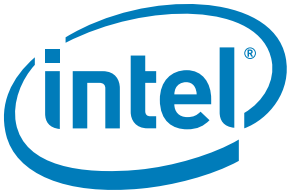

Intel® Architecture Instruction Set Extensions and Future Features Programming Reference

319433-030

OCTOBER 2017

Intel technologies features and benefits depend on system configuration and may require enabled hardware, software, or service activation. Learn more at intel.com, or from the OEM or retailer.

No computer system can be absolutely secure. Intel does not assume any liability for lost or stolen data or systems or any damages resulting from such losses.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

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

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Intel does not guarantee the availability of these interfaces in any future product. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting http://www.intel.com/design/literature.htm.

Intel, the Intel logo, Intel Atom, Intel Core, Intel SpeedStep, MMX, Pentium, VTune, and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Revision History

Revision	Description	Date
-025	• Removed instructions that now reside in the Intel® 64 and IA-32 Architectures Software Developer's Manual. • Minor updates to chapter 1. • Updates to Table 2-1, Table 2-2 and Table 2-8 (leaf 07H) to indicate support for AVX512_4VNNIW and AVX512_4FMAPS. • Minor update to Table 2-8 (leaf 15H) regarding ECX definition. • Minor updates to Section 4.6.2 and Section 4.6.3 to clarify the effects of "suppress all exceptions". • Footnote addition to CLWB instruction indicating operand encoding requirement. • Removed PCOMMIT.	September 2016
-026	• Removed CLWB instruction; it now resides in the Intel® 64 and IA-32 Architectures Software Developer's Manual. • Added additional 512-bit instruction extensions in chapter 6.	October 2016
-027	• Added TLB CPUID leaf in chapter 2. • Added VPOPCNTD/Q instruction in chapter 6,and CPUID details in chapter 2.	December 2016
-028	• Updated intrinsics for VPOPCNTD/Q instruction in chapter 6.	December 2016
-029	• Corrected typo in CPUID leaf 18H. • Updated operand encoding table format; extracted tuple information from operand encoding. • Added VPERMB back into chapter 5; inadvertently removed. • Moved all instructions from chapter 6 to chapter 5. • Updated operation section of VPMULTISHIFTQB.	April 2017
-030	• Removed unnecessary information from document (chapters 2, 3 and 4). • Added table listing recent instruction set extensions introduction in Intel 64 and IA-32 Processors. • Updated CPUID instruction with additional details. • Added the following instructions: GF2P8AFFINEINVQB, GF2P8AFFINEQB, GF2P8MULB, VAESDEC, VAESDECLAST, VAESENC, VAESENCLAST, VPCLMULQDQ, VPCOMPRESS, VPDPBUSD, VPDPBUSDS, VPDPWUU/VPDPWUS/VPDPWSU/VPDPWSS, VPDPWSSD, VPDPWSSDS, VPEXPAND, VPOPCNT, VPSHLD, VPSHLDV, VPSHRD, VPSHRDV, VPSHUFBITQMB. • Removed the following instructions: VPMADD52HUQ, VPMADD52LUQ, VPERMB, VPERMI2B, VPERMT2B, and VPMULTISHIFTQB. They can be found in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D. • Moved instructions unique to processors based on the Knights Mill microarchitecture to chapter 3. • Added chapter 4: EPT-Based Sub-Page Permissions. • Added chapter 5: Intel® Processor Trace: VMX Improvements.	October 2017

REVISION HISTORY

CHAPTER 1
FUTURE INTEL® ARCHITECTURE INSTRUCTION EXTENSIONS AND FEATURES

1.1 About This Document 1-1
1.2 Instruction Set Extensions Introduction in Intel 64 and IA-32 Processors 1-1
1.3 CPUID Instruction 1-4
CPUID—CPU Identification 1-4
1.4 Compressed Displacement (disp8*N) Support in EVEX 1-38

CHAPTER 2
INSTRUCTION SET REFERENCE, A-Z

2.1 Instruction SET Reference 2-1
GF2P8AFFINEINVQB — Galois Field Affine Transformation Inverse 2-2
GF2P8AFFINEQB — Galois Field Affine Transformation 2-5
GF2P8MULB — Galois Field Multiply Bytes 2-8
VAESDEC — Perform One Round of an AES Decryption Flow 2-10
VAESDECLAST — Perform Last Round of an AES Decryption Flow 2-12
VAESENC — Perform One Round of an AES Encryption Flow 2-14
VAESENCLAST — Perform Last Round of an AES Encryption Flow 2-16
VPCLMULQDQ — Carry-Less Multiplication Quadword 2-18
VPCOMPRESS — Store Sparse Packed Byte/Word Integer Values into Dense Memory/Register 2-21
VPDPBUSD — Multiply and Add Unsigned and Signed Bytes 2-24
VPDPBUSDS — Multiply and Add Unsigned and Signed Bytes with Saturation 2-26
VPDPWSSD — Multiply and Add Signed Word Integers 2-28
VPDPWSSDS — Multiply and Add Word Integers with Saturation 2-30
VPEXPAND — Expand Byte/Word Values 2-32
VPOPCNT — Return the Count of Number of Bits Set to 1 in BYTE/WORD/DWORD/QWORD 2-35
VPSHLD — Concatenate and Shift Packed Data Left Logical 2-38
VPSHLDV — Concatenate and Variable Shift Packed Data Left Logical 2-41
VPSHRD — Concatenate and Shift Packed Data Right Logical 2-44
VPSHRDV — Concatenate and Variable Shift Packed Data Right Logical 2-47
VPSHUFBITQMB — Shuffle Bits from Quadword Elements Using Byte Indexes into Mask 2-50

CHAPTER 3
INSTRUCTION SET REFERENCE UNIQUE TO PROCESSORS BASED ON THE KNIGHTS MILL MICROARCHITECTURE

3.1 Instruction SET Reference 3-1
V4FMADDPS/V4FNMADDPS — Packed Single-Precision Floating-Point Fused Multiply-Add (4-iterations) 3-2
V4FMADDSS/V4FNMADDSS —Scalar Single-Precision Floating-Point Fused Multiply-Add (4-iterations) 3-4
VP4DPWSSDS — Dot Product of Signed Words with Dword Accumulation and Saturation (4-iterations) 3-6
VP4DPWSSD — Dot Product of Signed Words with Dword Accumulation (4-iterations) 3-8

CHAPTER 4
EPT-BASED SUB-PAGE PERMISSIONS

4.1 Introduction 4-1
4.2 VMCS Changes 4-1
4.3 Changes to EPT Paging-Structure Entries 4-1
4.4 Changes to Guest-Physical Accesses 4-1
4.5 Sub-Page Permission Table 4-2
4.5.1 SPPT Overview 4-2
4.5.2 Operation of SPPT-based Write-Permission 4-2
4.5.3 SPP-Induced VM Exits 4-4
4.5.3.1 Sub-Page Permissions and EPT Violations 4-4
4.5.4 Invalidating Cached SPP Permissions 4-5
4.5.5 Sub-Page Permission Interaction with Accessed and Dirty Flags for EPT 4-5
4.5.6 Sub-Page Permission Interaction with Intel® TSX 4-5

4.5.7 Sub-Page Permission Interaction with Intel® SGX 4-5
4.5.7.1 Fault Priorities 4-5
4.5.8 Memory Type Used for Accessing SPPT 4-6
4.6 Changes to VM Entries 4-6
4.7 Changes to VMX Capability Reporting 4-7
4.8 Instructions to Which SPP Does Not Apply 4-7

CHAPTER 5
INTEL® PROCESSOR TRACE: VMX IMPROVEMENTS

5.1 Introduction 5-1
5.2 Architecture Details 5-1
5.2.1 IA32_RTIT_CTL in VMCS Guest State 5-1
5.2.2 Supporting EPT for Trace Output 5-1
5.2.2.1 VM Exits Due to Intel PT Output 5-2
Exit Qualification 5-2
Preserving Pending Events 5-2
Additional VM Exits 5-3
5.2.2.2 Trace Data Management with Output Events 5-3
5.2.2.3 Intel PT Output Errors 5-3
5.2.3 New VM-Entry Consistency Checks 5-4
5.2.3.1 Special Treatment for SMM VM Exits 5-4
5.3 Enumeration 5-4

TABLES

PAGE

1-1 Recent Instruction Set Extensions Introduction in Intel 64 and IA-32 Processors 1-1
1-2 Information Returned by CPUID Instruction 1-5
1-3 Highest CPUID Source Operand for Intel 64 and IA-32 Processors 1-18
1-4 Processor Type Field 1-19
1-5 Feature Information Returned in the ECX Register 1-21
1-6 More on Feature Information Returned in the EDX Register 1-23
1-7 Encoding of Cache and TLB Descriptors 1-25
1-8 Structured Extended Feature Leaf, Function 0, EBX Register 1-28
1-9 Processor Brand String Returned with Pentium 4 Processor 1-31
1-10 Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings 1-33
1-11 Compressed Displacement (DISP8*N) Affected by Embedded Broadcast 1-38
1-12 EVEX DISP8*N for Instructions Not Affected by Embedded Broadcast 1-38
2-1 Inverse Byte Listings 2-3
2-2 PCLMULQDQ Quadword Selection of Immediate Byte 2-18
2-3 Pseudo-Op and PCLMULQDQ Implementation 2-19
4-1 Format of SPPTP 4-2
4-2 Format of the SPPT L4E 4-3
4-3 Exit Qualification for SPPT-Induced VM Exits 4-4
4-4 Fault Behavior Summary 4-6
5-1 VMCS Controls for IA32_RTIT_CTL MSR 5-1
5-2 VMCS Control for Intel PT Output to Guest Physical Addresses 5-2
5-3 New Asynchronous Exit Qualification Bit 5-2

FIGURES

PAGE

Figure 1-1. Version Information Returned by CPUID in EAX. 1-19
Figure 1-2. Feature Information Returned in the ECX Register 1-21
Figure 1-3. Feature Information Returned in the EDX Register 1-23
Figure 1-4. Determination of Support for the Processor Brand String 1-30
Figure 1-5. Algorithm for Extracting Maximum Processor Frequency 1-32
Figure 3-1. Register Source-Block Dot Product of Two Signed Word Operands with Doubleword Accumulation. 3-8

CHAPTER 1
FUTURE INTEL® ARCHITECTURE INSTRUCTION EXTENSIONS AND FEATURES

1.1 ABOUT THIS DOCUMENT

This document describes the software programming interfaces of Intel® architecture instruction extensions and features which may be included in future Intel processor generations. Intel does not guarantee the availability of these interfaces and features in any future product.

The instruction set extensions cover a diverse range of application domains and programming usages. The 512-bit SIMD vector SIMD extensions, referred to as Intel® Advanced Vector Extensions 512 (Intel® AVX-512) instructions, deliver comprehensive set of functionality and higher performance than Intel® AVX and Intel® AVX2 instructions. Intel AVX, Intel AVX2 and many Intel AVX-512 instructions are covered in *Intel® 64 and IA-32 Architectures Software Developer's Manual sets.* The reader can refer to them for basic and more background information related to various features referenced in this document.

The base of the 512-bit SIMD instruction extensions are referred to as Intel AVX-512 Foundation instructions. They include extensions of the AVX and AVX2 family of SIMD instructions but are encoded using a new encoding scheme with support for 512-bit vector registers, up to 32 vector registers in 64-bit mode, and conditional processing using opmask registers.

Chapter 2 is devoted to additional 512-bit instruction extensions in the Intel AVX-512 family targeting broad application domains, and instruction set extensions encoded using the EVEX prefix encoding scheme to operate at vector lengths smaller than 512-bits.

Chapter 3 describes instruction set extensions that are offered on processors based on the Knights Mill microarchitecture only.

Chapter 4 describes EPT-Based Sub-Page Permissions.

Chapter 5 describes Intel® Processor Trace: VMX Improvements.

1.2 INSTRUCTION SET EXTENSIONS INTRODUCTION IN INTEL 64 AND IA-32 PROCESSORS

Recent instruction set extensions are listed in Table 1-1. Within these groups, most instructions are collected into functional subgroups.

Table 1-1. Recent Instruction Set Extensions Introduction in Intel 64 and IA-32 Processors

Instruction Set Architecture	Processor Generation Introduction	Supported in Microarchitecture
SSE4.1 Extensions	Intel® Xeon® processor 3100, 3300, 5200, 5400, 7400, 7500 series, Intel® Core™ 2 Extreme processors QX9000 series, Intel® Core™ 2 Quad processor Q9000 series, Intel® Core™ 2 Duo processors 8000 series, T9000 series.	Legacy and later
	Intel® Atom™ processor.	Silvermont and later
SSE4.2 Extensions, CRC32, POPCNT	Intel® Core™ i7 965 processor, Intel® Xeon® processors X3400, X3500, X5500, X6500, X7500 series.	Legacy and later
	Intel® Atom™ processor.	Silvermont and later

Table 1-1. Recent Instruction Set Extensions Introduction in Intel 64 and IA-32 Processors(Continued)

Instruction Set Architecture	Processor Generation Introduction	Supported in Microarchitecture
AESNI, PCLMULQDQ	Intel® Xeon® processor E7 series, Intel® Xeon® processors X3600, X5600, Intel® Core™ i7 980X processor. Use CPUID to verify presence of AESNI and PCLMULQDQ across Intel® Core™ processor families.	Westmere and later
	Intel® Atom™ processor.	Silvermont and later
Intel AVX	Intel® Xeon® processor E3 and E5 families. 2nd Generation Intel® Core™ i7, i5, i3 processor 2xxx families.	Sandy Bridge and later
F16C	3rd Generation Intel® Core™ processors, Intel® Xeon® processor E3-1200 v2 product family, Next Generation Intel® Xeon® processors, Intel® Xeon® processor E5 v2 and E7 v2 families.	Ivy Bridge and later
RDRAND	3rd Generation Intel® Core™ processors, Intel® Xeon® processor E3-1200 v2 product family, Next Generation Intel® Xeon® processors, Intel® Xeon® processor E5 v2 and E7 v2 families.	Ivy Bridge and later
	Intel® Atom™ processor.	Silvermont and later
FS/GS base access	3rd Generation Intel® Core™ processors, Intel® Xeon® processor E3-1200 v2 product family, Next Generation Intel® Xeon® processors, Intel® Xeon® processor E5 v2 and E7 v2 families.	Ivy Bridge and later
	Intel® Atom™ processor.	Goldmont and later
FMA, AVX2, BMI1, BMI2, INVPCID, LZCNT, TSX	Intel® Xeon® processor E3/E5/E7 v3 product families. 4th Generation Intel® Core™ processor family.	Haswell and later
MOVBE	Intel® Xeon® processor E3/E5/E7 v3 product families. 4th Generation Intel® Core™ processor family.	Haswell and later
	Intel® Atom™ processor.	Silvermont and later
PREFETCHW	Intel® Core™ M processor family; 5th Generation Intel® Core™ processor family.	Broadwell and later
	Intel® Atom™ processor.	Silvermont and later
ADX	Intel® Core™ M processor family; 5th Generation Intel® Core™ processor family.	Broadwell and later
CLAC, STAC	Intel® Core™ M processor family; 5th Generation Intel® Core™ processor family.	Broadwell and later
	Intel® Atom™ processor.	Goldmont and later
RDSEED	Intel® Core™ M processor family; 5th Generation Intel® Core™ processor family.	Broadwell and later
	Intel® Atom™ processor.	Goldmont and later
AVX512ER, AVX512PF, PREFETCHWT1	Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series.	Knights Landing
AVX512F, AVX512CD	Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series.	Knights Landing
	Intel® Xeon® Processor Scalable Family.	Skylake Server and later
	TBD	Cannon Lake and later

Table 1-1. Recent Instruction Set Extensions Introduction in Intel 64 and IA-32 Processors(Continued)

Instruction Set Architecture	Processor Generation Introduction	Supported in Microarchitecture
CLFLUSHOPT, XSAVEC, XSAVES, MPX	Intel® Xeon® Processor Scalable Family.	Skylake Server and later
	6th Generation Intel® Core™ processor family.	Skylake and later
	Intel® Atom™ processor.	Goldmont and later
SGX1	6th Generation Intel® Core™ processor family.	Skylake and later
	Intel® Atom™ processor.	Goldmont Plus and later
AVX512DQ, AVX512BW, AVX512VL	Intel® Xeon® Processor Scalable Family.	Skylake Server and later
	TBD	Cannon Lake and later
CLWB	Intel® Xeon® Processor Scalable Family.	Skylake Server and later
	TBD	Ice Lake and later
PKU	Intel® Xeon® Processor Scalable Family.	Skylake Server and later
AVX512_IFMA, AVX512_VBMI	TBD	Cannon Lake and later
SHA-NI	TBD	Cannon Lake and later
	Intel® Atom™ processor.	Goldmont and later
UMIP	TBD	Cannon Lake and later
	Intel® Atom™ processor.	Goldmont Plus and later
PTWRITE	Intel® Atom™ processor.	Goldmont Plus and later
RDPID	TBD	Ice Lake and later
	Intel® Atom™ processor.	Goldmont Plus and later
AVX512_4FMAPS, AVX512_4VNNI	Future Intel Xeon Phi processor.	Knights Mill
AVX512_VPOPCNTDQ	Future Intel Xeon Phi processor.	Knights Mill
	TBD	Ice Lake and later
Fast Short REP MOV	TBD	Ice Lake and later
AVX512_VNNI, GFNI, VAES, AVX512_VBMI2, VPCLMULQDQ, AVX512_BITALG	TBD	Ice Lake and later

1.3 CPUID INSTRUCTION

CPUID—CPU Identification

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F A2	CPUID	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).

