## GF2P8AFFINEINVQB — Galois Field Affine Transformation Inverse

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| 66 0F3A CF /r /ib GF2P8AFFINEINVQB xmm1, xmm2/m128, imm8 | A | V/V | GFNI | Computes inverse affine transformation in the finite field GF(2^8). |
| VEX.NDS.128.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB xmm1, xmm2, xmm3/m128, imm8 | B | V/V | AVX GFNI | Computes inverse affine transformation in the finite field GF(2^8). |
| VEX.NDS.256.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB ymm1, ymm2, ymm3/m256, imm8 | B | V/V | AVX GFNI | Computes inverse affine transformation in the finite field GF(2^8). |
| EVEX.NDS.128.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8 | C | V/V | AVX512VL GFNI | Computes inverse affine transformation in the finite field GF(2^8). |
| EVEX.NDS.256.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8 | C | V/V | AVX512VL GFNI | Computes inverse affine transformation in the finite field GF(2^8). |
| EVEX.NDS.512.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8 | C | V/V | AVX512F GFNI | Computes inverse affine transformation in the finite field GF(2^8). |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | NA | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |
| B | NA | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |
| C | Full | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |

### Description

The AFFINEINVB instruction computes an affine transformation in the Galois Field $2^8$. For this instruction, an affine transformation is defined by A * inv(x) + b where “A” is an 8 by 8 bit matrix, and “x” and “b” are 8-bit vectors. The inverse of the bytes in x is defined with respect to the reduction polynomial $x^8 + x^4 + x^3 + x + 1$.

One SIMD register (operand 1) holds “x” as either 16, 32 or 64 8-bit vectors. A second SIMD (operand 2) register or memory operand contains 2, 4, or 8 “A” values, which are operated upon by the correspondingly aligned 8 “x” values in the first register. The “b” vector is constant for all calculations and contained in the immediate byte.

The EVEX encoded form of this instruction does not support memory fault suppression. The SSE encoded forms of the instruction require 16B alignment on their memory operations.

The inverse of each byte is given by the following table. The upper nibble is on the vertical axis and the lower nibble is on the horizontal axis. For example, the inverse of 0x95 is 0x8A.

**Table 2-1. Inverse Byte Listings**

| - | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 8D | F6 | CB | 52 | 7B | D1 | E8 | 4F | 29 | C0 | B0 | E1 | E5 | C7 |
| 1 | 74 | B4 | AA | 4B | 99 | 2B | 60 | 5F | 58 | 3F | FD | CC | FF | 40 | EE | B2 |
| 2 | 3A | 6E | 5A | F1 | 55 | 4D | A8 | C9 | C1 | A | 98 | 15 | 30 | 44 | A2 | C2 |
| 3 | 2C | 45 | 92 | 6C | F3 | 39 | 66 | 42 | F2 | 35 | 20 | 6F | 77 | BB | 59 | 19 |
| 4 | 1D | FE | 37 | 67 | 2D | 31 | F5 | 69 | A7 | 64 | AB | 13 | 54 | 25 | E9 | 9 |
| 5 | ED | 5C | 5 | CA | 4C | 24 | 87 | BF | 18 | 3E | 22 | F0 | 51 | EC | 61 | 17 |
| 6 | 16 | 5E | AF | D3 | 49 | A6 | 36 | 43 | F4 | 47 | 91 | DF | 33 | 93 | 21 | 3B |
| 7 | 79 | B7 | 97 | 85 | 10 | B5 | BA | 3C | B6 | 70 | D0 | 6 | A1 | FA | 81 | 82 |
| 8 | 83 | 7E | 7F | 80 | 96 | 73 | BE | 56 | 9B | 9E | 95 | D9 | F7 | 2 | B9 | A4 |
| 9 | DE | 6A | 32 | 6D | D8 | 8A | 84 | 72 | 2A | 14 | 9F | 88 | F9 | DC | 89 | 9A |
| A | FB | 7C | 2E | C3 | 8F | B8 | 65 | 48 | 26 | C8 | 12 | 4A | CE | E7 | D2 | 62 |
| B | C | E0 | 1F | EF | 11 | 75 | 78 | 71 | A5 | 8E | 76 | 3D | BD | BC | 86 | 57 |
| C | B | 28 | 2F | A3 | DA | D4 | E4 | F | A9 | 27 | 53 | 4 | 1B | FC | AC | E6 |
| D | 7A | 7 | AE | 63 | C5 | DB | E2 | EA | 94 | 8B | C4 | D5 | 9D | F8 | 90 | 6B |
| E | B1 | D | D6 | EB | C6 | E | CF | AD | 8 | 4E | D7 | E3 | 5D | 50 | 1E | B3 |
| F | 5B | 23 | 38 | 34 | 68 | 46 | 3 | 8C | DD | 9C | 7D | A0 | CD | 1A | 41 | 1C |

### Operation

```
define affine_inverse_byte(tsrc2qw, src1byte, imm):
    FOR i ← 0 to 7:
        * parity(x) = 1 if x has an odd number of 1s in it, and 0 otherwise.*
        * inverse(x) is defined in the table above *
        retbyte.bit[i] ← parity(tsrc2qw.byte[7-i] AND inverse(src1byte)) XOR imm8.bit[i]
    return retbyte
```

**VGF2P8AFFINEINVQB dest, src1, src2, imm8 (EVEX encoded version)**

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1:
    IF SRC2 is memory and EVEX.b==1:
        tsrc2 ← SRC2.qword[0]
    ELSE:
        tsrc2 ← SRC2.qword[j]

FOR b ← 0 to 7:
    IF k1[j*8+b] OR *no writemask*:
        FOR i ← 0 to 7:
            DEST.qword[j].byte[b] ← affine_inverse_byte(tsrc2, SRC1.qword[j].byte[b], imm8)
    ELSE IF *zeroing*:
        DEST.qword[j].byte[b] ← 0
    *ELSE DEST.qword[j].byte[b] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

**VGF2P8AFFINEINVQB dest, src1, src2, imm8 (128b and 256b VEX encoded versions)**

```
(KL, VL) = (2, 128), (4, 256)
FOR j ← 0 TO KL-1:
    FOR b ← 0 to 7:
        DEST.qword[j].byte[b] ← affine_inverse_byte(SRC2.qword[j], SRC1.qword[j].byte[b], imm8)
DEST[MAX_VL-1:VL] ← 0
```

**GF2P8AFFINEINVQB srcdest, src1, imm8 (128b SSE encoded version)**

```
FOR j ← 0 TO 1:
    FOR b ← 0 to 7:
        SRCDEST.qword[j].byte[b] ← affine_inverse_byte(SRC1.qword[j], SRCDEST.qword[j].byte[b], imm8)
```

### Intel C/C++ Compiler Intrinsic Equivalent

GF2P8AFFINEINVQB __m128i _mm_gf2p8affineinv_epi64_epi8(__m128i, __m128i, int);
GF2P8AFFINEINVQB __m128i _mm_mask_gf2p8affineinv_epi64_epi8(__m128i, __mmask16, __m128i, __m128i, int);
GF2P8AFFINEINVQB __m128i _mm_maskz_gf2p8affineinv_epi64_epi8(__mmask16, __m128i, __m128i, int);
GF2P8AFFINEINVQB __m256i _mm256_gf2p8affineinv_epi64_epi8(__m256i, __m256i, int);
GF2P8AFFINEINVQB __m256i _mm256_mask_gf2p8affineinv_epi64_epi8(__m256i, __mmask32, __m256i, __m256i, int);
GF2P8AFFINEINVQB __m256i _mm256_maskz_gf2p8affineinv_epi64_epi8(__mmask32, __m256i, __m256i, int);
GF2P8AFFINEINVQB __m512i _mm512_gf2p8affineinv_epi64_epi8(__m512i, __m512i, int);
GF2P8AFFINEINVQB __m512i _mm512_mask_gf2p8affineinv_epi64_epi8(__m512i, __mmask64, __m512i, __m512i, int);
GF2P8AFFINEINVQB __m512i _mm512_maskz_gf2p8affineinv_epi64_epi8(__mmask64, __m512i, __m512i, int);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

Legacy-encoded and VEX-encoded: Exceptions Type 4.

EVEX-encoded: See Exceptions Type E4NF.

## GF2P8AFFINEQB — Galois Field Affine Transformation

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| 66 0F3A CE /r /ib GF2P8AFFINEQB xmm1, xmm2/m128, imm8 | A | V/V | GFNI | Computes affine transformation in the finite field GF(2^8). |
| VEX.NDS.128.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB xmm1, xmm2, xmm3/m128, imm8 | B | V/V | AVX GFNI | Computes affine transformation in the finite field GF(2^8). |
| VEX.NDS.256.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB ymm1, ymm2, ymm3/m256, imm8 | B | V/V | AVX GFNI | Computes affine transformation in the finite field GF(2^8). |
| EVEX.NDS.128.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8 | C | V/V | AVX512VL GFNI | Computes affine transformation in the finite field GF(2^8). |
| EVEX.NDS.256.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8 | C | V/V | AVX512VL GFNI | Computes affine transformation in the finite field GF(2^8). |
| EVEX.NDS.512.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8 | C | V/V | AVX512F GFNI | Computes affine transformation in the finite field GF(2^8). |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | NA | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |
| B | NA | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |
| C | Full | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |

### Description

The AFFINEB instruction computes an affine transformation in the Galois Field $2^8$. For this instruction, an affine transformation is defined by A * x + b where “A” is an 8 by 8 bit matrix, and “x” and “b” are 8-bit vectors. One SIMD register (operand 1) holds “x” as either 16, 32 or 64 8-bit vectors. A second SIMD (operand 2) register or memory operand contains 2, 4, or 8 “A” values, which are operated upon by the correspondingly aligned 8 “x” values in the first register. The “b” vector is constant for all calculations and contained in the immediate byte.

The EVEX encoded form of this instruction does not support memory fault suppression. The SSE encoded forms of the instruction require16B alignment on their memory operations.

### Operation

```
define parity(x):
    t ← 0                    // single bit
    FOR i ← 0 to 7:
        t = t xor x.bit[i]
    return t

define affine_byte(tsrc2qw, src1byte, imm):
    FOR i ← 0 to 7:
        * parity(x) = 1 if x has an odd number of 1s in it, and 0 otherwise.*
        retbyte.bit[i] ← parity(tsrc2qw.byte[7-i] AND src1byte) XOR imm8.bit[i]
    return retbyte
```

**VGF2P8AFFINEQB dest, src1, src2, imm8 (EVEX encoded version)**

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1:
    IF SRC2 is memory and EVEX.b==1:
        tsrc2 ← SRC2.qword[0]
    ELSE:
        tsrc2 ← SRC2.qword[j]

    FOR b ← 0 to 7:
        IF k1[j*8+b] OR *no writemask*:
            DEST.qword[j].byte[b] ← affine_byte(tsrc2, SRC1.qword[j].byte[b], imm8)
        ELSE IF *zeroing*:
            DEST.qword[j].byte[b] ← 0
        *ELSE DEST.qword[j].byte[b] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

**VGF2P8AFFINEQB dest, src1, src2, imm8 (128b and 256b VEX encoded versions)**

```
(KL, VL) = (2, 128), (4, 256)
FOR j ← 0 TO KL-1:
    FOR b ← 0 to 7:
        DEST.qword[j].byte[b] ← affine_byte(SRC2.qword[j], SRC1.qword[j].byte[b], imm8)
DEST[MAX_VL-1:VL] ← 0
```

**GF2P8AFFINEQB srcdest, src1, imm8 (128b SSE encoded version)**

```
FOR j ← 0 TO 1:
    FOR b ← 0 to 7:
        SRCDEST.qword[j].byte[b] ← affine_byte(SRC1.qword[j], SRCDEST.qword[j].byte[b], imm8)
```

### Intel C/C++ Compiler Intrinsic Equivalent

GF2P8AFFINEQB __m128i _mm_gf2p8affine_epi64_epi8(__m128i, __m128i, int);
GF2P8AFFINEQB __m128i _mm_mask_gf2p8affine_epi64_epi8(__m128i, __mmask16, __m128i, __m128i, int);
GF2P8AFFINEQB __m128i _mm_maskz_gf2p8affine_epi64_epi8(__mmask16, __m128i, __m128i, int);
GF2P8AFFINEQB __m256i _mm256_gf2p8affine_epi64_epi8(__m256i, __m256i, int);
GF2P8AFFINEQB __m256i _mm256_mask_gf2p8affine_epi64_epi8(__m256i, __mmask32, __m256i, __m256i, int);
GF2P8AFFINEQB __m256i _mm256_maskz_gf2p8affine_epi64_epi8(__mmask32, __m256i, __m256i, int);
GF2P8AFFINEQB __m512i _mm512_gf2p8affine_epi64_epi8(__m512i, __m512i, int);
GF2P8AFFINEQB __m512i _mm512_mask_gf2p8affine_epi64_epi8(__m512i, __mmask64, __m512i, __m512i, int);
GF2P8AFFINEQB __m512i _mm512_maskz_gf2p8affine_epi64_epi8(__mmask64, __m512i, __m512i, int);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

Legacy-encoded and VEX-encoded: Exceptions Type 4.
EVEX-encoded: See Exceptions Type E4NF.

