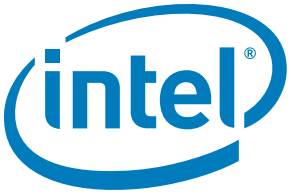

Intel® Architecture Instruction Set Extensions and Future Features Programming Reference

319433-037
MAY 2019

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 Deep Learning Boost, Intel DL Boost, 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, 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	Description	Date
-031	• Updated change log to correct typo in changes from previous release. • Updated instructions with imm8 operand missing in operand encoding table. • Replaced "VLMAX" with "MAXVL" to align terminology used across documentation. • Added back information on detection of Intel AVX-512 instructions. • Added Intel® Memory Encryption Technologies instructions PCONFIG and WBNOINVD. These instructions are also added to Table 1-1 "Recent Instruction Set Extensions Introduction in Intel 64 and IA-32 Processors". Added Section 1.5 "Detection of Intel® Memory Encryption Technologies (Intel® MKTME) Instructions". • CPUID instruction updated with PCONFIG and WBNOINVD details. • CPUID instruction updated with additional details on leaf 07H: Intel® Xeon Phi™ only features identified and listed. • CPUID instruction updated with new Intel® SGX features in leaf 12H. • CPUID instruction updated with new PCONFIG information sub-leaf 1BH. • Updated short descriptions in the following instructions: VPDPBUSD, VPDPBUSDS, VPDPWSSD and VPDPWSSDS. • Corrections and clarifications in Chapter 4 "EPT-Based Sub-Page Permissions". • Corrections and clarifications in Chapter 5 "Intel® Processor Trace: VMX Improvements".	January 2018
-032	• Corrected PCONFIG CPUID feature flag on instruction page. • Minor updates to PCONFIG instruction pages: Changed Table 2-2 to use Hex notation; changed "RSVD, MBZ" to "Reserved, must be zero" in two places in Table 2-3. • Minor typo correction in WBNOINVD instruction description.	January 2018
-033	• Updated Table 1-1 "Recent Instruction Set Extensions / Features Introduction in Intel® 64 and IA-32 Processors" . • Added Section 1.6, "Detection of Future Instructions". • Added CLDEMOTE, MOVDIRI, MOVDIR64B, TPAUSE, UMONITOR and UMWAIT instructions. • Updated the CPUID instruction with details on new instructions/features added, as well as new power management details and information on hardware feedback interface ISA extensions. • Corrections to PCONFIG instruction. • Moved instructions unique to processors based on the Knights Mill microarchitecture to the Intel® 64 and IA-32 Architectures Software Developer's Manual. • Added Chapter 5 "Hardware Feedback Interface ISA Extensions". • Added Chapter 6 "AC Split Lock Detection".	March 2018
-034	• Added clarification to leaf 07H in the CPUID instruction. • Added MSR index for IA32_UMWAIT_CONTROL MSR. • Updated registers in TPAUSE and UMWAIT instructions. • Updated TPAUSE and UMWAIT intrinsics.	May 2018

Revision	Description	Date
-035	• Updated Table 1-1 "Recent Instruction Set Extensions / Features Introduction in Intel® 64 and IA-32 Processors" to list the AVX512_VNNI instruction set architecture on a separate line due to presence on future processors available sooner than previously listed. • Updated CPUID instruction in various places. • Removal of NDD/DDS/NDS terms from instructions. Note: Previously, the terms NDS, NDD and DDS were used in instructions with an EVEX (or VEX) prefix. These terms indicated that the vvvv field was valid for encoding, and specified register usage. These terms are no longer necessary and are redundant with the instruction operand encoding tables provided with each instruction. The instruction operand encoding tables give explicit details on all operands, indicating where every operand is stored and if they are read or written. If vvvv is not listed as an operand in the instruction operand encoding table, then EVEX (or VEX) vvvv must be 0b1111. • Added additional #GP exception condition to TPAUSE and UMWAIT. • Updated Chapter 5 "Hardware Feedback Interface ISA Extensions" as follows: changed scheduler/software to operating system or OS, changed LP0 Scheduler Feedback to LP0 Capability Values, various description updates, clarified that capability updates are independent, and added an update to clarify that bits 0 and 1 will always be set together in Section 5.1.4. • Added IA32_CORE_CAPABILITY MSR to Chapter 6 "AC Split Lock Detection".	October 2018
-036	• Added AVX512_BF16 instructions in chapter 2; related CPUID information updated in chapter 1. • Added new section to chapter 1 describing bfloat16 format. • CPUID leaf updates to align with the Intel® 64 and IA-32 Architectures Software Developer's Manual. • Removed CLDEMOTE, TPAUSE, UMONITOR, and UMWAIT instructions; they now reside in the Intel® 64 and IA-32 Architectures Software Developer's Manual. • Changes now marked by green change bars and green font in order to view changes at a text level.	April 2019
-037	• Removed chapter 3, "EPT-Based Sub-Page Permissions", chapter 4, "Intel® Processor Trace: VMX Improvements", and chapter 6, "Split Lock Detection"; this information is in the Intel® 64 and IA-32 Architectures Software Developer's Manual. • Removed MOVDIRI and MOVDIR64B instructions; they now reside in the Intel® 64 and IA-32 Architectures Software Developer's Manual. • Updated Table 1-1 with new features in future processors. • Updated Table 1-2 with support for AVX512_VP2INTERSECT. • Updated Table 1-4 with support for ENQCMD: Enqueue Stores. • Added ENQCMD/ENQCMDS and VP2INTERSECTD/VP2INTERSECTQ instructions, and updated CPUID accordingly. • Added new chapter: Chapter 3, Enqueue Stores and Process Address Space Identifiers (PASIDs).	May 2019

REVISION HISTORY

CHAPTER 1
FUTURE INTEL® ARCHITECTURE INSTRUCTION EXTENSIONS AND FEATURES

CHAPTER 2
INSTRUCTION SET REFERENCE, A-Z

CHAPTER 3
ENQUEUE STORES AND PROCESS ADDRESS SPACE IDENTIFIERS (PASIDS)

CHAPTER 4
HARDWARE FEEDBACK INTERFACE ISA EXTENSIONS

TABLES

FIGURES

PAGE

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 ENQCMD/ENQCMDS details and virtualization support.

Chapter 4 describes Hardware Feedback Interface ISA Extensions.

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

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

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

Instruction Set Architecture	Processor Generation Introduction	Introduced 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
	Intel® Atom™ processor.	Silvermont
SSE4.2 Extensions, CRC32, POPCNT	Intel® Core™ i7 965 processor, Intel® Xeon® processors X3400, X3500, X5500, X6500, X7500 series.	Legacy
	Intel® Atom™ processor.	Silvermont
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
	Intel® Atom™ processor.	Silvermont

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

Instruction Set Architecture	Processor Generation Introduction	Introduced in Microarchitecture
Intel AVX	Intel® Xeon® processor E3 and E5 families. 2nd Generation Intel® Core™ i7, i5, i3 processor 2xxx families.	Sandy Bridge
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
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
	Intel® Atom™ processor.	Silvermont
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
	Intel® Atom™ processor.	Goldmont
FMA, AVX2, BMI1, BMI2, INVPCID, LZCNT, TSX	Intel® Xeon® processor E3/E5/E7 v3 product families. 4th Generation Intel® Core™ processor family.	Haswell
MOVBE	Intel® Xeon® processor E3/E5/E7 v3 product families. 4th Generation Intel® Core™ processor family.	Haswell
	Intel® Atom™ processor.	Silvermont
PREFETCHW	Intel® Core™ M processor family; 5th Generation Intel® Core™ processor family.	Broadwell
	Intel® Atom™ processor.	Silvermont
ADX	Intel® Core™ M processor family; 5th Generation Intel® Core™ processor family.	Broadwell
CLAC, STAC	Intel® Core™ M processor family; 5th Generation Intel® Core™ processor family.	Broadwell
	Intel® Atom™ processor.	Goldmont
RDSEED	Intel® Core™ M processor family; 5th Generation Intel® Core™ processor family.	Broadwell
	Intel® Atom™ processor.	Goldmont
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
	Intel® Core™ i3-8121U processor.	Cannon Lake
CLFLUSHOPT, XSAVEC, XSAVES, MPX	Intel® Xeon® Processor Scalable Family.	Skylake Server
	6th Generation Intel® Core™ processor family.	Skylake
	Intel® Atom™ processor.	Goldmont
SGX1	6th Generation Intel® Core™ processor family.	Skylake
	Intel® Atom™ processor.	Goldmont Plus

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

Instruction Set Architecture	Processor Generation Introduction	Introduced in Microarchitecture
AVX512DQ, AVX512BW, AVX512VL	Intel® Xeon® Processor Scalable Family.	Skylake Server
	Intel® Core™ i3-8121U processor.	Cannon Lake
CLWB	Intel® Xeon® Processor Scalable Family.	Skylake Server
	TBD	Future, Ice Lake
	TBD	Future, Tremont
PKU	Intel® Xeon® Processor Scalable Family.	Skylake Server
AVX512_IFMA, AVX512_VBMI	Intel® Core™ i3-8121U processor.	Cannon Lake
SHA-NI	Intel® Core™ i3-8121U processor.	Cannon Lake
	Intel® Atom™ processor.	Goldmont
UMIP	Intel® Core™ i3-8121U processor.	Cannon Lake
	Intel® Atom™ processor.	Goldmont Plus
PTWRITE	Intel® Atom™ processor.	Goldmont Plus
RDPID	TBD	Future, Ice Lake
	Intel® Atom™ processor.	Goldmont Plus
AVX512_4FMAPS, AVX512_4VNNIW	Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series.	Knights Mill
AVX512_VNNI	Future versions of Intel® Xeon® Processor Scalable Family.	Cascade Lake
AVX512_VPOPCNTDQ	Intel® Xeon Phi™ Processor 7215, 7285, 7295 Series.	Knights Mill
	TBD	Future, Ice Lake
Fast Short REP MOV	TBD	Future, Ice Lake
VAES, GFNI (AVX/AVX512), AVX512_VBMI2, VPCLMULQDQ, AVX512_BITALG	TBD	Future, Ice Lake
GFNI(SSE)	TBD	Future, Ice Lake
	TBD	Future, Tremont
PCONFIG, WBNOINVD	TBD	Future, Ice Lake Server
ENCLV	TBD	Future, Ice Lake Server
	TBD	Future, Tremont
Split Lock Detection	TBD	Future, Ice Lake
	TBD	Future, Tremont
CLDEMOTE	TBD	Future, Tremont
Direct stores: MOVDIRI, MOVDIR64B	TBD	Future, Tremont
	TBD	Future, Tiger Lake
User wait: TPAUSE, UMONITOR, UMWAIT	TBD	Future, Tremont
AVX512_BF16	TBD	Future, Cooper Lake
CET: Control-flow Enforcement Technology[1]	TBD	Future, Tiger Lake

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

Instruction Set Architecture	Processor Generation Introduction	Introduced in Microarchitecture
AVX512_VP2INTERSECT	TBD	Future, Tiger Lake
Enqueue Stores: ENQCMD and ENQCMDS	TBD	Future, Sapphire Rapids

NOTES:

1. Details on Control-flow Enforcement Technology can be found here: *CET specification*

1.3 DETECTION OF AVX-512 FOUNDATION INSTRUCTIONS

The majority of AVX-512 Foundation instructions are encoded using the EVEX encoding scheme. EVEX-encoded instructions can operate on the 512-bit ZMM register state plus 8 opmask registers. The opmask instructions in AVX-512 Foundation instructions operate only on opmask registers or with a general purpose register.

Processor support of AVX-512 Foundation instructions is indicated by CPUID.(EAX=07H, ECX=0):EBX.AVX512F[bit 16] = 1. Detection of AVX-512 Foundation instructions operating on ZMM states and opmask registers need to follow the general procedural flow in Figure 1-1.

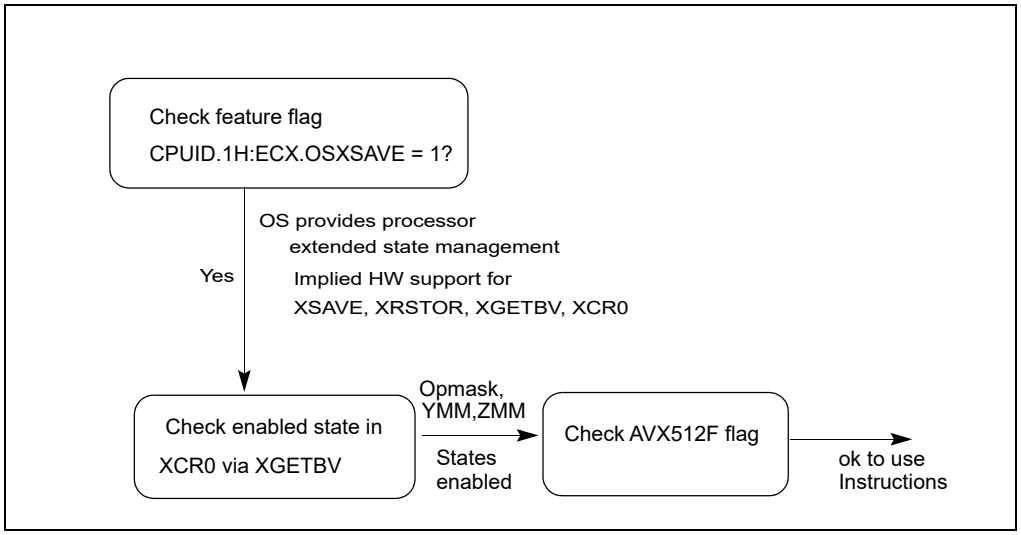

Figure 1-1. Procedural Flow of Application Detection of AVX-512 Foundation Instructions

Prior to using AVX-512 Foundation instructions, the application must identify that the operating system supports the XGETBV instruction, the ZMM register state, in addition to processor's support for ZMM state management using XSAVE/XRSTOR and AVX-512 Foundation instructions. The following simplified sequence accomplishes both and is strongly recommended.

1) Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 (XGETBV enabled for application use[1]).

2) Execute XGETBV and verify that XCR0[7:5] = '111b' (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that XCR0[2:1] = '11b' (XMM state and YMM state are enabled by OS).

3) Detect CPUID.0x7.0:EBX.AVX512F[bit 16] = 1.

1. If CPUID.01H:ECX.OSXSAVE reports 1, it also indirectly implies the processor supports XSAVE, XRSTOR, XGETBV, processor extended state bit vector XCR0 register. Thus an application may streamline the checking of CPUID feature flags for XSAVE and OSXSAVE. XSETBV is a privileged instruction.

1.4 DETECTION OF 512-BIT INSTRUCTION GROUPS OF INTEL® AVX-512 FAMILY

In addition to the Intel AVX-512 Foundation instructions, Intel AVX-512 family provides several additional 512-bit extensions in groups of instructions, each group is enumerated by a CPUID leaf 7 feature flag and can be encoded via EVEX.L'L field to support operation at vector lengths smaller than 512 bits. These instruction groups are listed in Table 1-2.

