Intel® 64 Architecture Processor Topology Enumeration

April 2023

Reference Number: 337015-002, Revision: 2.0

Notices & Disclaimers

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.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exception that a) you may publish an unmodified copy and b) code included in this document is licensed subject to the Zero-Clause BSD open source license (0BSD), https://opensource.org/licenses/0BSD. You may create software implementations based on this document and in compliance with the foregoing that are intended to execute on the Intel product(s) referenced in this document. No rights are granted to create modifications or derivatives of this document.

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

Contents

1 Intel® 64 Architecture Processor Topology Enumeration 5
- 1.1 Introduction 5
  - 1.1.1 Glossary 6
- 1.2 Unique APIC ID in a Multi-Processor System 7
  - 1.2.1 x2APIC ID/APIC ID Topology Bit Layouts 7
  - 1.2.2 Algorithm Enumerating Topology Discovery Method 9
  - 1.2.3 CPUID Leaf 1FH Supersedes CPUID Leaf 0BH 11
  - 1.2.4 Application Software and Topology Enumeration 11
- 1.3 System Topology Enumeration Using CPUID Leaf 1FH or CPUID Leaf 0BH 11
  - 1.3.1 Implementing a Three-Domain Topology 13
  - 1.3.2 Implementing Topology Domains Beyond Three 15
  - 1.3.3 Generating the Topology Masks from the Acquired Shift Values 18
- 1.4 Cache Topology Enumeration Using CPUID Leaf 04H 19
  - 1.4.1 Cache ID Extraction Parameters 20
- 1.5 TLB Topology Enumeration Using CPUID Leaf 18H 21
  - 1.5.1 TLB ID Extraction Parameters 23
- 1.6 Legacy Enumeration Without Extended Topology Leaves 24
  - 1.6.1 Legacy Extraction Algorithm 25
- 1.7 Analyzing Topology Enumeration Result and Customization 26
  - 1.7.1 Dynamic Software Visibility of Topology Enumeration 26

Figures

- 1-1 Choosing CPUID Leaf Information for System Topology Enumeration 10
- 1-2 Flow Chart Demonstrating How to Implement a Three-Domain Topology 15
- 1-3 Flow Chart Demonstrating How to Implement a Topology Greater Than Three Domains 16
- 1-4 Procedures to Extract Sub-IDs from the Initial APIC ID of Each Logical Processor 24

Tables

- 1-1 Example A: Topology Bit Layout 8
- 1-2 Example A Masks and Shifts 8
- 1-3 Reference for CPUID Leaf 1FH 12
- 1-4 Example B: A Topology Bit Layout With More Than Three Domains 13
- 1-5 Example B Converted to a Three-Domain Topology 14
- 1-6 Example C: A Five-Domain Topology Bit Layout 17
- 1-7 Example C Converted to a Four-Domain Topology 17
- 1-8 Example C Converted to a Three-Domain Topology 17
- 1-9 Example D: A Multi-Level Topology With Multiple Unknown Domains 17
- 1-10 Example D Converted to a Five-Domain Topology 18
- 1-11 Example D Converted to a Four-Domain Topology 18
- 1-12 Example D Converted to an Alternative Four-Domain Topology 18
- 1-13 Example D Converted to a Three-Domain Topology 18
- 1-14 A Three-Domain Topology With Logical_Processor_Shift = 1 and Core_Shift = 4 18
- 1-15 Reference for CPUID Leaf 04H 19
- 1-16 Reference for CPUID Leaf 18H 21

1-17 Reference for CPUID Leaf 01H 25

Revision History

Document Number	Revision Number	Description	Date
329176-001	1.1	• Updates to original paper.	January 2018
329176-002	2.0	• Updates to add clarity, new examples, and information on CPUID leaf 1FH.	April 2023

§

CHAPTER 1
INTEL® 64 ARCHITECTURE PROCESSOR TOPOLOGY ENUMERATION

1.1 INTRODUCTION

Processor topology information is important for a number of processor-resource management practices, ranging from task/thread scheduling, licensing policy enforcement, affinity control/migration, etc. Topology information of the cache hierarchy can be important to optimizing software performance. This technical paper covers the topology enumeration algorithm for single-socket to multiple-socket platforms using Intel® 64 and IA-32 processors. The initial 8-bit APIC ID was extended to use the 32-bit x2APIC ID in 2008 to support systems with more than 256 logical processors in a coherent domain. Most platforms today support the x2APIC ID, and in the near future may only support the x2APIC ID.