Description

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction. This instruction operates the same in non-64-bit modes and 64-bit mode.

CPUID returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers.[1] The instruction's output is dependent on the contents of the EAX register upon execution (in some cases, ECX as well). For example, the following pseudocode loads EAX with 00H and causes CPUID to return a Maximum Return Value and the Vendor Identification String in the appropriate registers:

```
MOV EAX, 00H
CPUID
```

Table 1-2 shows information returned, depending on the initial value loaded into the EAX register. Table 1-3 shows the maximum CPUID input value recognized for each family of IA-32 processors on which CPUID is implemented.

Two types of information are returned: basic and extended function information. If a value is entered for CPUID.EAX is invalid for a particular processor, the data for the highest basic information leaf is returned. For example, using the Intel Core 2 Duo E6850 processor, the following is true:

```
CPUID.EAX = 05H (* Returns MONITOR/MWAIT leaf. *)
CPUID.EAX = 0AH (* Returns Architectural Performance Monitoring leaf. *)
CPUID.EAX = 0BH (* INVALID: Returns the same information as CPUID.EAX = 0AH. *)
CPUID.EAX = 80000008H (* Returns virtual/physical address size data. *)
CPUID.EAX = 8000000AH (* INVALID: Returns same information as CPUID.EAX = 0AH. *)
```

When CPUID returns the highest basic leaf information as a result of an invalid input EAX value, any dependence on input ECX value in the basic leaf is honored.

CPUID can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.

See also:

"Serializing Instructions" in Chapter 8, "Multiple-Processor Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*

"Caching Translation Information" in Chapter 4, "Paging," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

1. On Intel 64 processors, CPUID clears the high 32 bits of the RAX/RBX/RCX/RDX registers in all modes.

Table 1-2. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
	Basic CPUID Information	
0H	EAX EBX ECX EDX	Maximum Input Value for Basic CPUID Information (see Table 1-3) "Genu" "ntel" "inel"
01H	EAX	Version Information: Type, Family, Model, and Stepping ID (see Figure 1-1)
	EBX	Bits 7-0: Brand Index Bits 15-8: CLFLUSH line size (Value ∗ 8 = cache line size in bytes) Bits 23-16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31-24: Initial APIC ID
	ECX	Feature Information (see Figure 1-2 and Table 1-5)
	EDX	Feature Information (see Figure 1-3 and Table 1-6)
		NOTES: * The nearest power-of-2 integer that is not smaller than EBX[23:16] is the maximum number of unique initial APIC IDs reserved for addressing different logical processors in a physical package.
02H	EAX EBX ECX EDX	Cache and TLB Information (see Table 1-7) Cache and TLB Information Cache and TLB Information Cache and TLB Information
03H	EAX	Reserved.
	EBX	Reserved.
	ECX	Bits 00-31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
	EDX	Bits 32-63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
		NOTES: Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature.
	CPUID leaves > 3 < 80000000 are visible only when IA32_MISC_ENABLES.BOOT_NT4[bit 22] = 0 (default).	
	Deterministic Cache Parameters Leaf	
04H		**NOTES:** Leaf 04H output depends on the initial value in ECX. See also: "INPUT EAX = 4: Returns Deterministic Cache Parameters for each level on page 1-27.
	EAX	Bits 4-0: Cache Type Field 0 = Null - No more caches 1 = Data Cache 2 = Instruction Cache 3 = Unified Cache 4-31 = Reserved

Table 1-2. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
		Bits 7-5: Cache Level (starts at 1) Bits 8: Self Initializing cache level (does not need SW initialization) Bits 9: Fully Associative cache Bits 13-10: Reserved Bits 25-14: Maximum number of addressable IDs for logical processors sharing this cache*, ** Bits 31-26: Maximum number of addressable IDs for processor cores in the physical package*, ***, ****
	EBX	Bits 11-00: L = System Coherency Line Size* Bits 21-12: P = Physical Line partitions* Bits 31-22: W = Ways of associativity*
	ECX	Bits 31-00: S = Number of Sets*
	EDX	Bit 0: WBINVD/INVD behavior on lower level caches Bit 10: Write-Back Invalidate/Invalidate 0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache 1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache. Bit 1: Cache Inclusiveness 0 = Cache is not inclusive of lower cache levels. 1 = Cache is inclusive of lower cache levels. Bit 2: Complex cache indexing 0 = Direct mapped cache 1 = A complex function is used to index the cache, potentially using all address bits. Bits 31-03: Reserved = 0 **NOTES:** * Add one to the return value to get the result. ** The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache *** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core_IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID. ****The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.
	MONITOR/MWAIT Leaf	
05H	EAX	Bits 15-00: Smallest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0
	EBX	Bits 15-00: Largest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0
	ECX	Bits 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported Bits 01: Supports treating interrupts as break-event for MWAIT, even when interrupts disabled Bits 31 - 02: Reserved

Table 1-2. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 03 - 00: Number of C0* sub C-states supported using MWait Bits 07 - 04: Number of C1* sub C-states supported using MWAIT Bits 11 - 08: Number of C2* sub C-states supported using MWAIT Bits 15 - 12: Number of C3* sub C-states supported using MWAIT Bits 19 - 16: Number of C4* sub C-states supported using MWAIT Bits 23 - 20: Number of C5* sub C-states supported using MWAIT Bits 27 - 24: Number of C6* sub C-states supported using MWAIT Bits 31 - 28: Number of C7* sub C-states supported using MWAIT **NOTE:** * The definition of C0 through C7 states for MWAIT extension are processor-specific C-states, not ACPI C-states.
	Thermal and Power Management Leaf	
06H	EAX	Bit 00: Digital temperature sensor is supported if set Bit 01: Intel Turbo Boost Technology Available (see description of IA32_MISC_ENABLE[38]). Bit 02: ARAT. APIC-Timer-always-running feature is supported if set. Bit 03: Reserved Bit 04: PLN. Power limit notification controls are supported if set. Bit 05: ECMD. Clock modulation duty cycle extension is supported if set. Bit 06: PTM. Package thermal management is supported if set. Bit 07: HWP. HWP base registers (IA32_PM_ENABLE[bit 0], IA32_HWP_CAPABILITIES, IA32_HWP_REQUEST, IA32_HWP_STATUS) are supported if set. Bit 08: HWP_Notification. IA32_HWP_INTERRUPT MSR is supported if set. Bit 09: HWP_Activity_Window. IA32_HWP_REQUEST[bits 41:32] is supported if set. Bit 10: HWP_Energy_Performance_Preference. IA32_HWP_REQUEST[bits 31:24] is supported if set. Bit 11: HWP_Package_Level_Request. IA32_HWP_REQUEST_PKG MSR is supported if set. Bit 12: Reserved. Bit 13: HDC. HDC base registers IA32_PKG_HDC_CTL, IA32_PM_CTL1, IA32_THREAD_STALL MSRs are supported if set. Bit 14: Intel® Turbo Boost Max Technology 3.0 available. Bits 31 - 15: Reserved
	EBX	Bits 03 - 00: Number of Interrupt Thresholds in Digital Thermal Sensor Bits 31 - 04: Reserved
	ECX	Bit 00: Hardware Coordination Feedback Capability (Presence of IA32_MPERF and IA32_APERF). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of the expected processor performance when running at the TSC frequency. Bits 02 - 01: Reserved = 0 Bit 03: The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H) Bits 31 - 04: Reserved = 0
	EDX	Reserved = 0
	Structured Extended Feature Flags Enumeration Leaf (Output depends on ECX input value)	
07H		**NOTES:** Leaf 07H main leaf (ECX = 0). If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0.
	EAX	Bits 31-00: Reports the maximum number sub-leaves that are supported in leaf 07H.

Table 1-2. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	EBX	Bit 00: FSGSBASE. Supports RDFSBASE/RDGSBASE/WRFSBASE/WRGSBASE if 1. Bit 01: IA32_TSC_ADJUST MSR is supported if 1. Bit 02: SGX Bit 03: BMI1 Bit 04: HLE Bit 05: AVX2 Bit 07: SMEP. Supports Supervisor Mode Execution Protection if 1. Bit 06: Reserved Bit 08: BMI2 Bit 09: Supports Enhanced REP MOVSB/STOSB if 1. Bit 10: INVPCID Bit 11: RTM Bit 12: Supports Platform Quality of Service Monitoring (PQM) capability if 1. Bit 13: Deprecates FPU CS and FPU DS values if 1. Bit 14: Intel Memory Protection Extensions Bit 15: Supports Platform Quality of Service Enforcement (PQE) capability if 1. Bit 16: AVX512F Bit 17: AVX512DQ Bit 18: RDSEED Bit 19: ADX Bit 20: SMAP Bit 21: AVX512_IFMA Bit 22: Reserved Bit 23: CLFLUSHOPT Bit 24: CLWB Bit 25: Intel Processor Trace Bit 26: AVX512PF Bit 27: AVX512ER Bit 28: AVX512CD Bit 29: SHA Bit 30: AVX512BW Bit 31: AVX512VL
	ECX	Bit 00: PREFETCHWT1 Bit 01: AVX512_VBMI Bit 02: UMIP. Supports user-mode instruction prevention if 1. Bit 03: PKU. Supports protection keys for user-mode pages if 1. Bit 04: OSPKE. If 1, OS has set CR4.PKE to enable protection keys (and the RDPKRU/WRPKRU instructions). Bit 05: Reserved Bit 06: AVX512_VBMI2 Bit 07: Reserved Bit 08: GFNI Bit 09: VAES Bit 10: VPCLMULQDQ Bit 11: AVX512_VNNI Bit 12: AVX512_BITALG Bit 13: Reserved Bit 14: AVX512_VPOPCNTDQ Bits 16 - 15: Reserved Bits 21 - 17: The value of MAWAU used by the BNDLDX and BNDSTX instructions in 64-bit mode. Bit 22: RDPID. Supports Read Processor ID if 1. Bits 29 - 23: Reserved. Bit 30: SGX_LC. Supports SGX Launch Configuration if 1. Bit 31: Reserved.

Table 1-2. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 01 - 00: Reserved Bit 02: AVX512_4VNNIW Bit 03: AVX512_4FMAPS Bits 31-04: Reserved
	Structured Extended Feature Enumeration Sub-leaves (EAX = 07H, ECX = n, n ≥ 1)	
07H		**NOTES:** Leaf 07H output depends on the initial value in ECX. If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0.
	EAX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	EBX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	ECX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	EDX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	Direct Cache Access Information Leaf	
09H	EAX EBX ECX EDX	Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H) Reserved Reserved Reserved
	Architectural Performance Monitoring Leaf	
0AH	EAX	Bits 07 - 00: Version ID of architectural performance monitoring Bits 15- 08: Number of general-purpose performance monitoring counter per logical processor Bits 23 - 16: Bit width of general-purpose, performance monitoring counter Bits 31 - 24: Length of EBX bit vector to enumerate architectural performance monitoring events
	EBX	Bit 00: Core cycle event not available if 1 Bit 01: Instruction retired event not available if 1 Bit 02: Reference cycles event not available if 1 Bit 03: Last-level cache reference event not available if 1 Bit 04: Last-level cache misses event not available if 1 Bit 05: Branch instruction retired event not available if 1 Bit 06: Branch mispredict retired event not available if 1 Bits 31- 07: Reserved = 0
	ECX EDX	Reserved = 0 Bits 04 - 00: Number of fixed-function performance counters (if Version ID > 1) Bits 12- 05: Bit width of fixed-function performance counters (if Version ID > 1) Reserved = 0
	Extended Topology Enumeration Leaf	
0BH		**NOTES:** Most of Leaf 0BH output depends on the initial value in ECX. The EDX output of leaf 0BH is always valid and does not vary with input value in ECX. Output value in ECX[7:0] always equals input value in ECX[7:0]. For sub-leaves that returns an invalid level-type of 0 in ECX[15:8]; EAX and EBX will return 0. If an input value N in ECX returns the invalid level-type of 0 in ECX[15:8], other input values with ECX > N also return 0 in ECX[15:8]
	EAX	Bits 04-00: Number of bits to shift right on x2APIC ID to get a unique topology ID of the next level type*. All logical processors with the same next level ID share current level. Bits 31-5: Reserved.

Table 1-2. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	EBX	Bits 15 - 00: Number of logical processors at this level type. The number reflects configuration as shipped by Intel**. Bits 31- 16: Reserved.
	ECX	Bits 07 - 00: Level number. Same value in ECX input. Bits 15 - 08: Level type***. Bits 31 - 16: Reserved.
	EDX	Bits 31- 00: x2APIC ID the current logical processor.
		NOTES: * Software should use this field (EAX[4:0]) to enumerate processor topology of the system. ** Software must not use EBX[15:0] to enumerate processor topology of the system. This value in this field (EBX[15:0]) is only intended for display/diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations. *** The value of the "level type" field is not related to level numbers in any way, higher "level type" values do not mean higher levels. Level type field has the following encoding: 0: invalid 1: SMT 2: Core 3-255: Reserved
Processor Extended State Enumeration Main Leaf (EAX = 0DH, ECX = 0)		
0DH		**NOTES:** Leaf 0DH main leaf (ECX = 0).
	EAX	Bits 31-00: Reports the valid bit fields of the lower 32 bits of the XFEATURE_ENABLED_MASK register. If a bit is 0, the corresponding bit field in XCR0 is reserved. Bit 00: legacy x87 Bit 01: 128-bit SSE Bit 02: 256-bit AVX Bits 04 - 03: MPX state Bit 07 - 05: AVX-512 state Bit 08: Used for IA32_XSS Bit 09: PKRU state Bits 31-10: Reserved
	EBX	Bits 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCR0. May be different than ECX if some features at the end of the XSAVE save area are not enabled.
	ECX	Bit 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e all the valid bit fields in XCR0.
	EDX	Bit 31-0: Reports the valid bit fields of the upper 32 bits of the XCR0 register. If a bit is 0, the corresponding bit field in XCR0 is reserved

Table 1-2. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1)	
0DH	EAX	Bit 00: XSAVEOPT is available Bit 01: Supports XSAVEC and the compacted form of XRSTOR if set Bit 02: Supports XGETBV with ECX = 1 if set Bit 03: Supports XSAVES/XRSTORS and IA32_XSS if set Bits 31-04: Reserved
	EBX	Bits 31-00: The size in bytes of the XSAVE area containing all states enabled by XCR0 | IA32_XSS.
	ECX	Bits 31-00: Reports the supported bits of the lower 32 bits of the IA32_XSS MSR. IA32_XSS[n] can be set to 1 only if ECX[n] is 1. Bits 07-00: Used for XCR0 Bit 08: PT state Bit 09: Used for XCR0 Bits 31-10: Reserved
	EDX	Bits 31-00: Reports the supported bits of the upper 32 bits of the IA32_XSS MSR. IA32_XSS[n+32] can be set to 1 only if EDX[n] is 1. Bits 31-00: Reserved
	Processor Extended State Enumeration Sub-leaves (EAX = 0DH, ECX = n, n > 1)	
0DH		**NOTES:** Leaf 0DH output depends on the initial value in ECX. Each sub-leaf index (starting at position 2) is supported if it corresponds to a supported bit in either the XCR0 register or the IA32_XSS MSR. * If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf n (0 ≤ n ≤ 31) is invalid if sub-leaf 0 returns 0 in EAX[n] and sub-leaf 1 returns 0 in ECX[n]. Sub-leaf n (32 ≤ n ≤ 63) is invalid if sub-leaf 0 returns 0 in EDX[n-32] and sub-leaf 1 returns 0 in EDX[n-32].
	EAX	Bits 31-0: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, *n*. This field reports 0 if the sub-leaf index, *n*, is invalid*.
	EBX	Bits 31-0: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area. This field reports 0 if the sub-leaf index, n, does not map to a valid bit in the XCR0 register*.
	ECX	Bit 0 is set if the bit n (corresponding to the sub-leaf index) is supported in the IA32_XSS MSR; it is clear if bit n is instead supported in XCR0. Bit 1 is set if, when the compacted format of an XSAVE area is used, this extended state component located on the next 64-byte boundary following the preceding state component (otherwise, it is located immediately following the preceding state component). Bits 31:02 are reserved. This field reports 0 if the sub-leaf index, n, is invalid*.
	EDX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	Platform QoS Monitoring Enumeration Sub-leaf (EAX = 0FH, ECX = 0)	
0FH		**NOTES:** Leaf 0FH output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource type starting at bit position 1 of EDX.
	EAX	Reserved.
	EBX	Bits 31-0: Maximum range (zero-based) of RMID within this physical processor of all types.
	ECX	Reserved.
	EDX	Bit 00: Reserved. Bit 01: Supports L3 Cache QoS Monitoring if 1. Bits 31 - 02: Reserved

