Intel® Architecture Instruction Set Extensions and Future Features

Programming Reference

December 2022

319433-047

Notices & Disclaimers

This document contains information on products in the design phase of development. The information here is subject to change without notice. Do not finalize a design with this information.

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

No product or component can be absolutely secure.

Your costs and results may vary.

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.

All product plans and roadmaps are subject to change without notice.

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

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

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not “commercial” names and not intended to function as trademarks.

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 1-2 "Recent Instruction Set Extensions / Features Introduction in Intel® 64 and IA-32 Processors" . • Added Section 1.4, "Detection of Future Instructions and Features". • 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 1-2 "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 1-2 with new features in future processors. • Updated Table 1-3 with support for AVX512_VP2INTERSECT. • Updated Table 1-5 with support for ENQCMD: Enqueue Stores. • Added ENQCMD/ENQCMDS and VP2INTERSECTD/VP2INTERSECTQ instructions, and updated CPUID accordingly. • Added new chapter: Chapter 4, Non-Write-Back Lock Disable Architecture.	May 2019

Revision	Description	Date
-038	• Removed instruction extensions/features from Table 1-2 "Recent Instruction Set Extensions / Features Introduction in Intel® 64 and IA-32 Processors" that are available in processors covered in the Intel® 64 and IA-32 Architectures Software Developer's Manual. This information can be found in Chapter 5 "Instruction Set Summary", of Volume 1. • In Section 1.7, "Detection of Future Instructions", removed instructions from Table 1-5 "Future Instructions" that are available in processors covered in the Intel® 64 and IA-32 Architectures Software Developer's Manual. • Removed instructions with the following CPUID feature flags: AVX512_VNNI, VAES, GFNI (AVX/AVX512), AVX512_VBMI2, VPCLMULQDQ, AVX512_BITALG; they now reside in the Intel® 64 and IA-32 Architectures Software Developer's Manual. • CPUID instruction updated with Hybrid information sub-leaf 1AH, SERIALIZE and TSXLDTRK support, updates to the L3 Cache Intel RDT Monitoring Capability Enumeration Sub-leaf, and updates to the Memory Bandwidth Allocation Enumeration Sub-leaf. • Replaced ← with := notation in operation sections of instructions. These changes are not marked with change bars. • Added the following instructions: SERIALIZE, XRESLDTRK, XSUSLDTRK. • Update to the VDPBF16PS instruction. • Updates to Chapter 4, "Hardware Feedback Interface ISA Extensions". • Added Chapter 5, "TSX Suspend Load Address Tracking". • Added Chapter 6, "Hypervisor-managed Linear Address Translation". • Added Chapter 7, "Architectural Last Branch Records (LBRs)". • Added Chapter 8, "Non-Write-Back Lock Disable Architecture". • Added Chapter 9, "Intel® Resource Director Technology Feature Updates".	March 2020
-039	• Updated Section 1.1 "About this Document" to reflect chapter changes in this release. • Added Section 1.2 "DisplayFamily and DisplayModel for Future Processors". • Updated Table 1-2 "Recent Instruction Set Extensions / Features Introduction in Intel® 64 and IA-32 Processors". • CPUID instruction updated. • Removed Chapter 4 "Hardware Feedback Interface". This information is now in the Intel® 64 and IA-32 Architectures Software Developer's Manual. • Updated Figure 5-1 "Example HLAT Software Usage". • Added Table 6-5 "Encodings for 64-Bit Guest-State Fields (0010_10xx_xxxx_xxxAb)" to Chapter 6. • Added Chapter 8 "Bus Lock and VM Notify".	June 2020
-040	• Updated Section 1.1 "About this Document" to reflect chapter changes in this release. • Updated Table 1-2 "Recent Instruction Set Extensions / Features Introduction in Intel® 64 and IA-32 Processors". • CPUID instruction updated. • Added notation updates to the beginning of Chapter 2. Updated ENQCMD and ENQCMDS instructions to use this notation. • Added Chapter 3, "Intel® AMX Instruction Set Reference, A-Z". • Minor updates to Chapter 6, "Hypervisor-managed Linear Address Translation".	June 2020

Revision	Description	Date
-041	• Updated Section 1.1 “About this Document” to reflect chapter changes in this release. • Updated Table 1-2 “Recent Instruction Set Extensions / Features Introduction in Intel® 64 and IA-32 Processors”. • CPUID instruction updated for enumeration of several new features. • PCONFIG instruction updated. • Added CLUI, HRESET, SENDUIPI, STUI, TESTUI, UIRET, VPDPBUSD, VPDPBUSDS, VPDPWSSD, and VPDPWSSDS instructions to Chapter 2. • Updated Figure 3-2, “The TMUL Unit”. • Update to pseudocode of TILELOADD/TILELOADDT1 instruction. • Addition to Section 6.2, “VMCS Changes”. • Update to Section 7.1.2.4, “Call-Stack Mode”. • Update to Section 9.1 “Bus Lock Debug Exception”. • Added Chapter 11, “User Interrupts”. • Added Chapter 12, “Performance Monitoring Updates”. • Added Chapter 13, “Enhanced Hardware Feedback Interface”.	October 2020
-042	• CPUID instruction updated. • Removed the following instructions: VCVTNE2PS2BF16, VCVTNEPS2BF16, VDPBF16PS, VP2INTERSECTD/VP2INTERSECTQ, and WBNOINVD. They can be found in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2C. • Updated bit positions in Section 6.12, “Changes to VMX Capability Reporting”. • Typo correction in Chapter 8, “Non-Write-Back Lock Disable Architecture”. • Several updates to Chapter 13, “Enhanced Hardware Feedback Interface (EHFI)”. • Added Chapter 14, “Linear Address Masking (LAM)”. • Added Chapter 15, “Error Codes for Processors Based on Sapphire Rapids Microarchitecture”.	December 2020
-043	• Updated CPUID instruction. • Typo correction in Table 8-2, “TEST_CTRL MSR”. • Typo corrections in Section 14.1, “Enumeration, Enabling, and Configuration”.	February 2021
-044	• Updated Table 1-2, “Recent Instruction Set Extensions / Features Introduction in Intel® 64 and IA-32 Processors”. • Updated CPUID instruction. • Updates to the ENQCMD and ENQCMDS instructions. • Removed the PCONFIG instruction; it can be found in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B. • Corrected typo in the VPDPBUSD instruction. • Updates to Table 3-1, “Intel® AMX Exception Classes ”. • Change in terminology updates in Chapter 7, “Architectural Last Branch Records (LBRs)”. • Updated Chapter 6 to introduce the official technology name: Intel® Virtualization Technology - Redirect Protection. • Added Chapter 16, “IPI Virtualization”.	May 2021

Revision	Description	Date
-045	• Chapter 1: Updated the CPUID instruction. • Chapter 2: Updated ENQCMD and ENQCMDS to remove statements that these instructions ignore unused bits; this is incorrect. Removed HRESET, SERIALIZE, VPDPBUSD, VPDPBUSDS, VPDPWSSD, and VPDPWSSDS instructions; these instructions can be found in the Intel 64 and IA-32 Architectures Software Developer's Manual. Updates to SENDUIPI instruction operand encoding and 64-bit mode exceptions. Update to UIRET pseudocode. • Chapter 3: Updated Section 3.3., "Recommendations for System Software". • Removed Chapter 6, "Intel® Virtualization Technology: Redirect Protection"; this information can be found in the Intel 64 and IA-32 Architectures Software Developer's Manual. • Removed Chapter 7, "Architectural Last Branch Records (LBRs)"; this information can be found in the Intel 64 and IA-32 Architectures Software Developer's Manual. • Removed Chapter 12, "Performance Monitoring Updates"; this information can be found in the Intel 64 and IA-32 Architectures Software Developer's Manual. • Removed Chapter 13, "Enhanced Hardware Feedback Interface (EHFI)"; this information can be found in the Intel 64 and IA-32 Architectures Software Developer's Manual. • Updated Section 7.1.1, "Bus Lock VM Exit" to provide additional clarity and details. • Updated Chapter 8, "Intel® Resource Director Technology Feature Updates" to update MBA 3.0 information. • Update to Section 9.5.1, "User-Interrupt Notification Identification". • Minor updates to Chapter 10, "Linear Address Masking (LAM)", to provide additional clarity. • Corrected two typos in the current Table 11-1, "Intel IMC MC Error Codes for IA32_MCi_STATUS (i= 13-20)." • Added Chapter 13, "Asynchronous Enclave Exit Notify and the EDECCSSA User Leaf Function."	June 2022
-046	• Chapter 1: Updated Table 1-1, "CPUID Signature Values of DisplayFamily_DisplayModel." Updated Table 1-2, "Recent Instruction Set Extensions / Features Introduction in Intel® 64 and IA-32 Processors." Updated the CPUID instruction. • Chapter 2: Added the following instructions: AADD, AAND, AOR, AXOR, CMPccXADD, RDMSRLIST, VBCSTNEBF162PS, VBCSTNESH2PS, VCVTNEEBF162PS, VCVTNEEPH2PS, VCVTNEOBF162PS, VCVTNEOPH2PS, VCVTNEPS2BF16, VPDPB[SU,UU,SS]D[,S], VPMADD52HUQ, VPMADD52LUQ, WRMSRLIST, and WRMSRNS. • Chapter 3: Added section 3.4, "Operand Restrictions," and added the TDPFP16PS instruction. • Added Chapter 14, "Code Prefetch Instruction Updates." • Added Chapter 15, "Next Generation Performance Monitoring Unit (PMU)."	September 2022

Revision	Description	Date
-047	• Chapter 1: Updated Table 1-1, "CPUID Signature Values of DisplayFamily_DisplayModel." Updated Table 1-2, "Recent Instruction Set Extensions / Features Introduction in Intel® 64 and IA-32 Processors." Updated the CPUID instruction. • Chapter 3: Notes added and naming updates as necessary. • Removed the following chapters: Chapter 4, "Enqueue Stores and Process Address Space Identifiers (PASIDs)," Chapter 5, "Intel® TSX Suspend Load Address Tracking," Chapter 9, "User Interrupts," Chapter 11, "Error Codes for Processors Based on Sapphire Rapids Microarchitecture," and Chapter 12, "IPI Virtualization." This information can be found in the Intel 64 and IA-32 Architectures Software Developer's Manuals. • Removed the following instructions: CLUI, ENQCMD, ENQCMDS, LDTILECFG, SENDUIPI, STTILECFG, STUI, TDPBF16PS, TDPBSSD/TDPBSUD/TDPBUSD/TDPBUUD, TESTUI, TILELOADD/TILELOADDT1, TILERELEASE, TILESTORED, TILEZERO, UIRET, XRESLDTRK, and XSUSLDTRK. These instructions can be found in the Intel 64 and IA-32 Architectures Software Developer's Manuals. • Chapter 4: Updates to MSR name and description of bits. • Chapter 6: Updates to information, including naming changes and typo corrections as necessary. • Chapter 10: Update to the description of the Retire Latency field given in Section 10.3.1, "Timed Processor Event Based Sampling." • Added Chapter 11, "Linear Address Space Separation (LASS)." • Added Chapter 12, "Virtualization of the IA32_SPEC_CTRL MSR." • Added Chapter 13, "Remote Atomic Operations in Intel Architecture."	December 2022

REVISION HISTORY

CHAPTER 1
FUTURE INTEL® ARCHITECTURE INSTRUCTION EXTENSIONS AND FEATURES
1.1 About This Document. 1-1
1.2 DisplayFamily and DisplayModel for Future Processors 1-1
1.3 Instruction Set Extensions and Feature Introduction in Intel® 64 and IA-32 Processors 1-2
1.4 Detection of Future Instructions and Features 1-3
1.5 CPUID Instruction. 1-3
CPUID—CPU Identification. 1-3
1.6 Compressed Displacement (disp8*N) Support in EVEX 1-49
1.7 bfloat16 Floating-Point Format. 1-50

CHAPTER 2
INSTRUCTION SET REFERENCE, A-Z
2.1 Instruction Set Reference. 2-1
AADD—Atomically Add. 2-2
AAND—Atomically AND 2-4
AOR—Atomically OR 2-6
AXOR—Atomically XOR 2-8
CMPccXADD—Compare and Add if Condition is Met 2-10
RDMSRLIST—Read List of Model Specific Registers 2-15
VBCSTNEBF162PS—Load BF16 Element and Convert to FP32 Element With Broadcast 2-18
VBCSTNESH2PS—Load FP16 Element and Convert to FP32 Element with Broadcast. 2-19
VCVTNEEBF162PS—Convert Even Elements of Packed BF16 Values to FP32 Values 2-20
VCVTNEEPH2PS—Convert Even Elements of Packed FP16 Values to FP32 Values 2-21
VCVTNEOBF162PS—Convert Odd Elements of Packed BF16 Values to FP32 Values 2-22
VCVTNEOPH2PS—Convert Odd Elements of Packed FP16 Values to FP32 Values. 2-23
VCVTNEPS2BF16—Convert Packed Single-Precision Floating-Point Values to BF16 Values 2-24
VPDPB[SU,UU,SS]D[,S]—Multiply and Add Unsigned and Signed Bytes With and Without Saturation 2-26
VPMADD52HUQ—Packed Multiply of Unsigned 52-Bit Integers and Add the High 52-Bit Products to Qword Accumulators 2-29
VPMADD52LUQ—Packed Multiply of Unsigned 52-Bit Integers and Add the Low 52-Bit Products to Qword Accumulators 2-30
WRMSRLIST—Write List of Model Specific Registers 2-31
WRMSRNS—Non-Serializing Write to Model Specific Register. 2-34

CHAPTER 3
INTEL® AMX INSTRUCTION SET REFERENCE, A-Z
3.1 Introduction. 3-1
3.1.1 Tile Architecture Details 3-3
3.1.2 TMUL Architecture Details 3-4
3.1.3 Handling of Tile Row and Column Limits 3-5
3.1.4 Exceptions and Interrupts. 3-5
3.2 Operand Restrictions 3-5
3.3 Implementation Parameters. 3-5
3.4 Helper Functions 3-6
3.5 Notation 3-7
3.6 Exception Classes 3-7
3.7 Instruction Set Reference. 3-9
TDPFP16PS—Dot Product of FP16 Tiles Accumulated into Packed Single Precision Tile 3-10

CHAPTER 4
NON-WRITE-BACK LOCK DISABLE ARCHITECTURE
4.1 Enumeration 4-1
4.2 Enabling 4-1
4.3 Interaction with Intel® Software Guard Extensions (Intel® SGX). 4-2

4.4 Interaction with VMX Architecture . . . 4-2
4.5 Expected Software Behavior . . . 4-2
4.6 Bus Locks . . . 4-3

CHAPTER 5
BUS LOCK AND VM NOTIFY

5.1 Bus Lock Debug Exception . . . 5-1
5.1.1 Bus Lock VM Exit . . . 5-1
5.2 Notify VM Exit . . . 5-1

CHAPTER 6
INTEL® RESOURCE DIRECTOR TECHNOLOGY FEATURE UPDATES

6.1 Intel® RDT Feature Changes . . . 6-1
6.1.1 Intel® RDT on the 3rd generation Intel® Xeon® Scalable Processor Family . . . 6-1
6.1.2 Intel® RDT on Intel Atom® Processors, Including the P5000 Series . . . 6-1
6.1.3 Intel® RDT in Future Processors Based on Sapphire Rapids Server Microarchitecture . . . 6-1
6.1.4 Intel® RDT in Processors Based on Emerald Rapids Server Microarchitecture . . . 6-2
6.1.5 Future Intel® RDT . . . 6-2
6.2 Enumerable Memory Bandwidth Monitoring Counter Width . . . 6-2
6.2.1 Memory Bandwidth Monitoring (MBM) Enabling . . . 6-2
6.2.2 Augmented MBM Enumeration and MSR Interfaces for Extensible Counter Width . . . 6-2
6.3 Second Generation Memory Bandwidth Allocation . . . 6-3
6.3.1 Second Generation MBA Advantages . . . 6-3
6.3.2 Second Generation MBA Software-Visible Changes . . . 6-4
6.4 Third Generation Memory Bandwidth Allocation . . . 6-5
6.4.1 Third Generation MBA Hardware Changes . . . 6-5
6.4.2 Third Generation MBA Software-Visible Changes . . . 6-5
6.5 Future MBA Enhancements . . . 6-5

CHAPTER 7
LINEAR ADDRESS MASKING (LAM)

7.1 Enumeration, Enabling, and Configuration . . . 7-1
7.2 Treatment of Data Accesses with LAM Active for User Pointers . . . 7-1
7.3 Treatment of Data Accesses with LAM Active for Supervisor Pointers . . . 7-3
7.4 Canonicality Checking for Data Addresses Written to Control Registers and MSRs . . . 7-4
7.5 Paging Interactions . . . 7-4
7.6 VMX Interactions . . . 7-4
7.6.1 Guest Linear Address . . . 7-4
7.6.2 VM-Entry Checking of Values of CR3 and CR4 . . . 7-5
7.6.3 CR3-Target Values . . . 7-5
7.6.4 Hypervisor-Managed Linear Address Translation (HLAT) . . . 7-5
7.7 Debug and Tracing Interactions . . . 7-5
7.7.1 Debug Registers . . . 7-5
7.7.2 Intel® Processor Trace . . . 7-5
7.8 Intel® SGX Interactions . . . 7-5
7.9 System Management Mode (SMM) Interactions . . . 7-6

CHAPTER 8
ASYNCHRONOUS ENCLAVE EXIT NOTIFY AND THE EDECCSSA USER LEAF FUNCTION

8.1 Introduction . . . 8-1
8.2 Enumeration and Enabling . . . 8-2
8.3 Changes to Enclave Data Structures . . . 8-2
8.3.1 TCS.FLAGS Changes . . . 8-2
8.3.2 SSA.GPRSGX Changes . . . 8-2
8.3.3 ATTRIBUTES Changes . . . 8-2
8.4 Changes to Intel® SGX User Leaf Functions . . . 8-2
8.5 New Intel® SGX User Leaf Function: EDECCSSA . . . 8-3
EDECCSSA—Decrements TCS.CSSA . . . 8-3

8.6 Implications for Enclave Code Debug and Profiling 8-6
8.7 Interaction with Intel® CET 8-6
8.8 Changes to Intel® SGX User Leaf Function Operation 8-6
8.8.1 Changes to EENTER Operation 8-7
8.8.2 Changes to ERESUME Operation 8-14

CHAPTER 9
CODE PREFETCH INSTRUCTION UPDATES

PREFETCHh—Prefetch Data or Code Into Caches 9-1

CHAPTER 10
NEXT GENERATION PERFORMANCE MONITORING UNIT (PMU)

10.1 New Enumeration Architecture 10-1
10.1.1 CPUID Sub-Leafing 10-1
10.1.2 Reporting of Hybrid Resources 10-2
10.1.3 General-Purpose Counters Bitmap 10-2
10.1.4 Fixed-Function Counters Hybrid Bitmap 10-2
10.1.5 Architectural Performance Monitoring Events Bitmap 10-2
10.1.6 Non-Architectural Performance Capabilities 10-2
10.2 New Architectural Events 10-3
10.2.1 Topdown Microarchitecture Analysis Level 1 10-3
10.2.1.1 Topdown Backend Bound-Event Select A4H, Umask 02H 10-3
10.2.1.2 Topdown Bad Speculation-Event Select 73H, Umask 00H 10-4
10.2.1.3 Topdown Frontend Bound-Event Select 9CH, Umask 01H 10-4
10.2.1.4 Topdown Retiring-Event Select C2H, Umask 02H 10-4
10.3 Processor Event Based Sampling (PEBS) Enhancements 10-4
10.3.1 Timed Processor Event Based Sampling 10-4

CHAPTER 11
LINEAR ADDRESS SPACE SEPARATION (LASS)

11.1 Introduction 11-1
11.2 Enumeration and Enabling 11-1
11.3 Operation of Linear-Address Space Separation 11-1
11.3.1 Data Accesses 11-2
11.3.2 Instruction Fetches 11-2

CHAPTER 12
VIRTUALIZATION OF THE IA32_SPEC_CTRL MSR

12.1 Introduction 12-1
12.2 VMCS Changes 12-1
12.2.1 New VMX Control 12-1
12.2.2 New VMCS Fields 12-1
12.3 Changes to VMX Non-Root Operation 12-1

CHAPTER 13
REMOTE ATOMIC OPERATIONS IN INTEL ARCHITECTURE

13.1 Introduction 13-1
13.2 Instructions 13-1
13.3 Alignment Requirements 13-1
13.4 Memory Ordering 13-2
13.5 Memory Type 13-2
13.6 Write Combining Behavior 13-2
13.7 Performance Expectations 13-2
13.7.1 Interaction Between RAO and Other Accesses 13-3
13.7.2 Updates of Contended Data 13-3
13.7.3 Updates of Uncontended Data 13-3
13.8 Examples 13-4

13.8.1 Histogram 13-4
13.8.2 Interrupt/Event Handler 13-4

TABLES

PAGE

1-1 CPUID Signature Values of DisplayFamily_DisplayModel. 1-1
1-2 Recent Instruction Set Extensions / Features Introduction in Intel® 64 and IA-32 Processors. 1-2
1-3 Information Returned by CPUID Instruction 1-4
1-4 Processor Type Field. 1-28
1-5 Feature Information Returned in the ECX Register 1-30
1-6 More on Feature Information Returned in the EDX Register. 1-32
1-7 Encoding of Cache and TLB Descriptors 1-34
1-8 Processor Brand String Returned with Pentium 4 Processor 1-40
1-9 Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings 1-42
1-10 Compressed Displacement (DISP8*N) Affected by Embedded Broadcast 1-49
1-11 EVEX DISP8*N for Instructions Not Affected by Embedded Broadcast 1-49
2-1 Type 14 Class Exception Conditions 2-14
3-1 Intel® AMX Treatment of Denormal Inputs and Outputs 3-5
3-2 Intel® AMX Exception Classes. 3-8
4-1 IA32_CORE_CAPABILITIES MSR. 4-1
4-2 MEMORY_CTRL MSR 4-2
4-3 Bus Locks from Non-WB Operation 4-3
6-1 MBA_CFG MSR Definition. 6-5
8-1 Base Concurrency Restrictions of EDECCSSA 8-3
8-2 Additional Concurrency Restrictions of EDECCSSA 8-3
10-2 New Architectural Performance Monitoring Events 10-3
10-1 IA32_PERF_CAPABILITIES Hybrid Enumeration. 10-3
10-3 Basic Info Group 10-4
13-1 RAO Instructions 13-1

FIGURES

PAGE

Figure 1-1. Version Information Returned by CPUID in EAX. 1-28
Figure 1-2. Feature Information Returned in the ECX Register 1-30
Figure 1-3. Feature Information Returned in the EDX Register 1-32
Figure 1-4. Determination of Support for the Processor Brand String 1-40
Figure 1-5. Algorithm for Extracting Maximum Processor Frequency 1-41
Figure 1-6. Comparison of BF16 to FP16 and FP32 1-50
Figure 3-1. Intel® AMX Architecture 3-2
Figure 3-2. The TMUL Unit 3-3
Figure 3-3. Matrix Multiply C+= A*B. 3-4
Figure 6-1. Second Generation MBA, Including a Fast-Responding Hardware Controller 6-4
Figure 7-1. Canonicality Check When LAM48 is Enabled for User Pointers 7-2
Figure 7-2. Canonicality Check When LAM57 is Enabled for User Pointers with 5-Level Paging 7-2
Figure 7-3. Canonicality Check When LAM57 is Enabled for User Pointers with 4-Level Paging 7-3
Figure 7-4. Canonicality Check When LAM57 is Enabled for Supervisor Pointers with 5-Level Paging 7-3
Figure 7-5. Canonicality Check When LAM48 is Enabled for Supervisor Pointers with 4-Level Paging 7-4

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® Advanced Vector Extensions (Intel® AVX) and Intel® Advanced Vector Extensions 2 (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*.* 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 Intel AVX and Intel 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 an instruction set reference, providing details on new instructions.

Chapter 3 describes the Intel® Advanced Matrix Extensions (Intel® AMX).

Chapter 4 describes non-write-back lock disable architecture.

Chapter 5 describes bus lock and VM notify features.

Chapter 6 describes Intel® Resource Director Technology feature updates.

Chapter 7 describes Linear Address Masking (LAM).

Chapter 8 describes Asynchronous Enclave Exit Notify, an extension to Intel® SGX; and EDECCSSA, a new Intel SGX user leaf function.

Chapter 9 describes updates to the code prefetch instructions available in future processors.

Chapter 10 describes the next generation Performance Monitoring Unit enhancements available in future processors.

Chapter 11 describes Linear Address Space Separation (LASS).

Chapter 12 describes the virtualization of the IA32_SPEC_CTRL MSR.

Chapter 13 describes Remote Atomic Operations (RAO) in Intel architecture.

1.2 DISPLAYFAMILY AND DISPLAYMODEL FOR FUTURE PROCESSORS

Table 1-1 lists the signature values of DisplayFamily and DisplayModel for future processor families discussed in this document.

Table 1-1. CPUID Signature Values of DisplayFamily_DisplayModel

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_8FH	Future processors based on Sapphire Rapids Server microarchitecture
06_AAH, 06_ACH	Future processors based on Meteor Lake microarchitecture
06_B6H	Future processors based on Grand Ridge microarchitecture
06_B7H	Future processors based on Raptor Lake microarchitecture
06_ADH, 06_AEH	Future processors based on Granite Rapids microarchitecture

Table 1-1. CPUID Signature(Continued)Values of DisplayFamily_DisplayModel (Continued)

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_AFH	Future processors based on Sierra Forest microarchitecture
06_CFH	Future processors based on Emerald Rapids Server microarchitecture

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

Recent instruction set extensions and features are listed in Table 1-2. Within these groups, most instructions and features are collected into functional subgroups.

Table 1-2. Recent Instruction Set Extensions / Features Introduction in Intel® 64 and IA-32 Processors[1]

Instruction Set Architecture / Feature	Introduction
Direct stores: MOVDIRI, MOVDIR64B	Tremont, Tiger Lake, Sapphire Rapids
AVX512_BF16	Cooper Lake, Sapphire Rapids
CET: Control-flow Enforcement Technology	Tiger Lake, Sapphire Rapids
AVX512_VP2INTERSECT	Tiger Lake (not currently supported in any other processors)
Enqueue Stores: ENQCMD and ENQCMDS	Sapphire Rapids
CLDEMOTE	Tremont, Alder Lake, Sapphire Rapids
PTWRITE	Goldmont Plus, Alder Lake, Sapphire Rapids
User Wait: TPAUSE, UMONITOR, UMWAIT	Tremont, Alder Lake, Sapphire Rapids
Architectural LBRs	Alder Lake, Sapphire Rapids
HLAT	Alder Lake, Sapphire Rapids
SERIALIZE	Alder Lake, Sapphire Rapids
Intel® TSX Suspend Load Address Tracking (TSXLDTRK)	Sapphire Rapids
Intel® Advanced Matrix Extensions (Intel® AMX) Includes CPUID Leaf 1EH, “TMUL Information Main Leaf”, and CPUID bits AMX-BF16, AMX-TILE, and AMX-INT8.	Sapphire Rapids
AVX-VNNI	Alder Lake[2], Sapphire Rapids
User Interrupts (UINTR)	Sapphire Rapids
Intel® Trust Domain Extensions (Intel® TDX)[3]	Future Processors
Supervisor Memory Protection Keys (PKS)[4]	Sapphire Rapids
Linear Address Masking (LAM)	Future Processors
IPI Virtualization	Sapphire Rapids
RAO-INT	Grand Ridge
PREFETCHIT0/1	Granite Rapids
AMX-FP16	Granite Rapids
CMPCCXADD	Sierra Forest, Grand Ridge
AVX-IFMA	Sierra Forest, Grand Ridge
AVX-NE-CONVERT	Sierra Forest, Grand Ridge
AVX-VNNI-INT8	Sierra Forest, Grand Ridge
RDMSRLIST/WRMSRLIST	Sierra Forest, Grand Ridge

Table 1-2. Recent Instruction Set Extensions / Features Introduction in Intel® 64 and IA-32 Processors[1] (Continued)

Instruction Set Architecture / Feature	Introduction
WRMSRNS	Sierra Forest, Grand Ridge
Linear Address Space Separation (LASS)	Sierra Forest
Virtualization of the IA32_SPEC_CTRL MSR	Sapphire Rapids

NOTES:

1. Visit Intel Ark for Intel® product specifications, features and compatibility quick reference guide, and code name decoder.
2. Alder Lake Intel Hybrid Technology will not support Intel® AVX-512. ISA features such as Intel® AVX, AVX-VNNI, Intel® AVX2, and UMONITOR/UMWAIT/TPAUSE are supported.
3. Details on Intel® Trust Domain Extensions can be found here: https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html.
4. Details on Supervisor Memory Protection Keys (PKS) can be found in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

1.4 DETECTION OF FUTURE INSTRUCTIONS AND FEATURES

Future instructions and features are enumerated by a CPUID feature flag; details can be found in Table 1-3.

1.5 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 1-3 shows information returned, depending on the initial value loaded into the EAX register.

