register, but contemporary compilers (e.g., gcc 11) generate code that behaves as shown in Figure 15, which illustrates the insertion of a value into position 13. The code sequence is a read-modify-write, since elements other than 13 must not be changed; the top lane is extracted first, then the desired element of that lane overwritten by the new scalar register, leaving the original elements in that lane unaltered, and then the entire lane written back to the desired position.

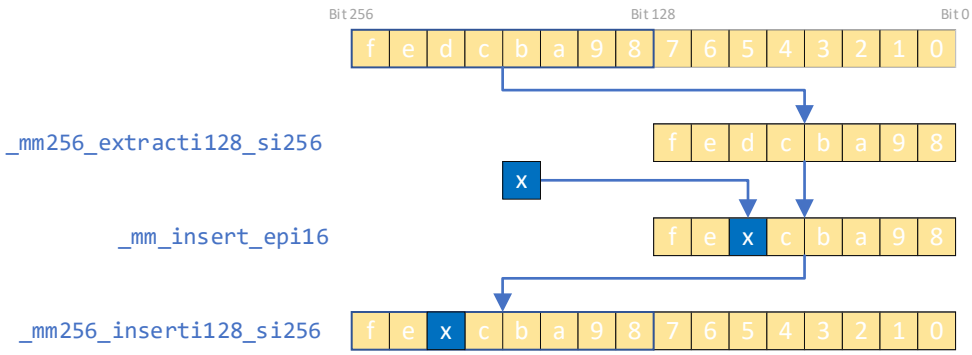


Figure 15. Behavior of Inter-Lane Insert and Extract Operations

There is more than one way to achieve this particular synthesis of the insert (or extract) intrinsics and at time-of-writing LLVM and Intel® oneAPI compilers synthesized this in a slightly different way than GCC.

While the compiler can implement what the programmer desires, as we have just seen, this can result in a synthesized instruction that may be less performant than expected. In such cases it may be worth considering the contents of the next section in which we look at a few ways to avoid insert and extract instructions entirely.

### 3.2.3.4 Alternative Implementation of Insert and Extract

Insertion and extraction can often be somewhat expensive, so in this section we shall look at ways of reducing that expense.

The first and most obvious way to reduce the expense is not to do the operation at all! It has been noted already that the different sizes of register are overlaid on top of each other; the xmm0 register is the lowest 128-bits of the ymm0 register, which in turn is the lowest 256-bits of the zmm0 register. Thus, rather than inserting and extracting to lane 0 (e.g., `_mm512_insertf32x4(x, 0)`) the SIMD value can simply be cast to a register of a different size instead (e.g., `_mm512_castps256_ps512(x)`). The compiler does that with no cost whatsoever.

The second way of reducing expense is to try to avoid using the scalar registers. Although the scalar registers are a natural way to store individual values, moving data to and from those registers from SIMD values (particularly multi-lane values) can be expensive. For example, consider the following code fragment that extracts a value from one SIMD register, and inserts it back into another:

```
__m256i moveElement(__m256i x, __m256i y)
{
  auto e = _mm256_extract_epi16(x, 6);
  return _mm256_insert_epi16(y, e, 2);
}
```

In this scenario, it would be cheaper to perform a permutation directly, where a two-source permute is used to copy all the values from y to the same position except for element 2 that is copied from the other source. This becomes a direct SIMD-to-SIMD instruction that is much more efficient. Unfortunately, the contemporary compilers at time of writing did not spot that opportunity in the code above, so the programmer needs to be aware of this optimization. Note that even when the scalar value being inserted or extracted is manipulated in some way, it may still be cheaper to do that by pretending that it is part of a simd value and operating on the complete SIMD than converting to a scalar, operating on it as a scalar, and then inserting it back again.

The final technique is to use a broadcast instruction instead of an insert instruction. A broadcast instruction, such as `vpbroadcastd`, can operate from a scalar register, and broadcast the scalar value to all elements of a SIMD register. Even better, it can be given a mask register to decide exactly which elements receive the broadcast value. If a mask with only a single set bit is presented, it inserts the element value into exactly one place. A single instruction can insert into any element position of an Intel AVX or Intel AVX-512 register without being converted into a synthesized multi-instruction sequence.

## 3.3 Alignment Operations

Alignment instructions all perform the generic operation illustrated in Figure 16, where two source objects are concatenated, and part of the resulting object extracted. They allow the upper part of one data object to be contiguously combined with the lower part of another data object.

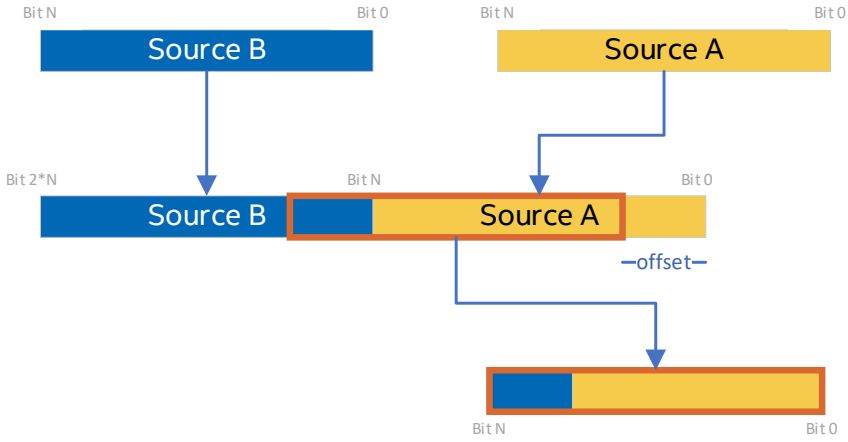


Figure 16. How to Perform Alignment of Two Data Objects

All Intel AVX-512 alignment instructions follow this basic form but can differ in the granularity at which the alignment is performed: register, lane, or element. Most alignment instructions are immediate-valued and can only extract the subset of values (i.e., the red box in Figure 16) from a fixed compile-time position, although a few of the more recent instructions introduced in 3rd Gen Intel Xeon Scalable processors are capable of dynamic (runtime) selection. A selection of the available alignment instructions is described in the following sections. In many of the cases the instruction used is called `alignr'. Here, the `r' suffix indicates that the combined value is being shifted to the right by a given offset to move data into the lowest part of the output register.

Note that a general-purpose permute instruction could replace most of the alignment instructions listed here, but these instructions are useful because they often have smaller encodings, can operate without having to load an indexing register first, and reduce register pressure since they do not require an index register.

### 3.3.1 Per-Register Alignment Instructions

Per-register alignment instructions allow the original source values to be entire registers of any supported SIMD bit-width (i.e., 128-bit, 256-bit, or 512-bit). They operate as per Figure 16, where the data sources are whole registers.

Per-register alignment instructions may only shift by offsets that are known at compile-time, and only at the granularity of whole 32-bit or 64-bit elements. This granularity restriction is imposed because an immediate value (the compile-time shift offset) has a limited range and would not be large enough to express a shift at a smaller granularity. Note that only shift-right is supported, but with suitable manipulation shift-left is possible too.

The per-register instructions have the same performance as a cross-lane permute instruction of the same size (e.g., 3 cycle latency and throughput of 1 for `_mm512_alignr_epi32`).