Table 1-2. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	L3 Cache QoS Monitoring Capability Enumeration Sub-leaf (EAX = 0FH, ECX = 1)	
0FH	**NOTES:** Leaf 0FH output depends on the initial value in ECX.	
	EAX	Reserved.
	EBX	Bits 31-0: Conversion factor from reported IA32_QM_CTR value to occupancy metric (bytes).
	ECX	Maximum range (zero-based) of RMID of this resource type.
	EDX	Bit 00: Supports L3 occupancy monitoring if 1. Bits 31:01: Reserved
	Platform QoS Enforcement Enumeration Sub-leaf (EAX = 10H, ECX = 0)	
10H	**NOTES:** Leaf 10H output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource identification (ResID) starting at bit position 1 of EBX.	
	EAX	Reserved.
	EBX	Bit 00: Reserved. Bit 01: Supports L3 Cache QoS Enforcement if 1. Bits 31 - 02: Reserved.
	ECX	Reserved.
	EDX	Reserved.
	L3 Cache QoS Enforcement Enumeration Sub-leaf (EAX = 10H, ECX = ResID =1)	
10H	**NOTES:** Leaf 10H output depends on the initial value in ECX.	
	EAX	Bits 4:0: Length of the capacity bit mask for the corresponding ResID. Bits 31:05: Reserved
	EBX	Bits 31-0: Bit-granular map of isolation/contention of allocation units.
	ECX	Bit 00: Reserved. Bit 01: Updates of COS should be infrequent if 1. Bit 02: Code and Data Prioritization Technology supported if 1. Bits 31:03: Reserved
	EDX	Bits 15:0: Highest COS number supported for this ResID. Bits 31:16: Reserved
	Intel Processor Trace Enumeration Main Leaf (EAX = 14H, ECX = 0)	
14H	**NOTES:** Leaf 14H main leaf (ECX = 0).	
	EAX	Bits 31-0: Reports the maximum sub-leaf supported in leaf 14H.
	EBX	Bit 00: If 1, Indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed. Bits 01: If 1, Indicates support of Configurable PSB and Cycle-Accurate Mode. Bits 02: If 1, Indicates support of IP Filtering, TraceStop filtering, and preservation of Intel PT MSRs across warm reset. Bits 03: If 1, Indicates support of MTC timing packet and suppression of COFI-based packets. Bits 31: 04: Reserved

Table 1-2. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	ECX	Bit 00: If 1, Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme; IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed. Bit 01: If 1, ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS. Bits 02: If 1, Indicates support of Single-Range Output scheme. Bits 03: If 1, Indicates support of output to Trace Transport subsystem. Bit 30:04: Reserved Bit 31: If 1, Generated packets which contain IP payloads have LIP values, which include the CS base component.
	EDX	Bits 31- 00: Reserved
	Intel Processor Trace Enumeration Sub-leaf (EAX = 14H, ECX = 1)	
14H	EAX	Bits 2:0: Number of configurable Address Ranges for filtering. Bits 15-03: Reserved Bit 31:16: Bitmap of supported MTC period encodings
	EBX	Bits 15-0: Bitmap of supported Cycle Threshold value encodings Bit 31:16: Bitmap of supported Configurable PSB frequency encodings
	ECX	Bits 31-00: Reserved
	EDX	Bits 31- 00: Reserved
	Time Stamp Counter and Core Crystal Clock Information Leaf	
15H		**NOTES:** If EBX[31:0] is 0, the TSC and "core crystal clock" ratio is not enumerated. EBX[31:0]/EAX[31:0] indicates the ratio of the TSC frequency and the core crystal clock frequency. If ECX is 0, the core crystal clock frequency is not enumerated. "TSC frequency" = "core crystal clock frequency" * EBX/EAX. The core crystal clock may differ from the reference clock, bus clock, or core clock frequencies.
	EAX	Bits 31:0: An unsigned integer which is the denominator of the TSC/"core crystal clock" ratio.
	EBX	Bits 31-0: An unsigned integer which is the numerator of the TSC/"core crystal clock" ratio.
	ECX	Bits 31:0: An unsigned integer which is the nominal frequency of the core crystal clock in Hz.
	EDX	Bits 31:0: Reserved = 0.

Table 1-2. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	Processor Frequency Information Leaf	
16H	EAX	Bits 15:0: Processor Base Frequency (in MHz). Bits 31:16: Reserved =0
	EBX	Bits 15:0: Maximum Frequency (in MHz). Bits 31:16: Reserved = 0
	ECX	Bits 15:0: Bus (Reference) Frequency (in MHz). Bits 31:16: Reserved = 0
	EDX	Reserved
		NOTES: * Data is returned from this interface in accordance with the processor's specification and does not reflect actual values. Suitable use of this data includes the display of processor information in like manner to the processor brand string and for determining the appropriate range to use when displaying processor information e.g. frequency history graphs. The returned information should not be used for any other purpose as the returned information does not accurately correlate to information / counters returned by other processor interfaces. While a processor may support the Processor Frequency Information leaf, fields that return a value of zero are not supported.
	System-On-Chip Vendor Attribute Enumeration Main Leaf (EAX = 17H, ECX = 0)	
17H		**NOTES:** Leaf 17H main leaf (ECX = 0). Leaf 17H output depends on the initial value in ECX. Leaf 17H sub-leaves 1 through 3 reports SOC Vendor Brand String. Leaf 17H is valid if MaxSOCID_Index >= 3. Leaf 17H sub-leaves 4 and above are reserved.
	EAX	Bits 31 - 00: MaxSOCID_Index. Reports the maximum input value of supported sub-leaf in leaf 17H.
	EBX	Bits 15 - 00: SOC Vendor ID. Bit 16: IsVendorScheme. If 1, the SOC Vendor ID field is assigned via an industry standard enumeration scheme. Otherwise, the SOC Vendor ID field is assigned by Intel. Bits 31 - 17: Reserved = 0.
	ECX	Bits 31 - 00: Project ID. A unique number an SOC vendor assigns to its SOC projects.
	EDX	Bits 31 - 00: Stepping ID. A unique number within an SOC project that an SOC vendor assigns.
	System-On-Chip Vendor Attribute Enumeration Sub-leaf (EAX = 17H, ECX = 1..3)	
17H	EAX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
	EBX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
	ECX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
	EDX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
		NOTES: Leaf 17H output depends on the initial value in ECX. SOC Vendor Brand String is a UTF-8 encoded string padded with trailing bytes of 00H. The complete SOC Vendor Brand String is constructed by concatenating in ascending order of EAX:EBX:ECX:EDX and from the sub-leaf 1 fragment towards sub-leaf 3.

Table 1-2. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	System-On-Chip Vendor Attribute Enumeration Sub-leaves (EAX = 17H, ECX > MaxSOCID_Index)	
17H		**NOTES:** Leaf 17H output depends on the initial value in ECX.
	EAX	Bits 31 - 00: Reserved = 0.
	EBX	Bits 31 - 00: Reserved = 0.
	ECX	Bits 31 - 00: Reserved = 0.
	EDX	Bits 31 - 00: Reserved = 0.
	Deterministic Address Translation Parameters Main Leaf (EAX = 18H, ECX = 0)	
18H		**NOTES:** Each sub-leaf enumerates a different address translations structure. Valid sub-leaves do not need to be contiguous or in any particular order. A valid sub-leaf may be in a higher input ECX value than an invalid sub-leaf or than a valid sub-leaf of a higher or lower-level structure. If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX. * Add one to the return value to get the result.
	EAX	Bits 31 - 00: Reports the maximum input value of supported sub-leaf in leaf 18H.
	EBX	Bit 00: 4K page size entries supported by this structure. Bit 01: 2MB page size entries supported by this structure. Bit 02: 4MB page size entries supported by this structure. Bit 03: 1 GB page size entries supported by this structure. Bits 07 - 04: Reserved. Bits 10 - 08: Partitioning (0: Soft partitioning between the logical processors sharing this structure). Bits 15 - 11: Reserved. Bits 31 - 16: W = Ways of associativity.
	ECX	Bits 31 - 00: S = Number of Sets.
	EDX	Bits 04 - 00: Translation cache type field. 00000b: Null (indicates this sub-leaf is not valid). 00001b: Data TLB. 00010b: Instruction TLB. 00011b: Unified TLB. All other encodings are reserved. Bits 07 - 05: Translation cache level (starts at 1). Bit 08: Fully associative structure. Bits 13 - 09: Reserved. Bits 25- 14: Maximum number of addressable IDs for logical processors sharing this translation cache* Bits 31 - 26: Reserved.
	Deterministic Address Translation Parameters Sub-leaf (EAX = 18H, ECX ≥ 1)	
18H		**NOTES:** If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX. * Add one to the return value to get the result.
	EAX	Bits 31 - 00: Reserved.

Table 1-2. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	EBX	Bit 00: 4K page size entries supported by this structure. Bit 01: 2MB page size entries supported by this structure. Bit 02: 4MB page size entries supported by this structure. Bit 03: 1 GB page size entries supported by this structure. Bits 07 - 04: Reserved. Bits 10 - 08: Partitioning (0: Soft partitioning between the logical processors sharing this structure). Bits 15 - 11: Reserved. Bits 31 - 16: W = Ways of associativity.
	ECX	Bits 31 - 00: S = Number of Sets.
	EDX	Bits 04 - 00: Translation cache type field. 0000b: Null (indicates this sub-leaf is not valid). 0001b: Data TLB. 0010b: Instruction TLB. 0011b: Unified TLB. All other encodings are reserved. Bits 07 - 05: Translation cache level (starts at 1). Bit 08: Fully associative structure. Bits 13 - 09: Reserved. Bits 25- 14: Maximum number of addressable IDs for logical processors sharing this translation cache* Bits 31 - 26: Reserved.
	Unimplemented CPUID Leaf Functions	
40000000H - 4FFFFFFFH		Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is in the range 40000000H to 4FFFFFFFH.
	Extended Function CPUID Information	
80000000H	EAX	Maximum Input Value for Extended Function CPUID Information (see Table 1-3).
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
80000001H	EAX	Extended Processor Signature and Feature Bits.
	EBX	Reserved
	ECX	Bit 0: LAHF/SAHF available in 64-bit mode Bits 4-1: Reserved Bit 5: LZCNT available Bits 7-6 Reserved Bit 8: PREFETCHW Bits 31-9: Reserved
	EDX	Bits 10-0: Reserved Bit 11: SYSCALL/SYSRET available (when in 64-bit mode) Bits 19-12: Reserved = 0 Bit 20: Execute Disable Bit available Bits 25-21: Reserved = 0 Bit 26: 1-GByte pages are available if 1 Bit 27: RDTSCP and IA32_TSC_AUX are available if 1 Bits 28: Reserved = 0 Bit 29: Intel® 64 Architecture available if 1 Bits 31-30: Reserved = 0

Table 1-2. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
80000002H	EAX EBX ECX EDX	Processor Brand String Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued
80000003H	EAX EBX ECX EDX	Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued
80000004H	EAX EBX ECX EDX	Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued
80000005H	EAX EBX ECX EDX	Reserved = 0 Reserved = 0 Reserved = 0 Reserved = 0
80000006H	EAX EBX ECX EDX	Reserved = 0 Reserved = 0 Bits 7-0: Cache Line size in bytes Bits 11-08: Reserved Bits 15-12: L2 Associativity field * Bits 31-16: Cache size in 1K units Reserved = 0 **NOTES:** * L2 associativity field encodings: 00H - Disabled 01H - Direct mapped 02H - 2-way 04H - 4-way 06H - 8-way 08H - 16-way 0FH - Fully associative
80000007H	EAX EBX ECX EDX	Reserved = 0 Reserved = 0 Reserved = 0 Bits 07-00: Reserved = 0 Bit 08: Invariant TSC available if 1 Bits 31-09: Reserved = 0
80000008H	EAX EBX ECX EDX	Virtual/Physical Address size Bits 7-0: #Physical Address Bits* Bits 15-8: #Virtual Address Bits Bits 31-16: Reserved = 0 Reserved = 0 Reserved = 0 Reserved = 0 **NOTES:** * If CPUID.80000008H:EAX[7:0] is supported, the maximum physical address number supported should come from this field.

INPUT EAX = 0H: Returns CPUID's Highest Value for Basic Processor Information and the Vendor Identification String

When CPUID executes with EAX set to 0H, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register (see Table 1-3) and is processor specific.

A vendor identification string is also returned in EBX, EDX, and ECX. For Intel processors, the string is "GenuineIntel" and is expressed:

EBX ← 756e6547h (* "Genu", with G in the low 4 bits of BL *)
EDX ← 49656e69h (* "inel", with i in the low 4 bits of DL *)
ECX ← 6c65746eh (* "ntel", with n in the low 4 bits of CL *)

INPUT EAX = 80000000H: Returns CPUID's Highest Value for Extended Processor Information

When CPUID executes with EAX set to 0H, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register (see Table 1-3) and is processor specific.

Table 1-3. Highest CPUID Source Operand for Intel 64 and IA-32 Processors

Intel 64 or IA-32 Processors	Highest Value in EAX	
	Basic Information	Extended Function Information
Earlier Intel486 Processors	CPUID Not Implemented	CPUID Not Implemented
Later Intel486 Processors and Pentium Processors	01H	Not Implemented
Pentium Pro and Pentium II Processors, Intel® Celeron® Processors	02H	Not Implemented
Pentium III Processors	03H	Not Implemented
Pentium 4 Processors	02H	80000004H
Intel Xeon Processors	02H	80000004H
Pentium M Processor	02H	80000004H
Pentium 4 Processor supporting Hyper-Threading Technology	05H	80000008H
Pentium D Processor (8xx)	05H	80000008H
Pentium D Processor (9xx)	06H	80000008H
Intel Core Duo Processor	0AH	80000008H
Intel Core 2 Duo Processor	0AH	80000008H
Intel Xeon Processor 3000, 5100, 5300 Series	0AH	80000008H
Intel Xeon Processor 3000, 5100, 5200, 5300, 5400 Series	0AH	80000008H
Intel Core 2 Duo Processor 8000 Series	0DH	80000008H
Intel Xeon Processor 5200, 5400 Series	0AH	80000008H

IA32_BIOS_SIGN_ID Returns Microcode Update Signature

For processors that support the microcode update facility, the IA32_BIOS_SIGN_ID MSR is loaded with the update signature whenever CPUID executes. The signature is returned in the upper DWORD. For details, see Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

INPUT EAX = 01H: Returns Model, Family, Stepping Information

When CPUID executes with EAX set to 01H, version information is returned in EAX (see Figure 1-1). For example: model, family, and processor type for the Intel Xeon processor 5100 series is as follows:

- Model — 1111B
- Family — 0101B
- Processor Type — 00B

See Table 1-4 for available processor type values. Stepping IDs are provided as needed.

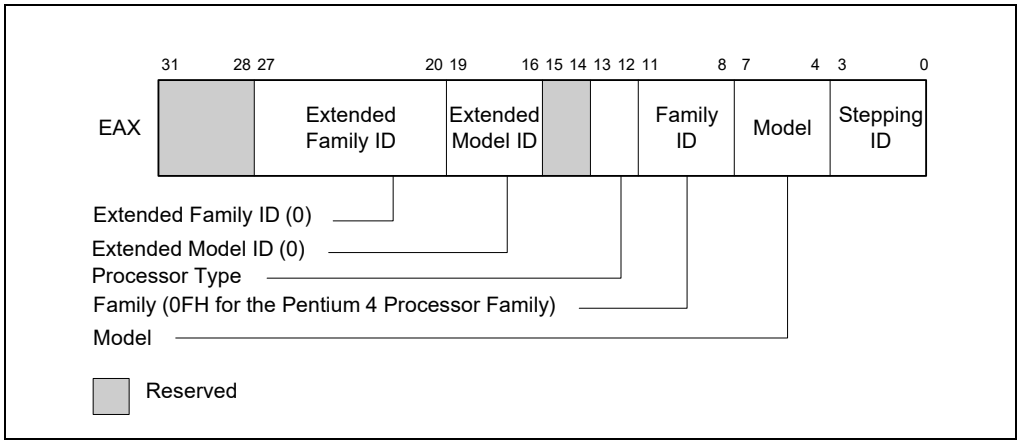

Figure 1-1. Version Information Returned by CPUID in EAX

Table 1-4. Processor Type Field

Type	Encoding
Original OEM Processor	00B
Intel OverDrive® Processor	01B
Dual processor (not applicable to Intel486 processors)	10B
Intel reserved	11B

NOTE

See "Caching Translation Information" in Chapter 4, “Paging,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, and Chapter 16 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for information on identifying earlier IA-32 processors.