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. *)
```

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

CPUID.EAX = 0BH (* INVALID: Returns the same information as CPUID.EAX = 0AH. *)[1]
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 9, “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.

Table 1-3. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
	Basic CPUID Information	
0H	EAX	Maximum Input Value for Basic CPUID Information
	EBX	“Genu”
	ECX	“ntel”
	EDX	“inel”
01H	EAX	Version Information: Type, Family, Model, and Stepping ID (see Figure 1-1)
	EBX	Bits 7-0: Brand Index Bits 15-8: CLFLUSH line size (Value ∗ 8 = cache line size in bytes) Bits 23-16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31-24: Initial APIC ID**
	ECX	Feature Information (see Figure 1-2 and Table 1-5)
	EDX	Feature Information (see Figure 1-3 and Table 1-6) **NOTES:** * The nearest power-of-2 integer that is not smaller than EBX[23:16] is the maximum number of unique initial APIC IDs reserved for addressing different logical processors in a physical package. **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	Cache and TLB Information (see Table 1-7)
	EBX	Cache and TLB Information
	ECX	Cache and TLB Information
	EDX	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.)

[1] 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 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
		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 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.

Table 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	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
	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 22-21: Reserved. Bit 23: Intel® Thread Director supported if set. IA32_HW_FEEDBACK_CHAR and IA32_HW_FEEDBACK_THREAD_CONFIG MSRs are supported if set. Bit 24: IA32_THERM_INTERRUPT MSR bit 25 is supported if set. Bits 31-25: Reserved.

Table 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	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 the IA32_ENERGY_PERF_BIAS MSR (MSR address 1B0H). Bits 07 -04: Reserved = 0. Bits 15-08: Number of Intel® Thread Director classes supported by the processor. Information for that many classes is written into the Intel Thread Director Table by the hardware. Bits 31-16: Reserved = 0.
	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 11-08: Enumerates the size of the hardware feedback interface structure in number of 4 KB pages; add one to the return value to get the result. Bits 31-16: Index (starting at 0) of this logical processor's row in the hardware feedback interface structure. Note that on some parts the index may be same for multiple logical processors. On some parts the indices may not be contiguous, i.e., there may be unused rows in the hardware feedback interface structure. **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. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX.
	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. Supports Intel® Advanced Vector Extensions 2 (Intel® AVX2) if 1. Bit 06: FDP_EXCPTN_ONLY. x87 FPU Data Pointer updated only on x87 exceptions if 1. Bit 07: SMEP. Supports Supervisor Mode Execution Protection if 1. Bit 08: BMI2 Bit 09: Supports Enhanced REP MOVSB/STOSB if 1. Bit 10: INVPCID Bit 11: RTM Bit 12: RDT-M. Supports Intel® Resource Director Technology (Intel® RDT) Monitoring capability if 1. Bit 13: Deprecates FPU CS and FPU DS values if 1. Bit 14: Intel® Memory Protection Extensions Bit 15: RDT-A. Supports Intel® Resource Director Technology (Intel® RDT) Allocation capability if 1. Bit 16: AVX512F Bit 17: AVX512DQ Bit 18: RDSEED Bit 19: ADX

Table 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
		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
	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: CET_SS. Supports CET shadow stack features if 1. Processors that set this bit define bits 1:0 of the IA32_U_CET and IA32_S_CET MSRs. Enumerates support for the following MSRs: IA32_INTERRUPT_SPP_TABLE_ADDR, IA32_PL3_SSP, IA32_PL2_SSP, IA32_PL1_SSP, and IA32_PL0_SSP. Bit 08: GFNI Bit 09: VAES Bit 10: VPCLMULQDQ Bit 11: AVX512_VNNI Bit 12: AVX512_BITALG Bit 13: TME_EN. If 1, the following MSRs are supported: IA32_TME_CAPABILITY, IA32_TME_ACTIVATE, IA32_TME_EXCLUDE_MASK, and IA32_TME_EXCLUDE_BASE. Bit 14: AVX512_VPOPCNTDQ Bit 15: Reserved Bit 16: LA57. Supports 57-bit linear addresses and five-level paging if 1. 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. Bit 23: KL. Supports Key Locker if 1. Bit 24: BUS_LOCK_DETECT. If 1, indicates support for bus lock debug exceptions. 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: PKS. Supports protection keys for supervisor-mode pages if 1.
	EDX	Bit 00: Reserved. Bit 01: SGX-KEYS. If 1, Attestation Services for Intel® SGX is supported. Bit 02: AVX512_4VNNIW (Intel® Xeon Phi™ only.) Bit 03: AVX512_4FMAPS (Intel® Xeon Phi™ only.) Bit 04: Fast Short REP MOV Bit 05: UINTR. If 1, the processor supports user interrupts. Bits 07-06: Reserved Bit 08: AVX512_VP2INTERSECT

Table 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor
	Bit 09: SRBDS_CTRL. If 1, enumerates support for the IA32_MCU_OPT_CTRL MSR and indicates that its bit 0 (RNGDS_MITG_DIS) is also supported. Bit 10: MD_CLEAR supported. Bit 11: RTM_ALWAYS_ABORT. If set, any execution of XBEGIN immediately aborts and transitions to the specified fallback address. Bit 12: Reserved Bit 13: If 1, RTM_FORCE_ABORT supported. Processors that set this bit support the TSX_FORCE_ABORT MSR. They allow software to set TSX_FORCE_ABORT[0] (RTM_FORCE_ABORT). Bit 14: SERIALIZE Bit 15: Hybrid. If 1, the processor is identified as a hybrid part. If CPUID.0.MAXLEAF ≥ 1AH and CPUID.1A.EAX ≠ 0, then the Native Model ID Enumeration Leaf 1AH exists. Bit 16: TSXLDTRK. If 1, the processor supports Intel TSX suspend/resume of load address tracking. Bit 17: Reserved Bit 18: PCONFIG Bit 19: Architectural LBRs. If 1, indicates support for architectural LBRs. Bit 20: CET_IBT. Supports CET indirect branch tracking features if 1. Processors that set this bit define bits 5:2 and bits 63:10 of the IA32_U_CET and IA32_S_CET MSRs. Bit 21: Reserved Bit 22: AMX-BF16. If 1, the processor supports tile computational operations on bfloat16 numbers. Bit 23: AVX512_FP16 Bit 24: AMX-TILE. If 1, the processor supports tile architecture. Bit 25: AMX-INT8. If 1, the processor supports tile computational operations on 8-bit integers. 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: Enumerates support for L1D_FLUSH. Processors that set this bit support the IA32_FLUSH_CMD MSR. They allow software to set IA32_FLUSH_CMD[0] (L1D_FLUSH). Bit 29: Enumerates support for the IA32_ARCH_CAPABILITIES MSR. Bit 30: Enumerates support for the IA32_CORE_CAPABILITIES MSR. IA32_CORE_CAPABILITIES is an architectural MSR that enumerates model-specific features. In general, a bit being set in this MSR indicates that a model-specific feature is supported; software should consult CPUID family/model/stepping to determine the behavior of these enumerated features, as that behavior may differ on different processor models. Some bits in the MSR enumerate features with behavior that is consistent across processor models (and for which consultation of CPUID family/model/stepping is not necessary); such bits are identified explicitly in the documentation of 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).
	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.

Table 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	EAX	This field reports 0 if the sub-leaf index, *1*, is invalid. Bits 02-00: Reserved. Bit 03: RAO-INT. If 1, supports the RAO-INT instructions. Bit 04: AVX-VNNI. AVX (VEX-encoded) versions of the Vector Neural Network Instructions. Bit 05: AVX512_BF16. Vector Neural Network Instructions supporting bfloat16 inputs and conversion instructions from IEEE single precision. Bit 06: LASS. If 1, supports Linear Address Space Separation. Bit 07: CMPCCXADD. If 1, supports the CMPccXADD instruction. Bit 08: ArchPerfmonExt. If 1, supports ArchPerfmonExt. When set, indicates that the Architectural Performance Monitoring Extended Leaf (EAX = 23H) is valid. Bit 09: Reserved. Bit 10: If 1, supports fast zero-length MOVSB. Bit 11: If 1, supports fast short STOSB. Bit 12: If 1, supports fast short CMPSB, SCASB. Bits 18-13: Reserved. Bit 19: WRMSRNS. If 1, supports the WRMSRNS instruction. Bit 20: Reserved. Bit 21: AMX-FP16. If 1, the processor supports tile computational operations on FP16 numbers. Bit 22: HRESET. If 1, supports history reset and the IA32_HRESET_ENABLE MSR. When set, indicates that the Processor History Reset Leaf (EAX = 20H) is valid. Bit 23: AVX-IFMA. If 1, supports the AVX-IFMA instructions. Bits 25-24: Reserved. Bit 26: LAM. If 1, supports Linear Address Masking. Bit 27: MSRLIST. If 1, supports the RDMSRLIST and WRMSRLIST instructions and the IA32_BARRIER MSR. Bits 31-28: Reserved.
	EBX	This field reports 0 if the sub-leaf index, 1, is invalid; otherwise it is reserved. Bit 00: Enumerates the presence of the IA32_PPIN and IA32_PPIN_CTL MSRs. If 1, these MSRs are supported. Bits 31-01: Reserved.
	ECX	This field reports 0 if the sub-leaf index, *1*, is invalid; otherwise it is reserved.
	EDX	This field reports 0 if the sub-leaf index, *1*, is invalid. Bits 03-00: Reserved. Bit 04: AVX-VNNI-INT8. If 1, supports the AVX-VNNI-INT8 instructions. Bit 05: AVX-NE-CONVERT. If 1, supports the AVX-NE-CONVERT instructions. Bits 13-06: Reserved. Bit 14: PREFETCHI. If 1, supports the PREFETCHIT0/1 instructions. Bits 17-15: Reserved Bit 18: CET_SSS. If 1, indicates that an operating system can enable supervisor shadow stacks as long as it ensures that certain supervisor shadow-stack pushes will not cause page faults (see Section 17.2.3 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1). When emulating the CPUID instruction, a virtual-machine monitor should return this bit as 0 if those pushes can cause VM exits. Bits 31-19: Reserved.
	Structured Extended Feature Enumeration Sub-leaf (EAX = 07H, ECX = 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, *2*, is invalid; otherwise it is reserved.
	EBX	This field reports 0 if the sub-leaf index, *2*, is invalid; otherwise it is reserved.

Table 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	ECX	This field reports 0 if the sub-leaf index, *2*, is invalid; otherwise it is reserved.
	EDX	This field reports 0 if the sub-leaf index, 2, is invalid. Bit 00: PSFD. If 1, indicates bit 7 of the IA32_SPEC_CTRL MSR is supported. Bit 7 of this MSR disables Fast Store Forwarding Predictor without disabling Speculative Store Bypass. Bit 01: IPRED_CTRL. If 1, indicates bits 3 and 4 of the IA32_SPEC_CTRL MSR are supported. Bit 3 of this MSR enables IPRED_DIS control for CPL3. Bit 4 of this MSR enables IPRED_DIS control for CPL0/1/2. Bit 02: RRSBA_CTRL. If 1, indicates bits 5 and 6 of the IA32_SPEC_CTRL MSR are supported. Bit 5 of this MSR disables RRSBA behavior for CPL3. Bit 6 of this MSR disables RRSBA behavior for CPL0/1/2. Bit 03: DDPD_U. If 1, indicates bit 8 of the IA32_SPEC_CTRL MSR is supported. Bit 8 of this MSR disables Data Dependent Prefetcher. Bit 04: BHI_CTRL. If 1, indicates bit 10 of the IA32_SPEC_CTRL MSR is supported. Bit 10 of this MSR enables BHI_DIS_S behavior. Bit 05: MCDT_NO. Processors that enumerate this bit as 1 do not exhibit MXCSR Configuration Dependent Timing (MCDT) behavior and do not need to be mitigated to avoid data-dependent behavior for certain instructions. Bits 31-06: 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. Architectural event x is supported if EBX[x]=0 && EAX[31:24] > x.
	EBX	Bit 00: Core cycle event not available if 1 or if EAX[31:24] < 1. Bit 01: Instruction retired event not available if 1 or if EAX[31:24] < 2. Bit 02: Reference cycles event not available if 1 or if EAX[31:24] < 3. Bit 03: Last-level cache reference event not available if 1 or if EAX[31:24] < 4. Bit 04: Last-level cache misses event not available if 1 or if EAX[31:24] < 5. Bit 05: Branch instruction retired event not available if 1 or if EAX[31:24] < 6. Bit 06: Branch mispredict retired event not available if 1 or if EAX[31:24] < 7. Bit 07: Top-down slots event not available if 1 or if EAX[31:24] < 8. Bits 31-08: Reserved = 0.
	ECX	Bits 31-00: Supported fixed counters. If bit 'i' is set, it implies that Fixed Counter 'i' is supported. Software is recommended to use the following logic to check if a Fixed Counter is supported on a given processor: FxCtr[i]_is_supported := ECX[i] || (EDX[4:0] > i);

Table 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 04-00: Number of contiguous fixed-function performance counters starting from 0 (if Version ID > 1). Bits 12-05: Bit width of fixed-function performance counters (if Version ID > 1). Bits 14-13: Reserved = 0. Bit 15: AnyThread deprecation. Bits 31-16: Reserved = 0.
	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

Table 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	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: x87 state. Bit 01: SSE state. Bit 02: AVX state. Bits 04-03: MPX state Bit 07-05: AVX-512 state. Bit 08: Used for IA32_XSS. Bit 09: PKRU state. Bits 16-10: Used for IA32_XSS. Bit 17: TILECFG state. Bit 18: TILEDATA state. Bits 31-19: Reserved.
	EBX	Bits 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCR0. May be different than ECX if some features at the end of the XSAVE save area are not enabled.
	ECX	Bit 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e all the valid bit fields in XCR0.
	EDX	Bit 31-00: 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. Bit 04: Supports Extended Feature Disable (XFD) if set. Bits 31-05: Reserved.
	EBX	Bits 31-00: The size in bytes of the XSAVE area containing all states enabled by XCRO | IA32_XSS. **NOTES:** If EAX[3] is enumerated as 0 and EAX[1] is enumerated as 1, EBX enumerates the size of the XSAVE area containing all states enabled by XCRO. If EAX[1] and EAX[3] are both enumerated as 0, EBX enumerates zero.
	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. Bit 10: PASID state. Bit 11: CET user state. Bit 12: CET supervisor state. Bit 13: HDC state. Bit 14: UINTR state. Bits 15: LBR state (only for the architectural LBR feature). Bit 16: HWP state. Bits 18-17: Used for XCR0. Bits 31-19: Reserved.

Table 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	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). Bit 2 is set to indicate support for XFD faulting. Bits 31-03 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.
	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

Table 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	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	No bits set: 24-bit counters. Bits 07 - 00: Encode counter width offset from 24b: 0x0 = 24-bit counters. 0x1 = 25-bit counters. 0x25 = 61-bit counters. Bit 08: Indicates that bit 61 in IA32_QM_CTR MSR is an overflow bit. Bits 31 - 09: 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.
	EDX	Bits 15-00: Highest COS number supported for this ResID. Bits 31-16: Reserved.

Table 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	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	Bit 00: Per-thread MBA controls are supported. Bit 01: 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 10-07: Reserved. Bit 11: If 1, indicates Intel SGX supports ENCLU instruction leaf EDECCSSA. Bits 31-12: 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 1-3. 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. If EAX[3:0] 0010b, then this section has confidentiality protection only. 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.
	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.

Table 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	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_RTIT_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. Bit 06: If 1, indicates support for PSB and PMI preservation. Writes can set IA32_RTIT_CTL[56] (InjectPsbPmiOnEnable), enabling the processor to set IA32_RTIT_STATUS[7] (PendTopaPMI) and/or IA32_RTIT_STATUS[6] (PendPSB) in order to preserve ToPA PMIs and/or PSBs otherwise lost due to Intel PT disable. Writes can also set PendToPAPMI and PendPSB. Bit 07: If 1, writes can set IA32_RTIT_CTL[31] (EventEn), enabling Event Trace packet generation. Bit 08: If 1, writes can set IA32_RTIT_CTL[55] (DisTNT), disabling TNT packet generation. Bits 31-09: 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 1-3. 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 1-3. 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. 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. A sub-leaf index is also invalid if EDX[4:0] returns 0. 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. * Some unified TLBs will allow a single TLB entry to satisfy data read/write and instruction fetches. Others will require separate entries (e.g., one loaded on data read/write and another loaded on an instruction fetch). See the Intel® 64 and IA-32 Architectures Optimization Reference Manual for details of a particular product. ** 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. 00100b: Load Only TLB. Hit on loads; fills on both loads and stores. 00101b: Store Only TLB. Hit on stores; fill on stores. 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.

Table 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	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. A sub-leaf index is also invalid if EDX[4:0] returns 0. 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. * Some unified TLBs will allow a single TLB entry to satisfy data read/write and instruction fetches. Others will require separate entries (e.g., one loaded on data read/write and another loaded on an instruction fetch). See the Intel® 64 and IA-32 Architectures Optimization Reference Manual for details of a particular product. ** Add one to the return value to get the result.
	EAX	Bits 31-00: Reserved.
	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.
	Key Locker Leaf (EAX = 19H)	
19H	EAX	Bit 00: Key Locker restriction of CPL0-only supported. Bit 01: Key Locker restriction of no-encrypt supported. Bit 02: Key Locker restriction of no-decrypt supported. Bits 31-03: Reserved.
	EBX	Bit 00: AESKLE. If 1, the AES Key Locker instructions are fully enabled. Bit 01: Reserved. Bit 02: If 1, the AES wide Key Locker instructions are supported. Bit 03: Reserved. Bit 04: If 1, the platform supports the Key Locker MSRs and backing up the internal wrapping key. Bits 31-05: Reserved.
	ECX	Bit 00: If 1, the NoBackup parameter to LOADIWKEY is supported. Bit 01: If 1, KeySource encoding of 1 (randomization of the internal wrapping key) is supported. Bits 31- 02: Reserved.
	EDX	Reserved.

Table 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	Native Model ID Enumeration Leaf (EAX = 1AH, ECX = 0)	
1AH		**NOTES:** This leaf exists on all hybrid parts, however this leaf is not only available on hybrid parts. The following algorithm is used for detection of this leaf: If CPUID.0.MAXLEAF ≥ 1AH and CPUID.1A.EAX ≠ 0, then the leaf exists.
	EAX	Enumerates the native model ID and core type. Bits 31-24: Core type 10H: Reserved 20H: Intel Atom® 30H: Reserved 40H: Intel® Core™ Bits 23-0: Native model ID of the core. The core-type and native model ID can be used to uniquely identify the microarchitecture of the core. This native model ID is not unique across core types, and not related to the model ID reported in CPUID leaf 01H, and does not identify the SOC.
	EBX	Reserved.
	ECX	Reserved.
	EDX	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.
	Last Branch Records Information Leaf (EAX = 1CH, ECX = 0)	
1CH		**NOTES:** This leaf pertains to the architectural feature. For leaf 01CH, CPUID will ignore the ECX value.
	EAX	Bits 07 - 00: Supported LBR Depth Values. For each bit n set in this field, the IA32_LBR_DEPTH.DEPTH value 8*(n+1) is supported. Bits 29 - 08: Reserved. Bit 30: Deep C-state Reset. If set, indicates that LBRs may be cleared on an MWAIT that requests a C-state numerically greater than C1. Bit 31: IP Values Contain LIP. If set, LBR IP values contain LIP. If clear, IP values contain Effective IP.

Table 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	EBX	Bit 00: CPL Filtering Supported. If set, the processor supports setting IA32_LBR_CTL[2:1] to non-zero value. Bit 01: Branch Filtering Supported. If set, the processor supports setting IA32_LBR_CTL[22:16] to non-zero value. Bit 02: Call-stack Mode Supported. If set, the processor supports setting IA32_LBR_CTL[3] to 1. Bits 31 - 03: Reserved.
	ECX	Bit 00: Mispredict Bit Supported. IA32_LBR_x_INFO[63] holds indication of branch misprediction (MISPRED). Bit 01: Timed LBRs Supported. IA32_LBR_x_INFO[15:0] holds CPU cycles since last LBR entry (CYC_CNT), and IA32_LBR_x_INFO[60] holds an indication of whether the value held there is valid (CYC_CNT_VALID). Bit 02: Branch Type Field Supported. IA32_LBR_INFO_x[59:56] holds indication of the recorded operation’s branch type (BR_TYPE). Bits 31 - 03: Reserved.
	EDX	Bits 31 - 00: Reserved.
	Tile Information Main Leaf (EAX = 1DH, ECX = 0)	
1DH		**NOTES:** For sub-leaves of 1DH, they are indexed by the palette id. Leaf 1DH sub-leaves 2 and above are reserved.
	EAX	Bits 31-00: max_palette. Highest numbered palette sub-leaf. Value = 1.
	EBX	Bits 31-00: Reserved = 0.
	ECX	Bits 31-00: Reserved = 0.
	EDX	Bits 31-00: Reserved = 0.
	Tile Palette 1 Sub-leaf (EAX = 1DH, ECX = 1)	
1DH	EAX	Bits 15-00: Palette 1 total_tile_bytes. Value = 8192. Bits 31-16: Palette 1 bytes_per_tile. Value = 1024.
	EBX	Bits 15-00: Palette 1 bytes_per_row. Value = 64. Bits 31-16: Palette 1 max_names (number of tile registers). Value = 8.
	ECX	Bits 15-00: Palette 1 max_rows. Value = 16. Bits 31-16: Reserved = 0.
	EDX	Bits 31-00: Reserved = 0.
	TMUL Information Main Leaf (EAX = 1EH, ECX = 0)	
1EH		**NOTE:** Leaf 1EH sub-leaf 1 and above are reserved.
	EAX	Bits 31-00: Reserved = 0.
	EBX	Bits 07-00: tmul_maxk (rows or columns). Value = 16. Bits 23-08: tmul_maxn (column bytes). Value = 64. Bits 31-24: Reserved = 0.
	ECX	Bits 31-00: Reserved = 0.
	EDX	Bits 31-00: Reserved = 0.

Table 1-3. 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.
	Processor History Reset Sub-leaf (EAX = 20H, ECX = 0)	
20H	EAX	Reports the maximum number of sub-leaves that are supported in leaf 20H.
	EBX	Indicates which bits may be set in the IA32_HRESET_ENABLE MSR to enable enhanced hardware feedback interface history. Bit 00: Indicates support for both HRESET's EAX[0] parameter, and IA32_HRESET_ENABLE[0] set by the OS to enable reset of EHFI history. Bits 31-01: Reserved for other history reset capabilities.
	ECX	Reserved.
	EDX	Reserved.

Table 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	Architectural Performance Monitoring Extended Leaf (Output depends on ECX input value)	
23H		**NOTES:** Leaf 23H main leaf (ECX = 0).
	EAX	Bits 31-00: Reports the valid sub-leaves that are supported in leaf 23H.
	EBX	Bits 31-00: Reserved.
	ECX	Bits 31-00: Reserved.
	EDX	Bits 31-00: Reserved.
	Architectural Performance Monitoring Extended Sub-Leaf (EAX = 23H, ECX = 1)	
23H	EAX	Bits 31-00: General counters bitmap. For each bit *n* set in this field, the processor supports general-purpose performance monitoring counter *n*.
	EBX	Bits 31-00: Fixed counters bitmap. For each bit *m* set in this field, the processor supports fixed-function performance monitoring counter *m*.
	ECX	Bits 31-00: Reserved.
	EDX	Bits 31-00: Reserved.
	Architectural Performance Monitoring Extended Sub-Leaf (EAX = 23H, ECX = 3)	
23H		Architectural Performance Monitoring Events Bitmap. For each bit *n* set in this field, the processor supports Architectural Performance Monitoring Event of index *n*.
	EAX	Bit 00: Core cycles. Bit 01: Instructions retired. Bit 02: Reference cycles. Bit 03: Last level cache references. Bit 04: Last level cache misses. Bit 05: Branch instructions retired. Bit 06: Branch mispredicts retired. Bit 07: Topdown slots. Bit 08: Topdown backend bound. Bit 09: Topdown bad speculation. Bit 10: Topdown frontend bound. Bit 11: Topdown retiring. Bits 31-12: Reserved.
	EBX	Bits 31-00: Reserved.
	ECX	Bits 31-00: Reserved.
	EDX	Bits 31-00: Reserved.
	Unimplemented CPUID Leaf Functions	
21H		Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is 21H. If the value returned by CPUID.0:EAX (the maximum input value for basic CPUID information) is at least 21H, 0 is returned in the registers EAX, EBX, ECX, and EDX. Otherwise, the data for the highest basic information leaf is returned.
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.

Table 1-3. Information Returned by CPUID Instruction(Continued)

Initial EAX Value	Information Provided about the Processor	
	Extended Function CPUID Information	
80000000H	EAX	Maximum Input Value for Extended Function CPUID Information.
	EBX ECX EDX	Reserved Reserved Reserved
80000001H	EAX EBX ECX	Extended Processor Signature and Feature Bits. Reserved 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 EBX ECX EDX	Processor Brand String Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued
80000003H	EAX EBX ECX EDX	Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued
80000004H	EAX EBX ECX EDX	Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued Processor Brand String Continued
80000005H	EAX EBX ECX EDX	Reserved = 0 Reserved = 0 Reserved = 0 Reserved = 0
80000006H	EAX EBX	Reserved = 0 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 1-3. 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	Reserved = 0
	EBX	Reserved = 0
	ECX	Reserved = 0
	EDX	Bits 07-00: Reserved = 0 Bit 08: Invariant TSC available if 1 Bits 31-09: Reserved = 0
80000008H	EAX	Virtual/Physical Address size Bits 07-00: #Physical Address Bits* Bits 15-08: #Virtual Address Bits Bits 31-16: Reserved = 0
	EBX	Bits 08-00: Reserved = 0 Bit 09: WBNOINVD is available if 1 Bits 31-10: Reserved = 0
	ECX	Reserved = 0
	EDX	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 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 and is processor specific.

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 11 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

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

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

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

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

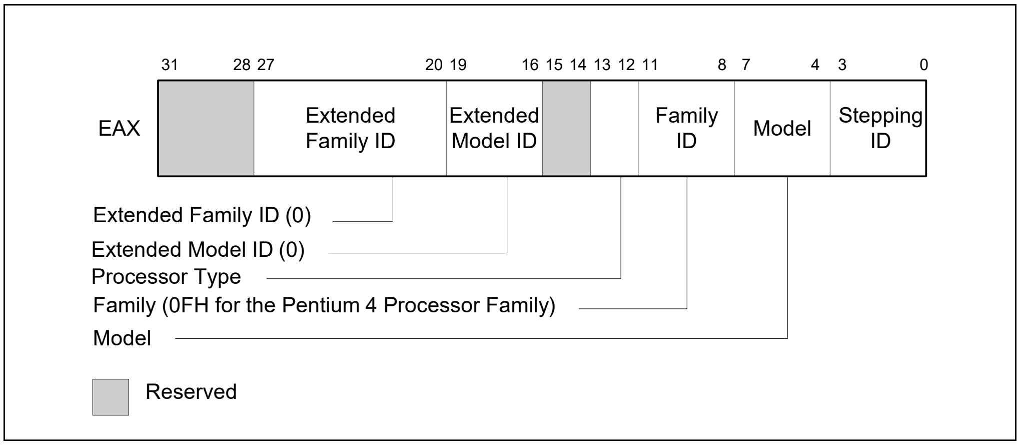


Figure 1-1. Version Information Returned by CPUID in EAX

Table 1-4. Processor Type Field

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

NOTE

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

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

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

```
FI;
(* Show Display_Family as HEX field. *)
```

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

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

INPUT EAX = 01H: Returns Additional Information in EBX

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

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

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

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

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

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

NOTE

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

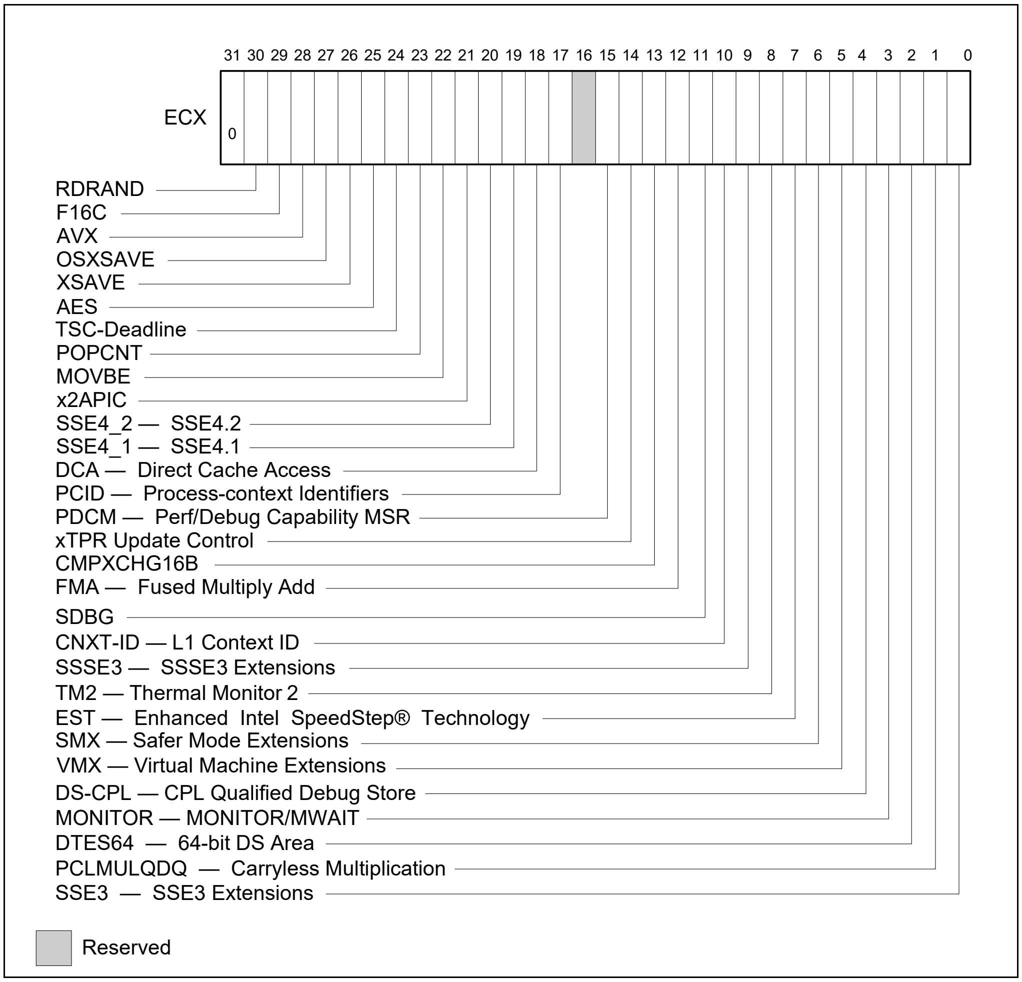


Figure 1-2. Feature Information Returned in the ECX Register

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

Bit #	Mnemonic	Description
0	SSE3	**Intel® Streaming SIMD Extensions 3 (Intel® 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 7, “Safer Mode Extensions Reference.”
7	EST	**Enhanced Intel SpeedStep® Technology.** A value of 1 indicates that the processor supports this technology.
8	TM2	**Thermal Monitor 2.** A value of 1 indicates whether the processor supports this technology.
9	SSSE3	A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor.

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

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

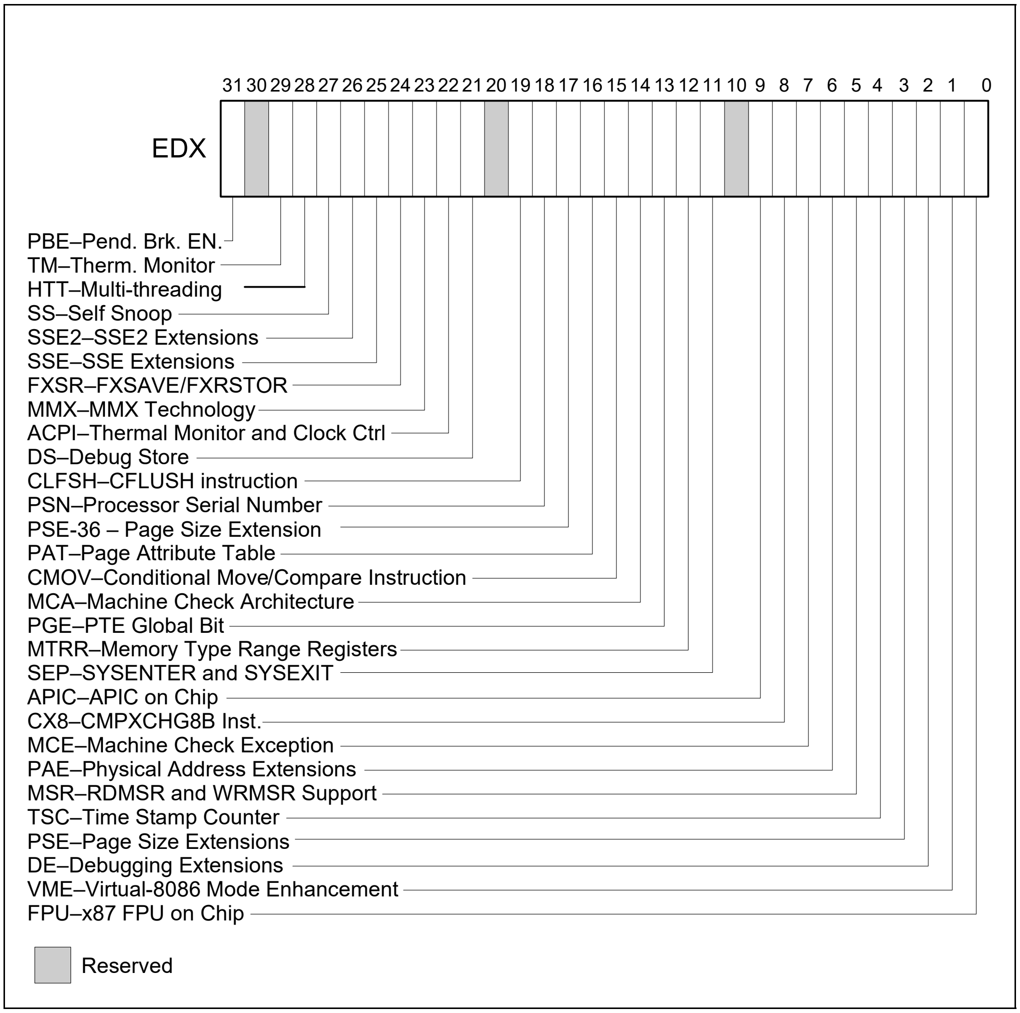


Figure 1-3. Feature Information Returned in the EDX Register

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

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

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

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

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

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

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

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

The encoding is as follows:

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

Table 1-7. Encoding of Cache and TLB Descriptors

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

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

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

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

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

Example 1-1. Example of Cache and TLB Interpretation

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

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

Which means:

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

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

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

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 1-3.

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 9, "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 1-3.

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 1-3.

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 1-3.

When CPUID executes with EAX set to 07H and ECX = n (n ≥ 1 and 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 1-3. 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 1-3.

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 1-3) is greater than Pn 0. See Table 1-3.

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 18, "Debug, Branch Profile, TSC, and Intel® Resource Director Technology (Intel® RDT) Features," 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 1-3.

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 1-3.

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 1-3. 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 1-3.

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 1-3.

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 1-3.

When CPUID executes with EAX set to 12H and ECX = 1H, the processor returns information about Intel SGX attributes. See Table 1-3.

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 1-3.

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 1-3.

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 1-3.

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 1-3.

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 1-3.

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 1-3.

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 1-3.

INPUT EAX = 19H: Returns Key Locker Information

When CPUID executes with EAX set to 19H, the processor returns information about Key Locker. See Table 1-3.

INPUT EAX = 1AH: Returns Hybrid Information

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

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 = 1CH: Returns Last Branch Record Information

When CPUID executes with EAX set to 1CH, the processor returns information about LBRs (the architectural feature). See Table 1-3.

INPUT EAX = 1DH: Returns Tile Information

When CPUID executes with EAX set to 1DH and ECX = 0H, the processor returns information about tile architecture. See Table 1-3.

When CPUID executes with EAX set to 1DH and ECX = 1H, the processor returns information about tile palette 1. See Table 1-3.

INPUT EAX = 1EH: Returns TMUL Information

When CPUID executes with EAX set to 1EH and ECX = 0H, the processor returns information about TMUL 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 1-3.

INPUT EAX = 20H: Returns Processor History Reset Information

When CPUID executes with EAX set to 20H, the processor returns information about processor history reset. See Table 1-3.

INPUT EAX = 23H: Returns Architectural Performance Monitoring Extended Information

When CPUID executes with EAX set to 23H, the processor returns architectural performance monitoring extended information. See Table 1-3.

METHODS FOR RETURNING BRANDING INFORMATION

Use the following techniques to access branding information:

1. Processor brand string method; this method also returns the processor’s maximum operating frequency

2. Processor brand index; this method uses a software supplied brand string table.

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

The Processor Brand String Method

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

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

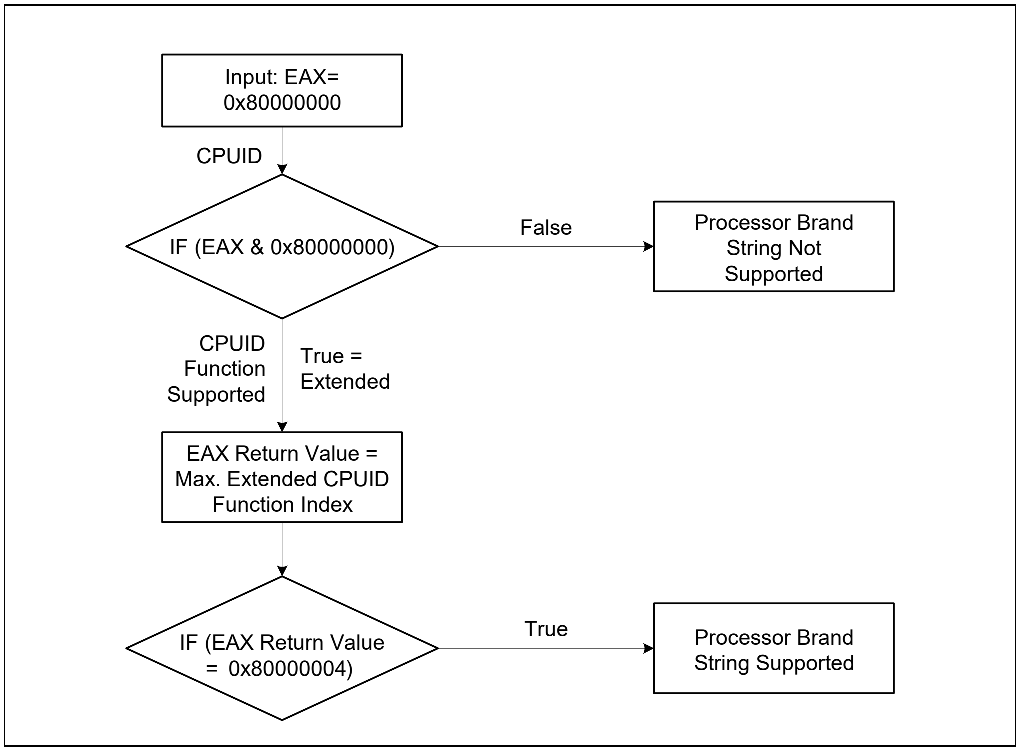


Figure 1-4. Determination of Support for the Processor Brand String

How Brand Strings Work

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

Table 1-8 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

Table 1-8. Processor Brand String Returned with Pentium 4 Processor

EAX Input Value	Return Values	ASCII Equivalent
80000002H	EAX = 20202020H	" "
	EBX = 20202020H	" "
	ECX = 20202020H	" "
	EDX = 6E492020H	"nl "

Table 1-8. Processor Brand String Returned with Pentium 4 Processor (Continued)

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

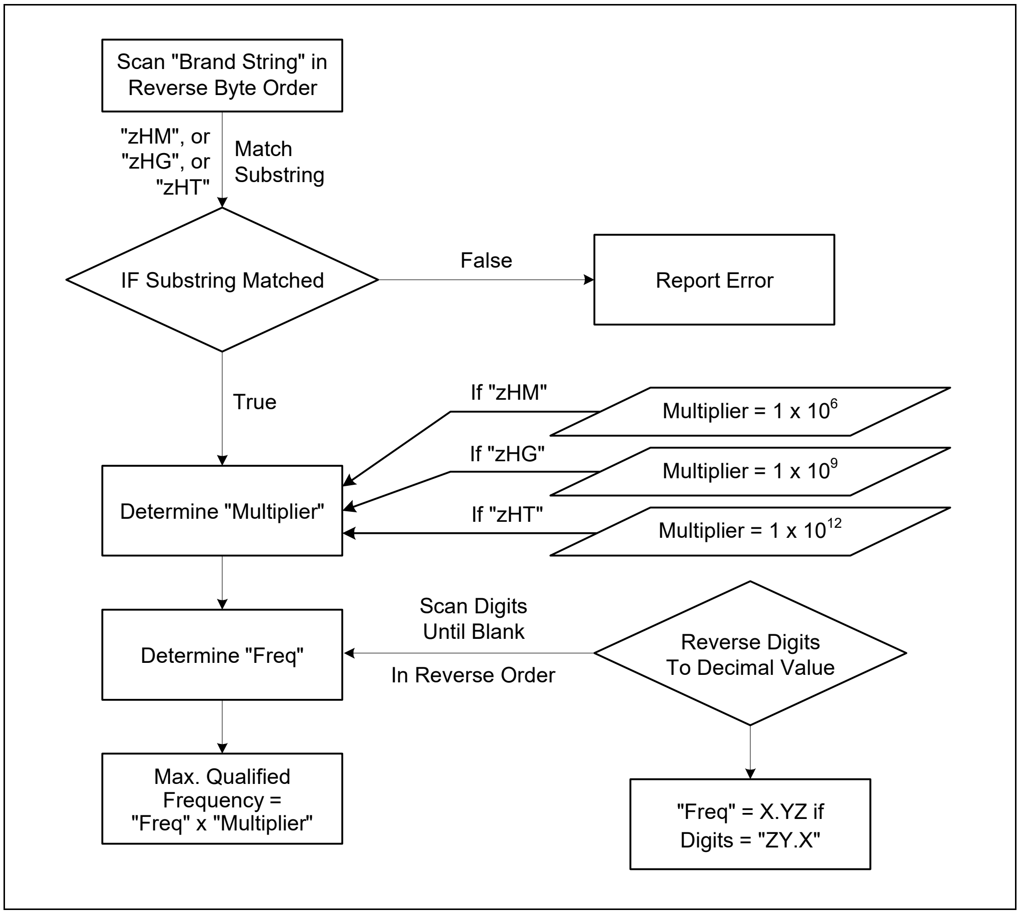


Figure 1-5. Algorithm for Extracting Maximum Processor Frequency

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.

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 1-9 shows brand indices that have identification strings associated with them.

Table 1-9. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings

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

NOTES:

[1] Indicates versions of these processors that were introduced after the Pentium III

IA-32 Architecture Compatibility

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

Operation