Examples of per-register alignment instructions include `_mm512_alignr_epi32`, and `_mm256_alignr_epi64`.

### 3.3.2 Per-Lane Alignment Instructions

Per-lane alignment instructions can perform as many alignments as there are lanes within the source data objects. A per-lane alignment is illustrated in Figure 17, where, in this case, the original source objects are 256-bit Intel AVX registers, each containing two 128-bit lanes. Respective lanes from both sources are aligned with the same lane in the other object, and the resulting output written to the respective lane of the output data object.

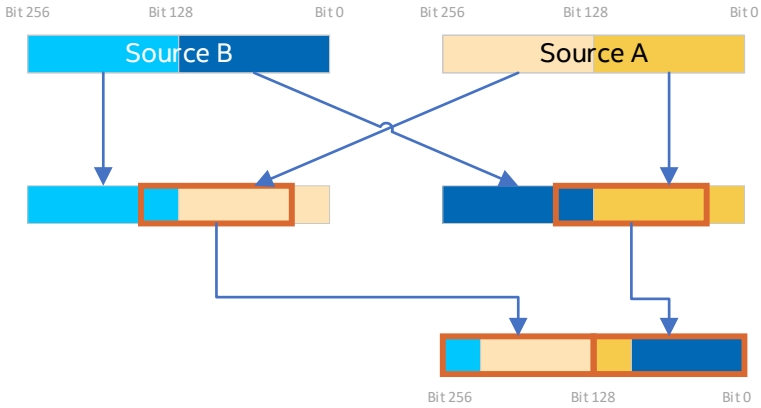


Figure 17. Per-lane Alignment

Per-lane alignment instructions may only shift by offsets that are known at compile-time but they may do so at byte granularity, which means they effectively allow any element granularity shift to be handled. Per-lane alignment may only shift right, but with suitable manipulation of the shift, left shift is also possible. All lanes are constrained to shift by the same offset.

The per-lane instructions have the same performance as an in-lane permute instruction of the same size (e.g., 1 cycle latency and throughput of 1).

Examples of per-lane alignment instructions include `_mm512_alignr_epi8` and `_mm256_alignr_epi8`.

### 3.3.3 Per-element Alignment Instructions (VBMI2)

The VBMI2 family of Intel AVX-512 instructions introduced in 3rd Gen Intel Xeon Scalable processors allows for finer grained alignment instructions than the previous alignment instructions described above, adding 16-, 32-, and 64-bit alignment granularities. An example of per-element alignment is shown in Figure 18. In this case each source value contains four data elements (e.g., 64-bit elements in a 256-bit SIMD value). Each respective pair of elements taken from the two sources is combined into a single contiguous value and part of that new value extracted and inserted into the respective element of the result.

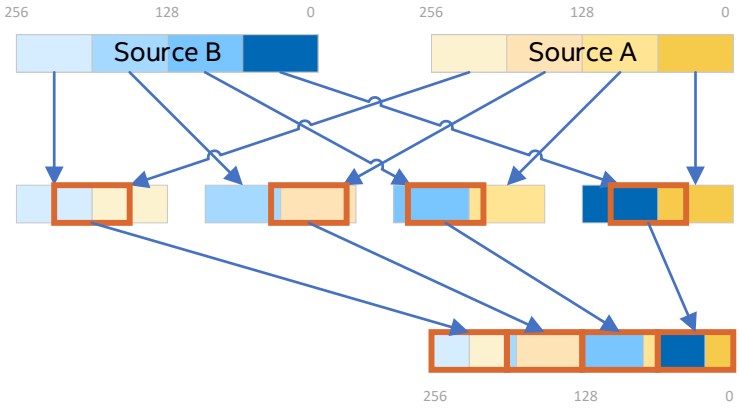


Figure 18. Per-element Alignment

Per-lane alignment instructions may shift by offsets that are either static (through an immediate) or dynamic (by using an additional register to store the shift offset for each element). These per-element alignments may also shift both left and right and with a bit-level granularity. Only 16-, 32-, and 64-bit elements are supported.

The per-element instructions have the same performance as other shifts and rotations (i.e., 1 cycle latency and throughput of 1).

Examples of per-lane alignment instructions include `_mm512_shdi_epi16` (shift left in 16-bit elements by an immediate) and `_mm256_shrdv_epi32` (shift right in 32-bit elements by a variable index).

### 3.3.4 Element Rotation

All of the alignment instructions discussed above can be used to perform an element rotation by using the incoming source value for both operands of the instruction. An example is shown in Figure 19 where an incoming SIMD value containing four elements has the elements rotated from the original order [0, 1, 2, 3] to the new order [3, 0, 1, 2].

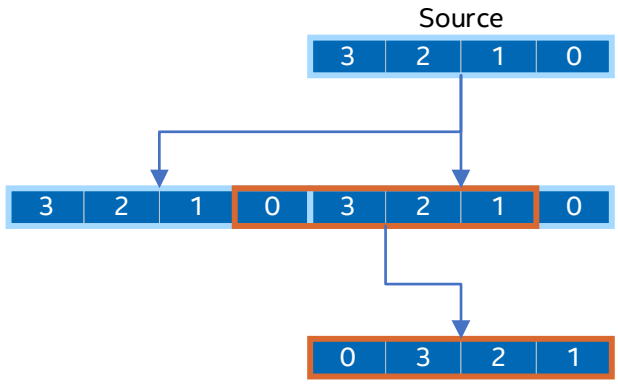


Figure 19. Element Rotation within a Single SIMD Value Using the Alignment Instruction

## 3.4 Bit-Wise Expansion and Compression

The expansion and compression instructions are special types of permute that use a bit mask to specify the indexes, rather than using a numeric index. An illustration of these instructions is shown in Figure 20. On the left, the expansion instruction writes consecutive elements from a source value into the output elements, which have an active (i.e., set) mask bit. Any inactive output elements are zeroed in this example, because the maskz variant is used, although a second source register may be used to supply the values to write into those elements. The compress instruction operates in the opposite sense, taking active masked values from an input source register and writing them into contiguous positions in the output register. Compress is useful for implementing operations such as C++ remove_if or copy_if functions from the standard algorithms (e.g., https://en.cppreference.com/w/cpp/algorithm/copy).