Table 1-2. 512-bit Instruction Groups in the Intel AVX-512 Family

CPUID Leaf 7 Feature Flag Bit	Feature Flag abbreviation of 512-bit Instruction Group	SW Detection Flow
CPUID.(EAX=07H, ECX=0):EBX[bit 16]	AVX512F: AVX-512 Foundation instructions.	Figure 1-1
CPUID.(EAX=07H, ECX=0):ECX[bit 06]	AVX512_VBMI2: Additional byte, word, dword and qword capabilities, an addition to AVX512.	Figure 1-2
CPUID.(EAX=07H, ECX=0):ECX[bit 08]	GFNI: Galois Field New Instructions; this bit is concatenated by software with either AVX512, AVX, or SSE to indicate the different supported instructions.	Figure 1-2
CPUID.(EAX=07H, ECX=0):ECX[bit 09]	VAES: Vector AES instructions; this bit is concatenated by software with AVX512 or AVX to indicate the different supported instructions.	Figure 1-2
CPUID.(EAX=07H, ECX=0):ECX[bit 10]	VPCLMULQDQ: Vector PCLMULQDQ instructions; this bit is concatenated by software with AVX512 or AVX to indicate the different supported instructions.	Figure 1-2
CPUID.(EAX=07H, ECX=0):ECX[bit 11]	AVX512_VNNI: Vector Neural Network Instructions, an addition to AVX512.	Figure 1-2
CPUID.(EAX=07H, ECX=0):ECX[bit 12]	AVX512_BITALG: Support for VPOPCNT[B,W] and VPSHUFBITQMB.	Figure 1-2
CPUID.(EAX=07H, ECX=0):ECX[bit 14]	AVX512_VPOPCNTDQ: Support for VPOPCNT[D,Q].	Figure 1-2
CPUID.(EAX=07H, ECX=1):EAX[bit 05]	AVX512_BF16: Support for BFLOAT16 instructions.	Figure 1-2
CPUID.(EAX=07H, ECX=0):EDX[bit 08]	AVX512_VP2INTERSECT: Support for VP2INTERSECT[D,Q]	Figure 1-2

Software must follow the detection procedure for the 512-bit AVX-512 Foundation instructions as described in Section 1.3.

Detection of other 512-bit sibling instruction groups listed in Table 1-2 (excluding AVX512F) follows the procedure described in Figure 1-2.

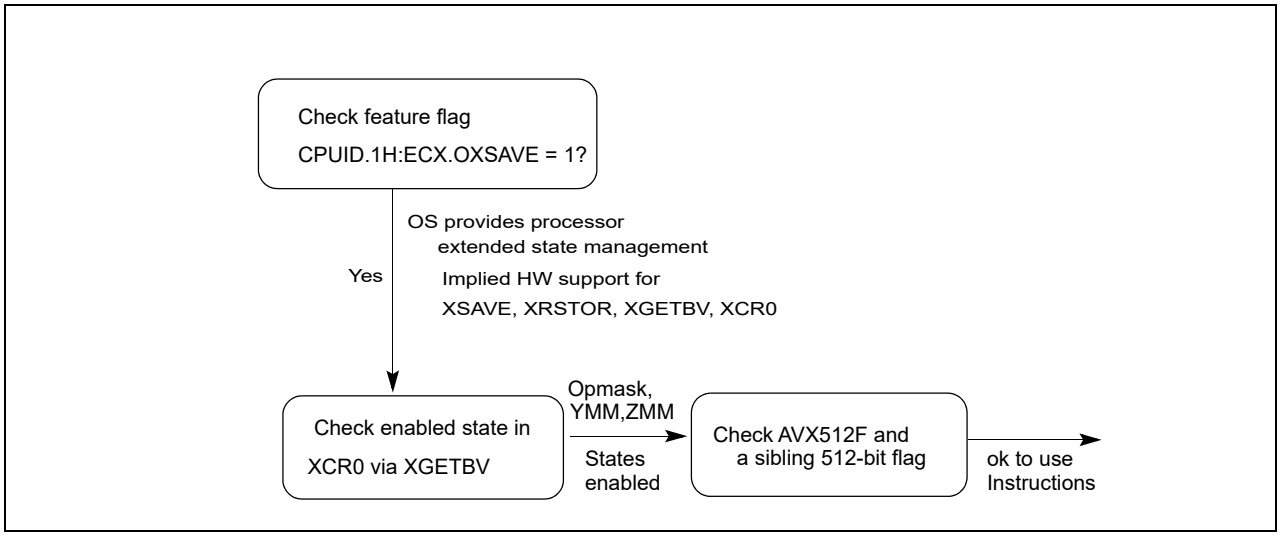

Figure 1-2. Procedural Flow of Application Detection of 512-bit Instruction Groups

To illustrated the detection procedure for 512-bit instructions enumerated by AVX512CD, the following sequence is strongly recommended.

1) Detect CPUID.1:ECX.OSXSAVE[bit 27] = 1 (XGETBV enabled for application use).

2) Execute XGETBV and verify that XCR0[7:5] = '111b' (OPMASK state, upper 256-bit of ZMM0-ZMM15 and ZMM16-ZMM31 state are enabled by OS) and that XCR0[2:1] = '11b' (XMM state and YMM state are enabled by OS).

3) Verify both CPUID.0x7.0:EBX.AVX512F[bit 16] = 1, CPUID.0x7.0:EBX.AVX512CD[bit 28] = 1.

Similarly, the detection procedure for enumerating 512-bit instructions reported by AVX512DW follows the same flow.

1.5 DETECTION OF INTEL® MEMORY ENCRYPTION TECHNOLOGIES (INTEL® MKTME) INSTRUCTIONS

Intel® Memory Encryption Technologies instructions are enumerated by a CPUID feature flag; details are listed in Table 1-3.

Table 1-3. Intel® Memory Encryption Technologies Instructions

CPUID Leaf Feature Flag Bit	Feature Flag Abbreviation of Intel® MKTME Instructions
CPUID.(EAX=07H, ECX=0):EDX[bit 18]	PCONFIG: Platform configuration
CPUID.(EAX=80000008H, ECX=0):EBX[bit 9]	WBNOINVD: Write back and do not invalidate cache

1.6 DETECTION OF FUTURE INSTRUCTIONS

Future instructions are enumerated by a CPUID feature flag; details are listed in Table 1-4.

Table 1-4. Future Instructions

CPUID Leaf Feature Flag Bit	Feature Flag Abbreviation
CPUID.(EAX=07H, ECX=0):ECX[bit 25]	CLDEMOTE: Cache Line Demote
CPUID.(EAX=07H, ECX=0):EDX[bit 4]	Fast Short REP MOV
CPUID.(EAX=07H, ECX=0):ECX[bit 5]	WAITPKG: Wait and Pause Enhancements
CPUID.(EAX=07H, ECX=0):ECX[bit 27]	MOVDIRI: Direct Stores
CPUID.(EAX=07H, ECX=0):ECX[bit 28]	MOVDIR64B: Direct Stores
CPUID.(EAX=07H, ECX=0):ECX[bit 29]	ENQCMD: Enqueue Stores

1.7 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 1-5 shows information returned, depending on the initial value loaded into the EAX register. Table 1-6 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. *)[2]
CPUID.EAX =1FH (* Returns V2 Extended Topology Enumeration leaf. *)[2]
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.
2. CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of CPUID leaf 1FH before using leaf 0BH.

Table 1-5. 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 1-6) "Genu" "ntel" "inel"
01H	EAX	Version Information: Type, Family, Model, and Stepping ID (see Figure 1-3)
	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 1-4 and Table 1-8)
	EDX	Feature Information (see Figure 1-5 and Table 1-9)
		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. ***The 8-bit initial APIC ID in EBX[31:24] is replaced by the 32-bit x2APIC ID, available in Leaf 0BH and Leaf 1FH.*
02H	EAX EBX ECX EDX	Cache and TLB Information (see Table 1-10) 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 1-36.
	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 1-5. 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	Bit 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported Bit 01: Supports treating interrupts as break-event for MWAIT, even when interrupts disabled Bits 31-02: Reserved

Table 1-5. 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. Bit 15: HWP Capabilities. Highest Performance change is supported if set. Bit 16: HWP PECI override is supported if set. Bit 17: Flexible HWP is supported if set. Bit 18: Fast access mode for the IA32_HWP_REQUEST MSR is supported if set. Bit 19: HW_FEEDBACK. IA32_HW_FEEDBACK_PTR, IA32_HW_FEEDBACK_CONFIG, IA32_PACKAGE_THERM_STATUS bit 26 and IA32_PACKAGE_THERM_INTERRUPT bit 25 are supported if set. Bit 20: Ignoring Idle Logical Processor HWP request is supported if set. Bits 31 - 21: 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

Table 1-5. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 7-0: Bitmap of supported hardware feedback interface capabilities. 0 = When set to 1, indicates support for performance capability reporting. 1 = When set to 1, indicates support for energy efficiency capability reporting. 2-7 = Reserved Bits 11-8: Enumerates the size of the hardware feedback interface structure in number of 4 KB pages using minus-one notation. Bits 31-16: Index (starting at 0) of this logical processor's row in the hardware feedback interface structure. Note that the index may be same for multiple logical processors on some parts. On some parts the indices may not be contiguous, i.e., there may be unused rows in the table. **NOTE:** Bits 0 and 1 will always be set together.
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.
	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 (Intel® Xeon Phi™ only.) Bit 27: AVX512ER (Intel® Xeon Phi™ only.) Bit 28: AVX512CD Bit 29: SHA Bit 30: AVX512BW Bit 31: AVX512VL

Table 1-5. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	ECX	Bit 00: PREFETCHWT1 (Intel® Xeon Phi™ only.) 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: WAITPKG 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 (Intel® Xeon Phi™ only.) 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 and IA32_TSC_AUX are available if 1. Bits 24 - 23: Reserved Bit 25: CLDEMOTE. Supports cache line demote if 1. Bit 26: Reserved Bit 27: MOVDIRI. Supports MOVDIRI if 1. Bit 28: MOVDIR64B. Supports MOVDIR64B if 1. Bit 29: ENQCMD: Supports Enqueue Stores if 1. Bit 30: SGX_LC. Supports SGX Launch Configuration if 1. Bit 31: Reserved
	EDX	Bits 01-00: Reserved Bit 02: AVX512_4VNNIW (Intel® Xeon Phi™ only.) Bit 03: AVX512_4FMAPS (Intel® Xeon Phi™ only.) Bit 04: Fast Short REP MOV Bits 07-05: Reserved Bit 08: AVX512_VP2INTERSECT Bits 17-09: Reserved Bit 18: PCONFIG Bits 25-19: Reserved Bit 26: Enumerates support for indirect branch restricted speculation (IBRS) and the indirect branch predictor barrier (IBPB). Processors that set this bit support the IA32_SPEC_CTRL MSR and the IA32_PRED_CMD MSR. They allow software to set IA32_SPEC_CTRL[0] (IBRS) and IA32_PRED_CMD[0] (IBPB). Bit 27: Enumerates support for single thread indirect branch predictors (STIBP). Processors that set this bit support the IA32_SPEC_CTRL MSR. They allow software to set IA32_SPEC_CTRL[1] (STIBP). Bit 28: Reserved Bit 29: Enumerates support for the IA32_ARCH_CAPABILITIES MSR. Bit 30: Enumerates support for the IA32_CORE_CAPABILITIES MSR. Bit 31: Enumerates support for Speculative Store Bypass Disable (SSBD). Processors that set this bit support the IA32_SPEC_CTRL MSR. They allow software to set IA32_SPEC_CTRL[2] (SSBD). **NOTE:** * 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.

Table 1-5. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	Structured Extended Feature Enumeration Sub-leaf (EAX = 07H, ECX = 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*. Bits 04-00: Reserved. Bit 05: AVX512_BF16. Vector Neural Network Instructions supporting BFLOAT16 inputs and conversion instructions from IEEE single precision. Bits 31-06: 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.
	Structured Extended Feature Enumeration Sub-leaves (EAX = 07H, ECX = n, n ≥ 2)	
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

Table 1-5. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	Extended Topology Enumeration Leaf	
0BH		**NOTES:** *CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of Leaf 1FH before using leaf 0BH.* 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 31-05: Reserved.
	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 12-10: Reserved. Bit 13: Used for IA32_XSS. Bits 31-14: Reserved.

Table 1-5. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	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 31-00: 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
	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 12-10: Reserved. Bit 13: HWP state. Bits 31-14: 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 31-00: 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 31-00: 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.

Table 1-5. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
Intel Resource Director Technology (Intel RDT) 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 Intel RDT Monitoring if 1. Bits 31-02: Reserved
L3 Cache Intel RDT 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 31-00: Conversion factor from reported IA32_QM_CTR value to occupancy metric (bytes) and Memory Bandwidth Monitoring (MBM) metrics.
	ECX	Maximum range (zero-based) of RMID of this resource type.
	EDX	Bit 00: Supports L3 occupancy monitoring if 1. Bit 01: Supports L3 Total Bandwidth monitoring if 1. Bit 02: Supports L3 Local Bandwidth monitoring if 1. Bits 31-03: Reserved
Intel Resource Director Technology (Intel RDT) Allocation 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 Allocation Technology if 1. Bit 02: Supports L2 Cache Allocation Technology if 1. Bit 03: Supports Memory Bandwidth Allocation if 1. Bits 31-04: Reserved.
	ECX	Reserved.
	EDX	Reserved.
L3 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID =1)		
10H		**NOTES:** Leaf 10H output depends on the initial value in ECX.
	EAX	Bits 04-00: Length of the capacity bit mask for the corresponding ResID using minus-one notation. Bits 31-05: Reserved
	EBX	Bits 31-00: 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

Table 1-5. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 15-00: Highest COS number supported for this ResID. Bits 31-16: Reserved
L2 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID =2)		
10H		**NOTES:** Leaf 10H output depends on the initial value in ECX.
	EAX	Bits 04 - 00: Length of the capacity bit mask for the corresponding ResID using minus-one notation. Bits 31 - 05: Reserved.
	EBX	Bits 31 - 00: Bit-granular map of isolation/contention of allocation units.
	ECX	Bits 31 - 00: Reserved.
	EDX	Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.
Memory Bandwidth Allocation Enumeration Sub-leaf (EAX = 10H, ECX = ResID =3)		
10H		**NOTES:** Leaf 10H output depends on the initial value in ECX.
	EAX	Bits 11 - 00: Reports the maximum MBA throttling value supported for the corresponding ResID using minus-one notation. Bits 31 - 12: Reserved.
	EBX	Bits 31 - 00: Reserved.
	ECX	Bits 01 - 00: Reserved. Bit 02: Reports whether the response of the delay values is linear. Bits 31 - 03: Reserved.
	EDX	Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.
Intel® Software Guard Extensions (Intel® SGX) Capability Enumeration Leaf, sub-leaf 0 (EAX = 12H, ECX = 0)		
12H		**NOTES:** Leaf 12H sub-leaf 0 (ECX = 0) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.
	EAX	Bit 00: SGX1. If 1, Indicates Intel SGX supports the collection of SGX1 leaf functions. Bit 01: SGX2. If 1, Indicates Intel SGX supports the collection of SGX2 leaf functions. Bits 04-02: Reserved. Bit 05: If 1, indicates Intel SGX supports ENCLV instruction leaves EINCVIRTCHILD, EDECVIRTCHILD, and ESETCONTEXT. Bit 06: If 1, indicates Intel SGX supports ENCLS instruction leaves ETRACKC, ERDINFO, ELDBC, and ELDUC. Bits 31-07: Reserved.
	EBX	Bits 31-00: MISCSELECT. Bit vector of supported extended Intel SGX features.
	ECX	Bits 31-00: Reserved.
	EDX	Bits 07-00: MaxEnclaveSize_Not64. The maximum supported enclave size in non-64-bit mode is 2^(EDX[7:0]). Bits 15-08: MaxEnclaveSize_64. The maximum supported enclave size in 64-bit mode is 2^(EDX[15:8]). Bits 31-16: Reserved.