```
IA32_BIOS_SIGN_ID MSR := Update with installed microcode revision number;

CASE (EAX) OF
    EAX = 0:
        EAX := Highest basic function input value understood by CPUID;
        EBX := Vendor identification string;
        EDX := Vendor identification string;
        ECX := Vendor identification string;
    BREAK;
    EAX = 1H:
        EAX[3:0] := Stepping ID;
        EAX[7:4] := Model;
        EAX[11:8] := Family;
        EAX[13:12] := Processor type;
        EAX[15:14] := Reserved;
        EAX[19:16] := Extended Model;
        EAX[27:20] := Extended Family;
        EAX[31:28] := Reserved;
        EBX[7:0] := Brand Index; (* Reserved if the value is zero. *)
        EBX[15:8] := CLFLUSH Line Size;
        EBX[16:23] := Reserved; (* Number of threads enabled = 2 if MT enable fuse set. *)
        EBX[24:31] := Initial APIC ID;
        ECX := Feature flags; (* See Figure 1-2. *)
        EDX := Feature flags; (* See Figure 1-3. *)
    BREAK;
    EAX = 2H:
        EAX := Cache and TLB information;
        EBX := Cache and TLB information;
        ECX := Cache and TLB information;
        EDX := Cache and TLB information;
    BREAK;
    EAX = 3H:
        EAX := Reserved;
        EBX := Reserved;
        ECX := ProcessorSerialNumber[31:0];
        (* Pentium III processors only, otherwise reserved. *)
        EDX := ProcessorSerialNumber[63:32];
        (* Pentium III processors only, otherwise reserved. *
    BREAK
    EAX = 4H:
        EAX := Deterministic Cache Parameters Leaf; (* See Table 1-3. *)
        EBX := Deterministic Cache Parameters Leaf;
        ECX := Deterministic Cache Parameters Leaf;
        EDX := Deterministic Cache Parameters Leaf;
    BREAK;
    EAX = 5H:
        EAX := MONITOR/MWAIT Leaf; (* See Table 1-3. *)
        EBX := MONITOR/MWAIT Leaf;
        ECX := MONITOR/MWAIT Leaf;
        EDX := MONITOR/MWAIT Leaf;
    BREAK;
    EAX = 6H:
        EAX := Thermal and Power Management Leaf; (* See Table 1-3. *)
        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 1-3. *);
        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 1-3. *)
        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 1-3. *)
        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 1-3. *)
        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 1-3. *)
        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 1-3. *)
        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 1-3. *)
        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 1-3. *)
        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 1-3. *)
        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 1-3. *)
        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 1-3. *)
        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 1-3. *)
        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 1-3. *)
        EBX := Deterministic Address Translation Parameters Enumeration Leaf;
        ECX :=Deterministic Address Translation Parameters Enumeration Leaf;
        EDX := Deterministic Address Translation Parameters Enumeration Leaf;
    BREAK;
    EAX = 19H:
        EAX := Key Locker Enumeration Leaf; (* See Table 1-3. *)
        EBX := Key Locker Enumeration Leaf;
        ECX := Key Locker Enumeration Leaf;
        EDX := Key Locker Enumeration Leaf;
    BREAK;
    EAX = 1AH:
        EAX := Hybrid Information Enumeration Leaf; (* See Table 1-3. *)
        EBX := Hybrid Information Enumeration Leaf;
        ECX := Hybrid Information Enumeration Leaf;
        EDX := Hybrid Information Enumeration Leaf;
    BREAK;
    EAX = 1BH:
        EAX := PCONFIG Information Enumeration Leaf; (* See Table 1-3. *)
        EBX := PCONFIG Information Enumeration Leaf;
        ECX := PCONFIG Information Enumeration Leaf;
        EDX := PCONFIG Information Enumeration Leaf;
    BREAK;
    EAX = 1CH:
        EAX := Last Branch Record Information Enumeration Leaf; (* See Table 1-3. *)
        EBX := Last Branch Record Information Enumeration Leaf;
        ECX := Last Branch Record Information Enumeration Leaf;
        EDX := Last Branch Record Information Enumeration Leaf;
    BREAK;
    EAX = 1DH:
        EAX := Tile Information Enumeration Leaf; (* See Table 1-3. *)
        EBX := Tile Information Enumeration Leaf;
        ECX := Tile Information Enumeration Leaf;
        EDX := Tile Information Enumeration Leaf;
    BREAK;
    EAX = 1EH:
        EAX := TMUL Information Enumeration Leaf; (* See Table 1-3. *)
        EBX := TMUL Information Enumeration Leaf;
        ECX := TMUL Information Enumeration Leaf;
        EDX := TMUL Information Enumeration Leaf;
    BREAK;
    EAX = 1FH:
        EAX := V2 Extended Topology Enumeration Leaf; (* See Table 1-3. *)
        EBX := V2 Extended Topology Enumeration Leaf;
        ECX := V2 Extended Topology Enumeration Leaf;
        EDX := V2 Extended Topology Enumeration Leaf;
    BREAK;
    EAX = 20H:
        EAX := Processor History Reset Enumeration Leaf; (* See Table 1-3. *)
        EBX := Processor History Reset Enumeration Leaf;
        ECX := Processor History Reset Enumeration Leaf;
        EDX := Processor History Reset Enumeration Leaf;
    BREAK;
    EAX = 23H:
        EAX := Architectural Performance Monitoring Extended Leaf; (* See Table 1-3. *)
        EBX := Architectural Performance Monitoring Extended Leaf;
        ECX := Architectural Performance Monitoring Extended Leaf;
        EDX := Architectural Performance Monitoring Extended 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 1-3.*);
        EDX := Extended Feature Bits (* See Table 1-3. *);
    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.§

1.6 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 1-10 and Table 1-11 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 1-10 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 1-11. Table 1-11 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 1-11. Instruction classified in Table 1-11 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 1-10. 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 1-11. 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 1-11. 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

1.7 BFLOAT16 FLOATING-POINT FORMAT

Intel® Deep Learning Boost (Intel® DL Boost) uses bfloat16 format (BF16). Figure 1-6 illustrates BF16 versus FP16 and FP32.

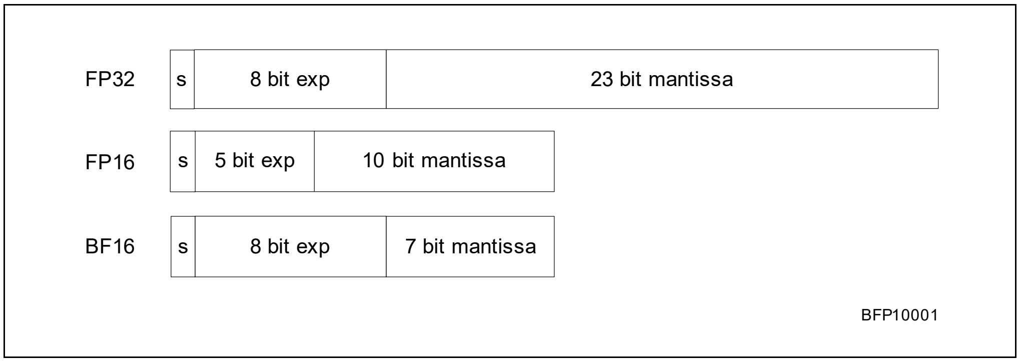


Figure 1-6. Comparison of BF16 to FP16 and FP32

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*. Additionally, some instructions use notation conventions as described below.

In the instruction encoding, the MODRM byte is represented several ways depending on the role it plays. The MODRM byte has 3 fields: 2-bit MODRM.MOD field, a 3-bit MODRM.REG field and a 3-bit MODRM.RM field. When all bits of the MODRM byte have fixed values for an instruction, the 2-hex nibble value of that byte is presented after the opcode in the encoding boxes on the instruction description pages. When only some fields of the MODRM byte must contain fixed values, those values are specified as follows:

- If only the MODRM.MOD must be 0b11, and MODRM.REG and MODRM.RM fields are unrestricted, this is denoted as **11:rrr:bbb**. The **rrr** correspond to the 3-bits of the MODRM.REG field and the **bbb** correspond to the 3-bits of the MODMR.RM field.
- If the MODRM.MOD field is constrained to be a value other than 0b11, i.e., it must be one of 0b00, 0b01, or 0b10, then we use the notation !(11).
- If for example only the MODRM.REG field had a specific required value, e.g., 0b101, that would be denoted as mm:101:bbb.

NOTE

Historically the Intel® 64 and IA-32 Architectures Software Developer's Manual only specified the MODRM.REG field restrictions with the notation /0 ... /7 and did not specify restrictions on the MODRM.MOD and MODRM.RM fields in the encoding boxes.

2.1 INSTRUCTION SET REFERENCE

AADD—Atomically Add

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F38 FC !(11):rrr:bbb AADD *my, ry*	A	V/V	RAO-INT	Atomically add *my* with *ry* and store the result in *my*.

Instruction Operand Encoding

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

Description

This instruction atomically adds the destination operand (first operand) and the source operand (second operand), and then stores the result in the destination operand.

The destination operand is a memory location and the source operand is a register. In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. The destination operand must be naturally aligned with respect to the data size, at a 4-byte boundary, or an 8-byte boundary if used with a REX.W prefix in 64-bit mode.

This instruction requires that the destination operand has a write-back (WB) memory type and it is implemented using the weakly-ordered memory consistency model of write combining (WC) memory type. Before the operation, the cache line is written-back (if modified) and invalidated from the processor cache. When the operation completes, the processor may optimize the cacheability of the destination address by writing the result only to specific levels of the cache hierarchy. Because this instructions uses a weakly-ordered memory consistency model, a fencing operation implemented with LFENCE, SFENCE, or MFENCE instruction should be used in conjunction with AADD if a stronger ordering is required. However, note that AADD is not ordered with respect to a younger LFENCE, as this instruction is not loading data from memory into the processor.

Any attempt to execute the AADD instruction inside an Intel TSX transaction will result in a transaction abort.

Operation

AADD dest, src

```
dest := dest + src;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If the memory address is not naturally aligned to the operand size.
	If the memory address memory type is not write-back (WB).
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=01H):EAX.RAO-INT[bit 3] = 0.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside the effective address space from 0 to FFFFH.
If the memory address is not naturally aligned to the operand size.
If the memory address memory type is not write-back (WB).

#SS For an illegal address in the SS segment.

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=01H):EAX.RAO-INT[bit 3] = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
If the memory address is not naturally aligned to the operand size.
If the memory address memory type is not write-back (WB).

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=01H):EAX.RAO-INT[bit 3] = 0.

AAND—Atomically AND

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F38 FC !(11):rrr:bbb AAND *my, ry*	A	V/V	RAO-INT	Atomically AND *my* with *ry* and store the result in *my*.

Instruction Operand Encoding

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

Description

This instruction atomically performs a bitwise AND operation of the destination operand (first operand) and the source operand (second operand), and then stores the result in the destination operand.

The destination operand is a memory location and the source operand is a register. In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. The destination operand must be naturally aligned with respect to the data size, at a 4-byte boundary, or an 8-byte boundary if used with a REX.W prefix in 64-bit mode.

This instruction requires that the destination operand has a write-back (WB) memory type and it is implemented using the weakly-ordered memory consistency model of write combining (WC) memory type. Before the operation, the cache line is written-back (if modified) and invalidated from the processor cache. When the operation completes, the processor may optimize the cacheability of the destination address by writing the result only to specific levels of the cache hierarchy. Because this instructions uses a weakly-ordered memory consistency model, a fencing operation implemented with LFENCE, SFENCE, or MFENCE instruction should be used in conjunction with AAND if a stronger ordering is required. However, note that AAND is not ordered with respect to a younger LFENCE, as this instruction is not loading data from memory into the processor.

Any attempt to execute the AAND instruction inside an Intel TSX transaction will result in a transaction abort.

Operation

AAND dest, src

```
dest := dest AND src;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If the memory address is not naturally aligned to the operand size.
	If the memory address memory type is not write-back (WB).
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=01H):EAX.RAO-INT[bit 3] = 0.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside the effective address space from 0 to FFFFH.
If the memory address is not naturally aligned to the operand size.
If the memory address memory type is not write-back (WB).

#SS For an illegal address in the SS segment.

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=01H):EAX.RAO-INT[bit 3] = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
If the memory address is not naturally aligned to the operand size.
If the memory address memory type is not write-back (WB).

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=01H):EAX.RAO-INT[bit 3] = 0.

AOR—Atomically OR

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F38 FC !(11):rrr:bbb AOR *my, ry*	A	V/V	RAO-INT	Atomically OR *my* with *ry* and store the result in *my*.

Instruction Operand Encoding

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

Description

This instruction atomically performs a bitwise OR operation of the destination operand (first operand) and the source operand (second operand), and then stores the result in the destination operand.

The destination operand is a memory location and the source operand is a register. In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. The destination operand must be naturally aligned with respect to the data size, at a 4-byte boundary, or an 8-byte boundary if used with a REX.W prefix in 64-bit mode.

This instruction requires that the destination operand has a write-back (WB) memory type and it is implemented using the weakly-ordered memory consistency model of write combining (WC) memory type. Before the operation, the cache line is written-back (if modified) and invalidated from the processor cache. When the operation completes, the processor may optimize the cacheability of the destination address by writing the result only to specific levels of the cache hierarchy. Because this instructions uses a weakly-ordered memory consistency model, a fencing operation implemented with LFENCE, SFENCE, or MFENCE instruction should be used in conjunction with AOR if a stronger ordering is required. However, note that AOR is not ordered with respect to a younger LFENCE, as this instruction is not loading data from memory into the processor.

Any attempt to execute the AOR instruction inside an Intel TSX transaction will result in a transaction abort.

Operation

AOR dest, src

```
dest := dest OR src;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If the memory address is not naturally aligned to the operand size.
	If the memory address memory type is not write-back (WB).
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=01H):EAX.RAO-INT[bit 3] = 0.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside the effective address space from 0 to FFFFH.
If the memory address is not naturally aligned to the operand size.
If the memory address memory type is not write-back (WB).

#SS For an illegal address in the SS segment.

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=01H):EAX.RAO-INT[bit 3] = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
If the memory address is not naturally aligned to the operand size.
If the memory address memory type is not write-back (WB).

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=01H):EAX.RAO-INT[bit 3] = 0.

AXOR—Atomically XOR

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F38 FC !(11):rrr:bbb AXOR *my, ry*	A	V/V	RAO-INT	Atomically XOR *my* with *ry* and store the result in *my*.

Instruction Operand Encoding

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

Description

This instruction atomically performs a bitwise XOR operation of the destination operand (first operand) and the source operand (second operand), and then stores the result in the destination operand.

The destination operand is a memory location and the source operand is a register. In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. The destination operand must be naturally aligned with respect to the data size, at a 4-byte boundary, or an 8-byte boundary if used with a REX.W prefix in 64-bit mode.

This instruction requires that the destination operand has a write-back (WB) memory type and it is implemented using the weakly-ordered memory consistency model of write combining (WC) memory type. Before the operation, the cache line is written-back (if modified) and invalidated from the processor cache. When the operation completes, the processor may optimize the cacheability of the destination address by writing the result only to specific levels of the cache hierarchy. Because this instructions uses a weakly-ordered memory consistency model, a fencing operation implemented with LFENCE, SFENCE, or MFENCE instruction should be used in conjunction with AXOR if a stronger ordering is required. However, note that AXOR is not ordered with respect to a younger LFENCE, as this instruction is not loading data from memory into the processor.

Any attempt to execute the AXOR instruction inside an Intel TSX transaction will result in a transaction abort.

Operation

```
AXOR dest, src

dest := dest XOR src;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the memory address is not naturally aligned to the operand size. If the memory address memory type is not write-back (WB).
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=01H):EAX.RAO-INT[bit 3] = 0.

Real-Address Mode Exceptions

#GP If any part of the operand lies outside the effective address space from 0 to FFFFH.
If the memory address is not naturally aligned to the operand size.
If the memory address memory type is not write-back (WB).

#SS For an illegal address in the SS segment.

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=01H):EAX.RAO-INT[bit 3] = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
If the memory address is not naturally aligned to the operand size.
If the memory address memory type is not write-back (WB).

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=01H):EAX.RAO-INT[bit 3] = 0.

CMP*cc*XADD—Compare and Add if Condition is Met

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 E6 !(11):rrr:bbb CMPBEXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If below or equal (CF=1 or ZF=1), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E6 !(11):rrr:bbb CMPBEXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If below or equal (CF=1 or ZF=1), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 E2 !(11):rrr:bbb CMPBXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If below (CF=1), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E2 !(11):rrr:bbb CMPBXADD m64, r64, r64	A	V/N.E	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If below (CF=1), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 EE !(11):rrr:bbb CMPLEXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If less or equal (ZF=1 or SF≠OF), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 EE !(11):rrr:bbb CMPLEXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If less or equal (ZF=1 or SF≠OF), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 EC !(11):rrr:bbb CMPLXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If less (SF≠OF), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 EC !(11):rrr:bbb CMPLXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If less (SF≠OF), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 E7 !(11):rrr:bbb CMPNBEXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If not below or equal (CF=0 and ZF=0), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 E7 !(11):rrr:bbb CMPNBEXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If not below or equal (CF=0 and ZF=0), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 E3 !(11):rrr:bbb CMPNBXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If not below (CF=0), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E3 !(11):rrr:bbb CMPNBXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If not below (CF=0), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 EF !(11):rrr:bbb CMPNLEXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If not less or equal (ZF=0 and SF=OF), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 EF !(11):rrr:bbb CMPNLEXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If not less or equal (ZF=0 and SF=OF), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 ED !(11):rrr:bbb CMPNLXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If not less (SF=OF), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 ED !(11):rrr:bbb CMPNLXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If not less (SF=OF), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 E1 !(11):rrr:bbb CMPNOXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If not overflow (OF=0), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E1 !(11):rrr:bbb CMPNOXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If not overflow (OF=0), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 EB !(11):rrr:bbb CMPNPXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If not parity (PF=0), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 EB !(11):rrr:bbb CMPNPXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If not parity (PF=0), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 E9 !(11):rrr:bbb CMPNSXADD m32, r32, r32	A	V/N.E	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If not sign (SF=0), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E9 !(11):rrr:bbb CMPNSXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If not sign (SF=0), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 E5 !(11):rrr:bbb CMPNZXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If not zero (ZF=0), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E5 !(11):rrr:bbb CMPNZXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If not zero (ZF=0), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 E0 !(11):rrr:bbb CMPOXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If overflow (OF=1), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E0 !(11):rrr:bbb CMPOXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If overflow (OF=1), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 EA !(11):rrr:bbb CMPPXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If parity (PF=1), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 EA !(11):rrr:bbb CMPPXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If parity (PF=1), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 E8 !(11):rrr:bbb CMPSXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If sign (SF=1), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E8 !(11):rrr:bbb CMPSXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If sign (SF=1), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.
VEX.128.66.0F38.W0 E4 !(11):rrr:bbb CMPZXADD m32, r32, r32	A	V/N.E.	CMPCCXADD	Compare value in r32 (second operand) with value in m32. If zero (ZF=1), add value from r32 (third operand) to m32 and write new value in m32. The second operand is always updated with the original value from m32.
VEX.128.66.0F38.W1 E4 !(11):rrr:bbb CMPZXADD m64, r64, r64	A	V/N.E.	CMPCCXADD	Compare value in r64 (second operand) with value in m64. If zero (ZF=1), add value from r64 (third operand) to m64 and write new value in m64. The second operand is always updated with the original value from m64.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:r/m (r, w)	ModRM:reg (r, w)	VEX.vvvv (r)	N/A

Description

This instruction compares the value from memory with the value of the second operand. If the specified condition is met, then the processor will add the third operand to the memory operand and write it into memory, else the memory is unchanged by this instruction.

This instruction must have MODRM.MOD equal to 0, 1, or 2. The value 3 for MODRM.MOD is reserved and will cause an invalid opcode exception (#UD).

The second operand is always updated with the original value of the memory operand. The EFLAGS conditions are updated from the results of the comparison.The instruction uses an implicit lock. This instruction does not permit the use of an explicit lock prefix.

Operation

```
CMPCCXADD srcdest1, srcdest2, src3
tmp1 := load lock srcdest1
tmp2 := tmp1 + src3
EFLAGS.CS,OF,SF,ZF,AF,PF := CMP tmp1, srcdest2
IF <condition>:
    srcdest1 := store unlock tmp2
ELSE
    srcdest1 := store unlock tmp1
srcdest2 :=tmp1
```

Flags Affected

The EFLAGS conditions are updated from the results of the comparison.

SIMD Floating-Point Exceptions

None.

Exceptions

Exceptions Type 14. See Table 2-1, "Type 14 Class Exception Conditions".

Table 2-1. Type 14 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X		Only supported in 64-bit mode.
				X	If any LOCK, REX, F2, F3, or 66 prefixes precede a VEX prefix.
				X	If any corresponding CPUID feature flag is '0'.
Stack, #SS(0)				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)				X	If not naturally aligned (4/8 bytes).
				X	If the memory address is in a non-canonical form.
Page Fault #PF(fault-code)				X	If a page fault occurs.

RDMSRLIST—Read List of Model Specific Registers

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 01 C6 RDMSRLIST	ZO	V/N.E.	MSRLIST	Read the requested list of MSRs, and store the read values to memory.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

This instruction reads a software-provided list of up to 64 MSRs and stores their values in memory.

RDMSRLIST takes three implied input operands:

- RSI: Linear address of a table of MSR addresses (8 bytes per address).
- RDI: Linear address of a table into which MSR data is stored (8 bytes per MSR).
- RCX: 64-bit bitmask of valid bits for the MSRs. Bit 0 is the valid bit for entry 0 in each table, etc.

For each RCX bit [n] from 0 to 63, if RCX[n] is 1, RDMSRLIST will read the MSR specified at entry [n] in the RSI table and write it out to memory at the entry [n] in the RDI table.

This implies a maximum of 64 MSRs that can be processed by this instruction. The processor will clear RCX[n] after it finishes handling that MSR. Similar to repeated string operations, RDMSRLIST supports partial completion for interrupts, exceptions, and traps. In these situations, the RIP register saved will point to the RDMSRLIST instruction while the RCX register will have cleared bits corresponding to all completed iterations.

This instruction must be executed at privilege level 0; otherwise, a general protection exception #GP(0) is generated. This instruction performs MSR specific checks and respects the VMX MSR VM-execution controls in the same manner as RDMSR.

Although RDMSRLIST accesses the entries in the two tables in order, the actual reads of the MSRs may be performed out of order: for table entries m < n, the processor may read the MSR for entry n before reading the MSR for entry m. (This may be true also for a sequence of executions of RDMSR.) Ordering is guaranteed if the address of the IA32_BARRIER MSR (2FH) appears in the table of MSR addresses. Specifically, if IA32_BARRIER appears at entry m, then the MSR read for any entry n with n > m will not occur until (1) all instructions prior to RDMSRLIST have completed locally; and (2) MSRs have been read for all table entries before entry m.

The processor is allowed to (but not required to) "load ahead" in the list. Examples:

- Use old memory type or TLB translation for loads/stores to list memory despite an MSR written by a previous iteration changing MTRR or invalidating TLBs.
- Cause a page fault or EPT violation for a memory access to an entry > "n" in MSR address or data tables, despite the processor only having read or written "n" MSRs.[1]

Virtualization Behavior—VM Exit Causes

Like RDMSR, the RDMSRLIST instruction executed in VMX non-root operation causes a VM exit if any of the following are true:

- The "use MSR bitmaps" VM-execution control is 0.
- The value of MSR address is not in the ranges 00000000H–00001FFFH and C0000000H–C0001FFFH.
- The value of MSR address is in the range 00000000H–00001FFFH and bit n in read bitmap for low MSRs is 1, where n is the value of the MSR address.

[1] For example, the processor may take a page fault due to a linear address for the 10th entry in the MSR address table despite only having completed the MSR writes up to entry 5.

- The value of ECX is in the range C0000000H–C0001FFFH and bit n in read bitmap for high MSRs is 1, where n is the value of the MSR address & 00001FFFH.

A VM exit for the above reasons for the RDMSRLIST instruction will specify exit reason 78 (decimal). The exit qualification is set to the MSR address causing the VM exit if "use MSR bitmaps" VM-execution control is 1. If "use MSR bitmaps" VM-execution control is 0, then the VM-exit qualification will be 0.

If software wants to emulate a single iteration of RDMSRLIST after a VM exit, it can use the exit qualification to identify the MSR. Such software will need to write to the table of data. It can calculate the guest-linear address of the table entry to write by using the values of RDI (the guest-linear address of the table) and RCX (the lowest bit set in RCX identifies the specific table entry.

Virtualization Behavior—Changed Behavior in Non-Root Operation

The previous section identifies when executions of the RDMSRLIST instruction cause VM exits. Under the following situations, a #UD will occur instead of a VM exit or a fault due to CPL 0:

- The "Enable MSRLIST Instructions" VM-execution control is 0.
- The "Activate tertiary controls" VM-execution control is 0.

If that does not occur and there is no fault due to CPL > 0 nor a VM exit, the instruction's behavior may be modified for certain values of MSR address in the same manner as RDMSR for a read of the same MSR.

Operation

```
WHILE (RCX != 0) {
    MSR_index = TZCNT(RCX)
    MSR_address = mem[RSI + (MSR_index * 8) ]
    VM exit if specified by VM-execution controls (for specified MSR_address)
    #GP(0) if MSR_address[61:32] !=0
    #GP(0) if MSR_address is not accessible for RDMSR
    mem[RDI + (MSR_index * 8)]) = RDMSR (MSR_address)
    Clear RCX [MSR_index]
    Take any pending interrupts/traps
}
```

Flags Affected

None.

Protected Mode Exceptions

#UD	The RDMSRLIST instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The RDMSRLIST instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

#UD	The RDMSRLIST instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#GP(0) If the current privilege level is not 0.

If RSI [2:0] ≠ 0 OR RDI [2:0] ≠ 0.

If an execution of RDMSR from a specified MSR would generate a general protection exception #GP(0).

#UD If the LOCK prefix is used.

If not in 64-bit mode.

If CPUID.(EAX=07H, ECX=01H):EAX.MSRLIST[bit 27] = 0.

VBCSTNEBF162PS—Load BF16 Element and Convert to FP32 Element With Broadcast

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F3.0F38.W0 B1 !(11):rrr:bbb VBCSTNEBF162PS xmm1, m16	A	V/V	AVX-NE-CONVERT	Load one BF16 floating-point element from m16, convert to FP32 and store result in xmm1.
VEX.256.F3.0F38.W0 B1 !(11):rrr:bbb VBCSTNEBF162PS ymm1, m16	A	V/V	AVX-NE-CONVERT	Load one BF16 floating-point element from m16, convert to FP32 and store result in ymm1.

Instruction Operand Encoding

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

Description

This instruction loads one BF16 element from memory, converts it to FP32, and broadcasts it to a SIMD register.

This instruction does not generate floating-point exceptions and does not consult or update MXCSR.

Denormal BF16 input operands are treated as zeros (DAZ). Since any BF16 number can be represented in FP32, the conversion result is exact and no rounding is needed.

Operation

VBCSTNEBF162PS dest, src (VEX encoded version)

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

FOR i in range(0, KL):
    tmp.dword[i].word[0] = src.word[0] // reads 16b from memory

FOR i in range(0, KL):
    dest.dword[i] = make_fp32(TMP.dword[i].word[0])
DEST[MAXVL-1:VL] := 0
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5.

VBCSTNESH2PS—Load FP16 Element and Convert to FP32 Element with Broadcast

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 B1 !(11):rrr:bbb VBCSTNESH2PS xmm1, m16	A	V/V	AVX-NE-CONVERT	Load one FP16 element from m16, convert to FP32, and store result in xmm1.
VEX.256.66.0F38.W0 B1 !(11):rrr:bbb VBCSTNESH2PS ymm1, m16	A	V/V	AVX-NE-CONVERT	Load one FP16 element from m16, convert to FP32, and store result in ymm1.

Instruction Operand Encoding

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

Description

This instruction loads one FP16 element from memory, converts it to FP32, and broadcasts it to a SIMD register.

This instruction does not generate floating-point exceptions and does not consult or update MXCSR.

Input FP16 denormals are converted to normal FP32 numbers and not treated as zero. Since any FP16 number can be represented in FP32, the conversion result is exact and no rounding is needed.

Operation

VBCSTNESH2PS dest, src (VEX encoded version)

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

FOR i in range(0, KL):
    tmp.dword[i].word[0] = src.word[0] // read 16b from memory

FOR i in range(0, KL):
    dest.dword[i] = convert_fp16_to_fp32(tmp.dword[i].word[0])  //SAE
DEST[MAXVL-1:VL] := 0
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exception Type 5.

VCVTNEEBF162PS—Convert Even Elements of Packed BF16 Values to FP32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F3.0F38.W0 B0 !(11):rrr:bbb VCVTNEEBF162PS xmm1, m128	A	V/V	AVX-NE-CONVERT	Convert even elements of packed BF16 values from m128 to FP32 values and store in xmm1.
VEX.256.F3.0F38.W0 B0 !(11):rrr:bbb VCVTNEEBF162PS ymm1, m256	A	V/V	AVX-NE-CONVERT	Convert even elements of packed BF16 values from m256 to FP32 values and store in ymm1.

Instruction Operand Encoding

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

Description

This instruction loads packed BF16 elements from memory, converts the even elements to FP32, and writes the result to the destination SIMD register.

This instruction does not generate floating-point exceptions and does not consult or update MXCSR.

Denormal BF16 input operands are treated as zeros (DAZ). Since any BF16 number can be represented in FP32, the conversion result is exact and no rounding is needed.

Operation

VCVTNEEBF162PS dest, src (VEX encoded version)

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

FOR i in range(0, KL):
    dest.dword[i] = make_fp32(src.dword[i].word[0])
DEST[MAXVL-1:VL] := 0
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exception Type 4.

VCVTNEEPH2PS—Convert Even Elements of Packed FP16 Values to FP32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 B0 !(11):rrr:bbb VCVTNEEPH2PS xmm1, m128	A	V/V	AVX-NE-CONVERT	Convert even elements of packed FP16 values from m128 to FP32 values and store in xmm1.
VEX.256.66.0F38.W0 B0 !(11):rrr:bbb VCVTNEEPH2PS ymm1, m256	A	V/V	AVX-NE-CONVERT	Convert even elements of packed FP16 values from m256 to FP32 values and store in ymm1.

Instruction Operand Encoding

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

Description

This instruction loads packed FP16 elements from memory, converts the even elements to FP32, and writes the result to the destination SIMD register.

This instruction does not generate floating-point exceptions and does not consult or update MXCSR.

Input FP16 denormals are converted to normal FP32 numbers and not treated as zero. Since any FP16 number can be represented in FP32, the conversion result is exact and no rounding is needed.

Operation

VCVTNEEPH2PS dest, src (VEX encoded version)

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

FOR i in range(0, KL):
    dest.dword[i] = convert_fp16_to_fp32(src.dword[i].word[0]) //SAE
DEST[MAXVL-1:VL] := 0
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exception Type 4.

VCVTNEOBF162PS—Convert Odd Elements of Packed BF16 Values to FP32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F2.0F38.W0 B0 !(11):rrr:bbb VCVTNEOBF162PS xmm1, m128	A	V/V	AVX-NE-CONVERT	Convert odd elements of packed BF16 values from m128 to FP32 values and store in xmm1.
VEX.256.F2.0F38.W0 B0 !(11):rrr:bbb VCVTNEOBF162PS ymm1, m256	A	V/V	AVX-NE-CONVERT	Convert odd elements of packed BF16 values from m256 to FP32 values and store in ymm1.

Instruction Operand Encoding

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

Description

This instruction loads packed BF16 elements from memory, converts the odd elements to FP32, and writes the result to the destination SIMD register.

This instruction does not generate floating-point exceptions and does not consult or update MXCSR.

Denormal BF16 input operands are treated as zeros (DAZ). Since any BF16 number can be represented in FP32, the conversion result is exact and no rounding is needed.

Operation

VCVTNEOBF162PS dest, src (VEX encoded version)

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

FOR i in range(0, KL):
    dest.dword[i] = make_fp32(src.dword[i].word[1])
DEST[MAXVL-1:VL] := 0
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exception Type 4.

VCVTNEOPH2PS—Convert Odd Elements of Packed FP16 Values to FP32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.NP.0F38.W0 B0 !(11):rrr:bbb VCVTNEOPH2PS xmm1, m128	A	V/V	AVX-NE-CONVERT	Convert odd elements of packed FP16 values from m128 to FP32 values and store in xmm1.
VEX.256.NP.0F38.W0 B0 !(11):rrr:bbb VCVTNEOPH2PS ymm1, m256	A	V/V	AVX-NE-CONVERT	Convert odd elements of packed FP16 values from m256 to FP32 values and store in ymm1.