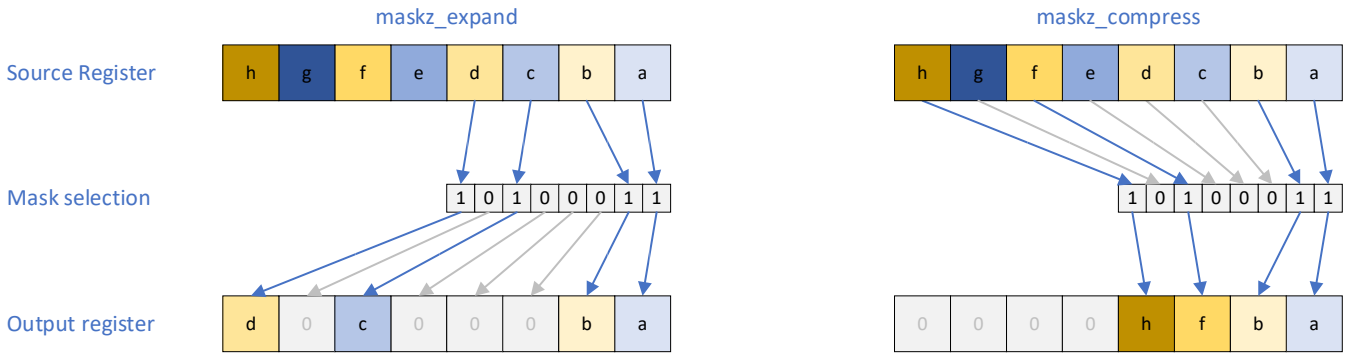


Figure 20. How the Bit-Wise Expansion and Compression Instructions Work, using a Zeroing Bit Mask

The compression and expansion instructions were first added in Intel AVX-512, and then only for 32-bit and 64-bit granularity elements. Support for 8- and 16-bit granularity elements was added in the 3rd Gen Intel Xeon Scalable processors VBMI2 instruction set. If a compress or expand operation must operate on 8- or 16-bit granularity on machines that do not support VBMI2, then you should convert the values from one format to another. For example, 16-bit values could be converted to 32-bit, then the compress or expand operation performed, and the 32-bit result turned back into the final 16-bit output. Note that such a conversion means that only one half or one quarter as many elements can be processed.

The compress and expand instructions are more expensive compared to the other permutes described in this document. We do not recommend using these instructions when other permutations can perform the same operation. For example, rather than using the bit mask 100100100100 to expand or compress every third element it would be more effective to set up an indexed permute with the index [0, 3, 6, 9, ...].

Compression and expansion of values without Intel AVX-512 is more difficult, and code sequences to synthesis these operations must be used instead. A future document will describe how to achieve those operations.

## 3.5 Gather and Scatter

The gather and scatter instructions allow data to be permuted into or out of memory using a supplied SIMD index value. An example of a gather instruction is shown in Figure 21. A SIMD of index values is supplied, along with a base pointer that addresses a region of memory. Each index is then used to dereference the memory allocation at the index' offset from the base pointer. The value at that memory location is then placed into the respective element in the output register. This continues for every index element until all the result elements have been read.

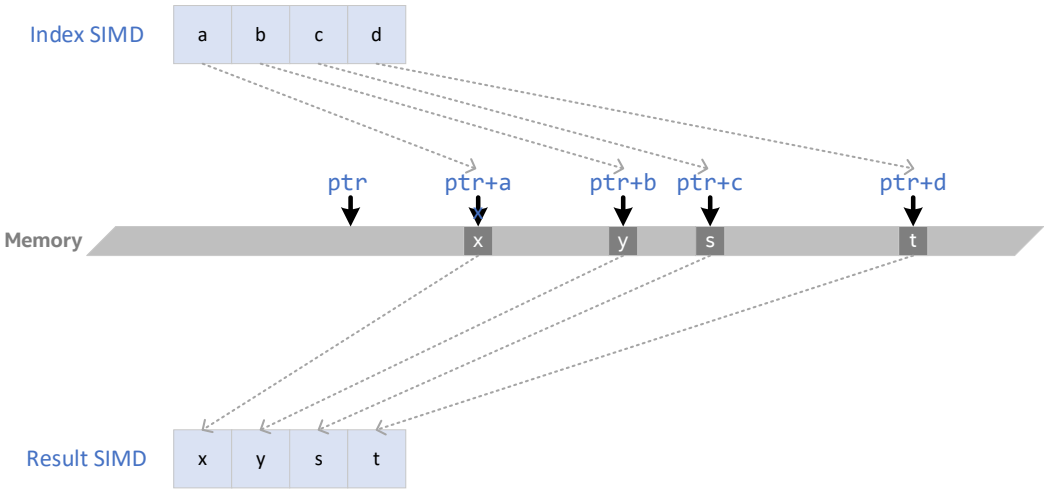


Figure 21. How Memory Gather Instruction Operates

The scatter instruction works using the same general principle but writing the value of a source register element into the memory at the given index from a base pointer.

The gather and scatter instructions are excellent for working with data that is stored in arbitrary dynamic memory locations. The instructions handle all the necessary insert/extract operations of the indexes and the values that are passed back-and-forth to

the memory system. However, the performance of the gather and scatter instructions is comparable to using individual read and write operations. For example, a gather operation with 16 indexes requires 16 individual load instructions. The gather handles the indexing, which adds a little efficiency, but it still needs to perform each of those 16 load operations.

When the data being read or written is stored in memory in a structured pattern it is often faster to use a software sequence instead. For example, if data were to be read at a stride of 4, this could be done using a single gather instruction and an index sequence of [0, 4, 8, 12, ...]. However, performing fewer loads of the data into registers and then using register permute instructions to extract every fourth element is faster.

## 3.6 Reductions

A common use for permutation instructions is to be able to perform a reduction operation. For example, given a SIMD value of multiple elements, compute a single value representing the sum of all elements in that register. One way of achieving this operation is through a reduction tree, which can be generated automatically by the compiler. For example, the intrinsic named `_mm512_reduce_add_ps` is turned into a sequence of operations that repeatedly split a SIMD into two pieces, combine them, split them again, and so on. There are a number of such reduction intrinsics available from the compiler to cover minimum, maximum, addition, and multiplication reductions.

The Intel AVX-512 instruction set family does not currently have any instructions that perform any type of full reduction across an entire SIMD data value. However, there are a few instructions that perform partial reductions, typically on pairs of adjacent values. Figure 22 shows an example where all the elements in one SIMD value are multiplied by their respective element in another SIMD source value, and then adjacent pairs of data added together. The multiplication and the addition take place in a data type that is twice as large as the original elements, thereby allowing high precision to be accumulated. An example of such an instruction is `_mm512_madd_epi16`, which takes 16-bit input elements and generates 32-bit accumulated output elements.

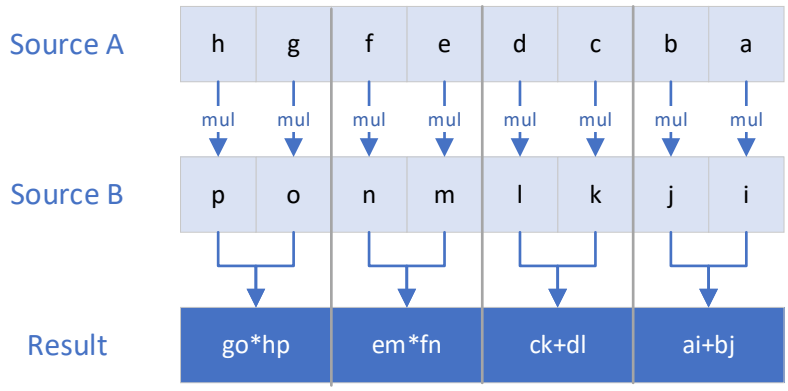


Figure 22. Basic Behavior of a VNNI Dot-Product-Like Instruction (e.g., `_mXXX_madd_epi16`)

In addition to the older instructions that operate on this principle, many of the newer Vector Neural Network Instructions (VNNI) available as part of Intel's DLBoost technology also work in this way, although they also are able to accumulate their values as well (i.e., act as a fused-multiply-add). Some of the new VNNI instructions can accumulate up to four horizontal values.