The majority of Intel® Architecture Processors shipping today provide one or more forms of hardware multi-threading support: multi-core and/or simultaneous multi-threading (SMT), the latter introduced as Intel® Hyper-Threading Technology in 2002. From a processor hardware perspective, the physical package of an Intel 64 processor can support SMT and multi-core. Consequently, a physical processor is effectively a hierarchically ordered collection of logical processors with some forms of shared system resources (e.g., memory, bus/system links, caches). From a platform hardware perspective, hardware multi-threading that exists in a multi-processor system may consist of two or more physical processors organized in either a uniform or non-uniform configuration with respect to the memory subsystem.

Application programming using hardware multi-threading features must follow the programming models and software constructs provided by the underlying operating system. For example, an OS scheduler generally assigns a software task from a queue using hardware resource at the granularity of a logical processor. An operating system may define its own data structure and provide services to applications that allow them to customize the assignment between task and logical processor via an affinity construct for multithreaded applications. The operating system and the software stack underneath an application (e.g., the BIOS, the OS loader) also play significant roles in bringing up the hardware multi-threading features and configuring the software constructs defined by the operating system.

The CPUID instruction in Intel 64 architecture defines a rich set of information to assist BIOS, OS, and applications to query processor topology that is needed for efficient operation by each respective member of the software stack. Generally, the BIOS needs to gather topology information of a physical processor, determine how many physical processors are present in the system, prepare the necessary software constructs related to system topology, and pass along the system topology information to the next layer of the software stack that takes over control of the system. The operating system and the application layers have a wide range of uses for topology information. This document covers several common software usages by operating systems and applications for using CPUID to analyze processor topology in a single-processor or multi-processor system.

The primary software usage of processor topology enumeration deals with querying and identifying the hierarchical relationship of logical processor, processor cores, and physical packages in a single-processor or multi-processor system. This usage is referred to as system topology enumeration. System topology enumeration may be needed by the operating system or certain applications to implement licensing policy based on physical processors. It is used by the operating system to implement efficient task-scheduling, minimize thread migration, configure application thread management interfaces, and configure memory allocation services appropriate to the processor/memory topology. Multi-threaded applications need system topology information to determine optimal thread binding, manage memory allocation for optimal locality, and improve performance scaling in multi-processor systems.

Intel 64 processors fulfill system topology enumeration requirements:

1. Each logical processor in an Intel 64 or IA-32 platform supporting coherent memory is assigned a unique ID (APIC ID) within the coherent domain. A multi-node cluster installation may employ vendor-specific BIOS that preserves the APIC IDs assigned (during processor reset) within each coherent domain, extend with node IDs to form a superset of unique IDs within the clustered system. This document will only cover the CPUID interfaces providing unique IDs within a coherent domain.
2. The values of unique IDs assigned within a coherent Intel 64 or IA-32 platform conform to an algorithm based on bit field decomposition of the APIC ID into three sub-fields. The three sets of sub-fields correspond to three hierarchical domains defined as "logical processor," "processor core" (or "core"), and "physical package" (or "package"). This allows each hierarchical domain to be mapped to a sub-field (a sub-ID) within the APIC ID.

Conceptually, a topology enumeration algorithm is simply to extract the sub-ID corresponding to a given hierarchical domain from the APIC ID, based on deriving two parameters that defines the subset of bits within an APIC ID. The relevant parameters are: (a) the width of a mask that can be used to mask off unneeded bits in the APIC ID, (b) an offset relative to bit 0 of the APIC ID.

The "logical processor" domain corresponds to the innermost constituent of the processor topology, therefore it is located in the least significant portion of the APIC ID. If the corresponding width for "logical processor" is 0, it implies there is only one logical processor within the next outer domain of the hierarchy. If the corresponding width is 1 bit wide, there could be two logical processors within the next outer domain of the hierarchy.