Instruction Operand Encoding

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

Description

This instruction loads packed FP16 elements from memory, converts the odd elements to FP32, and writes the result to the destination SIMD register.

This instruction does not generate floating-point exceptions and does not consult or update MXCSR.

Input FP16 denormals are converted to normal FP32 numbers and not treated as zero. Since any FP16 number can be represented in FP32, the conversion result is exact and no rounding is needed.

Operation

VCVTNEOPH2PS dest, src (VEX encoded version)

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

FOR i in range(0, KL):
    dest.dword[i] = convert_fp16_to_fp32(src.dword[i].word[1]) //SAE
DEST[MAXVL-1:VL] := 0
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exception Type 4.

VCVTNEPS2BF16—Convert Packed Single-Precision Floating-Point Values to BF16 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F3.0F38.W0 72 /r VCVTNEPS2BF16 xmm1, xmm2/m128	A	V/V	AVX-NE-CONVERT	Convert packed single-precision floating-point values from xmm2/m128 to packed BF16 values and store in xmm1.
VEX.256.F3.0F38.W0 72 /r VCVTNEPS2BF16 xmm1, ymm2/m256	A	V/V	AVX-NE-CONVERT	Convert packed single-precision floating-point values from ymm2/m256 to packed BF16 values and store in xmm1.

Instruction Operand Encoding

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

Description

This instruction loads packed FP32 elements from a SIMD register or memory, converts the elements to BF16, and writes the result to the destination SIMD register.

The upper bits of the destination register beyond the down-converted BF16 elements are zeroed.

This instruction uses “Round to nearest (even)” rounding mode. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated.

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 (VEX encoded version)

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

FOR i := 0 to KL/2-1:
    t := src.fp32[i]
    dest.word[i] := convert_fp32_to_bfloat16(t)

DEST[MAXVL-1:VL/2] := 0
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

VPDPB[SU,UU,SS]D[,S]—Multiply and Add Unsigned and Signed Bytes With and Without Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F2.0F38.W0 50 /r VPDPBSSD xmm1, xmm2, xmm3/m128	A	V/V	AVX-VNNI-INT8	Multiply groups of 4 pairs of signed bytes in xmm3/m128 with corresponding signed bytes of xmm2, summing those products and adding them to the doubleword result in xmm1.
VEX.256.F2.0F38.W0 50 /r VPDPBSSD ymm1, ymm2, ymm3/m256	A	V/V	AVX-VNNI-INT8	Multiply groups of 4 pairs of signed bytes in ymm3/m256 with corresponding signed bytes of ymm2, summing those products and adding them to the doubleword result in ymm1.
VEX.128.F2.0F38.W0 51 /r VPDPBSSDS xmm1, xmm2, xmm3/m128	A	V/V	AVX-VNNI-INT8	Multiply groups of 4 pairs of signed bytes in xmm3/m128 with corresponding signed bytes of xmm2, summing those products and adding them to the doubleword result, with signed saturation in xmm1.
VEX.256.F2.0F38.W0 51 /r VPDPBSSDS ymm1, ymm2, ymm3/m256	A	V/V	AVX-VNNI-INT8	Multiply groups of 4 pairs of signed bytes in ymm3/m256 with corresponding signed bytes of ymm2, summing those products and adding them to the doubleword result, with signed saturation in ymm1.
VEX.128.F3.0F38.W0 50 /r VPDPBSUD xmm1, xmm2, xmm3/m128	A	V/V	AVX-VNNI-INT8	Multiply groups of 4 pairs of signed bytes in xmm3/m128 with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result in xmm1.
VEX.256.F3.0F38.W0 50 /r VPDPBSUD ymm1, ymm2, ymm3/m256	A	V/V	AVX-VNNI-INT8	Multiply groups of 4 pairs of signed bytes in ymm3/m256 with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result in ymm1.
VEX.128.F3.0F38.W0 51 /r VPDPBSUDS xmm1, xmm2, xmm3/m128	A	V/V	AVX-VNNI-INT8	Multiply groups of 4 pairs of signed bytes in xmm3/m128 with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result, with signed saturation in xmm1.
VEX.256.F3.0F38.W0 51 /r VPDPBSUDS ymm1, ymm2, ymm3/m256	A	V/V	AVX-VNNI-INT8	Multiply groups of 4 pairs of signed bytes in ymm3/m256 with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result, with signed saturation in ymm1.
VEX.128.NP.0F38.W0 50 /r VPDPBUUD xmm1, xmm2, xmm3/m128	A	V/V	AVX-VNNI-INT8	Multiply groups of 4 pairs of unsigned bytes in xmm3/m128 with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result in xmm1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.NP.0F38.W0 50 /r VPDPBUUD ymm1, ymm2, ymm3/m256	A	V/V	AVX-VNNI-INT8	Multiply groups of 4 pairs of unsigned bytes in ymm3/m256 with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result in ymm1.
VEX.128.NP.0F38.W0 51 /r VPDPBUUDS xmm1, xmm2, xmm3/m128	A	V/V	AVX-VNNI-INT8	Multiply groups of 4 pairs of unsigned bytes in xmm3/m128 with corresponding unsigned bytes of xmm2, summing those products and adding them to the doubleword result, with unsigned saturation in xmm1.
VEX.256.NP.0F38.W0 51 /r VPDPBUUDS ymm1, ymm2, ymm3/m256	A	V/V	AVX-VNNI-INT8	Multiply groups of 4 pairs of unsigned bytes in ymm3/m256 with corresponding unsigned bytes of ymm2, summing those products and adding them to the doubleword result, with unsigned saturation in ymm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

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

For unsigned saturation, when an individual result value is beyond the range of an unsigned doubleword (that is, greater than FFFFF_FFFFH), the saturated unsigned doubleword integer value of FFFF_FFFFH is stored in the doubleword destination.

For signed saturation, when an individual result is beyond the range of a signed doubleword integer (that is, greater than 7FFF_FFFFH or less than 8000_0000H), the saturated value of 7FFF_FFFFH or 8000_0000H, respectively, is written to the destination operand.

Operation

VPDPB[SU,UU,SS]D[,S] dest, src1, src2 (VEX encoded version)

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

ORIGDEST := DEST
FOR i := 0 TO KL-1:

    IF *src1 is signed*:
        src1extend := SIGN_EXTEND // SU, SS
    ELSE:
        src1extend := ZERO_EXTEND // UU
    IF *src2 is signed*:
        src2extend := SIGN_EXTEND // SS
    ELSE:
        src2extend := ZERO_EXTEND // UU, SU

    p1word := src1extend(SRC1.byte[4*i+0]) * src2extend(SRC2.byte[4*i+0])
    p2word := src1extend(SRC1.byte[4*i+1]) * src2extend(SRC2.byte[4*i+1])
    p3word := src1extend(SRC1.byte[4*i+2]) * src2extend(SRC2.byte[4*i+2])
    p4word := src1extend(SRC1.byte[4*i+3]) * src2extend(SRC2.byte[4*i+3])

    IF *saturating*:
        IF *UU instruction version*:
            DEST.dword[i] := UNSIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)
        ELSE:
            DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)
    ELSE:
        DEST.dword[i] := ORIGDEST.dword[i] + p1word + p2word + p3word + p4word

DEST[MAXVL-1:VL] := 0
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

VPMADD52HUQ—Packed Multiply of Unsigned 52-Bit Integers and Add the High 52-Bit Products to Qword Accumulators

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 B5 /r VPMADD52HUQ xmm1, xmm2, xmm3/m128	A	V/V	AVX-IFMA	Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in xmm1.
VEX.256.66.0F38.W1 B5 /r VPMADD52HUQ ymm1, ymm2, ymm3/m256	A	V/V	AVX-IFMA	Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the high 52 bits of the 104-bit product to the qword unsigned integers in ymm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The high 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand).

Operation

VPMADDHUQ srcdest, src1, src2 (VEX version)

```
VL = (128,256)
KL = VL/64

FOR i in 0 .. KL-1:
    temp128 := zeroextend64(src1.qword[i][51:0]) *zeroextend64(src2.qword[i][51:0])
    srcdest.qword[i] := srcdest.qword[i] +zeroextend64(temp128[103:52])
srcdest[MAXVL:VL] := 0
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

VPMADD52LUQ—Packed Multiply of Unsigned 52-Bit Integers and Add the Low 52-Bit Products to Qword Accumulators

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 B4 /r VPMADD52LUQ xmm1, xmm2, xmm3/m128	A	V/V	AVX-IFMA	Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in xmm1.
VEX.256.66.0F38.W1 B4 /r VPMADD52LUQ ymm1, ymm2, ymm3/m256	A	V/V	AVX-IFMA	Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the low 52 bits of the 104-bit product to the qword unsigned integers in ymm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The low 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand).

Operation

VPMADDLUQ srcdest, src1, src2 (VEX version)

```
VL = (128,256)
KL = VL/64

FOR i in 0 .. KL-1:
    temp128 := zeroextend64(src1.qword[i][51:0]) *zeroextend64(src2.qword[i][51:0])
    srcdest.qword[i] := srcdest.qword[i] +zeroextend64(temp128[51:0])
srcdest[MAXVL:VL] := 0
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

WRMSRLIST—Write List of Model Specific Registers

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 01 C6 WRMSRLIST	ZO	V/N.E.	MSRLIST	Write requested list of MSRs with the values specified in memory.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

This instruction writes a software provided list of up to 64 MSRs with values loaded from memory.

WRMSRLIST takes three implied input operands:

- RSI: Linear address of a table of MSR addresses (8 bytes per address).
- RDI: Linear address of a table from which MSR data is loaded (8 bytes per MSR).
- RCX: 64-bit bitmask of valid bits for the MSRs. Bit 0 is the valid bit for entry 0 in each table, etc.

For each RCX bit [n] from 0 to 63, if RCX[n] is 1, WRMSRLIST will write the MSR specified at entry [n] in the RSI table with the value read from memory at the entry [n] in the RDI table.

This implies a maximum of 64 MSRs that can be processed by this instruction. The processor will clear RCX[n] after it finishes handling that MSR. Similar to repeated string operations, WRMSRLIST supports partial completion for interrupts, exceptions, and traps. In these situations, the RIP register saved will point to the MSRLIST instruction while the RCX register will have cleared bits corresponding to all completed iterations.

This instruction must be executed at privilege level 0; otherwise, a general protection exception #GP(0) is generated. This instruction performs MSR specific checks and respects the VMX MSR VM-execution controls in the same manner as WRMSR.

Like WRMSRNS (and unlike WRMSR), WRMSRLIST is not defined as a serializing instruction (see “Serializing Instructions” in Chapter 9 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). This means that software should not rely on WRMSRLIST to drain all buffered writes to memory before the next instruction is fetched and executed. For implementation reasons, some processors may serialize when writing certain MSRs, even though that is not guaranteed.

Like WRMSR and WRMSRNS, WRMSRLIST will ensure that all operations before the WRMSRLIST do not use the new MSR value and that all operations after the WRMSRLIST do use the new value. An exception to this rule is certain store-related performance monitor events that only count when those stores are drained to memory. Since WRMSRLIST is not a serializing instruction, if software is using WRMSRLIST to change the controls for such performance monitor events, then stores before the WRMSRLIST may be counted with new MSR values written by WRMSRLIST. Software can insert the SERIALIZE instruction before the WRMSRLIST if so desired.

Those MSRs that cause a TLB invalidation when they are written via WRMSR (e.g., MTRRs) will also cause the same TLB invalidation when written by WRMSRLIST.

In places where WRMSR is being used as a proxy for a serializing instruction, a different serializing instruction can be used (e.g., SERIALIZE).

WRMSRLIST writes MSRs in order, which means the processor will ensure that an MSR in iteration “n” will be written only after previous iterations (“n-1”). If the older MSR writes had a side effect that affects the behavior of the next MSR, the processor will ensure that side effect is honored.

The processor is allowed to (but not required to) “load ahead” in the list. Examples:

- Use old memory type or TLB translation for loads from list memory despite an MSR written by a previous iteration changing MTRR or invalidating TLBs.
- Cause a page fault or EPT violation for a memory access to an entry > “n” in MSR address or data tables, despite the processor only having read or written “n” MSRs.[1]

Virtualization Behavior—VM Exit Causes

Like WRMSR, the WRMSRLIST instruction executed in VMX non-root operation causes a VM exit if any of the following are true:

- The "use MSR bitmaps" VM-execution control is 0.
- The value of MSR address is not in the ranges 00000000H–00001FFFH and C0000000H–C0001FFFH.
- The value of MSR address is in the range 00000000H–00001FFFH and bit n in read bitmap for low MSRs is 1, where n is the value of the MSR address.
- The value of ECX is in the range C0000000H–C0001FFFH and bit n in read bitmap for high MSRs is 1, where n is the value of the MSR address & 00001FFFH.

A VM exit for the above reasons for the WRMSRLIST instruction will specify exit reason 79 (decimal). The exit qualification is set to the MSR address causing the VM exit if "use MSR bitmaps" VM-execution control is 1. If "use MSR bitmaps" VM-execution control is 0, then the VM-exit qualification will be 0.

If software wants to emulate a single iteration of WRMSRLIST after a VM exit, it can use the exit qualification to identify the MSR. Such software will need to read from the table of data. It can calculate the guest-linear address of the table entry to read by using the values of RDI (the guest-linear address of the table) and RCX (the lowest bit set in RCX identifies the specific table entry).

Virtualization Behavior—Changed Behavior in Non-Root Operation

The previous section identifies when executions of the WRMSRLIST instruction cause VM exits. Under the following situations, a #UD will occur instead of a VM exit or a fault due to CPL 0:

- The "Enable MSRLIST Instructions" VM-execution control is 0.
- The "Activate tertiary controls" VM-execution control is 0.

If that does not occur and there is no fault due to CPL > 0 nor a VM exit, the instruction's behavior may be modified for certain values of MSR address in the same manner as WRMSR for a read of the same MSR.

Operation

```
WHILE (RCX != 0) {
    MSR_index = TZCNT(RCX)
    MSR_address = mem[RSI + (MSR_index * 8) ]
    MSR_data = mem[RDI + (MSR_index * 8)]
    VM exit if specified by VM-execution controls (for specified MSR_address)
    #GP(0) if MSR_address[61:32] !=0
    #GP(0) if MSR_address is not accessible for WRMSR
    #GP(0) if MSR_data has reserved bits set for MSR
    #GP(0) for any other MSR_address specific checks
    WRMSRNS (MSR_address) = MSR_data
    Clear RCX [MSR_index]
    Take any pending interrupts/traps
}
```

Flags Affected

None.

[1] For example, the processor may take a page fault due to a linear address for the 10th entry in the MSR address table despite only having completed the MSR writes up to entry 5.

Protected Mode Exceptions

#UD The WRMSRLIST instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD The WRMSRLIST instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

#UD The WRMSRLIST instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#GP(0) If the current privilege level is not 0.

If RSI [2:0] $\neq$ 0 OR RDI [2:0] $\neq$ 0.

If an execution of WRMSR to a specified MSR with a specified value would generate a general-protection exception (#GP(0)).

#UD If the LOCK prefix is used.

If not in 64-bit mode.

If CPUID.(EAX=07H, ECX=01H):EAX.MSRLIST[bit 27] = 0.

WRMSRNS—Non-Serializing Write to Model Specific Register

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 C6 WRMSRNS	ZO	V/V	WRMSRNS	Write the value in EDX:EAX to MSR specified by ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A

Description

WRMSNRS is an instruction that behaves exactly like WRMSR, with the only difference being that it is not a serializing instruction by default.

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. The contents of the EDX register are copied to the high-order 32 bits of the selected MSR and the contents of the EAX register are copied to the low-order 32 bits of the MSR. The high-order 32 bits of RAX, RCX, and RDX are ignored.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated.

Unlike WRMSR, WRMSRNS is not defined as a serializing instruction (see “Serializing Instructions” in Chapter 9 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). This means that software should not rely on it to drain all buffered writes to memory before the next instruction is fetched and executed. For implementation reasons, some processors may serialize when writing certain MSRs, even though that is not guaranteed.

Like WRMSR, WRMSRNS will ensure that all operations before it do not use the new MSR value and that all operations after the WRMSRNS do use the new value. An exception to this rule is certain store related performance monitor events that only count when those stores are drained to memory. Since WRMSRNS is not a serializing instruction, if software is using WRMSRNS to change the controls for such performance monitor events, then stores before the WRMSRMS may be counted with new MSR values written by WRMSRNS. Software can insert the SERIALIZE instruction before the WRMSRNS if so desired.

Those MSRs that cause a TLB invalidation when they are written via WRMSR (e.g., MTRRs) will also cause the same TLB invalidation when written by WRMSRNS.

In order to improve performance, software may replace WRMSR with WRMSRNS. In places where WRMSR is being used as a proxy for a serializing instruction, a different serializing instruction can be used (e.g., SERIALIZE).

Operation

```
MSR[ECX] := EDX:EAX;
```

Flags Affected

None.

CHAPTER 3
INTEL® AMX INSTRUCTION SET REFERENCE, A-Z

NOTES

The following Intel® AMX instructions have moved to the Intel® 64 and IA-32 Architectures Software Developer's Manual: LDTILECFG, STTILECFG, TDPBF16PS, TDPBSSD/TDPBSUD/TDPBUSD/TDPBUUD, TILELOADD/TILELOADDT1, TILERELEASE, TILESTORED, and TILEZERO.

The Intel Advanced Matrix Extensions introductory material and helper functions will be maintained here, as well as in the Intel® 64 and IA-32 Architectures Software Developer's Manual, for the reader's convenience. For information on Intel AMX and the XSAVE feature set, and recommendations for system software, see the latest version of the Intel® 64 and IA-32 Architectures Software Developer's Manual.

3.1 INTRODUCTION

Intel® Advanced Matrix Extensions (Intel® AMX) is a new 64-bit programming paradigm consisting of two components: a set of 2-dimensional registers (tiles) representing sub-arrays from a larger 2-dimensional memory image, and an accelerator able to operate on tiles, the first implementation is called TMUL (tile matrix multiply unit).

An Intel AMX implementation enumerates to the programmer how the tiles can be programmed by providing a palette of options. Two palettes are supported; palette 0 represents the initialized state, and palette 1 consists of 8 KB of storage spread across 8 tile registers named TMM0..TMM7. Each tile has a maximum size of 16 rows x 64 bytes, (1 KB), however the programmer can configure each tile to smaller dimensions appropriate to their algorithm. The tile dimensions supplied by the programmer (rows and bytes_per_row, i.e., **colsb**) are metadata that drives the execution of tile and accelerator instructions. In this way, a single instruction can launch autonomous multi-cycle execution in the tile and accelerator hardware. The palette value (**palette_id**) and metadata are held internally in a tile related control register (TILECFG). The TILECFG contents will be commensurate with that reported in the palette_table (see "CPUID—CPU Identification" in Chapter 1 for a description of the available parameters).

Intel AMX is an extensible architecture. New accelerators can be added, or the TMUL accelerator may be enhanced to provide higher performance. In these cases, the state (TILEDATA) provided by tiles may need to be made larger, either in one of the metadata dimensions (more rows or colsb) and/or by supporting more tile registers (names). The extensibility is carried out by adding new palette entries describing the additional state. Since execution is driven through metadata, an existing Intel AMX binary could take advantage of larger storage sizes and higher performance TMUL units by selecting the most powerful palette indicated by CPUID and adjusting loop and pointer updates accordingly.

Figure 3-1 shows a conceptual diagram of the Intel AMX architecture. An Intel architecture host drives the algorithm, the memory blocking, loop indices and pointer arithmetic. Tile loads and stores and accelerator commands are sent to multi-cycle execution units. Status, if required, is reported back. Intel AMX instructions are synchronous in the Intel architecture instruction stream and the memory loaded and stored by the tile instructions is coherent with respect to the host's memory accesses. There are no restrictions on interleaving of Intel architecture and Intel AMX code or restrictions on the resources the host can use in parallel with Intel AMX (e.g., Intel AVX-512). There is also no architectural requirement on the Intel architecture compute capability of the Intel architecture host other than it supports 64-bit mode.

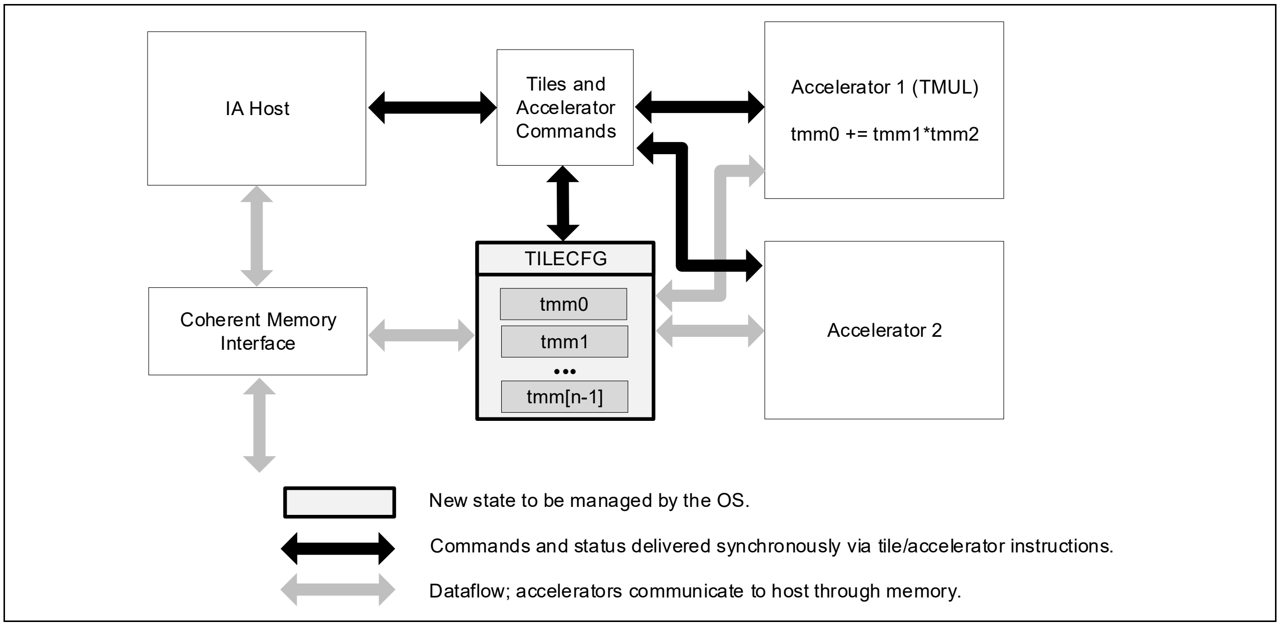

IA Host
Tiles and Accelerator Commands
Accelerator 1 (TMUL)
tmm0 += tmm1*tmm2
TILECFG
tmm0
tmm1
•••
tmm[n-1]
Coherent Memory Interface
Accelerator 2
New state to be managed by the OS.
Commands and status delivered synchronously via tile/accelerator instructions.
Dataflow; accelerators communicate to host through memory.

Figure 3-1. Intel® AMX Architecture

Intel AMX instructions use new registers and inherit basic behavior from Intel architecture in the same manner that Intel SSE and Intel AVX did. Tile instructions include loads and stores using the traditional Intel architecture register set as pointers. The TMUL instruction set (defined to be CPUID bits AMX-BF16 and AMX-INT8) only supports reg-reg operations.

TILECFG is programmed using the LDTILECFG instruction. The selected palette defines the available storage and general configuration while the rest of the memory data specifies the number of rows and column bytes for each tile. Consistency checks are performed to ensure the TILECFG matches the restrictions of the palette. A General Protection fault (#GP) is reported if the LDTILECFG fails consistency checks. A successful load of TILECFG with a palette_id other than 0 is represented in this document with TILES_CONFIGURED = 1. When the TILECFG is initialized (palette_id = 0), it is represented in the document as TILES_CONFIGURED = 0. Nearly all Intel AMX instructions will generate a #UD exception if TILES_CONFIGURED is not equal to 1; the exceptions are those that do TILECFG maintenance: LDTILECFG, STTILECFG and TILERELEASE.

If a tile is configured to contain M rows by N column bytes, LDTILECFG will ensure that the metadata values are appropriate to the palette (e.g., that M ≤ 16 and N ≤ 64 for palette 1). The four M and N values can all be different as long as they adhere to the restrictions of the palette. Further dynamic checks are done in the tile and the TMUL instruction set to deal with cases where a legally configured tile may be inappropriate for the instruction operation. Tile registers can be set to ‘invalid’ by configuring the rows and colsb to ‘0’.

Tile loads and stores are strided accesses from the application memory to packed rows of data. Algorithms are expressed assuming row major data layout. Column major users should translate the terms according to their orientation.

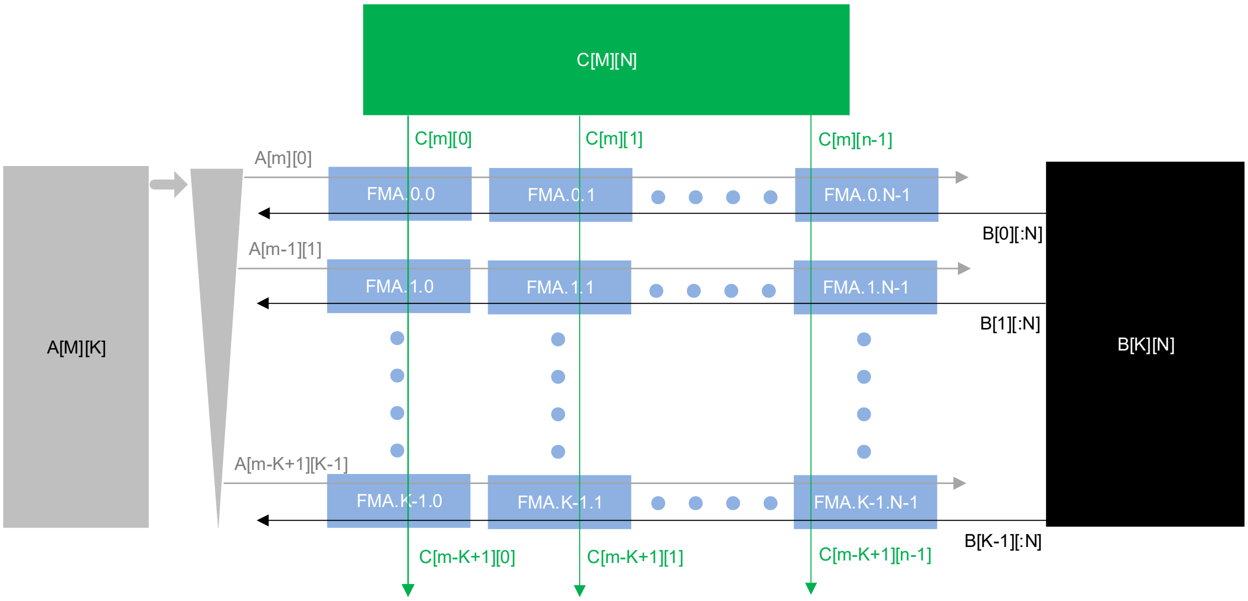
TILELOAD* and TILESTORE* instructions are restartable and can handle (up to) 2*rows page faults per instruction. Restartability is provided by a **start_row** parameter in the TILECFG register.

The TMUL unit is conceptually a grid of fused multiply-add units able to read and write tiles. The dimensions of the TMUL unit (tmul_maxk and tmul_maxn) are enumerated similar to the maximum dimensions of the tiles (see “CPUID—CPU Identification” in Chapter 1 for details).

The matrix multiplications in the TMUL instruction set compute C[M][N] += A[M][K] * B[K][N]. The M, N, and K values will cause the TMUL instruction set to generate a #UD exception if the dimensions do not match for matrix multiply or do not match the palette.

In Figure 3-2, the number of rows in tile B matches the K dimension in the matrix multiplication pseudocode. K dimensions smaller than that enumerated in the TMUL grid are also possible and any additional computation the TMUL unit can support will not affect the result.

The number of elements specified by colsb of the B matrix is also less than or equal to tmul_maxn. Any remaining values beyond that specified by the metadata will be set to zero.



```
for m < M:          // time steps
  for k < K:        // grid height
    for n < N:      // SIMD dimension
      C[m][n] += VNNI_MUL(A[m][k], B[k][n])
```

Figure 3-2. The TMUL Unit

The XSAVE feature sets supports context management of the new state defined for Intel AMX. This support is described in Section 3.2.

3.1.1 Tile Architecture Details

The supported parameters for the tile architecture are reported via CPUID; this includes information about how the number of tile registers (max_names) can be configured (the palette). Configuring the tile architecture is intended to be done once when entering a region of tile code using the LDTILECFG instruction specifying the selected palette and describing in detail the configuration for each tile. Incorrect assignments will result in a General Protection fault (#GP). Successful LDTILECFG initializes (zeroes) TILEDATA.

Exiting a tile region is done with the TILERELEASE instruction. It takes no parameters and invalidates all tiles (indicating that the data no longer needs any saving or restoring). Essentially, it is an optimization of LDTILECFG with an implicit palette of 0.

For applications that execute consecutive Intel AMX regions with differing configurations, TILERELEASE is not required between them since the second LDTILECFG will clear all the data while loading the new configuration. There is no instruction set support for automatic nesting of tile regions, though with sufficient effort software can accomplish this by saving and restoring TILEDATA and TILECFG either through the XSAVE architecture or the Intel AMX instructions.

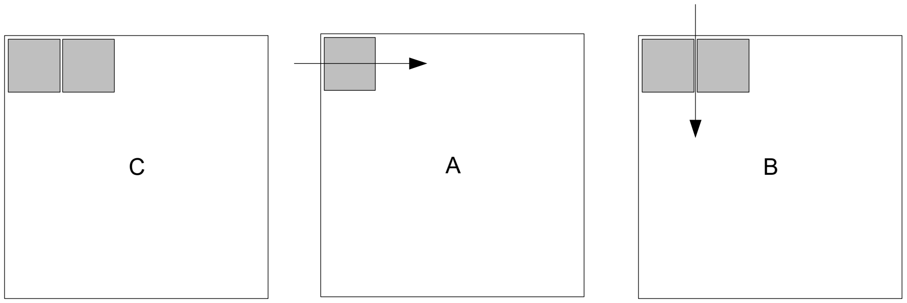
The tile architecture boots in its INIT state, with TILECFG and TILEDATA set to zero. A successfully executing LDTILECFG instruction to a non-zero palette sets the TILES_CONFIGURED=1, indicating the TILECFG is not in the INIT state. The TILERELEASE instruction sets TILES_CONFIGURED = 0 and initializes (zeroes) TILEDATA.

To facilitate handling of tile configuration data, there is a STTILECFG instruction. If the tile configuration is in the INIT state (TILES_CONFIGURED == 0), then STTILECFG will write 64 bytes of zeros. Otherwise STTILECFG will store the TILECFG to memory in the format used by LDTILECFG.

3.1.2 TMUL Architecture Details

The supported parameters for the TMUL architecture are reported via CPUID; see "CPUID—CPU Identification" in Chapter 1, page 1-23, for details. These parameters include a maximum height (**tmul_maxk**) and a maximum SIMD dimension (**tmul_maxn**). The metadata that accompanies the srcdest, src1 and src2 tiles to the TMUL unit will be dynamically checked to see that they match the TMUL unit support for the data type and match the requirements of a meaningful matrix multiplication.

Figure 3-3 shows an example of the inner loop of an algorithm of using the TMUL architecture to compute a matrix multiplication. In this example, we use two result tiles, tmm0 and tmm1, from matrix C to accumulate the intermediate results. One tile from the A matrix (tmm2) is re-used twice as we multiply it by two tiles from the B matrix. The algorithm then advances pointers to load a new A tile and two new B tiles from the directions indicated by the arrows. An outer loop, not shown, adjusts the pointers for the C tiles.



```
   LDTILECFG [rax]
// assume some outer loops driving the cache tiling (not shown)
{
   TILELOADD tmm0, [rsi+rdi]     // srcdst, RSI points to C, RDI is strided value
   TILELOADD tmm1, [rsi+rdi+N]   // second tile of C, unrolling in SIMD dimension N
   MOV r14, 0
LOOP:
   TILELOADD tmm2, [r8+r9]       // src2 is strided load of A, reused for 2 TMUL instr.
   TILELOADD tmm3, [r10+r11]     // src1 is strided load of B
   TDPBUSD   tmm0, tmm2, tmm3  // update left tile of C
   TILELOADD tmm3, [r10+r11+N]   // src1 loaded with B from next rightmost tile
   TDPBUSD   tmm1, tmm2, tmm3  // update right tile of C
   ADD r8, K                // update pointers by constants known outside of loop
   ADD r10, K*r11
   ADD r14, K
   CMP r14, LIMIT
   JNE LOOP

   TILESTORED [rsi+rdi],   tmm0  // update the C matrix in memory
   TILESTORED [rsi+rdi+M], tmm1
 } // end of outer loop

   TILERELEASE        // return tiles to INIT state
```

Figure 3-3. Matrix Multiply C+= A*B

3.1.3 Handling of Tile Row and Column Limits

Intel AMX operations will zero any rows and any columns beyond the dimensions specified by TILECFG. Tile operations will zero the data beyond the configured number of column bytes as each row is written. For example, with 64-byte rows and a tile configured with 10 rows and 48 columns, an operation writing dword elements would write each of the first 10 rows with 48 bytes of output/result data and zero the remaining 16 bytes in each row. Tile operations also fully zero any rows after the first 10 configured rows. When using a 1 KByte tile with 64-byte rows, there would be 16 rows, so in this example, the last 6 rows would also be zeroed.

Intel AMX instructions will always obey the metadata on reads and the zeroing rules on writes, and so a subsequent XSAVE would see zeros in the appropriate locations. Tiles that are not written by Intel AMX instructions between XRSTOR and XSAVE will write back with the same image they were loaded with regardless of the value of TILECFG.

3.1.4 Exceptions and Interrupts

Tile instructions are restartable so that operations that access strided memory can restart after page faults. To support restarting instructions after these events, the instructions store information in the **TILECFG.start_row** register. TILECFG.start_row indicates the row that should be used for restart; i.e., it indicates **next row after** the rows that have already been successfully loaded (on a TILELOAD) or written to memory (on a TILESTORE) and prevents repeating work that was successfully done.

The TMUL instruction set is not sensitive to the TILECFG.start_row value; this is due to there not being TMUL instructions with memory operands or any restartable faults.

3.2 OPERAND RESTRICTIONS

Floating-point exceptions, denormal handling, and floating-point rounding: some of the Intel AMX instructions operate on floating-point values. These instructions all function as if floating-point exceptions are masked, and use the round-to-nearest-even (RNE) rounding mode. They also do not set any of the floating-point exception flags in MXCSR. Table 3-1 describes the treatment of denormal inputs and outputs for Intel AMX operations.