Partial reduction operations can be used simplify bigger full-width reductions by replacing some of the reduction tree stages with a partial reduction instruction.

## 3.7 Fake Permutes

This document mostly considers real permutation instructions, but sometimes it is possible to achieve the same effect as a permutation instruction by using another instruction entirely. This may have one of two advantages:

**Execution Parallelism** — If the alternative instruction executes on a different port, then it can execute in parallel with other permute instructions. For example, Table 3 shows that there is only one 512-bit execution port capable of performing permutes, so using an instruction that has the same effect as such an instruction but runs on a different execution port increases parallelism and performance.

**Encoding** — No need to use a register to store an index to achieve a particular permutation

The following subsections describe a few common ways to use other instructions to achieve permutations.

### 3.7.1 Shifts and Rotates

Shifts and rotates allow data to be moved back and forth with a lane or element at the bit-level, rather than the byte or element level. This can be very useful when operating on data that has an irregular size. Shifts and rotates do not allow data to be easily combined with other sources (with the exception of mask-like operations), so they are more useful in moving data within a register rather than across registers, but this is still very useful. Shifts and rotates exist at all levels from entire 128-bit lanes (e.g., `_mm256_bslli_epi128`), and down through most element sizes (i.e., 64-, 32-, and 16-bit). Shifts and rotates are not directly supported for 8-bit elements, but in 3rd Gen Intel Xeon Scalable processors onwards the GFNI ISA can be used to provide those too. For more information, see Galois Field New Instructions (GFNI) Technology Guide.

### 3.7.2 Data Conversions

Integer data conversions provide a way to expand and contract data on what is effectively a power-of-2 stride. In Figure 23 an illustration of a stride-by-2 permute (extract even elements) is performed. It works by treating the original set of 32-bit values as though they were actually 64-bit, and then extracts the lower 32-bits of each 64-bit element.

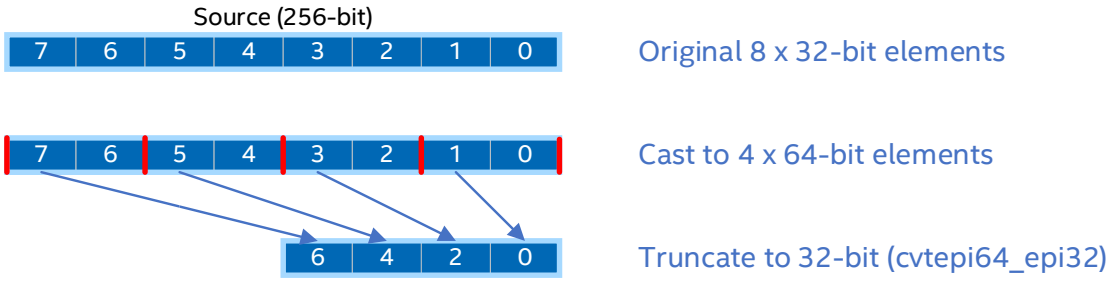


Figure 23. Use of Truncation-Conversion to Perform a Stride-By-2 Permutation

Note that conversions to very small granularities (e.g., using `cvtepi64_epi8` to perform a stride-by-8 conversion) may be implemented in a series of smaller steps by the compiler or processor, so you should bear this in mind while optimizing SIMD reordering algorithms that use them.

## 4 Worked Examples

This document has so far described many different ways to exploit different types of permutation and data-movement instructions, but without concrete examples it may be difficult to fully understand the advantage of using some permutation instructions over others. In this section we go through some case studies of common permutation algorithms, showing how different combinations of instructions may be used to good effect, and highlighting some of the factors about permutation that should be considered. We look at matrix transpose, horizontal reductions, and full-width SIMD bit shifts.

### 4.1 Matrix Transpose

Matrix transpose operations are useful in many different basic linear algebra subprograms (BLAS) and also for performing array-to-struct and struct-to-array conversion for enabling high-performance SIMD algorithms to be implemented on multiple data sets. The transpose is also very similar to building blocks for operations such as reductions (examined in more detail in Section 4.2) and an understanding of how they work is useful there too. An example of a simple 8x8 matrix transpose is shown in Figure 24. Note that conventional mathematical element ordering is used (left-to-right) rather than the reversed ordering used in the other diagrams in this document.

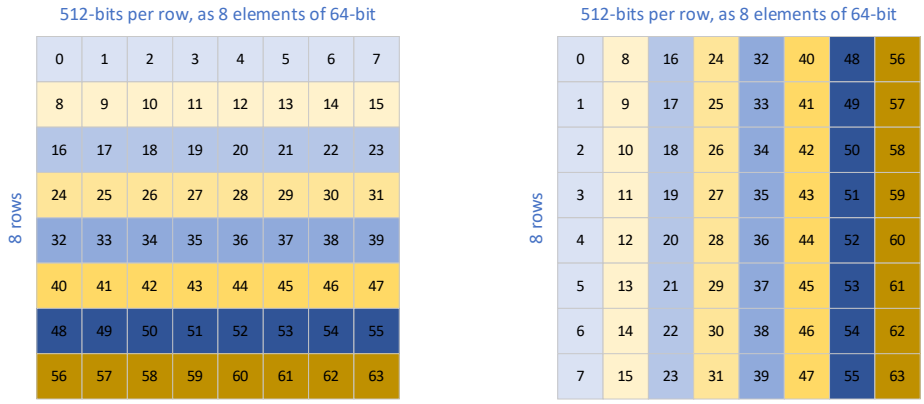


Figure 24. Effect of an 8x8 Matrix Transpose

There are many different ways to perform transpose. The exact method used can vary depending upon different factors, such as matrix size and shape, the granularity of transposed elements, and whether the matrix is in memory or registers. In this section we illustrate one way to transpose an 8x8 matrix of 64-bit elements (e.g., 64-bit double, 32-bit complex<float>) from memory and back into memory. We shall use a hierarchical decomposition method for performing the transpose, which is efficient, works well across a range of Intel processor generations, is easy to comprehend, and is easy to adapt for different use cases. It is expected that for specific use cases the code could be optimized further to exploit the properties of those specific scenarios.

A matrix transpose can be implemented as a series of sub-transposes, and this is illustrated in Figure 25. On the left-hand-side, the original matrix is laid out, with colors showing each different row. The data is then transposed in a series of stages, where each stage performs a transpose at a different granularity. For example, the top row shows how the matrix is originally treated as a 2x2 matrix, and elements (0,1) and (1,0) are transposed with each other, leaving the diagonal elements where they are. Each quadrant of that transpose then is treated as smaller matrices that are themselves transposed again. This continues until in Stage 3 the final transpose at the smallest element granularity results in the total transposing being completed. The bottom row of that same diagram shows that the reverse sequencing of transposes could be performed with the same effect. In this case the smallest transposes are done in Stage 1, leading through to the full transposes in Stage 3. The overall effect is the same.