Table 1-5. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	Intel SGX Attributes Enumeration Leaf, sub-leaf 1 (EAX = 12H, ECX = 1)	
12H		**NOTES:** Leaf 12H sub-leaf 1 (ECX = 1) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.
	EAX	Bit 31-00: Reports the valid bits of SECS.ATTRIBUTES[31:0] that software can set with ECREATE.
	EBX	Bit 31-00: Reports the valid bits of SECS.ATTRIBUTES[63:32] that software can set with ECREATE.
	ECX	Bit 31-00: Reports the valid bits of SECS.ATTRIBUTES[95:64] that software can set with ECREATE.
	EDX	Bit 31-00: Reports the valid bits of SECS.ATTRIBUTES[127:96] that software can set with ECREATE.
	Intel SGX EPC Enumeration Leaf, sub-leaves (EAX = 12H, ECX = 2 or higher)	
12H		**NOTES:** Leaf 12H sub-leaf 2 or higher (ECX >= 2) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1. For sub-leaves (ECX = 2 or higher), definition of EDX,ECX,EBX,EAX[31:4] depends on the sub-leaf type listed below.
	EAX	Bit 03-00: Sub-leaf Type 0000b: Indicates this sub-leaf is invalid. 0001b: This sub-leaf enumerates an EPC section. EBX:EAX and EDX:ECX provide information on the Enclave Page Cache (EPC) section. All other type encodings are reserved.
	Type	0000b. This sub-leaf is invalid. EDX:ECX:EBX:EAX return 0.
	Type	0001b. This sub-leaf enumerates an EPC sections with EDX:ECX, EBX:EAX defined as follows. EAX[11:04]: Reserved (enumerate 0). EAX[31:12]: Bits 31:12 of the physical address of the base of the EPC section. EBX[19:00]: Bits 51:32 of the physical address of the base of the EPC section. EBX[31:20]: Reserved. ECX[03:00]: EPC section property encoding defined as follows: If EAX[3:0] 0000b, then all bits of the EDX:ECX pair are enumerated as 0. If EAX[3:0] 0001b, then this section has confidentiality and integrity protection. All other encodings are reserved. ECX[11:04]: Reserved (enumerate 0). ECX[31:12]: Bits 31:12 of the size of the corresponding EPC section within the Processor Reserved Memory. EDX[19:00]: Bits 51:32 of the size of the corresponding EPC section within the Processor Reserved Memory. EDX[31:20]: Reserved.

Table 1-5. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	Intel Processor Trace Enumeration Main Leaf (EAX = 14H, ECX = 0)	
14H		**NOTES:** Leaf 14H main leaf (ECX = 0).
	EAX	Bits 31-00: 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. Bit 04: If 1, indicates support of PTWRITE. Writes can set IA32_RTIT_CTL[12] (PTWEn) and IA32_R-TIT_CTL[5] (FUPonPTW), and PTWRITE can generate packets. Bit 05: If 1, indicates support of Power Event Trace. Writes can set IA32_RTIT_CTL[4] (PwrEvtEn), enabling Power Event Trace packet generation. Bits 31-06: Reserved
	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 02-00: Number of configurable Address Ranges for filtering. Bits 15-03: Reserved Bit 31-16: Bitmap of supported MTC period encodings
	EBX	Bits 15-00: 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 31-00: An unsigned integer which is the denominator of the TSC/"core crystal clock" ratio.
	EBX	Bits 31-00: An unsigned integer which is the numerator of the TSC/"core crystal clock" ratio.
	ECX	Bits 31-00: An unsigned integer which is the nominal frequency of the core crystal clock in Hz.
	EDX	Bits 31-00: Reserved = 0.

Table 1-5. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	Processor Frequency Information Leaf	
16H	EAX	Bits 15-00: Processor Base Frequency (in MHz). Bits 31-16: Reserved =0
	EBX	Bits 15-00: Maximum Frequency (in MHz). Bits 31-16: Reserved = 0
	ECX	Bits 15-00: 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 1-5. 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 1-5. 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.
	PCONFIG Information Sub-leaf (EAX = 1BH, ECX ≥ 0)	
1BH		**NOTES:** Leaf 1BH is supported if CPUID.(EAX=07H, ECX=0H):EDX[18] = 1. For sub-leaves of 1BH, the definition of EDX, ECX, EBX, EAX depends on the sub-leaf type listed below. * Currently MKTME is the only defined target and is indicated by identifier 1. An identifier of 0 indicates an invalid target. If MKTME is a supported target, the MKTME_KEY_PROGRAM leaf of PCONFIG is available.
	EAX	Bits 11-00: Sub-leaf type 0: Invalid sub-leaf. On an invalid sub-leaf type returned, subsequent sub-leaves are also invalid. EBX, ECX and EDX all return 0 for this case. 1: Target Identifier. This sub-leaf enumerates PCONFIG targets supported on the platform. Software must scan until an invalid sub-leaf type is returned. EBX, ECX and EDX are defined below for this case. Bits 31-12: 0
	EBX	* Identifier of target 3n+1 (where n is the sub-leaf number, the initial value of ECX).
	ECX	* Identifier of target 3n+2.
	EDX	* Identifier of target 3n+3.

Table 1-5. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	V2 Extended Topology Enumeration Leaf	
1FH		**NOTES:** *CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of Leaf 1FH and using this if available.* Most of Leaf 1FH output depends on the initial value in ECX. The EDX output of leaf 1FH 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]. Sub-leaf index 0 enumerates SMT level. Each subsequent higher sub-leaf index enumerates a higher-level topological entity in hierarchical order. For sub-leaves that return 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 - 05: Reserved.
	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: Module. 4: Tile. 5: Die. 6-255: 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 1-6).
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved

Table 1-5. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
80000001H	EAX	Extended Processor Signature and Feature Bits.
	EBX	Reserved
	ECX	Bit 00: LAHF/SAHF available in 64-bit mode Bits 04-01: Reserved Bit 05: LZCNT available Bits 07-06: Reserved Bit 08: PREFETCHW Bits 31-09: Reserved
	EDX	Bits 10-00: 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
80000002H	EAX	Processor Brand String
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
80000003H	EAX	Processor Brand String Continued
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
80000004H	EAX	Processor Brand String Continued
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
80000005H	EAX	Reserved = 0
	EBX	Reserved = 0
	ECX	Reserved = 0
	EDX	Reserved = 0
80000006H	EAX	Reserved = 0
	EBX	Reserved = 0
	ECX	Bits 07-00: Cache Line size in bytes Bits 11-08: Reserved Bits 15-12: L2 Associativity field * Bits 31-16: Cache size in 1K units
	EDX	Reserved = 0

Table 1-5. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
		NOTES: * L2 associativity field encodings: 00H - Disabled 01H - 1 way (direct mapped) 02H - 2 ways 03H - Reserved 04H - 4 ways 05H - Reserved 06H - 8 ways 07H - See CPUID leaf 04H, sub-leaf 2** 08H - 16 ways 09H - Reserved 0AH - 32 ways 0BH - 48 ways 0CH - 64 ways 0DH - 96 ways 0EH - 128 ways 0FH - Fully associative ** CPUID leaf 04H provides details of deterministic cache parameters, including the L2 cache in sub-leaf 2
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 07-00: #Physical Address Bits* Bits 15-08: #Virtual Address Bits Bits 31-16: Reserved = 0 Bits 08-00: Reserved = 0 Bit 09: WBNOINVD is available if 1 Bits 31-10: 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 1-6) 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 1-6) and is processor specific.

Table 1-6. 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 1-3). 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 1-7 for available processor type values. Stepping IDs are provided as needed.

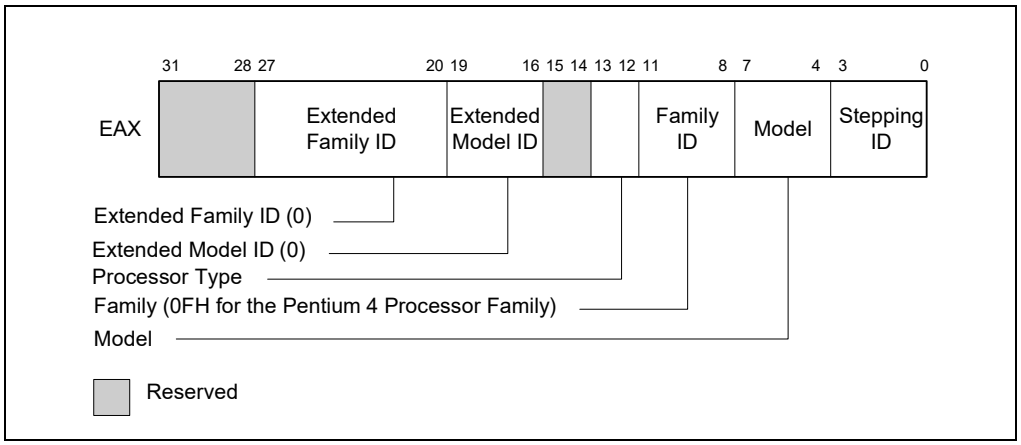

Figure 1-3. Version Information Returned by CPUID in EAX

Table 1-7. 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 1-4 and Table 1-8 show encodings for ECX.
- Figure 1-5 and Table 1-9 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.

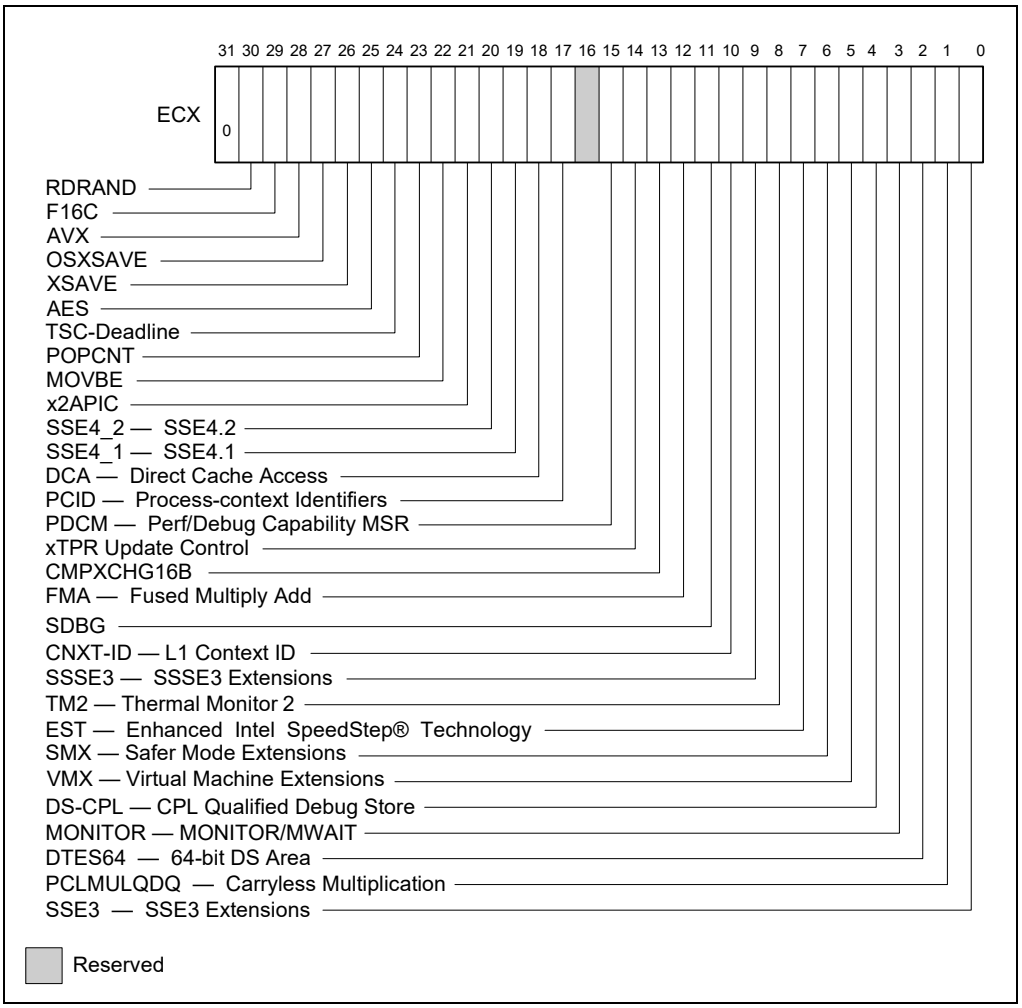

Figure 1-4. Feature Information Returned in the ECX Register

Table 1-8. 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.
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.

Table 1-8. Feature Information Returned in the ECX Register (Continued)

Bit #	Mnemonic	Description
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.

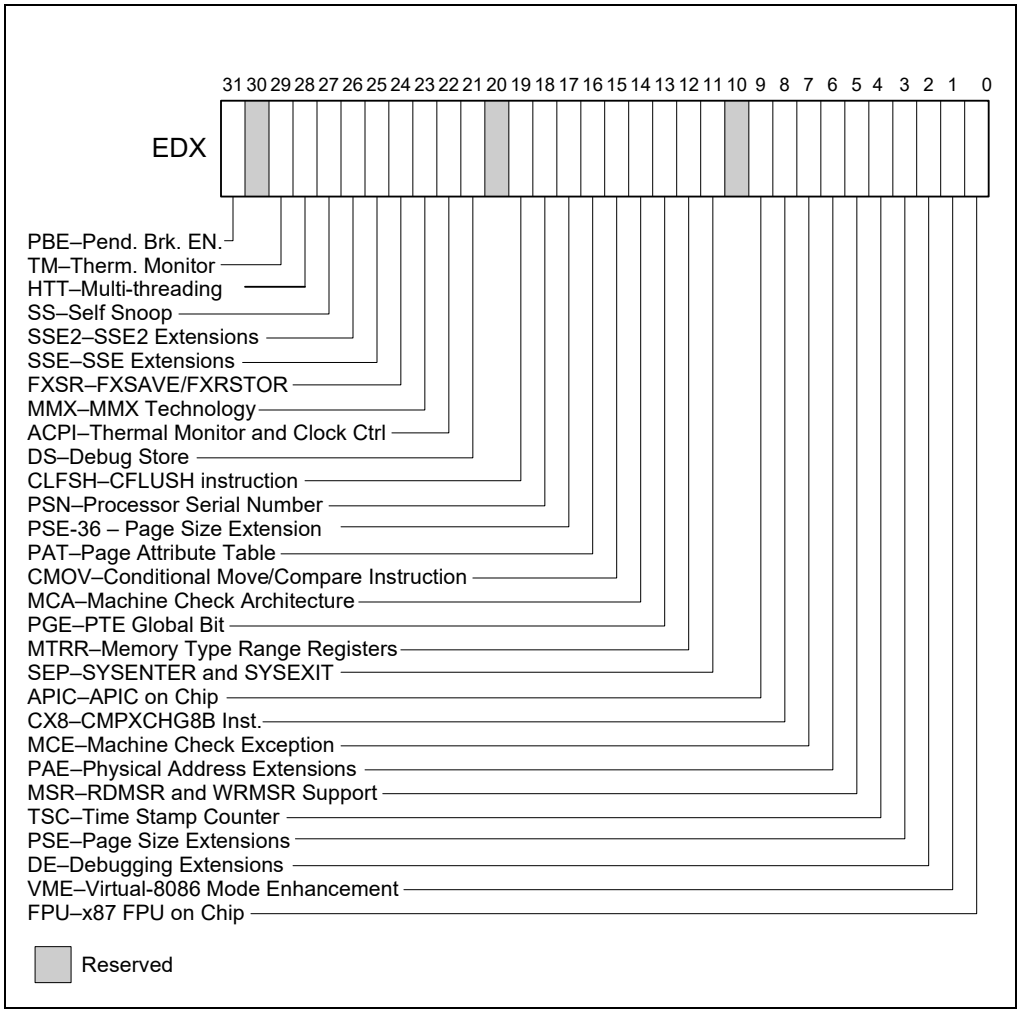

Figure 1-5. Feature Information Returned in the EDX Register

Table 1-9. 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.

