TECHNOLOGY GUIDE

Intel Corporation

Galois Field New Instructions (GFNI)

Authors

Daniel Towner

Ray Kinsella

1 Introduction

The Galois Field New Instructions (GFNI) introduced in 3rd Gen Intel® Xeon® Scalable processors were designed to accelerate cryptographic and security applications. With some imagination, one of the instructions – the affine transformation instruction – also can be used to perform a host of other useful operations on individual bits within byte elements. For example, the GFNI affine instruction can be used to reorder bits, implement 8-bit bit shift operations, and much more. The Intel® Advanced Vector Extensions (Intel® AVX) instruction set itself does not have native versions of many of these instructions that operate at byte granularity, so GFNI allows the instruction set to be extended to cover fast implementation of these functions.

In this document we give an overview of the GFNI affine transformation instruction, describe what it does in detail, and show how it can be used to implement a range of interesting functions for byte-element bit-bashing. Functions that could use such features can be found in many different applications, such as mobile-access signal processing, network packet processing, and media applications.

This document is intended for low-level software developers who are using intrinsics from the Intel® Advanced Vector Extensions 512 (Intel® AVX-512) family of instructions to implement highly optimized signal or packet-processing network applications. A basic background in understanding AVX-512 is useful. For more information about AVX-512, see the Reference Documentation.

This document is part of the Network Transformation Experience Kit, which is available at https://networkbuilders.intel.com/network-technologies/network-transformation-exp-kits.

Table of Contents

1 Introduction 1
1.1 Terminology 3
1.2 Reference Documentation 3

2 Introduction to GFNI Bit-Matrix Products 3
2.1 Brief Introduction to GF(2) 3
2.2 GF(2) Matrix-Vector Bit Products 3
2.3 Using the Matrix Product to Manipulate Individual Bits within a Byte 5
2.4 Understanding the Affine Transformation Operand Encodings 6
2.5 Computing and Using the Transformation Matrix 7

3 Cookbook of GFNI Byte-Level Bit Manipulations 8

4 Summary 10

Figures

Figure 1. Multiplying an Arbitrary Bit-Vector by the Identity Matrix 4
Figure 2. Method for Generating a Single Output Bit Using a GF(2) Dot-Product 4
Figure 3. Matrix Product Using Horizontally Flipped Identity Matrix to Perform Bit-Reversal 4
Figure 4. Using the Immediate Value to Invert the Matrix Product Output 5
Figure 5. 5-Bit Sign-Extension Operation 5
Figure 6. Ways to Transform Input Byte 6
Figure 7. Application of Single 8x8 Transformation Matrix to Eight Individual Bytes 6
Figure 8. Transformation Matrix and Incoming Bytes Packed into AVX Registers 7
Figure 9. 5-Bit Sign-Extension Example 7

Tables

Table 1. Terminology 3
Table 2. Reference Documents 3
Table 3. Bit-Width Variants of Affine Transformation Instruction 7
Table 4. Bit Rotate 8
Table 5. Logical Shift Left 8
Table 6. Arithmetic Right Shift 9
Table 7. Extract Signed Value from Bit-Field 9
Table 8. Bit Reverse 9
Table 9. Arbitrary Bit Reorder 10

Document Revision History

REVISION	DATE	DESCRIPTION
001	July 2021	Initial release.
002	December 2021	Minor update to Section 2.5 and Figure 9.

1.1 Terminology

Table 1. Terminology

ABBREVIATION	DESCRIPTION
AVX	Advanced Vector Extensions
CRC	Cyclic Redundancy Check
GFNI	Galois Field New Instructions
MSB	Most Significant Bit

1.2 Reference Documentation

Table 2. Reference Documents

REFERENCE	SOURCE
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® Xeon® Processor Scalable Family Specification Update	https://www.intel.in/content/www/in/en/processors/xeon/xeon-technical-resources.html
2nd Gen Intel® Xeon® Scalable Processors Specification Update	https://www.intel.com/content/www/us/en/products/docs/processors/xeon/2nd-gen-xeon-scalable-spec-update.html
New 3rd Gen Intel® Xeon® Scalable Processor	https://hotchips.org/assets/program/conference/day1/HotChips2020_Server_Processors_Intel_Irma_ICX-CPU-final3.pdf
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 - 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 - 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

2 Introduction to GFNI Bit-Matrix Products

The Galois Field New Instructions ISA (GFNI) has several instructions, but the one of most interesting for the bit-bashing requirements of network processing and DSP functions is the affine transformation instruction. Although originally intended for cryptographic work, its operation as a way of performing bit-matrix products allows it to be used for many different bit reordering operations. In this section we introduce how matrix bit-products work and how they may be used to copy, reorder, and invert the bits within a byte.