Table 3-1. Intel® AMX Treatment of Denormal Inputs and Outputs

Data Type	Denormal Input	Denormal Output
FP16	Allowed	N/A
FP32	Treated as zero	Flushed to zero
BF16	Treated as zero	N/A

3.3 IMPLEMENTATION PARAMETERS

The parameters are reported via CPUID leaf 1DH. Index 0 reports all zeros for all fields.

```
define palette_table[id]:
    uint16_t total_tile_bytes
    uint16_t bytes_per_tile
    uint16_t bytes_per_row
    uint16_t max_names
    uint16_t max_rows
```

The tile parameters are set by LDTILECFG or XRSTOR* of TILECFG:

```
define tile[tid]:
    byte rows
    word colsb // bytes_per_row
    bool valid
```

3.4 HELPER FUNCTIONS

The helper functions used in Intel AMX instructions are defined below.

```
define write_row_and_zero(treg, r, data, nbytes):
    for j in 0 ...nbytes-1:
        treg.row[r].byte[j] := data.byte[j]

    // zero the rest of the row
    for j in nbytes ... palette_table[tilecfg.palette_id].bytes_per_row-1:
        treg.row[r].byte[j] := 0
```

```
define zero_upper_rows(treg, r):
    for i in r ... palette_table[tilecfg.palette_id].max_rows-1:
        for j in 0 ... palette_table[tilecfg.palette_id].bytes_per_row-1:
                treg.row[i].byte[j] := 0
```

```
define zero_tilecfg_start():
    tilecfg.start_row :=0
```

```
define zero_all_tile_data():
    if XCR0[TILEDATA]:
        b := CPUID(0xD,TILEDATA).EAX // size of feature
        for j in 0 ...b:
                TILEDATA.byte[j] := 0
```

```
define xcr0_supports_palette(palette_id):
   if palette_id == 0:
      return 1
   elif palette_id == 1:
      if XCR0[TILECFG] and XCR0[TILEDATA]:
         return 1
   return 0
```

3.5 NOTATION

Instructions described in this chapter follow the general documentation convention established in *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A*. Additionally, Intel® Advanced Matrix Extensions use notation conventions as described below.

In the instruction encoding boxes, **sibmem** is used to denote an encoding where a MODRM byte and SIB byte are used to indicate a memory operation where the base and displacement are used to point to memory, and the index register (if present) is used to denote a stride between memory rows. The index register is scaled by the sib.scale field as usual. The base register is added to the displacement, if present.

In the instruction encoding, the MODRM byte is represented several ways depending on the role it plays. The MODRM byte has 3 fields: 2-bit MODRM.MOD field, a 3-bit MODRM.REG field and a 3-bit MODRM.RM field. When all bits of the MODRM byte have fixed values for an instruction, the 2-hex nibble value of that byte is presented after the opcode in the encoding boxes on the instruction description pages. When only some fields of the MODRM byte must contain fixed values, those values are specified as follows:

- If only the MODRM.MOD must be 0b11, and MODRM.REG and MODRM.RM fields are unrestricted, this is denoted as **11:rrr:bbb**. The **rrr** correspond to the 3-bits of the MODRM.REG field and the **bbb** correspond to the 3-bits of the MODMR.RM field.
- If the MODRM.MOD field is constrained to be a value other than 0b11, i.e., it must be one of 0b00, 0b01, or 0b10, then we use the notation !(11).
- If the MODRM.REG field had a specific required value, e.g., 0b101, that would be denoted as mm:101:bbb.

NOTE

Historically the Intel® 64 and IA-32 Architectures Software Developer's Manual only specified the MODRM.REG field restrictions with the notation /0 ... /7 and did not specify restrictions on the MODRM.MOD and MODRM.RM fields in the encoding boxes.

3.6 EXCEPTION CLASSES

Alignment exceptions: The Intel AMX instructions that access memory will never generate #AC exceptions.

Table 3-2. Intel® AMX Exception Classes

Class	Description
AMX-E1	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if VVVV ≠ 0b1111.
	• #GP based on palette and configuration checks (see pseudocode). • #GP if the memory address is in a non-canonical form.
	• #SS(0) if the memory address referencing the SS segment is in a non-canonical form.
	• #PF if a page fault occurs.
AMX-E2	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if VVVV ≠ 0b1111.
	• #GP if the memory address is in a non-canonical form.
	• #SS(0) if the memory address referencing the SS segment is in a non-canonical form.
	• #PF if a page fault occurs.
AMX-E3	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if VVVV ≠ 0b1111. • #UD if not using SIB addressing. • #UD if TILES_CONFIGURED == 0. • #UD if tsrc or tdest are not valid tiles. • #UD if tsrc/tdest are ≥ palette_table[tilecfg.palette_id].max_names. • #UD if tsrc.colbytes mod 4 ≠ 0 OR tdest.colbytes mod 4 ≠ 0. • #UD if tilecfg.start_row ≥ tsrc.rows OR tilecfg.start_row ≥ tdest.rows.
	• #GP if the memory address is in a non-canonical form.
	• #SS(0) if the memory address referencing the SS segment is in a non-canonical form.
	• #PF if any memory operand causes a page fault.
	• #NM if XFD[18] == 1.

Table 3-2. Intel® AMX Exception Classes (Continued)

Class	Description
AMX-E4	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if srcdest == src1 OR src1 == src2 OR srcdest == src2. • #UD if TILES_CONFIGURED == 0. • #UD if srcdest.colbytes mod 4 ≠ 0. • #UD if src1.colbytes mod 4 ≠ 0. • #UD if src2.colbytes mod 4 ≠ 0. • #UD if srcdest/src1/src2 are not valid tiles. • #UD if srcdest/src1/src2 are ≥ palette_table[tilecfg.palette_id].max_names. • #UD if srcdest.colbytes ≠ src2.colbytes. • #UD if srcdest.rows ≠ src1.rows. • #UD if src1.colbytes / 4 ≠ src2.rows. • #UD if srcdest.colbytes > tmul_maxn. • #UD if src2.colbytes > tmul_maxn. • #UD if src1.colbytes/4 > tmul_maxk. • #UD if src2.rows > tmul_maxk.
	• #NM if XFD[18] == 1.
AMX-E5	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if VVVV ≠ 0b1111. • #UD if TILES_CONFIGURED == 0. • #UD if tdest is not a valid tile. • #UD if tdest is ≥ palette_table[tilecfg.palette_id].max_names.
	• #NM if XFD[18] == 1.
AMX-E6	• #UD if preceded by LOCK, 66H, F2H, F3H or REX prefixes. • #UD if CR4.OSXSAVE ≠ 1. • #UD if XCR0[18:17] ≠ 0b11. • #UD if IA32_EFER.LMA ≠ 1 OR CS.L ≠ 1. • #UD if VVVV ≠ 0b1111.

3.7 INSTRUCTION SET REFERENCE

TDPFP16PS—Dot Product of FP16 Tiles Accumulated into Packed Single Precision Tile

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F2.0F38.W0 5C 11:rrr:bbb TDPFP16PS tmm1, tmm2, tmm3	A	V/N.E.	AMX-FP16	Matrix multiply FP16 elements from tmm2 and tmm3, and accumulate the packed single precision elements in tmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:reg (r, w)	ModRM:r/m (r)	VEX.vvvv (r)	N/A

Description

This instruction performs a set of SIMD dot-products of two FP16 elements and accumulates the results into a packed single precision tile. Each dword element in input tiles tmm2 and tmm3 is interpreted as a FP16 pair. For each possible combination of (row of tmm2, column of tmm3), the instruction performs a set of SIMD dot-products on all corresponding FP16 pairs (one pair from tmm2 and one pair from tmm3), adds the results of those dot-products, and then accumulates the result into the corresponding row and column of tmm1.

“Round to nearest even” rounding mode is used when doing each accumulation of the Fused Multiply-Add (FMA). Output FP32 denormals are always flushed to zero. Input FP16 denormals are always handled and not treated as zero.

MXCSR is not consulted nor updated.

Any attempt to execute the TDPFP16PS instruction inside an Intel TSX transaction will result in a transaction abort.

Operation

TDPFP16PS tsrcdest, tsrc1, tsrc2

```
// C = m x n (tsrcdest), A = m x k (tsrc1), B = k x n (tsrc2)

# src1 and src2 elements are pairs of fp16
elements_src1 := tsrc1.colsb / 4
elements_src2 := tsrc2.colsb / 4
elements_dest := tsrcdest.colsb / 4
elements_temp := tsrcdest.colsb / 2 // Count is in fp16 prior to horizontal

for m in 0 … tsrcdest.rows-1:
    temp1[ 0 … elements_temp-1 ] := 0
    for k in 0 … elements_src1-1:
        for n in 0 … elements_dest-1:

        // For this operation:
        // Handle FP16 denorms. Not forcing input FP16 denorms to 0.
        // FP32 FMA with DAZ=FTZ=1, RNE rounding.
        // MXCSR is neither consulted nor updated.
        // No exceptions raised or denoted.

        temp1.fp32[2*n+0] += cvt_fp16_to_fp32(tsrc1.row[m].fp16[2*k+0]) *cvt_fp16_to_fp32(tsrc2.row[k].fp16[2*n+0])
        temp1.fp32[2*n+1] += cvt_fp16_to_fp32(tsrc1.row[m].fp16[2*k+1]) *cvt_fp16_to_fp32(tsrc2.row[k].fp16[2*n+1])

    for n in 0 … elements_dest-1:
        // DAZ=FTZ=1, RNE rounding.
        // MXCSR is neither consulted nor updated.
```

```
            // No exceptions raised or denoted.
            tmpf32 := temp1.fp32[2*n] + temp1.fp32[2*n+1]
            srcdest.row[m].fp32[n] := srcdest.row[m].fp32[n] + tmpf32
    write_row_and_zero(tsrcdest, m, tmp, tsrcdest.colsb)
zero_upper_rows(tsrcdest, tsrcdest.rows)
zero_tileconfig_start()
```

Flags Affected

None.

Exceptions

AMX-E4; see Section 3.6, “Exception Classes” for details.

CHAPTER 4
NON-WRITE-BACK LOCK DISABLE ARCHITECTURE

Locked read-modify-write (RMW) to a memory operation is used explicitly by several Intel architecture set instructions, such as ADD with a lock prefix, and implicitly by other instructions and flows, such as updating a segment access bit or page tables access/dirty bits.

Locked RMW access is usually handled through processor cache in the lower hierarchies, and it only impacts software running on same logical processors that share this cache.

If the memory type of this locked RMW is not write-back, the processor can't handle it within the internal cache and will issue a bus lock operation. This operation will block all logical processors and devices from accessing memory until the operation has completed.

Having a burst of bus locks by one of the logical processors may cause starvation to the rest of the logical processors and devices.

This architecture will allow software to disable non-WB lock operation. Once the feature is enabled, performing a non-WB lock operation by software will generate a general protection fault (#GP).

4.1 ENUMERATION

The non-write-back lock disable capability will be enumerated through a model-specific bit (bit 4) in the IA32_CORE_CAPABILITIES MSR.

Table 4-1. IA32_CORE_CAPABILITIES MSR

Register Address		Architectural MSR Name / Bit Fields	Description
Hex	Decimal		
CFH	207	IA32_CORE_CAPABILITIES	IA32 Core Capability Register
		3:0	Reserved
		4	UC_LOCK_DISABLE_SUPPORTED If 1, software can set bit 28 of MSR_MEMORY_CTRL (MSR address 33H) to cause a #GP when a locked access to UC memory would occur. This behavior is consistent across processor models.
		5	SPLIT_LOCK_DISABLE_SUPPORTED If 1, software can set bit 29 of MSR_MEMORY_CTRL to cause an #AC when a locked access that includes multiple cache lines would occur. This behavior is consistent across processor models.
		63:6	Reserved

4.2 ENABLING

This model-specific feature will add an MSR control bit (bit 28) in MSR_MEMORY_CTRL (MSR address 33H) in order to generate a general protection fault (#GP) each time a non-WB load lock is detected.

Table 4-2. MEMORY_CTRL MSR

Register Address		Architectural MSR Name / Bit Fields	Description
Hex	Decimal		
33H	51	MSR_MEMORY_CTRL	Memory Control Register
		27:0	Reserved
		28	Enable #GP(0) exception for non-write-back locked accesses.
		29	Enable #AC(0) exception for split locked accesses.
		31:30	Reserved

4.3 INTERACTION WITH INTEL® SOFTWARE GUARD EXTENSIONS (INTEL® SGX)

Processor Reserved Memory (PRM) used for Intel® SGX can run with non-WB memory accesses by following the steps below.

1. Configure the Memory Type field (bits 2:0) of MSR_PRMRR_BASE_0 (MSR address 2A0H) to be non-WB.
2. Set the cache disable (bit 30) of CR0.

When the processor is configured in this manner, the processor will not generate #GP(0) as a result of locked accesses to non-WB memory when EPT is enabled, even if the non-WB lock disable (bit 28) of MSR_MEMORY_CTRL is set to 1.

4.4 INTERACTION WITH VMX ARCHITECTURE

There are two cases where a locked cycle can be issued on a VMM configuration with non-WB memory type.

1. VMM enabled EPT and EPT A/D and configured EPT memory type to non-WB. In this case, EPT A/D assist will issue a locked load to non-WB memory.
2. VMM set “process posted interrupts” VM-execution control, posted-interrupt descriptor mapped to non-WB memory type. Posted interrupt processing will update the descriptor with locked load to non-WB memory.

When the processor is configured in this manner, the processor will not generate #GP(0) as a result of a locked access to non-WB memory when EPT is enabled even if the non-WB lock disable (bit 28) of MSR_MEMORY_CTRL is set to 1.

4.5 EXPECTED SOFTWARE BEHAVIOR

Software can ensure that bus locks as a result of non-WB locked access are never taken, or at least a general protection fault is signaled, by performing the following operations:

- Set Non-WB Lock Disable (bit 28) of MSR_MEMORY_CTRL.
- Do not set Cache Disable (bit 30) of CR0.
- Configure MSR_PRMRR_BASE_0 Memory Type field (bits 2:0) to WB memory type only.
- For a VMM that enabled EPT and EPT A/D, bits must configure EPT paging structures to WB memory type.
- For a VMM that enabled posted-interrupt via the “process posted interrupts” VM-execution control, ensure the posted-interrupt descriptor is mapped to WB memory type.

4.6 BUS LOCKS

Cases for bus locks than can come from non-WB Lock operation are shown in Table 4-3.

Table 4-3. Bus Locks from Non-WB Operation

Category	Instructions/Flows	Conditions
Arithmetic	LOCK + {ADD, SUB, AND, OR, XOR, ADC, SBB, INC, DEC, NOT, NEG}	
Compare/Test	LOCK + {BTC, BTR, BTS}	
Exchange	XCHG, LOCK XADD/CMPXCHG/XCHG	
Segmentation	LSL, LAR, VERR, VERW LDS, LES, LFS, LGS, LSS MOV DS, MOV ES, MOV FS, MOV GS, MOV SS POP DS, POP ES, POP FS, POP GS, POP SS	Setting segment accessed bit in descriptor in non-WB memory.
Call / Interrupt / Exception	Far call, Far JMP Far RET, IRET INTn, INT3, INTO, INT1 Call through interrupt/trap gate	Setting segment accessed bit in descriptor in non-WB memory.
Tasking	LTR, Task Switch	Setting/Clearing TSS busy when TSS in non-WB memory. Setting segment accessed bit in descriptor in non-WB memory.
Paging	Code fetch (A bit update), All instructions that have memory operands (A/D bits update)	Page tables in non-WB memory.
Enclave	ENCLU, ENCLS, AEX	
Posted Interrupts	Updating the posted interrupt descriptor uses locked RMW for atomic operations.	Posted interrupt descriptor in non-WB memory.

CHAPTER 5
BUS LOCK AND VM NOTIFY

5.1 BUS LOCK DEBUG EXCEPTION

A logical processor can be configured to generate a debug exception (#DB) as a trap delivered in the instruction boundary following acquisition of a bus lock if the processor is at privilege level > 0 on this instruction boundary. Software enables these debug exceptions by setting bit 2 of the IA32_DEBUGCTL MSR. The CPU enumerates support for the 1-setting of this bit using CPUID.(EAX=7, ECX=0).ECX[24].

A debug exception due to acquisition of a bus lock is reported as a trap following execution of the instruction acquiring the bus lock if the privilege level is > 0. The processor identifies such debug exceptions using bit 11 of DR6. Because DR6[11] has formerly always been 1, delivery of a bus-lock #DB **clears** DR6[11]. All other debug exceptions leave bit 11 unmodified. To avoid confusion in identifying debug exceptions, software debug-exception handlers should set bit 11 to 1 before returning to the interrupted task.

A VM exit sets bit 11 of the pending debug exception field in the guest-state area of the VMCS to indicate that a bus lock debug exception was pending but not delivered. A VM exit that sets this bit also sets bit 12 of that field. (VM exits also sets bit 12 to indicate that at least one data or I/O breakpoint was met and was enabled in DR7, or that a debug exception related to advanced debugging of RTM transactional regions occurred.)

5.1.1 Bus Lock VM Exit

A new VM-execution control, “bus-lock detection,” can be used to cause VM exits on bus locks acquired in VMX non-root operation.

If the “bus-lock detection” VM-execution control is 1, there will be a VM exit following any operation that causes a bus lock. (The VM exit is thus trap-like and does not prevent the bus lock from occurring.) The VM exit uses basic exit reason 74, storing this value in bits 15:0 of the exit-reason field in the VMCS.

An operation may cause a bus lock and then incur a VM exit for some other reason. If this happens, the other VM exit is delivered normally and no bus-lock VM exit (with basic exit reason 74) occurs.

In either case, any VM exit following an operation that caused a bus lock will also set bit 26 of the exit-reason field to indicate that a bus lock had occurred. (The bit is set only if the “bus-lock detection” VM-execution control is 1.)

“Bus-lock detection” is secondary processor-based execution control bit 30. A processor enumerates support for the 1-setting of this control by setting bit 62 of the IA32_VMX_PROCBASED_CTLS2 MSR.

5.2 NOTIFY VM EXIT

A VMM can enable **notification VM exits to occur** if no interrupt windows occur in VMX non-root operation for a specified amount of time (**notify window**). These VM exits are enabled by setting bit 31 of the secondary processor-based execution control. A processor enumerates support for the 1-setting of this control by setting bit 63 of the IA32_VMX_PROCBASED_CTLS2 MSR. The VMM configures the notify window in units of crystal clock cycles in a new 32-bit VM-execution control field in the VMCS (notify window) that can be accessed with the VMREAD and VMWRITE instructions using encoding 00004024H.

A notification VM exit reports basic exit reason 75 and exit qualification determined as follows:

- Bit 0 - VM context invalid.
- Bits 11:1 are reserved.
- Bit 12 - if set the VM exit was incident to an execution of IRET that unblocked NMIs.
- All other bits are reserved.

If the VMM-notify VM exit occurred incident to delivery of a vectored event, then IDT vectoring information and applicable error code are recorded in the VMCS.

CHAPTER 6 INTEL® RESOURCE DIRECTOR TECHNOLOGY FEATURE UPDATES

Intel® Resource Director Technology (Intel® RDT) provides a number of monitoring and control capabilities for shared resources in multiprocessor systems. This chapter covers updates to the feature that will be available in future Intel processors, starting with brief descriptions followed by technical details.

6.1 INTEL® RDT FEATURE CHANGES

6.1.1 Intel® RDT on the 3rd generation Intel® Xeon® Scalable Processor Family

The 3rd generation Intel® Xeon® Scalable Processor Family based on Ice Lake Server microarchitecture adds the following Intel RDT enhancements:

- 32-bit MBM counters (vs. 24-bit in prior generations), and new CPUID enumeration capabilities for counter width.
- Second generation Memory Bandwidth Allocation (MBA): Introduces an advanced hardware feedback controller that operates at microsecond timescales, and software-selectable min/max throttling value resolution capabilities. Baseline descriptions of the MBA "throttling values" applied to the threads running on a core are described in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

  Second generation MBA capabilities also add a work-conserving feature in which applications that frequently access the L3 cache may be throttled by a lesser amount until they exceed the user-specified memory bandwidth usage threshold, enhancing system throughput and efficiency, in addition to adding more precise calibration and controls. Certain BIOS implementations may further aid flexibility by providing selectable calibration profiles for various usages.
- 15 MBA / L3 CAT CLOS: Improved feature consistency and interface flexibility. The previous generation of processors supported 16 L3 CAT Class of Service tags (CLOS), but only 8 MBA CLOS. The changes in enumerated CLOS counts per-feature are enumerated in the processor as before, via CPUID.

6.1.2 Intel® RDT on Intel Atom® Processors, Including the P5000 Series

Intel Atom® processors, such as the P5000 series, based on Tremont microarchitecture add the following Intel RDT enhancements:

- L2 CAT/CDP: L2 CAT/CDP and L3 CAT/CDP may be enabled simultaneously on supported processors. As these are existing features defined in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B, no new software enabling should be required.
- Supported processors match the capabilities of the 3rd generation Intel Xeon Scalable Processor Family based on Ice Lake Server microarchitecture, including traditional Intel RDT uncore features: L3 CAT/CDP, CMT, MBM, and second-generation MBA. As these features are architectural, no new software enabling is required. Related enhancements in Intel Xeon processors also carry forward to supported Intel Atom processors, with consistent software enabling. These features include 32-bit MBM counters, second generation MBA, and 15 MBA/L3 CAT CLOS.

6.1.3 Intel® RDT in Future Processors Based on Sapphire Rapids Server Microarchitecture

Processors based on Sapphire Rapids Server microarchitecture add the following Intel RDT enhancements:

- STLB QoS: Capability to manage the second-level translation lookaside buffer structure within the core (STLB) in a manner quite similar to CAT (CLOS-based, with capacity masks). This may enable software that is sensitive to TLB performance to achieve better determinism. This is a model-specific feature due to the microarchitectural nature of the STLB structure. The code regions of interest should be manually accessed.

6.1.4 Intel® RDT in Processors Based on Emerald Rapids Server Microarchitecture

Processors based on Emerald Rapids Server microarchitecture add the following Intel RDT enhancements:

- L2 CAT and CDP: Includes control over the L2 cache and the ability to partition the L2 cache into separate code and data virtual caches. No new software enabling is required; this is the same architectural feature described in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

6.1.5 Future Intel® RDT

Future processors add the following Intel RDT enhancements:

- Third generation Memory Bandwidth Allocation (MBA): New per-logical-processor capability for bandwidth control (rather than the more coarse-grained core-level throttling value resolution in prior generations). This capability enables more precise bandwidth shaping and noisy neighbor control. Some portions of the control infrastructure now operate at core frequencies for controls that are responsive at the nanosecond level.

6.2 ENUMERABLE MEMORY BANDWIDTH MONITORING COUNTER WIDTH

Memory Bandwidth Monitoring (MBM) is an Intel RDT feature that tracks total and local bandwidth generated that misses the L3 cache.

The original Memory Bandwidth Monitoring (MBM) architectural definition defines counters of up to 62 bits in the IA32_QM_CTR MSR, and the first-generation MBM implementation provided 24-bit counters. Software is required to poll at ≥ 1Hz to ensure that data is retrieved before a counter rollover occurs more than once. This ≥ 1Hz sampling ensures that under worst-case conditions rollover between samples occurs at most once, but under typical conditions rollover often requires multiple seconds to occur.

As bandwidths scale, extensions to more elegantly handle high-bandwidth future systems are desirable. One of these extensions, detailed in this section, includes an enumerable MBM counter width. The 3rd generation Intel Xeon Scalable Processor Family, and corresponding Intel Atom processors, utilize this definition to implement 32-bit MBM counters, and future growth should be anticipated.

6.2.1 Memory Bandwidth Monitoring (MBM) Enabling

Memory Bandwidth Monitoring, like other Intel RDT features, uses CPUID for enumeration, and MSRs for assigning RMIDs and retrieving counter data. For CPUID enumeration details, see the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A. For additional MBM details, see Chapter 18 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

6.2.2 Augmented MBM Enumeration and MSR Interfaces for Extensible Counter Width

A field is added to CPUID to enumerate the MBM counter width in platforms that support the extensible MBM counter width feature.

Before this point, CPUID.0F.[ECX=1]:EAX was reserved. This CPUID output register (EAX) is redefined to provide two new fields:

- Encode counter width as offset from 24b in bits[7:0].
- Enumeration of the presence of an overflow bit in the IA32_QM_CTR MSR via EAX bit[8].

See "CPUID—CPU Identification" in Chapter 1 for details.

In EAX bits 7:0, the counter width is encoded as an offset from 24b. A value of zero in this field means 24-bit counters are supported. A value of 8 indicates that 32-bit counters are supported, as in the 3rd generation Intel Xeon Scalable Processor Family.

With the addition of this enumerable counter width, the requirement that software poll at ≥ 1Hz is removed. Software may poll at a varying rate with reduced risk of rollover, and under typical conditions rollover is likely to require hundreds of seconds (though this value is not explicitly specified and may vary and decrease in future processor generations as memory bandwidths increase). If software seeks to ensure that rollover does not occur more than once between samples, then sampling at ≥ 1Hz while consuming the enumerated counter widths' worth of data will provide this guarantee, for a specific platform and counter width, under all conditions.

Software that uses the MBM event retrieval MSR interface should be updated to comprehend this new format, which enables up to 62-bit MBM counters to be provided by future platforms. Higher-level software that consumes the resulting bandwidth values is not expected to be affected.

An overflow bit is defined in the IA32_QM_CTR MSR, bit 61, if CPUID.0F.[ECX=1]:EAX[bit 8] is set. This rollover bit will be set on overflow of the MBM counters and reset upon read. Current processors do not support this capability.

6.3 SECOND GENERATION MEMORY BANDWIDTH ALLOCATION

The second generation of Memory Bandwidth Allocation (MBA) is implemented in the 3rd generation Intel Xeon Scalable Processor Family, and related Intel Atom processors such as the P5000 Series. This enhanced MBA capability provides improved efficiency and accuracy in throttling, along with providing increased system throughput. Rather than a strict bandwidth control mechanism, a dynamic hardware controller is implemented, which can react to changing bandwidth conditions at the microsecond level.

Before using the second generation MBA feature, the MBA hardware controller requires a BIOS-assisted calibration process that may include inputs such as the number of memory channels populated and other system parameters; this is a change from the first generation of MBA. Intel BIOS reference code includes a default configuration that is recommended for general usage, and BIOS profiles may be created with alternate tuning values to optimize for certain usages (such as stricter throttling).

Second generation MBA moves from static throttling at the core/uncore interface, to a more dynamic control method based on a hardware controller that tracks actual DRAM bandwidth. This allows software that uses primarily the L3 cache to observe increased throughput for a given throttling level, or fine-grained throughput benefits for software that exhibits L3-bound phases. Due to the closer consideration of memory bandwidth loading, this enhancement may lead to an increase in system efficiency when using second generation MBA relative to prior implementations of the feature. Backward compatibility of the software interfaces is preserved, and second generation MBA changes manifest as enhancements atop the MBA feature baseline.

As with the prior generation feature, second generation MBA uses CPUID for enumeration, and throttling is performed using a mapping created from software thread-to-CLOS (in the IA32_PQR_ASSOC MSR), which is then mapped per-CLOS to delay values via the IA32_L2_QoS_Ext_BW_Thrtl_n MSRs. A privileged operating system or virtual machine manager software may specify a per-CLOS delay value, 0-90% bandwidth throttling for instance, though the max and granularity values are platform dependent and enumerated in CPUID.

6.3.1 Second Generation MBA Advantages

Additional features added over first generation MBA are described below.

1. Previously, only the maximum delay value across two CLOS on a physical core could be selected in MBA. Second generation MBA allows a minimum delay value to be selected instead, which may enhance usage with Intel® Hyper-Threading Technology.
2. Only a single preprogrammed calibration table was possible in first generation MBA, meaning different memory configurations had the potential for different linearity and percent delay value error values depending on the configuration. This is addressed by the BIOS support in the second generation of MBA, and certain BIOS implementations may program a different calibration table per memory configuration, for instance.
3. The second generation MBA controller provides the ability to more closely monitor the memory bandwidth loading and deliver more optimal results.
4. The new MBA hardware controller reduces the need for a fine-grained software controller to manage application phases for optimal efficiency. Note that a software controller may still be valuable to translate MBA throttling values to bandwidths in GB/s or application Service Level Objectives (SLOs), such as performance targets.

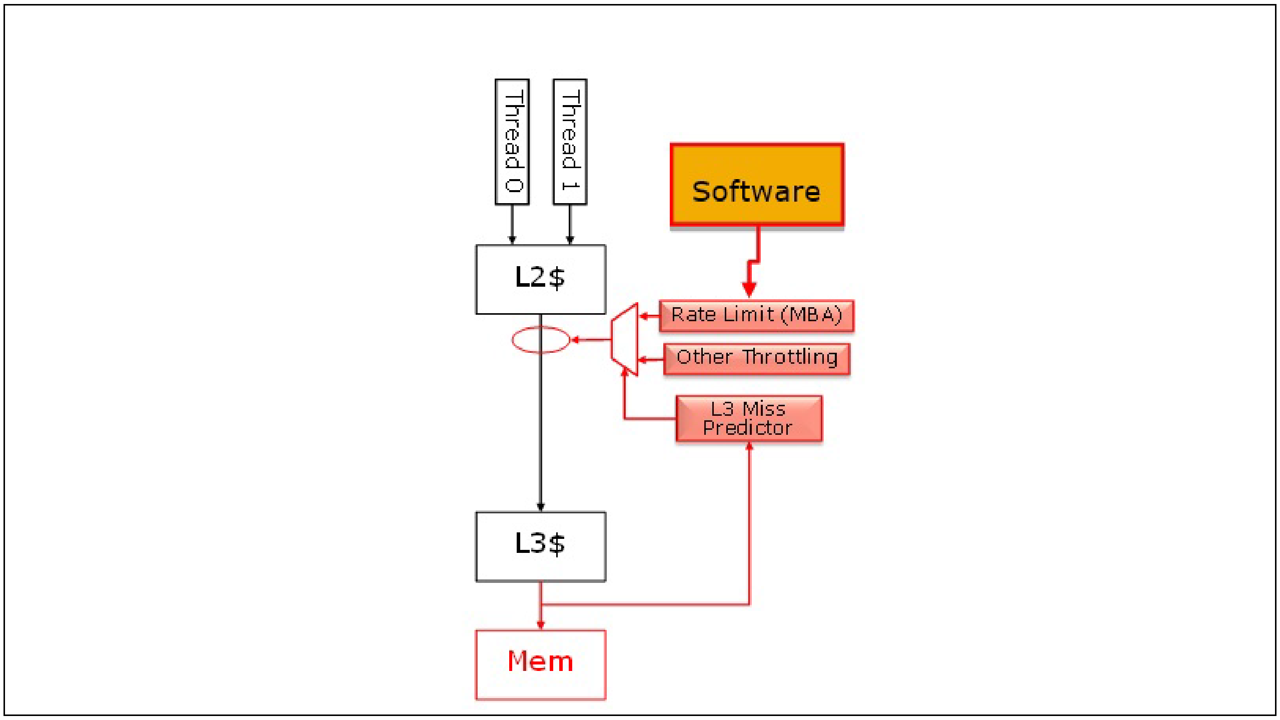


Figure 6-1. Second Generation MBA, Including a Fast-Responding Hardware Controller

The second generation MBA implementation is shown in Figure 6-1. The feature now operates through the use of an advanced new hardware controller and feedback mechanism, which allows automated hardware monitoring and control around the user-provided delay value set point. This set point and associated throttling value infrastructure remains unchanged from prior generation MBA, preserving software compatibility.

MBA enhancements, in addition to the new hardware controller, include:

1. Configurable delay selection across threads.
   - MBA 1.0 implementation statically picks the max MBA Throttling Level (MBAThrotLvl) across the threads running on a core (by calculating value = max(MBAThrotLvl(CLOS[thread0]), MBAThrotLvl(CLOS[thread1]))).
   - Software may have the option to pick either maximum or minimum delay to be resolved and applied across the threads; maximum value remains the default.
2. Increasing CLOSIDs from 8 to 15.
   - Previous generations of microarchitecture provided 8 CLOS tags for MBA.
   - The 3rd generation Intel Xeon Scalable Processor Family and related Intel Atom processors, such as the P5000 Series, increase this value to 15 (also consistent with L3 CAT).

6.3.2 Second Generation MBA Software-Visible Changes

A new model-specific MSR is introduced with second generation MBA to allow software to select from the maximum (default) or minimum of resolved throttling values (see formula above). This capability is controlled via a bit in the new MBA_CFG MSR, shown in Table 6-1.

Table 6-1. MBA_CFG MSR Definition

Register Address		Architectural MSR Name / Bit Fields	Description
Hex	Decimal		
C84H	3204	MBA_CFG	MBA Configuration Register
		0	Min (1) or max (0) of per-thread MBA delays.
		63:1	Reserved. Attempts to write to reserved bits result in a #GP(0).

Note that bit[0] for min/max configuration is supported in second generation MBA, but is removed in third generation MBA when the controller logic becomes capable of managing throttling values on a per-logical-processor basis. The transient nature of this enhancement is why the min/max control remains model-specific.

To enumerate and manage support for the model-specific min/max feature, software may use processor family/model/stepping to match supported products, then CPUID to later detect enhanced third generation MBA support.

6.4 THIRD GENERATION MEMORY BANDWIDTH ALLOCATION

The third generation MBA feature on future processors based on Granite Rapids microarchitecture further enhances the feature with per-logical-processor control and a further improved controller design. Total memory bandwidth (all LLC miss traffic) is now managed by MBA 3.0.

This implementation follows the past MBA precedent of delivering significant enhancements without a major software overhaul, and while preserving backward compatibility.

6.4.1 Third Generation MBA Hardware Changes

The third generation of MBA builds upon the hardware controller introduced in the previous generation, which enabled significant system-level benefits, by providing the capability to independently throttle logical processors, rather than more coarse-grained per-core throttling in prior generations. Throttling values are no longer selected as the “min” or “max” of the two throttling values for the threads running on the core; instead throttling values are independently and directly applied to each logical processor.

While this enhancement means that more direct throttling of threads is possible, future usage guidance may be necessary to help explain the effects of Intel® Hyper-Threading Technology contention vs. cache and memory contention, and how these effects may be understood by software.

6.4.2 Third Generation MBA Software-Visible Changes

In order to allow software to change its tuning behavior and detect that per-logical-processor throttling is supported on a particular product generation, a new CPUID bit is added to the MBA CPUID leaf to indicate support. See “CPUID—CPU Identification” in Chapter 1 for details.

Despite another significant improvement of the hardware controller infrastructure architecture and improved capabilities, controller responsiveness, new internal microarchitecture, and transient-arresting capabilities, no new software interface changes are required to make use of the third generation of MBA relative to prior generations. Software previously using the second generation MBA min/max selection capability should discontinue use of the MBA_CFG MSR. MBA 3.0 is the default mode of operation on the future Granite Rapids Server microarchitecture.