Table 1-9. More on Feature Information Returned in the EDX Register(Continued)

Bit #	Mnemonic	Description
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.
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*).

Table 1-9. More on Feature Information Returned in the EDX Register(Continued)

Bit #	Mnemonic	Description
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.
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 1-10 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 1-10. 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

Table 1-10. Encoding of Cache and TLB Descriptors (Continued)

Descriptor Value	Cache or TLB Description
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
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

Table 1-10. Encoding of Cache and TLB Descriptors (Continued)

Descriptor Value	Cache or TLB Description
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
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 1-5.

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 1-5.

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 1-5.

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 1-5.

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 1-5. In sub-leaf 0, only EAX has the number of sub-leaves. In sub-leaf 0, EBX, ECX & EDX all contain extended feature flags.

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 1-5.

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 1-5) is greater than Pn 0. See Table 1-5.

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

CPUID leaf 1FH is a preferred superset to leaf 0BH. Intel recommends first checking for the existence of Leaf 1FH before using leaf 0BH.

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 1-5.

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 1-5.

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 1-5. 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 Intel Resource Director Technology (Intel RDT) 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 1-5.

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 Intel Resource Director Technology (Intel RDT) Allocation 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 1-5.

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 = 12H: Returns Intel SGX Enumeration Information

When CPUID executes with EAX set to 12H and ECX = 0H, the processor returns information about Intel SGX capabilities. See Table 1-5.

When CPUID executes with EAX set to 12H and ECX = 1H, the processor returns information about Intel SGX attributes. See Table 1-5.

When CPUID executes with EAX set to 12H and ECX = n (n > 1), the processor returns information about Intel SGX Enclave Page Cache. See Table 1-5.

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 1-5.

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 1-5.

INPUT EAX = 15H: Returns Time Stamp Counter and Nominal 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 1-5.

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 1-5.

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 1-5.

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 1-5.

INPUT EAX = 1BH: Returns PCONFIG Information

When CPUID executes with EAX set to 1BH, the processor returns information about PCONFIG capabilities. See Table 1-3.

INPUT EAX = 1FH: Returns V2 Extended Topology Information

When CPUID executes with EAX set to 1FH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 1FH by verifying (a) the highest leaf index supported by CPUID is >= 1FH, and (b) CPUID.1FH:EBX[15:0] reports a non-zero value. See Table 1-5.

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 1-6 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.

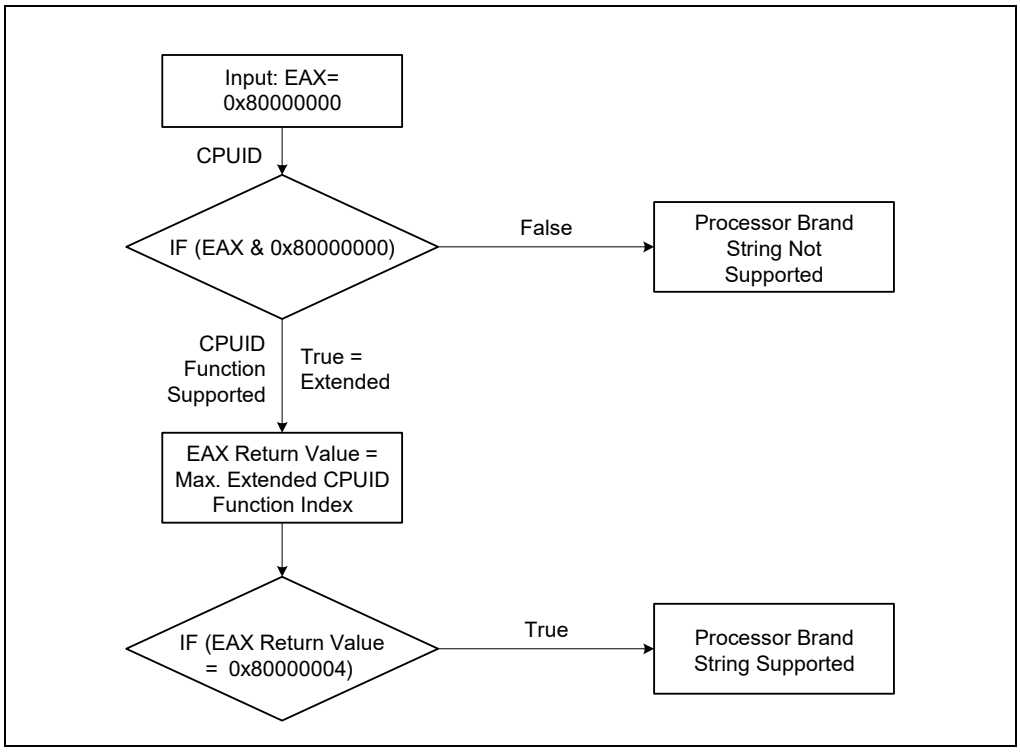

Figure 1-6. Determination of Support for the Processor Brand String

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 1-11 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

Table 1-11. 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 EBX = 50202952H ECX = 69746E65H EDX = 52286D75H	“(let” “P)R” “itne” “R(mu”
80000004H	EAX = 20342029H EBX = 20555043H ECX = 30303531H EDX = 007A484DH	“ 4)” “ UPC” “0051” “\0zHM”

Extracting the Maximum Processor Frequency from Brand Strings

Figure 1-7 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.

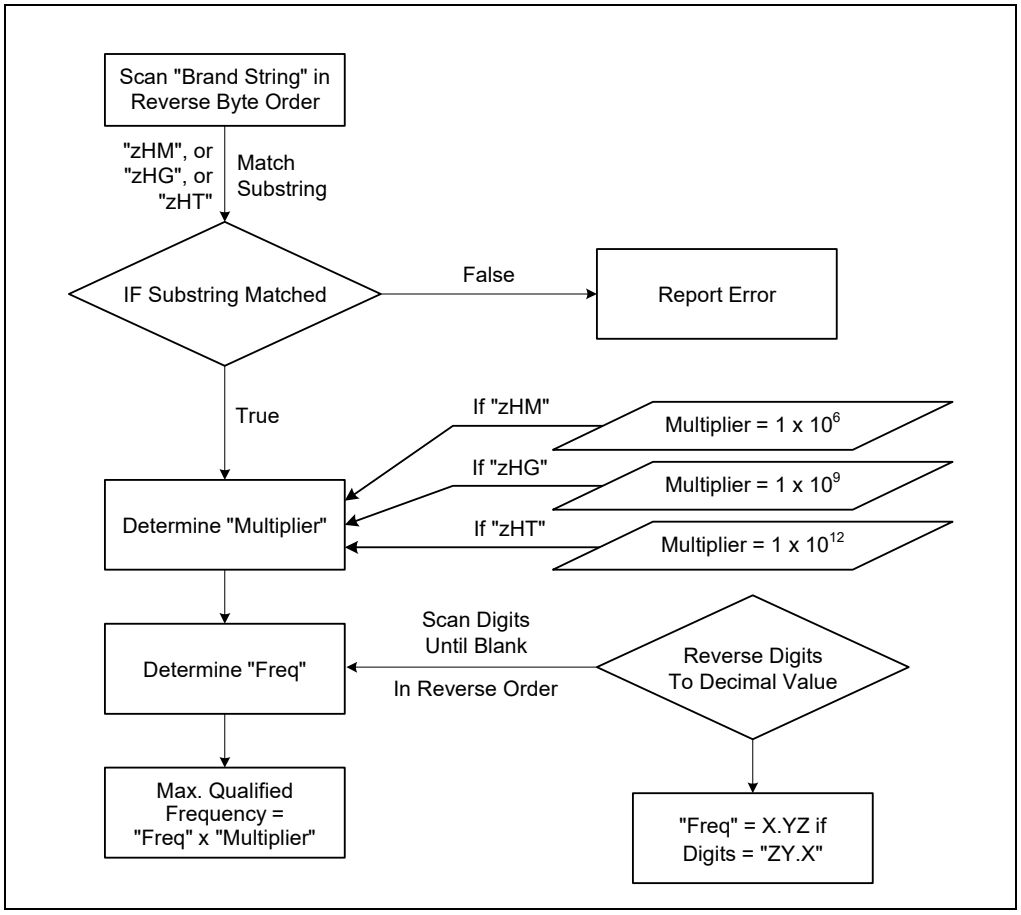

Figure 1-7. Algorithm for Extracting Maximum Processor Frequency

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 1-12 shows brand indices that have identification strings associated with them.

Table 1-12. 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 1-4. *)
        EDX ← Feature flags; (* See Figure 1-5. *)
    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 1-5. *)
        EBX ← Deterministic Cache Parameters Leaf;
        ECX ← Deterministic Cache Parameters Leaf;
        EDX ← Deterministic Cache Parameters Leaf;
    BREAK;
    EAX = 5H:
        EAX ← MONITOR/MWAIT Leaf; (* See Table 1-5. *)
        EBX ← MONITOR/MWAIT Leaf;
        ECX ← MONITOR/MWAIT Leaf;
        EDX ← MONITOR/MWAIT Leaf;
    BREAK;
    EAX = 6H:
        EAX ← Thermal and Power Management Leaf; (* See Table 1-5. *)
        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 1-5. *);
        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 1-5. *)
        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 1-5. *)
        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 1-5. *)
        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 1-5. *)
        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 1-5. *)
        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 1-5. *)
        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 = 12H:
        EAX ← Intel SGX Enumeration Leaf; (* See Table 1-5. *)
```

```
        EBX ← Intel SGX Enumeration Leaf;
        ECX ← Intel SGX Enumeration Leaf;
        EDX ← Intel SGX Enumeration Leaf;
    BREAK;
    EAX = 14H:
        EAX ← Intel Processor Trace Enumeration Leaf; (* See Table 1-5. *)
        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 1-5. *)
        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 1-5. *)
        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 1-5. *)
        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 1-5. *)
        EBX ← Deterministic Address Translation Parameters Enumeration Leaf;
        ECX ←Deterministic Address Translation Parameters Enumeration Leaf;
        EDX ← Deterministic Address Translation Parameters Enumeration Leaf;
    BREAK;
    EAX = 1BH:
        EAX ← PCONFIG Information Enumeration Leaf; (* See Table 1-5. *)
        EBX ← PCONFIG Information Enumeration Leaf;
        ECX ←PCONFIG Information Enumeration Leaf;
        EDX ← PCONFIG Information Enumeration Leaf;
    BREAK;
    EAX = 1FH:
        EAX ← V2 Extended Topology Enumeration Leaf; (* See Table 1-5. *)
        EBX ← V2 Extended Topology Enumeration Leaf;
        ECX ← V2 Extended Topology Enumeration Leaf;
        EDX ← V2 Extended Topology 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 1-5.*);
        EDX ← Extended Feature Bits (* See Table 1-5. *);
    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.§

1.8 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 1-13 and Table 1-14 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 1-13 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 1-14. Table 1-14 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 1-14. Instruction classified in Table 1-14 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 1-13. 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 1-14. 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 1-14. 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

1.9 BFLOAT16 FLOATING-POINT FORMAT

Intel® Deep Learning Boost (Intel® DL Boost) uses bfloat16 format (BF16). Figure 1-8 illustrates BF16 versus FP16 and FP32.

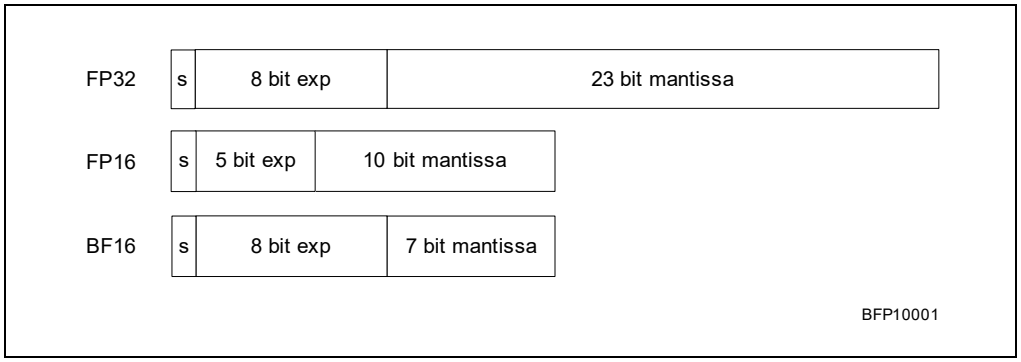

Figure 1-8. Comparison of BF16 to FP16 and FP32

BF16 has several advantages over FP16:

- It can be seen as a short version of FP32, skipping the least significant 16 bits of mantissa.
- There is no need to support denormals; FP32, and therefore also BF16, offer more than enough range for deep learning training tasks.
- FP32 accumulation after the multiply is essential to achieve sufficient numerical behavior on an application level.
- Hardware exception handling is not needed as this is a performance optimization; industry is designing algorithms around checking inf/NaN.

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

ENQCMD — Enqueue Command

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 38 F8 /r ENQCMD r32/64, m512	A	V/V	ENQCMD	Atomically enqueue 64-byte user command with PASID from source memory operand to destination offset in ES segment specified in register operand as offset in ES segment.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

The ENQCMD instruction allows software to write commands to **enqueue registers**, which are special device registers accessed using memory-mapped I/O (MMIO).

Enqueue registers expect writes to have the following format:

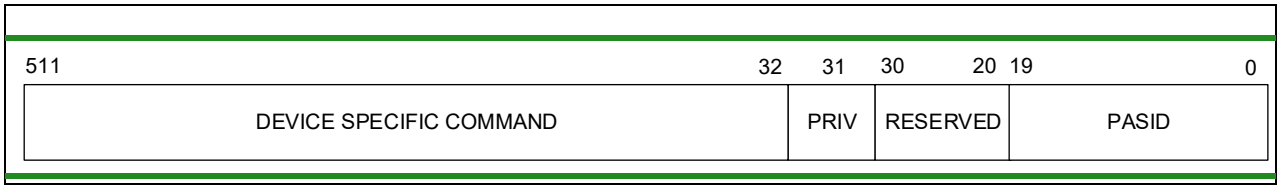

Figure 2-1. 64-Byte Data Written to Enqueue Registers

Bits 19:0 conveys the process address space identifier (PASID), a value which system software may assign to individual software threads. Bit 31 contains privilege identification (0 = user; 1 = supervisor). Devices implementing enqueue registers may use these two values along with a device-specific command in the upper 60 bytes. Chapter 3 provides more details regarding how ENQCMD uses PASIDs.