2.1 Brief Introduction to GF(2)

A knowledge of Galois Fields is not necessary to understand the basic operation of the instruction that is described in this paper, nor to understand the applications of this instruction to bit-processing. However, a brief background to Galois Fields may be of interest.

A Galois Field is a field containing a finite number of elements. As with other fields, a Galois Field has well defined elements, and operations for addition, subtraction, and so on, when applied to those elements generates a result in the same field. The GF(2) field is simply a Galois Field with only two elements – 0 and 1. The addition of two values in GF(2) is equivalent to addition-modulo-2, or an exclusive-OR. The multiplication of two values in GF(2) is equivalent to multiplication-modulo-2, or the logical AND operation. Other operations are similarly defined.

2.2 GF(2) Matrix-Vector Bit Products

The GFNI affine instruction can best be described as the multiplication of an 8x8 bit matrix, with a single 8-bit byte column vector, in the GF(2) field. The figure below shows the effect of multiplying an arbitrary bit-vector by the identity matrix:

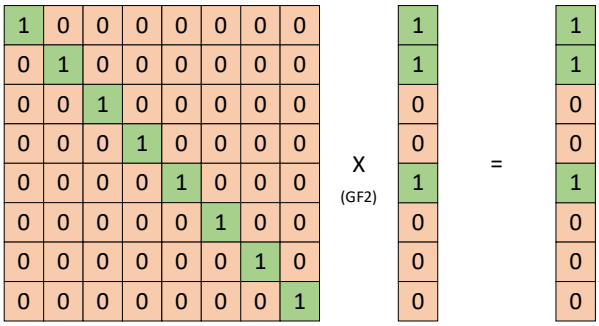


Figure 1. Multiplying an Arbitrary Bit-Vector by the Identity Matrix

Any product of a matrix and a vector can be decomposed into a series of dot products, and to understand how the above multiplication works in GF(2) it is useful to do this here too. Consider how we derive the second bit, for example, using the following figure.