The choice of whether to use a big-to-small or small-to-big series of transposes can affect the overall performance of the function. The small transposes (i.e., Stage 3 of the upper flow or Stage 1 of the lower flow) only move the data within a lane and tend to be cheaper than the multi-element inter-lane transposes at the opposite end of their respective flows. For a complete transpose, the total number of instructions needed to do the cheap or expensive stages remains the same, so the order does not matter for this example. However, it can sometimes be desirable to choose one order over another if that exposes useful instructions. For example, it may be cheaper to do a small-to-big flow order if that means that the VNNI unit could be used to perform an initial partial reduction, or the load unit's element-duplication feature can be exploited to shift the data by one element position when reading from memory. The choice of which flow order to use for different scenarios is left as an exercise for the reader according to the precise size, type, and context of transpose required.

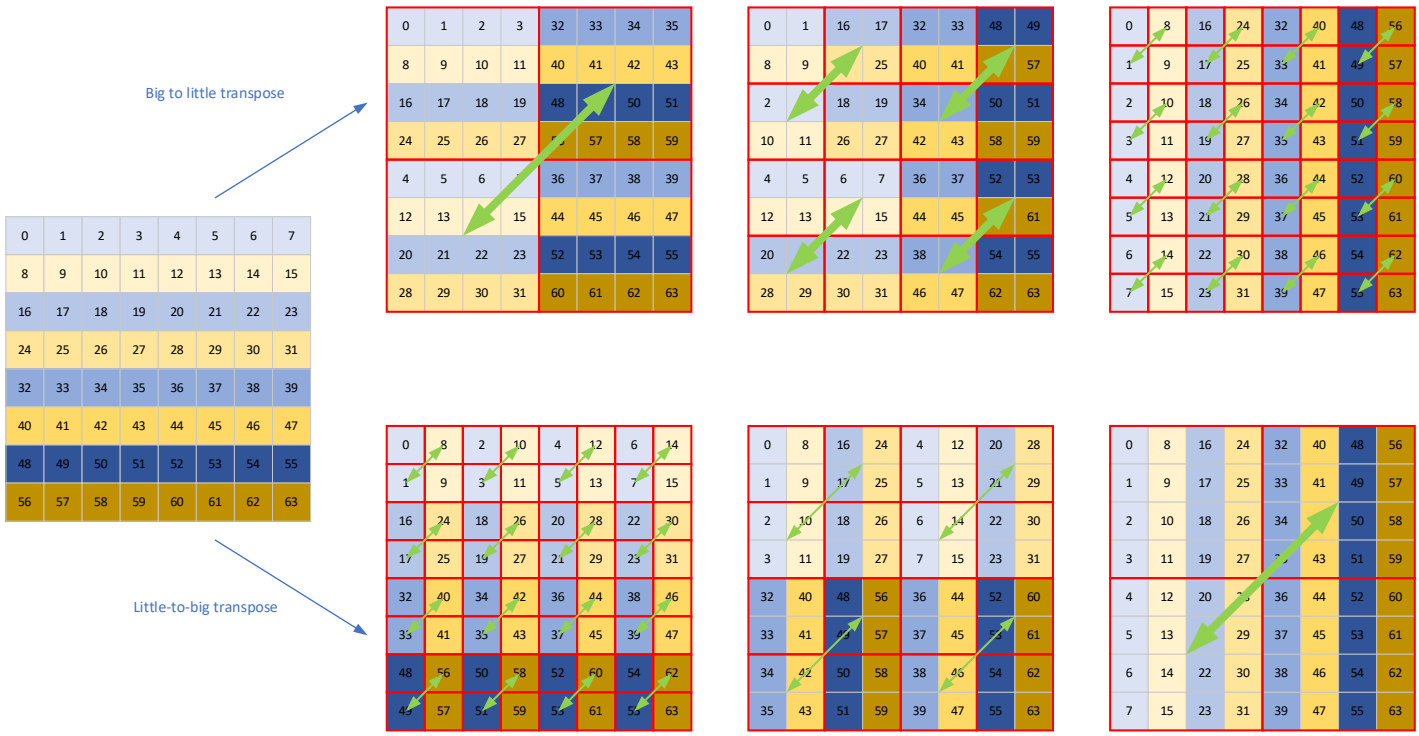


Figure 25. How to Perform a Transpose as a Series of Sub-Transposes of Different Sizes

The idea of using a series of smaller transposes to transpose bigger matrices works for matrices of any size, shape, element type, and storage medium (register of memory).

It is possible to use the technique illustrated in this section to build transposes for matrix objects that are bigger than would fit into registers too. Such an example is beyond the scope of this document.

## 4.2 Multi-data-set Reductions

Consider the data structure illustrated in Figure 26. This shows an array of values in memory, where the elements are divided into small groups of 8 contiguous elements (we assume that each element is 64-bits) and each group must be reduced into a single scalar value by summing all the values in the group. Note that the reduction could equally well be another type of operation (e.g., minimum).

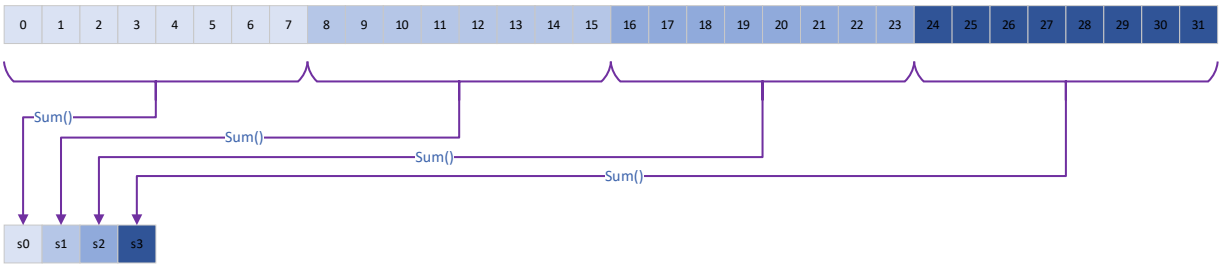


Figure 26. Example Data Structure Showing a Series of Values That Must Be Reduced

Leaving aside the most trivial implementation - reading each element one-by-one and adding it to the appropriate output accumulation element - there are three increasingly sophisticated ways to implement this code.

### 4.2.1 Iterative Reduction Intrinsic

Most modern C/C++ compilers implement reduction intrinsics and these allow the code to be written as shown in Figure 27. A simple loop reads each group into a single Intel AVX-512 register (8 x 64-bit elements) and then directly calls the appropriate reduction intrinsic (https://godbolt.org/z/soEnGqhvz). The code this generates is illustrated in the same diagram.

```
for (int i=0; i<n; ++i)
{
```

```
    __m512i values = _mm512_loadu_si512(values_to_sum + i * 8);
    reductions[i] = _mm512_reduce_add_epi64(values);
  }
```

```
.LBB0_5:
        vmovdqu ymm0, ymmword ptr [rdi + 8*rcx + 32]
        vpaddq  zmm0, zmm0, zmmword ptr [rdi + 8*rcx]
        vextracti128    xmm1, ymm0, 1
        vpaddq  xmm0, xmm0, xmm1
        vpshufd xmm1, xmm0, 238
        vpaddq  xmm0, xmm0, xmm1
        vmovq   qword ptr [rax + rcx], xmm0
        add     rcx, 8
        cmp     r8, rcx
        jne     .LBB0_5
```

Figure 27. Source Code Showing How to Perform a Reduction as a Series of Permute and Reduce Steps

Notice that there is no single instruction that can implement a horizontal instruction addition so the compiler has synthesized the operation using a series of permutes and adds. The code is organized into a reduction network, where the SIMD value is successively split into two pieces that are added together, continuing until only one element remains.