The Extended Family ID needs to be examined only when the Family ID is 0FH. Integrate the fields into a display using the following rule:

```
IF Family_ID ≠ 0FH
    THEN Displayed_Family = Family_ID;
    ELSE Displayed_Family = Extended_Family_ID + Family_ID;
    (* Right justify and zero-extend 4-bit field. *)
FI;
(* Show Display_Family as HEX field. *)
```

The Extended Model ID needs to be examined only when the Family ID is 06H or 0FH. Integrate the field into a display using the following rule:

```
IF (Family_ID = 06H or Family_ID = 0FH)
    THEN Displayed_Model = (Extended_Model_ID << 4) + Model_ID;
    (* Right justify and zero-extend 4-bit field; display Model_ID as HEX field.*)
    ELSE Displayed_Model = Model_ID;
FI;
(* Show Display_Model as HEX field. *)
```

INPUT EAX = 01H: Returns Additional Information in EBX

When CPUID executes with EAX set to 01H, additional information is returned to the EBX register:

- Brand index (low byte of EBX) — this number provides an entry into a brand string table that contains brand strings for IA-32 processors. More information about this field is provided later in this section.
- CLFLUSH instruction cache line size (second byte of EBX) — this number indicates the size of the cache line flushed with CLFLUSH instruction in 8-byte increments. This field was introduced in the Pentium 4 processor.
- Local APIC ID (high byte of EBX) — this number is the 8-bit ID that is assigned to the local APIC on the processor during power up. This field was introduced in the Pentium 4 processor.

INPUT EAX = 01H: Returns Feature Information in ECX and EDX

When CPUID executes with EAX set to 01H, feature information is returned in ECX and EDX.

- Figure 1-2 and Table 1-5 show encodings for ECX.
- Figure 1-3 and Table 1-6 show encodings for EDX.

For all feature flags, a 1 indicates that the feature is supported. Use Intel to properly interpret feature flags.

NOTE

Software must confirm that a processor feature is present using feature flags returned by CPUID prior to using the feature. Software should not depend on future offerings retaining all features.

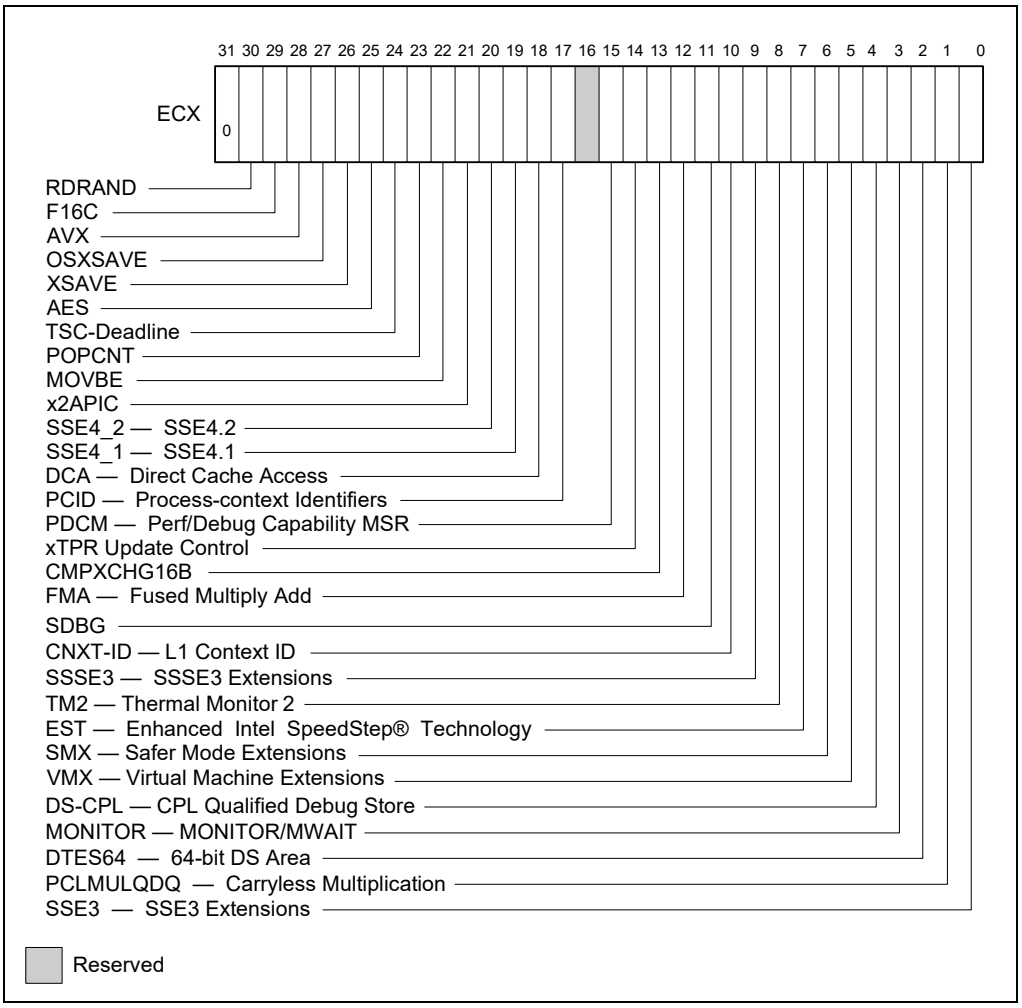

Figure 1-2. Feature Information Returned in the ECX Register

Table 1-5. Feature Information Returned in the ECX Register

Bit #	Mnemonic	Description
0	SSE3	**Streaming SIMD Extensions 3 (SSE3).** A value of 1 indicates the processor supports this technology.
1	PCLMULQDQ	A value of 1 indicates the processor supports PCLMULQDQ instruction.
2	DTES64	**64-bit DS Area.** A value of 1 indicates the processor supports DS area using 64-bit layout.
3	MONITOR	**MONITOR/MWAIT.** A value of 1 indicates the processor supports this feature.
4	DS-CPL	**CPL Qualified Debug Store.** A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5	VMX	Virtual Machine Extensions. A value of 1 indicates that the processor supports this technology.
6	SMX	Safer Mode Extensions. A value of 1 indicates that the processor supports this technology. See Chapter 6, "Safer Mode Extensions Reference".
7	EST	**Enhanced Intel SpeedStep® Technology.** A value of 1 indicates that the processor supports this technology.
8	TM2	**Thermal Monitor 2.** A value of 1 indicates whether the processor supports this technology.
9	SSSE3	A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor.

Table 1-5. Feature Information Returned in the ECX Register (Continued)

Bit #	Mnemonic	Description
10	CNXT-ID	**L1 Context ID.** A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.
11	SDBG	A value of 1 indicates the processor supports IA32_DEBUG_INTERFACE MSR for silicon debug.
12	FMA	A value of 1 indicates the processor supports FMA extensions using YMM state.
13	CMPXCHG16B	**CMPXCHG16B Available.** A value of 1 indicates that the feature is available.
14	xTPR Update Control	**xTPR Update Control.** A value of 1 indicates that the processor supports changing IA32_MISC_ENABLES[bit 23].
15	PDCM	**Perfmon and Debug Capability.** A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.
16	Reserved	Reserved
17	PCID	**Process-context identifiers.** A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.
18	DCA	A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	SSE4.1	A value of 1 indicates that the processor supports SSE4.1.
20	SSE4.2	A value of 1 indicates that the processor supports SSE4.2.
21	x2APIC	A value of 1 indicates that the processor supports x2APIC feature.
22	MOVBE	A value of 1 indicates that the processor supports MOVBE instruction.
23	POPCNT	A value of 1 indicates that the processor supports the POPCNT instruction.
24	TSC-Deadline	A value of 1 indicates that the processor's local APIC timer supports one-shot operation using a TSC deadline value.
25	AES	A value of 1 indicates that the processor supports the AESNI instruction extensions.
26	XSAVE	A value of 1 indicates that the processor supports the XSAVE/XRSTOR processor extended states feature, the XSETBV/XGETBV instructions, and XCR0.
27	OSXSAVE	A value of 1 indicates that the OS has set CR4.OSXSAVE[bit 18] to enable XSETBV/XGETBV instructions to access XCR0 and to support processor extended state management using XSAVE/XRSTOR.
28	AVX	A value of 1 indicates that processor supports AVX instructions operating on 256-bit YMM state, and three-operand encoding of 256-bit and 128-bit SIMD instructions.
29	F16C	A value of 1 indicates that processor supports 16-bit floating-point conversion instructions.
30	RDRAND	A value of 1 indicates that processor supports RDRAND instruction.
31	Not Used	Always return 0.

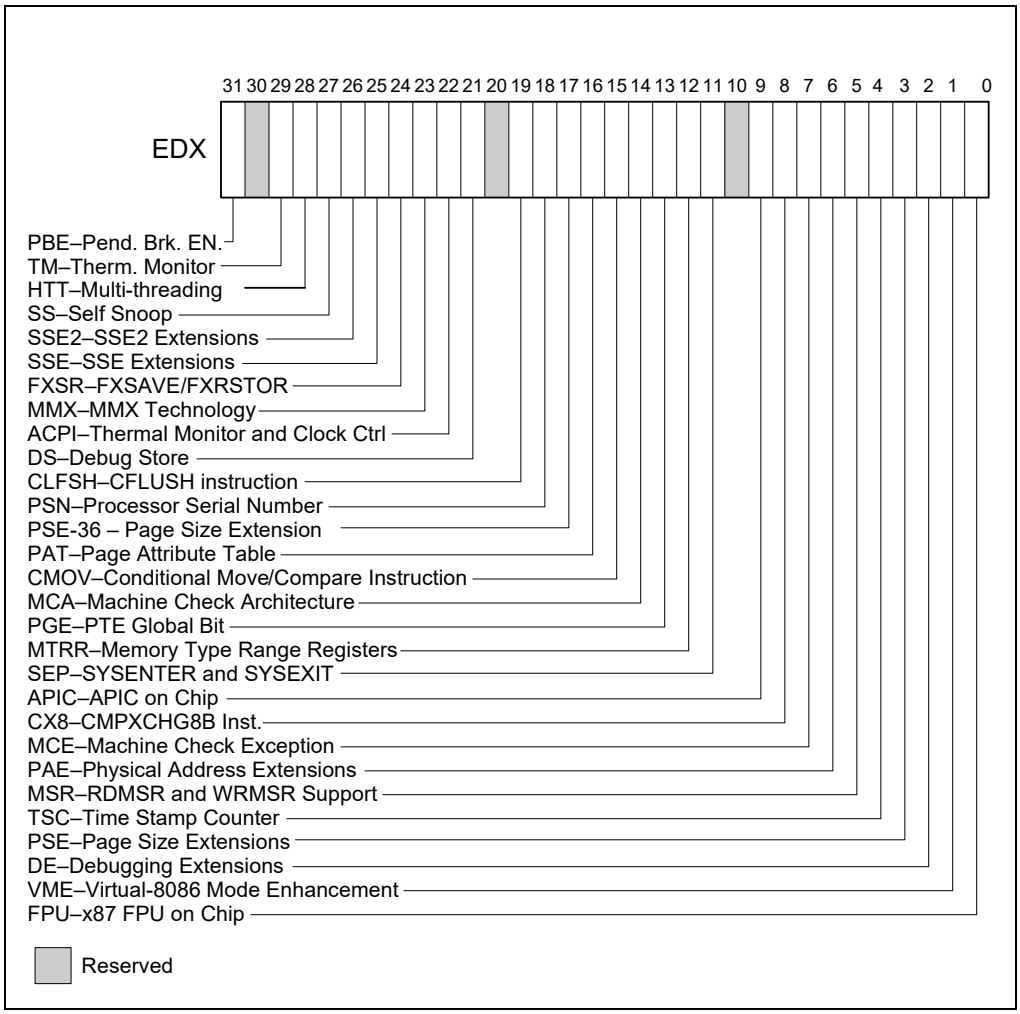

Figure 1-3. Feature Information Returned in the EDX Register

Table 1-6. More on Feature Information Returned in the EDX Register

Bit #	Mnemonic	Description
0	FPU	**Floating-point Unit On-Chip.** The processor contains an x87 FPU.
1	VME	**Virtual 8086 Mode Enhancements.** Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.
2	DE	**Debugging Extensions.** Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.
3	PSE	**Page Size Extension.** Large pages of size 4 MByte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.
4	TSC	**Time Stamp Counter.** The RDTSC instruction is supported, including CR4.TSD for controlling privilege.
5	MSR	**Model Specific Registers RDMSR and WRMSR Instructions.** The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.

Table 1-6. More on Feature Information Returned in the EDX Register(Continued)

Bit #	Mnemonic	Description
6	PAE	**Physical Address Extension.** Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2-MByte pages are supported instead of 4 Mbyte pages if PAE bit is 1. The actual number of address bits beyond 32 is not defined, and is implementation specific.
7	MCE	**Machine Check Exception.** Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature.
8	CX8	**CMPXCHG8B Instruction.** The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).
9	APIC	**APIC On-Chip.** The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated).
10	Reserved	Reserved
11	SEP	**SYSENTER and SYSEXIT Instructions.** The SYSENTER and SYSEXIT and associated MSRs are supported.
12	MTRR	**Memory Type Range Registers.** MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported.
13	PGE	**Page Global Bit.** The global bit is supported in paging-structure entries that map a page, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.
14	MCA	**Machine Check Architecture.** The Machine Check Architecture, which provides a compatible mechanism for error reporting in P6 family, Pentium 4, Intel Xeon processors, and future processors, is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.
15	CMOV	**Conditional Move Instructions.** The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported
16	PAT	**Page Attribute Table.** Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory accessed through a linear address on a 4KB granularity.
17	PSE-36	**36-Bit Page Size Extension.** 4-MByte pages addressing physical memory beyond 4 GBytes are supported with 32-bit paging. This feature indicates that upper bits of the physical address of a 4-MByte page are encoded in bits 20:13 of the page-directory entry. Such physical addresses are limited by MAXPHYADDR and may be up to 40 bits in size.
18	PSN	**Processor Serial Number.** The processor supports the 96-bit processor identification number feature and the feature is enabled.
19	CLFSH	**CLFLUSH Instruction.** CLFLUSH Instruction is supported.
20	Reserved	Reserved
21	DS	**Debug Store.** The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and precise event-based sampling (PEBS) facilities (see Chapter 23, "Introduction to Virtual-Machine Extensions," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*).
22	ACPI	**Thermal Monitor and Software Controlled Clock Facilities.** The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.
23	MMX	**Intel MMX Technology.** The processor supports the Intel MMX technology.
24	FXSR	**FXSAVE and FXRSTOR Instructions.** The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating-point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions.

Table 1-6. More on Feature Information Returned in the EDX Register(Continued)

Bit #	Mnemonic	Description
25	SSE	**SSE.** The processor supports the SSE extensions.
26	SSE2	**SSE2.** The processor supports the SSE2 extensions.
27	SS	**Self Snoop.** The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus.
28	HTT	**Max APIC IDs reserved field is Valid.** A value of 0 for HTT indicates there is only a single logical processor in the package and software should assume only a single APIC ID is reserved. A value of 1 for HTT indicates the value in CPUID.1.EBX[23:16] (the Maximum number of addressable IDs for logical processors in this package) is valid for the package.
29	TM	**Thermal Monitor.** The processor implements the thermal monitor automatic thermal control circuitry (TCC).
30	Reserved	Reserved
31	PBE	**Pending Break Enable.** The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt. Bit 10 (PBE enable) in the IA32_MISC_ENABLE MSR enables this capability.

INPUT EAX = 02H: Cache and TLB Information Returned in EAX, EBX, ECX, EDX

When CPUID executes with EAX set to 02H, the processor returns information about the processor's internal caches and TLBs in the EAX, EBX, ECX, and EDX registers.

The encoding is as follows:

- The least-significant byte in register EAX (register AL) indicates the number of times the CPUID instruction must be executed with an input value of 02H to get a complete description of the processor's caches and TLBs. The first member of the family of Pentium 4 processors will return a 01H.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. Table 1-7 shows the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache or TLB types. The descriptors may appear in any order.