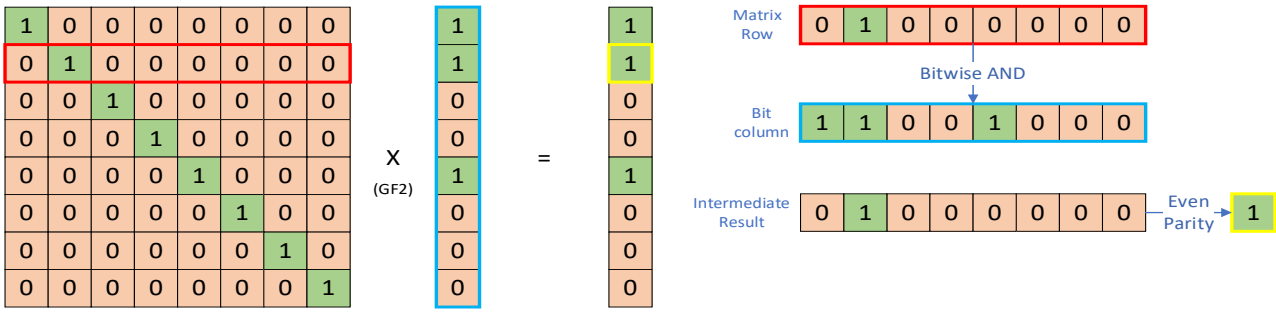


Figure 2. Method for Generating a Single Output Bit Using a GF(2) Dot-Product

The yellow output bit is the result of taking a GF(2) dot-product between the second row of the input matrix and the bit vector. A GF(2) dot-product can be implemented using the sequence of steps shown on the right of the figure. Firstly, the two bit-vectors - the row and the column are multiplied together, which in GF(2) is equivalent to a bitwise AND of the values to form an intermediate result. Those intermediate bits are then reduced to a single output bit, using a GF(2) addition operation, which is equivalent to applying an exclusive-OR (XOR) across the bits. This could also be thought of as an even-parity operation, where either a 0 or 1 is generated as an output, such that the total numbers in the intermediate result becomes even. Multiplying a bit vector by the identity matrix served to illustrate the mechanics of what the GFNI instruction does but is uninteresting. A more interesting example is shown in the figure below, where the input transformation matrix has been flipped horizontally. The effect of this transformation matrix is to reverse the order of the bits in the output.

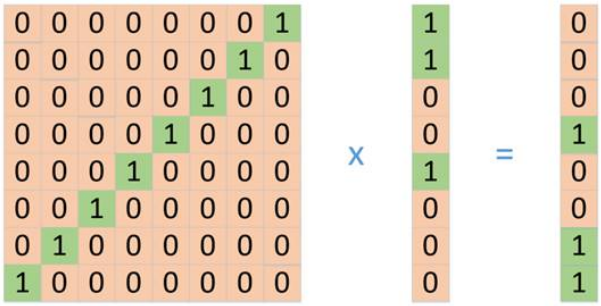


Figure 3. Matrix Product Using Horizontally Flipped Identity Matrix to Perform Bit-Reversal

This GFNI transformation is immediately useful, as there is no other easy way to achieve a bit-reversal operation in the current AVX family of instruction sets.

There remains one final twist to the GFNI affine instruction that can prove very useful. In addition to doing the bit-matrix-product, the output of the matrix-product and even-parity function can be XORed with an 8-bit immediate, as shown in the following diagram.

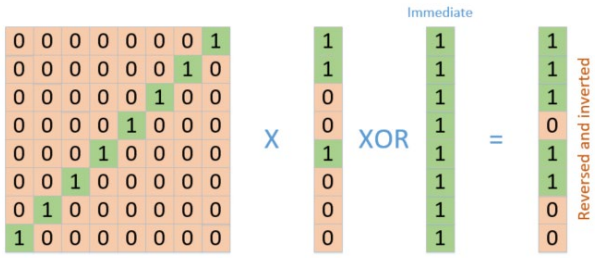


Figure 4. Using the Immediate Value to Invert the Matrix Product Output

Notice how in this example the left matrix again performs a bit-reverse operation on the incoming byte, but the XOR by the immediate of all ones then inverts each bit. Again, a bit-reverse and invert instruction does not exist in AVX, so this is another useful addition.

Note that each row or column of the left bit-matrix can contain more than one bit, which is useful for operations such as CRC calculations and cryptographic operations. Using multiple bits per row is beyond the scope of this white paper, which concentrates on using the GFNI affine transformation instruction for bit ordering extensions. The use of the GFNI instruction set for those alternative applications will be covered in separate white papers.

2.3 Using the Matrix Product to Manipulate Individual Bits within a Byte

The combination of the matrix-product for reordering and copying bits, along with the immediate to invert individual output bits, gives an incredibly powerful base from which to perform many types of bit-bashing within a byte element. For example, consider the following GFNI matrix product, which implements a 5-bit sign-extension operation.