The ENQCMD instruction begins by reading 64 bytes of command data from its source memory operand. (The source operand is a normal memory operand; ModR/M.mod ≠ 11b.) This is an ordinary load with cacheability and memory ordering implied normally by the memory type. The source operand need not be aligned, and there is no guarantee that all 64 bytes are loaded atomically.

The instruction then formats those 64 bytes into **command data** with a format consistent with that given in Figure 2-1:

- Command[19:0] get IA32_PASID[19:0].[1]
- Command[30:20] are zero.
- Command[31] is 0 (indicating user).
- Command[511:32] get bits 511:32 of the source operand that was read from memory.

(The instruction ignores bits 31:0 of the source operand.)

The ENQCMD instruction uses an **enqueue store** (defined below) to write these command data to the destination operand. The address of the destination operand is specified in a general purpose register as an offset into the ES segment (the segment cannot be overridden).[2] The destination linear address must be 64-byte aligned. The operation of an enqueue store disregards the memory type of the destination memory address.

1. It is expected that system software will load the IA32_PASID MSR so that bits 19:0 contain the PASID of the current software thread. The MSR's valid bit, IA32_PASID[31], must be 1. The PASID MSR is discussed in more detail in Section 3.1.
2. In 64-bit mode, the width of the register operand is 64 bits (32 bits with a 67H prefix). Outside 64-bit mode when CS.D = 1, the width is 32 bits (16 bits with a 67H prefix). Outside 64-bit mode when CS.D=0, the width is 16 bits (32 bits with a 67H prefix).

An enqueue store is not ordered relative to older stores to WB or WC memory (including non-temporal stores) or to executions of the CLFLUSHOPT or CLWB (when applied to addresses other than that of the enqueue store). Software can enforce such ordering by executing a fencing instruction such as SFENCE or MFENCE before the enqueue store.

An enqueue store does not write the data into the cache hierarchy, nor does it fetch any data into the cache hierarchy. An enqueue store's command data is never combined with that of any other store to the same address.

Unlike other stores, an enqueue store returns a status, which the ENQCMD instruction loads into the ZF flag in the RFLAGS register:

- ZF = 0 (success) reports that the 64-byte command data was written atomically to a device's enqueue register and has been accepted by the device. (It does not guarantee that the device has acted on the command; it may have queued it for later execution.)
- ZF = 1 (retry) reports that the command data was not accepted. This status is returned if the destination address is an enqueue register but the command was not accepted due to capacity or other temporal reasons. This status is also returned if the destination address was not an enqueue register (including the case of a memory address); in these cases, the store is dropped and is written neither to MMIO nor to memory.

Availability of the ENQCMD instruction is indicated by the presence of the CPUID feature flag ENQCMD (bit 29 of the ECX register, see "CPUID Instruction" in Chapter 1).

Operation

```
IF IA32_PASID[31] = 0
    THEN #GP;
ELSE
    COMMAND ← (SRC & ~FFFFFFFFH) | (IA32_PASID & FFFFFH);
    DEST ← COMMAND;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

ENQCMD int_enqcmd(void *dst, const void *src)

Flags Affected

The ZF flag is set if the enqueue-store completion returns the retry status; otherwise it is cleared. All other flags are cleared.

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If destination linear address is not aligned to a 64-byte boundary.
	If the PASID Valid field (bit 31) is 0 in IA32_PASID MSR.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.ENQCMD[bit 29] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If destination linear address is not aligned to a 64-byte boundary.
	If the PASID Valid field (bit 31) is 0 in IA32_PASID MSR.
#UD	If CPUID.07H.0H:ECX.ENQCMD[bit 29] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode. Additionally:

#PF(fault-code)	For a page fault.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in non-canonical form.
#GP(0)	If the memory address is in non-canonical form. If destination linear address is not aligned to a 64-byte boundary. If the PASID Valid field (bit 31) is 0 in IA32_PASID MSR.
#PF(fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.ENQCMD[bit 29]. If the LOCK prefix is used.

ENQCMDS — Enqueue Command Supervisor

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 38 F8 /r ENQCMDS r32/64, m512	A	V/V	ENQCMD	Atomically enqueue 64-byte command from source memory operand to destination offset in ES segment specified in register operand as offset in ES segment.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

The ENQCMDS instruction allows system software to write commands to **enqueue registers**, which are special device registers accessed using memory-mapped I/O (MMIO).

Enqueue registers expect writes to have the format given in Figure 2-1 and explained in the section on "ENQCMD — Enqueue Command."

The ENQCMDS instruction begins by reading 64 bytes of command data from its source memory operand. (The source operand is a normal memory operand; ModR/M.mod ≠ 11b.) This is an ordinary load with cacheability and memory ordering implied normally by the memory type. The source operand need not be aligned, and there is no guarantee that all 64 bytes are loaded atomically.

ENQCMDS formats its source data differently from ENQCMD. Specifically, it formats them into **command data** as follows:

- Command[19:0] get bits 19:0 of the source operand that was read from memory. These 20 bits communicate a process address-space identifier (PASID). Chapter 3 provides more details regarding how ENQCMDS uses PASIDs.
- Command[30:20] are zero.
- Command[511:31] get bits 511:31 of the source operand that was read from memory. Bit 31 communicates a privilege identification (0 = user; 1 = supervisor)

(The instruction ignores bits 30:20 of the source operand.)

The ENQCMDS instruction then uses an **enqueue store** (defined below) to write these command data to the destination operand. The address of the destination operand is specified in a general purpose register as an offset into the ES segment (the segment cannot be overridden).[1] The destination linear address must be 64-byte aligned. The operation of an enqueue store disregards the memory type of the destination memory address.

An enqueue store is not ordered relative to older stores to WB or WC memory (including non-temporal stores) or to executions of the CLFLUSHOPT or CLWB (when applied to addresses other than that of the enqueue store). Software can enforce such ordering by executing a fencing instruction such as SFENCE or MFENCE before the enqueue store.

An enqueue store does not write the data into the cache hierarchy, nor does it fetch any data into the cache hierarchy. An enqueue store's command data is never combined with that of any other store to the same address.

Unlike other stores, an enqueue store returns a status, which the ENQCMDS instruction loads into the ZF flag in the RFLAGS register:

- ZF = 0 (success) reports that the 64-byte command data was written atomically to a device's enqueue register and has been accepted by the device. (It does not guarantee that the device has acted on the command; it may have queued it for later execution.)

1. In 64-bit mode, the width of the register operand is 64 bits (32 bits with a 67H prefix). Outside 64-bit mode when CS.D = 1, the width is 32 bits (16 bits with a 67H prefix). Outside 64-bit mode when CS.D=0, the width is 16 bits (32 bits with a 67H prefix).

- ZF = 1 (retry) reports that the command data was not accepted. This status is returned if the destination address is an enqueue register but the command was not accepted due to capacity or other temporal reasons. This status is also returned if the destination address was not an enqueue register (including the case of a memory address); in these cases, the store is dropped and is written neither to MMIO nor to memory.

The ENQCMDS instruction may be executed only if CPL = 0. Availability of the ENQCMDS instruction is indicated by the presence of the CPUID feature flag ENQCMD (bit 29 of the ECX register, see "CPUID Instruction" in Chapter 1).

Operation

```
DEST ← SRC & ~7FF00000H;        // clear bits 30:20
```

Intel C/C++ Compiler Intrinsic Equivalent

ENQCMDS int_enqcmds(void *dst, const void *src)

Flags Affected

The ZF flag is set if the enqueue-store completion returns the retry status; otherwise it is cleared. All other flags are cleared.

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If destination linear address is not aligned to a 64-byte boundary.
	If the current privilege level is not 0.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.ENQCMD[bit 29] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If destination linear address is not aligned to a 64-byte boundary.
#UD	If CPUID.07H.0H:ECX.ENQCMD[bit 29] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The ENQCMDS instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in non-canonical form.
#GP(0)	If the memory address is in non-canonical form.
	If destination linear address is not aligned to a 64-byte boundary.
	If the current privilege level is not 0.
#PF(fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.ENQCMD[bit 29].
	If the LOCK prefix is used.

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.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.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.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.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.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)	imm8 (r)	NA
B	NA	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

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.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.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.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.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.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)	imm8 (r)	NA
B	NA	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

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.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.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.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.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.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 E4.

PCONFIG — Platform Configuration

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 C5 PCONFIG	A	V/V	PCONFIG	This instruction is used to execute functions for configuring platform features. EAX: Leaf function to be invoked. RBX/RCX/RDX: Leaf-specific purpose.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	NA	NA	NA	NA

Description

PCONFIG allows software to configure certain platform features. PCONFIG supports multiple leaf functions, with a leaf function identified by the value in EAX. The registers RBX, RCX, and RDX have leaf-specific purposes.

Each PCONFIG leaf function applies to a specific hardware block called a PCONFIG target, and each PCONFIG target is associated with a numerical identifier. The identifiers of the PCONFIG targets supported by the CPU (which imply the supported leaf functions) are enumerated in the sub-leaves of the PCONFIG-information leaf of CPUID (EAX = 1BH). An attempt to execute an undefined leaf function results in a general-protection exception (#GP).

Addresses and operands are 32 bits outside 64-bit mode (IA32_EFER.LMA = 0 || CS.L = 0) and are 64 bits in 64-bit mode (IA32_EFER.LMA = 1 && CS.L = 1). The value of CS.D has no effect on address calculation.

Table 2-2 shows the leaf encodings for PCONFIG.

Table 2-2. PCONFIG Leaf Encodings

Leaf	Encoding	Description
MKTME_KEY_PROGRAM	00000000H	This leaf is used to program the key and encryption mode associated with a KeyID.
RESERVED	00000001H - FFFFFFFFH	Reserved for future use (#GP(0) if used).

The MKTME_KEY_PROGRAM leaf of PCONFIG pertains to the MKTME target, which has target identifier 1. It is used by software to manage the key associated with a KeyID. The leaf function is invoked by setting the leaf value of 0 in EAX and the address of MKTME_KEY_PROGRAM_STRUCT in RBX. Successful execution of the leaf clears RAX (set to zero) and ZF, CF, PF, AF, OF, and SF are cleared. In case of failure, the failure reason is indicated in RAX with ZF set to 1 and CF, PF, AF, OF, and SF are cleared. The MKTME_KEY_PROGRAM leaf uses the MKTME_KEY_PROGRAM_STRUCT in memory shown in Table 2-3.

Table 2-3. MKTME_KEY_PROGRAM_STRUCT Format

Field	Offset (bytes)	Size (bytes)	Comments
KEYID	0	2	Key Identifier.
KEYID_CTRL	2	4	KeyID control: ▪ Bits [7:0]: COMMAND. ▪ Bits [23:8]: ENC_ALG. ▪ Bits [31:24]: Reserved, must be zero.
RESERVED	6	58	Reserved, must be zero.
KEY_FIELD_1	64	64	Software supplied KeyID data key or entropy for KeyID data key.
KEY_FIELD_2	128	64	Software supplied KeyID tweak key or entropy for KeyID tweak key.

A description of each of the fields in MKTME_KEY_PROGRAM_STRUCT is provided below:

- **KEYID:** Key Identifier being programmed to the MKTME engine.
- **KEYID_CTRL:** The KEYID_CTRL field carries two sub-fields used by software to control the behavior of a KeyID: Command and KeyID encryption algorithm.

  The command used controls the encryption mode for a KeyID. Table 2-4 provides a summary of the commands supported.

Table 2-4. Supported Key Programming Commands

Command	Encoding	Description
KEYID_SET_KEY_DIRECT	0	Software uses this mode to directly program a key for use with KeyID.
KEYID_SET_KEY_RANDOM	1	CPU generates and assigns an ephemeral key for use with a KeyID. Each time the instruction is executed, the CPU generates a new key using a hardware random number generator and the keys are discarded on reset.
KEYID_CLEAR_KEY	2	Clear the (software programmed) key associated with the KeyID. On execution of this command, the KeyID gets TME behavior (encrypt with platform TME key).
KEYID_NO_ENCRYPT	3	Do not encrypt memory when this KeyID is in use.

  The encryption algorithm field (ENC_ALG) allows software to select one of the activated encryption algorithms for the KeyID. The BIOS can activate a set of algorithms to allow for use when programming keys using the IA32_TME_ACTIVATE MSR (does not apply to KeyID 0 which uses TME policy). The ISA checks to ensure that the algorithm selected by software is one of the algorithms that has been activated by the BIOS.
- **KEY_FIELD_1:** This field carries the software supplied data key to be used for the KeyID if the direct key programming option is used (KEYID_SET_KEY_DIRECT). When the random key programming option is used (KEYID_SET_KEY_RANDOM), this field carries the software supplied entropy to be mixed in the CPU generated random data key. It is software's responsibility to ensure that the key supplied for the direct programming option or the entropy supplied for the random programming option does not result in weak keys. There are no explicit checks in the instruction to detect or prevent weak keys. When AES XTS-128 is used, the upper 48B are treated as reserved and must be zeroed out by software before executing the instruction.
- **KEY_FIELD_2:** This field carries the software supplied tweak key to be used for the KeyID if the direct key programming option is used (KEYID_SET_KEY_DIRECT). When the random key programming option is used (KEYID_SET_KEY_RANDOM), this field carries the software supplied entropy to be mixed in the CPU generated random tweak key. It is software's responsibility to ensure that the key supplied for the direct programming option or the entropy supplied for the random programming option does not result in weak keys. There are no explicit checks in the instruction to detect or prevent weak keys. When AES XTS-128 is used, the upper 48B are treated as reserved and must be zeroed out by software before executing the instruction.

  All KeyIDs use the TME key on MKTME activation. Software can at any point decide to change the key for a KeyID using the PCONFIG instruction. Change of keys for a KeyID does NOT change the state of the TLB caches or memory pipeline. It is software's responsibility to take appropriate actions to ensure correct behavior.

  Table 2-5 shows the return values associated with the MKTME_KEY_PROGRAM leaf of PCONFIG. On instruction execution, RAX is populated with the return value.

Table 2-5. Supported Key Programming Commands

Return Value	Encoding	Description
PROG_SUCCESS	0	KeyID was successfully programmed.
INVALID_PROG_CMD	1	Invalid KeyID programming command.
ENTROPY_ERROR	2	Insufficient entropy.
INVALID_KEYID	3	KeyID not valid.
INVALID_ENC_ALG	4	Invalid encryption algorithm chosen (not supported).
DEVICE_BUSY	5	Failure to access key table.

PCONFIG Virtualization

Software in VMX root mode can control the execution of PCONFIG in VMX non-root mode using the following execution controls introduced for PCONFIG:

- PCONFIG_ENABLE: This control is a single bit control and enables the PCONFIG instruction in VMX non-root mode. If 0, the execution of PCONFIG in VMX non-root mode causes #UD. Otherwise, execution of PCONFIG works according to PCONFIG_EXITING.
- PCONFIG_EXITING: This is a 64b control and allows VMX root mode to cause a VM-exit for various leaf functions of PCONFIG. This control does not have any effect if the PCONFIG_ENABLE control is clear.

PCONFIG Concurrency

In a scenario, where the MKTME_KEY_PROGRAM leaf of PCONFIG is executed concurrently on multiple logical processors, only one logical processor will succeed in updating the key table. PCONFIG execution will return with an error code (DEVICE_BUSY) on other logical processors and software must retry. In cases where the instruction execution fails with a DEVICE_BUSY error code, the key table is not updated, thereby ensuring that either the key table is updated in its entirety with the information for a KeyID, or it is not updated at all. In order to accomplish this, the MKTME_KEY_PROGRAM leaf of PCONFIG maintains a writer lock for updating the key table. This lock is referred to as the Key table lock and denoted in the instruction flows as KEY_TABLE_LOCK. The lock can either be unlocked, when no logical processor is holding the lock (also the initial state of the lock) or be in an exclusive state where a logical processor is trying to update the key table. There can be only one logical processor holding the lock in exclusive state. The lock, being exclusive, can only be acquired when the lock is in unlocked state.

PCONFIG uses the following syntax to acquire KEY_TABLE_LOCK in exclusive mode and release the lock:

- KEY_TABLE_LOCK.ACQUIRE(WRITE)
- KEY_TABLE_LOCK.RELEASE()

Operation

Table 2-6. PCONFIG Operation Variables

Variable Name	Type	Size (Bytes)	Description
TMP_KEY_PROGRAM_STRUCT	MKTME_KEY_PROGRAM_STRUCT	192	Structure holding the key programming structure.
TMP_RND_DATA_KEY	UINT128	16	Random data key generated for random key programming option.
TMP_RND_TWEAK_KEY	UINT128	16	Random tweak key generated for random key programming option.