Table 1-7. Encoding of Cache and TLB Descriptors

Descriptor Value	Cache or TLB Description
00H	Null descriptor
01H	Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries
02H	Instruction TLB: 4 MByte pages, 4-way set associative, 2 entries
03H	Data TLB: 4 KByte pages, 4-way set associative, 64 entries
04H	Data TLB: 4 MByte pages, 4-way set associative, 8 entries
05H	Data TLB1: 4 MByte pages, 4-way set associative, 32 entries
06H	1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size
08H	1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size
0AH	1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size
0BH	Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries
0CH	1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size
22H	3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector
23H	3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
25H	3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector

Table 1-7. Encoding of Cache and TLB Descriptors (Continued)

Descriptor Value	Cache or TLB Description
29H	3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
2CH	1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size
30H	1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size
40H	No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache
41H	2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size
42H	2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size
43H	2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size
44H	2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size
45H	2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size
46H	3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size
47H	3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size
49H	3rd-level cache: 4MB, 16-way set associative, 64-byte line size (Intel Xeon processor MP, Family 0FH, Model 06H); 2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size
4AH	3rd-level cache: 6MByte, 12-way set associative, 64 byte line size
4BH	3rd-level cache: 8MByte, 16-way set associative, 64 byte line size
4CH	3rd-level cache: 12MByte, 12-way set associative, 64 byte line size
4DH	3rd-level cache: 16MByte, 16-way set associative, 64 byte line size
4EH	2nd-level cache: 6MByte, 24-way set associative, 64 byte line size
50H	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries
51H	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries
52H	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries
56H	Data TLB0: 4 MByte pages, 4-way set associative, 16 entries
57H	Data TLB0: 4 KByte pages, 4-way associative, 16 entries
5BH	Data TLB: 4 KByte and 4 MByte pages, 64 entries
5CH	Data TLB: 4 KByte and 4 MByte pages,128 entries
5DH	Data TLB: 4 KByte and 4 MByte pages,256 entries
60H	1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size
66H	1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size
67H	1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size
68H	1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size
70H	Trace cache: 12 K-µop, 8-way set associative
71H	Trace cache: 16 K-µop, 8-way set associative
72H	Trace cache: 32 K-µop, 8-way set associative
78H	2nd-level cache: 1 MByte, 4-way set associative, 64byte line size
79H	2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7AH	2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7BH	2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7CH	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector
7DH	2nd-level cache: 2 MByte, 8-way set associative, 64byte line size
7FH	2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size
82H	2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size

Table 1-7. Encoding of Cache and TLB Descriptors (Continued)

Descriptor Value	Cache or TLB Description
83H	2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size
84H	2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size
85H	2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size
86H	2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size
87H	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size
B0H	Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries
B1H	Instruction TLB: 2M pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries
B3H	Data TLB: 4 KByte pages, 4-way set associative, 128 entries
B4H	Data TLB1: 4 KByte pages, 4-way associative, 256 entries
F0H	64-Byte prefetching
F1H	128-Byte prefetching

Example 1-1. Example of Cache and TLB Interpretation

The first member of the family of Pentium 4 processors returns the following information about caches and TLBs when the CPUID executes with an input value of 2:

EAX 66 5B 50 01H
EBX 0H
ECX 0H
EDX 00 7A 70 00H

Which means:

- The least-significant byte (byte 0) of register EAX is set to 01H. This indicates that CPUID needs to be executed once with an input value of 2 to retrieve complete information about caches and TLBs.
- The most-significant bit of all four registers (EAX, EBX, ECX, and EDX) is set to 0, indicating that each register contains valid 1-byte descriptors.
- Bytes 1, 2, and 3 of register EAX indicate that the processor has:
  - 50H - a 64-entry instruction TLB, for mapping 4-KByte and 2-MByte or 4-MByte pages.
  - 5BH - a 64-entry data TLB, for mapping 4-KByte and 4-MByte pages.
  - 66H - an 8-KByte 1st level data cache, 4-way set associative, with a 64-Byte cache line size.
- The descriptors in registers EBX and ECX are valid, but contain NULL descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor has:
  - 00H - NULL descriptor.
  - 70H - Trace cache: 12 K-µop, 8-way set associative.
  - 7AH - a 256-KByte 2nd level cache, 8-way set associative, with a sectored, 64-byte cache line size.
  - 00H - NULL descriptor.

INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level

When CPUID executes with EAX set to 04H and ECX contains an index value, the processor returns encoded data that describe a set of deterministic cache parameters (for the cache level associated with the input in ECX). Valid index values start from 0.

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 1-2.

The CPUID leaf 4 also reports data that can be used to derive the topology of processor cores in a physical package. This information is constant for all valid index values. Software can query the raw data reported by executing CPUID with EAX=04H and ECX=0H and use it as part of the topology enumeration algorithm described in Chapter 8, “Multiple-Processor Management,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

INPUT EAX = 05H: Returns MONITOR and MWAIT Features

When CPUID executes with EAX set to 05H, the processor returns information about features available to MONITOR/MWAIT instructions. The MONITOR instruction is used for address-range monitoring in conjunction with MWAIT instruction. The MWAIT instruction optionally provides additional extensions for advanced power management. See Table 1-2.

INPUT EAX = 06H: Returns Thermal and Power Management Features

When CPUID executes with EAX set to 06H, the processor returns information about thermal and power management features. See Table 1-2.

INPUT EAX = 07H: Returns Structured Extended Feature Enumeration Information

When CPUID executes with EAX set to 07H and ECX = 0H, the processor returns information about the maximum number of sub-leaves that contain extended feature flags. See Table 1-2.

When CPUID executes with EAX set to 07H and ECX = n (n > 1and less than the number of non-zero bits in CPUID.(EAX=07H, ECX= 0H).EAX, the processor returns information about extended feature flags. See Table 1-2. In sub-leaf 0, only EAX has the number of sub-leaves. In sub-leaf 0, EBX, ECX & EDX all contain extended feature flags.

Table 1-8. Structured Extended Feature Leaf, Function 0, EBX Register

Bit #	Mnemonic	Description
0	RWFSGSBASE	A value of 1 indicates the processor supports RD/WR FSGSBASE instructions
1-31	Reserved	Reserved

INPUT EAX = 09H: Returns Direct Cache Access Information

When CPUID executes with EAX set to 09H, the processor returns information about Direct Cache Access capabilities. See Table 1-2.

INPUT EAX = 0AH: Returns Architectural Performance Monitoring Features

When CPUID executes with EAX set to 0AH, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 1-2) is greater than Pn 0. See Table 1-2.

For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 17, “Debug, Branch Profile, TSC, and Quality of Service,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

INPUT EAX = 0BH: Returns Extended Topology Information

When CPUID executes with EAX set to 0BH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 0BH by verifying (a) the highest leaf index supported by CPUID is >= 0BH, and (b) CPUID.0BH:EBX[15:0] reports a non-zero value. See Table 1-2.

INPUT EAX = 0DH: Returns Processor Extended States Enumeration Information

When CPUID executes with EAX set to 0DH and ECX = 0H, the processor returns information about the bit-vector representation of all processor state extensions that are supported in the processor and storage size requirements of the XSAVE/XRSTOR area. See Table 1-2.

When CPUID executes with EAX set to 0DH and ECX = n (n > 1, and is a valid sub-leaf index), the processor returns information about the size and offset of each processor extended state save area within the XSAVE/XRSTOR area. See Table 1-2. Software can use the forward-extendable technique depicted below to query the valid sub-leaves and obtain size and offset information for each processor extended state save area:

```
For i = 2 to 62 // sub-leaf 1 is reserved
    IF (CPUID.(EAX=0DH, ECX=0):VECTOR[i] = 1 ) // VECTOR is the 64-bit value of EDX:EAX
        Execute CPUID.(EAX=0DH, ECX = i) to examine size and offset for sub-leaf i;
    FI;
```

INPUT EAX = 0FH: Returns Platform Quality of Service (PQoS) Monitoring Enumeration Information

When CPUID executes with EAX set to 0FH and ECX = 0, the processor returns information about the bit-vector representation of QoS monitoring resource types that are supported in the processor and maximum range of RMID values the processor can use to monitor of any supported resource types. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS monitoring capability available for that type. See Table 1-2.

When CPUID executes with EAX set to 0FH and ECX = n (n >= 1, and is a valid ResID), the processor returns information software can use to program IA32_PQR_ASSOC, IA32_QM_EVTSEL MSRs before reading QoS data from the IA32_QM_CTR MSR.

INPUT EAX = 10H: Returns Platform Quality of Service (PQoS) Enforcement Enumeration Information

When CPUID executes with EAX set to 10H and ECX = 0, the processor returns information about the bit-vector representation of QoS Enforcement resource types that are supported in the processor. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS enforcement capability available for that type. See Table 1-2.

When CPUID executes with EAX set to 10H and ECX = n (n >= 1, and is a valid ResID), the processor returns information about available classes of service and range of QoS mask MSRs that software can use to configure each class of services using capability bit masks in the QoS Mask registers, IA32_resourceType_Mask_n.

INPUT EAX = 14H: Returns Intel Processor Trace Enumeration Information

When CPUID executes with EAX set to 14H and ECX = 0H, the processor returns information about Intel Processor Trace extensions. See Table 1-2.

When CPUID executes with EAX set to 14H and ECX = n (n > 0 and less than the number of non-zero bits in CPUID.(EAX=14H, ECX= 0H).EAX), the processor returns information about packet generation in Intel Processor Trace. See Table 1-2.

INPUT EAX = 15H: Returns Time Stamp Counter and Core Crystal Clock Information

When CPUID executes with EAX set to 15H and ECX = 0H, the processor returns information about Time Stamp Counter and Core Crystal Clock. See Table 1-2.

INPUT EAX = 16H: Returns Processor Frequency Information

When CPUID executes with EAX set to 16H, the processor returns information about Processor Frequency Information. See Table 1-2.

INPUT EAX = 17H: Returns System-On-Chip Information

When CPUID executes with EAX set to 17H, the processor returns information about the System-On-Chip Vendor Attribute Enumeration. See Table 1-2.

INPUT EAX = 18H: Returns Deterministic Address Translation Parameters Information

When CPUID executes with EAX set to 18H, the processor returns information about the Deterministic Address Translation Parameters. See Table 1-2.

METHODS FOR RETURNING BRANDING INFORMATION

Use the following techniques to access branding information:

1. Processor brand string method; this method also returns the processor's maximum operating frequency
2. Processor brand index; this method uses a software supplied brand string table.

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: "Identification of Earlier IA-32 Processors" in Chapter 16 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

The Processor Brand String Method

Figure 1-4 describes the algorithm used for detection of the brand string. Processor brand identification software should execute this algorithm on all Intel 64 and IA-32 processors.

This method (introduced with Pentium 4 processors) returns an ASCII brand identification string and the maximum operating frequency of the processor to the EAX, EBX, ECX, and EDX registers.

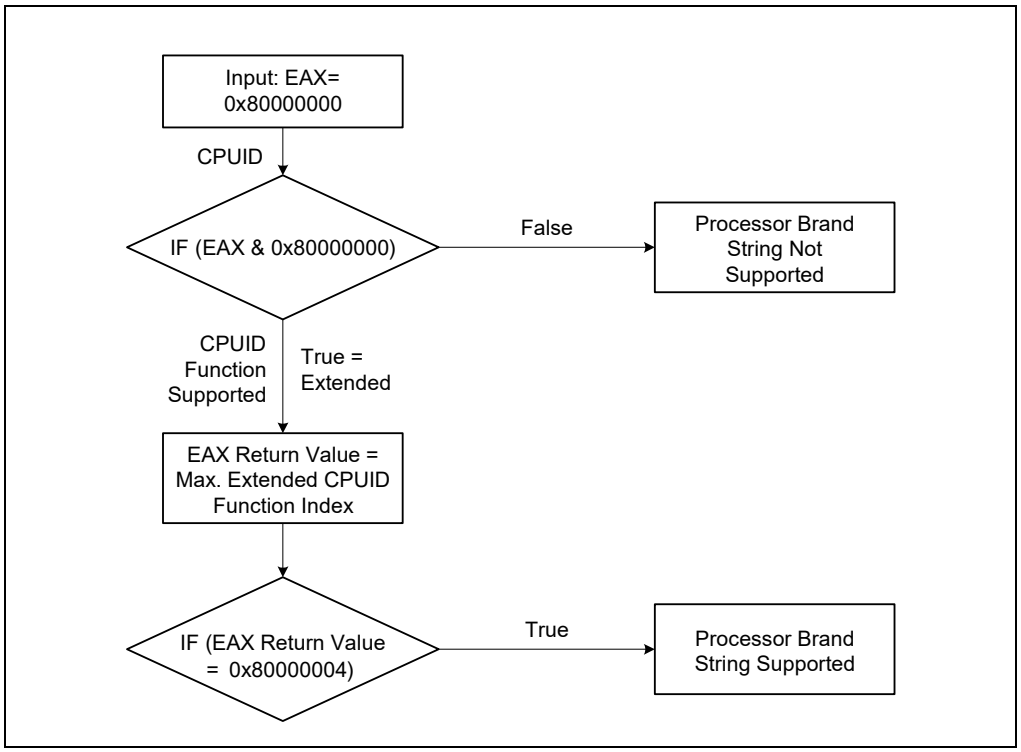

Figure 1-4. Determination of Support for the Processor Brand String

How Brand Strings Work

To use the brand string method, execute CPUID with EAX input of 8000002H through 80000004H. For each input value, CPUID returns 16 ASCII characters using EAX, EBX, ECX, and EDX. The returned string will be NULL-terminated.

Table 1-9 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

Table 1-9. Processor Brand String Returned with Pentium 4 Processor

EAX Input Value	Return Values	ASCII Equivalent
80000002H	EAX = 20202020H	" "
	EBX = 20202020H	" "
	ECX = 20202020H	" "
	EDX = 6E492020H	"nI "
80000003H	EAX = 286C6574H	"(let"
	EBX = 50202952H	"P)R"
	ECX = 69746E65H	"itne"
	EDX = 52286D75H	"R(mu"
80000004H	EAX = 20342029H	" 4)"
	EBX = 20555043H	" UPC"
	ECX = 30303531H	"0051"
	EDX = 007A484DH	"\0zHM"

Extracting the Maximum Processor Frequency from Brand Strings

Figure 1-5 provides an algorithm which software can use to extract the maximum processor operating frequency from the processor brand string.

NOTE

When a frequency is given in a brand string, it is the maximum qualified frequency of the processor, not the frequency at which the processor is currently running.

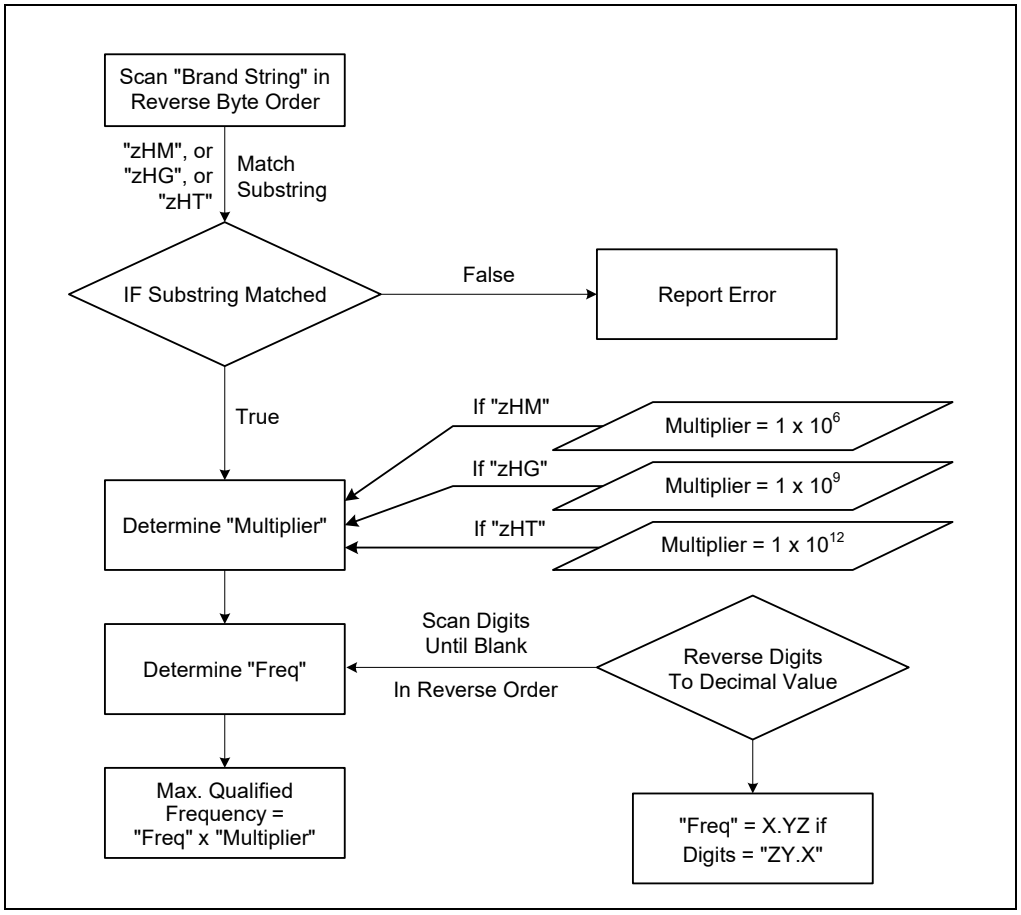

Figure 1-5. Algorithm for Extracting Maximum Processor Frequency

The Processor Brand Index Method

The brand index method (introduced with Pentium® III Xeon® processors) provides an entry point into a brand identification table that is maintained in memory by system software and is accessible from system- and user-level code. In this table, each brand index is associate with an ASCII brand identification string that identifies the official Intel family and model number of a processor.