This code performs reasonably well but there are two sources of inefficiency:

1. The full width of the processor's SIMD capabilities is left unused. The first stage only uses 256-bits of the full Intel AVX-512 register, the second stage only 64-bits, and so on. Ideally, SIMD processors should do something with the entire width of a register at any given point to maximize efficiency.
2. There is a critical path exposed in this code caused by each permute-and-add stage being dependent on the results of the previous stage. There is no opportunity to exploit parallelism, and the full latency of the permute instructions is exposed, which leads to processor stalls

By unrolling the loop, it becomes possible to overlap multiple iterations, which hides some of the effects of the stalls on the critical path, but it does not fix the SIMD inefficiency issue. Our next code fragment attempts to address that.

### 4.2.2 Using Transpose to Improve a Reduction

To improve the efficiency of the code, we could exploit the fact that multiple groups are being reduced, and each group is independent of the others. This opens the way to allow us to reduce several groups in parallel with each other.

SIMD processors are typically better at vertical (map-like) operations than horizontal (reduce-like) operations, so we could improve the efficiency of this code by turning the horizontal reduction operation into a vertical summation by using a transpose on the data first. This is illustrated in Figure 28. On the left we have our 8 original rows of data, and we call `reduce_add_epi64` on each row. This in turn generates the reduce-add network described in the previous section. In contrast, if we first transpose the data so that respective elements are positioned in the same index of each of a series of SIMD rows then we can trivially add all the rows together using the efficient `_add_epi64` vertical operations, which make full use of SIMD. This is faster overall because a single large transpose of multiple data sets uses the entire SIMD data width, while the series of individual row reductions only uses partially filled registers and requires more instructions overall. Note that the larger the matrix, or the smaller the granularity of the individual elements, the more efficient it is to perform one large transpose than many smaller reductions.

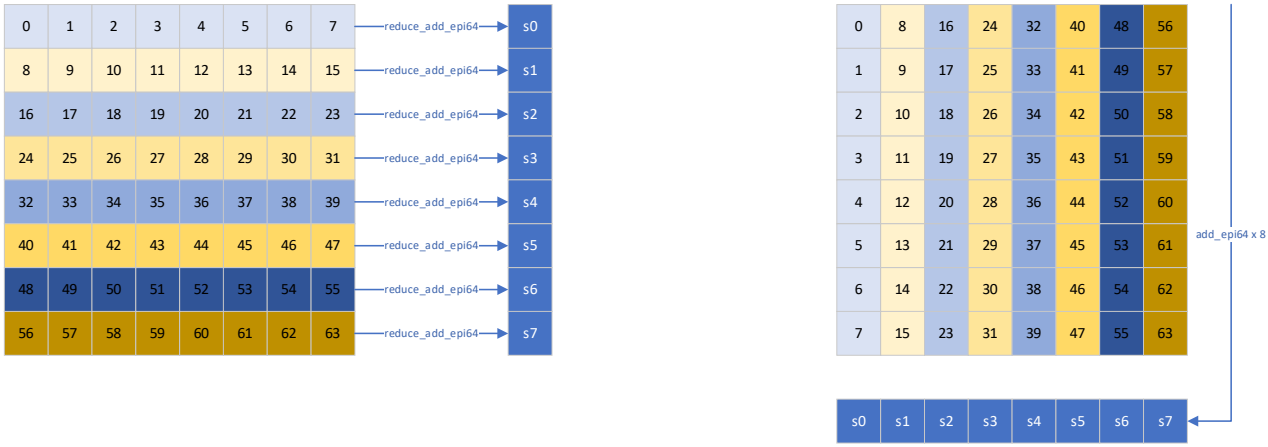


Figure 28. How Multiple Reduce-Add Groups Can Be Transposed to Convert the Horizontal Reduction Operation into a Vertical Summation

Note that we leave the mechanics of handling multiple iterations to the reader. In practice this code might need to be able to handle arbitrary numbers of groups, and such code would require suitable masking, unrolling, or specialization to deal with those issues. Also note that if it were possible to arrange for the data to be transposed while it was generated, or during a storage phase of computation, so that the data as in an efficient storage format to begin with, the cost of this code is entirely eliminated.

You should also beware of the destination of the transpose. If the data is transposed into memory, and then read back again for the summation, this may end up wiping out any performance gains because the memory system could potentially introduce a bottleneck. It is preferable to sum the values after they have been transposed into registers and before they get stored back to memory.

### 4.2.3 Integrated Transpose-and-Reduce

We can further optimize the code from the previous section by exploiting the knowledge that the data is undergoing a reduction. At each stage of the transpose, rather than simply transposing the data, we can reduce it too. The reduction decreases the amount of data in flight, making the computation of subsequent stages cheaper. This is illustrated in Figure 29. The original data starts out on the left-hand-side, where we are computing the sum of all the values in each row to form a single output value. In Stage 1 we transpose with adjacent rows and sum the resulting data. This means that every pair of rows uses a summing reduction to create one output row. In Stage 2 we now do the same transpose-and-sum operation again. In Stage 3, the number of rows of data has reduced again, and we continue to transpose-and-reduce until we end up with the final answer.

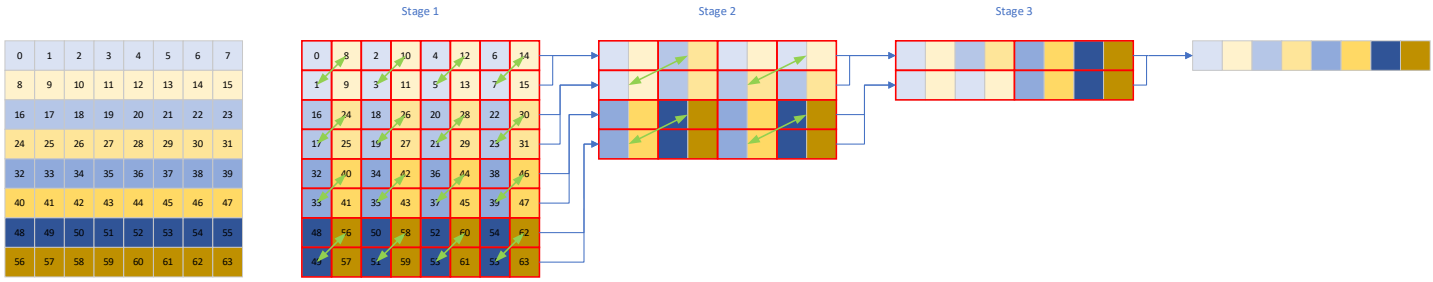


Figure 29. How Combined Transpose-And-Reduce Can Result in the Amount of Data Being Processed at Each Stage of the Transpose Can Be Reduced

In the previous section we described how we could transpose the entire data matrix first, and then sum it. This required three stages, each containing 8 permutes, giving a total of 24 permute instructions. In the new combined transpose-and-reduce version, the first stage needs 8 permutes, the second 4, and the third only 2, giving a total of 14 permutes. This is almost a halving in the number of required permutes.