If the corresponding width for "core" is 0, it implies there is only one processor core within a physical processor. If the corresponding width for "core" is 1 bit wide, there could be two processor cores within a physical processor.

The values of APIC ID that are assigned across all logical processors in the system need not be contiguous. But the subsets of bit fields corresponding to three hierarchical domains are contiguous at bit boundary. Due to this requirement, the bit offset of the mask to extract a given Sub-ID can be derived from the "mask width" of the inner hierarchical domains.

1.1.1 Glossary

Physical Processor: The physical package of a microprocessor capable of executing one or more threads of software at the same time. Each physical package plugs into a physical socket. Each physical package may contain one or more processor cores, also referred to as a **physical package**.

Processor Core: The circuitry that provides dedicated functionalities to decode, execute instructions, and transfer data between certain subsystems in a physical package. A processor core may contain one or more logical processors.

Logical Processor: The basic modularity of processor hardware resource that allows software executive (OS) to dispatch task or execute a thread context. Each logical processor can execute only one thread context at a time.

Intel Hyper-Threading Technology: A feature within the IA-32 family of processors, where each processor core provides the functionality of more than one logical processor.

SMT: Abbreviated name for Simultaneous Multi-Threading. An efficient means in silicon to provide the functionalities of multiple logical processors within the same processor core by sharing execution resources and cache hierarchy between logical processors.

Multi-Core Processor: A physical processor that contains more than one processor cores.

Multi-Processor Platform: A computer system made of two or more physical sockets.

Hardware Multi-Threading: Refers to any combination of hardware support to allow a system to run multi-threaded software. The forms of hardware support for multi-threading are: SMT, multi-core, and multi-processor.

Processor Topology: Hierarchical relationships processor entities (logical processors, processor cores) within a physical package relative to the sharing hierarchy of hardware resources within the physical processor.

Cache Hierarchy: Physical arrangement of cache levels that buffers data transport between a processor entity and the physical memory subsystem.

Cache Topology: Hierarchical relationships of a cache level relative to the logical processors in a physical processor.

Die: A software-visible chip inside a package. When a processor package contains a single die, die-scope and package-scope are synonymous. When a package contains multiple die, they are distinct.

DieGrp: A group of die that share certain resources.

Tile: A set of cores that share certain resources. The TILE_ID sub-field distinguishes different tiles. If there are no software visible tiles, the width of this bit field is 0.

Module: A set of cores that share certain resources. The MODULE_ID sub-field distinguishes different modules. If there are no software visible modules, the width of this bit field is 0.

Domain: A layer of the processor's topological hierarchy and represents a grouping of a single topology type. For example, a three-domain topology would consist of a domain of logical processors, processor cores, and packages. (Previous documentation referred to this as a "Level," e.g., a "three-level topology.")

1.2 UNIQUE APIC ID IN A MULTI-PROCESSOR SYSTEM

Although legacy IA-32 multi-processor systems assign unique APIC IDs for each logical processor in the system, the programming interfaces have evolved several times in the past. For Pentium Pro and Pentium III Xeon processors, APIC IDs are accessible only from local APIC registers (local APIC registers use Memory-Mapped I/O interfaces and are managed by the operating system). In the first generation of Pentium 4 and Intel® Xeon® processors (2000, 2001), the CPUID instruction provided information on the initial APIC ID that is assigned during processor reset. The CPUID instruction in the first generation of Intel Xeon MP processors and Pentium 4 processors supporting Intel Hyper-Threading Technology provided additional information that allows software to decompose initial APIC IDs into a two-level topology enumeration. With the introduction of dual-core Intel 64 processors in 2005, system topology enumeration using CPUID evolved into a three-level algorithm on the 8-bit wide initial APIC IDs. Intel 64 platforms are capable of supporting a large number of logical processors that exceed the capacity of the 8-bit initial APIC ID field. The x2APIC extension in Intel 64 architecture defines a 32-bit x2APIC ID, and the CPUID instruction in current Intel 64 processors allow software to enumerate system topology using x2APIC IDs.