When CPUID executes with EAX set to 01H, the processor returns a brand index to the low byte in EBX. Software can then use this index to locate the brand identification string for the processor in the brand identification table. The first entry (brand index 0) in this table is reserved, allowing for backward compatibility with processors that do not support the brand identification feature. Starting with processor signature family ID = 0FH, model = 03H, brand index method is no longer supported. Use brand string method instead.

Table 1-10 shows brand indices that have identification strings associated with them.

Table 1-10. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings

Brand Index	Brand String
00H	This processor does not support the brand identification feature
01H	Intel(R) Celeron(R) processor[1]
02H	Intel(R) Pentium(R) III processor[1]
03H	Intel(R) Pentium(R) III Xeon(R) processor; If processor signature = 000006B1h, then Intel(R) Celeron(R) processor
04H	Intel(R) Pentium(R) III processor
06H	Mobile Intel(R) Pentium(R) III processor-M
07H	Mobile Intel(R) Celeron(R) processor[1]
08H	Intel(R) Pentium(R) 4 processor
09H	Intel(R) Pentium(R) 4 processor
0AH	Intel(R) Celeron(R) processor[1]
0BH	Intel(R) Xeon(R) processor; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor MP
0CH	Intel(R) Xeon(R) processor MP
0EH	Mobile Intel(R) Pentium(R) 4 processor-M; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor
0FH	Mobile Intel(R) Celeron(R) processor[1]
11H	Mobile Genuine Intel(R) processor
12H	Intel(R) Celeron(R) M processor
13H	Mobile Intel(R) Celeron(R) processor[1]
14H	Intel(R) Celeron(R) processor
15H	Mobile Genuine Intel(R) processor
16H	Intel(R) Pentium(R) M processor
17H	Mobile Intel(R) Celeron(R) processor[1]
18H – 0FFH	RESERVED

NOTES:
1.Indicates versions of these processors that were introduced after the Pentium III

IA-32 Architecture Compatibility

CPUID is not supported in early models of the Intel486 processor or in any IA-32 processor earlier than the Intel486 processor.

Operation

```
IA32_BIOS_SIGN_ID MSR ← Update with installed microcode revision number;

CASE (EAX) OF
    EAX = 0:
        EAX ← Highest basic function input value understood by CPUID;
        EBX ← Vendor identification string;
        EDX ← Vendor identification string;
        ECX ← Vendor identification string;
    BREAK;
    EAX = 1H:
        EAX[3:0] ← Stepping ID;
```

```
        EAX[7:4] ← Model;
        EAX[11:8] ← Family;
        EAX[13:12] ← Processor type;
        EAX[15:14] ← Reserved;
        EAX[19:16] ← Extended Model;
        EAX[27:20] ← Extended Family;
        EAX[31:28] ← Reserved;
        EBX[7:0] ← Brand Index; (* Reserved if the value is zero. *)
        EBX[15:8] ← CLFLUSH Line Size;
        EBX[16:23] ← Reserved; (* Number of threads enabled = 2 if MT enable fuse set. *)
        EBX[24:31] ← Initial APIC ID;
        ECX ← Feature flags; (* See Figure 1-2. *)
        EDX ← Feature flags; (* See Figure 1-3. *)
    BREAK;
    EAX = 2H:
        EAX ← Cache and TLB information;
        EBX ← Cache and TLB information;
        ECX ← Cache and TLB information;
        EDX ← Cache and TLB information;
    BREAK;
    EAX = 3H:
        EAX ← Reserved;
        EBX ← Reserved;
        ECX ← ProcessorSerialNumber[31:0];
        (* Pentium III processors only, otherwise reserved. *)
        EDX ← ProcessorSerialNumber[63:32];
        (* Pentium III processors only, otherwise reserved. *
    BREAK
    EAX = 4H:
        EAX ← Deterministic Cache Parameters Leaf; (* See Table 1-2. *)
        EBX ← Deterministic Cache Parameters Leaf;
        ECX ← Deterministic Cache Parameters Leaf;
        EDX ← Deterministic Cache Parameters Leaf;
    BREAK;
    EAX = 5H:
        EAX ← MONITOR/MWAIT Leaf; (* See Table 1-2. *)
        EBX ← MONITOR/MWAIT Leaf;
        ECX ← MONITOR/MWAIT Leaf;
        EDX ← MONITOR/MWAIT Leaf;
    BREAK;
    EAX = 6H:
        EAX ← Thermal and Power Management Leaf; (* See Table 1-2. *)
        EBX ← Thermal and Power Management Leaf;
        ECX ← Thermal and Power Management Leaf;
        EDX ← Thermal and Power Management Leaf;
    BREAK;
    EAX = 7H:
        EAX ← Structured Extended Feature Leaf; (* See Table 1-2. *);
        EBX ← Structured Extended Feature Leaf;
        ECX ← Structured Extended Feature Leaf;
        EDX ← Structured Extended Feature Leaf;
    BREAK;
    EAX = 8H:
        EAX ← Reserved = 0;
```

```
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    EAX = 9H:
        EAX ← Direct Cache Access Information Leaf; (* See Table 1-2. *)
        EBX ← Direct Cache Access Information Leaf;
        ECX ← Direct Cache Access Information Leaf;
        EDX ← Direct Cache Access Information Leaf;
    BREAK;
    EAX = AH:
        EAX ← Architectural Performance Monitoring Leaf; (* See Table 1-2. *)
        EBX ← Architectural Performance Monitoring Leaf;
        ECX ← Architectural Performance Monitoring Leaf;
        EDX ← Architectural Performance Monitoring Leaf;
        BREAK
    EAX = BH:
        EAX ← Extended Topology Enumeration Leaf; (* See Table 1-2. *)
        EBX ← Extended Topology Enumeration Leaf;
        ECX ← Extended Topology Enumeration Leaf;
        EDX ← Extended Topology Enumeration Leaf;
    BREAK;
    EAX = CH:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    EAX = DH:
        EAX ← Processor Extended State Enumeration Leaf; (* See Table 1-2. *)
        EBX ← Processor Extended State Enumeration Leaf;
        ECX ← Processor Extended State Enumeration Leaf;
        EDX ← Processor Extended State Enumeration Leaf;
    BREAK;
    EAX = EH:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    EAX = FH:
        EAX ← Platform Quality of Service Monitoring Enumeration Leaf; (* See Table 1-2. *)
        EBX ← Platform Quality of Service Monitoring Enumeration Leaf;
        ECX ← Platform Quality of Service Monitoring Enumeration Leaf;
        EDX ← Platform Quality of Service Monitoring Enumeration Leaf;
    BREAK;
    EAX = 10H:
        EAX ← Platform Quality of Service Enforcement Enumeration Leaf; (* See Table 1-2. *)
        EBX ← Platform Quality of Service Enforcement Enumeration Leaf;
        ECX ← Platform Quality of Service Enforcement Enumeration Leaf;
        EDX ← Platform Quality of Service Enforcement Enumeration Leaf;
    BREAK;
    EAX = 14H:
        EAX ← Intel Processor Trace Enumeration Leaf; (* See Table 1-2. *)
```

```
        EBX ← Intel Processor Trace Enumeration Leaf;
        ECX ← Intel Processor Trace Enumeration Leaf;
        EDX ← Intel Processor Trace Enumeration Leaf;
    BREAK;
    EAX = 15H:
        EAX ← Time Stamp Counter and Core Crystal Clock Information Leaf; (* See Table 1-2. *)
        EBX ← Time Stamp Counter and Core Crystal Clock Information Leaf;
        ECX ← Time Stamp Counter and Core Crystal Clock Information Leaf;
        EDX ← Time Stamp Counter and Core Crystal Clock Information Leaf;
    BREAK;
    EAX = 16H:
        EAX ← Processor Frequency Information Enumeration Leaf; (* See Table 1-2. *)
        EBX ← Processor Frequency Information Enumeration Leaf;
        ECX ← Processor Frequency Information Enumeration Leaf;
        EDX ← Processor Frequency Information Enumeration Leaf;
    BREAK;
    EAX = 17H:
        EAX ← System-On-Chip Vendor Attribute Enumeration Leaf; (* See Table 1-2. *)
        EBX ← System-On-Chip Vendor Attribute Enumeration Leaf;
        ECX ← System-On-Chip Vendor Attribute Enumeration Leaf;
        EDX ← System-On-Chip Vendor Attribute Enumeration Leaf;
    BREAK;
    EAX = 18H:
        EAX ← Deterministic Address Translation Parameters Enumeration Leaf; (* See Table 1-2. *)
        EBX ← Deterministic Address Translation Parameters Enumeration Leaf;
        ECX ←Deterministic Address Translation Parameters Enumeration Leaf;
        EDX ← Deterministic Address Translation Parameters Enumeration Leaf;
    BREAK;
    EAX = 80000000H:
        EAX ← Highest extended function input value understood by CPUID;
        EBX ← Reserved;
        ECX ← Reserved;
        EDX ← Reserved;
    BREAK;
    EAX = 80000001H:
        EAX ← Reserved;
        EBX ← Reserved;
        ECX ← Extended Feature Bits (* See Table 1-2.*);
        EDX ← Extended Feature Bits (* See Table 1-2. *);
    BREAK;
    EAX = 80000002H:
        EAX ← Processor Brand String;
        EBX ← Processor Brand String, continued;
        ECX ← Processor Brand String, continued;
        EDX ← Processor Brand String, continued;
    BREAK;
    EAX = 80000003H:
        EAX ← Processor Brand String, continued;
        EBX ← Processor Brand String, continued;
        ECX ← Processor Brand String, continued;
        EDX ← Processor Brand String, continued;
    BREAK;
    EAX = 80000004H:
        EAX ← Processor Brand String, continued;
```

```
        EBX ← Processor Brand String, continued;
        ECX ← Processor Brand String, continued;
        EDX ← Processor Brand String, continued;
    BREAK;
    EAX = 80000005H:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    EAX = 80000006H:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Cache information;
        EDX ← Reserved = 0;
    BREAK;
    EAX = 80000007H:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    EAX = 80000008H:
        EAX ← Reserved = 0;
        EBX ← Reserved = 0;
        ECX ← Reserved = 0;
        EDX ← Reserved = 0;
    BREAK;
    DEFAULT: (* EAX = Value outside of recognized range for CPUID. *)
        (* If the highest basic information leaf data depend on ECX input value, ECX is honored.*)
        EAX ← Reserved; (* Information returned for highest basic information leaf. *)
        EBX ← Reserved; (* Information returned for highest basic information leaf. *)
        ECX ← Reserved; (* Information returned for highest basic information leaf. *)
        EDX ← Reserved; (* Information returned for highest basic information leaf. *)
    BREAK;
ESAC;
```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

In earlier IA-32 processors that do not support the CPUID instruction, execution of the instruction results in an invalid opcode (#UD) exception being generated.§

1.4 COMPRESSED DISPLACEMENT (DISP8*N) SUPPORT IN EVEX

For memory addressing using disp8 form, EVEX-encoded instructions always use a compressed displacement scheme by multiplying disp8 in conjunction with a scaling factor N that is determined based on the vector length, the value of EVEX.b bit (embedded broadcast) and the input element size of the instruction. In general, the factor N corresponds to the number of bytes characterizing the internal memory operation of the input operand (e.g., 64 when the accessing a full 512-bit memory vector). The scale factor N is listed in Table 1-11 and Table 1-12 below, where EVEX encoded instructions are classified using the **tupletype** attribute. The scale factor N of each tupletype is listed based on the vector length (VL) and other factors affecting it.

Table 1-11 covers EVEX-encoded instructions which has a load semantic in conjunction with additional computational or data element movement operation, operating either on the full vector or half vector (due to conversion of numerical precision from a wider format to narrower format). EVEX.b is supported for such instructions for data element sizes which are either dword or qword.

EVEX-encoded instruction that are pure load/store, and "Load+op" instruction semantic that operate on data element size less then dword do not support broadcasting using EVEX.b. These are listed in Table 1-12. Table 1-12 also includes many broadcast instructions which perform broadcast using a subset of data elements without using EVEX.b. These instructions and a few data element size conversion instruction are covered in Table 1-12. Instruction classified in Table 1-12 do not use EVEX.b and EVEX.b must be 0, otherwise #UD will occur.

The tupletype will be referenced in the instruction operand encoding table in the reference page of each instruction, providing the cross reference for the scaling factor N to encoding memory addressing operand.

Note that the disp8*N rules still apply when using 16b addressing.

Table 1-11. Compressed Displacement (DISP8*N) Affected by Embedded Broadcast

TupleType	EVEX.b	InputSize	EVEX.W	Broadcast	N (VL=128)	N (VL=256)	N (VL= 512)	Comment
Full	0	32bit	0	none	16	32	64	Load+Op (Full Vector Dword/Qword)
	1	32bit	0	{1tox}	4	4	4	
	0	64bit	1	none	16	32	64	
	1	64bit	1	{1tox}	8	8	8	
Half	0	32bit	0	none	8	16	32	Load+Op (Half Vector)
	1	32bit	0	{1tox}	4	4	4	

Table 1-12. EVEX DISP8*N for Instructions Not Affected by Embedded Broadcast

TupleType	InputSize	EVEX.W	N (VL= 128)	N (VL= 256)	N (VL= 512)	Comment
Full Mem	N/A	N/A	16	32	64	Load/store or subDword full vector
Tuple1 Scalar	8bit	N/A	1	1	1	1Tuple
	16bit	N/A	2	2	2	
	32bit	0	4	4	4	
	64bit	1	8	8	8	
Tuple1 Fixed	32bit	N/A	4	4	4	1 Tuple, memsize not affected by EVEX.W
	64bit	N/A	8	8	8	
Tuple1_4X	32bit	0	16[1]	N/A	16	4FMA(PS)
Tuple2	32bit	0	8	8	8	Broadcast (2 elements)
	64bit	1	NA	16	16	
Tuple4	32bit	0	NA	16	16	Broadcast (4 elements)
	64bit	1	NA	NA	32	
Tuple8	32bit	0	NA	NA	32	Broadcast (8 elements)

Table 1-12. EVEX DISP8*N for Instructions Not Affected by Embedded Broadcast(Continued)

TupleType	InputSize	EVEX.W	N (VL= 128)	N (VL= 256)	N (VL= 512)	Comment
Half Mem	N/A	N/A	8	16	32	SubQword Conversion
Quarter Mem	N/A	N/A	4	8	16	SubDword Conversion
Eighth Mem	N/A	N/A	2	4	8	SubWord Conversion
Mem128	N/A	N/A	16	16	16	Shift count from memory
MOVDDUP	N/A	N/A	8	32	64	VMOVDDUP

NOTES:

1. Scalar

CHAPTER 2
INSTRUCTION SET REFERENCE, A-Z

Instructions described in this document follow the general documentation convention established in *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*.

2.1 INSTRUCTION SET REFERENCE

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 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 VGF2P8MULB xmm1, xmm2, xmm3/m128	B	V/V	AVX GFNI	Multiplies elements in the finite field GF(2^8).
VEX.NDS.256.66.0F38.W0 CF /r VGF2P8MULB ymm1, ymm2, ymm3/m256	B	V/V	AVX GFNI	Multiplies elements in the finite field GF(2^8).
EVEX.NDS.128.66.0F38.W0 CF /r VGF2P8MULB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL GFNI	Multiplies elements in the finite field GF(2^8).
EVEX.NDS.256.66.0F38.W0 CF /r VGF2P8MULB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL GFNI	Multiplies elements in the finite field GF(2^8).
EVEX.NDS.512.66.0F38.W0 CF /r VGF2P8MULB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512F 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);

CHAPTER 3
INSTRUCTION SET REFERENCE UNIQUE TO PROCESSORS BASED ON THE KNIGHTS MILL MICROARCHITECTURE

This chapter describes the instruction set that is unique to processors based on the Knights Mill microarchitecture.

3.1 INSTRUCTION SET REFERENCE

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.