In the previous section we also showed how the transpose could be implemented from little-to-big or big-to-little, as illustrated in Figure 25. Some of the stages are more expensive than others, but the overall transpose has to perform all the permutes at each stage, so it ultimately does not really matter in what order the data is processed. In contrast, it does make a difference in the combined transpose-and-reduce. Since we have more data in the Stage 1, it makes sense to use the cheapest permute to start

with when we have many permutes to do, and to use the more expensive permutes in the later stages when we have fewer of them.

We can go a tiny bit further still in optimizing this code by exploiting another property of the reduction: each reduction stage for this example (and others like a minimum) is commutative. To illustrate why this is important first consider the original implementation described above. In Figure 30 we show the transpose-and-sum of the first two rows. This operation requires two permutes and an add.

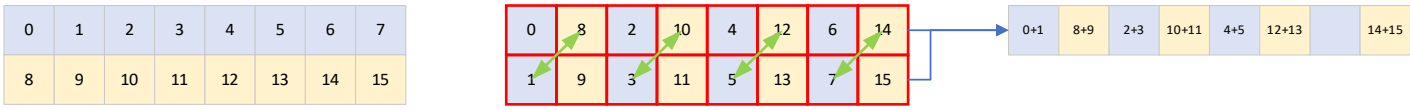


Figure 30. Transposition and Summation of Two Rows of Data

However, the addition step does not require that the values in Stage 1 are necessarily in the order that these are currently given. We could change the order of the Stage 1 values to be slightly different, as shown in Figure 31. The difference is very subtle. Every odd column contains the same elements as previously, but in reversed order. This slight change means that the instructions needed to implement this operation are a mask, a permute, and an add. One permute has been swapped for a potentially cheaper mask instruction, giving a slight performance improvement.


Figure 31. An Alternative Form of Transposition and summation That Exploits Commutativity Addition Reduction

## 4.3 Full-Register-Width Shifts (and Rotates) Using VBMI

Suppose that you wish to shift (or rotate) the bits across an entire 512-bit register. There are many ways to achieve this, taking advantage of properties such as whether the shift is dynamic or compile-time, whether it is a multiple of some number of bytes or is at the bit granularity, and whether it is logical or arithmetic. Specific types of shift or rotate can be coded up using fast specialist sequences, and optimizing specific sizes and shifts is left as an exercise for the reader. However, it is useful to illustrate the thinking behind the implementation of one specific type of shift and the problems that have to be solved, and to this end we shall implement a shift of an Intel AVX-512 value by the compile-time offset of 12 bits, as illustrated in Figure 32. That figure shows the 64 bytes within a register, and how the value has been shifted right by 42 bits.

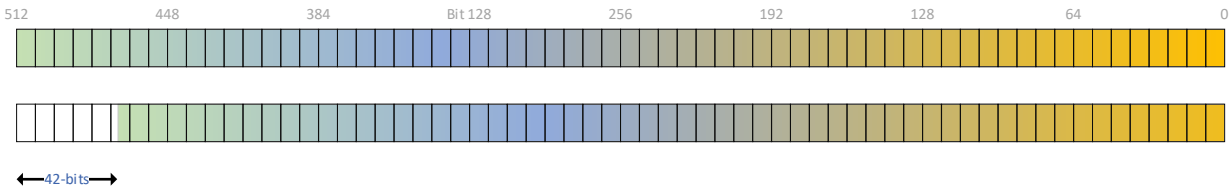


Figure 32. A Bit-Level Shift by a Large Offset in an Intel AVX-512 Register

The main issue with a bit-level shift of this type is that bit-level shifts operate within elements. The naïve use of a plain shift in 64-bit elements would result in bits being destroyed by shifting in zeros, as illustrated in Figure 33.

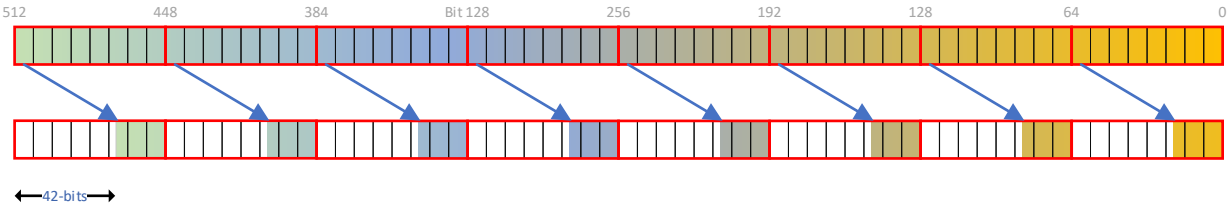


Figure 33. How 64-bit Element Shifts Destroy Data Bits

To fix this issue, it is necessary to take the bits from the adjacent 64-bit element, shift those in the opposite direction, and then combine the top sets of bits together. However, with the VBMI instruction set this can be conveniently done using the new element-wise alignment instructions (e.g., `VPSHLDQ`) as illustrated in Figure 34.

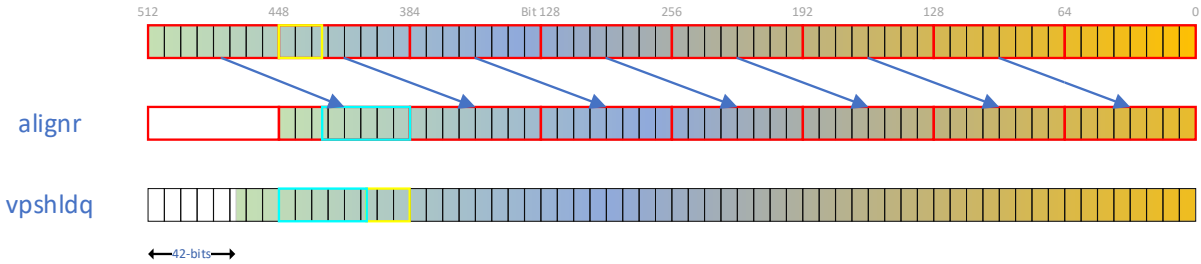


Figure 34. Implementing Compile-Time Shift Using Lane-Wise and Element-Wise valign{d,q} Instructions from VBMI

In this example, the bit shift is less than the size of the 64-bit element, so we begin by moving all the 64-bit half-lanes to the right by one element offset, so that two adjacent 64-bit groups from our original SIMD value are now positioned in the same 64-bit element position in two different registers. Thus, the top bits of one half-lane and the bottom bits of the adjacent half-lane that will be shifted in are now in the same element position. Finally, we use the VBMI 64-bit aligned shift to combine the two sets of bits – shown in yellow and blue boxes – into a contiguous block of bits that represents the final output. This is of course repeated across every 64-bit element.

Shifts by variable bit-level offsets are a little harder since the initial alignment instructions to get the correct 64-bit half-lanes in place do not allow dynamic shifts, and a permute should be used instead. After the data is in the correct half-lane, the element-wise align can be used as that allows a shift by a dynamic offset. Building the dynamic alignment is left as an exercise for the reader.

# 5 Summary

In this document we have looked at the many and varied ways that Intel AVX-512 allows data elements within a SIMD value to be reordered, moved, and combined with each other. At one extreme, elements can be moved very short distances in fixed patterns, while at the other extreme, data can be moved across entire registers in essentially arbitrary patterns, albeit at greater computation cost.