```
(* #UD if PCONFIG is not enumerated or CPL>0 *)
if (CPUID.7.0:EDX[18] == 0 OR CPL > 0) #UD;

if (in VMX non-root mode)
{
    if (VMCS.PCONFIG_ENABLE == 1)
    {
        if ((EAX > 62 AND VMCS.PCONFIG_EXITING[63] ==1) OR
            (EAX < 63 AND VMCS.PCONFIG_EXITING[EAX] == 1))
        {
            Set VMCS.EXIT_REASON = PCONFIG; //No Exit qualification
            Deliver VMEXIT;
        }
    }
    else
    {
        #UD
```

```
    }
}

(* #GP(0) for an unsupported leaf *)
if(EAX != 0) #GP(0)

(* KEY_PROGRAM leaf flow *)
if (EAX == 0)
{
    (* #GP(0) if TME_ACTIVATE MSR is not locked or does not enable TME or multiple keys are not enabled *)
    if (IA32_TME_ACTIVATE.LOCK != 1 OR IA32_TME_ACTIVATE.ENABLE != 1 OR IA32_TME_ACTIVATE.MK_TME_KEYID_BITS == 0)
#GP(0)

    (* Check MKTME_KEY_PROGRAM_STRUCT is 256B aligned *)
    if(DS:RBX is not 256B aligned) #GP(0);

    (* Check that MKTME_KEY_PROGRAM_STRUCT is read accessible *)
    <<DS: RBX should be read accessible>>

    (* Copy MKTME_KEY_PROGRAM_STRUCT to a temporary variable *)
    TMP_KEY_PROGRAM_STRUCT = DS:RBX.*;

    (* RSVD field check *)
    if(TMP_KEY_PROGRAM_STRUCT.RSVD != 0) #GP(0);

    if(TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.RSVD !=0) #GP(0);

    if(TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1.BYTES[63:16] != 0) #GP(0);

    if(TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2.BYTES[63:16] != 0) #GP(0);

    (* Check for a valid command *)
    if(TMP_KEY_PROGRAM_STRUCT. KEYID_CTRL.COMMAND is not a valid command)
    {
        RFLAGS.ZF = 1;
        RAX = INVALID_PROG_CMD;
        goto EXIT;
    }
    (* Check that the KEYID being operated upon is a valid KEYID *)
    if(TMP_KEY_PROGRAM_STRUCT.KEYID >
                2^IA32_TME_ACTIVATE.MK_TME_KEYID_BITS - 1
        OR TMP_KEY_PROGRAM_STRUCT.KEYID >
                IA32_TME_CAPABILITY.MK_TME_MAX_KEYS
        OR TMP_KEY_PROGRAM_STRUCT.KEYID == 0)
    {
        RFLAGS.ZF = 1;
        RAX = INVALID_KEYID;
        goto EXIT;
    }

    (* Check that only one algorithm is requested for the KeyID and it is one of the activated algorithms *)
    if(NUM_BITS(TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG) != 1 ||
        (TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG &
            IA32_TME_ACTIVATE. MK_TME_CRYPTO_ALGS == 0))
```

```
{
    RFLAGS.ZF = 1;
    RAX = INVALID_ENC_ALG;
    goto EXIT;
}
(* Try to acquire exclusive lock *)
if (NOT KEY_TABLE_LOCK.ACQUIRE(WRITE))
{
    //PCONFIG failure
    RFLAGS.ZF = 1;
    RAX = DEVICE_BUSY;
    goto EXIT;
}

(* Lock is acquired and key table will be updated as per the command
    Before this point no changes to the key table are made *)

switch(TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.COMMAND)
{
case KEYID_SET_KEY_DIRECT:
    <<Write
        DATA_KEY=TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1,
        TWEAK_KEY=TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2,
        ENCRYPTION_MODE=ENCRYPT_WITH_KEYID_KEY,
        to MKTME Key table at index TMP_KEY_PROGRAM_STRUCT.KEYID
    >>
    break;

case KEYID_SET_KEY_RANDOM:
    TMP_RND_DATA_KEY = <<Generate a random key using hardware RNG>>
    if (NOT ENOUGH ENTROPY)
    {
        RFLAGS.ZF = 1;
        RAX = ENTROPY_ERROR;
        goto EXIT;
    }
    TMP_RND_TWEAK_KEY = <<Generate a random key using hardware RNG>>
    if (NOT ENOUGH ENTROPY)
    {
        RFLAGS.ZF = 1;
        RAX = ENTROPY_ERROR;
        goto EXIT;
    }
    (* Mix user supplied entropy to the data key and tweak key *)
    TMP_RND_DATA_KEY = TMP_RND_KEY XOR
        TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1.BYTES[15:0];
    TMP_RND_TWEAK_KEY = TMP_RND_TWEAK_KEY XOR
        TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2.BYTES[15:0];

    <<Write
        DATA_KEY=TMP_RND_DATA_KEY,
        TWEAK_KEY=TMP_RND_TWEAK_KEY,
        ENCRYPTION_MODE=ENCRYPT_WITH_KEYID_KEY,
        to MKTME_KEY_TABLE at index TMP_KEY_PROGRAM_STRUCT.KEYID
```

```
        >>
        break;

    case KEYID_CLEAR_KEY:
        <<Write
        DATA_KEY='0,
        TWEAK_KEY='0,
        ENCRYPTION_MODE = ENCRYPT_WITH_TME_KEY,
        to MKTME_KEY_TABLE at index TMP_KEY_PROGRAM_STRUCT.KEYID
        >>

        break;
    case KD_NO_ENCRYPT:
        <<Write
        ENCRYPTION_MODE=NO_ENCRYPTION,
        to MKTME_KEY_TABLE at index TMP_KEY_PROGRAM_STRUCT.KEYID
        >>
        break;
    }

    RAX = 0;
    RFLAGS.ZF = 0;

    //Release Lock
    KEY_TABLE_LOCK(RELEASE);

    EXIT:
    RFLAGS.CF=0;
    RFLAGS.PF=0;
    RFLAGS.AF=0;
    RFLAGS.OF=0;
    RFLAGS.SF=0;
}

end_of_flow
```

Intel C/C++ Compiler Intrinsic Equivalent

TBD

Protected Mode Exceptions

#GP(0)	If input value in EAX encodes an unsupported leaf. If IA32_TME_ACTIVATE MSR is not locked. If TME and MKTME capability are not enabled in IA32_TME_ACTIVATE MSR. If the memory operand is not 256B aligned. If any of the reserved bits in MKTME_KEY_PROGRAM_STRUCT are set. If a memory operand effective address is outside the DS segment limit.
#PF(fault-code)	If a page fault occurs in accessing memory operands.
#UD	If any of the LOCK/REP/OSIZE/VEX prefixes are used. If current privilege level is not 0. If CPUID.7.0:EDX[bit 18] = 0 If in VMX non-root mode and VMCS.PCONFIG_ENABLE = 0.

Real Address Mode Exceptions

#GP	If input value in EAX encodes an unsupported leaf.
	If IA32_TME_ACTIVATE MSR is not locked.
	If TME and MKTME capability is not enabled in IA32_TME_ACTIVATE MSR.
	If a memory operand is not 256B aligned.
	If any of the reserved bits in MKTME_KEY_PROGRAM_STRUCT are set.
#UD	If any of the LOCK/REP/OSIZE/VEX prefixes are used.
	If current privilege level is not 0.
	If CPUID.7.0:EDX.PCONFIG[bit 18] = 0
	If in VMX non-root mode and VMCS.PCONFIG_ENABLE = 0.

Virtual 8086 Mode Exceptions

#UD	PCONFIG instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If input value in EAX encodes an unsupported leaf.
	If IA32_TME_ACTIVATE MSR is not locked.
	If TME and MKTME capability is not enabled in IA32_TME_ACTIVATE MSR.
	If a memory operand is not 256B aligned.
	If any of the reserved bits in MKTME_KEY_PROGRAM_STRUCT are set.
	If a memory operand is non-canonical form.
#PF(fault-code)	If a page fault occurs in accessing memory operands.
#UD	If any of the LOCK/REP/OSIZE/VEX prefixes are used.
	If the current privilege level is not 0.
	If CPUID.7.0:EDX.PCONFIG[bit 18] = 0.
	If in VMX non-root mode and VMCS.PCONFIG_ENABLE = 0.

VAESDEC — Perform One Round of an AES Decryption Flow

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.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.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.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.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[MAXVL-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[MAXVL-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[MAXVL-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.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.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.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.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[MAXVL-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[MAXVL-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[MAXVL-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.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.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.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.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[MAXVL-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[MAXVL-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[MAXVL-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.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.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.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.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[MAXVL-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[MAXVL-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[MAXVL-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.

VCVTNE2PS2BF16 — Convert Two Packed Single Data to One Packed BF16 Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.0F38.W0 72 /r VCVTNE2PS2BF16 xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512VL AVX512_BF16	Convert packed single data from xmm2 and xmm3/m128/m32bcst to packed BF16 data in xmm1 with writemask k1.
EVEX.256.F2.0F38.W0 72 /r VCVTNE2PS2BF16 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512VL AVX512_BF16	Convert packed single data from ymm2 and ymm3/m256/m32bcst to packed BF16 data in ymm1 with writemask k1.
EVEX.512.F2.0F38.W0 72 /r VCVTNE2PS2BF16 zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512F AVX512_BF16	Convert packed single data from zmm2 and zmm3/m512/m32bcst to packed BF16 data in zmm1 with writemask k1.

Instruction Operand Encoding

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

Description

This instruction converts two SIMD registers of packed single data into a single register of packed BF16 data.

This instruction does not support memory fault suppression.

"Round to nearest even" rounding mode is used. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated.

Operation

VCVTNE2PS2BF16 dest, src1, src2

```
VL = (128, 256, 512)
KL = VL/16

origdest ← dest
FOR i ← 0 to KL-1:
    IF k1[ i ] or *no writemask*:
        IF src is memory and evex.b == 1:
            t ← src2.fp32[0]
        ELSE if i < k1/2:
            t ← src2.fp32[ i ]
        ELSE:
            t ← src1.fp32[ i-KL/2]

        // see for definition of convert helper function
        dest.word[i] ← convert_fp32_to_bfloat16(t)

    ELSE IF *zeroing*:
        dest.word[ i ] ← 0
    ELSE:  // merge masking, dest element unchanged
        dest.word[ i ] ← origdest.word[ i ]
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTNE2PS2BF16 __m128bh _mm_cvtne2ps_pbh (__m128, __m128);
VCVTNE2PS2BF16 __m128bh _mm_mask_cvtne2ps_pbh (__m128bh, __mmask8, __m128, __m128);
VCVTNE2PS2BF16 __m128bh _mm_maskz_cvtne2ps_pbh (__mmask8, __m128, __m128);
VCVTNE2PS2BF16 __m256bh _mm256_cvtne2ps_pbh (__m256, __m256);
VCVTNE2PS2BF16 __m256bh _mm256_mask_cvtne2ps_pbh (__m256bh, __mmask16, __m256, __m256);
VCVTNE2PS2BF16 __m256bh _mm256_maskz_cvtne2ps_ pbh (__mmask16, __m256, __m256);
VCVTNE2PS2BF16 __m512bh _mm512_cvtne2ps_pbh (__m512, __m512);
VCVTNE2PS2BF16 __m512bh _mm512_mask_cvtne2ps_pbh (__m512bh, __mmask32, __m512, __m512);
VCVTNE2PS2BF16 __m512bh _mm512_maskz_cvtne2ps_pbh (__mmask32, __m512, __m512);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.

VCVTNEPS2BF16 — Convert Packed Single Data to Packed BF16 Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 72 /r VCVTNEPS2BF16 xmm1{k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512_BF16	Convert packed single data from xmm2/m128 to packed BF16 data in xmm1 with writemask k1.
EVEX.256.F3.0F38.W0 72 /r VCVTNEPS2BF16 xmm1{k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512_BF16	Convert packed single data from ymm2/m256 to packed BF16 data in xmm1 with writemask k1.
EVEX.512.F3.0F38.W0 72 /r VCVTNEPS2BF16 ymm1{k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512F AVX512_BF16	Convert packed single data from zmm2/m512 to packed BF16 data in ymm1 with writemask k1.

Instruction Operand Encoding

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

Description

This instruction converts one SIMD register of packed single data into a single register of packed BF16 data.

"Round to nearest even" rounding mode is used. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated.

As the instruction operand encoding table shows, the EVEX.vvvv field is not used for encoding an operand. EVEX.vvvv is reserved and must be 0b1111 otherwise instructions will #UD.