6.5 FUTURE MBA ENHANCEMENTS

Further model-specific enhancements to MBA may be introduced on the Granite Rapids Server microarchitecture to support specific usages; contact your Intel representative for details.

CHAPTER 7
LINEAR ADDRESS MASKING (LAM)

This chapter describes a new feature called **linear-address masking** (**LAM**). LAM modifies the checking that is applied to 64-bit linear addresses, allowing software to use of the untranslated address bits for metadata.

In 64-bit mode, linear address have 64 bits and are translated either with 4-level paging, which translates the low 48 bits of each linear address, or with 5-level paging, which translates 57 bits. The upper linear-address bits are reserved through the concept of **canonicality**. A linear address is 48-bit canonical if bits 63:47 of the address are identical; it is 57-bit canonical if bits 63:56 are identical. (Clearly, any linear address that is 48-bit canonical is also 57-bit canonical.) When 4-level paging is active, the processor requires all linear addresses used to access memory to be 48-bit canonical; similarly, 5-level paging ensures that all linear addresses are 57-bit canonical.

Software usages that associate metadata with a pointer might benefit from being able to place metadata in the upper (untranslated) bits of the pointer itself. However, the canonicality enforcement mentioned earlier implies that software would have to mask the metadata bits in a pointer (making it canonical) before using it as a linear address to access memory. LAM allows software to use pointers with metadata without having to mask the metadata bits. With LAM enabled, the processor masks the metadata bits in a pointer before using it as a linear address to access memory.

LAM is supported only in 64-bit mode and applies only addresses used for data accesses. LAM doe not apply to addresses used for instruction fetches or to those that specify the targets of jump and call instructions.

7.1 ENUMERATION, ENABLING, AND CONFIGURATION

LAM support by the processor is enumerated by the CPUID feature flag CPUID.(EAX=07H, ECX=01H):EAX.LAM[bit 26]. Enabling and configuration of LAM is controlled by the following new bits in control registers: CR3[62] (**LAM_U48**), CR3[61] (**LAM_U57**), and CR4[28] (**LAM_SUP**). The use of these control bit is explained below.

LAM supports configurations that differ regarding which pointer bits are masked and can be used for metadata. With **LAM48**, pointer bits in positions 62:48 are masked (resulting in a **LAM width** of 15); with **LAM57**, pointer bits in positions 62:57 are masked (a LAM width of 6). The LAM width may be configured differently for user and supervisor pointers. LAM identifies pointer as a user pointer if bit 63 of the pointer is 0 and as a supervisor pointer if bit 63 of the pointer is 1.

CR3.LAM_U48 and CR3.LAM_U57 enable and configure LAM for user pointers:

- If CR3.LAM_U48 = CR3.LAM_U57 = 0, LAM is not enabled for user pointers.
- If CR3.LAM_U48 = 1 and CR3.LAM_U57 = 0, LAM48 is enabled for user pointers (a LAM width of 15).
- If CR3.LAM_U57 = 1, LAM57 applies to user pointers (a LAM width of 6; CR3.LAM_U48 is ignored).

CR4.LAM_SUP enables and configures LAM for supervisor pointers:

- If CR3.LAM_SUP = 0, LAM is not enabled for supervisor pointers.
- If CR3.LAM_SUP = 1, LAM is enabled for supervisor pointers with a width determined by the paging mode:
  - If 4-level paging is enabled, LAM48 is enabled for supervisor pointers (a LAM width of 15).
  - If 5-level paging is enabled, LAM57 is enabled for supervisor pointers (a LAM width of 6).

Note that the LAM identification of a pointer as user or supervisor is based solely on the value of pointer bit 63 and does not, for the purposes of LAM, depend on the CPL.

7.2 TREATMENT OF DATA ACCESSES WITH LAM ACTIVE FOR USER POINTERS

Recall that, without LAM, canonicality checks are defined so that 4-level paging requires bits 63:47 of each pointer to be identical, while 5-level paging requires bits 63:56 to be identical. LAM allows some of these bits to be used as metadata by modifying canonicality checking.

When LAM48 is enabled for user pointers (see Section 7.1), the processor allows bits 62:48 of a user pointer to be used as metadata. Regardless of the paging mode, the processor performs a modified canonicality check that enforces that bit 47 of the pointer matches bit 63. As illustrated in Figure 7-1, bits 62:48 are not checked and are thus available for software metadata. After this modified canonicality check is performed, bits 62:48 are masked by sign-extending the value of bit 47 (0), and the resulting (48-bit canonical) address is then passed on for translation by paging.

(Note also that, without LAM, canonicality checking with 5-level paging does not apply to bit 47 of a user pointer; when LAM48 is enabled for user pointers, bit 47 of a user pointer must be 0. Note also that linear-address bits 56:47 are translated by 5-level paging. When LAM48 is enabled for user pointers, these bits are always 0 in any linear address derived from a user pointer: bits 56:48 of the pointer contained metadata, while bit 47 is required to be 0.)

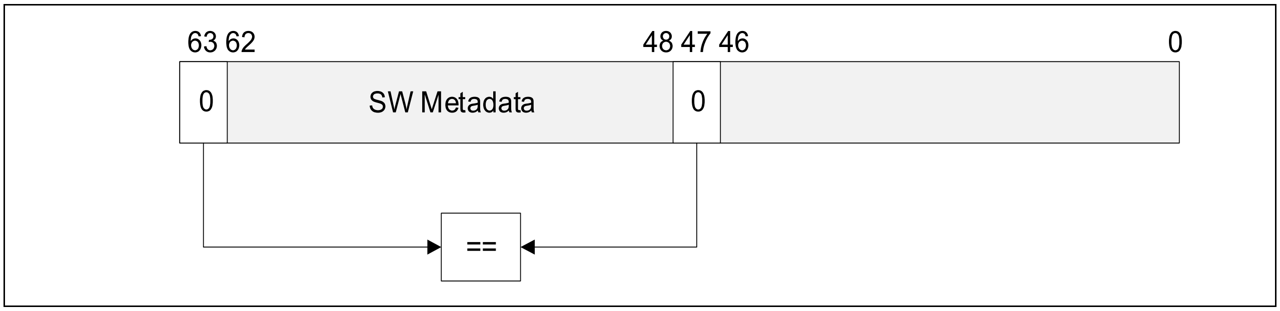


Figure 7-1. Canonicality Check When LAM48 is Enabled for User Pointers

When LAM57 is enabled for user pointers, the processor allows bits 62:57 of a user pointer to be used as metadata. With 5-level paging, the processor performs a modified canonicality check that enforces only that bit 56 of the pointer matches bit 63. As illustrated in Figure 7-2, bits 62:57 are not checked and are thus available for software metadata. After this modified canonicality check is performed, bits 62:57 are masked by sign-extending the value of bit 56 (0), and the resulting (57-bit canonical) address is then passed on for translation by 5-level paging.

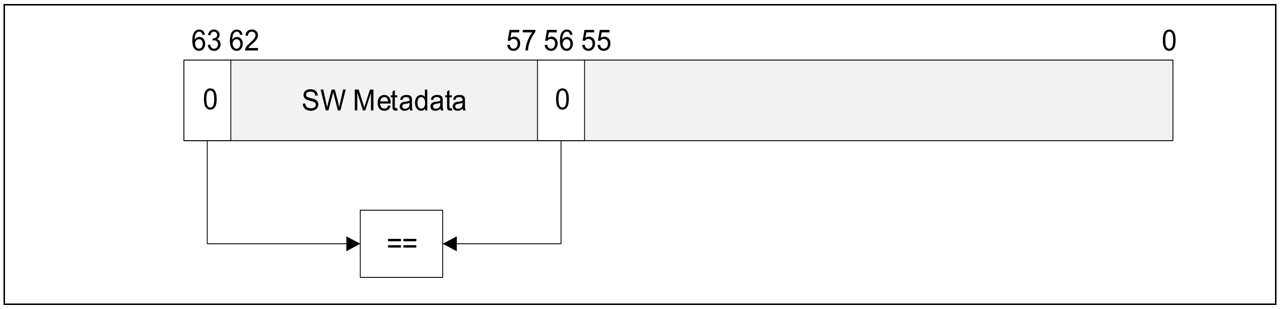


Figure 7-2. Canonicality Check When LAM57 is Enabled for User Pointers with 5-Level Paging

When LAM57 is enabled for user pointers with 4-level paging, the processor performs a modified canonicality check that enforces only that bits 56:47 of a user pointer match bit 63. As illustrated in Figure 7-3, bits 62:57 are not checked and are thus available for software metadata. After this modified canonicality check is performed, bits 62:57 are masked by sign-extending the value of bit 56 (0), and the resulting (48-bit canonical) address is then passed on for translation by 4-level paging.

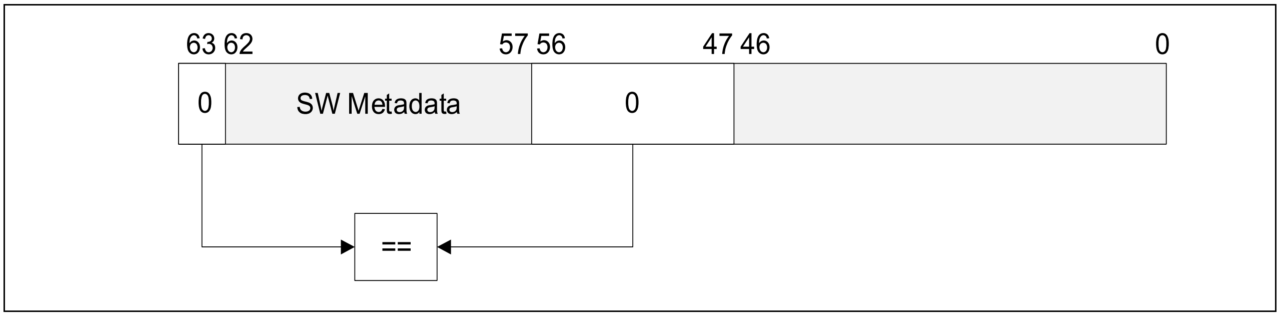


Figure 7-3. Canonicality Check When LAM57 is Enabled for User Pointers with 4-Level Paging

7.3 TREATMENT OF DATA ACCESSES WITH LAM ACTIVE FOR SUPERVISOR POINTERS

As with user pointers (Section 7.2), LAM can be configured to modify canonicality checking to allow use of metadata in supervisor pointers. For supervisor pointers, the number of metadata bits (the LAM width) available depends on the paging mode active: with 5-level paging, enabling LAM for supervisor pointers results in LAM57; with 4-level paging, it results in LAM48 (see Section 7.1).

When LAM57 is enabled for supervisor pointers (5-level paging), the processor performs a modified canonicality check that enforces only that bit 56 of a supervisor pointer matches bit 63. As illustrated in Figure 7-4, bits 62:57 are not checked and are thus available for software metadata. After this modified canonicality check is performed, bits 62:57 are masked by sign-extending the value of bit 56 (1), and the resulting (57-bit canonical) address is then passed on for translation by 5-level paging.

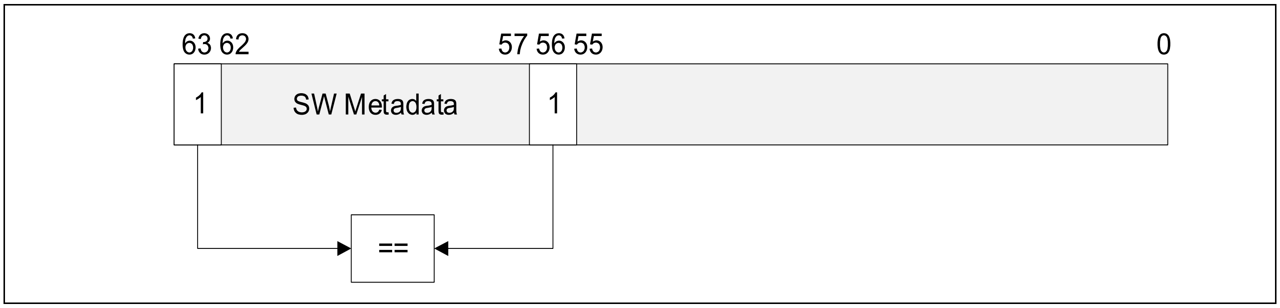


Figure 7-4. Canonicality Check When LAM57 is Enabled for Supervisor Pointers with 5-Level Paging

When LAM48 is enabled for supervisor pointers (4-level paging), the processor performs a modified canonicality check that enforces only that bit 47 of a supervisor pointer matches bit 63. As illustrated in Figure 7-5, bits 62:48 are not checked and are thus available for software metadata. After this modified canonicality check is performed, bits 62:48 are masked by sign-extending the value of bit 47 (1), and the resulting (48-bit canonical) address is then passed on for translation by 4-level paging.

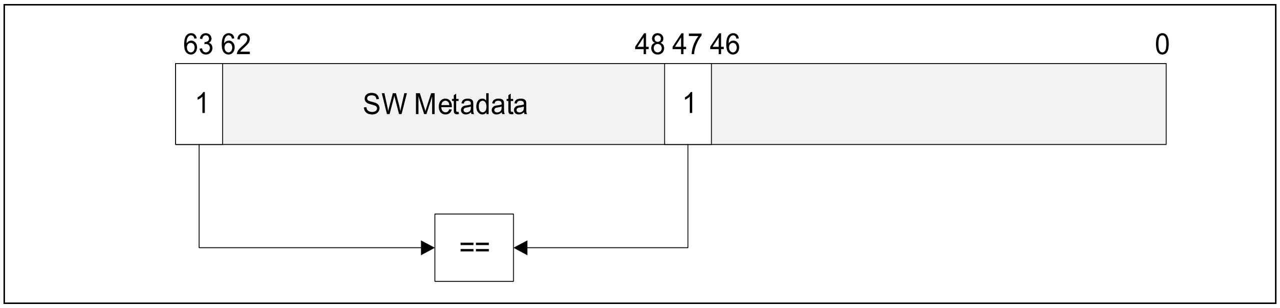


Figure 7-5. Canonicality Check When LAM48 is Enabled for Supervisor Pointers with 4-Level Paging

7.4 CANONICALITY CHECKING FOR DATA ADDRESSES WRITTEN TO CONTROL REGISTERS AND MSRS

Processors that support LAM continue to require the addresses written to control registers or MSRs be 57-bit canonical if the processor supports 5-level paging or 48-bit canonical if it supports only 4-level paging; LAM masking is not performed on these writes. When the contents of such registers are used as pointers to access memory, the processor performs canonicality checking and masking based on paging mode and LAM mode configuration active at the time of access.

7.5 PAGING INTERACTIONS

As explained in Section 7.2 and Section 7.3, LAM masks certain bits in a pointer by sign-extension, resulting in a linear address to be translated by paging.

In most cases, the address bits in the masked positions are not used by address translation. However, if 5-level paging is active and LAM48 is enabled for user pointers, bit 47 of a user pointer must be zero and is extended over bits 62:48 to form a linear address — even though bits 56:48 are used by 5-level paging. This implies that, when LAM48 is enabled for user pointers, bits 56:47 are 0 in any linear address translated for a user pointer.

Page faults report the faulting linear address in CR2. Because LAM masking (by sign-extension) applies before paging, the faulting linear address recorded in CR2 does not contain the masked metadata.

The INVLPG instruction is used to invalidate any translation lookaside buffer (TLB) entries for a memory address specified with the source operand. LAM does not apply to the specified memory address. Thus, in 64-bit mode, if the memory address specified is in non-canonical form then the INVLPG is the same as a NOP.

The INVPCID instruction invalidates mappings in the TLB and paging structure caches based on the processor context identifier (PCID). The INVPCID descriptor provides the memory address to invalidate when the descriptor is of type 0 (individual-address invalidation). LAM does not apply to the specified memory address, and in 64-bit mode if this memory address is in non-canonical form then the processor generates a #GP(0) exception.

7.6 VMX INTERACTIONS

7.6.1 Guest Linear Address

Certain VM exits save in a VMCS field the guest linear address pertaining to the VM exit. Because such a linear address results from masking the original pointer, the processor does not report the masked metadata in the VMCS. The guest linear address saved is always the result of the sign-extension described in Section 7.2 and Section 7.3.

7.6.2 VM-Entry Checking of Values of CR3 and CR4

VM entry checks the values of the CR3 and CR4 fields in the guest-area and host-state area of the VMCS. In particular, the bits in these fields that correspond to bits reserved in the corresponding register are checked and must be 0.

On processors that enumerate support for LAM (Section 7.1), VM entry allows bits 62:61 to be set in either CR3 field and allows bit 28 to be set in either CR4 field.

7.6.3 CR3-Target Values

If the "CR3-load exiting" VM-execution control is 1, execution of MOV to CR3 in VMX non-root operation causes a VM exit unless the value of the instruction's source operand is equal to one of the CR3-target values specified in the VMCS.

Processor support for LAM does not change this behavior. The comparison of the instruction source operand to each of the CR3-target values considers all 64 bits, including the two new bits that determine LAM enabling for user pointers (see Section 7.1).

7.6.4 Hypervisor-Managed Linear Address Translation (HLAT)

Hypervisor-managed linear-address translation (HLAT) is enabled when the "enable HLAT" tertiary processor-based VM-execution control is 1. See Chapter 4, "Non-Write-Back Lock Disable Architecture", for additional details.

When HLAT is enabled for a guest, the processor translates a linear address using HLAT paging structures (instead of guest paging structures) if the address matches the Protected Linear Range (PLR). When LAM is active, it is the linear address (derived from a pointer by masking) that is checked for a PLR match.

The hierarchy of HLAT paging structures is located using a guest-physical address in the VMCS (instead of the guest-physical address in CR3). Nevertheless, LAM enabling and configuration for user pointers is based on the value of CR3[62:61] (see Section 7.1) even when the guest-physical address in CR3 is not used for translating the linear addresses derived from user pointers.

7.7 DEBUG AND TRACING INTERACTIONS

7.7.1 Debug Registers

Debug registers DR0-DR3 can be programmed with linear addresses that are matched against memory accesses for data breakpoints or instruction breakpoints. When LAM is active, it is the linear address (derived from a pointer by masking) that is checked for matching the contents of the debug registers.

7.7.2 Intel® Processor Trace

Intel Processor Trace supports a CR3-filtering mechanism by which generation of packets containing architectural states can be enabled or disabled based on the value of CR3 matching the contents of the IA32_RTIT_CR3_MATCH MSR. On processors that support LAM, bits 62:61 of the CR3 (see Section 7.1) must also match bits 62:61 of this MSR to enable tracing.

7.8 INTEL® SGX INTERACTIONS

Memory operands of ENCLS, ENCLU, and ENCLV that are data pointers follow the LAM architecture and mask suitably. Code pointers continue to not mask metadata bits. ECREATE does not mask BASEADDR specified in SECS, and the unmasked BASEADDR must be canonical.

Two new SECS attribute bits are defined for LAM support in enclave mode:

- ATTRIBUTE.LAM_U48 (bit 9) - Activate LAM for user data pointers and use of bits 62:48 as masked metadata in enclave mode. This bit can be set if CPUID.(EAX=12H, ECX=01H):EAX[9] is 1.
- ATTRIBUTE.LAM_U57 (bit 8) - Activate LAM for user data pointers and use of bits 62:57 as masked metadata in enclave mode. This bit can be set if CPUID.(EAX=12H, ECX=01H):EAX[8] is 1.

ECREATE causes #GP(0) if ATTRIBUTE.LAM_U48 bit is 1 and CPUID.(EAX=12H, ECX=01H):EAX[9] is 0, or if ATTRIBUTE.LAM_U57 bit is 1 and CPUID.(EAX=12H, ECX=01H):EAX[8] is 0.

If SECS.ATTRIBUTES.LAM_U57 is 1, then LAM57 is enabled for user pointers during execution of an enclave controlled by the SECS (regardless of the value of CR3). If SECS.ATTRIBUTES.LAM_U57 is 0 and SECS.ATTRIBUTES.LAM_U48 is 1, then LAM48 is enabled for user pointers during execution of an enclave controlled by the SECS (regardless of the value of CR3).

When in enclave mode, supervisor data pointers are not subject to any masking.

The following ENCLU leaf functions check for linear addresses to be within the ELRANGE. When LAM is active, this check is performed on the linear addresses that result from masking metadata bits in user pointers used by the leaf functions.

- EACCEPT
- EACCEPTCOPY
- EGETKEY
- EMODPE
- EREPORT

The following linear address fields in the Intel SGX data structures hold linear addresses that are either loaded into the EPCM or are written out from the EPCM and do not contain any metadata.

- SECS.BASEADDR
- PAGEINFO.LINADDR

7.9 SYSTEM MANAGEMENT MODE (SMM) INTERACTIONS

On processors that enumerate support for LAM (Section 7.1), RSM allows restoring CR3 with a value that sets either or both bit 62 and bit 61 and restoring a value of CR4 with a value that sets bit 28.

CHAPTER 8
ASYNCHRONOUS ENCLAVE EXIT NOTIFY AND THE EDECCSSA USER LEAF FUNCTION

8.1 INTRODUCTION

Asynchronous Enclave Exit Notify (AEX-Notify) is an extension to Intel® SGX that allows Intel SGX enclaves to be notified after an asynchronous enclave exit (AEX) has occurred. EDECCSSA is a new Intel SGX user leaf function (ENCLU[EDECCSSA]) that can facilitate AEX notification handling, as well as software exception handling. This chapter provides information about changes to the Intel SGX architecture that support AEX-Notify and ENCLU[EDECCSSA].

The following list summarizes the additions to existing Intel SGX data structures to support AEX-Notify (further details are provided in Section 8.3):

- SECS.ATTRIBUTES.AEXNOTIFY: This enclave supports AEX-Notify.
- TCS.FLAGS.AEXNOTIFY: This enclave thread may receive AEX notifications.
- SSA.GPRSGX.AEXNOTIFY: Enclave-writable byte that allows enclave software to dynamically enable/disable AEX notifications.

An AEX notification is delivered by ENCLU[ERESUME] when the following conditions are met:

1. TCS.FLAGS.AEXNOTIFY is set.
2. TCS.CSSA (the current slot index of an SSA frame) is greater than zero.
3. TCS.SSA[TCS.CSSA-1].GPRSGX.AEXNOTIFY[0] is set.

Note that AEX increments TCS.CSSA, and ENCLU[ERESUME] decrements TCS.CSSA, except when an AEX notification is delivered. Instead of decrementing TCS.CSSA and restoring state from the SSA, ENCLU[ERESUME] delivers an AEX notification by behaving as ENCLU[EENTER]. Implications of this behavior include:

- The enclave thread is resumed at EnclaveBase + TCS.OENTRY.
- EAX contains the (non-decremented) value of TCS.CSSA.
- RCX contains the address of the IP following ENCLU[ERESUME].
- The architectural state saved by the most recent AEX is preserved in TCS.SSA[TCS.CSSA-1].

The enclave thread can return to the previous SSA context by invoking ENCLU[EDECCSSA], which decrements TCS.CSSA.

NOTE

A thread can only enter an enclave if SECS.ATTRIBUTES.AEXNOTIFY is equal to TCS.FLAGS.AEXNOTIFY, unless TCS.FLAGS.DBGOPTIN is set to 1.

8.2 ENUMERATION AND ENABLING

Processor support for ENCLU[EDECCSSA] is enumerated by the Intel SGX Capability Enumeration Leaf. If CPUID.(EAX=12H, ECX=0):EAX[11] is set to 1, then a user thread executing in enclave mode can invoke the EDECCSSA user leaf function.

Processor support for AEX-Notify is enumerated by the Intel SGX Attributes Enumeration Leaf. If CPUID.(EAX=12H, ECX=1):EAX[10] is set to 1, then software can set the SECS.ATTRIBUTES.AEXNOTIFY bit (see Section 8.3.3) with ENCLS[ECREATE].

Enclave threads can choose to receive AEX notifications only if the enclave has set the AEXNOTIFY attribute bit to 1. Furthermore, an enclave thread can choose to receive AEX notifications only if it enters the enclave through a TCS with TCS.FLAGS.AEXNOTIFY set to 1. An enclave thread can choose to receive AEX notifications by setting TCS.SSA.GPRSGX.AEXNOTIFY[0] to 1 for each SSA context in which the thread should receive AEX notifications.

NOTE

On some platforms, AEX-Notify and the EDECCSSA user leaf function may be enumerated by CPUID following a microcode update.

8.3 CHANGES TO ENCLAVE DATA STRUCTURES

8.3.1 TCS.FLAGS Changes

A new flag, AEXNOTIFY, is defined. The bit position is 1. A thread that enters the enclave cannot receive AEX notifications unless this flag is set to 1.

8.3.2 SSA.GPRSGX Changes

A new byte, AEXNOTIFY, is defined. The byte position is 167.

A new bit is defined within SSA.GPRSGX.AEXNOTIFY at bit position 0. This bit, SSA.GPRSGX.AEXNOTIFY[0], allows enclave software to dynamically enable/disable AEX notifications. All other bits are reserved.

8.3.3 ATTRIBUTES Changes

A new bit, AEXNOTIFY, is defined. The bit position is 10. The bit indicates that threads within the enclave may receive AEX notifications. Note that this bit also has a corresponding bit in ATTRIBUTEMASK, in the same bit position.

8.4 CHANGES TO INTEL® SGX USER LEAF FUNCTIONS

When a thread enters an enclave through a given TCS, ENCLU[EENTER] and ENCLU[ERESUME] will cause a general protection fault (#GP) if SECS.ATTRIBUTES.AEXNOTIFY is not equal to TCS.FLAGS.AEXNOTIFY.

If the EPCM checks succeed for all pages within TCS.SSA[TCS.CSSA-1], then ENCLU[ERESUME] checks the TCS.SSA[TCS.CSSA-1].GPRSGX.AEXNOTIFY[0] bit (see Section 8.3.2). If this bit is set, then ENCLU[ERESUME] will behave as ENCLU[EENTER]. The implications of this behavior are discussed in Section 8.1.

The operational changes to ENCLU[EENTER] and ENCLU[ERESUME] are detailed in Section 8.8.

8.5 NEW INTEL® SGX USER LEAF FUNCTION: EDECCSSA

EDECCSSA—Decrements TCS.CSSA

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 09H ENCLU[EDECCSSA]	IR	V/V	EDECCSSA	This leaf function decrements TCS.CSSA.

Instruction Operand Encoding

Op/En	EAX
IR	EDECCSSA (In)

Description

This leaf function switches the current SSA frame by decrementing TCS.CSSA for the current enclave thread. This instruction leaf can only be executed inside an enclave.

EDECCSSA Memory Parameter Semantics

TCS
Read/Write access by Enclave

The instruction faults if any of the following occurs:

EDECCSSA Faulting Conditions

TCS.CSSA is 0.	TCS is not valid or available or locked.
The SSA frame is not valid or in use.	

Concurrency Restrictions

Table 8-1. Base Concurrency Restrictions of EDECCSSA

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDECCSSA	TCS [CR_TCS_PA]	Shared	GP	

Table 8-2. Additional Concurrency Restrictions of EDECCSSA

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EDECCSSA	TCS [CR_TCS_PA]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EDECCSSA Operational Flow

Name	Type	Size (bits)	Description
TMP_SSA	Effective Address	32/64	Address of current SSA frame.
TMP_XSIZE	Integer	64	Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.
TMP_SSA_PAGE	Effective Address	32/64	Pointer used to iterate over the SSA pages in the target frame.
TMP_GPR	Effective Address	32/64	Address of the GPR area within the target SSA frame.
TMP_XSAVE_PAGE_PA_n	Physical Address	32/64	Physical address of the nth page within the target SSA frame.
TMP_CET_SAVE_AREA	Effective Address	32/64	Address of the current CET save area.
TMP_CET_SAVE_PAGE	Effective Address	32/64	Address of the current CET save area page.

```
IF (CR_TCS_PA.CSSA = 0)
    THEN #GP(0); FI;

(* Compute linear address of SSA frame *)
TMP_SSA := CR_TCS_PA.OSSA + CR_ACTIVE_SECS.BASEADDR + 4096 * CR_ACTIVE_SECS.SSAFRAMESIZE * (CR_TCS_PA.CSSA - 1);
TMP_XSIZE := compute_XSAVE_frame_size(CR_ACTIVE_SECS.ATTRIBUTES.XFRM);

FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
    (* Check page is read/write accessible *)
    Check that DS:TMP_SSA_PAGE is read/write accessible;
    If a fault occurs, release locks, abort and deliver that fault;
    IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
         THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
         THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
         THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or (EPCM(DS:TMP_SSA_PAGE_.MODIFIED = 1))
         THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ( ( EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMPSSA_PAGE) or
    (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
    (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(CR_TCS_PA).ENCLAVESECS) or
    (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0))
         THEN #PF(DS:TMP_SSA_PAGE); FI;
    TMP_XSAVE_PAGE_PA_n := Physical_Address(DS:TMP_SSA_PAGE);
ENDFOR

(* Compute address of GPR area*)
TMP_GPR := TMP_SSA + 4096 * CR_ACTIVE_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
Check that DS:TMP_SSA_PAGE is read/write accessible;
If a fault occurs, release locks, abort and deliver that fault;
IF (DS:TMP_GPR does not resolve to EPC page)
    THEN #PF(DS:TMP_GPR); FI;
```

```
IF (EPCM(DS:TMP_GPR).VALID = 0)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
    THEN #PF(DS:TMP_GPR); FI;
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
    THEN #PF(DS:TMP_GPR); FI;
IF ( ( EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or
    (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
    (EPCM(DS:TMP_GPR).ENCLAVESECS ≠ EPCM(CR_TCS_PA).ENCLAVESECS) or
    (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0) )
        THEN #PF(DS:TMP_GPR); FI;

IF (TMP_MODE64 = 0)
    THEN
        IF (TMP_GPR + (sizeof(GPRSGX_AREA) -1) is not in DS segment)
            THEN #GP(0); FI;
FI;

IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        IF ((CR_ACTIVE_SECS.CET_ATTRIBUTES.SH_STK_EN == 1) OR (CR_ACTIVE_SECS.CET_ATTRIBUTES.ENDBR_EN == 1))
            THEN
                (* Compute linear address of what will become new CET state save area and cache its PA *)
                TMP_CET_SAVE_AREA := CR_TCS_PA.OCETSSA + CR_ACTIVE_SECS.BASEADDR + (CR_TCS_PA.CSSA - 1) * 16;
                TMP_CET_SAVE_PAGE := TMP_CET_SAVE_AREA & ~0xFFF;
                Check the TMP_CET_SAVE_PAGE page is read/write accessible
                If fault occurs release locks, abort and deliver fault

                (* read the EPCM VALID, PENDING, MODIFIED, BLOCKED and PT fields atomically *)
                IF ((DS:TMP_CET_SAVE_PAGE Does NOT RESOLVE TO EPC PAGE) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).VALID = 0) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).PENDING = 1) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).MODIFIED = 1) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).BLOCKED = 1) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).R = 0) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).W = 0) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVEADDRESS ≠ DS:TMP_CET_SAVE_PAGE) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).PT ≠ PT_SS_REST) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVESECS ≠ EPCM(CR_TCS_PA).ENCLAVESECS))
            THEN #PF(DS:TMP_CET_SAVE_PAGE); FI;
        FI;
FI;

(* At this point, the instruction is guaranteed to complete *)
CR_TCS_PA.CSSA := CR_TCS_PA.CSSA - 1;

CR_GPR_PA := Physical_Address(DS:TMP_GPR);

FOR EACH TMP_XSAVE_PAGE_n
    CR_XSAVE_PAGE_n := TMP_XSAVE_PAGE_PA_n;
ENDFOR

IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
```

```
        IF ((TMP_SECS.CET_ATTRIBUTES.SH_STK_EN == 1) OR
        (TMP_SECS.CET_ATTRIBUTES.ENDBR_EN == 1))
            THEN
                CR_CET_SAVE_AREA_PA := Physical_Address(DS:TMP_CET_SAVE_AREA);
        FI;
FI;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0) If executed outside an enclave.

If CR_TCS_PA.CSSA = 0.

#PF(error code) If a page fault occurs in accessing memory.

If one or more pages of the target SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

If CET is enabled for the enclave and the target CET SSA frame is not readable/writable, or does not resolve to a valid PT_REG EPC page.

64-Bit Mode Exceptions

#GP(0) If executed outside an enclave.

If CR_TCS_PA.CSSA = 0.

#PF(error code) If a page fault occurs in accessing memory.

If one or more pages of the target SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

If CET is enabled for the enclave and the target CET SSA frame is not readable/writable, or does not resolve to a valid PT_REG EPC page.

8.6 IMPLICATIONS FOR ENCLAVE CODE DEBUG AND PROFILING

Whenever an opt-in enclave entry is used to perform enclave code debugging or profiling, the debugger or profiling tool may clear TCS.FLAGS.AEXNOTIFY to prevent AEX notifications from being delivered whenever an AEX occurs.

8.7 INTERACTION WITH INTEL® CET

Because the current CET SSA frame is indicated by TCS.CSSA, ENCLU[EDECCSSA] changes the current CET SSA frame as well as the current SSA frame.

8.8 CHANGES TO INTEL® SGX USER LEAF FUNCTION OPERATION

All changes to existing operation are highlighted in green.

8.8.1 Changes to EENTER Operation

EENTER is a serializing instruction. The instruction faults if any of the following occurs:

Address in RBX is not properly aligned.	Any TCS.FLAGS's must-be-zero bit is not zero.
TCS pointed to by RBX is not valid or available or locked.	Current 32/64 mode does not match the enclave mode in SECS.ATTRIBUTES.MODE64.
The SECS is in use.	Either of TCS-specified FS and GS segment is not a subsets of the current DS segment.
Any one of DS, ES, CS, SS is not zero.	If XSAVE available, CR4.OSXSAVE = 0, but SECS.ATTRIBUTES.XFRM ≠ 3.
CR4.OSFXSR ≠ 1.	If CR4.OSXSAVE = 1, SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
If SECS.ATTRIBUTES.AEXNOTIFY ≠ TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN = 0.	

Operation

```
TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Make sure DS is usable, expand up *)
IF (TMP_MODE64 = 0 and (DS not usable or ((DS[S] = 1) and (DS[bit 11] = 0) and DS[bit 10] = 1)))
    THEN #GP(0); FI;

(* Check that CS, SS, DS, ES.base is 0 *)
IF (TMP_MODE64 = 0)
    THEN
        IF(CS.base ≠ 0 or DS.base ≠ 0) #GP(0); FI;
        IF(ES usable and ES.base ≠ 0) #GP(0); FI;
        IF(SS usable and SS.base ≠ 0) #GP(0); FI;
        IF(SS usable and SS.B = 0) #GP(0); FI;
FI;

IF (DS:RBX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

(* Check AEP is CPU-canonical*)
IF (TMP_MODE64 = 1 and (CS:RCX is not CPU-canonical))
    THEN #GP(0); FI;