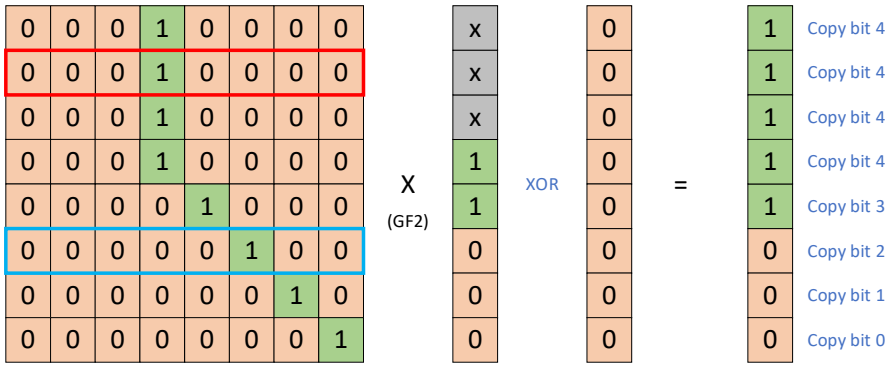


Figure 5. 5-Bit Sign-Extension Operation

In the top part of the matrix, as illustrated by the red boundary, the transformation matrix is arranged so that the sign-bit of the incoming byte is copied into all the top bits of the output, effecting the sign-extension. The lower half of the transformation matrix, as exemplified by the blue boundary area, is copying the bits through to the output unaltered. This 5-bit sign-extension is easily done using GFNI affine instruction, but in previous AVX instruction sets would have required a series of shift-operations to effect the same result.

The transformation matrix in the figure below shows all the different possible ways to transform the input byte.

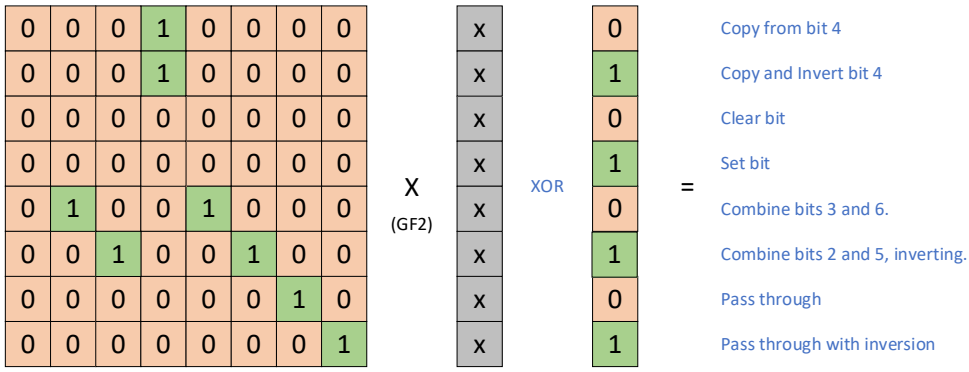


Figure 6. Ways to Transform Input Byte

In particular:

- The first two rows contain a single 1 bit and effectively copy whatever the input bit is at the position of the 1 to the output, either passing it through unaltered or inverting it. Any input bit can be copied/inverted to any output bit position.
- Rows 3 and 4 have no input bits in the transformation matrix, and so their output bits are whatever the bit-value of the corresponding immediate bit is. This allows an explicit set or clear of any output bit position.
- The remaining rows illustrate other possibilities that could be useful for CRC or cryptographic functions but are not covered further in this paper.

Notice how we can arbitrarily move, duplicate, reorder, invert, clear, and set bits from any position in a bit position to any other position. From this foundation, we can build a wealth of new byte-oriented bit-manipulation instructions.

2.4 Understanding the Affine Transformation Operand Encodings

In our examples above we have assumed that a single 8x8 bit matrix is applied to a single-byte operand. In reality, the GFNI instruction set works within the AVX framework, where it can apply a single bit-matrix transformation to as many as eight individual bytes and perform several such operations in parallel in different parts of the AVX register.

To begin with, consider how a single 8x8 transformation matrix could be applied to eight individual bytes, as follows.

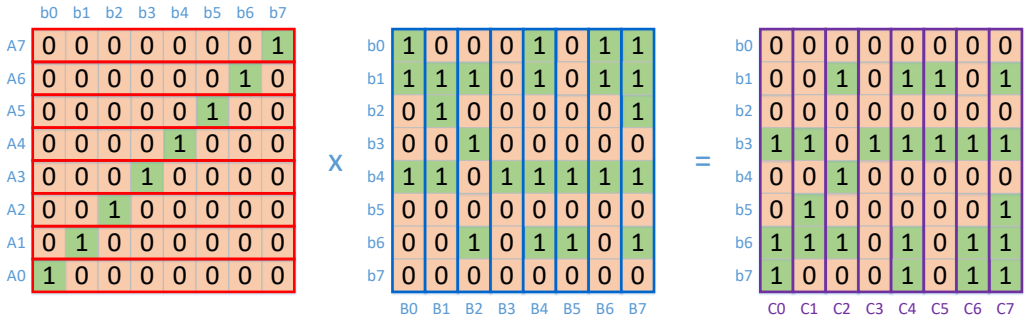


Figure 7. Application of Single 8x8 Transformation Matrix to Eight Individual Bytes

In this example, note how the left transformation matrix is applying a bit-reversal operation. On the right side of the new matrix-matrix product, each of the eight vectors represents a different incoming byte, and the output consists of eight new bytes, each of which has undergone a bit-reversal operation.