Operation

```
Define convert_fp32_to_bfloat16(x):
    IF x is zero or denormal:
        dest[15] ← x[31] // sign preserving zero (denormal go to zero)
        dest[14:0] ← 0
    ELSE IF x is infinity:
        dest[15:0] ← x[31:16]
    ELSE IF x is NAN:
        dest[15:0] ← x[31:16] // truncate and set MSB of the mantisa force QNAN
        dest[6] ← 1
    ELSE // normal number
        LSB ← x[16]
        rounding_bias ← 0x00007FFF + LSB
        temp[31:0] ← x[31:0] + rounding_bias // integer add
        dest[15:0] ← temp[31:16]
    RETURN dest
```

VCVTNEPS2BF16 dest, src

```
VL = (128, 256, 512)
KL = VL/16

origdest ← dest
FOR i ← 0 to KL/2-1:
    IF k1[ i ] or *no writemask*:
        IF src is memory and evex.b == 1:
            t ← src.fp32[0]
        ELSE:
            t ← src.fp32[ i ]

        dest.word[i] ← convert_fp32_to_bfloat16(t)

    ELSE IF *zeroing*:
        dest.word[ i ] ← 0
    ELSE:  // merge masking, dest element unchanged
        dest.word[ i ] ← origdest.word[ i ]
DEST[MAXVL-1:VL/2] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTNEPS2BF16 __m128bh _mm_cvtneps_pbh (__m128);
VCVTNEPS2BF16 __m128bh _mm_mask_cvtneps_pbh (__m128bh, __mmask8, __m128);
VCVTNEPS2BF16 __m128bh _mm_maskz_cvtneps_pbh (__mmask8, __m128);
VCVTNEPS2BF16 __m128bh _mm256_cvtneps_pbh (__m256);
VCVTNEPS2BF16 __m128bh _mm256_mask_cvtneps_pbh (__m128bh, __mmask8, __m256);
VCVTNEPS2BF16 __m128bh _mm256_maskz_cvtneps_pbh (__mmask8, __m256);
VCVTNEPS2BF16 __m256bh _mm512_cvtneps_pbh (__m512);
VCVTNEPS2BF16 __m256bh _mm512_mask_cvtneps_pbh (__m256bh, __mmask16, __m512);
VCVTNEPS2BF16 __m256bh _mm512_maskz_cvtneps_pbh (__mmask16, __m512);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.

VDPBF16PS — Dot Product of BF16 Pairs Accumulated into Packed Single Precision

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 52 /r VDPBF16PS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512VL AVX512_BF16	Multiply BF16 pairs from xmm2 and xmm3/m128, and accumulate the resulting packed single precision results in xmm1 with writemask k1.
EVEX.256.F3.0F38.W0 52 /r VDPBF16PS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512VL AVX512_BF16	Multiply BF16 pairs from ymm2 and ymm3/m256, and accumulate the resulting packed single precision results in ymm1 with writemask k1.
EVEX.512.F3.0F38.W0 52 /r VDPBF16PS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512F AVX512_BF16	Multiply BF16 pairs from zmm2 and zmm3/m512, and accumulate the resulting packed single precision results in zmm1 with writemask k1.

Instruction Operand Encoding

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

Description

This instruction performs a SIMD dot-product of two BF16 pairs and accumulates into a packed single precision register.

"Round to nearest even" rounding mode is used. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated.

Operation

```
Define make_fp32(x):
    // The x parameter is bfloat16. Pack it in to upper 16b of a dword. The bit pattern is a legal fp32 value. Return that bit pattern.
    dword ← 0
    dword[31:16] ← x
    RETURN dword
```

VDPBF16PS srcdest, src1, src2

```
VL = (128, 256, 512)
KL = VL/32

origdest ← srcdest
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 ]

        // FP32 FMA with daz in, ftz out and RNE rounding. MXCSR neither consulted nor updated.

        srcdest.fp32[ i ] += make_fp32(src1.bfloat16[2*i+1]) * make_fp32(t.bfloat[1])
        srcdest.fp32[ i ] += make_fp32(src1.bfloat16[2*i+0]) * make_fp32(t.bfloat[0])

    ELSE IF *zeroing*:
        srcdest.dword[ i ] ← 0
    ELSE: // merge masking, dest element unchanged
        srcdest.dword[ i ] ← origdest.dword[ i ]

srcdest[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VDPBF16PS __m128 _mm_dpbf16_ps(__m128, __m128bh, __m128bh);
VDPBF16PS __m128 _mm_mask_dpbf16_ps(__m128, __mmask8, __m128bh, __m128bh);
VDPBF16PS __m128 _mm_maskz_dpbf16_ps(__mmask8, __m128, __m128bh, __m128bh);
VDPBF16PS __m256 _mm256_dpbf16_ps(__m256, __m256bh, __m256bh);
VDPBF16PS __m256 _mm256_mask_dpbf16_ps(__m256, __mmask8, __m256bh, __m256bh);
VDPBF16PS __m256 _mm256_maskz_dpbf16_ps(__mmask8, __m256, __m256bh, __m256bh);
VDPBF16PS __m512 _mm512_dpbf16_ps(__m512, __m512bh, __m512bh);
VDPBF16PS __m512 _mm512_mask_dpbf16_ps(__m512, __mmask16, __m512bh, __m512bh);
VDPBF16PS __m512 _mm512_maskz_dpbf16_ps(__mmask16, __m512, __m512bh, __m512bh);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.

VP2INTERSECTD/VP2INTERSECTQ — Compute Intersection Between DWORDS/QUADWORDS to a Pair of Mask Registers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.F2.0F38.W0 68 /r VP2INTERSECTD k1+1, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512VL AVX512_VP2INTERSECT	Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between dwords in xmm3/m128/m32bcst and xmm2.
EVEX.NDS.256.F2.0F38.W0 68 /r VP2INTERSECTD k1+1, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512VL AVX512_VP2INTERSECT	Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between dwords in ymm3/m256/m32bcst and ymm2.
EVEX.NDS.512.F2.0F38.W0 68 /r VP2INTERSECTD k1+1, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512F AVX512_VP2INTERSECT	Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between dwords in zmm3/m512/m32bcst and zmm2.
EVEX.NDS.128.F2.0F38.W1 68 /r VP2INTERSECTQ k1+1, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512VL AVX512_VP2INTERSECT	Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between quadwords in xmm3/m128/m64bcst and xmm2.
EVEX.NDS.256.F2.0F38.W1 68 /r VP2INTERSECTQ k1+1, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512VL AVX512_VP2INTERSECT	Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between quadwords in ymm3/m256/m64bcst and ymm2.
EVEX.NDS.512.F2.0F38.W1 68 /r VP2INTERSECTQ k1+1, zmm2, zmm3/m512/m64bcst	A	V/V	AVX512F AVX512_VP2INTERSECT	Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between quadwords in zmm3/m512/m64bcst and zmm2.

Instruction Operand Encoding

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

Description

This instruction writes an even/odd pair of mask registers. The mask register destination indicated in the MODRM.REG field is used to form the basis of the register pair. The low bit of that field is masked off (set to zero) to create the first register of the pair.

EVEX.aaa and EVEX.z must be zero.

Operation

VP2INTERSECTD destmask, src1, src2

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
// dest_mask_reg_id is the register id specified in the instruction for destmask
dest_base ← dest_mask_reg_id & ~1

// maskregs[ ] is an array representing the mask registers
maskregs[dest_base+0][MAX_KL-1:0] ← 0
maskregs[dest_base+1][MAX_KL-1:0] ← 0

FOR i ← 0 to KL-1:
    FOR j ← 0 to KL-1:
        match ← (src1.dword[i] == src2.dword[j])
        maskregs[dest_base+0].bit[i] |= match
        maskregs[dest_base+1].bit[j] |= match
```

VP2INTERSECTQ destmask, src1, src2

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
// dest_mask_reg_id is the register id specified in the instruction for destmask
dest_base ← dest_mask_reg_id & ~1

// maskregs[ ] is an array representing the mask registers
maskregs[dest_base+0][MAX_KL-1:0] ← 0
maskregs[dest_base+1][MAX_KL-1:0] ← 0

FOR i = 0 to KL-1:
    FOR j = 0 to KL-1:
        match ← (src1.qword[i] == src2.qword[j])
        maskregs[dest_base+0].bit[i] |=  match
        maskregs[dest_base+1].bit[j] |=  match
```

Intel C/C++ Compiler Intrinsic Equivalent

VP2INTERSECTD void _mm_2intersect_epi32(__m128i, __m128i, __mmask8 *, __mmask8 *);
VP2INTERSECTD void _mm256_2intersect_epi32(__m256i, __m256i, __mmask8 *, __mmask8 *);
VP2INTERSECTD void _mm512_2intersect_epi32(__m512i, __m512i, __mmask16 *, __mmask16 *);
VP2INTERSECTQ void _mm_2intersect_epi64(__m128i, __m128i, __mmask8 *, __mmask8 *);
VP2INTERSECTQ void _mm256_2intersect_epi64(__m256i, __m256i, __mmask8 *, __mmask8 *);
VP2INTERSECTQ void _mm512_2intersect_epi64(__m512i, __m512i, __mmask8 *, __mmask8 *);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.

VPCLMULQDQ — Carry-Less Multiplication Quadword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.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.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.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.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)	imm8 (r)
B	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

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 2-7. 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 2-8. 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[MAXVL-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[MAXVL-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[MAXVL-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.128.66.0F38.W0 50 /r VPDPBUSD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 4 pairs of signed bytes in xmm3/m128/m32bcst with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result in xmm1 under writemask k1.
EVEX.256.66.0F38.W0 50 /r VPDPBUSD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 4 pairs of signed bytes in ymm3/m256/m32bcst with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result in ymm1 under writemask k1.
EVEX.512.66.0F38.W0 50 /r VPDPBUSD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512_VNNI	Multiply groups of 4 pairs of signed bytes in zmm3/m512/m32bcst with corresponding unsigned bytes of zmm2, summing those products and adding them to doubleword result 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 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.128.66.0F38.W0 51 /r VPDPBUSDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 4 pairs signed bytes in xmm3/m128/m32bcst with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result, with signed saturation in xmm1, under writemask k1.
EVEX.256.66.0F38.W0 51 /r VPDPBUSDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 4 pairs signed bytes in ymm3/m256/m32bcst with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result, with signed saturation in ymm1, under writemask k1.
EVEX.512.66.0F38.W0 51 /r VPDPBUSDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512_VNNI	Multiply groups of 4 pairs signed bytes in zmm3/m512/m32bcst with corresponding unsigned bytes of zmm2, summing those products and adding them to doubleword result, with signed saturation 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 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.128.66.0F38.W0 52 /r VPDPWSSD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 2 pairs signed words in xmm3/m128/m32bcst with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, under writemask k1.
EVEX.256.66.0F38.W0 52 /r VPDPWSSD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 2 pairs signed words in ymm3/m256/m32bcst with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, under writemask k1.
EVEX.512.66.0F38.W0 52 /r VPDPWSSD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512_VNNI	Multiply groups of 2 pairs signed words in zmm3/m512/m32bcst with corresponding signed words of zmm2, summing those products and adding them to doubleword result 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 Signed Word Integers with Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 53 /r VPDPWSSDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 2 pairs of signed words in xmm3/m128/m32bcst with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, with signed saturation, under writemask k1.
EVEX.256.66.0F38.W0 53 /r VPDPWSSDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 2 pairs of signed words in ymm3/m256/m32bcst with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, with signed saturation, under writemask k1.
EVEX.512.66.0F38.W0 53 /r VPDPWSSDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512_VNNI	Multiply groups of 2 pairs of signed words in zmm3/m512/m32bcst with corresponding signed words of zmm2, summing those products and adding them to doubleword result in zmm1, with signed saturation, 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.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.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.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.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.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.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.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.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.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)	imm8 (r)
B	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	imm8 (r)

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.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.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.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.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.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.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.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.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.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.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.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.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.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.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.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.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.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.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)	imm8 (r)
B	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	imm8 (r)

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.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.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.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.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.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.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.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.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.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.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.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.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);

WBNOINVD—Write Back and Do Not Invalidate Cache

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 09 WBNOINVD	A	V/V	WBNOINVD	Write back and do not flush internal caches; initiate writing-back without flushing of external caches.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	NA	NA	NA	NA

Description

The WBNOINVD instruction writes back all modified cache lines in the processor's internal cache to main memory but does not invalidate (flush) the internal caches.

After executing this instruction, the processor does not wait for the external caches to complete their write-back operation before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back signal. The amount of time or cycles for WBNOINVD to complete will vary due to size and other factors of different cache hierarchies. As a consequence, the use of the WBNOINVD instruction can have an impact on logical processor interrupt/event response time.

The WBNOINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see "Serializing Instructions" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).

In situations where cache coherency with main memory is not a concern, software can use the INVD instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

The WBNOINVD instruction is implementation dependent, and its function may be implemented differently on future Intel 64 and IA-32 processors.

Operation

```
WriteBack(InternalCaches);
Continue; (* Continue execution *)
```

Intel C/C++ Compiler Intrinsic Equivalent

WBNOINVD void _wbnoinvd(void);

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	WBNOINVD cannot be executed at the virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

CHAPTER 3
ENQUEUE STORES AND PROCESS ADDRESS SPACE IDENTIFIERS (PASIDS)

Chapter 2 described the ENQCMD and ENQCMDS instructions. These instructions perform **enqueue stores**, which write command data to special device registers called **enqueue registers**.

Bits 19:0 of the 64-byte command data written by an enqueue store conveys the process address space identifier (PASID) associated with the command. Software can use PASIDs to identify individual software threads. Devices supporting enqueue registers may use these PASIDs in responding to commands submitted through those registers.

As explained in Chapter 2, an execution of ENQCMD formats the command data with the PASID specified in bits 19:0 of the IA32_PASID MSR. It is expected that system software will configure that MSR to contain the PASID associated with the software thread that is executing.

ENQCMDS can be executed only by system software operating with CPL > 0. It is the responsibility of system software executing ENQCMDS to configure the command data with the appropriate PASID.

Section 3.1 provides details of the IA32_PASID MSR. Section 3.2 describes how the XSAVE feature set supports that MSR. Section 3.3 presents PASID virtualization, a virtualization feature that allows a virtual-machine monitor to control the PASID values produced by enqueue stores executed by software in a virtual machine.

3.1 THE IA32_PASID MSR

This section describes the IA32_PASID MSR used by the ENQCMD instruction. The MSR can be read and written with the RDMSR and WRMSR instructions, using MSR index D93H. The MSR has format given in Table 3-1.

Table 3-1. IA32_PASID MSR

Bit Offset	Description
19:0	Process address space identifier (PASID). Specifies the PASID of the currently running software thread.
30:20	Reserved
31	Valid. Execution of ENQCMD causes a #GP if this bit is clear.
63:32	Reserved

An execution of WRMSR causes a general-protection exception (#GP) in response to an attempt to set any bit in the ranges 30:20 or 63:32. Executions of RDMSR always return zero for those bits.

Because system software may associate a PASID with a software thread, it may choose to update the IA32_PASID MSR on context switches. To facilitate such a usage, the XSAVE feature set is extended to manage the IA32_PASID MSR. These extensions are detailed in Section 3.2.

3.2 THE PASID STATE COMPONENT FOR THE XSAVE FEATURE SET

As noted in Section 3.1, system software may choose to update the IA32_PASID MSR on context switches. This usage is supported by extensions to the XSAVE feature set.

The XSAVE feature set supports the saving and restoring of state components. These state components are organized using state-component bitmaps (each bit in such a bitmap corresponds to a state component).

A new state component is introduced called **PASID state**. PASID state comprises the IA32_PASID MSR. It is defined to be state component 10, so PASID state is associated with bit 10 in state component bitmaps. It is a

supervisor state component, meaning that it can be managed only by the XSAVES and XRSTORS instructions. System software can enable those instructions to manage PASID state by setting bit 10 in the IA32_XSS MSR.

Processor support for this management of PASID state is enumerated by the CPUID instruction as follows:

- CPUID function 0DH, sub-function 1, enumerates in EDX:ECX a bitmap of the supervisor state components. ECX[10] will be enumerated as 1 to indicate that PASID state is supported.
- If PASID state is supported, CPUID function 0DH, sub-function 10 enumerates details for state component as follows:
  - EAX enumerates 8 as the size (in bytes) required for PASID state. (The state component comprises only the one MSR.)
  - EBX enumerates value 0, as is the case for supervisor state components.
  - ECX[0] enumerates 1, indicating that PASID state is a supervisor state component.
  - ECX[1] enumerates 0, indicating that state component 10 is located immediately following the preceding state component when the compacted format of the extended region of an XSAVE area is used.
  - ECX[31:2] and EDX enumerate 0, as is the case for all state components.