## GF2P8MULB — Galois Field Multiply Bytes

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| 66 0F38 CF /r<br>GF2P8MULB xmm1, xmm2/m128 | A | V/V | GFNI | Multiplies elements in the finite field GF(2^8). |
| VEX.NDS.128.66.0F38.W0 CF /r<br>VGF2P8MULB xmm1, xmm2, xmm3/m128 | B | V/V | AVX<br>GFNI | Multiplies elements in the finite field GF(2^8). |
| VEX.NDS.256.66.0F38.W0 CF /r<br>VGF2P8MULB ymm1, ymm2, ymm3/m256 | B | V/V | AVX<br>GFNI | Multiplies elements in the finite field GF(2^8). |
| EVEX.NDS.128.66.0F38.W0 CF /r<br>VGF2P8MULB xmm1{k1}{z}, xmm2, xmm3/m128 | C | V/V | AVX512VL<br>GFNI | Multiplies elements in the finite field GF(2^8). |
| EVEX.NDS.256.66.0F38.W0 CF /r<br>VGF2P8MULB ymm1{k1}{z}, ymm2, ymm3/m256 | C | V/V | AVX512VL<br>GFNI | Multiplies elements in the finite field GF(2^8). |
| EVEX.NDS.512.66.0F38.W0 CF /r<br>VGF2P8MULB zmm1{k1}{z}, zmm2, zmm3/m512 | C | V/V | AVX512F<br>GFNI | Multiplies elements in the finite field GF(2^8). |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | NA | ModRM:reg (r, w) | ModRM:r/m (r) | NA | NA |
| B | NA | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |
| C | Full Mem | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |

### Description

The instruction multiplies elements in the finite field GF($2^8$), operating on a byte (field element) in the first source operand and the corresponding byte in a second source operand. The field GF($2^8$) is represented in polynomial representation with the reduction polynomial $x^8 + x^4 + x^3 + x + 1$.

This instruction does not support broadcasting.

The EVEX encoded form of this instruction supports memory fault suppression. The SSE encoded forms of the instruction require16B alignment on their memory operations.

### Operation

```
define gf2p8mul_byte(src1byte, src2byte):
    tword ← 0
    FOR i ← 0 to 7:
        IF src2byte.bit[i]:
            tword ← tword XOR (src1byte<< i)
        * carry out polynomial reduction by the characteristic polynomial p*
    FOR i ← 14 downto 8:
        p ← 0x11B << (i-8)          *0x11B = 0000_0001_0001_1011 in binary*
        IF tword.bit[i]:
            tword ← tword XOR p
return tword.byte[0]
```

**VGF2P8MULB dest, src1, src2 (EVEX encoded version)**

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[j] ← gf2p8mul_byte(SRC1.byte[j], SRC2.byte[j])
    ELSE iF *zeroing*:
        DEST.byte[j] ← 0
    * ELSE DEST.byte[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

**VGF2P8MULB dest, src1, src2 (128b and 256b VEX encoded versions)**

```
(KL, VL) = (16, 128), (32, 256)
FOR j ← 0 TO KL-1:
    DEST.byte[j] ← gf2p8mul_byte(SRC1.byte[j], SRC2.byte[j])
DEST[MAX_VL-1:VL] ← 0
```

**GF2P8MULB srcdest, src1 (128b SSE encoded version)**

```
FOR j ← 0 TO 15:
    SRCDEST.byte[j] ← gf2p8mul_byte(SRCDEST.byte[j], SRC1.byte[j])
```

### Intel C/C++ Compiler Intrinsic Equivalent

VGF2P8MULB __m128i _mm_gf2p8mul_epi8(__m128i, __m128i);
VGF2P8MULB __m128i _mm_mask_gf2p8mul_epi8(__m128i, __mmask16, __m128i, __m128i);
VGF2P8MULB __m128i _mm_maskz_gf2p8mul_epi8(__mmask16, __m128i, __m128i);
VGF2P8MULB __m256i _mm256_gf2p8mul_epi8(__m256i, __m256i);
VGF2P8MULB __m256i _mm256_mask_gf2p8mul_epi8(__m256i, __mmask32, __m256i, __m256i);
VGF2P8MULB __m256i _mm256_maskz_gf2p8mul_epi8(__mmask32, __m256i, __m256i);
VGF2P8MULB __m512i _mm512_gf2p8mul_epi8(__m512i, __m512i);
VGF2P8MULB __m512i _mm512_mask_gf2p8mul_epi8(__m512i, __mmask64, __m512i, __m512i);
VGF2P8MULB __m512i _mm512_maskz_gf2p8mul_epi8(__mmask64, __m512i, __m512i);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

Legacy-encoded and VEX-encoded: Exceptions Type 4.

EVEX-encoded: See Exceptions Type E4NF.

# VAESDEC — Perform One Round of an AES Decryption Flow

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| VEX.NDS.256.66.0F38.WIG DE /r VAESDEC ymm1, ymm2, ymm3/m256 | A | V/V | VAES | Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from ymm2 with a 128-bit round key from ymm3/m256; store the result in ymm1. |
| EVEX.NDS.128.66.0F38.WIG DE /r VAESDEC xmm1, xmm2, xmm3/m128 | B | V/V | AVX512VL VAES | Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from xmm3/m128; store the result in xmm1. |
| EVEX.NDS.256.66.0F38.WIG DE /r VAESDEC ymm1, ymm2, ymm3/m256 | B | V/V | AVX512VL VAES | Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from ymm2 with a 128-bit round key from ymm3/m256; store the result in ymm1. |
| EVEX.NDS.512.66.0F38.WIG DE /r VAESDEC zmm1, zmm2, zmm3/m512 | B | V/V | AVX512F VAES | Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from zmm2 with a 128-bit round key from zmm3/m512; store the result in zmm1. |

## Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | NA | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |
| B | Full Mem | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |

### Description

This instruction performs a single round of the AES decryption flow using the Equivalent Inverse Cipher, with the round key from the second source operand, operating on a 128-bit data (state) from the first source operand, and store the result in the destination operand.

Use the AESDEC instruction for all but the last decryption round. For the last decryption round, use the AESDEC-CLAST instruction.

VEX and EVEX encoded versions of the instruction allows 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

### Operation

**AESDEC**

```
STATE ← SRC1
RoundKey ← SRC2
STATE ← InvShiftRows( STATE )
STATE ← InvSubBytes( STATE )
STATE ← InvMixColumns( STATE )
DEST[127:0] ← STATE XOR RoundKey
DEST[VLMAX-1:128] (Unmodified)
```

**VAESDEC (128b and 256b VEX encoded versions)**

```
(KL,V) = (1,128), (2,256)
FOR i = 0 to KL-1:
    STATE ← SRC1.xmm[i]
    RoundKey ← SRC2.xmm[i]
    STATE ← InvShiftRows( STATE )
    STATE ← InvSubBytes( STATE )
    STATE ← InvMixColumns( STATE )
    DEST.xmm[i] ← STATE XOR RoundKey
DEST[VLMAX-1:VL] ← 0
```

**VAESDEC (EVEX encoded version)**

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE ← SRC1.xmm[i]
    RoundKey ← SRC2.xmm[i]
    STATE ← InvShiftRows( STATE )
    STATE ← InvSubBytes( STATE )
    STATE ← InvMixColumns( STATE )
    DEST.xmm[i] ← STATE XOR RoundKey
DEST[VLMAX-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VAESDEC __m256i _mm256_aesdec_epi128(__m256i, __m256i);
VAESDEC __m512i _mm512_aesdec_epi128(__m512i, __m512i);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

VEX-encoded: Exceptions Type 4.

EVEX-encoded: See Exceptions Type E4NF.

## VAESDECLAST — Perform Last Round of an AES Decryption Flow

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| VEX.NDS.256.66.0F38.WIG DF /r VAESDECLAST ymm1, ymm2, ymm3/m256 | A | V/V | VAES | Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from ymm2 with a 128-bit round key from ymm3/m256; store the result in ymm1. |
| EVEX.NDS.128.66.0F38.WIG DF /r VAESDECLAST xmm1, xmm2, xmm3/m128 | B | V/V | AVX512VL VAES | Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from xmm3/m128; store the result in xmm1. |
| EVEX.NDS.256.66.0F38.WIG DF /r VAESDECLAST ymm1, ymm2, ymm3/m256 | B | V/V | AVX512VL VAES | Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from ymm2 with a 128-bit round key from ymm3/m256; store the result in ymm1. |
| EVEX.NDS.512.66.0F38.WIG DF /r VAESDECLAST zmm1, zmm2, zmm3/m512 | B | V/V | AVX512F VAES | Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from zmm2 with a 128-bit round key from zmm3/m512; store the result in zmm1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | NA | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |
| B | Full Mem | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |

### Description

This instruction performs the last round of the AES decryption flow using the Equivalent Inverse Cipher, with the round key from the second source operand, operating on a 128-bit data (state) from the first source operand, and store the result in the destination operand.

VEX and EVEX encoded versions of the instruction allows 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

### Operation

**AESDECLAST**

```
STATE ← SRC1
RoundKey ← SRC2
STATE ← InvShiftRows( STATE )
STATE ← InvSubBytes( STATE )
DEST[127:0] ← STATE XOR RoundKey
DEST[VLMAX-1:128] (Unmodified)
```

**VAESDECLAST (128b and 256b VEX encoded versions)**

```
(KL,VL) = (1,128), (2,256)
FOR i = 0 to KL-1:
    STATE ← SRC1.xmm[i]
    RoundKey ← SRC2.xmm[i]
    STATE ← InvShiftRows( STATE )
    STATE ← InvSubBytes( STATE )
    DEST.xmm[i] ← STATE XOR RoundKey
DEST[VLMAX-1:VL] ← 0
```

**VAESDECLAST (EVEX encoded version)**

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE ← SRC1.xmm[i]
    RoundKey ← SRC2.xmm[i]
    STATE ← InvShiftRows( STATE )
    STATE ← InvSubBytes( STATE )
    DEST.xmm[i] ← STATE XOR RoundKey
DEST[VLMAX-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VAESDECLAST __m256i _mm256_aesdeclast_epi128(__m256i, __m256i);
VAESDECLAST __m512i _mm512_aesdeclast_epi128(__m512i, __m512i);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

VEX-encoded: Exceptions Type 4.

EVEX-encoded: See Exceptions Type E4NF.

# VAESENC — Perform One Round of an AES Encryption Flow

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| VEX.NDS.256.66.0F38.WIG DC /r VAESENC ymm1, ymm2, ymm3/m256 | A | V/V | VAES | Perform one round of an AES encryption flow, operating on a 128-bit data (state) from ymm2 with a 128-bit round key from the ymm3/m256; store the result in ymm1. |
| EVEX.NDS.128.66.0F38.WIG DC /r VAESENC xmm1, xmm2, xmm3/m128 | B | V/V | AVX512VL VAES | Perform one round of an AES encryption flow, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from the xmm3/m128; store the result in xmm1. |
| EVEX.NDS.256.66.0F38.WIG DC /r VAESENC ymm1, ymm2, ymm3/m256 | B | V/V | AVX512VL VAES | Perform one round of an AES encryption flow, operating on a 128-bit data (state) from ymm2 with a 128-bit round key from the ymm3/m256; store the result in ymm1. |
| EVEX.NDS.512.66.0F38.WIG DC /r VAESENC zmm1, zmm2, zmm3/m512 | B | V/V | AVX512F VAES | Perform one round of an AES encryption flow, operating on a 128-bit data (state) from zmm2 with a 128-bit round key from the zmm3/m512; store the result in zmm1. |

## Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | NA | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |
| B | Full Mem | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |

### Description

This instruction performs a single round of an AES encryption flow using a round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand.

Use the AESENC instruction for all but the last encryption rounds. For the last encryption round, use the AESENC-CLAST instruction.

VEX and EVEX encoded versions of the instruction allows 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

### Operation

**AESENC**

```
STATE ← SRC1
RoundKey ← SRC2
STATE ← ShiftRows( STATE )
STATE ← SubBytes( STATE )
STATE ← MixColumns( STATE )
DEST[127:0] ← STATE XOR RoundKey
DEST[VLMAX-1:128] (Unmodified)
```

**VAESENC (128b and 256b VEX encoded versions)**

```
(KL,VL) = (1,128), (2,256)
FOR I ← 0 to KL-1:
    STATE ← SRC1.xmm[i]
    RoundKey ← SRC2.xmm[i]
    STATE ← ShiftRows( STATE )
    STATE ← SubBytes( STATE )
    STATE ← MixColumns( STATE )
    DEST.xmm[i] ← STATE XOR RoundKey
DEST[VLMAX-1:VL] ← 0
```

**VAESENC (EVEX encoded version)**

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i ← 0 to KL-1:
    STATE ← SRC1.xmm[i] // xmm[i] is the i'th xmm word in the SIMD register
    RoundKey ← SRC2.xmm[i]
    STATE ← ShiftRows( STATE )
    STATE ← SubBytes( STATE )
    STATE ← MixColumns( STATE )
    DEST.xmm[i] ← STATE XOR RoundKey
DEST[VLMAX-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VAESENC __m256i _mm256_aesenc_epi128(__m256i, __m256i);
VAESENC __m512i _mm512_aesenc_epi128(__m512i, __m512i);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

VEX-encoded: Exceptions Type 4.

EVEX-encoded: See Exceptions Type E4NF.