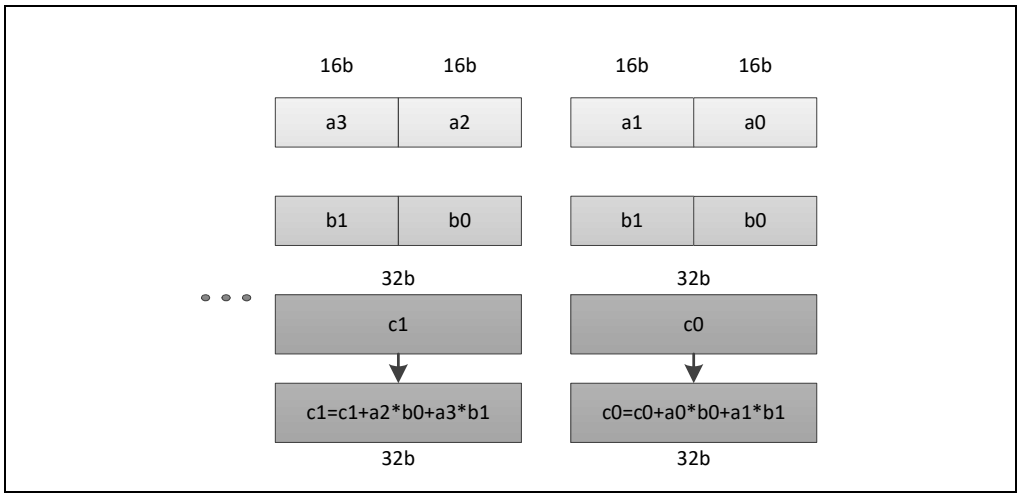

Figure 3-1. Register Source-Block Dot Product of Two Signed Word Operands with Doubleword Accumulation[1]

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.

CHAPTER 4
EPT-BASED SUB-PAGE PERMISSIONS

4.1 INTRODUCTION

This chapter describes an EPT-based sub-page permissions capability to allow Virtual Machine Monitors (VMM) to specify write-permission for guest physical memory at a sub-page (128 byte) granularity. When this capability is utilized, the CPU establishes write-access permissions for sub-page regions of 4-KByte pages as specified by the VMM. EPT-based sub-page permissions is intended to enable fine-grained memory write enforcement by a VMM for security (guest OS monitoring).

4.2 VMCS CHANGES

A new secondary processor-based VM-execution control is defined as “sub-page write permission”. The bit position of this control is 23.

If bit 31 of the primary processor-based VM-execution controls is 0, the logical processor operates as if the sub-page write permission VM-execution control is 0.

A new 64-bit control field is defined as “sub-page permission table pointer” (SPPTP). The encodings for this field are 00002030H (all 64 bits in 64-bit mode; low 32 bits in legacy mode) and 00002031H (high 32 bits).

4.3 CHANGES TO EPT PAGING-STRUCTURE ENTRIES

Bit 61 of an EPT leaf paging-structure entry that maps a 4-KByte page is defined as a “Sub-Page Permission” (SPP bit). Setting this bit allows write permissions for the mapped page to be enforced on a sub-page basis (see Section 6.4). The processor ignores this bit in all other EPT paging-structure entries (as it does if the “sub-page write permission” VM-execution control is 0).

4.4 CHANGES TO GUEST-PHYSICAL ACCESSES

If the logical processor is in VMX non-root operation with EPT enabled, and if the sub-page write permission VM-execution control (see Section 4.2) is 0, an EPT violation occurs if a memory store uses a guest-physical address and the write-access bit (bit 1) is clear in any of the EPT paging-structure entries used to translate the guest physical address. (This is same as legacy behavior.)

If the sub-page write permission VM-execution control is 1, treatment of write accesses to guest-physical accesses depends on the state of the accumulated write-access bit (position 1) and sub-page permission bit (position 61) in the leaf EPT paging-structure used to translate guest-physical addresses.

If the translation of the linear address accessed results in a guest-physical address which has the accumulated write-access bit set to 0 and the SPP bit set to 1 in the leaf EPT paging-structure entry that maps a 4KB page, then the processor will look up a VMM-managed Sub-Page Permission Table (SPPT). The processor uses the guest-physical address and bits 11:7 of the address accessed to lookup the SPPT to fetch a write permission bit for the 128 byte sub-page region being accessed within the 4-KByte guest-physical page. If the sub-page region write permission bit is set, the write is allowed; otherwise the write is disallowed and results in an EPT violation normally.

In other cases, the processor does not consult the SPPT. Guest-physical pages mapped via leaf EPT-paging-structures for which the accumulated write-access bit and the SPP bits are both clear (0) generate EPT violations on memory writes accesses. Guest-physical pages mapped via EPT-paging-structure for which the accumulated write-access bit is set (1) allow writes, effectively ignoring the SPP bit on the leaf EPT-paging structure.

4.5 SUB-PAGE PERMISSION TABLE

The sub-page permission table is referenced via a 64-bit control field called Sub-Page Permission Table Pointer (SPPTP) which contains a 4K-aligned physical address. The SPPT allows specification of write-permissions for 32 128 byte sub-page regions for 4KB guest-physical memory pages accessed via the EPT. The format of SPPTP is shown in Table 4-1 below.

Table 4-1. Format of SPPTP

Bit Position	Contents
11:0	Reserved.
M-1:12	Bits M-1:12 of the physical address of the 4-KByte aligned SPPT L4 table.[1]
63:N	Reserved (must be 0).

NOTES:

1. M is the physical-address width supported by the processor.

The memory type used for SPPT accesses will be the memory type reported in IA32_VMX_BASIC MSR.

When SPPT is in use, write accesses to any guest-physical addresses produced via a mapping for a 4KB page in the EPT can be controlled at a 128 byte granularity sub-page region within the 4KB guest-physical page. Note that reads and instruction fetches are not affected by the SPPT.

4.5.1 SPPT Overview

SPPT is active when the sub-page write permission VM-execution control is 1. SPPT looks up the guest-physical addresses to derive a 64 bit sub-page permission value containing sub-page region write permissions. The lookup from guest-physical addresses to the sub-page region permissions is determined by a set of SPPT paging structures. Section 4.5.2 gives the details of the SPPT structures.

When the sub-page write permission VM-execution control is 1, the SPPT is used to look up write permission bits for the 128 byte sub-page regions contained in the 4KB guest-physical page. EPT specifies the 4KB page-level privileges that software is allowed when accessing the guest-physical address, whereas SPPT defines the write permissions for software at the 128 byte granularity regions within a 4KB page. Similar to EPT, a logical processor uses SPPT to look up sub-page region write permissions for guest-physical addresses only when those addresses are used to access memory.

4.5.2 Operation of SPPT-based Write-Permission

The SPPT translation mechanism uses only bits 47:7 of a guest-physical address. The SPPT is a 4-level paging structure. Four SPPT paging structures are accessed to look up a sub-page region write permission bit for a guest-physical address. The 48 bits are partitioned by the logical processor to traverse the SPPT paging structures as follows.

— A 4KB naturally aligned SPPT L4 table is located at the physical address specified in bits 51:12 of the SPPTP. An SPPT L4 table comprises 512 64-bit entries (SPPT L4Es). An SPPT L4E is selected at the physical address defined as follows.

- Bits 63:52 are all 0.
- Bits 51:12 are from the SPPTP.
- Bits 11:3 are bits 47:39 of the guest-physical address.
- Bits 2:0 are all 0.

The format of a SPPT L4E is given in Table 4-2.

Table 4-2. Format of the SPPT L4E

Bit Position	Contents
0	Valid entry when set; indicates whether the entry is present.
11:1	Reserved (must be 0).
M-1:12	Physical address of 4KB naturally aligned SPPT L3 table referenced by this entry.[1]
63:M	Reserved (must be 0).

NOTES:

1. M is the physical-address width supported by the processor. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

— A 4KB naturally aligned SPPT L3 table is located at the physical address specified in bits 51:12 of the SPPT L4E. An SPPT L3 table comprises 512 64-bit entries (SPPT L3Es). An SPPT L3E is selected at the physical address defined as follows.

- Bits 63:52 are all 0.
- Bits 51:12 are from the SPPT L4E.
- Bits 11:3 are bits 38:30 of the guest-physical address.
- Bits 2:0 are all 0.

The format of the SPPT L3E is the same as that given in Table 4-2 for for SPPT L4Es. The SPPT L3E references a 4KB naturally aligned SPPT L2 Table.

— A 4KB naturally aligned SPPT L2 table is located at the physical address specified in bits 51:12 of the SPPT L3E. An SPPT L2 table comprises 512 64-bit entries (SPPT L2Es). An SPPT L2E is selected at the physical address defined as follows.

- Bits 63:52 are all 0.
- Bits 51:12 are from the SPPT L3E.
- Bits 11:3 are bits 29:21 of the guest-physical address.
- Bits 2:0 are all 0.

The format of a SPPT L2E is the same as that given in Table 4-2 for SPPT L4Es. The SPPT L2E references a 4KB naturally aligned SPPT L1 Table.

— A 4KB naturally aligned SPPT L1 table is located at the physical address specified in bits 51:12 of the SPPT L2E. An SPPT L1 table comprises 512 64-bit entries (SPPT L1Es). An SPPT L1E is selected at the physical address defined as follows.

- Bits 63:52 are all 0.
- Bits 51:12 are from the SPPT L2E.
- Bits 11:3 are bits 20:12 of the guest-physical address.
- Bits 2:0 are all 0.

The processor then consults bit 2i of the SPPT L1E, where i is the value of bits 11:7 of the guest-physical address; a write access to the guest-physical address is allowed if the bit is 1. (The odd bits in the SPPT L1E are reserved and must be 0.)

4.5.3 SPP-Induced VM Exits

Accesses using guest-physical addresses may cause SPP-induced VM exits due to an SPPT misconfiguration or an SPPT miss. The basic VM exit reason reported for SPP-induced VM exits is 66.

An SPPT misconfiguration VM exit occurs when, in the course of an SPPT lookup, an SPPT paging-structure entry is encountered that sets a reserved bit. See Section 4.5.3.1 for which bits are reserved in SPPT paging-structure entries.

An SPPT miss VM exit occurs when, in the course of an SPPT lookup, an SPPT paging-structure entry is encountered in which the valid bit is clear.

SPPT misconfigurations and SPPT misses can occur only due to an attempt to write memory with a guest-physical address.

SPP-induced VM exits save an exit qualification with the format given in Table 4-3. These VM exits also save a guest-linear address and a guest-physical address.

Table 4-3. Exit Qualification for SPPT-Induced VM Exits

Bit Position	Contents
10:0	Not used.
11	SPPT VM exit type. Set for SPPT miss; cleared for SPPT misconfiguration VM exit.
12	NMI unblocking due to IRET.
63:13	Not used.

Guest Linear Address: In addition to the existing cases for which this field is reported, for a VM exit due to an SPPT misconfiguration or SPPT miss, this field receives a linear address that caused the SPPT misconfiguration or SPPT miss VM exit.

Guest Physical Address: In addition to the existing cases for which this field is reported, for a VM exit due to an SPPT misconfiguration or SPPT miss, this field receives the guest-physical address that caused the SPPT misconfiguration or SPPT miss VM exit.

4.5.3.1 Sub-Page Permissions and EPT Violations

Memory writes that consult but are not permitted by the SPPT cause EPT violations normally.

For memory writes that access memory across sub-page regions on the same 4K page, the processor will check writeability of both sub-pages and will generate an EPT violation if either of the accessed sub-page regions is not writeable.

For memory writes that access memory across sub-page regions that fall on adjoining guest-physical 4K pages, the processor will generate an EPT violation if either one of the 4K guest-physical pages has the sub-page permission control bit set in the leaf EPT paging-structure for the affected pages.

Sub-page write permissions are intended principally for simple instructions (such as AND, MOV, OR, TEST, XCHG, INC, XOR, etc.). Execution of an instruction that normally performs multiple memory-writes may or may not ignore the sub-page permissions and cause EPT violations unconditionally if an accessed page is mapped with an EPT PTE in which the W bit is 0.

Accesses to any guest-physical address that translates to an address on the APIC-access page that also is specified by the VMM to have sub-page permissions associated with it may operate as if the virtualize APIC accesses VM-execution control is 0.

Updates to accessed and dirty flags in guest paging structures ignore the setting of the “sub-page write permission” VM-execution control. Such an update always causes an EPT violation if it would write to a guest-physical address to which EPT does not allow writes.

4.5.4 Invalidating Cached SPP Permissions

Sub-page permission permissions may be cached by the CPU. Any modification to the sub-page permission permissions specified in SPPT entries must be invalidated using INVEPT. The EPTP switching VM function may flush any information cached about sub-page permissions, as well as intermediate EPT and SPPT caches.

4.5.5 Sub-Page Permission Interaction with Accessed and Dirty Flags for EPT

Software can enable accessed and dirty flags for EPT using bit 6 of the Extended-Page-Table Pointer (EPTP). If this bit is 1, the processor will set the accessed and dirty flags for EPT. Whenever there is a write to a guest-physical address, the processor sets the dirty flag (if it is not already set) in the EPT paging-structure entry that identifies the final physical address for the guest-physical address (either an EPT PTE or an EPT paging-structure entry in which bit 7 is 1). In addition, when accessed and dirty flags for EPT are enabled, processor accesses to guest paging-structure entries are treated as writes. Thus, such an access will cause the processor to set the dirty flag in the leaf EPT paging-structure entry that identifies the final physical address of the guest paging-structure entry. (This is legacy behavior.)

A write access does not cause an EPT violation or SPP miss/misconfiguration VM exit until after the dirty flags are set in the appropriate paging structure and EPT paging structure (if enabled) that maps the affected guest physical address. If page-modification logging is enabled in addition, the processor will also update the page-modification log.

Additionally, when accessed and dirty flags for EPT are enabled, processor accesses to guest paging-structure entries are treated as writes, and the processor will ignore the sub-page permission control bit in the leaf EPT paging-structure, hence, such processor accesses to guest-physical pages for which the EPT paging-structure entry has the sub-page permission control bit set will cause EPT violations if the W permission on the leaf EPT paging-structure is cleared.

4.5.6 Sub-Page Permission Interaction with Intel® TSX

Instructions that begin or execute within a transactional region may attempt to write to guest-physical addresses to which EPT does not allow writes. Such cases result in transactional aborts.

This behavior is retained even with sub-page permissions. A write by an instruction that begins or executes within a transactional region ignores sub-page permissions and causes a transactional abort if EPT does not allow writes to the guest-physical address.

4.5.7 Sub-Page Permission Interaction with Intel® SGX

A VMM cannot access memory in the enclave page cache (EPC) and cannot easily determine how to protect those pages selectively with SPP.

- SPP checks have higher priority than EPC/EPCM checks. Write accesses made by enclave code to guest physical memory pages with SPP attribute that are mapped via addresses inside the ELRANGE will be treated as read-only and cause EPT violations. Write accesses made by SGX instructions to EPC are enforced to be inside the ELRANGE per the SGX architecture and are hence treated similarly.
- Write accesses made by enclave code to SPP guest physical memory pages mapped via addresses that fall out of the ELRANGE are treated as per the SPP architecture, with SPP permissions applying.

4.5.7.1 Fault Priorities

SPP and SGX access control interactions are as listed below.

- SPP checks have higher priority than EPC/EPCM checks.
- All accesses to SPP pages by SGX instructions shall cause an EPT violation.
- All accesses to SPP pages by an enclave into ELRANGE shall cause an EPT violation.
- All accesses to SPP pages by an enclave outside of ELRANGE shall be treated as per SPP architecture.

The fault behavior summarized in Table 4-4 below.

Table 4-4. Fault Behavior Summary

ID	Enclave Access	APIC Access	In EPC	EPTE.W	EPTE.SPP	Comments
1	0	0	NA	X	X	See notes[1].
2	0	1	NA	X	0	See notes[2].
3	0	1	NA	1	1	See notes[3].
4	0	1	NA	0	1	See notes[4].
5	1	X	X	X	0	See notes[5].
6	1	X	X	X	1	See notes[6].

NOTES:

1. Fault behavior as per SPP architecture described in this chapter.
2. Fault behavior as per the APIC virtualization architecture.
3. (SPP is ignored since EPT is writeable)
   If violation of EPT permissions then EPT violation
   Else If Implementation_supports_vAPIC_AND_SPP_Together
   Then APIC redirection or exit
   Else Access Allowed
4. If violation of EPT permissions then EPT violation exit
   Else If Implementation_supports_vAPIC_AND_SPP_Together
   If write access then EPT violation
   Else APIC redirection or exit
   Else If write access - fault behavior per SPP architecture in this specification
   Else If read/execute access - access allowed
5. If violation of EPT permissions then EPT violation
   Else If PA not in EPC then #PF
   Else If PA matches APIC access page then #PF
   Else If violation of EPCM permissions then #PF
   Else Access Allowed
6. If violation of EPT permissions – EPT violation
   Else EPT violation (SPP on enclave access)

4.5.8 Memory Type Used for Accessing SPPT

The memory type used for any such reference will be the memory type reported in IA32_VMX_BASIC MSR. Bits 53:50 of the IA32_VMX_BASIC MSR report the memory type that the processor uses to access the VMCS and data structures referenced by pointers in the VMCS. Software should ensure that the VMCS and referenced data structures are located at physical addresses that are mapped to WB memory type by the MTRRs.