Note also that the left and right matrices are grouped differently. On the left, each incoming byte forms a row, and on the right, each byte forms a column. This feature allows us to transpose a bit-matrix, or to gather numbered bits from a set of bytes into a single byte. We shall explore this feature further later in this paper.

Given that the transformation matrix and the incoming bytes are both grouped into 64-bit chunks of bits, we now need to look at how these are packed into AVX registers.

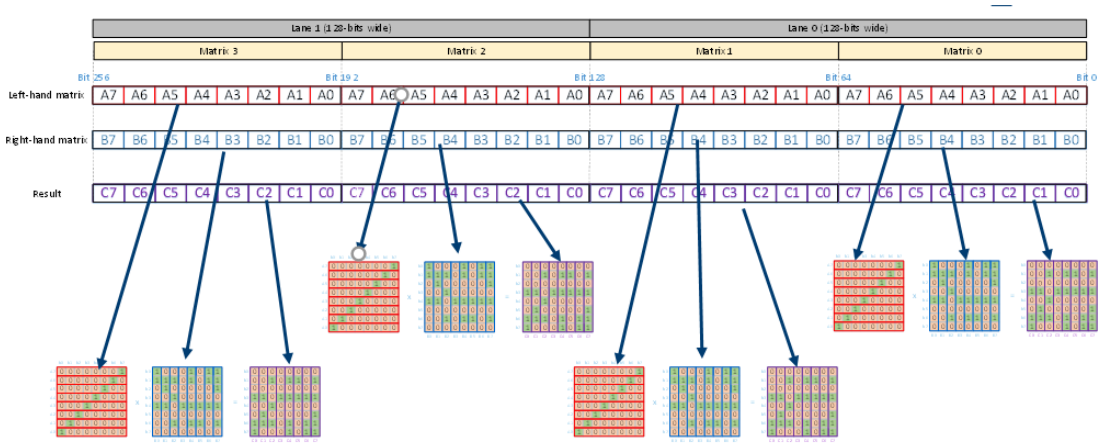


Figure 8. Transformation Matrix and Incoming Bytes Packed into AVX Registers

In this example, we are processing data in 256-bit registers. Each 256-bit register can contain 4x64-bit transformation and operand matrices. Therefore, the instruction enables each 64-bit group of bits to have a different transformation applied to it.

Table 3. Bit-Width Variants of Affine Transformation Instruction

C INTRINSIC	BIT WIDTH
__m128i _mm_gf2p8affine_epi64_epi8 (__m128i x, __m128i A, int b)	128
__m256i _mm256_gf2p8affine_epi64_epi8 (__m256i x, __m256i A, int b)	256
__m512i _mm512_gf2p8affine_epi64_epi8 (__m512i x, __m512i A, int b)	512

2.5 Computing and Using the Transformation Matrix

Remember from earlier that a transformation matrix is an 8x8 bit matrix, organized into multiple rows. For example, our 5-bit sign-extension example from earlier looked like this.