Like WRMSR, XRSTORS causes a general-protection exception (#GP) in response to an attempt to set any bit in the IA32_PASID MSR in the ranges 30:20 or 63:32. Like RDMSR, XSAVES always saves zero for those bits.

3.3 PASID TRANSLATION

As noted earlier, an operating system (OS) may use PASIDs to identify individual software threads that are allowed to access devices supporting enqueue registers.

Intel® Scalable I/O Virtualization (Scalable IOV) defines an approach to hardware-assisted I/O virtualization, extending it to support seamless addition of resources and dynamic provisioning of containers.[1] With Scalable IOV, a virtual-machine monitor (VMM) needs to control the PASIDs that are used by different virtual machines just as the guest OS controls the PASIDs used by software threads.

To allow a VMM to control the PASIDs used by enqueue stores while still allowing efficient use by a guest OS, a new virtualization feature is introduced, called **PASID translation**. PASID translation, if enabled, applies to any enqueue store performed by software in a virtual machine: the 20-bit PASID value specified by the guest operating system (**guest PASID**) for ENQCMD or ENQCMDS is translated into a 20-bit value (**host PASID**) that is used in the resulting enqueue store.

3.3.1 PASID Translation Structures

PASID translation is implemented by two hierarchies of data structures (**PASID-translation hierarchies**) configured by a VMM. Guest PASIDs 00000H to 7FFFFH are translated through the low PASID-translation hierarchy, while guest PASIDs 80000 to FFFFFH are translated through the high PASID-translation hierarchy.

Each PASID-translation hierarchy includes a 4-KByte **PASID directory**. A PASID directory comprises 512 8-byte entries, each of which has the following format:

- Bit 0 is the entry's present bit. The entry is used only if this bit is 1.
- Bits 11:1 are reserved and must be 0.
- Bits M–1:12 specify the 4-KByte aligned address of a PASID table (see below), where M is the physical-address width supported by the processor.
- Bits 63:M are reserved and must be 0.

A PASID-translation hierarchy also includes up to 512 4-KByte **PASID tables**; these are referenced by PASID directory entries (see above). A PASID table comprises 1024 4-byte entries, each of which has the following format:

- Bits 19:0 are the host PASID specified by the entry.

1. See the *Intel® Scalable I/O Virtualization Technical Specification* for more details.

- Bits 30:20 are reserved and must be 0.
- Bits 31 is the entry's valid bit. The entry is used only if this bit is 1.

Section 3.3.2 explains how the PASID-translation hierarchies are used to translate the PASIDs used for enqueue stores.

3.3.2 The PASID Translation Process

Each execution of ENQCMD or ENQCMDS results in an enqueue store with a PASID value. (ENQCMD obtains the PASID from the IA32_PASID MSR; ENQCMDS obtains it from the instruction's source operand.) When PASID translation is enabled, this PASID value is interpreted as a guest PASID. The guest PASID is converted to a host PASID; the enqueue store uses the host PASID for bits 19:0 of the command data that it writes.

The PASID translation process is illustrated in Figure 3-1.

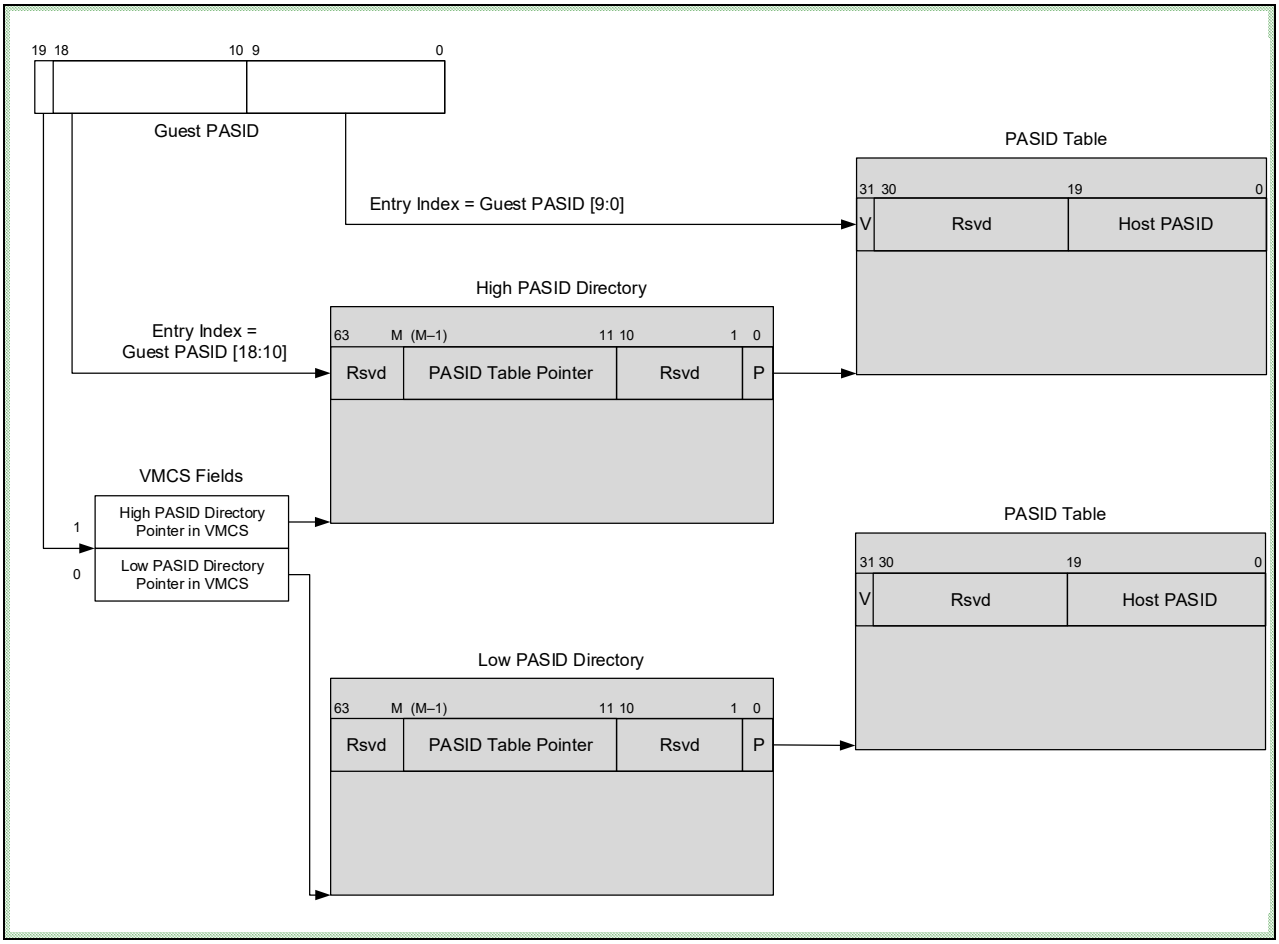

Figure 3-1. PASID Translation Process

The process operates as follows:

- If bit 19 of guest PASID is clear, the low PASID directory is used; otherwise, the high PASID directory is used.
- Bits 18:10 of the guest PASID select an entry from the PASID directory. A VM exit occurs if the entry's valid bit is clear or if any reserved bit is set. Otherwise, bits M:0 of the entry (with bit 0 cleared) contain the physical address of a PASID table, where M is the physical-address width supported by the processor..
- Bits 9:0 of the guest PASID select an entry from the PASID table. A VM exit occurs if the entry's present bit is clear or if any reserved bit is set. Otherwise, bits 19:0 of the entry are the host PASID.

An execution of ENQCMD or ENQCMDS performs PASID translation only after checking for conditions that may result in general-protection exception (the check of IA32_PASID.Valid for ENQCMD; the check of CPL for ENQCMDS) and after loading the instruction's source operand from memory. PASID translation occurs before the actual enqueue store and thus before any faults or VM exits that it may cause (e.g., page faults or EPT violations).

3.3.3 VMX Support

A VMM enables PASID translation by setting secondary processor-based VM-execution control 21. A processor enumerates support for the 1-setting of this control in the normal way (by setting bit 53 of the IA32_VMX_PROCBASED_CTLS2 MSR). It is expected that any processor that supports the ENQCMD and ENQCMDS instructions will also support PASID virtualization and vice versa.

PASID translation uses two new 64-bit VM-execution control fields in the VMCS: the **low PASID directory address** and the **high PASID directory address**. These are the physical addresses of the low PASID directory and the high PASID directory, respectively. Software can access these new VMCS fields using the encoding pairs 00002038H/00002039H and 0000203AH/0000203BH, respectively.

If the “PASID translation” VM-execution control is 1, VM entry fails if either PASID directory address sets any bit in the ranges 11:0 or 63:M, where M is the physical-address width supported by the processor.

Section 3.3.2 identified situations that may cause a VM exit during PASID translation. Such a VM exit uses basic exit reason 72 (for ENQCMD PASID translation failure) or 73 (ENQCMDS PASID translation failure). The exit qualification is determined as follows:

- For ENQCMD, it is IA32_PASID & 7FFFFH (bits 63:20 are cleared).
- For ENQCMD, it is SRC & FFFFFFFFH, where SRC is the instruction’s source operand (only bits 31:0 may be set).

CHAPTER 4
HARDWARE FEEDBACK INTERFACE ISA EXTENSIONS

4.1 HARDWARE FEEDBACK INTERFACE

Hardware provides guidance to the Operating System (OS) scheduler to perform optimal workload scheduling through a hardware feedback interface structure in memory. This structure has a global header that is 16 byte in size. Following this global header, there is one 8 byte entry per logical processor in the socket. The structure is designed as follows.

Table 4-1. Hardware Feedback Interface Structure

Byte Offset	Size (Bytes)	Description
0	16	Global Header
16	8	LP0 Capability Values
24	8	LP1 Capability Values
…	…	…
16 + n*8	8	LPn Capability Values

The global header is structured as follows.

Table 4-2. Hardware Feedback Interface Global Header Structure

Byte Offset	Size (Bytes)	Field Name	Description
0	8	Timestamp	Timestamp of when the table was last updated by hardware. This is a timestamp in crystal clock units. Initialized by OS to 0.
8	1	Performance Capability Changed	If set to 1, indicates the performance capability field for one or more logical processors was updated in the table. Initialized by OS to 0.
9	1	Energy Efficiency Capability Changed	If set to 1, indicates the energy efficiency capability field for one or more logical processors was updated in the table. Initialized by OS to 0.
10	6	Reserved	Initialized by OS to 0.

The per logical processor scheduler feedback entry is structured as follows.

Table 4-3. Hardware Feedback Interface Logical Processor Entry Structure

Byte Offset	Size (Bytes)	Field Name	Description
0	1	Performance Capability	Performance capability is an 8-bit value (0 ... 255) specifying the relative performance level of a logical processor. Higher values indicate higher performance; the lowest performance level of 0 indicates a recommendation to the OS to not schedule any software threads on it for performance reasons. Initialized by OS to 0.
1	1	Energy Efficiency Capability	Energy Efficiency capability is an 8-bit value (0 ... 255) specifying the relative energy efficiency level of a logical processor. Higher values indicate higher energy efficiency; the lowest energy efficiency capability of 0 indicates a recommendation to the OS to not schedule any software threads on it for efficiency reasons. Initialized by OS to 0.
2	6	Reserved	Initialized by OS to 0.

4.1.1 Hardware Feedback Interface Pointer

The physical address of the hardware feedback interface structure is programmed by the OS into a package scoped MSR named IA32_HW_FEEDBACK_PTR. The MSR is structured as follows:

- Bits 63:MAXPHYADDR[1] - Reserved.
- Bits MAXPHYADDR-1:12 - ADDR. This is the physical address of the page frame of the first page of this structure.
- Bits 11:1 - Reserved.
- Bit 0 - Valid. When set to 1, indicates a valid pointer is programmed into the MSR.

The address of this MSR is 17D0H.

See Section 4.1.4 for details on how the OS detects the size of memory to allocate for this structure. This MSR is cleared on reset to its default value of 0. The MSR retains its state on INIT.

4.1.2 Hardware Feedback Interface Configuration

The operating system enables the hardware feedback interface using a package scoped MSR named IA32_HW_-FEEDBACK_CONFIG (address 17D1H).

The MSR is structured as follows:

- Bits 63:1 - Reserved.
- Bit 0 - Enable. When set to 1, enables the hardware feedback interface.

This MSR is cleared on reset to its default value of 0. The MSR retains its state on INIT.

When the Enable bit transitions from 1 to 0, hardware sets the IA32_PACKAGE_THERM_STATUS bit 26 to 1 to acknowledge disabling of the interface. The OS should wait for this bit to be set to 1 after disabling the interface before reclaiming the memory allocated for this structure. When this bit is set to 1, it is safe to reclaim the memory as it is guaranteed that there are no writes in progress to this structure by hardware.

SENTER clears the enable bit to 0 on all sockets.

4.1.3 Hardware Feedback Interface Notifications

The IA32_PACKAGE_THERM_STATUS MSR is extended with a new bit, hardware feedback interface structure change status (bit 26, R/WC0), to indicate that the hardware has updated the hardware feedback interface structure. This is a sticky bit and once set, indicates that the OS should read the structure to determine the change and adjust its scheduling decisions. Once set, the hardware will not generate any further updates to this structure until

1. MAXPHYADDR is reported in CPUID.80000008H:EAX[7:0].

the OS clears this bit by writing 0. The hardware guarantees that all writes to the hardware feedback interface structure are globally observed.

The OS can enable interrupt-based notifications when the structure is updated by hardware through a new enable bit, hardware feedback interrupt enable (bit 25, R/W), in the IA32_PACKAGE_THERM_INTERRUPT MSR. When this bit is set to 1, it enables the generation of an interrupt when the hardware feedback interface structure is updated by hardware.

4.1.4 Hardware Feedback Interface Enumeration

CPUID.06H.0H:EAX.HW_FEEDBACK[bit 19] enumerates support for this feature. When this bit is enumerated to 1, the following MSR (or bits in the MSR) are supported by the hardware:

- IA32_HW_FEEDBACK_PTR (address 17D0H)
- IA32_HW_FEEDBACK_CONFIG (address 17D1H)
- IA32_PACKAGE_THERM_STATUS bit 26
- IA32_PACKAGE_THERM_INTERRUPT bit 25

When CPUID.06H.0H:EAX.HW_FEEDBACK[bit 19] = 1, then CPUID.06H.0H:EDX reports the following:

- EDX[7:0] - Bitmap of supported hardware feedback interface capabilities.

  Bit 0: When set to 1, indicates support for performance capability reporting.

  Bit 1: When set to 1, indicates support for energy efficiency capability reporting.

  Bits 0 and 1 will always be set together. Other bits are reserved.
- EDX[11:8] - Enumerates the size of the hardware feedback interface structure in number of 4 KB pages using minus-one notation.
- EDX[31:16] - Index (starting at 0) of this logical processor’s row in the hardware feedback interface structure. Note that the index may be same for multiple logical processors on some parts. On some parts the indices may not be contiguous, i.e., there may be unused rows in the table.

INDEX

B

C

F

I

L

M

P

R

S

T

V

W

X