4.6 CHANGES TO VM ENTRIES

If the activate secondary controls and sub-page write permission VM-execution controls are both 1, VM entries ensure that the enable EPT VM-execution control is 1. Additionally, the sub-page permission table control field is checked for consistency per Section 4.5. VM entry fails if these checks fail. When such a failure occurs, control is passed to the next instruction, RFLAGS.ZF is set to 1 to indicate the failure, and the VM-instruction error field is loaded with value 7, indicating “VM entry with invalid control field(s)”. This check may be performed in any order with respect to other checks on VMX controls and the host-state area. Different processors may thus give different error numbers for the same VMCS.

4.7 CHANGES TO VMX CAPABILITY REPORTING

Section 4.2 specified that secondary processor-based VM-execution control 23 is defined as “sub-page write permission”. A processor that supports the 1-setting of the control sets bit 55 of the IA32_VMX_PROCBASED_CTLS2 MSR (index 48BH). RDMSR of that MSR returns 1 in bit 23 of EDX.

4.8 INSTRUCTIONS TO WHICH SPP DOES NOT APPLY

Sub-page permission treatment of accesses to pages is principally intended for basic instructions such as ADD, AND, DEC, MOV, OR, SUB, TEST, XCHG, and XOR. Use of complex instructions such as those that operate on floating-point, SSE, AVX, or AVX-512 registers, among others, may cause an EPT violation VM exit ignoring any sub-page permissions specified for the page(s) accessed.

CHAPTER 5
INTEL® PROCESSOR TRACE: VMX IMPROVEMENTS

5.1 INTRODUCTION

Intel® Processor Trace (Intel® PT) is an extension of Intel® Architecture that captures information about software execution using dedicated hardware facilities that cause only minimal performance perturbation to the software being traced. Details on the Intel PT infrastructure and trace capabilities can be found in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*.

This chapter describes the architecture for VMX support improvements made for Intel PT. The suite of architecture changes described below serve to simplify the process of virtualizing Intel PT for use by a guest software. There are two primary elements to this new architecture support.

1. **Addition of a new guest IA32_RTIT_CTL value field to the VMCS. —** This serves to speed and simplify the process of disabling trace on VM exit, and restoring it on VM entry.
2. **Enabling use of EPT to redirect PT output. —** This enables the VMM to elect to virtualize the PT output buffer using EPT. In this mode, the CPU will treat PT output addresses as Guest Physical Addresses (GPAs) and translate them using EPT. This means that Intel PT output reads (of the ToPA table) and writes (of trace output) can cause EPT violations, and other output events.

5.2 ARCHITECTURE DETAILS

5.2.1 IA32_RTIT_CTL in VMCS Guest State

A new 64-bit field will be added to the VMCS Guest State, to hold the value of IA32_RTIT_CTL. This field will use encodings 2814H and 2815H. On VM exit, the MSR value will be written to this field unconditionally. Additionally, there are two new controls to govern use of this field; see Table 5-1 below.

Table 5-1. VMCS Controls for IA32_RTIT_CTL MSR

Name	Position	Description
Clear IA32_RTIT_CTL on exit.	Exit control 25.	When set, the IA32_RTIT_CTL MSR will be cleared on VM exit, after it has been saved. This disables PT before entering the VMX host.
Load IA32_RTIT_CTL on entry.	Entry control 18.	When set, the IA32_RTIT_CTL MSR will be written with the value of the associated Guest State field of the VMCS on VM entry. This restores PT before entering the guest. VM entry fails if the value to be loaded sets reserved bits or a reserved values in an encoded field.

On a VM exit that clears IA32_RTIT_CTL, or a VM entry that updates it, the XSAVE INIT and MOD tracking optimizations will be updated.

5.2.2 Supporting EPT for Trace Output

In order to enable use of EPT to redirect PT trace output, a new secondary processor-based VM-execution control is added; see Table 5-2 below.

Table 5-2. VMCS Control for Intel PT Output to Guest Physical Addresses

Name	Position	Description
Guest PT uses Guest Physical Addresses.	Execution control 24.	When set, all PT output addresses, including those in the IA32_RTIT_OUTPUT_BASE MSR and in ToPA tables, will be treated as guest physical addresses (GPAs) and translated with EPT.

Setting this new VM-execution control to 1 requires also setting the VM-exit and VM-entry controls described above in Table 5-1. This ensures that PT is disabled before entering root operation, where EPT does not apply. See details on new consistency checks in Section 5.2.3.

5.2.2.1 VM Exits Due to Intel PT Output

Using EPT to translate PT output addresses introduces the possibility of taking events on PT output reads and writes. Event possibilities include EPT violations, EPT misconfigurations, PML log-full VM exits, and APIC access VM exits.

EFLAGS.RF is not modified by VM exits due to Intel PT output.

Exit Qualification

Intel PT output reads and writes are asynchronous to instruction execution, as a result of the internal buffering of trace data. Trace packets are output some unpredictable number of cycles after the completion of the instructions or events that generated them. For this reason, any VM exit caused by Intel PT output will set the following new exit qualification bit.

Table 5-3. New Asynchronous Exit Qualification Bit

Name	Position	Description
Asynchronous to Instruction Execution.	Exit qualification bit 16 for EPT violations, PML log-full VM exits, and APIC-access VM exits due to guest-physical accesses.	This VM exit results neither from the instruction referenced by the RIP saved into the VMCS, nor from any event delivery recorded in the VMCS IDT-vectoring fields.

Because Intel PT output is using GPAs, there is no relevant Guest Linear Address (GLA). As a result, the exit qualification bit 7 (guest linear address field valid) and the GLA field are cleared. Bits 11:8 of the exit qualification are also cleared.

Preserving Pending Events

A VM entry that enables Intel PT can cause an immediate VM exit, if PT output is configured to use GPA addressing and the access to the page causes a VM exit (e.g., EPT violation). This VM exit will be taken after the completion of the VM entry, but before other events which may be pending or injected by the VM entry. To ensure that no events are lost, VM exits caused by PT output will take the following measures.

- The guest pending debug exceptions field in the VMCS is not cleared, and the value saved will match the behavior of existing VM exits (e.g., INIT) that do not clear the field.
- The VMCS VM-entry interrupt information field is saved to the VMCS IDT-vectoring information field. This serves to simplify the process of re-injecting the event on the next VM entry. Note that this introduces a scenario where Pending MTF VM exit can be set in the IDT-vectoring information field.

Additional VM Exits

EPT violations caused by Intel PT output will always cause VM exits; virtualization exceptions (#VEs) are not supported.

APIC page accesses by Intel PT cause VM exits unconditionally, with no virtualization by the processor. This is consistent with other guest-physical accesses to the APIC page.

If the “Guest PT uses Guest Physical Addresses” VM-execution control is 1 and IA32_RTIT_CTL.TraceEn = 1, any invocation of the VM function 0 (EPTP switching) causes a VM exit. The VM exit gives a VMM the opportunity to disable tracing (if desired for certain EPT contexts) and ensures that the processor does not retain a PT-specific EPT-based translation a change of EPTP. Reporting is the same as any VM exit caused by a VM function, setting the basic exit reason to 59 (indicating “VMFUNC”) and saving the length of the VMFUNC instruction into the VM-exit instruction-length field.

5.2.2.2 Trace Data Management with Output Events

Because PT packet data is buffered within the CPU before being written out through the memory subsystem or other trace transport mechanism, the CPU takes measures to ensure that buffered trace data is not lost on the PT disable during VM exit. This requires ensuring that there is sufficient space left in the current output page to write out the buffer. Without such care, buffered trace data could be lost, and the resulting trace corrupted.

The CPU will employ an early page lookup mechanism in order to avoid trace corruption. It will try to cache the physical addresses (PAs) of the current PT output block and the next PT output block, in order to ensure no event is needed when transitioning from the current block to the next. An output block is defined as the smaller of the EPT page and the PT output buffer segment, which is either a ToPA output region or the single-range output buffer. Using this scheme, the CPU will always lookup the next block when it begins writing the current block, so that any events needed in order to translate the next block base address can be taken long before writes to that next block commence.

When PT is enabled, the CPU will lookup the first 2 output block translations, and cache the resulting PAs internally. PT enable flows include WRMSR (as well as loads from the MSR-load areas by VMX transitions), XRSTORS, VM entry, and RSM.

If either EPT lookup requires a VM exit, the exit will be taken before tracing begins. However, the value of IA32_RTIT_CTL saved into the new VMCS field will have the new value, with TraceEn set. This ensures that the subsequent VM entry will try again to enable PT.

These VM exits resulting from the use of Intel PT are taken after the completion of the current instruction or operation. On VM entry, any Intel PT-induced VM exit will be taken after transition to the guest completes, but before any event injection or guest instructions execute.

Once the PAs for the first two output blocks are cached (this could require multiple events, and hence multiple VM exits/VM entries), tracing will commence. Henceforth, anytime an output block is filled with trace data, output will transition to the next (cached) output block, and the CPU will lookup the EPT translation for the output block that follows the new current block. Here again, an event may need to be taken, which would result in a VM exit. If the lookup encounters a ToPA entry with the STOP bit set, it will cease to lookup further entries beyond that entry.

This early page lookup mechanism serves to reduce the likelihood that the trace could fill all available, translated output blocks. The CPU should typically have the current and next block cached and ready for output. In cases where trace data nonetheless has to be dropped, which could happen if an EPT violation VM exit for the next page translation is not taken for an extended period of time, the CPU will signal an internal buffer overflow and drop packets until the new translation can be cached.

5.2.2.3 Intel PT Output Errors

Improper configuration of Intel PT output can result in operation errors that cause tracing to be disabled. See the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, Section 35.3.9, “Operational Errors” for details.

When Intel PT output is redirected using EPT, all address-based checks continue to be executed using the guest physical address specified in the ToPA table or MSR, with one exception. Checks against restricted memory (see the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, Section 35.2.6.4, “Restricted Memory Access” for details) are done using the translated, platform physical address to which output will be written.

5.2.3 New VM-Entry Consistency Checks

The following consistency checks will cause the VM entry to fall through to the next sequential instruction, and RFLAGS.ZF to be set, if failed.

- If the “Guest PT uses Guest Physical Addresses” execution control is 1, the “Clear IA32_RTIT_CTL on exit” exit control and the “Load IA32_RTIT_CTL on entry” entry control must also be 1. This ensures that the processor will not switch from treating Intel PT output addresses as GPAs to treating them as PPAs.
- If the “Guest PT uses Guest Physical Addresses” execution control is 1, the "enable EPT" execution control must also be 1.

If the following consistency check fails, VM entry fails by loading processor state from the guest-state area of the VMCS.

- If the “Load IA32_RTIT_CTL on entry” is 1, IA32_RTIT_CTL.TraceEn must be zero.

The lower 16 bits of the exit reason VMCS field will hold value 33, indicating failure due to invalid guest state.

5.2.3.1 Special Treatment for SMM VM Exits

The consistency checks above do not ensure that an SMM VM exit that occurs with the 1-setting of the “Guest PT uses Guest Physical Addresses” VM-execution control will find the “Clear IA32_RTIT_CTL on exit” VM-exit control set to 1. For this reason, such VM exits always clear the IA32_RTIT_CTL MSR, regardless of the setting of the VM-exit control.

5.3 ENUMERATION

Section 5.2 identified three new controls in the VMCS. The following paragraphs provide details of how processors enumerate for support of those controls:

- “Guest PT uses Guest Physical Addresses” is a new secondary processor-based VM-execution control, located at bit position 24. Processors supporting the 1-setting of this control enumerate that support by setting bit 56 of the IA32_VMX_PROCBASED_CTLS2 MSR (index 48BH).
- “Clear IA32_RTIT_CTL on exit” is a new VM-exit control, located at bit position 25. Processors supporting the 1-settings of this control enumerate that support by setting bit 57 in both the IA32_VMX_EXIT_CTLS MSR (index 483H) and the IA32_VMX_TRUE_EXIT_CTLS MSR (index 48FH).
- “Load IA32_RTIT_CTL on entry” is a new VM-entry control, located at bit position 18. Processors supporting the 1-settings of this control enumerate that support by setting bit 40 in both the IA32_VMX_ENTRY_CTLS MSR (index 484H) and the IA32_VMX_TRUE_ENTRY_CTLS MSR (index 490H).

INDEX

B

Brand information 1-30
    processor brand index 1-32
    processor brand string 1-30

C

Cache and TLB information 1-25
Cache Inclusiveness 1-6
CLFLUSH instruction
    CPUID flag 1-24
CMOVcc flag 1-24
CMOVcc instructions
    CPUID flag 1-24
CMPXCHG16B instruction
    CPUID bit 1-22
CMPXCHG8B instruction
    CPUID flag 1-24
CPUID instruction 1-4, 1-24
    36-bit page size extension 1-24
    APIC on-chip 1-24
    basic CPUID information 1-5
    cache and TLB characteristics 1-5, 1-25
    CLFLUSH flag 1-24
    CLFLUSH instruction cache line size 1-20
    CMPXCHG16B flag 1-22
    CMPXCHG8B flag 1-24
    CPL qualified debug store 1-21
    debug extensions, CR4.DE 1-23
    debug store supported 1-24
    deterministic cache parameters leaf 1-5, 1-7, 1-9, 1-10, 1-11, 1-12, 1-13
    extended function information 1-16
    feature information 1-23
    FPU on-chip 1-23
    FSAVE flag 1-24
    FXRSTOR flag 1-24
    IA-32e mode available 1-16
    input limits for EAX 1-18
    L1 Context ID 1-22
    local APIC physical ID 1-20
    machine check architecture 1-24
    machine check exception 1-24
    memory type range registers 1-24
    MONITOR feature information 1-28
    MONITOR/MWAIT flag 1-21
    MONITOR/MWAIT leaf 1-6, 1-7, 1-8, 1-9, 1-10, 1-14
    MWAIT feature information 1-28
    page attribute table 1-24
    page size extension 1-23
    performance monitoring features 1-28
    physical address bits 1-17
    physical address extension 1-24
    power management 1-28, 1-29, 1-30
    processor brand index 1-20, 1-30
    processor brand string 1-17, 1-30
    processor serial number 1-24
    processor type field 1-19
    RDMSR flag 1-23
    returned in EBX 1-20
    returned in ECX & EDX 1-20
    self snoop 1-25
    SpeedStep technology 1-21
    SS2 extensions flag 1-25

SSE extensions flag 1-25
SSE3 extensions flag 1-21
SSSE3 extensions flag 1-21
SYSENTER flag 1-24
SYSEXIT flag 1-24
thermal management 1-28, 1-29, 1-30
thermal monitor 1-21, 1-24, 1-25
time stamp counter 1-23
using CPUID 1-4
vendor ID string 1-18
version information 1-5, 1-27
virtual 8086 Mode flag 1-23
virtual address bits 1-17
WRMSR flag 1-23

F

Feature information, processor 1-4
FXRSTOR instruction
CPUID flag 1-24
FXSAVE instruction
CPUID flag 1-24

I

IA-32e mode
CPUID flag 1-16
Instruction set
grouped by processor 1-1

L

L1 Context ID 1-22

M

Machine check architecture
CPUID flag 1-24
description 1-24
MMX instructions
CPUID flag for technology 1-24
Model & family information 1-27
MONITOR instruction
CPUID flag 1-21
feature data 1-28
MWAIT instruction
CPUID flag 1-21
feature data 1-28

P

Pending break enable 1-25
Performance-monitoring counters
CPUID inquiry for 1-28

R

RDMSR instruction
CPUID flag 1-23

S

Self Snoop 1-25
SpeedStep technology 1-21
SSE extensions
CPUID flag 1-25
SSE2 extensions
CPUID flag 1-25
SSE3

CPUID flag 1-21
SSE3 extensions
CPUID flag 1-21
SSSE3 extensions
CPUID flag 1-21
Stepping information 1-27
SYSENTER instruction
CPUID flag 1-24
SYSEXIT instruction
CPUID flag 1-24

T

Thermal Monitor
CPUID flag 1-25
Thermal Monitor 2 1-21
CPUID flag 1-21
Time Stamp Counter 1-23

V

Version information, processor 1-4
VPERMI2B - Full Permute of Bytes from Two Tables Overwriting the Index 3-4
VPMULTISHIFTQB – Select Packed Unaligned Bytes from Quadword Source 2-26

W

WBINVD/INVD bit 1-6
WRMSR instruction
CPUID flag 1-23

X

XRSTOR 1-29
XSAVE 1-22, 1-29