## VAESENCLAST — Perform Last Round of an AES Encryption Flow

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| VEX.NDS.256.66.0F38.WIG DD /r VAESENCLAST ymm1, ymm2, ymm3/m256 | A | V/V | VAES | Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from ymm2 with a 128 bit round key from ymm3/m256; store the result in ymm1. |
| EVEX.NDS.128.66.0F38.WIG DD /r VAESENCLAST xmm1, xmm2, xmm3/m128 | B | V/V | AVX512VL VAES | Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from xmm2 with a 128 bit round key from xmm3/m128; store the result in xmm1. |
| EVEX.NDS.256.66.0F38.WIG DD /r VAESENCLAST ymm1, ymm2, ymm3/m256 | B | V/V | AVX512VL VAES | Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from ymm2 with a 128 bit round key from ymm3/m256; store the result in ymm1. |
| EVEX.NDS.512.66.0F38.WIG DD /r VAESENCLAST zmm1, zmm2, zmm3/m512 | B | V/V | AVX512F VAES | Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from zmm2 with a 128 bit round key from zmm3/m512; store the result in zmm1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | NA | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |
| B | Full Mem | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |

### Description

This instruction performs the last round of an AES encryption flow using a round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand.

VEX and EVEX encoded versions of the instruction allows 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

### Operation

**AESENCLAST**

```
STATE ← SRC1
RoundKey ← SRC2
STATE ← ShiftRows( STATE )
STATE ← SubBytes( STATE )
DEST[127:0] ← STATE XOR RoundKey
DEST[VLMAX-1:128] (Unmodified)
```

**VAESENCLAST (128b and 256b VEX encoded versions)**

```
(KL, VL) = (1,128), (2,256)
FOR I=0 to KL-1:
    STATE ← SRC1.xmm[i]
    RoundKey ← SRC2.xmm[i]
    STATE ← ShiftRows( STATE )
    STATE ← SubBytes( STATE )
    DEST.xmm[i]← STATE XOR RoundKey
DEST[VLMAX-1:VL] ← 0
```

**VAESENCLAST (EVEX encoded version)**

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE ← SRC1.xmm[i]
    RoundKey ← SRC2.xmm[i]
    STATE ← ShiftRows( STATE )
    STATE ← SubBytes( STATE )
    DEST.xmm[i] ← STATE XOR RoundKey
DEST[VLMAX-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VAESENCLAST __m256i _mm256_aesenclast_epi128(__m256i, __m256i);
VAESENCLAST __m512i _mm512_aesenclast_epi128(__m512i, __m512i);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

VEX-encoded: Exceptions Type 4.

EVEX-encoded: See Exceptions Type E4NF.

## VPCLMULQDQ — Carry-Less Multiplication Quadword

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| VEX.NDS.256.66.0F3A.WIG 44 /r /ib VPCLMULQDQ ymm1, ymm2, ymm3/m256, imm8 | A | V/V | VPCLMULQDQ | Carry-less multiplication of one quadword of ymm2 by one quadword of ymm3/m256, stores the 128-bit result in ymm1. The immediate is used to determine which quadwords of ymm2 and ymm3/m256 should be used. |
| EVEX.NDS.128.66.0F3A.WIG 44 /r /ib VPCLMULQDQ xmm1, xmm2, xmm3/m128, imm8 | B | V/V | AVX512VL VPCLMULQDQ | Carry-less multiplication of one quadword of xmm2 by one quadword of xmm3/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm2 and xmm3/m128 should be used. |
| EVEX.NDS.256.66.0F3A.WIG 44 /r /ib VPCLMULQDQ ymm1, ymm2, ymm3/m256, imm8 | B | V/V | AVX512VL VPCLMULQDQ | Carry-less multiplication of one quadword of ymm2 by one quadword of ymm3/m256, stores the 128-bit result in ymm1. The immediate is used to determine which quadwords of ymm2 and ymm3/m256 should be used. |
| EVEX.NDS.512.66.0F3A.WIG 44 /r /ib VPCLMULQDQ zmm1, zmm2, zmm3/m512, imm8 | B | V/V | AVX512F VPCLMULQDQ | Carry-less multiplication of one quadword of zmm2 by one quadword of zmm3/m512, stores the 128-bit result in zmm1. The immediate is used to determine which quadwords of zmm2 and zmm3/m512 should be used. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | NA | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |
| B | Full Mem | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |

### Description

Performs a carry-less multiplication of two quadwords, selected from the first source and second source operand according to the value of the immediate byte. Bits 4 and 0 are used to select which 64-bit half of each operand to use according to the table below, other bits of the immediate byte are ignored.

The EVEX encoded form of this instruction does not support memory fault suppression.

### Table 2-2. PCLMULQDQ Quadword Selection of Immediate Byte

| imm[4] | imm[0] | PCLMULQDQ Operation |
|---|---|---|
| 0 | 0 | CL_MUL( SRC2[63:0], SRC1[63:0] ) |
| 0 | 1 | CL_MUL( SRC2[63:0], SRC1[127:64] ) |
| 1 | 0 | CL_MUL( SRC2[127:64], SRC1[63:0] ) |
| 1 | 1 | CL_MUL( SRC2[127:64], SRC1[127:64] ) |

NOTES:

SRC2 denotes the second source operand, which can be a register or memory; SRC1 denotes the first source and destination operand.

The first source operand and the destination operand are the same and must be a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. Bits (VL_MAX-1:128) of the corresponding YMM destination register remain unchanged.

Compilers and assemblers may implement the following pseudo-op syntax to simply programming and emit the required encoding for imm8.

**Table 2-3. Pseudo-Op and PCLMULQDQ Implementation**

| Pseudo-Op | Imm8 Encoding |
|---|---|
| PCLMULLQLQDQ xmm1, xmm2 | 0000_0000B |
| PCLMULHQLQDQ xmm1, xmm2 | 0000_0001B |
| PCLMULLQHQDQ xmm1, xmm2 | 0001_0000B |
| PCLMULHQHQDQ xmm1, xmm2 | 0001_0001B |

### Operation

```
define PCLMUL128(X,Y):                    // helper function
    FOR i ← 0 to 63:
        TMP [ i ] ← X[ 0 ] and Y[ i ]
        FOR j ← 1 to i:
            TMP [ i ] ← TMP [ i ] xor (X[ j ] and Y[ i - j ])
        DEST[ i ] ← TMP[ i ]
    FOR i ← 64 to 126:
        TMP [ i ] ← 0
        FOR j ← i - 63 to 63:
            TMP [ i ] ← TMP [ i ] xor (X[ j ] and Y[ i - j ])
        DEST[ i ] ← TMP[ i ]
    DEST[127] ← 0;
    RETURN DEST                           // 128b vector
```

**PCLMULQDQ (SSE version)**

```
IF Imm8[0] = 0:
    TEMP1 ← SRC1.qword[0]
ELSE:
    TEMP1 ← SRC1.qword[1]
IF Imm8[4] = 0:
    TEMP2 ← SRC2.qword[0]
ELSE:
    TEMP2 ← SRC2.qword[1]
DEST[127:0] ← PCLMUL128(TEMP1, TEMP2)
DEST[VLMAX-1:128] (Unmodified)
```

**VPCLMULQDQ (128b and 256b VEX encoded versions)**

```
(KL,VL) = (1,128), (2,256)
FOR i= 0 to KL-1:
    IF Imm8[0] = 0:
        TEMP1 ← SRC1.xmm[i].qword[0]
    ELSE:
        TEMP1 ← SRC1.xmm[i].qword[1]
    IF Imm8[4] = 0:
        TEMP2 ← SRC2.xmm[i].qword[0]
    ELSE:
        TEMP2 ← SRC2.xmm[i].qword[1]
    DEST.xmm[i] ← PCLMUL128(TEMP1, TEMP2)
DEST[VLMAX-1:VL] ← 0
```

**VPCLMULQDQ (EVEX encoded version)**

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    IF Imm8[0] = 0:
        TEMP1 ← SRC1.xmm[i].qword[0]
    ELSE:
        TEMP1 ← SRC1.xmm[i].qword[1]
    IF Imm8[4] = 0:
        TEMP2 ← SRC2.xmm[i].qword[0]
    ELSE:
        TEMP2 ← SRC2.xmm[i].qword[1]
    DEST.xmm[i] ← PCLMUL128(TEMP1, TEMP2)
DEST[VLMAX-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VPCLMULQDQ __m256i _mm256_clmulepi64_epi128(__m256i, __m256i, const int);
VPCLMULQDQ __m512i _mm512_clmulepi64_epi128(__m512i, __m512i, const int);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

VEX-encoded: Exceptions Type 4.

EVEX-encoded: See Exceptions Type E4NF.

## VPCOMPRESS — Store Sparse Packed Byte/Word Integer Values into Dense Memory/Register

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| EVEX.128.66.0F38.W0 63 /r VPCOMPRESSB m128{k1}, xmm1 | A | V/V | AVX512_VBMI2 AVX512VL | Compress up to 128 bits of packed byte values from xmm1 to m128 with writemask k1. |
| EVEX.128.66.0F38.W0 63 /r VPCOMPRESSB xmm1{k1}{z}, xmm2 | B | V/V | AVX512_VBMI2 AVX512VL | Compress up to 128 bits of packed byte values from xmm2 to xmm1 with writemask k1. |
| EVEX.256.66.0F38.W0 63 /r VPCOMPRESSB m256{k1}, ymm1 | A | V/V | AVX512_VBMI2 AVX512VL | Compress up to 256 bits of packed byte values from ymm1 to m256 with writemask k1. |
| EVEX.256.66.0F38.W0 63 /r VPCOMPRESSB ymm1{k1}{z}, ymm2 | B | V/V | AVX512_VBMI2 AVX512VL | Compress up to 256 bits of packed byte values from ymm2 to ymm1 with writemask k1. |
| EVEX.512.66.0F38.W0 63 /r VPCOMPRESSB m512{k1}, zmm1 | A | V/V | AVX512_VBMI2 | Compress up to 512 bits of packed byte values from zmm1 to m512 with writemask k1. |
| EVEX.512.66.0F38.W0 63 /r VPCOMPRESSB zmm1{k1}{z}, zmm2 | B | V/V | AVX512_VBMI2 | Compress up to 512 bits of packed byte values from zmm2 to zmm1 with writemask k1. |
| EVEX.128.66.0F38.W1 63 /r VPCOMPRESSW m128{k1}, xmm1 | A | V/V | AVX512_VBMI2 AVX512VL | Compress up to 128 bits of packed word values from xmm1 to m128 with writemask k1. |
| EVEX.128.66.0F38.W1 63 /r VPCOMPRESSW xmm1{k1}{z}, xmm2 | B | V/V | AVX512_VBMI2 AVX512VL | Compress up to 128 bits of packed word values from xmm2 to xmm1 with writemask k1. |
| EVEX.256.66.0F38.W1 63 /r VPCOMPRESSW m256{k1}, ymm1 | A | V/V | AVX512_VBMI2 AVX512VL | Compress up to 256 bits of packed word values from ymm1 to m256 with writemask k1. |
| EVEX.256.66.0F38.W1 63 /r VPCOMPRESSW ymm1{k1}{z}, ymm2 | B | V/V | AVX512_VBMI2 AVX512VL | Compress up to 256 bits of packed word values from ymm2 to ymm1 with writemask k1. |
| EVEX.512.66.0F38.W1 63 /r VPCOMPRESSW m512{k1}, zmm1 | A | V/V | AVX512_VBMI2 | Compress up to 512 bits of packed word values from zmm1 to m512 with writemask k1. |
| EVEX.512.66.0F38.W1 63 /r VPCOMPRESSW zmm1{k1}{z}, zmm2 | B | V/V | AVX512_VBMI2 | Compress up to 512 bits of packed word values from zmm2 to zmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | Tuple1 Scalar | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |
| B | NA | ModRM:r/m (w) | ModRM:reg (r) | NA | NA |

### Description

Compress (stores) up to 64 byte values or 32 word values from the source operand (second operand) to the destination operand (first operand), based on the active elements determined by the writemask operand. Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Moves up to 512 bits of packed byte values from the source operand (second operand) to the destination operand (first operand). This instruction is used to store partial contents of a vector register into a byte vector or single memory location using the active elements in operand writemask.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

This instruction supports memory fault suppression.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

**VPCOMPRESSB store form**

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
k ← 0
FOR j ← 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[k] ← SRC.byte[j]
        k ← k +1
```

**VPCOMPRESSB reg-reg form**

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
k ← 0
FOR j ← 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[k] ← SRC.byte[j]
        k ← k + 1
    IF *merging-masking*:
        *DEST[VL-1:k*8] remains unchanged*
        ELSE DEST[VL-1:k*8] ← 0
DEST[MAX_VL-1:VL] ← 0
```