The extended topology enumeration leaf of CPUID (leaf 0BH) was introduced in 2009 along with the x2APIC IDs. This leaf has been superseded by the v2 extended topology enumeration leaf (CPUID leaf 1FH), which is the preferred interface for system topology enumeration for current Intel 64 processors.

The CPUID instruction in current Intel 64 processors may support leaf 0BH or leaf 1FH independent of x2APIC hardware. For many current Intel 64 platforms, system topology enumeration may be performed using CPUID leaf 1FH, CPUID leaf 0BH, or the legacy initial APIC ID (via CPUID leaf 01H and CPUID leaf 04H). However, modern platforms should not use the legacy method for reasons described in Section 1.7. Figure 1-1 shows an example of how software should choose which CPUID leaf information to use for system topology enumeration.

1.2.1 x2APIC ID/APIC ID Topology Bit Layouts

The x2APIC ID/APIC ID can be parsed via bitmasks to extract the LOGICAL_PROCESSOR_ID, CORE_ID, PACKAGE_ID, etc. These can then be used to determine the topology and the location of any particular logical processor. Table 1-1 illustrates a basic example of a topology bit layout showing where each ID would be found.

Table 1-1. Example A: Topology Bit Layout

Domain	Bit Layout	Description
LOGICAL_PROCESSOR_ID	[0]	Two logical processors per core.
CORE_ID	[3:1]	Eight cores per package.

The PACKAGE_ID Mask can be used across all APIC ID/x2APIC IDs in the system to determine which logical processors belong to each package. This will then enumerate how many packages there are in the system. Table 1-2 shows the specific number identifier that each package can be labeled with after using the mask and right shifting the value by four bits.

The cores within the package are also then identifiable using the CORE_ID (Relative to Package) Mask. For each logical processor in a package, the number identifier of each core and the set of logical processors belonging to that core can then be determined. Table 1-2 shows the numeric identifier created after using the mask and right shifting the value by one bit.

Another way to identify cores across the system without determining the package is by using the CORE_ID (Relative to System) Mask. This can be applied across the system and identifies each core and the set of logical processors within that core.

To individually identify a logical processor within the core, identified using either method described, the LOGICAL_PROCESSOR_ID (Relative to Core) Mask can be used. This simply provides the number identifier of each logical processor within that core. These number identifiers do not need to be contiguous, but they do need to be unique relative to the upper layer in which they are being identified within.

Table 1-2. Example A Masks and Shifts

APIC ID	PACKAGE_ID Mask: 0xFFFFFFF0, Right Shift by 4 Bits	CORE_ID (Relative to Package) Mask: 0xE, Right Shift by 1 Bit	CORE_ID (Relative to System) Mask: 0xFFFFFFFE	LOGICAL_PROCESSOR_ID (Relative to Core) Mask: 0x1
0 (0x0)	0	0	0 (0x0)	0
1 (0x1)	0	0	0 (0x0)	1
2 (0x2)	0	1	2 (0x2)	0
3 (0x3)	0	1	2 (0x2)	1
4 (0x4)	0	2	4 (0x4)	0
5 (0x5)	0	2	4 (0x4)	1
6 (0x6)	0	3	6 (0x6)	0
7 (0x7)	0	3	6 (0x6)	1
8 (0x8)	0	4	8 (0x8)	0
9 (0x9)	0	4	8 (0x8)	1
10 (0xA)	0	5	10 (0xA)	0
11 (0xB)	0	5	10 (0xA)	1
12 (0xC)	0	6	12 (0xC)	0
13 (0xD)	0	6	12 (0xC)	1
14 (0xE)	0	7	14 (0xE)	0
15 (0xF)	0	7	14 (0xE)	1
16 (0x10)	1	0	16 (0x10)	0
17 (0x11)	1	0	16 (0x10)	1
18 (0x12)	1	1	18 (0x12)	0