(* Check concurrency of TCS operation*)
IF (Other Intel SGX instructions are operating on TCS)
    THEN #GP(0); FI;

(* TCS verification *)
IF (EPCM(DS:RBX).VALID = 0)
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)
    THEN #PF(DS:RBX); FI;

IF ( (EPCM(DS:RBX).ENCLAVEADDRESS ≠ DS:RBX) or (EPCM(DS:RBX).PT ≠ PT_TCS))
```

```
    THEN #PF(DS:RBX); FI;

IF ((EPCM(DS:RBX).PENDING = 1) or (EPCM(DS:RBX).MODIFIED = 1))
    THEN #PF(DS:RBX); FI;

IF ( (DS:RBX).OSSA is not 4KByte Aligned)
    THEN #GP(0); FI;

(* Check proposed FS and GS *)
IF (((DS:RBX).OFSBASE is not 4KByte Aligned) or ((DS:RBX).OGSBASE is not 4KByte Aligned))
    THEN #GP(0); FI;

(* Get the SECS for the enclave in which the TCS resides *)
TMP_SECS := Address of SECS for TCS;

(* Ensure that the FLAGS field in the TCS does not have any reserved bits set *)
IF (((DS:RBX).FLAGS & FFFFFFFFFFFFFFFCH) ≠ 0)
    THEN #GP(0); FI;

(* SECS must exist and enclave must have previously been EINITted *)
IF (the enclave is not already initialized)
    THEN #GP(0); FI;

(* make sure the logical processor’s operating mode matches the enclave *)
IF ((TMP_MODE64 ≠ TMP_SECS.ATTRIBUTES.MODE64BIT))
    THEN #GP(0); FI;

IF (CR4.OSFXSR = 0)
    THEN #GP(0); FI;

(* Check for legal values of SECS.ATTRIBUTES.XFRM *)
IF (CR4.OSXSAVE = 0)
    THEN
        IF (TMP_SECS.ATTRIBUTES.XFRM ≠ 03H) THEN #GP(0); FI;
    ELSE
        IF ((TMP_SECS.ATTRIBUTES.XFRM & XCR0) ≠ TMP_SECS.ATTRIBUES.XFRM) THEN #GP(0); FI;
FI;

IF ((DS:RBX).CSSA.FLAGS.DBGOPTIN = 0) and (DS:RBX).CSSA.FLAGS.AEXNOTIFY ≠ TMP_SECS.ATTRIBUTES.AEXNOTIFY))
    THEN #GP(0); FI;

(* Make sure the SSA contains at least one more frame *)
IF ( (DS:RBX).CSSA ≥ (DS:RBX).NSSA)
    THEN #GP(0); FI;

(* Compute linear address of SSA frame *)
TMP_SSA := (DS:RBX).OSSA + TMP_SECS.BASEADDR + 4096 * TMP_SECS.SSAFRAMESIZE * (DS:RBX).CSSA;
TMP_XSIZE := compute_XSAVE_frame_size(TMP_SECS.ATTRIBUTES.XFRM);

FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
    (* Check page is read/write accessible *)
    Check that DS:TMP_SSA_PAGE is read/write accessible;
    If a fault occurs, release locks, abort and deliver that fault;
    IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or (EPCM(DS:TMP_SSA_PAGE).MODIFIED = 1))
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ( ( EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMP_SSA_PAGE) or (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
        (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
        (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0) )
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    CR_XSAVE_PAGE_n := Physical_Address(DS:TMP_SSA_PAGE);
ENDFOR

(* Compute address of GPR area*)
TMP_GPR := TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
If a fault occurs; release locks, abort and deliver that fault;

IF (DS:TMP_GPR does not resolve to EPC page)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).VALID = 0)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
    THEN #PF(DS:TMP_GPR); FI;
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
    THEN #PF(DS:TMP_GPR); FI;
IF ((EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
    (EPCM(DS:TMP_GPR).ENCLAVESECS EPCM(DS:RBX).ENCLAVESECS) or
    (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0))
    THEN #PF(DS:TMP_GPR); FI;

IF (TMP_MODE64 = 0)
    THEN
        IF (TMP_GPR + (GPR_SIZE -1) is not in DS segment) THEN #GP(0); FI;
FI;

CR_GPR_PA := Physical_Address (DS: TMP_GPR);

(* Validate TCS.OENTRY *)
TMP_TARGET := (DS:RBX).OENTRY + TMP_SECS.BASEADDR;
IF (TMP_MODE64 = 1)
    THEN
        IF (TMP_TARGET is not CPU-canonical) THEN #GP(0); FI;
    ELSE
        IF (TMP_TARGET > CS limit) THEN #GP(0); FI;
FI;

(* Check proposed FS/GS segments fall within DS *)
IF (TMP_MODE64 = 0)
    THEN
        TMP_FSBASE := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
        TMP_FSLIMIT := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR + (DS:RBX).FSLIMIT;
        TMP_GSBASE := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
        TMP_GSLIMIT := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR + (DS:RBX).GSLIMIT;
        (* if FS wrap-around, make sure DS has no holes*)
        IF (TMP_FSLIMIT < TMP_FSBASE)
            THEN
                IF (DS.limit < 4GB) THEN #GP(0); FI;
            ELSE
                IF (TMP_FSLIMIT > DS.limit) THEN #GP(0); FI;
        FI;
        (* if GS wrap-around, make sure DS has no holes*)
        IF (TMP_GSLIMIT < TMP_GSBASE)
            THEN
                IF (DS.limit < 4GB) THEN #GP(0); FI;
            ELSE
                IF (TMP_GSLIMIT > DS.limit) THEN #GP(0); FI;
        FI;
    ELSE
        TMP_FSBASE := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
        TMP_GSBASE := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
        IF ( (TMP_FSBASE is not CPU-canonical) or (TMP_GSBASE is not CPU-canonical))
            THEN #GP(0); FI;
FI;

(* Ensure the enclave is not already active and this thread is the only one using the TCS*)
IF (DS:RBX.STATE = ACTIVE)
    THEN #GP(0); FI;

TMP_IA32_U_CET := 0
TMP_SSP : = 0

IF CPUID.(EAX=12H, ECX=1):EAX[6] = 1
    THEN
        IF ( CR4.CET = 0 )
            THEN
                (* If part does not support CET or CET has not been enabled and enclave requires CET then fail *)
                IF (TMP_SECS.CET_ATTRIBUTES ≠ 0 OR TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0) #GP(0); FI;
        FI;
        (* If indirect branch tracking or shadow stacks enabled but CET state save area is not 16B aligned then fail EENTER *)
        IF (TMP_SECS.CET_ATTRIBUTES.SH_STK_EN = 1 OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN = 1)
            THEN
                IF (DS:RBX.OCETSSA is not 16B aligned) #GP(0); FI;
        FI;

    IF (TMP_SECS.CET_ATTRIBUTES.SH_STK_EN OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN)
        THEN
            (* Setup CET state from SECS, note tracker goes to IDLE *)
            TMP_IA32_U_CET = TMP_SECS.CET_ATTRIBUTES;
            IF (TMP_IA32_U_CET.LEG_IW_EN = 1 AND TMP_IA32_U_CET.ENDBR_EN = 1)
                THEN
                    TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.BASEADDR;
                    TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.CET_LEG_BITMAP_BASE;
            FI;

            (* Compute linear address of what will become new CET state save area and cache its PA *)
            TMP_CET_SAVE_AREA = DS:RBX.OCETSSA + TMP_SECS.BASEADDR + (DS:RBX.CSSA) * 16
            TMP_CET_SAVE_PAGE = TMP_CET_SAVE_AREA & ~0xFFF;

            Check the TMP_CET_SAVE_PAGE page is read/write accessible
            If fault occurs release locks, abort and deliver fault

            (* Read the EPCM VALID, PENDING, MODIFIED, BLOCKED and PT fields atomically *)
            IF ((DS:TMP_CET_SAVE_PAGE Does NOT RESOLVE TO EPC PAGE) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).VALID = 0) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).PENDING = 1) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).MODIFIED = 1) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).BLOCKED = 1) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).R = 0) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).W = 0) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVEADDRESS ≠ DS:TMP_CET_SAVE_PAGE) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).PT ≠ PT_SS_REST) OR
                (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS))
                    THEN
                        #PF(DS:TMP_CET_SAVE_PAGE);
                FI;

                CR_CET_SAVE_AREA_PA := Physical address(DS:TMP_CET_SAVE_AREA)

                IF TMP_IA32_U_CET.SH_STK_EN = 1
                    THEN
                        TMP_SSP = TCS.PREVSSP;
                FI;
        FI;
FI;

CR_ENCLAVE_MODE := 1;
CR_ACTIVE_SECS := TMP_SECS;
CR_ELRANGE := (TMPSECS.BASEADDR, TMP_SECS.SIZE);

(* Save state for possible AEXs *)
CR_TCS_PA := Physical_Address (DS:RBX);
CR_TCS_LA := RBX;
CR_TCS_LA.AEP := RCX;

(* Save the hidden portions of FS and GS *)
CR_SAVE_FS_selector := FS.selector;
CR_SAVE_FS_base := FS.base;
CR_SAVE_FS_limit := FS.limit;
CR_SAVE_FS_access_rights := FS.access_rights;
CR_SAVE_GS_selector := GS.selector;
CR_SAVE_GS_base := GS.base;
CR_SAVE_GS_limit := GS.limit;
CR_SAVE_GS_access_rights := GS.access_rights;

(* If XSAVE is enabled, save XCR0 and replace it with SECS.ATTRIBUTES.XFRM*)
IF (CR4.OSXSAVE = 1)
    CR_SAVE_XCR0 := XCR0;
    XCR0 := TMP_SECS.ATTRIBUTES.XFRM;
FI;
```

```
RCX := RIP;
RIP := TMP_TARGET;
RAX := (DS:RBX).CSSA;
(* Save the outside RSP and RBP so they can be restored on interrupt or EEXIT *)
DS:TMP_SSA.U_RSP := RSP;
DS:TMP_SSA.U_RBP := RBP;

(* Do the FS/GS swap *)
FS.base := TMP_FSBASE;
FS.limit := DS:RBX.FSLIMIT;
FS.type := 0001b;
FS.W := DS.W;
FS.S := 1;
FS.DPL := DS.DPL;
FS.G := 1;
FS.B := 1;
FS.P := 1;
FS.AVL := DS.AVL;
FS.L := DS.L;
FS.unusable := 0;
FS.selector := 0BH;

GS.base := TMP_GSBASE;
GS.limit := DS:RBX.GSLIMIT;
GS.type := 0001b;
GS.W := DS.W;
GS.S := 1;
GS.DPL := DS.DPL;
GS.G := 1;
GS.B := 1;
GS.P := 1;
GS.AVL := DS.AVL;
GS.L := DS.L;
GS.unusable := 0;
GS.selector := 0BH;

CR_DBGOPTIN := TCS.FLAGS.DBGOPTIN;
Suppress_all_code_breakpoints_that_are_outside_ELRANGE;

IF (CR_DBGOPTIN = 0)
    THEN
        Suppress_all_code_breakpoints_that_overlap_with_ELRANGE;
        CR_SAVE_TF := RFLAGS.TF;
        RFLAGS.TF := 0;
        Suppress_monitor_trap_flag for the source of the execution of the enclave;
        Suppress any pending debug exceptions;
        Suppress any pending MTF VM exit;
    ELSE
        IF RFLAGS.TF = 1
            THEN pend a single-step #DB at the end of EENTER; FI;
        IF the "monitor trap flag" VM-execution control is set
            THEN pend an MTF VM exit at the end of EENTER; FI;
FI;
```

```
IF ((CPUID.(EAX=7H, ECX=0):EDX[CET_IBT] = 1) OR (CPUID.(EAX=7H, ECX=0):ECX[CET_SS] = 1)
    THEN
        (* Save enclosing application CET state into save registers *)
        CR_SAVE_IA32_U_CET := IA32_U_CET
        (* Setup enclave CET state *)
        IF CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1
            THEN
                CR_SAVE_SSP := SSP
                SSP := TMP_SSP
        FI;

        IA32_U_CET := TMP_IA32_U_CET;

FI;

Flush_linear_context;
Allow_front_end_to_begin_fetch_at_new_RIP;
```

Protected Mode Exceptions

#GP(0)	If DS:RBX is not page aligned.
	If the enclave is not initialized.
	If part or all of the FS or GS segment specified by TCS is outside the DS segment or not properly aligned.
	If the thread is not in the INACTIVE state.
	If CS, DS, ES or SS bases are not all zero.
	If executed in enclave mode.
	If any reserved field in the TCS FLAG is set.
	If the target address is not within the CS segment.
	If CR4.OSFXSR = 0.
	If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
	If CR4.OSXSAVE = 1 and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
	If SECS.ATTRIBUTES.AEXNOTIFY ≠ TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN = 0.
#PF(error code)	If a page fault occurs in accessing memory.
	If DS:RBX does not point to a valid TCS.
	If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

64-Bit Mode Exceptions

#GP(0)	If DS:RBX is not page aligned.
	If the enclave is not initialized.
	If the thread is not in the INACTIVE state.
	If CS, DS, ES or SS bases are not all zero.
	If executed in enclave mode.
	If part or all of the FS or GS segment specified by TCS is outside the DS segment or not properly aligned.
	If the target address is not CPU-canonical.
	If CR4.OSFXSR = 0.
	If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
	If CR4.OSXSAVE = 1and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
	If SECS.ATTRIBUTES.AEXNOTIFY ≠ TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN = 0.

#PF(error code) If a page fault occurs in accessing memory operands.
If DS:RBX does not point to a valid TCS.
If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

8.8.2 Changes to ERESUME Operation

The instruction faults if any of the following occurs:

Address in RBX is not properly aligned.	Any TCS.FLAGS's must-be-zero bit is not zero.
TCS pointed to by RBX is not valid or available or locked.	Current 32/64 mode does not match the enclave mode in SECS.ATTRIBUTES.MODE64.
The SECS is in use by another enclave.	Either of TCS-specified FS and GS segment is not a subset of the current DS segment.
Any one of DS, ES, CS, SS is not zero.	If XSAVE available, CR4.OSXSAVE = 0, but SECS.ATTRIBUTES.XFRM ≠ 3.
CR4.OSFXSR ≠ 1.	If CR4.OSXSAVE = 1, SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
Offsets 520-535 of the XSAVE area not 0.	The bit vector stored at offset 512 of the XSAVE area must be a subset of SECS.ATTRIBUTES.XFRM.
The SSA frame is not valid or in use.	If SECS.ATTRIBUTES.AEXNOTIFY ≠ TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN = 0.

Operation

Temp Variables in ERESUME Operational Flow

Name	Type	Size	Description
TMP_FSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_GSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_FSLIMIT	Effective Address	32/64	Highest legal address in proposed FS segment.
TMP_GSLIMIT	Effective Address	32/64	Highest legal address in proposed GS segment.
TMP_TARGET	Effective Address	32/64	Address of first instruction inside enclave at which execution is to resume.
TMP_SECS	Effective Address	32/64	Physical address of SECS for this enclave.
TMP_SSA	Effective Address	32/64	Address of current SSA frame.
TMP_XSIZE	integer	64	Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.
TMP_SSA_PAGE	Effective Address	32/64	Pointer used to iterate over the SSA pages in the current frame.
TMP_GPR	Effective Address	32/64	Address of the GPR area within the current SSA frame.
TMP_BRANCH_RECORD	LBR Record		From/to addresses to be pushed onto the LBR stack.
TMP_NOTIFY	Boolean	1	When set to 1, deliver an AEX notification.

```
TMP_MODE64 := ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Make sure DS is usable, expand up *)
IF (TMP_MODE64 = 0 and (DS not usable or ( ( DS[S] = 1) and (DS[bit 11] = 0) and DS[bit 10] = 1)))
    THEN #GP(0); FI;

(* Check that CS, SS, DS, ES.base is 0 *)
IF (TMP_MODE64 = 0)
```

```
    THEN
        IF(CS.base ≠ 0 or DS.base ≠ 0) #GP(0); FI;
        IF(ES usable and ES.base ≠ 0) #GP(0); FI;
        IF(SS usable and SS.base ≠ 0) #GP(0); FI;
        IF(SS usable and SS.B = 0) #GP(0); FI;
FI;

IF (DS:RBX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

(* Check AEP is CPU-canonical*)
IF (TMP_MODE64 = 1 and (CS:RCX is not CPU-canonical))
    THEN #GP(0); FI;

(* Check concurrency of TCS operation*)
IF (Other Intel SGX instructions are operating on TCS)
    THEN #GP(0); FI;

(* TCS verification *)
IF (EPCM(DS:RBX).VALID = 0)
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)
    THEN #PF(DS:RBX); FI;

IF ((EPCM(DS:RBX).PENDING = 1) or (EPCM(DS:RBX).MODIFIED = 1))
    THEN #PF(DS:RBX); FI;

IF ( (EPCM(DS:RBX).ENCLAVEADDRESS ≠ DS:RBX) or (EPCM(DS:RBX).PT ≠ PT_TCS))
    THEN #PF(DS:RBX); FI;

IF ( (DS:RBX).OSSA is not 4KByte Aligned)
    THEN #GP(0); FI;

(* Check proposed FS and GS *)
IF ( ( (DS:RBX).OFSBASE is not 4KByte Aligned) or ( (DS:RBX).OGSBASE is not 4KByte Aligned))
    THEN #GP(0); FI;

(* Get the SECS for the enclave in which the TCS resides *)
TMP_SECS := Address of SECS for TCS;

(* Make sure that the FLAGS field in the TCS does not have any reserved bits set *)
IF ( ( (DS:RBX).FLAGS & FFFFFFFFFFFFFFFCH) ≠ 0)
    THEN #GP(0); FI;

(* SECS must exist and enclave must have previously been EINITted *)
IF (the enclave is not already initialized)
    THEN #GP(0); FI;

(* make sure the logical processor’s operating mode matches the enclave *)
IF ( (TMP_MODE64 ≠ TMP_SECS.ATTRIBUTES.MODE64BIT))
```

```
    THEN #GP(0); FI;

IF (CR4.OSFXSR = 0)
    THEN #GP(0); FI;

(* Check for legal values of SECS.ATTRIBUTES.XFRM *)
IF (CR4.OSXSAVE = 0)
    THEN
        IF (TMP_SECS.ATTRIBUTES.XFRM ≠ 03H) THEN #GP(0); FI;
    ELSE
        IF ( (TMP_SECS.ATTRIBUTES.XFRM & XCR0) ≠ TMP_SECS.ATTRIBUTES.XFRM) THEN #GP(0); FI;
FI;

IF ( (DS:RBX).CSSA.FLAGS.DBGOPTIN = 0) and (DS:RBX).CSSA.FLAGS.AEXNOTIFY ≠ TMP_SECS.ATTRIBUTES.AEXNOTIFY))
    THEN #GP(0); FI;

(* Make sure the SSA contains at least one active frame *)
IF ( (DS:RBX).CSSA = 0)
    THEN #GP(0); FI;

(* Compute linear address of SSA frame *)
TMP_SSA := (DS:RBX).OSSA + TMP_SECS.BASEADDR + 4096 * TMP_SECS.SSAFRAMESIZE * ( (DS:RBX).CSSA - 1);
TMP_XSIZE := compute_XSAVE_frame_size(TMP_SECS.ATTRIBUTES.XFRM);

FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
    (* Check page is read/write accessible *)
    Check that DS:TMP_SSA_PAGE is read/write accessible;
    If a fault occurs, release locks, abort and deliver that fault;
    IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or (EPCM(DS:TMP_SSA_PAGE_.MODIFIED = 1))
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ( ( EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMPSSA_PAGE) or (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
        (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
        (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0) )
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    CR_XSAVE_PAGE_n := Physical_Address(DS:TMP_SSA_PAGE);
ENDFOR

(* Compute address of GPR area*)
TMP_GPR := TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
Check that DS:TMP_SSA_PAGE is read/write accessible;
If a fault occurs, release locks, abort and deliver that fault;
IF (DS:TMP_GPR does not resolve to EPC page)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).VALID = 0)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
    THEN #PF(DS:TMP_GPR); FI;
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
```

```
    THEN #PF(DS:TMP_GPR); FI;
IF ( ( EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
    (EPCM(DS:TMP_GPR).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
    (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0))
    THEN #PF(DS:TMP_GPR); FI;

IF (TMP_MODE64 = 0)
    THEN
        IF (TMP_GPR + (GPR_SIZE -1) is not in DS segment) THEN #GP(0); FI;
FI;

CR_GPR_PA := Physical_Address (DS: TMP_GPR);

IF ((DS:RBX).FLAGS.AEXNOTIFY = 1) and (DS:TMP_GPR.AEXNOTIFY[0] = 1))
    THEN
        TMP_NOTIFY := 1;
    ELSE
        TMP_NOTIFY := 0;
FI;

IF (TMP_NOTIFY = 1)
    THEN
        (* Make sure the SSA contains at least one more frame *)
        IF ((DS:RBX).CSSA ≥ (DS:RBX).NSSA)
            THEN #GP(0); FI;

        TMP_SSA := TMP_SSA + 4096 * TMP_SECS.SSAFRAMESIZE;
        TMP_XSIZE := compute_XSAVE_frame_size(TMP_SECS.ATTRIBUTES.XFRM);

        FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
            (* Check page is read/write accessible *)
            Check that DS:TMP_SSA_PAGE is read/write accessible;
            If a fault occurs, release locks, abort and deliver that fault;

            IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
                THEN #PF(DS:TMP_SSA_PAGE); FI;
            IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
                THEN #PF(DS:TMP_SSA_PAGE); FI;
            IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
                THEN #PF(DS:TMP_SSA_PAGE); FI;
            IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or
            (EPCM(DS:TMP_SSA_PAGE).MODIFIED = 1))
                THEN #PF(DS:TMP_SSA_PAGE); FI;
            IF ((EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMP_SSA_PAGE) or
            (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
            (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
            (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0))
                THEN #PF(DS:TMP_SSA_PAGE); FI;
            CR_XSAVE_PAGE_n := Physical_Address(DS:TMP_SSA_PAGE);
        ENDFOR

        (* Compute address of GPR area*)
        TMP_GPR := TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
        If a fault occurs; release locks, abort and deliver that fault;
        IF (DS:TMP_GPR does not resolve to EPC page)
            THEN #PF(DS:TMP_GPR); FI;
        IF (EPCM(DS:TMP_GPR).VALID = 0)
            THEN #PF(DS:TMP_GPR); FI;
        IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
            THEN #PF(DS:TMP_GPR); FI;
        IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
            THEN #PF(DS:TMP_GPR); FI;
        IF ((EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or
        (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
        (EPCM(DS:TMP_GPR).ENCLAVESECS EPCM(DS:RBX).ENCLAVESECS) or
        (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0))
            THEN #PF(DS:TMP_GPR); FI;

        IF (TMP_MODE64 = 0)
            THEN
                IF (TMP_GPR + (GPR_SIZE -1) is not in DS segment) THEN #GP(0); FI;
        FI;

        CR_GPR_PA := Physical_Address (DS: TMP_GPR);

        TMP_TARGET := (DS:RBX).OENTRY + TMP_SECS.BASEADDR;
    ELSE
        TMP_TARGET := (DS:TMP_GPR).RIP;
FI;

IF (TMP_MODE64 = 1)
    THEN
        IF (TMP_TARGET is not CPU-canonical) THEN #GP(0); FI;
    ELSE
        IF (TMP_TARGET > CS limit) THEN #GP(0); FI;
FI;

(* Check proposed FS/GS segments fall within DS *)
IF (TMP_MODE64 = 0)
    THEN
        TMP_FSBASE := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
        TMP_FSLIMIT := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR + (DS:RBX).FSLIMIT;
        TMP_GSBASE := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
        TMP_GSLIMIT := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR + (DS:RBX).GSLIMIT;
        (* if FS wrap-around, make sure DS has no holes*)
        IF (TMP_FSLIMIT < TMP_FSBASE)
            THEN
                IF (DS.limit < 4GB) THEN #GP(0); FI;
            ELSE
                IF (TMP_FSLIMIT > DS.limit) THEN #GP(0); FI;
        FI;
        (* if GS wrap-around, make sure DS has no holes*)
        IF (TMP_GSLIMIT < TMP_GSBASE)
            THEN
                IF (DS.limit < 4GB) THEN #GP(0); FI;
            ELSE
                IF (TMP_GSLIMIT > DS.limit) THEN #GP(0); FI;
        FI;
    ELSE
        IF (TMP_NOTIFY = 1)
            THEN
                TMP_FSBASE := (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
                TMP_GSBASE := (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
            ELSE
                 TMP_FSBASE := DS:TMP_GPR.FSBASE;
                 TMP_GSBASE := DS:TMP_GPR.GSBASE;
        FI;
        IF ((TMP_FSBASE is not CPU-canonical) or (TMP_GSBASE is not CPU-canonical))
            THEN #GP(0); FI;
FI;

(* Ensure the enclave is not already active and this thread is the only one using the TCS*)
IF (DS:RBX.STATE = ACTIVE))
    THEN #GP(0); FI;

TMP_IA32_U_CET := 0
TMP_SSP := 0

IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        IF ( CR4.CET = 0 )
            THEN
                (* If part does not support CET or CET has not been enabled and enclave requires CET then fail *)
                IF (TMP_SECS.CET_ATTRIBUTES ≠ 0 OR TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0) #GP(0); FI;
        FI;
        (* If indirect branch tracking or shadow stacks enabled but CET state save area is not 16B aligned then fail ERESUME *)
        IF (TMP_SECS.CET_ATTRIBUTES.SH_STK_EN = 1 OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN = 1)
            THEN
                IF (DS:RBX.OCETSSA is not 16B aligned) #GP(0); FI;
        FI;

IF (TMP_SECS.CET_ATTRIBUTES.SH_STK_EN OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN)
    THEN
        (* Setup CET state from SECS, note tracker goes to IDLE *)
        TMP_IA32_U_CET = TMP_SECS.CET_ATTRIBUTES;
        IF (TMP_IA32_U_CET.LEG_IW_EN = 1 AND TMP_IA32_U_CET.ENDBR_EN = 1)
            THEN
                TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.BASEADDR;
                TMP_IA32_U_CET := TMP_IA32_U_CET + TMP_SECS.CET_LEG_BITMAP_BASE;
        FI;

        (* Compute linear address of what will become new CET state save area and cache its PA *)
        IF (TMP_NOTIFY = 1)
            THEN
                TMP_CET_SAVE_AREA = DS:RBX.OCETSSA + TMP_SECS.BASEADDR + (DS:RBX.CSSA) * 16;
            ELSE
                TMP_CET_SAVE_AREA = DS:RBX.OCETSSA + TMP_SECS.BASEADDR + (DS:RBX.CSSA - 1) * 16;
        FI;
        TMP_CET_SAVE_PAGE = TMP_CET_SAVE_AREA & ~0xFFF;

        Check the TMP_CET_SAVE_PAGE page is read/write accessible
        If fault occurs release locks, abort and deliver fault

        (* read the EPCM VALID, PENDING, MODIFIED, BLOCKED and PT fields atomically *)
        IF ((DS:TMP_CET_SAVE_PAGE Does NOT RESOLVE TO EPC PAGE) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).VALID = 0) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).PENDING = 1) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).MODIFIED = 1) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).BLOCKED = 1) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).R = 0) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).W = 0) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVEADDRESS ≠ DS:TMP_CET_SAVE_PAGE) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).PT ≠ PT_SS_REST) OR
        (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS))
            THEN
                #PF(DS:TMP_CET_SAVE_PAGE);
        FI;

        CR_CET_SAVE_AREA_PA := Physical address(DS:TMP_CET_SAVE_AREA)
        IF (TMP_NOTIFY = 1)
            THEN
                IF TMP_IA32_U_CET.SH_STK_EN = 1
                    THEN TMP_SSP = TCS.PREVSSP; FI;
            ELSE
                TMP_SSP = CR_CET_SAVE_AREA_PA.SSP
                TMP_IA32_U_CET.TRACKER = CR_CET_SAVE_AREA_PA.TRACKER;
                TMP_IA32_U_CET.SUPPRESS = CR_CET_SAVE_AREA_PA.SUPPRESS;
                IF ( (TMP_MODE64 = 1 AND TMP_SSP is not CPU-canonical) OR
                     (TMP_MODE64 = 0 AND (TMP_SSP & 0xFFFFFFFF00000000) ≠ 0) OR
                    (TMP_SSP is not 4 byte aligned) OR
                    (TMP_IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH AND TMP_IA32_U_CET.SUPPRESS = 1) OR
                    (CR_CET_SAVE_AREA_PA.Reserved ≠ 0) ) #GP(0); FI;
                FI;
        FI;
FI;

IF (TMP_NOTIFY = 0)
    THEN
        (* SECS.ATTRIBUTES.XFRM selects the features to be saved. *)
        (* CR_XSAVE_PAGE_n: A list of 1 or more physical address of pages that contain the XSAVE area. *)
        XRSTOR(TMP_MODE64, SECS.ATTRIBUTES.XFRM, CR_XSAVE_PAGE_n);

        IF (XRSTOR failed with #GP)
            THEN
                DS:RBX.STATE := INACTIVE;
                #GP(0);
        FI;
FI;

CR_ENCLAVE_MODE := 1;
CR_ACTIVE_SECS := TMP_SECS;
CR_ELRANGE := (TMP_SECS.BASEADDR, TMP_SECS.SIZE);

(* Save sate for possible AEXs *)
CR_TCS_PA := Physical_Address (DS:RBX);
```

```
CR_TCS_LA := RBX;
CR_TCS_LA.AEP := RCX;

(* Save the hidden portions of FS and GS *)
CR_SAVE_FS_selector := FS.selector;
CR_SAVE_FS_base := FS.base;
CR_SAVE_FS_limit := FS.limit;
CR_SAVE_FS_access_rights := FS.access_rights;
CR_SAVE_GS_selector := GS.selector;
CR_SAVE_GS_base := GS.base;
CR_SAVE_GS_limit := GS.limit;
CR_SAVE_GS_access_rights := GS.access_rights;

IF (TMP_NOTIFY = 1)
    THEN
        (* If XSAVE is enabled, save XCR0 and replace it with SECS.ATTRIBUTES.XFRM*)
        IF (CR4.OSXSAVE = 1)
            THEN
                CR_SAVE_XCR0 := XCR0;
                XCR0 := TMP_SECS.ATTRIBUTES.XFRM;
        FI;
FI;

RIP := TMP_TARGET;

IF (TMP_NOTIFY = 1)
    THEN
        RCX := RIP;
        RAX := (DS:RBX).CSSA;
        (* Save the outside RSP and RBP so they can be restored on interrupt or EEXIT *)
        DS:TMP_SSA.U_RSP := RSP;
        DS:TMP_SSA.U_RBP := RBP;
    ELSE
        Restore_GPRs from DS:TMP_GPR;

        (*Restore the RFLAGS values from SSA*)
        RFLAGS.CF := DS:TMP_GPR.RFLAGS.CF;
        RFLAGS.PF := DS:TMP_GPR.RFLAGS.PF;
        RFLAGS.AF := DS:TMP_GPR.RFLAGS.AF;
        RFLAGS.ZF := DS:TMP_GPR.RFLAGS.ZF;
        RFLAGS.SF := DS:TMP_GPR.RFLAGS.SF;
        RFLAGS.DF := DS:TMP_GPR.RFLAGS.DF;
        RFLAGS.OF := DS:TMP_GPR.RFLAGS.OF;
        RFLAGS.NT := DS:TMP_GPR.RFLAGS.NT;
        RFLAGS.AC := DS:TMP_GPR.RFLAGS.AC;
        RFLAGS.ID := DS:TMP_GPR.RFLAGS.ID;
        RFLAGS.RF := DS:TMP_GPR.RFLAGS.RF;
        RFLAGS.VM := 0;
        IF (RFLAGS.IOPL = 3)
            THEN RFLAGS.IF := DS:TMP_GPR.RFLAGS.IF; FI;

        IF (TCS.FLAGS.OPTIN = 0)
            THEN RFLAGS.TF := 0; FI;
        (* If XSAVE is enabled, save XCR0 and replace it with SECS.ATTRIBUTES.XFRM*)
        IF (CR4.OSXSAVE = 1)
            THEN
                CR_SAVE_XCR0 := XCR0;
                XCR0 := TMP_SECS.ATTRIBUTES.XFRM;
        FI;

        (* Pop the SSA stack*)
        (DS:RBX).CSSA := (DS:RBX).CSSA -1;
FI;

(* Do the FS/GS swap *)
FS.base := TMP_FSBASE;
FS.limit := DS:RBX.FSLIMIT;
FS.type := 0001b;
FS.W := DS.W;
FS.S := 1;
FS.DPL := DS.DPL;
FS.G := 1;
FS.B := 1;
FS.P := 1;
FS.AVL := DS.AVL;
FS.L := DS.L;
FS.unusable := 0;
FS.selector := 0BH;

GS.base := TMP_GSBASE;
GS.limit := DS:RBX.GSLIMIT;
GS.type := 0001b;
GS.W := DS.W;
GS.S := 1;
GS.DPL := DS.DPL;
GS.G := 1;
GS.B := 1;
GS.P := 1;
GS.AVL := DS.AVL;
GS.L := DS.L;
GS.unusable := 0;
GS.selector := 0BH;

CR_DBGOPTIN := TCS.FLAGS.DBGOPTIN;
Suppress all code breakpoints that are outside ELRANGE;

IF (CR_DBGOPTIN = 0)
    THEN
        Suppress all code breakpoints that overlap with ELRANGE;
        CR_SAVE_TF := RFLAGS.TF;
        RFLAGS.TF := 0;
        Suppress any MTF VM exits during execution of the enclave;
        Clear all pending debug exceptions;
        Clear any pending MTF VM exit;
    ELSE
        IF (TMP_NOTIFY = 1)
            THEN
                IF RFLAGS.TF = 1
                    THEN pend a single-step #DB at the end of EENTER; FI;
        IF the “monitor trap flag” VM-execution control is set
            THEN pend an MTF VM exit at the end of EENTER; FI;
        ELSE
            Clear all pending debug exceptions;
            Clear pending MTF VM exits;
        FI;
FI;

IF ((CPUID.(EAX=7H, ECX=0):EDX[CET_IBT] = 1) OR (CPUID.(EAX=7, ECX=0):ECX[CET_SS] = 1)
    THEN
        (* Save enclosing application CET state into save registers *)
        CR_SAVE_IA32_U_CET := IA32_U_CET
        (* Setup enclave CET state *)
        IF CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1
            THEN
                CR_SAVE_SSP := SSP
                SSP := TMP_SSP;
        FI;
        IA32_U_CET := TMP_IA32_U_CET;
FI;