**VPCOMPRESSW store form**

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
k ← 0
FOR j ← 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.word[k] ← SRC.word[j]
        k ← k + 1
```

**VPCOMPRESSW reg-reg form**

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
k ← 0
FOR j ← 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.word[k] ← SRC.word[j]
        k ← k + 1
    IF *merging-masking*:
        *DEST[VL-1:k*16] remains unchanged*
        ELSE DEST[VL-1:k*16] ← 0
DEST[MAX_VL-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VPCOMPRESSB __m128i _mm_mask_compress_epi8(__m128i, __mmask16, __m128i);
VPCOMPRESSB __m128i _mm_maskz_compress_epi8(__mmask16, __m128i);
VPCOMPRESSB __m256i _mm256_mask_compress_epi8(__m256i, __mmask32, __m256i);
VPCOMPRESSB __m256i _mm256_maskz_compress_epi8(__mmask32, __m256i);
VPCOMPRESSB __m512i _mm512_mask_compress_epi8(__m512i, __mmask64, __m512i);
VPCOMPRESSB __m512i _mm512_maskz_compress_epi8(__mmask64, __m512i);
VPCOMPRESSB void _mm_mask_compressstoreu_epi8(void*, __mmask16, __m128i);
VPCOMPRESSB void _mm256_mask_compressstoreu_epi8(void*, __mmask32, __m256i);
VPCOMPRESSB void _mm512_mask_compressstoreu_epi8(void*, __mmask64, __m512i);
VPCOMPRESSW __m128i _mm_mask_compress_epi16(__m128i, __mmask8, __m128i);
VPCOMPRESSW __m128i _mm_maskz_compress_epi16(__mmask8, __m128i);
VPCOMPRESSW __m256i _mm256_mask_compress_epi16(__m256i, __mmask16, __m256i);
VPCOMPRESSW __m256i _mm256_maskz_compress_epi16(__mmask16, __m256i);
VPCOMPRESSW __m512i _mm512_mask_compress_epi16(__m512i, __mmask32, __m512i);
VPCOMPRESSW __m512i _mm512_maskz_compress_epi16(__mmask32, __m512i);
VPCOMPRESSW void _mm_mask_compressstoreu_epi16(void*, __mmask8, __m128i);
VPCOMPRESSW void _mm256_mask_compressstoreu_epi16(void*, __mmask16, __m256i);
VPCOMPRESSW void _mm512_mask_compressstoreu_epi16(void*, __mmask32, __m512i);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type E4.

## VPDPBUSD — Multiply and Add Unsigned and Signed Bytes

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| EVEX.DDS.128.66.0F38.W0 50 /r VPDPBUSD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst | A | V/V | AVX512_VNNI AVX512VL | Multiply unsigned and signed bytes, add horizontal pair of signed words, pack signed dwords to xmm1 under writemask k1. |
| EVEX.DDS.256.66.0F38.W0 50 /r VPDPBUSD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst | A | V/V | AVX512_VNNI AVX512VL | Multiply unsigned and signed bytes, add horizontal pair of signed words, pack signed dwords to ymm1 under writemask k1. |
| EVEX.DDS.512.66.0F38.W0 50 /r VPDPBUSD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst | A | V/V | AVX512_VNNI | Multiply unsigned and signed bytes, add horizontal pair of signed words, pack signed dwords to zmm1 under writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | Full | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |

### Description

Multiplies the individual unsigned bytes of the first source operand by the corresponding signed bytes of the second source operand, producing intermediate signed word results. The word results are then summed and accumulated in the destination dword element size operand.

This instruction supports memory fault suppression.

### Operation

**VPDPBUSD dest, src1, src2**

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST ← DEST
FOR i ← 0 TO KL-1:
    IF k1[i] or *no writemask*:
        // Byte elements of SRC1 are zero-extended to 16b and
        // byte elements of SRC2 are sign extended to 16b before multiplication.
        IF SRC2 is memory and EVEX.b == 1:
            t ← SRC2.dword[0]
        ELSE:
            t ← SRC2.dword[i]
        p1word ← ZERO_EXTEND(SRC1.byte[4*i]) * SIGN_EXTEND(t.byte[0])
        p2word ← ZERO_EXTEND(SRC1.byte[4*i+1]) * SIGN_EXTEND(t.byte[1])
        p3word ← ZERO_EXTEND(SRC1.byte[4*i+2]) * SIGN_EXTEND(t.byte[2])
        p4word ← ZERO_EXTEND(SRC1.byte[4*i+3]) * SIGN_EXTEND(t.byte[3])
        DEST.dword[i] ← ORIGDEST.dword[i] + p1word + p2word + p3word + p4word
    ELSE IF *zeroing*:
        DEST.dword[i] ← 0
    ELSE:    // Merge masking, dest element unchanged
        DEST.dword[i] ← ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VPDPBUSD __m128i _mm_dpbusd_epi32(__m128i, __m128i, __m128i);
VPDPBUSD __m128i _mm_mask_dpbusd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPBUSD __m128i _mm_maskz_dpbusd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPBUSD __m256i _mm256_dpbusd_epi32(__m256i, __m256i, __m256i);
VPDPBUSD __m256i _mm256_mask_dpbusd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPBUSD __m256i _mm256_maskz_dpbusd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPBUSD __m512i _mm512_dpbusd_epi32(__m512i, __m512i, __m512i);
VPDPBUSD __m512i _mm512_mask_dpbusd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPBUSD __m512i _mm512_maskz_dpbusd_epi32(__mmask16, __m512i, __m512i, __m512i);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type E4.

## VPDPBUSDS — Multiply and Add Unsigned and Signed Bytes with Saturation

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| EVEX.DDS.128.66.0F38.W0 51 /r VPDPBUSDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst | A | V/V | AVX512_VNNI AVX512VL | Multiply unsigned and signed bytes, add horizontal pair of signed words, pack signed dwords, or saturated signed dwords, to xmm1 under writemask k1. |
| EVEX.DDS.256.66.0F38.W0 51 /r VPDPBUSDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst | A | V/V | AVX512_VNNI AVX512VL | Multiply unsigned and signed bytes, add horizontal pair of signed words, pack signed dwords, or saturated signed dwords, to ymm1 under writemask k1. |
| EVEX.DDS.512.66.0F38.W0 51 /r VPDPBUSDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst | A | V/V | AVX512_VNNI | Multiply unsigned and signed bytes, add horizontal pair of signed words, pack signed dwords, or saturated signed dwords, to zmm1 under writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | Full | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |

### Description

Multiplies the individual unsigned bytes of the first source operand by the corresponding signed bytes of the second source operand, producing intermediate signed word results. The word results are then summed and accumulated in the destination dword element size operand. If the intermediate sum overflows a 32b signed number the result is saturated to either 0x7FFF_FFFF for positive numbers of 0x8000_0000 for negative numbers.

This instruction supports memory fault suppression.

### Operation

**VPDPBUSDS dest, src1, src2**

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST ← DEST
FOR i ← 0 TO KL-1:
    IF k1[i] or *no writemask*:
        // Byte elements of SRC1 are zero-extended to 16b and
        // byte elements of SRC2 are sign extended to 16b before multiplication.
        IF SRC2 is memory and EVEX.b == 1:
            t ← SRC2.dword[0]
        ELSE:
            t ← SRC2.dword[i]
        p1word ← ZERO_EXTEND(SRC1.byte[4*i]) * SIGN_EXTEND(t.byte[0])
        p2word ← ZERO_EXTEND(SRC1.byte[4*i+1]) * SIGN_EXTEND(t.byte[1])
        p3word ← ZERO_EXTEND(SRC1.byte[4*i+2]) * SIGN_EXTEND(t.byte[2])
        p4word ← ZERO_EXTEND(SRC1.byte[4*i+3]) *SIGN_EXTEND(t.byte[3])
        DEST.dword[i] ← SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)
    ELSE IF *zeroing*:
        DEST.dword[i] ← 0
    ELSE:    // Merge masking, dest element unchanged
        DEST.dword[i] ← ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VPDPBUSDS __m128i _mm_dpbusds_epi32(__m128i, __m128i, __m128i);
VPDPBUSDS __m128i _mm_mask_dpbusds_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPBUSDS __m128i _mm_maskz_dpbusds_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPBUSDS __m256i _mm256_dpbusds_epi32(__m256i, __m256i, __m256i);
VPDPBUSDS __m256i _mm256_mask_dpbusds_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPBUSDS __m256i _mm256_maskz_dpbusds_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPBUSDS __m512i _mm512_dpbusds_epi32(__m512i, __m512i, __m512i);
VPDPBUSDS __m512i _mm512_mask_dpbusds_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPBUSDS __m512i _mm512_maskz_dpbusds_epi32(__mmask16, __m512i, __m512i, __m512i);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type E4.

## VPDPWSSD — Multiply and Add Signed Word Integers

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| EVEX.DDS.128.66.0F38.W0 52 /r VPDPWSSD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst | A | V/V | AVX512_VNNI AVX512VL | Multiply signed word integers in xmm2 by the signed word integers in xmm3/m128, add adjacent doubleword results, and store in xmm1 under writemask k1. |
| EVEX.DDS.256.66.0F38.W0 52 /r VPDPWSSD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst | A | V/V | AVX512_VNNI AVX512VL | Multiply the signed word integers in ymm2 by the signed word integers in ymm3/m256, add adjacent doubleword results, and store in ymm1 under writemask k1. |
| EVEX.DDS.512.66.0F38.W0 52 /r VPDPWSSD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst | A | V/V | AVX512_VNNI | Multiply the signed word integers in zmm2 by the signed word integers in zmm3/m512, add adjacent doubleword results, and store in zmm1 under writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | Full | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |

### Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing intermediate signed, doubleword results. The adjacent doubleword results are then summed and accumulated in the destination operand.

This instruction supports memory fault suppression.

### Operation

**VPDPWSSD dest, src1, src2**

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST ← DEST
FOR i ← 0 TO KL-1:
    IF k1[i] or *no writemask*:
        IF SRC2 is memory and EVEX.b == 1:
            t ← SRC2.dword[0]
        ELSE:
            t ← SRC2.dword[i]
        p1dword ← SRC1.word[2*i] * t.word[0]
        p2dword ← SRC1.word[2*i+1] * t.word[1]
        DEST.dword[i] ← ORIGDEST.dword[i] + p1dword + p2dword
    ELSE IF *zeroing*:
        DEST.dword[i] ← 0
    ELSE:       // Merge masking, dest element unchanged
        DEST.dword[i] ← ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VPDPWSSD __m128i _mm_dpwssd_epi32(__m128i, __m128i, __m128i);
VPDPWSSD __m128i _mm_mask_dpwssd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPWSSD __m128i _mm_maskz_dpwssd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPWSSD __m256i _mm256_dpwssd_epi32(__m256i, __m256i, __m256i);
VPDPWSSD __m256i _mm256_mask_dpwssd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPWSSD __m256i _mm256_maskz_dpwssd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPWSSD __m512i _mm512_dpwssd_epi32(__m512i, __m512i, __m512i);
VPDPWSSD __m512i _mm512_mask_dpwssd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPWSSD __m512i _mm512_maskz_dpwssd_epi32(__mmask16, __m512i, __m512i, __m512i);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type E4.

## VPDPWSSDS — Multiply and Add Word Integers with Saturation

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| EVEX.DDS.128.66.0F38.W0 53 /r VPDPWSSDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst | A | V/V | AVX512_VNNI AVX512VL | Multiply word integers in xmm2 by the word integers in xmm3/m128, add adjacent doubleword results with signed saturation, and store in xmm1 under writemask k1. |
| EVEX.DDS.256.66.0F38.W0 53 /r VPDPWSSDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst | A | V/V | AVX512_VNNI AVX512VL | Multiply the word integers in ymm2 by the word integers in ymm3/m256, add adjacent doubleword results with signed saturation, and store in ymm1 under writemask k1. |
| EVEX.DDS.512.66.0F38.W0 53 /r VPDPWSSDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst | A | V/V | AVX512_VNNI | Multiply the word integers in zmm2 by the word integers in zmm3/m512, add adjacent doubleword results with signed saturation, and store in zmm1 under writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | Full | ModRM:reg (r, w) | EVEX.vvvv | ModRM:r/m (r) | NA |

### Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing intermediate signed, doubleword results. The adjacent doubleword results are then summed and accumulated in the destination operand. If the intermediate sum overflows a 32b signed number, the result is saturated to either 0x7FFF_FFFF for positive numbers of 0x8000_0000 for negative numbers.

This instruction supports memory fault suppression.

### Operation

**VPDPWSSDS dest, src1, src2**

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST ← DEST
FOR i ← 0 TO KL-1:
    IF k1[i] or *no writemask*:
        IF SRC2 is memory and EVEX.b == 1:
            t ← SRC2.dword[0]
        ELSE:
            t ← SRC2.dword[i]
        p1dword ← SRC1.word[2*i] * t.word[0]
        p2dword ← SRC1.word[2*i+1] * t.word[1]
        DEST.dword[i] ← SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)
    ELSE IF *zeroing*:
        DEST.dword[i] ← 0
    ELSE:      // Merge masking, dest element unchanged
        DEST.dword[i] ← ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VPDPWSSDS __m128i _mm_dpwssds_epi32(__m128i, __m128i, __m128i);
VPDPWSSDS __m128i _mm_mask_dpwssd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPWSSDS __m128i _mm_maskz_dpwssd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPWSSDS __m256i _mm256_dpwssd_epi32(__m256i, __m256i, __m256i);
VPDPWSSDS __m256i _mm256_mask_dpwssd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPWSSDS __m256i _mm256_maskz_dpwssd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPWSSDS __m512i _mm512_dpwssd_epi32(__m512i, __m512i, __m512i);
VPDPWSSDS __m512i _mm512_mask_dpwssd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPWSSDS __m512i _mm512_maskz_dpwssd_epi32(__mmask16, __m512i, __m512i, __m512i);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type E4.

## VPEXPAND — Expand Byte/Word Values

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| EVEX.128.66.0F38.W0 62 /r VPEXPANDB xmm1{k1}{z}, m128 | A | V/V | AVX512_VBMI2 AVX512VL | Expands up to 128 bits of packed byte values from m128 to xmm1 with writemask k1. |
| EVEX.128.66.0F38.W0 62 /r VPEXPANDB xmm1{k1}{z}, xmm2 | B | V/V | AVX512_VBMI2 AVX512VL | Expands up to 128 bits of packed byte values from xmm2 to xmm1 with writemask k1. |
| EVEX.256.66.0F38.W0 62 /r VPEXPANDB ymm1{k1}{z}, m256 | A | V/V | AVX512_VBMI2 AVX512VL | Expands up to 256 bits of packed byte values from m256 to ymm1 with writemask k1. |
| EVEX.256.66.0F38.W0 62 /r VPEXPANDB ymm1{k1}{z}, ymm2 | B | V/V | AVX512_VBMI2 AVX512VL | Expands up to 256 bits of packed byte values from ymm2 to ymm1 with writemask k1. |
| EVEX.512.66.0F38.W0 62 /r VPEXPANDB zmm1{k1}{z}, m512 | A | V/V | AVX512_VBMI2 | Expands up to 512 bits of packed byte values from m512 to zmm1 with writemask k1. |
| EVEX.512.66.0F38.W0 62 /r VPEXPANDB zmm1{k1}{z}, zmm2 | B | V/V | AVX512_VBMI2 | Expands up to 512 bits of packed byte values from zmm2 to zmm1 with writemask k1. |
| EVEX.128.66.0F38.W1 62 /r VPEXPANDW xmm1{k1}{z}, m128 | A | V/V | AVX512_VBMI2 AVX512VL | Expands up to 128 bits of packed word values from m128 to xmm1 with writemask k1. |
| EVEX.128.66.0F38.W1 62 /r VPEXPANDW xmm1{k1}{z}, xmm2 | B | V/V | AVX512_VBMI2 AVX512VL | Expands up to 128 bits of packed word values from xmm2 to xmm1 with writemask k1. |
| EVEX.256.66.0F38.W1 62 /r VPEXPANDW ymm1{k1}{z}, m256 | A | V/V | AVX512_VBMI2 AVX512VL | Expands up to 256 bits of packed word values from m256 to ymm1 with writemask k1. |
| EVEX.256.66.0F38.W1 62 /r VPEXPANDW ymm1{k1}{z}, ymm2 | B | V/V | AVX512_VBMI2 AVX512VL | Expands up to 256 bits of packed word values from ymm2 to ymm1 with writemask k1. |
| EVEX.512.66.0F38.W1 62 /r VPEXPANDW zmm1{k1}{z}, m512 | A | V/V | AVX512_VBMI2 | Expands up to 512 bits of packed word values from m512 to zmm1 with writemask k1. |
| EVEX.512.66.0F38.W1 62 /r VPEXPANDW zmm1{k1}{z}, zmm2 | B | V/V | AVX512_VBMI2 | Expands up to 512 bits of packed byte integer values from zmm2 to zmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | Tuple1 Scalar | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |
| B | NA | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

### Description

Expands (loads) up to 64 byte integer values or 32 word integer values from the source operand (memory operand) to the destination operand (register operand), based on the active elements determined by the writemask operand.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Moves 128, 256 or 512 bits of packed byte integer values from the source operand (memory operand) to the destination operand (register operand). This instruction is used to load from an int8 vector register or memory location while inserting the data into sparse elements of destination vector register using the active elements pointed out by the operand writemask.

This instruction supports memory fault suppression.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

**VPEXPANDB**

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
k ← 0
FOR j ← 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[j] ← SRC.byte[k];
        k ← k + 1
        ELSE:
            IF *merging-masking*:
                *DEST.byte[j] remains unchanged*
                ELSE:                       ; zeroing-masking
                    DEST.byte[j] ← 0
DEST[MAX_VL-1:VL] ← 0
```