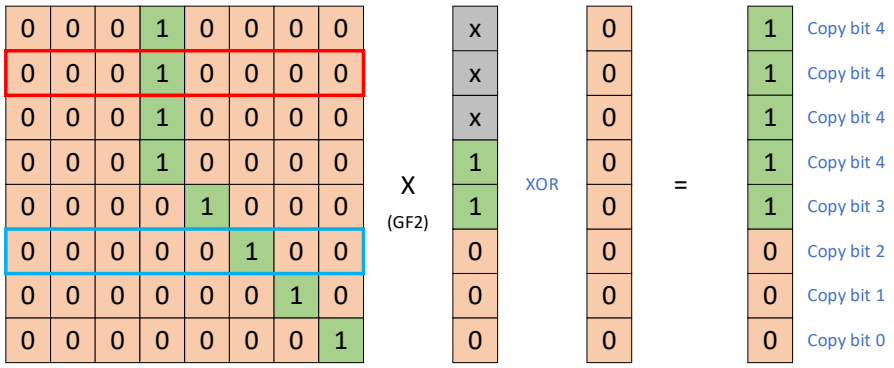


Figure 9. 5-Bit Sign-Extension Example

The transformation matrix on the left comprises eight rows of bytes. To encode the matrix, we first encode each row in turn to form bytes, and then we pack those bytes together, starting with row 0. Our example above would therefore be represented by the following eight bytes: 0x10, 0x10, 0x10, 0f10, 0x08, 0x4, 0x2, 0x1

The complete transformation byte could therefore be encoded as the 64-bit integer: 0x0102040810101010. In this example, the immediate is zero as no output bits are inverted.

To use this to manipulate some bytes, use the following code sequence.

```
const uint64_t k_signExtend5 = 0x0102040810101010;
const __m128i k_matrix = _mm_set1_epi64(k_signExtend5);
const auto result = _mm_gf2p8affine_epi64_epi8 (dataToTransform, k_matrix, 0);
```

This code sequence first initializes a 64-bit integer with the transformation matrix. It then broadcasts it to the complete contents of a 128-bit register (i.e., two copies, in the upper and lower halves). The final line loads a block of data to transform, comprising 16 individual byte values, and applies the 5-big sign extension transformation to each of those 16 bytes to generate the result.

3 Cookbook of GFNI Byte-Level Bit Manipulations

In this section we shall outline some of the byte transformations that are made possible by using the GFNI affine instruction. In each case we use the following abilities:

Clear	Explicitly clear a given bit position to 0
Set	Explicitly set a given bit position to 1
Copy(n)	Copy a bit from the given position index
Invert(n)	Copy and invert a bit from the given position index

These basic bit primitives are sufficient to handle all the following byte-level manipulations, none of which exist as real instructions in any of the AVX-family of instruction sets.

Note that it is most likely that the transformations described below will be static. Programmers are likely to know in advance if they want to extract a signed bit-field from a specific location, for example. Some of the transformations could be dynamic though, such as the bit reorder. In that case, the transformation matrix would need to be generated dynamically.

Table 4. Bit Rotate

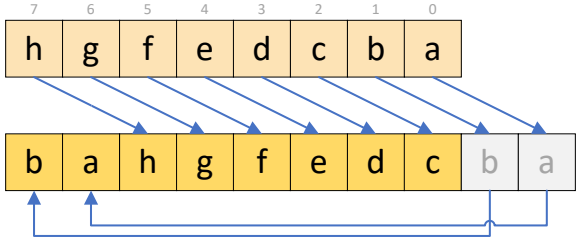
ITEM	DESCRIPTION
Description	Bits are moved left or right within the byte, with any that fall off either end of the byte rotated back to the other end.
Example	2-bit right rotate 
Bit operations (MSB first)	Copy(1), Copy(0), Copy(7), Copy(6), Copy(5), Copy(4), Copy(3), Copy(2)

Table 5. Logical Shift Left

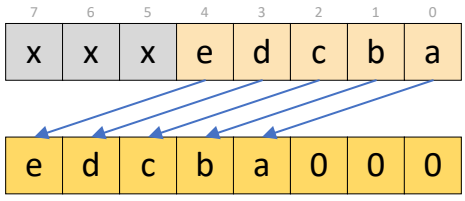
ITEM	DESCRIPTION
Description	Bits are moved to the left, and any empty positions at the bottom of the byte are filled with zeros. The bits at the top of the byte are lost.
Example	3-bit left shift 
Bit operations (MSB first)	Copy(4), Copy(3), Copy(2), Copy(1), Copy(0), Clear, Clear, Clear