(* Assure consistent translations *)
Flush_linear_context;
Clear_Monitor_FSM;
Allow_front_end_to_begin_fetch_at_new_RIP;
```

Flags Affected

RFLAGS.TF is cleared on opt-out entry

Protected Mode Exceptions

#GP(0) If DS:RBX is not page aligned.

If the enclave is not initialized.

If the thread is not in the INACTIVE state.

If CS, DS, ES or SS bases are not all zero.

If executed in enclave mode.

If part or all of the FS or GS segment specified by TCS is outside the DS segment.

If any reserved field in the TCS FLAG is set.

If the target address is not within the CS segment.

If CR4.OSFXSR = 0.

If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.

If CR4.OSXSAVE = 1and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.

If SECS.ATTRIBUTES.AEXNOTIFY ≠ TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN = 0.

#PF(error code) If a page fault occurs in accessing memory.

If DS:RBX does not point to a valid TCS.

If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

64-Bit Mode Exceptions

#GP(0)	If DS:RBX is not page aligned.
	If the enclave is not initialized.
	If the thread is not in the INACTIVE state.
	If CS, DS, ES or SS bases are not all zero.
	If executed in enclave mode.
	If part or all of the FS or GS segment specified by TCS is outside the DS segment.
	If any reserved field in the TCS FLAG is set.
	If the target address is not CPU-canonical.
	If CR4.OSFXSR = 0.
	If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
	If CR4.OSXSAVE = 1and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
	If SECS.ATTRIBUTES.AEXNOTIFY ≠ TCS.FLAGS.AEXNOTIFY and TCS.FLAGS.DBGOPTIN = 0.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If DS:RBX does not point to a valid TCS.
	If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

CHAPTER 9
CODE PREFETCH INSTRUCTION UPDATES

All changes to existing operation are highlighted in green.

PREFETCH*h*—Prefetch Data or Code Into Caches

Opcode/ Instruction	Op/ En	64/32 Bit Mode Support	Description
0F 18 /1 PREFETCHT0 *m8*	M	V/V	Move data from *m8* closer to the processor using T0 hint.
0F 18 /2 PREFETCHT1 *m8*	M	V/V	Move data from *m8* closer to the processor using T1 hint.
0F 18 /3 PREFETCHT2 *m8*	M	V/V	Move data from *m8* closer to the processor using T2 hint.
0F 18 /0 PREFETCHNTA *m8*	M	V/V	Move data from *m8* closer to the processor using NTA hint.
0F 18 /7 PREFETCHIT0 m8	M	V/I	Move code from relative address closer to the processor using IT0 hint.
0F 18 /6 PREFETCHIT1 m8	M	V/I	Move code from relative address closer to the processor using IT1 hint.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	N/A	N/A	N/A

Description

Fetches the line of data or code (instructions' bytes) from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by a locality hint:

- T0 (temporal data)—prefetch data into all levels of the cache hierarchy.
- T1 (temporal data with respect to first level cache misses)—prefetch data into level 2 cache and higher.
- T2 (temporal data with respect to second level cache misses)—prefetch data into level 3 cache and higher, or an implementation-specific choice.
- NTA (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.
- IT0 (temporal code)—prefetch code into all levels of the cache hierarchy.
- IT1 (temporal code with respect to first level cache misses)—prefetch code into all but the first-level of the cache hierarchy.

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte.) Some locality hints may prefetch only for RIP-relative memory addresses; see additional details below. The address to prefetch is NextRIP + 32-bit displacement, where NextRIP is the first byte of the instruction that follows the prefetch instruction itself.

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The PREFETCH*h* instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data or code lines prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes. Additional details of the implementation-dependent locality hints are described in Section 7.4 of Intel® 64 and IA-32 Architectures Optimization Reference Manual.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). A PREFETCH*h* instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCH*h* instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCH*h* instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCH*h* instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

PREFETCHIT0/1 apply when in 64-bit mode with RIP-relative addressing; they stay NOPs otherwise. For optimal performance, the addresses used with these instructions should be the starting byte of a real instruction.

PREFETCHIT0/1 instructions are enumerated by CPUID.(EAX=07H, ECX=01H).EDX.PREFETCHI[bit 14].The encodings stay NOPs in processors that do not enumerate these instructions.

Operation

```
FETCH (m8);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_prefetch(char *p, int i)
```

The argument "*p" gives the address of the byte (and corresponding cache line) to be prefetched. The value "i" gives a constant (_MM_HINT_T0, _MM_HINT_T1, _MM_HINT_T2, or _MM_HINT_NTA, _MM_HINT_IT0, _MM_HINT_IT1) that specifies the type of prefetch operation to be performed.

Numeric Exceptions

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

CHAPTER 10
NEXT GENERATION PERFORMANCE MONITORING UNIT (PMU)

The next generation Performance Monitoring Unit (PMU)[1] offers additional enhancements beyond what is available in both the 12th generation Intel® Core™ processor based on Alder Lake performance hybrid architecture and the 13th generation Intel® Core™ processor:

- Timed PEBS
- New (Hybrid) Enumeration Architecture
  - General-Purpose Counters
  - Fixed-Function Counters
  - Architectural Performance Monitoring Events
    - Topdown Microarchitecture Analysis (TMA) Level 1 Architectural Performance Monitoring Events
  - Non-Architectural Capabilities

10.1 NEW ENUMERATION ARCHITECTURE

A new Architectural Performance Monitoring Extended Leaf 23H is added to the CPUID instruction for enhanced enumeration of PMU architectural features; see Chapter 1, "Architectural Performance Monitoring Extended Leaf (Output depends on ECX input value)" on page 25 for details.

NOTE

CPUID leaf 0AH continues to report useful attributes, such as architectural performance monitoring version ID and counter width (# bits).

CPUID leaf 23H enhances previous enumeration of PMU capabilities:

- Employs CPUID sub-leafing to accommodate future PMU extensions.
- Exposes hybrid resources per core-type.
- Introduces a bitmap enumeration of general-purpose counters availability.
- A bitmap enumeration of fixed-function counters availability.
- A bitmap enumeration of architectural performance monitoring events.

Processors that support this enhancement set CPUID.(EAX=07H, ECX=01H):EAX.ArchPerfmonExt[bit 8].

Additionally, the IA32_PERF_CAPABILITIES MSR enhances enumeration for PMU non-architectural features (see Section 10.1.6).

10.1.1 CPUID Sub-Leafing

CPUID leaf 23H contains additional architectural PMU capabilities. This leaf supports sub-leafing, providing each distinct PMU feature with an individual sub-leaf for enumerating its details.

The availability of sub-leaves is enumerated via CPUID.(EAX=23H, ECX=0H):EAX. For each bit *n* set in this field, sub-leaf *n* under CPUID leaf 23H is supported.

[1] The next generation PMU incorporates PEBS_FMT=5h as described in Section 19.6.2.4.2 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

10.1.2 Reporting of Hybrid Resources

CPUID leaf 23H provides a true-view of per core-type PMU capabilities. For hybrid processors, those that set CPUID.(EAX=07H, ECX=0H):EDX.Hybrid[bit 15], the new leaf reports the actual support of the individual core-type the CPUID instruction was executed on. This implies that values returned by this leaf may vary based on the core-type. This applies to all sub-leaves and registers.

Conversely, CPUID leaf 0AH provides a maximum common set of capabilities across core types when a feature is not supported by all core types.

NOTE

Locating a PMU feature under CPUID leaf 023H alerts software that the features may be not supported uniformly across all core types.

10.1.3 General-Purpose Counters Bitmap

CPUID.(EAX=23H, ECX=01H):EAX reports a bitmap for available general-purpose counters. (CPUID leaf 0AH reports only the total number of general-purpose counters).

This capability enables a virtual-machine monitor to reserve lower-index counters for its own use, while exposing higher-index counters to guest software. This is especially important should the general-purpose counters not be fully homogeneous.

Software should utilize the new bitmap reporting, including for detecting the number of available general-purpose counters. To facilitate this transition, the number of general-purpose counters in CPUID leaf 0AH will not go beyond eight, even if the processor has support for more than eight general-purpose counters.

10.1.4 Fixed-Function Counters Hybrid Bitmap

CPUID.(EAX=23H, ECX=01H):EBX reports a bitmap for available fixed-function counters. (CPUID leaf 0AH reports the common number of contiguous fixed-function counters in addition to a common bitmap of fixed-function counters availability.)

This capability enables privileged software to expose per core-type enumeration of fixed-function counters. This is especially important should the fixed-function counters not be available on all logical processors.

10.1.5 Architectural Performance Monitoring Events Bitmap

CPUID.(EAX=23H, ECX=03H):EAX provides a true-view of per core-type available architectural performance monitoring events. For each bit *n* set in this field, the processor supports Architectural Performance Monitoring Event of index *n* (positive polarity).

Conversely, CPUID leaf 0AH provides a maximum common set of architectural performance monitoring events supported by all core types, where if bit *n* is set, it denotes the processor does not necessarily support Architectural Performance Monitoring Event of index *n* on all logical processors (negative polarity).

10.1.6 Non-Architectural Performance Capabilities

The IA32_PERF_CAPABILITIES MSR provides enumeration of non-architectural PMU features. Some fields in the MSR are of type "common," meaning that they report the same value on all cores in a hybrid part. Other fields have type "hybrid" and report values that may differ across cores (the value reported on each core pertains only to that core). Table 10-1 enumerates the fields in the MSR and indicates the type of each.

Table 10-1. IA32_PERF_CAPABILITIES Hybrid Enumeration

Field Name	Bits[1]	Type
LBR FMT	5:0	Common
PEBS Trap	6	Common
PEBS Arch Regs	7	Common
PEBS FMT	11:8	Common
Freeze while SMM	12	Common
Full Write	13	Common
PEBS Baseline	14	Common
Perf Metrics Available	15	Hybrid
PEBS Output PT Available	16	Hybrid
PEBS Timing Info	17	Common

NOTES:

1. For more information on bit 17, see Section 10.3.1. For information on other bits, see Section 19.8 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

10.2 NEW ARCHITECTURAL EVENTS

Next generation PMU introduces additional architectural performance monitoring events with details summarized in Table 10-2. Descriptions are provided in the sub-sections that follow.

Table 10-2. New Architectural Performance Monitoring Events

Bit Position in CPUID.0AH.EBX and CPUID.023H.03H.EAX	Event Name	Event Select	UMask
8	Topdown Backend Bound	A4H	02H
9	Topdown Bad Speculation	73H	00H
10	Topdown Frontend Bound	9CH	01H
11	Topdown Retiring	C2H	02H

10.2.1 Topdown Microarchitecture Analysis Level 1

10.2.1.1 Topdown Backend Bound—Event Select A4H, Umask 02H

This event counts a subset of the Topdown Slots event that was not consumed by the back-end pipeline due to lack of back-end resources, as a result of memory subsystem delays, execution unit limitations, or other conditions.

The count may be distributed among unhalted logical processors who share the same physical core, in processors that support Intel® Hyper-Threading Technology.

Software can use this event as the numerator for the Backend Bound metric (or top-level category) of the Topdown Microarchitecture Analysis method.

10.2.1.2 Topdown Bad Speculation—Event Select 73H, Umask 00H

This event counts a subset of the Topdown Slots event that was wasted due to incorrect speculation as a result of incorrect control-flow or data speculation. Common examples include branch mispredictions and memory ordering clears.

The count may be distributed among impacted logical processors who share the same physical core, for some processors that support Intel Hyper-Threading Technology.

Software can use this event as the numerator for the Bad Speculation metric (or top-level category) of the Topdown Microarchitecture Analysis method.

10.2.1.3 Topdown Frontend Bound—Event Select 9CH, Umask 01H

This event counts a subset of the Topdown Slots event that had no operation delivered to the back-end pipeline due to instruction fetch limitations when the back-end could have accepted more operations. Common examples include instruction cache misses and x86 instruction decode limitations.

The count may be distributed among unhalted logical processors who share the same physical core, in processors that support Intel Hyper-Threading Technology.

Software can use this event as the numerator for the Frontend Bound metric (or top-level category) of the Topdown Microarchitecture Analysis method.

10.2.1.4 Topdown Retiring—Event Select C2H, Umask 02H

This event counts a subset of the Topdown Slots event that is utilized by operations that eventually get retired (committed) by the processor pipeline. Usually, this event positively correlates with higher performance as measured by the instructions-per-cycle metric.

Software can use this event as the numerator for the Retiring metric (or top-level category) of the Topdown Microarchitecture Analysis method.

10.3 PROCESSOR EVENT BASED SAMPLING (PEBS) ENHANCEMENTS

10.3.1 Timed Processor Event Based Sampling

Timed Processor Event Based Sampling (Timed PEBS) enables recording of time in every PEBS record. It extends all PEBS records with timing information in a new “Retire Latency” field that is placed in the Basic Info group of the PEBS record as shown in Table 10-3.

Table 10-3. Basic Info Group

Offset	Field Name	Bits
0x0	Record Format	[31:0]
	Retire Latency	[47:32]
	Record Size	[63:48]
0x8	Instruction Pointer	[63:0]
0x10	Applicable Counters	[63:0]
0x18	TSC	[63:0]

The Retire Latency field reports the number of unhalted core clocks between the retirement of the current instruction (as indicated by the Instruction Pointer field of the PEBS record) and the retirement of the prior instruction. All ones are reported when the number exceeds 16 bits.

Processors that support this enhancement set a new bit: IA32_PERF_CAPABILITIES.PEBS_TIMING_INFO[bit 17].

CHAPTER 11
LINEAR ADDRESS SPACE SEPARATION (LASS)

This chapter describes a new feature called **linear address space separation** (**LASS**).

11.1 INTRODUCTION

Chapter 4 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A describes **paging**, which is the process of translating linear addresses to physical addresses and determining, for each translation, the linear address's **access rights**; these determine what accesses to a linear address are allowed.

Every access to a linear address is either a **supervisor-mode access** or a **user-mode access**. A linear address's access rights include an indication of whether address is a **supervisor-mode address** or a **user-mode address**. Paging prevents user-mode accesses to supervisor-mode addresses; in addition, there are features that can prevent supervisor-mode accesses to user-mode addresses. (These features are supervisor-mode execution prevention — SMEP — and supervisor-mode access prevention — SMAP.) In most cases, the blocked accesses cause page-fault exceptions (#PF); for some cases (e.g., speculative accesses), the accesses are dropped without fault.

With these mode-based protections, paging can prevent malicious software from directly reading or writing memory inappropriately. To enforce these protections, the processor must traverse the hierarchy of paging structures in memory. Unprivileged software can use timing information resulting from this traversal to determine details about the paging structures, and these details may be used to determine the layout of supervisor memory.

Linear-address space separation (LASS) is an independent mechanism that enforces the same mode-based protections as paging but without traversing the paging structures. Because the protections enforced by LASS are applied before paging, "probes" by malicious software will provide no paging-based timing information.

LASS is based on a linear-address organization established by many operating systems: all linear addresses whose most significant bit is 0 ("low" or "positive" addresses) are user-mode addresses, while all linear addresses whose most significant bit is 1 ("high" or "negative" addresses) are supervisor-mode addresses. An operating system should enable LASS only if it uses this organization of linear addresses.

11.2 ENUMERATION AND ENABLING

Support for LASS is enumerated with CPUID.(EAX=07H.ECX=1):EAX.LASS[bit 6].

If a processor enumerates CPUID.(EAX=07H.ECX=1):EAX.LASS[bit 6] as 1, software can set CR4.LASS[bit 27]. Setting CR4.LASS to 1 enables LASS in IA-32e mode (when IA32_EFER.LMA = 1). LASS is not used in legacy mode, even if CR4.LASS = 1.

11.3 OPERATION OF LINEAR-ADDRESS SPACE SEPARATION

This section describes the operation of linear-address space separation (LASS). The discussion in this section applies only if IA32_EFER.LMA = CR4.LASS = 1. (If either of those control bits is zero, LASS does not apply.)

As indicated in Section 11.1, LASS enforces mode-based protections similar to those enforced by paging. Violations of these protections are called **LASS violations**. The processor will consult neither the paging structures nor the TLBs for an access that causes a LASS violation.

Like paging, LASS violations typically result in faults. Instead of page faults (#PF), an access causing a LASS violation results in the same fault that would occur if the access used an address that was not canonical relative to the current paging mode. In most cases, this is a general protection exception (#GP); for stack accesses (those due to stack-oriented instructions, as well as accesses that implicitly or explicitly use the SS segment register), it would be a stack fault (#SS).

Some accesses do not cause faults when they would violate the mode-based protections established by paging. These include prefetches (e.g., those resulting from execution of one of the PREFETCH*h* instructions), executions of the CLDEMOTE instruction, and accesses resulting from the speculative fetch or execution of an instruction. Such an access may cause a LASS violation; if it does, the access is not performed but no fault occurs. (When such an access would violate the mode-based protections of paging, the access is not performed but no page fault occurs.)

In 64-bit mode, LASS violations have priority just below that of canonicality violations; in compatibility mode, they have priority just below that of segment-limit violations.

The remainder of this section describes how LASS applies to different types of accesses to linear addresses. Chapter 4, "Paging," of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A provides full definitions of these access types. The sections below discuss specific LASS violations based on bit 63 of a linear address. For a linear address with only 32 bits (or 16 bits), the processor treats bit 63 as if it were 0.

11.3.1 Data Accesses

A linear-address access is a **data access** if it is not for the fetch of an instruction. Such an access is a **user-mode access** if CPL = 3 and the access is not one that implicitly accesses a system data structure (e.g., the global descriptor table); it is a **supervisor-mode access** if CPL < 3 or if it implicitly accesses a system data structure.[1]

A user-mode data access causes a LASS violation if it would access a linear address of which bit 63 is 1. It is expected that the operating system will configure paging so that any such address is a supervisor-mode address.

A supervisor-mode data access may cause a LASS violation if it would access a linear address of which bit 63 is 0. It is expected that the operating system will configure paging so that any such address is a user-mode address.

A supervisor-mode data access causes a LASS violation only if supervisor-mode access protection is enabled (because CR4.SMAP = 1) and either RFLAGS.AC = 0 or the access implicitly accesses a system data structure.

11.3.2 Instruction Fetches

Instruction fetches are always performed with linear addresses. An instruction fetch is **user-mode** if CPL = 3 and is supervisor mode if CPL < 3.

A user-mode instruction fetch causes a LASS violation if it would fetch an instruction using a linear address of which bit 63 is 1.

A **supervisor-mode** instruction fetch causes a LASS violation if it would accesses a linear address of which bit 63 is 0.

(Paging blocks supervisor-mode instruction fetches from user-mode linear addresses only if supervisor-mode execution protection has been enabled by setting CR4.SMEP to 1. Such instructions fetches cause LASS violations regardless of the setting of CR4.SMEP.)

It was noted earlier that LASS violations produce the same faults as canonicality violations and with a similar priority. LASS violations differ from canonicality violations in particular way as regards instruction flow. An instruction that loads RIP (a branch instruction) causes a general-protection exception (#GP) as a fault if it would load RIP with a value that is not canonical relative to the current paging mode; RIP is not updated, and the fault is reported on the branch instruction. In contrast, branch instructions do not check the target RIP for LASS violations, and thus LASS does not prevent branch instructions from completing. Fetch of the next instruction (at the target RIP) may cause a LASS violation and a #GP. In that case, the fault is reported on the branch target, not the branch instruction.

[1] The WRUSS instruction is an exception; although it can be executed only if CPL = 0, the processor treats its shadow-stack accesses as user accesses.

CHAPTER 12
VIRTUALIZATION OF THE IA32_SPEC_CTRL MSR

12.1 INTRODUCTION

This chapter describes the VMX extension that supports virtualization of the IA32_SPEC_CTRL MSR. This feature allows a virtual-machine monitor (VMM) to specify that certain bits of the MSR cannot be modified by guest software.

12.2 VMCS CHANGES

Section 12.2.1 identifies the new VM-execution control that enables the feature, and Section 12.2.2 enumerates the new VMCS fields that support the feature.

12.2.1 New VMX Control

Bit 7 of the tertiary processor-based VM-execution controls is defined as "virtualize IA32_SPEC_CTRL." This control allows software in VMX non-root operation to access the IA32_SPEC_CTRL MSR under control of a VMM. Section 12.3 provides more details.

12.2.2 New VMCS Fields

Two new VM-execution control fields are defined:

- The IA32_SPEC_CTRL mask (encoding pair 204AH/204BH). Setting a bit in this field prevents guest software from modifying the corresponding bit in the IA32_SPEC_CTRL MSR.
- The IA32_SPEC_CTRL shadow (encoding pair 204CH/204DH). This field contains the value that guest software expects to be in the IA32_SPEC_CTRL MSR.

See Section 12.3 for a discussion of how these fields are used in VMX non-root operation. These fields exist on any processor that supports the 1-setting of the "virtualize IA32_SPEC_CTRL" VM-execution control.

12.3 CHANGES TO VMX NON-ROOT OPERATION

If the "virtualize IA32_SPEC_CTRL" VM-execution control is 1, the operation of reads and writes to the IA32_SPEC_CTRL MSR is modified:

- Any read from the IA32_SPEC_CTRL MSR (e.g., by RDMSR) that does not cause a fault or a VM exit returns the value of the IA32_SPEC_CTRL shadow (see Section 2.2).
- Any write to the IA32_SPEC_CTRL MSR (e.g., by WRMSR) operates as follows:
  - If the instruction would fault due to CPL, such a fault occurs.
  - If the instruction would cause a VM exit, such a VM exit occurs.
  - The instruction will attempt to write the IA32_SPEC_CTRL MSR using the instruction's source operand, but it will attempt to modify only those bits in positions corresponding to bits cleared in the IA32_SPEC_CTRL mask.

    Specifically, the instruction attempts to write the MSR with the following value: (MSR_VAL & ISC_MASK) OR (SRC & NOT ISC_MASK), where MSR_VAL is the original value of the MSR, ISC_MASK is the IA32_SPEC_CTRL mask, and SRC is the instruction's source operand.

    Any fault that would result from writing that value to the MSR (e.g., due to a reserved-bit violation) occurs normally. Otherwise, the value is written to the MSR.

Such a write to the MSR will have any side effects that would occur normally had the MSR been written with the value indicated above (including any side effects that may result from writing unchanged values to the masked bits).

- The source operand is written to the IA32_SPEC_CTRL shadow.

CHAPTER 13
REMOTE ATOMIC OPERATIONS IN INTEL ARCHITECTURE

13.1 INTRODUCTION

Remote Atomic Operations (RAO) are a set of instructions to improve synchronization performance. RAO is especially useful in multiprocessor applications that have a set of characteristics commonly found together:

- A need to update, i.e., read and modify, one or more variables atomically, e.g., because multiple processors may attempt to update the same variable simultaneously.
- Updates are not expected to be interleaved with other reads or writes of the variables.
- The order in which the updates happen is unimportant.

One example of this scenario is a multiprocessor histogram computation, where multiple processors cooperate to compute a shared histogram, which is then used in the next phase of computation. This is described in more detail in Section 13.8.1.

RAO instructions aim to provide high performance in this scenario by:

- Atomically updating memory without returning any information to the processor itself.
- Relaxing the ordering of RAO instructions with respect to other updates or writes to the variables.

RAO instructions are defined such that, unlike conventional atomics (e.g., LOCK ADD), their operations may be performed closer to memory, such as at a shared cache or memory controller. Performing operations closer to memory reduces or even eliminates movement of data between memory and the processor executing the instruction. They also have weaker ordering guarantees than conventional atomics. This facilitates execution closer to memory, and can also lead to reduced stalls in the processor pipeline. These properties mean that using RAO instead of conventional atomics may provide a significant performance boost for the scenario outlined above.

13.2 INSTRUCTIONS

The current set of RAO instructions can be found in Chapter 2, “Instruction Set Reference, A-Z.” These instructions include integer addition and bitwise AND, OR, and XOR. These operations may be performed on 32-bit (doubleword) or 64-bit (quadword) data elements. The destination, which is also one of the inputs, is always a location in memory. The other input is a general-purpose register, ry, in Table 13-1. The instructions do not change any registers or flags.

Table 13-1. RAO Instructions

Instruction	Operation	Function	Data Types
AADD	Atomic addition	mem = mem + ry	Doubleword, quadword
AAND	Atomic bitwise AND	mem = mem AND ry	Doubleword, quadword
AOR	Atomic bitwise OR	mem = mem OR ry	Doubleword, quadword
AXOR	Atomic bitwise XOR	mem = mem XOR ry	Doubleword, quadword

13.3 ALIGNMENT REQUIREMENTS

The memory location updated by an RAO instruction must be naturally aligned. That is, a doubleword update must be four-byte aligned and a quadword update must be eight-byte aligned. This facilitates implementations closer to memory; otherwise, a single update may straddle a cache line boundary.

13.4 MEMORY ORDERING

RAO instructions have weaker memory ordering guarantees than conventional atomic instructions. Thus, other instructions are not ordered with respect to RAO instructions as they are with conventional atomics.

More specifically, the memory operations from RAO instructions follow the Write Combining (WC) memory protocol. From software's point of view, they behave similarly to non-temporal stores. Unlike non-temporal stores, RAO instructions update a memory location, i.e., use the value in that location as an input, rather than overwrite the current contents. Another critical difference is that with RAO, the memory location may be cached upon completion of the instruction.

RAO instructions are not reordered with other memory accesses to the same memory location. That is, reads, writes, and RAO instructions to the same location by the same processor will execute in program order.

However, RAO instructions may be reordered with certain memory accesses to other memory locations. In particular, RAO instructions may be reordered with writes or RAO instructions to other memory locations. This means, for example, that if a processor executes a set of RAO instructions to a set of distinct addresses, those instructions may appear to update memory in any order.

If a stronger ordering is required, software should use a fencing operation such as those implemented by the LFENCE, SFENCE, and MFENCE instructions. However, note that RAO instructions are not ordered with respect to younger LFENCE instructions since they do not load data from memory into the processor.

13.5 MEMORY TYPE

RAO instructions are restricted to operating on Write Back (WB) memory. Other memory types place restrictions on the writing of and/or cacheability of data, which conflicts with RAO instructions' ability to cache data. Use of an RAO instruction to access non-WB memory results in a general-protection exception (#GP).

13.6 WRITE COMBINING BEHAVIOR

RAO implementations that execute updates closer to memory require interconnect traffic between a processor and the memory subsystem. To reduce such traffic, and increase the throughput of RAO operations, implementations may combine multiple RAO memory operations before execution. This is similar to how multiple writes via a WC protocol may combine before going to memory.

Implementations that combine RAO instructions take advantage of spatial locality, i.e., that a cache line contains multiple data elements, and that separate instructions may update distinct elements in a given cache line. For example, a first RAO instruction may update the first element in a cache line, and a second RAO instruction may update the third element.

Implementations may have restrictions on combining operations. For example, they may only be able to combine operations doing the same type of update (e.g., addition) and/or the same data element size.

Operations to the same cache line that are not combined must be serialized, and this could hurt performance. For example, an operation to a given cache line may need to complete before a second operation to that cache line may begin; otherwise, the memory system could have multiple concurrent accesses from the same processor to the same cache line, and some implementations do not support this.

13.7 PERFORMANCE EXPECTATIONS

RAO instructions are expected to provide higher performance than conventional atomics under certain conditions. The actual performance depends on both the implementation and the data access pattern for the memory location (at the cache line granularity) updated with RAO instructions.

13.7.1 Interaction Between RAO and Other Accesses

As discussed in Section 13.4, weak ordering allows RAO instructions to be reordered with respect to other memory operations. This is a key difference from conventional atomics, which follow strong memory ordering, and can allow a processor to execute RAO instructions with higher throughput. However, only certain reordering is allowed. If a fence is used to enforce stronger ordering, or if a processor interleaves RAO updates with reads of the same memory location, for example, this may result in serialized accesses, and hurt performance. If software performs an RAO update to a memory location, and soon after reads that memory location, then the read needs to wait for the update to complete. If the RAO is done close to memory, then the cache closest to the processor may not hold a copy of the cache line after the RAO instruction executes, and the read may need to access a cache farther away from the processor, or even go all the way to memory.

Mixing of RAO updates to a given memory location from one or more processors with non-RAO accesses to the same memory location can also reduce the benefits of RAO. Implementations that perform RAO updates close to memory can reduce data movement between a series of RAO updates to the same location. However, a non-RAO access may cause a processor to cache the data close to itself; a subsequent RAO instruction from another processor may require the line to be moved to a lower level of the cache hierarchy. Therefore, interleaving RAO and non-RAO accesses to a given memory location can reduce or eliminate the data movement and/or performance benefits of RAO.

13.7.2 Updates of Contended Data

Contended data is defined as data for which the memory system has memory accesses from multiple processors in-flight simultaneously. That is, for contended data, the memory system is at some point in time handling at least two accesses from different processors. Contended read-only data does not present a fundamental performance problem, but if at least one of the contending processors attempts to write the data, e.g., perform an update on it, the writer needs exclusive access to the data. Gaining exclusive access can be costly, in terms of latency and traffic; in a system with caches, hardware must invalidate all other copies of the data to provide a processor exclusive access.

For software performing a set of contended updates to a memory location with conventional atomic instructions, data may “ping-pong” between processors. As each processor executes its update, it will obtain exclusive access to the data, perform its update, and then have to send its new version of the data to the next processor wanting to update it. The time to pass data from one processor to another, and the time that a processor takes to perform its atomic update, limits the throughput in this scenario.

In contrast, if software uses RAO for such contended updates, and if the implementation performs the updates in a central location such as a shared cache or at the memory controller, then this bottleneck is alleviated. In such a scenario, each update will not have to fetch the current contents of the memory location or invalidate any other copies of the data because the only valid copy is already at the hardware performing the update. The only fundamental limit to the throughput in this case is the time taken for each update. Therefore, we may expect that for updates to contended lines, throughput is much higher with RAO. Further, reducing data movement means reducing traffic between processors and memory. This may improve the performance of other memory accesses.

13.7.3 Updates of Uncontended Data

In contrast to contended data, uncontended data is data that is accessed by only a single processor or by multiple processors, but far enough apart in time that at most a single memory access is executed at a time.

For uncontended data accessed by multiple processors, most of the above discussion about contended data still applies. However, the frequency of updates is by definition lower for uncontended data. Therefore, the performance benefits of RAO are expected to be lower in this situation.

For data accessed by only a single processor, data movement between processors is not an issue, and conventional atomics can take advantage of the processor's caches. Performance may still be impacted by the strong ordering of conventional atomics; memory accesses to other memory locations may not be reordered with these instructions. If software uses RAO instructions instead, the weaker ordering may provide some performance benefits. However, if an implementation performs RAO updates closer to memory, it may not take advantage of all of the processor's caches, and may even require removing the data from some of those caches. This could lead to an increase in data movement, and potentially lower performance. Of course, if software is aware that only a single processor will access the data, then it does not need to use atomic updates, but it may not always be so aware.

13.8 EXAMPLES

13.8.1 Histogram

Histogram is a common computational pattern, including in multiprocessor programming, but achieving an efficient parallel implementation can be tricky. In a conventional histogram computation, software sweeps over a set of input values; it maps each input value to a histogram bin, and increments that bin.

Common multiprocessor histogram implementations partition the inputs across the processors, so each processor works on a subset of the inputs. Straightforward implementations have each processor directly update the shared histogram. To ensure correctness, since multiple processors may attempt updates to the same histogram bin simultaneously, the updates must use atomics. As described above, using conventional atomics can be expensive, especially when we have highly contended cache lines in the histogram. That may occur for small histograms or for histograms where many inputs map to a small number of histogram bins.

A common alternative approach uses a technique called privatization, where each processor gets its own "local" histogram; as each processor works on its subset of the inputs, it updates its local histogram. As a final "extra" step, software must accumulate the local histograms into the globally shared histogram, a step called a reduction. This reduction step is where processors synchronize and communicate; using it allows the computation of local histograms to be embarrassingly parallel and require no atomics or inter-processor communication, and can often lead to good performance. However, privatization has downsides:

- The reduction step can take a lot of time if the histogram has many bins.
- The time for a reduction is relatively constant regardless of the number of processors. As the number of processors grows, therefore, the fraction of time spent on the reduction tends to grow.
- The local histograms require extra memory, and that memory footprint grows with the number of processors.
- The reduction is an "extra" step that complicates the software.

With RAO, software can use the simpler multiprocessor algorithm and achieve reliably good performance. The following pseudo-code lists a RAO-based histogram implementation.

```
int *histogram; // "histogram" is a global histogram array

// in each processor:
double *data; // "data" is a per-processor array, holding a subset of all inputs
data = get_data(); // populate "data" values

for (size_t i = 0; i < data_size; ++i) {
    int bin = map(data[i]); // map data[i] to a histogram bin
    _aadd(&histogram[bin], 1); // RAO AADD instruction
}
```

The above code can provide good performance under various scenarios, i.e., sizes of histograms and biases in which histogram bins are updated. RAO avoids data "ping-ponging" between processors, even under high contention. Further, the weak ordering of RAO allows a series of AADD instructions to overlap with each other in the pipeline, and thus provide for instruction level parallelism.

In addition to the performance benefits, the RAO code is simple and is thus easier to maintain.

While we specifically show and discuss histogram above, this computation pattern is very common, e.g., software packet processing workloads exhibit this in how they track statistics of the packets. Other algorithms exhibiting this pattern should similarly see benefits from RAO.

13.8.2 Interrupt/Event Handler

An interrupt/event handler, running either in a dedicated thread or preemptively in a specific processor, notifies a set of receivers (e.g., all processors or threads in a waiting list) of the occurrence of an event by atomically setting flags in the receivers' specific data fields. The example below shows how this may be done with RAO instructions.

```
// One processor sets event bits to notify other processors:
01: void handle_event(event_t *e) {
02:    uint32_t event_bits = process_event(e);
03:    for (int i = 0; i < num_of_receivers; ++i) {
04:        core_t *core = receivers[i];
05:        _aor(&core->flags, event_bits); // RAO AOR instruction
06:        if (some_condition) {
07:            _aor(&core->extra_flags, event_bits); // combining of RAO could occur
08:        }              // if "extra_flags" and "flags" are in the same cache line
09:    }
10:    _mm_sfence();  // ensure event_bits are visible before leaving the handler
11: }

// In other processors:
12: if (my_core->flags & SOME_EVENT) {
13:    ...... // react to the occurrence of SOME_EVENT
14:    clear_bits(&my_core->flags, SOME_EVENT);
15: }
```

With conventional atomics (e.g., LOCK OR), a significant portion of execution time of handle_event would be spent accessing core->flags (line 5) and core->extra_flags (line 7). It is likely that when handle_event begins, the two fields are in another processor's cache, e.g., if that processor updated some bits in the fields. Therefore, the data would need to migrate to the cache of the processor executing handle_event.

In contrast, for the above code example, for RAO implementations that perform updates close to memory, the RAO AOR instruction should reduce data movement of core->flags and core->extra_flags and thus result in a lower execution latency. Further, when other processors later access these fields (lines 12-15), they will also benefit from a lower latency due to reduced data movement, since they may get the data from a more central location.

Also note that since the order of notifications does not matter in this case, the function further takes advantage of RAO's weak ordering, allowing multiple RAO AOR instructions to be executed concurrently. It does, however, include a memory fence at the end (line 10), to ensure that all updates are visible to all processors before leaving the handler.