**VPEXPANDW**

```
(KL, VL) = (8,128), (16,256), (32, 512)
k ← 0
FOR j ← 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.word[j] ← SRC.word[k];
        k ← k + 1
        ELSE:
            IF *merging-masking*:
                *DEST.word[j] remains unchanged*
                ELSE:                       ; zeroing-masking
                    DEST.word[j] ← 0
DEST[MAX_VL-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VPEXPAND __m128i _mm_mask_expand_epi8(__m128i, __mmask16, __m128i);
VPEXPAND __m128i _mm_maskz_expand_epi8(__mmask16, __m128i);
VPEXPAND __m128i _mm_mask_expandloadu_epi8(__m128i, __mmask16, const void*);
VPEXPAND __m128i _mm_maskz_expandloadu_epi8(__mmask16, const void*);
VPEXPAND __m256i _mm256_mask_expand_epi8(__m256i, __mmask32, __m256i);
VPEXPAND __m256i _mm256_maskz_expand_epi8(__mmask32, __m256i);
VPEXPAND __m256i _mm256_mask_expandloadu_epi8(__m256i, __mmask32, const void*);
VPEXPAND __m256i _mm256_maskz_expandloadu_epi8(__mmask32, const void*);
VPEXPAND __m512i _mm512_mask_expand_epi8(__m512i, __mmask64, __m512i);
VPEXPAND __m512i _mm512_maskz_expand_epi8(__mmask64, __m512i);
VPEXPAND __m512i _mm512_mask_expandloadu_epi8(__m512i, __mmask64, const void*);
VPEXPAND __m512i _mm512_maskz_expandloadu_epi8(__mmask64, const void*);
VPEXPANDW __m128i _mm_mask_expand_epi16(__m128i, __mmask8, __m128i);
VPEXPANDW __m128i _mm_maskz_expand_epi16(__mmask8, __m128i);
VPEXPANDW __m128i _mm_mask_expandloadu_epi16(__m128i, __mmask8, const void*);
VPEXPANDW __m128i _mm_maskz_expandloadu_epi16(__mmask8, const void *);
VPEXPANDW __m256i _mm256_mask_expand_epi16(__m256i, __mmask16, __m256i);
VPEXPANDW __m256i _mm256_maskz_expand_epi16(__mmask16, __m256i);
VPEXPANDW __m256i _mm256_mask_expandloadu_epi16(__m256i, __mmask16, const void*);
VPEXPANDW __m256i _mm256_maskz_expandloadu_epi16(__mmask16, const void*);
VPEXPANDW __m512i _mm512_mask_expand_epi16(__m512i, __mmask32, __m512i);
VPEXPANDW __m512i _mm512_maskz_expand_epi16(__mmask32, __m512i);
VPEXPANDW __m512i _mm512_mask_expandloadu_epi16(__m512i, __mmask32, const void*);
VPEXPANDW __m512i _mm512_maskz_expandloadu_epi16(__mmask32, const void*);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type E4.

## VPOPCNT — Return the Count of Number of Bits Set to 1 in BYTE/WORD/DWORD/QWORD

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| EVEX.128.66.0F38.W0 54 /r VPOPCNTB xmm1{k1}{z}, xmm2/m128 | A | V/V | AVX512_BITALG AVX512VL | Counts the number of bits set to one in xmm2/m128 and puts the result in xmm1 with writemask k1. |
| EVEX.256.66.0F38.W0 54 /r VPOPCNTB ymm1{k1}{z}, ymm2/m256 | A | V/V | AVX512_BITALG AVX512VL | Counts the number of bits set to one in ymm2/m256 and puts the result in ymm1 with writemask k1. |
| EVEX.512.66.0F38.W0 54 /r VPOPCNTB zmm1{k1}{z}, zmm2/m512 | A | V/V | AVX512_BITALG | Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1. |
| EVEX.128.66.0F38.W1 54 /r VPOPCNTW xmm1{k1}{z}, xmm2/m128 | A | V/V | AVX512_BITALG AVX512VL | Counts the number of bits set to one in xmm2/m128 and puts the result in xmm1 with writemask k1. |
| EVEX.256.66.0F38.W1 54 /r VPOPCNTW ymm1{k1}{z}, ymm2/m256 | A | V/V | AVX512_BITALG AVX512VL | Counts the number of bits set to one in ymm2/m256 and puts the result in ymm1 with writemask k1. |
| EVEX.512.66.0F38.W1 54 /r VPOPCNTW zmm1{k1}{z}, zmm2/m512 | A | V/V | AVX512_BITALG | Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1. |
| EVEX.128.66.0F38.W0 55 /r VPOPCNTD xmm1{k1}{z}, xmm2/m128/m32bcst | B | V/V | AVX512_VPOPCNTDQ AVX512VL | Counts the number of bits set to one in xmm2/m128/m32bcst and puts the result in xmm1 with writemask k1. |
| EVEX.256.66.0F38.W0 55 /r VPOPCNTD ymm1{k1}{z}, ymm2/m256/m32bcst | B | V/V | AVX512_VPOPCNTDQ AVX512VL | Counts the number of bits set to one in ymm2/m256/m32bcst and puts the result in ymm1 with writemask k1. |
| EVEX.512.66.0F38.W0 55 /r VPOPCNTD zmm1{k1}{z}, zmm2/m512/m32bcst | B | V/V | AVX512_VPOPCNTDQ | Counts the number of bits set to one in zmm2/m512/m32bcst and puts the result in zmm1 with writemask k1. |
| EVEX.128.66.0F38.W1 55 /r VPOPCNTQ xmm1{k1}{z}, xmm2/m128/m64bcst | B | V/V | AVX512_VPOPCNTDQ AVX512VL | Counts the number of bits set to one in xmm2/m128/m32bcst and puts the result in xmm1 with writemask k1. |
| EVEX.256.66.0F38.W1 55 /r VPOPCNTQ ymm1{k1}{z}, ymm2/m256/m64bcst | B | V/V | AVX512_VPOPCNTDQ AVX512VL | Counts the number of bits set to one in ymm2/m256/m32bcst and puts the result in ymm1 with writemask k1. |
| EVEX.512.66.0F38.W1 55 /r VPOPCNTQ zmm1{k1}{z}, zmm2/m512/m64bcst | B | V/V | AVX512_VPOPCNTDQ | Counts the number of bits set to one in zmm2/m512/m64bcst and puts the result in zmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | Full Mem | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |
| B | Full | ModRM:reg (w) | ModRM:r/m (r) | NA | NA |

### Description

This instruction counts the number of bits set to one in each byte, word, dword or qword element of its source (e.g., zmm2 or memory) and places the results in the destination register (zmm1). This instruction supports memory fault suppression.

### Operation

**VPOPCNTB**

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.byte[j] ← POPCNT(SRC.byte[j])
    ELSE IF *merging-masking*:
        *DEST.byte[j] remains unchanged*
    ELSE:
        DEST.byte[j] ← 0
DEST[MAX_VL-1:VL] ← 0
```

**VPOPCNTW**

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.word[j] ← POPCNT(SRC.word[j])
    ELSE IF *merging-masking*:
        *DEST.word[j] remains unchanged*
    ELSE:
        DEST.word[j] ← 0
DEST[MAX_VL-1:VL] ← 0
```

**VPOPCNTD**

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        IF SRC is broadcast memop:
            t ← SRC.dword[0]
        ELSE:
            t ← SRC.dword[j]
        DEST.dword[j] ← POPCNT(t)
    ELSE IF *merging-masking*:
        *DEST..dword[j] remains unchanged*
    ELSE:
        DEST..dword[j] ← 0
DEST[MAX_VL-1:VL] ← 0
```

**VPOPCNTQ**

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        IF SRC is broadcast memop:
            t ← SRC.qword[0]
        ELSE:
            t ← SRC.qword[j]
        DEST.qword[j] ← POPCNT(t)
    ELSE IF *merging-masking*:
        *DEST..qword[j] remains unchanged*
    ELSE:
        DEST..qword[j] ← 0
DEST[MAX_VL-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VPOPCNTW __m128i _mm_popcnt_epi16(__m128i);
VPOPCNTW __m128i _mm_mask_popcnt_epi16(__m128i, __mmask8, __m128i);
VPOPCNTW __m128i _mm_maskz_popcnt_epi16(__mmask8, __m128i);
VPOPCNTW __m256i _mm256_popcnt_epi16(__m256i);
VPOPCNTW __m256i _mm256_mask_popcnt_epi16(__m256i, __mmask16, __m256i);
VPOPCNTW __m256i _mm256_maskz_popcnt_epi16(__mmask16, __m256i);
VPOPCNTW __m512i _mm512_popcnt_epi16(__m512i);
VPOPCNTW __m512i _mm512_mask_popcnt_epi16(__m512i, __mmask32, __m512i);
VPOPCNTW __m512i _mm512_maskz_popcnt_epi16(__mmask32, __m512i);
VPOPCNTQ __m128i _mm_popcnt_epi64(__m128i);
VPOPCNTQ __m128i _mm_mask_popcnt_epi64(__m128i, __mmask8, __m128i);
VPOPCNTQ __m128i _mm_maskz_popcnt_epi64(__mmask8, __m128i);
VPOPCNTQ __m256i _mm256_popcnt_epi64(__m256i);
VPOPCNTQ __m256i _mm256_mask_popcnt_epi64(__m256i, __mmask8, __m256i);
VPOPCNTQ __m256i _mm256_maskz_popcnt_epi64(__mmask8, __m256i);
VPOPCNTQ __m512i _mm512_popcnt_epi64(__m512i);
VPOPCNTQ __m512i _mm512_mask_popcnt_epi64(__m512i, __mmask8, __m512i);
VPOPCNTQ __m512i _mm512_maskz_popcnt_epi64(__mmask8, __m512i);
VPOPCNTD __m128i _mm_popcnt_epi32(__m128i);
VPOPCNTD __m128i _mm_mask_popcnt_epi32(__m128i, __mmask8, __m128i);
VPOPCNTD __m128i _mm_maskz_popcnt_epi32(__mmask8, __m128i);
VPOPCNTD __m256i _mm256_popcnt_epi32(__m256i);
VPOPCNTD __m256i _mm256_mask_popcnt_epi32(__m256i, __mmask8, __m256i);
VPOPCNTD __m256i _mm256_maskz_popcnt_epi32(__mmask8, __m256i);
VPOPCNTD __m512i _mm512_popcnt_epi32(__m512i);
VPOPCNTD __m512i _mm512_mask_popcnt_epi32(__m512i, __mmask16, __m512i);
VPOPCNTD __m512i _mm512_maskz_popcnt_epi32(__mmask16, __m512i);
VPOPCNTB __m128i _mm_popcnt_epi8(__m128i);
VPOPCNTB __m128i _mm_mask_popcnt_epi8(__m128i, __mmask16, __m128i);
VPOPCNTB __m128i _mm_maskz_popcnt_epi8(__mmask16, __m128i);
VPOPCNTB __m256i _mm256_popcnt_epi8(__m256i);
VPOPCNTB __m256i _mm256_mask_popcnt_epi8(__m256i, __mmask32, __m256i);
VPOPCNTB __m256i _mm256_maskz_popcnt_epi8(__mmask32, __m256i);
VPOPCNTB __m512i _mm512_popcnt_epi8(__m512i);
VPOPCNTB __m512i _mm512_mask_popcnt_epi8(__m512i, __mmask64, __m512i);
VPOPCNTB __m512i _mm512_maskz_popcnt_epi8(__mmask64, __m512i);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Type E4.