We have provided a few worked examples showing how different types of permute can be layered together to build up complex permutations such as transposes, reductions, and very wide shift operations.

Other examples of how permute instructions can be used are provided in the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

# Appendix A Compiler-Assisted Permute Generation

Throughout this document we have described the various types of permutation possible, and the intrinsics necessary to use them. We recommended that the programmer should try to use the cheapest and most efficient types of permute instructions that solve the task at hand, rather than relying on using the general purpose permute. However, knowing about and being able to use the best instruction for the job can be difficult due to the size and complexity of the Intel AVX instruction sets. To solve this issue, we can enlist the compiler to help us.

Recent versions of GCC, LLVM, and Intel oneAPI compilers include some support for SIMD vector convenience built-ins, data types, and a convenient syntax for manipulating SIMD values. One little-appreciated built-in compiler function is `__builtin_shufflevector`. This built-in allows the programmer to specify what permutation they desire, and the compiler tries to determine the cheapest way to provide it. Table 5 shows a small selection of permute index sequences and the code generated for each and demonstrates how the programmer specifies the operation they want and is given an efficient way to achieve it.

Table 5. A Selection of Shuffle Indexes and the Intel AVX Instruction Sequence Generated by the Compiler

| Shuffle Pattern Given to `__builtin_vector_shuffle` for 32-bit Index Elements | Purpose | Output from Clang 14.0.0 |
|---|---|---|
| 0, 0, 2, 2, 4, 4, 6, 6 | Duplicate even elements | `vmovsldup ymm0, ymm0` |
| 1, 2, 3, 0, 5, 6, 7, 4 | Rotate in lane | `vpermilps ymm0, ymm0, 57` |
| 3, 4, 7, 1, 2, 3, 4, 5 | Arbitrary permute | `vmovups ymm1, ymmword ptr [$idx]`<br>`vpermps ymm0, ymm1, ymm0` |

You should beware that the compiler is not perfect, and it sometimes misses opportunities that would provide faster code sequences. It is hoped that over time the compiler's understanding of what instruction sequences are fast will improve and any code written to use index sequences will get faster with future compilers. Also note that the Clang 14.0.0 and Intel oneAPI 2022.0 compilers at time of writing were more sophisticated than their contemporary GCC 11 and selected better permute instructions.

This section seems to have rendered the rest of the document obsolete. If the compiler can choose for itself which is the best instruction to use, what is the purpose of this document? There are several reasons, including:

- As already noted, the compiler is not perfect – there are some performance issues at time of writing, and falling back to intrinsics may still be necessary in places if the compiler is missing an opportunity.
- The programmer may want to be able to interpret the assembly generated by the compiler to understand why it has chosen a particular sequence for a given type of permutation.
- Understanding what can be done efficiently in the processor can help guide the programmer to use certain index sequences over others. The compiler is aiding the programmer by allowing a more obvious syntax to be used and avoiding the need to memorize intrinsic names, but the compiler is ultimately guided by the programmer to combine the permutes together in particular ways.
- The built-in shuffle syntax opens up the path to C++ compile-time permutation functions!

The last of these bullet points is interesting because it permits a way of programming in C++ that allows the index sequence to be specified at compile-time using a function rather than having to manually compute each index.

To illustrate why compile-time permute functions are interesting, consider what would be required to write a numerics library that needs to be able to transpose matrices of different data types and matrix dimensions. If conventional intrinsics are used, then the programmer needs to write many different versions of the function, parameterized for different types and sizes. In C++ it might be a little easier than in C due to the ability to use templates, but at some level the source code needs to know how to move elements around within a SIMD value using a named intrinsic. This explicit naming breaks portability since it ties the source code to a specific instruction, width, element size, and target. However, by using the `__builtin_vectorshuffle` function the compiler can be told to generate a certain permutation, and not only will it be applied portably to any data type and size, but it will do so efficiently without the programmer having to specify which intrinsic to use.

To aid us in building compile-time permute functions, consider the C++20[4] code fragment shown in Figure 35. This code fragment takes a pair of source SIMD values of any type and size (although they must be the same) and an index generation function that converts from an input index to an output index. The index generator is called repeatedly for all values between

[4] C++20 is required because this code uses template parameters on the lambda and the lambda is also written as an Immediately Invoked Function Expression (IIFE). This can be implemented in C++14 or 17 as well at the expense of more verbosity.

[0..N), where N is the size of the result vector. For each index it returns a new index representing the source index from which that index takes its value (e.g., given a function that does "`index * 2`" the sequential `iota' indexes [0, 1, 2, 3…] are mapped to the new index sequence [0, 2, 4, 6, …). Those new index values are then passed to the `__builtin_shufflevector` function to perform the actual permute.

```
template <typename _Result = void, typename _Vec, typename IndexGenerator>
constexpr auto permute(_Vec v0, _Vec v1, IndexGenerator fn) {
  // If the user explicitly specified an output type use that, otherwise assume it is
  // the same as an input SIMD.
  using _outVec = std::conditional_t<std::is_same_v<void, _Result>, _Vec, _Result>;
  constexpr auto _numElements = sizeof(_outVec) / sizeof(decltype(_outVec()[0]));

  return [=] <std::size_t... _Idx> (std::index_sequence<_Idx...>)
  {
      return __builtin_shufflevector(v0, v1, fn(_Idx)...);
  } (std::make_index_sequence<_numElements>{});
}
```

Figure 35. Source Code Showing a C++20 Code Fragment that uses a Compile-Time Lambda Function to Generate an Index Sequence for Use in the Compiler's Own Built-In Permute

Some examples of how to use this compile-time permute function are shown in Table 6.

Table 6. A Selection of Compile-Time Permute Function Calls and the Intel AVX Instruction Sequence Generated by the Compiler

| Permute Call | Purpose | Output from Clang 14.0.0[5] |
|---|---|---|
| `permute (x, x, [](auto idx) { return idx & ~1; });` | Duplicate even elements | `vmovsldup zmm0, zmm0` |
| `permute (x, x, [](auto idx) { return idx ^ 1; });` | Swap even/odd elements in each pair (complex-valued IQ swap) | `vpermilps ymm0, ymm0, 177` |
| `permute (x, x, [](auto idx) { return idx + 8; });` | Extract upper half of a 16-element vector. Note that the instruction sequence accepts a zmm input and returns a ymm output. | `vextractf64x4 ymm0, zmm0, 1` |
| `permute(x, y, [](size_t idx) { return idx % 2 ? idx / 2 + 16 : idx; });` | Insert a set of 8 contiguous values from one vector into the odd elements of a 16-element vector. | `vmovups zmm2, ptr [rip + .LC]`<br>`# zmm2 = [0,16,2,17,4,18,6,19,8,20,10,21,12,22,14,23]`<br>`vpermt2ps zmm0, zmm2, zmm1` |

Notice how any two vectors can be combined and even resized into a new vector, and the compiler efficiently determines how to do this for each new function it encounters. This compile-time permute instruction is extremely powerful and flexible and is very useful for building generic SIMD manipulation routines.

[5] https://godbolt.org/z/a5bhK7d6j



1022/DN/WIPRO/PDF 742407-001US