Table 6. Arithmetic Right Shift

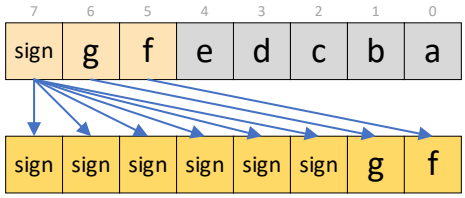
ITEM	DESCRIPTION
Description	Bits are moved to the left, with copies of the sign-bit added to provide extra bits. The lowest bits are lost.
Example	5-bit arithmetic right shift 
Bit operations (MSB first)	Copy(7), Copy(7), Copy(7), Copy(7), Copy(7), Copy(7), Copy(6), Copy(5)

Table 7. Extract Signed Value from Bit-Field

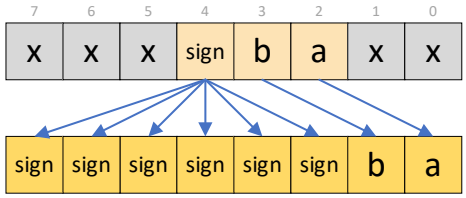
ITEM	DESCRIPTION
Description	Extract a subset of bits from a packed bit-field, and sign extend those bits to the full byte. All other bits are discarded.
Example	Extract bits [2..5) as a signed value 
Bit operations (MSB first)	Copy(4), Copy(4), Copy(4), Copy(4), Copy(4), Copy(4), Copy(3), Copy(2)

Table 8. Bit Reverse

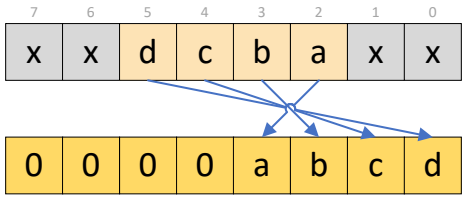
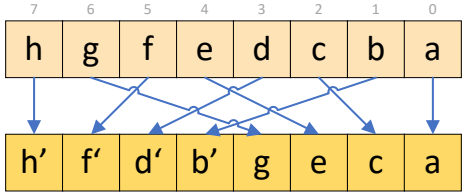
ITEM	DESCRIPTION
Description	Reorder bits so that they appear in the opposite order. Either the entire byte, or smaller subgroups, can be reordered.
Example	Extract the subgroup of bits from [2..6) and reverse them, before writing back to the lowest position of the output byte 
Bit operations (MSB first)	Clear, Clear, Clear, Clear, Copy(2), Copy(3), Copy(4), Copy(5)

Table 9. Arbitrary Bit Reorder

ITEM	DESCRIPTION
Description	Reorder bits so that they appear in the opposite order. Either the entire byte, or smaller subgroups, can be reordered.
Example	Read the four even bits and store in the lower four bits of the output. Read the four odd bits and store in the upper four bits of the output but invert them. 
Bit operations (MSB first)	Invert(7), Invert(6), Invert(5), Invert(4), Copy(3), Copy(2), Copy(1), Copy(0)

The examples above give just a few ideas about what is possible with this bit-bashing instruction, and many of them allow holes in the existing AVX-family of instructions sets to be filled (e.g., there are no byte granularity shift or rotate instructions, or bit-reversal instructions). The ability of the instruction to reorder, copy, invert, set, and clear arbitrarily across the byte opens many other possibilities to the imagination!

4 Summary

In this document we have described how the Galois Field New Instruction set contains a relatively unknown but extremely useful instruction called an affine transformation. This instruction allows the bits within a byte element of a SIMD register to be arbitrarily reordered, copied, inverted, cleared, and set. By using this bit-level primitive, it is possible to construct many byte-granularity instructions that do not exist in the AVX-family of instruction sets, including rotate, shift, bit-reversal, and reorder. In previous instruction sets, these operations had to be synthesized from potentially long sequences of other instructions, but they can now be implemented using a single instruction. We have given some examples of how to use the new instruction, but also note that many imaginative ways of exploiting this instruction can also be found. For details on how to implement this instruction, see the Intel® 64 and IA-32 Architectures Software Developer Manuals.

We have not covered the use of the GFNI instructions for building CRC, hashing, and cryptographic algorithms. Intel plans future white papers covering these topics.

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

Results have been estimated or simulated.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Intel technologies may require enabled hardware, software or service activation.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

1221/DN/WIPRO/PDF 644497-002US