## VPSHLD — Concatenate and Shift Packed Data Left Logical

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| EVEX.NDS.128.66.0F3A.W1 70 /r /ib VPSHLDW xmm1{k1}{z}, xmm2, xmm3/m128, imm8 | A | V/V | AVX512_VBMI2 AVX512VL | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1. |
| EVEX.NDS.256.66.0F3A.W1 70 /r /ib VPSHLDW ymm1{k1}{z}, ymm2, ymm3/m256, imm8 | A | V/V | AVX512_VBMI2 AVX512VL | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1. |
| EVEX.NDS.512.66.0F3A.W1 70 /r /ib VPSHLDW zmm1{k1}{z}, zmm2, zmm3/m512, imm8 | A | V/V | AVX512_VBMI2 | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1. |
| EVEX.NDS.128.66.0F3A.W0 71 /r /ib VPSHLDD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8 | B | V/V | AVX512_VBMI2 AVX512VL | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1. |
| EVEX.NDS.256.66.0F3A.W0 71 /r /ib VPSHLDD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8 | B | V/V | AVX512_VBMI2 AVX512VL | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1. |
| EVEX.NDS.512.66.0F3A.W0 71 /r /ib VPSHLDD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8 | B | V/V | AVX512_VBMI2 | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1. |
| EVEX.NDS.128.66.0F3A.W1 71 /r /ib VPSHLDQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8 | B | V/V | AVX512_VBMI2 AVX512VL | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1. |
| EVEX.NDS.256.66.0F3A.W1 71 /r /ib VPSHLDQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8 | B | V/V | AVX512_VBMI2 AVX512VL | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1. |
| EVEX.NDS.512.66.0F3A.W1 71 /r /ib VPSHLDQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8 | B | V/V | AVX512_VBMI2 | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | Full Mem | ModRM:reg (w) | EVEX.vvvv | ModRM:r/m (r) | NA |
| B | Full | ModRM:reg (w) | EVEX.vvvv | ModRM:r/m (r) | NA |

### Description

Concatenate packed data, extract result shifted to the left by constant value.

This instruction supports memory fault suppression.

### Operation

**VPSHLDW DEST, SRC2, SRC3, imm8**

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        tmp ← concat(SRC2.word[j], SRC3.word[j]) << (imm8 & 15)
        DEST.word[j] ← tmp.word[1]
    ELSE IF *zeroing*:
        DEST.word[j] ← 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

**VPSHLDD DEST, SRC2, SRC3, imm8**

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 ← SRC3.dword[0]
    ELSE:
        tsrc3 ← SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp ← concat(SRC2.dword[j], tsrc3) << (imm8 & 31)
        DEST.dword[j] ← tmp.dword[1]
    ELSE IF *zeroing*:
        DEST.dword[j] ← 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

**VPSHLDQ DEST, SRC2, SRC3, imm8**

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 ← SRC3.qword[0]
    ELSE:
        tsrc3 ← SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp ← concat(SRC2.qword[j], tsrc3) << (imm8 & 63)
        DEST.qword[j] ← tmp.qword[1]
    ELSE IF *zeroing*:
        DEST.qword[j] ← 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VPSHLDD __m128i _mm_shldi_epi32(__m128i, __m128i, int);
VPSHLDD __m128i _mm_mask_shldi_epi32(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDD __m128i _mm_maskz_shldi_epi32(__mmask8, __m128i, __m128i, int);
VPSHLDD __m256i _mm256_shldi_epi32(__m256i, __m256i, int);
VPSHLDD __m256i _mm256_mask_shldi_epi32(__m256i, __mmask8, __m256i, __m256i, int);
VPSHLDD __m256i _mm256_maskz_shldi_epi32(__mmask8, __m256i, __m256i, int);
VPSHLDD __m512i _mm512_shldi_epi32(__m512i, __m512i, int);
VPSHLDD __m512i _mm512_mask_shldi_epi32(__m512i, __mmask16, __m512i, __m512i, int);
VPSHLDD __m512i _mm512_maskz_shldi_epi32(__mmask16, __m512i, __m512i, int);
VPSHLDQ __m128i _mm_shldi_epi64(__m128i, __m128i, int);
VPSHLDQ __m128i _mm_mask_shldi_epi64(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDQ __m128i _mm_maskz_shldi_epi64(__mmask8, __m128i, __m128i, int);
VPSHLDQ __m256i _mm256_shldi_epi64(__m256i, __m256i, int);
VPSHLDQ __m256i _mm256_mask_shldi_epi64(__m256i, __mmask8, __m256i, __m256i, int);
VPSHLDQ __m256i _mm256_maskz_shldi_epi64(__mmask8, __m256i, __m256i, int);
VPSHLDQ __m512i _mm512_shldi_epi64(__m512i, __m512i, int);
VPSHLDQ __m512i _mm512_mask_shldi_epi64(__m512i, __mmask8, __m512i, __m512i, int);
VPSHLDQ __m512i _mm512_maskz_shldi_epi64(__mmask8, __m512i, __m512i, int);
VPSHLDW __m128i _mm_shldi_epi16(__m128i, __m128i, int);
VPSHLDW __m128i _mm_mask_shldi_epi16(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDW __m128i _mm_maskz_shldi_epi16(__mmask8, __m128i, __m128i, int);
VPSHLDW __m256i _mm256_shldi_epi16(__m256i, __m256i, int);
VPSHLDW __m256i _mm256_mask_shldi_epi16(__m256i, __mmask16, __m256i, __m256i, int);
VPSHLDW __m256i _mm256_maskz_shldi_epi16(__mmask16, __m256i, __m256i, int);
VPSHLDW __m512i _mm512_shldi_epi16(__m512i, __m512i, int);
VPSHLDW __m512i _mm512_mask_shldi_epi16(__m512i, __mmask32, __m512i, __m512i, int);
VPSHLDW __m512i _mm512_maskz_shldi_epi16(__mmask32, __m512i, __m512i, int);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Type E4.

## VPSHLDV — Concatenate and Variable Shift Packed Data Left Logical

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| EVEX.DDS.128.66.0F38.W1 70 /r VPSHLDVW xmm1{k1}{z}, xmm2, xmm3/m128 | A | V/V | AVX512_VBMI2 AVX512VL | Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1. |
| EVEX.DDS.256.66.0F38.W1 70 /r VPSHLDVW ymm1{k1}{z}, ymm2, ymm3/m256 | A | V/V | AVX512_VBMI2 AVX512VL | Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1. |
| EVEX.DDS.512.66.0F38.W1 70 /r VPSHLDVW zmm1{k1}{z}, zmm2, zmm3/m512 | A | V/V | AVX512_VBMI2 | Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1. |
| EVEX.DDS.128.66.0F38.W0 71 /r VPSHLDVD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst | B | V/V | AVX512_VBMI2 AVX512VL | Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1. |
| EVEX.DDS.256.66.0F38.W0 71 /r VPSHLDVD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst | B | V/V | AVX512_VBMI2 AVX512VL | Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1. |
| EVEX.DDS.512.66.0F38.W0 71 /r VPSHLDVD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst | B | V/V | AVX512_VBMI2 | Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1. |
| EVEX.DDS.128.66.0F38.W1 71 /r VPSHLDVQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst | B | V/V | AVX512_VBMI2 AVX512VL | Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1. |
| EVEX.DDS.256.66.0F38.W1 71 /r VPSHLDVQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst | B | V/V | AVX512_VBMI2 AVX512VL | Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1. |
| EVEX.DDS.512.66.0F38.W1 71 /r VPSHLDVQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst | B | V/V | AVX512_VBMI2 | Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | Full Mem | ModRM:reg (r, w) | EVEX.vvvv | ModRM:r/m (r) | NA |
| B | Full | ModRM:reg (r, w) | EVEX.vvvv | ModRM:r/m (r) | NA |

### Description

Concatenate packed data, extract result shifted to the left by variable value.

This instruction supports memory fault suppression.

### Operation

```
FUNCTION concat(a,b):
    IF words:
        d.word[1] ← a
        d.word[0] ← b
        return d
    ELSE IF dwords:
        q.dword[1] ← a
        q.dword[0] ← b
        return q
    ELSE IF qwords:
        o.qword[1] ← a
        o.qword[0] ← b
        return o
```

**VPSHLDVW DEST, SRC2, SRC3**

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        tmp ← concat(DEST.word[j], SRC2.word[j]) << (SRC3.word[j] & 15)
        DEST.word[j] ← tmp.word[1]
    ELSE IF *zeroing*:
        DEST.word[j] ← 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

**VPSHLDVD DEST, SRC2, SRC3**

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 ← SRC3.dword[0]
    ELSE:
        tsrc3 ← SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp ← concat(DEST.dword[j], SRC2.dword[j]) << (tsrc3 & 31)
        DEST.dword[j] ← tmp.dword[1]
    ELSE IF *zeroing*:
        DEST.dword[j] ← 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

**VPSHLDVQ DEST, SRC2, SRC3**

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 ← SRC3.qword[0]
    ELSE:
        tsrc3 ← SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp ← concat(DEST.qword[j], SRC2.qword[j]) << (tsrc3 & 63)
        DEST.qword[j] ← tmp.qword[1]
    ELSE IF *zeroing*:
        DEST.qword[j] ← 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VPSHLDVW __m128i _mm_shldv_epi16(__m128i, __m128i, __m128i);
VPSHLDVW __m128i _mm_mask_shldv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVW __m128i _mm_maskz_shldv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVW __m256i _mm256_shldv_epi16(__m256i, __m256i, __m256i);
VPSHLDVW __m256i _mm256_mask_shldv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHLDVW __m256i _mm256_maskz_shldv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHLDVQ __m512i _mm512_shldv_epi64(__m512i, __m512i, __m512i);
VPSHLDVQ __m512i _mm512_mask_shldv_epi64(__m512i, __mmask8, __m512i, __m512i);
VPSHLDVQ __m512i _mm512_maskz_shldv_epi64(__mmask8, __m512i, __m512i, __m512i);
VPSHLDVW __m128i _mm_shldv_epi16(__m128i, __m128i, __m128i);
VPSHLDVW __m128i _mm_mask_shldv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVW __m128i _mm_maskz_shldv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVW __m256i _mm256_shldv_epi16(__m256i, __m256i, __m256i);
VPSHLDVW __m256i _mm256_mask_shldv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHLDVW __m256i _mm256_maskz_shldv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHLDVW __m512i _mm512_shldv_epi16(__m512i, __m512i, __m512i);
VPSHLDVW __m512i _mm512_mask_shldv_epi16(__m512i, __mmask32, __m512i, __m512i);
VPSHLDVW __m512i _mm512_maskz_shldv_epi16(__mmask32, __m512i, __m512i, __m512i);
VPSHLDVD __m128i _mm_shldv_epi32(__m128i, __m128i, __m128i);
VPSHLDVD __m128i _mm_mask_shldv_epi32(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVD __m128i _mm_maskz_shldv_epi32(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVD __m256i _mm256_shldv_epi32(__m256i, __m256i, __m256i);
VPSHLDVD __m256i _mm256_mask_shldv_epi32(__m256i, __mmask8, __m256i, __m256i);
VPSHLDVD __m256i _mm256_maskz_shldv_epi32(__mmask8, __m256i, __m256i, __m256i);
VPSHLDVD __m512i _mm512_shldv_epi32(__m512i, __m512i, __m512i);
VPSHLDVD __m512i _mm512_mask_shldv_epi32(__m512i, __mmask16, __m512i, __m512i);
VPSHLDVD __m512i _mm512_maskz_shldv_epi32(__mmask16, __m512i, __m512i, __m512i);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Type E4.

## VPSHRD — Concatenate and Shift Packed Data Right Logical

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| EVEX.NDS.128.66.0F3A.W1 72 /r /ib VPSHRDW xmm1{k1}{z}, xmm2, xmm3/m128, imm8 | A | V/V | AVX512_VBMI2 AVX512VL | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1. |
| EVEX.NDS.256.66.0F3A.W1 72 /r /ib VPSHRDW ymm1{k1}{z}, ymm2, ymm3/m256, imm8 | A | V/V | AVX512_VBMI2 AVX512VL | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1. |
| EVEX.NDS.512.66.0F3A.W1 72 /r /ib VPSHRDW zmm1{k1}{z}, zmm2, zmm3/m512, imm8 | A | V/V | AVX512_VBMI2 | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1. |
| EVEX.NDS.128.66.0F3A.W0 73 /r /ib VPSHRDD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8 | B | V/V | AVX512_VBMI2 AVX512VL | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1. |
| EVEX.NDS.256.66.0F3A.W0 73 /r /ib VPSHRDD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8 | B | V/V | AVX512_VBMI2 AVX512VL | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1. |
| EVEX.NDS.512.66.0F3A.W0 73 /r /ib VPSHRDD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8 | B | V/V | AVX512_VBMI2 | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1. |
| EVEX.NDS.128.66.0F3A.W1 73 /r /ib VPSHRDQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8 | B | V/V | AVX512_VBMI2 AVX512VL | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1. |
| EVEX.NDS.256.66.0F3A.W1 73 /r /ib VPSHRDQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8 | B | V/V | AVX512_VBMI2 AVX512VL | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1. |
| EVEX.NDS.512.66.0F3A.W1 73 /r /ib VPSHRDQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8 | B | V/V | AVX512_VBMI2 | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | Full Mem | ModRM:reg (w) | EVEX.vvvv | ModRM:r/m (r) | NA |
| B | Full | ModRM:reg (w) | EVEX.vvvv | ModRM:r/m (r) | NA |

### Description

Concatenate packed data, extract result shifted to the right by constant value.

This instruction supports memory fault suppression.

### Operation

**VPSHRDW DEST, SRC2, SRC3, imm8**

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.word[j] ← concat(SRC3.word[j], SRC2.word[j]) >> (imm8 & 15)
    ELSE IF *zeroing*:
        DEST.word[j] ← 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

**VPSHRDD DEST, SRC2, SRC3, imm8**

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 ← SRC3.dword[0]
    ELSE:
        tsrc3 ← SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.dword[j] ← concat(tsrc3, SRC2.dword[j]) >> (imm8 & 31)
    ELSE IF *zeroing*:
        DEST.dword[j] ← 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

**VPSHRDQ DEST, SRC2, SRC3, imm8**

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 ← SRC3.qword[0]
    ELSE:
        tsrc3 ← SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.qword[j] ← concat(tsrc3, SRC2.qword[j]) >> (imm8 & 63)
    ELSE IF *zeroing*:
        DEST.qword[j] ← 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VPSHRDQ __m128i _mm_shrdi_epi64(__m128i, __m128i, int);
VPSHRDQ __m128i _mm_mask_shrdi_epi64(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDQ __m128i _mm_maskz_shrdi_epi64(__mmask8, __m128i, __m128i, int);
VPSHRDQ __m256i _mm256_shrdi_epi64(__m256i, __m256i, int);
VPSHRDQ __m256i _mm256_mask_shrdi_epi64(__m256i, __mmask8, __m256i, __m256i, int);
VPSHRDQ __m256i _mm256_maskz_shrdi_epi64(__mmask8, __m256i, __m256i, int);
VPSHRDQ __m512i _mm512_shrdi_epi64(__m512i, __m512i, int);
VPSHRDQ __m512i _mm512_mask_shrdi_epi64(__m512i, __mmask8, __m512i, __m512i, int);
VPSHRDQ __m512i _mm512_maskz_shrdi_epi64(__mmask8, __m512i, __m512i, int);
VPSHRDD __m128i _mm_shrdi_epi32(__m128i, __m128i, int);
VPSHRDD __m128i _mm_mask_shrdi_epi32(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDD __m128i _mm_maskz_shrdi_epi32(__mmask8, __m128i, __m128i, int);
VPSHRDD __m256i _mm256_shrdi_epi32(__m256i, __m256i, int);
VPSHRDD __m256i _mm256_mask_shrdi_epi32(__m256i, __mmask8, __m256i, __m256i, int);
VPSHRDD __m256i _mm256_maskz_shrdi_epi32(__mmask8, __m256i, __m256i, int);
VPSHRDD __m512i _mm512_shrdi_epi32(__m512i, __m512i, int);
VPSHRDD __m512i _mm512_mask_shrdi_epi32(__m512i, __mmask16, __m512i, __m512i, int);
VPSHRDD __m512i _mm512_maskz_shrdi_epi32(__mmask16, __m512i, __m512i, int);
VPSHRDW __m128i _mm_shrdi_epi16(__m128i, __m128i, int);
VPSHRDW __m128i _mm_mask_shrdi_epi16(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDW __m128i _mm_maskz_shrdi_epi16(__mmask8, __m128i, __m128i, int);
VPSHRDW __m256i _mm256_shrdi_epi16(__m256i, __m256i, int);
VPSHRDW __m256i _mm256_mask_shrdi_epi16(__m256i, __mmask16, __m256i, __m256i, int);
VPSHRDW __m256i _mm256_maskz_shrdi_epi16(__mmask16, __m256i, __m256i, int);
VPSHRDW __m512i _mm512_shrdi_epi16(__m512i, __m512i, int);
VPSHRDW __m512i _mm512_mask_shrdi_epi16(__m512i, __mmask32, __m512i, __m512i, int);
VPSHRDW __m512i _mm512_maskz_shrdi_epi16(__mmask32, __m512i, __m512i, int);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Type E4.

## VPSHRDV — Concatenate and Variable Shift Packed Data Right Logical

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| EVEX.DDS.128.66.0F38.W1 72 /r VPSHRDVW xmm1{k1}{z}, xmm2, xmm3/m128 | A | V/V | AVX512_VBMI2 AVX512VL | Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1. |
| EVEX.DDS.256.66.0F38.W1 72 /r VPSHRDVW ymm1{k1}{z}, ymm2, ymm3/m256 | A | V/V | AVX512_VBMI2 AVX512VL | Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1. |
| EVEX.DDS.512.66.0F38.W1 72 /r VPSHRDVW zmm1{k1}{z}, zmm2, zmm3/m512 | A | V/V | AVX512_VBMI2 | Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1. |
| EVEX.DDS.128.66.0F38.W0 73 /r VPSHRDVD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst | B | V/V | AVX512_VBMI2 AVX512VL | Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1. |
| EVEX.DDS.256.66.0F38.W0 73 /r VPSHRDVD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst | B | V/V | AVX512_VBMI2 AVX512VL | Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1. |
| EVEX.DDS.512.66.0F38.W0 73 /r VPSHRDVD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst | B | V/V | AVX512_VBMI2 | Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1. |
| EVEX.DDS.128.66.0F38.W1 73 /r VPSHRDVQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst | B | V/V | AVX512_VBMI2 AVX512VL | Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1. |
| EVEX.DDS.256.66.0F38.W1 73 /r VPSHRDVQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst | B | V/V | AVX512_VBMI2 AVX512VL | Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1. |
| EVEX.DDS.512.66.0F38.W1 73 /r VPSHRDVQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst | B | V/V | AVX512_VBMI2 | Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | Full Mem | ModRM:reg (r, w) | EVEX.vvvv | ModRM:r/m (r) | NA |
| B | Full | ModRM:reg (r, w) | EVEX.vvvv | ModRM:r/m (r) | NA |

### Description

Concatenate packed data, extract result shifted to the right by variable value.

This instruction supports memory fault suppression.

### Operation

**VPSHRDVW DEST, SRC2, SRC3**

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.word[j] ← concat(SRC2.word[j], DEST.word[j]) >> (SRC3.word[j] & 15)
    ELSE IF *zeroing*:
        DEST.word[j] ← 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

**VPSHRDVD DEST, SRC2, SRC3**

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 ← SRC3.dword[0]
    ELSE:
        tsrc3 ← SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.dword[j] ← concat(SRC2.dword[j], DEST.dword[j]) >> (tsrc3 & 31)
    ELSE IF *zeroing*:
        DEST.dword[j] ← 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

**VPSHRDVQ DEST, SRC2, SRC3**

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 ← SRC3.qword[0]
    ELSE:
        tsrc3 ← SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.qword[j] ← concat(SRC2.qword[j], DEST.qword[j]) >> (tsrc3 & 63)
    ELSE IF *zeroing*:
        DEST.qword[j] ← 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VPSHRDVQ __m128i _mm_shrdv_epi64(__m128i, __m128i, __m128i);
VPSHRDVQ __m128i _mm_mask_shrdv_epi64(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVQ __m128i _mm_maskz_shrdv_epi64(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVQ __m256i _mm256_shrdv_epi64(__m256i, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_mask_shrdv_epi64(__m256i, __mmask8, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_maskz_shrdv_epi64(__mmask8, __m256i, __m256i, __m256i);
VPSHRDVQ __m512i _mm512_shrdv_epi64(__m512i, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_mask_shrdv_epi64(__m512i, __mmask8, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_maskz_shrdv_epi64(__mmask8, __m512i, __m512i, __m512i);
VPSHRDVD __m128i _mm_shrdv_epi32(__m128i, __m128i, __m128i);
VPSHRDVD __m128i _mm_mask_shrdv_epi32(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVD __m128i _mm_maskz_shrdv_epi32(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVD __m256i _mm256_shrdv_epi32(__m256i, __m256i, __m256i);
VPSHRDVD __m256i _mm256_mask_shrdv_epi32(__m256i, __mmask8, __m256i, __m256i);
VPSHRDVD __m256i _mm256_maskz_shrdv_epi32(__mmask8, __m256i, __m256i, __m256i);
VPSHRDVD __m512i _mm512_shrdv_epi32(__m512i, __m512i, __m512i);
VPSHRDVD __m512i _mm512_mask_shrdv_epi32(__m512i, __mmask16, __m512i, __m512i);
VPSHRDVD __m512i _mm512_maskz_shrdv_epi32(__mmask16, __m512i, __m512i, __m512i);
VPSHRDVW __m128i _mm_shrdv_epi16(__m128i, __m128i, __m128i);
VPSHRDVW __m128i _mm_mask_shrdv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVW __m128i _mm_maskz_shrdv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVW __m256i _mm256_shrdv_epi16(__m256i, __m256i, __m256i);
VPSHRDVW __m256i _mm256_mask_shrdv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHRDVW __m256i _mm256_maskz_shrdv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHRDVW __m512i _mm512_shrdv_epi16(__m512i, __m512i, __m512i);
VPSHRDVW __m512i _mm512_mask_shrdv_epi16(__m512i, __mmask32, __m512i, __m512i);
VPSHRDVW __m512i _mm512_maskz_shrdv_epi16(__mmask32, __m512i, __m512i, __m512i);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Type E4.

## VPSHUFBITQMB — Shuffle Bits from Quadword Elements Using Byte Indexes into Mask

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| EVEX.NDS.128.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, xmm2, xmm3/m128 | A | V/V | AVX512_BITALG AVX512VL | Extract values in xmm2 using control bits of xmm3/m128 with writemask k2 and leave the result in mask register k1. |
| EVEX.NDS.256.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, ymm2, ymm3/m256 | A | V/V | AVX512_BITALG AVX512VL | Extract values in ymm2 using control bits of ymm3/m256 with writemask k2 and leave the result in mask register k1. |
| EVEX.NDS.512.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, zmm2, zmm3/m512 | A | V/V | AVX512_BITALG | Extract values in zmm2 using control bits of zmm3/m512 with writemask k2 and leave the result in mask register k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | Full Mem | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |

### Description

The VPSHUFBITQMB instruction performs a bit gather select using second source as control and first source as data. Each bit uses 6 control bits (2nd source operand) to select which data bit is going to be gathered (first source operand). A given bit can only access 64 different bits of data (first 64 destination bits can access first 64 data bits, second 64 destination bits can access second 64 data bits, etc.).

Control data for each output bit is stored in 8 bit elements of SRC2, but only the 6 least significant bits of each element are used.

This instruction uses write masking (zeroing only). This instruction supports memory fault suppression.

The first source operand is a ZMM register. The second source operand is a ZMM register or a memory location. The destination operand is a mask register.

### Operation

**VPSHUFBITQMB DEST, SRC1, SRC2**

```
(KL, VL) = (16,128), (32,256), (64, 512)
FOR i ← 0 TO KL/8-1:                    //Qword
    FOR j ← 0 to 7:                     // Byte
        IF k2[i*8+j] or *no writemask*:
            m ← SRC2.qword[i].byte[j] & 0x3F
            k1[i*8+j] ← SRC1.qword[i].bit[m]
        ELSE:
            k1[i*8+j] ← 0
k1[MAX_KL-1:KL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFBITQMB __mmask16 _mm_bitshuffle_epi64_mask(__m128i, __m128i);
VPSHUFBITQMB __mmask16 _mm_mask_bitshuffle_epi64_mask(__mmask16, __m128i, __m128i);
VPSHUFBITQMB __mmask32 _mm256_bitshuffle_epi64_mask(__m256i, __m256i);
VPSHUFBITQMB __mmask32 _mm256_mask_bitshuffle_epi64_mask(__mmask32, __m256i, __m256i);
VPSHUFBITQMB __mmask64 _mm512_bitshuffle_epi64_mask(__m512i, __m512i);
VPSHUFBITQMB __mmask64 _mm512_mask_bitshuffle_epi64_mask(__mmask64, __m512i, __m512i);

# V4FMADDPS/V4FNMADDPS — Packed Single-Precision Floating-Point Fused Multiply-Add (4-iterations)

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| EVEX.DDS.512.F2.0F38.W0 9A /r V4FMADDPS zmm1{k1}{z}, zmm2+3, m128 | A | V/V | AVX512_4FMAPS | Multiply packed single-precision floating-point values from source register block indicated by zmm2 by values from m128 and accumulate the result in zmm1. |
| EVEX.DDS.512.F2.0F38.W0 AA /r V4FNMADDPS zmm1{k1}{z}, zmm2+3, m128 | A | V/V | AVX512_4FMAPS | Multiply and negate packed single-precision floating-point values from source register block indicated by zmm2 by values from m128 and accumulate the result in zmm1. |

## Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | Tuple1_4X | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |

### Description

This instruction computes 4 sequential packed fused single-precision floating-point multiply-add instructions with a sequentially selected memory operand in each of the four steps.

In the above box, the notation of "+3" is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any of the 16 lowest significant mask bits is set to 1 or if a "no masking" encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

Rounding is performed at every FMA (fused multiply and add) boundary. Exceptions are also taken sequentially. Pre- and post-computational exceptions of the first FMA take priority over the pre- and post-computational exceptions of the second FMA, etc.

### Operation

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

```
define NFMA_PS(kl, vl, dest, k1, msrc, regs_loaded, src_base, posneg):
    tmpdest ← dest

    // reg[] is an array representing the SIMD register file.
    FOR j ← 0 to regs_loaded-1:
        FOR i ← 0 to kl-1:
            IF k1[i] or *no writemask*:
                IF posneg = 0:
                    tmpdest.single[i] ← RoundFPControl_MXCSR(tmpdest.single[i] - reg[src_base + j ].single[i] * msrc.single[j])
            ELSE:
                    tmpdest.single[i] ← RoundFPControl_MXCSR(tmpdest.single[i] + reg[src_base + j ].single[i] * msrc.single[j])
        ELSE IF *zeroing*:
            tmpdest.single[i] ← 0
    dest ← tmpdst
    dest[MAX_VL-1:VL] ← 0

V4FMADDPS and V4FNMADDPS dest{k1}, src1, msrc (AVX512)
KL, VL = (16,512)

regs_loaded ← 4
src_base ← src_reg_id & ~3 // for src1 operand
posneg ← 0 if negative form, 1 otherwise
NFMA_PS(kl, vl, dest, k1, msrc, regs_loaded, src_base, posneg)
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
V4FMADDPS __m512 _mm512_4fmadd_ps( __m512, __m512x4, __m128 *);
V4FMADDPS __m512 _mm512_mask_4fmadd_ps(__m512, __mmask16, __m512x4, __m128 *);
V4FMADDPS __m512 _mm512_maskz_4fmadd_ps(__mmask16, __m512, __m512x4, __m128 *);
V4FNMADDPS __m512 _mm512_4fnmadd_ps(__m512, __m512x4, __m128 *);
V4FNMADDPS __m512 _mm512_mask_4fnmadd_ps(__m512, __mmask16, __m512x4, __m128 *);
V4FNMADDPS __m512 _mm512_maskz_4fnmadd_ps(__mmask16, __m512, __m512x4, __m128 *);
```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Other Exceptions

See Type E2; additionally

| | |
|---|---|
| #UD | If the EVEX broadcast bit is set to 1. |
| #UD | If the MODRM.mod = 0b11. |

# V4FMADDSS/V4FNMADDSS —Scalar Single-Precision Floating-Point Fused Multiply-Add (4-iterations)

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| EVEX.DDS.LLIG.F2.0F38.W0 9B /r V4FMADDSS xmm1{k1}{z}, xmm2+3, m128 | A | V/V | AVX512_4FMAPS | Multiply scalar single-precision floating-point values from source register block indicated by xmm2 by values from m128 and accumulate the result in xmm1. |
| EVEX.DDS.LLIG.F2.0F38.W0 AB /r V4FNMADDSS xmm1{k1}{z}, xmm2+3, m128 | A | V/V | AVX512_4FMAPS | Multiply and negate scalar single-precision floating-point values from source register block indicated by xmm2 by values from m128 and accumulate the result in xmm1. |

## Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | Tuple1_4X | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |

### Description

This instruction computes 4 sequential scalar fused single-precision floating-point multiply-add instructions with a sequentially selected memory operand in each of the four steps.

In the above box, the notation of "+3" is used to denote that the instruction accesses 4 source registers based that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if the least significant mask bit is set to 1 or if a "no masking" encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

Rounding is performed at every FMA boundary. Exceptions are also taken sequentially. Pre- and post-computational exceptions of the first FMA take priority over the pre- and post-computational exceptions of the second FMA, etc.

### Operation

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

```
define NFMA_SS(vl, dest, k1, msrc, regs_loaded, src_base, posneg):
    tmpdest ← dest
    // reg[] is an array representing the SIMD register file.
    IF k1[0] or *no writemask*:
        FOR j ← 0 to regs_loaded - 1:
            IF posneg = 0:
                tmpdest.single[0] ← RoundFPControl_MXCSR(tmpdest.single[0] - reg[src_base + j ].single[0] * msrc.single[j])
            ELSE:
                tmpdest.single[0] ← RoundFPControl_MXCSR(tmpdest.single[0] + reg[src_base + j ].single[0] * msrc.single[j])
    ELSE IF *zeroing*:
        tmpdest.single[0] ← 0
    dest ← tmpdst
    dest[MAX_VL-1:VL] ← 0
```

```
V4FMADDSS and V4FNMADDSS dest{k1}, src1, msrc (AVX512)
VL = 128

regs_loaded ← 4
src_base ← src_reg_id & ~3 // for src1 operand
posneg ← 0 if negative form, 1 otherwise
NFMA_SS(vl, dest, k1, msrc, regs_loaded, src_base, posneg)
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
V4FMADDSS __m128 _mm_4fmadd_ss(__m128, __m128x4, __m128 *);
V4FMADDSS __m128 _mm_mask_4fmadd_ss(__m128, __mmask8, __m128x4, __m128 *);
V4FMADDSS __m128 _mm_maskz_4fmadd_ss(__mmask8, __m128, __m128x4, __m128 *);
V4FNMADDSS __m128 _mm_4fnmadd_ss(__m128, __m128x4, __m128 *);
V4FNMADDSS __m128 _mm_mask_4fnmadd_ss(__m128, __mmask8, __m128x4, __m128 *);
V4FNMADDSS __m128 _mm_maskz_4fnmadd_ss(__mmask8, __m128, __m128x4, __m128 *);
```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Other Exceptions

See Type E2; additionally

| | |
|---|---|
| #UD | If the EVEX broadcast bit is set to 1. |
| #UD | If the MODRM.mod = 0b11. |

# VP4DPWSSDS — Dot Product of Signed Words with Dword Accumulation and Saturation (4-iterations)

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| EVEX.DDS.512.F2.0F38.W0 53 /r VP4DPWSSDS zmm1{k1}{z}, zmm2+3, m128 | A | V/V | AVX512_4VNNIW | Multiply signed words from source register block indicated by zmm2 by signed words from m128 and accumulate the resulting dword results with signed saturation in zmm1. |

## Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | Tuple1_4X | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |

### Description

This instruction computes 4 sequential register source-block dot-products of two signed word operands with doubleword accumulation and signed saturation. The memory operand is sequentially selected in each of the four steps.

In the above box, the notation of "+3" is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any bit of the lowest 16-bits of the mask is set to 1 or if a "no masking" encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

### Operation

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

```
VP4DPWSSDS dest, src1, src2
(KL,VL) = (16,512)
N ← 4

ORIGDEST ← DEST
src_base ← src_reg_id & ~ (N-1) // for src1 operand

FOR i ← 0 to KL-1:
    IF k1[i] or *no writemask*:
        FOR m ← 0 to N-1:
            t ← SRC2.dword[m]
            p1dword ← reg[src_base+m].word[2*i] * t.word[0]
            p2dword ← reg[src_base+m].word[2*i+1] * t.word[1]
            DEST.dword[i] ← SIGNED_DWORD_SATURATE(DEST.dword[i] + p1dword + p2dword)
    ELSE IF *zeroing*:
        DEST.dword[i] ← 0
    ELSE
        DEST.dword[i] ← ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VP4DPWSSDS __m512i _mm512_4dpwssds_epi32(__m512i, __m512ix4, __m128i *);
VP4DPWSSDS __m512i _mm512_mask_4dpwssds_epi32(__m512i, __mmask16, __m512ix4, __m128i *);
VP4DPWSSDS __m512i _mm512_maskz_4dpwssds_epi32(__mmask16, __m512i, __m512ix4, __m128i *);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Type E4; additionally

| | |
|---|---|
| #UD | If the EVEX broadcast bit is set to 1. |
| #UD | If the MODRM.mod = 0b11. |

## VP4DPWSSD — Dot Product of Signed Words with Dword Accumulation (4-iterations)

| Opcode/ Instruction | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
|---|---|---|---|---|
| EVEX.DDS.512.F2.0F38.W0 52 /r VP4DPWSSD zmm1{k1}{z}, zmm2+3, m128 | A | V/V | AVX512_4VNNIW | Multiply signed words from source register block indicated by zmm2 by signed words from m128 and accumulate resulting signed dwords in zmm1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|---|
| A | Tuple1_4X | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | NA |

### Description

This instruction computes 4 sequential register source-block dot-products of two signed word operands with doubleword accumulation; see Figure 3-1 below. The memory operand is sequentially selected in each of the four steps.

In the above box, the notation of "+3"' is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any bit of the lowest 16-bits of the mask is set to 1 or if a "no masking" encoding is used.

The tuple type Tuple1_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

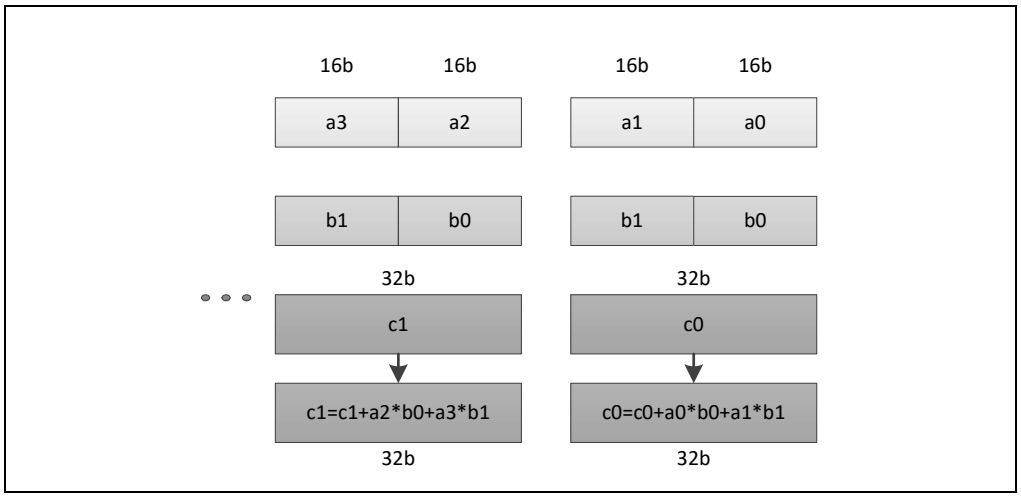


**Figure 3-1. Register Source-Block Dot Product of Two Signed Word Operands with Doubleword Accumulation[1]**

NOTES:

1. For illustration purposes, one source-block dot product instance is shown out of the four.

### Operation

src_reg_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

```
VP4DPWSSD dest, src1, src2
(KL,VL) = (16,512)
N ← 4

ORIGDEST ← DEST
src_base ← src_reg_id & ~ (N-1) // for src1 operand

FOR i ← 0 to KL-1:
    IF k1[i] or *no writemask*:
        FOR m ← 0 to N-1:
            t ← SRC2.dword[m]
            p1dword ← reg[src_base+m].word[2*i] * t.word[0]
            p2dword ← reg[src_base+m].word[2*i+1] * t.word[1]
            DEST.dword[i] ← DEST.dword[i] + p1dword + p2dword
    ELSE IF *zeroing*:
        DEST.dword[i] ← 0
    ELSE
        DEST.dword[i] ← ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VP4DPWSSD __m512i _mm512_4dpwssd_epi32(__m512i, __m512ix4, __m128i *);
VP4DPWSSD __m512i _mm512_mask_4dpwssd_epi32(__m512i, __mmask16, __m512ix4, __m128i *);
VP4DPWSSD __m512i _mm512_maskz_4dpwssd_epi32(__mmask16, __m512i, __m512ix4, __m128i *);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Type E4; additionally

| | |
|---|---|
| #UD | If the EVEX broadcast bit is set to 1. |
| #UD | If the MODRM.mod = 0b11. |