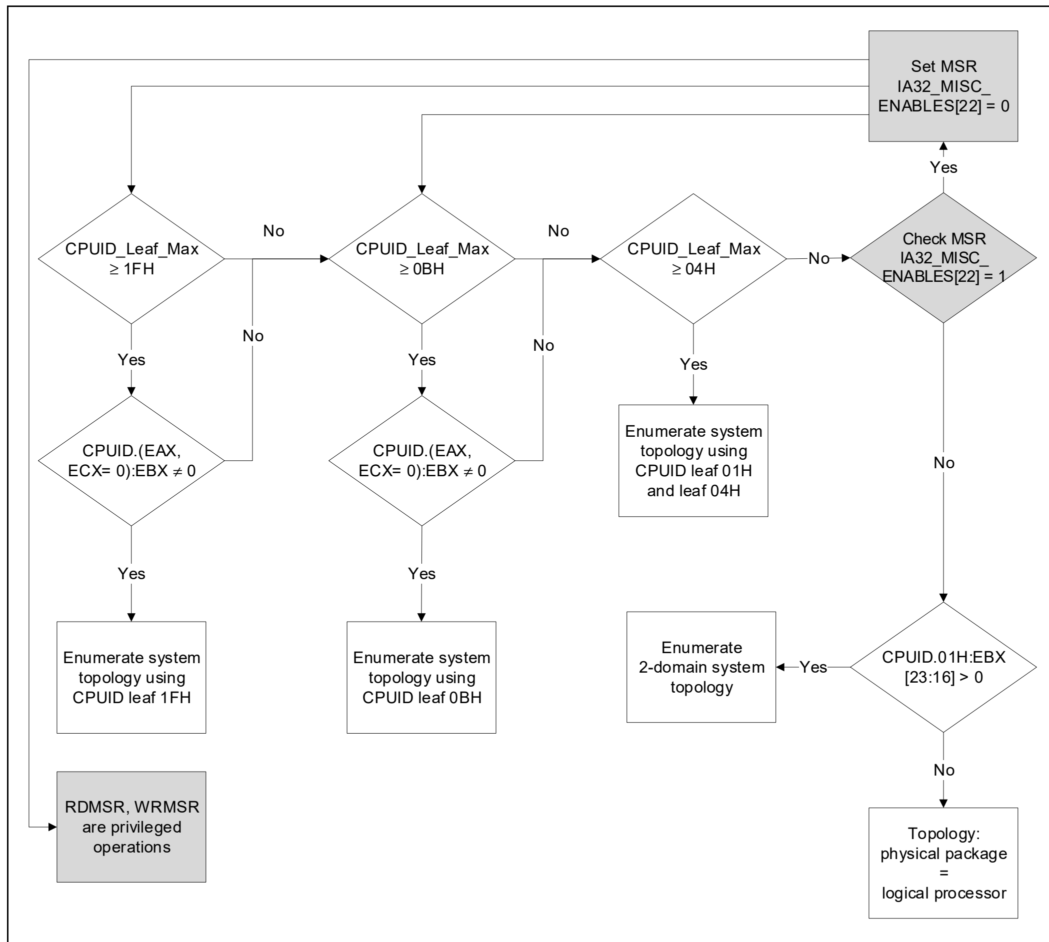
Table 1-2. Example A Masks and Shifts (Continued)

APIC ID	PACKAGE_ID Mask: 0xFFFFFFF0, Right Shift by 4 Bits	CORE_ID (Relative to Package) Mask: 0xE, Right Shift by 1 Bit	CORE_ID (Relative to System) Mask: 0xFFFFFFFE	LOGICAL_PROCESSOR_ID (Relative to Core) Mask: 0x1
19 (0x13)	1	1	18 (0x12)	1
20 (0x14)	1	2	20 (0x14)	0
21 (0x15)	1	2	20 (0x14)	1
22 (0x16)	1	3	22 (0x16)	0
23 (0x17)	1	3	22 (0x16)	1
24 (0x18)	1	4	24 (0x18)	0
25 (0x19)	1	4	24 (0x18)	1
26 (0x1A)	1	5	26 (0x1A)	0
27 (0x1B)	1	5	26 (0x1A)	1
28 (0x1C)	1	6	28 (0x1C)	0
29 (0x1D)	1	6	28 (0x1C)	1
30 (0x1E)	1	7	30 (0x1E)	0
31 (0x1F)	1	7	30 (0x1E)	1

1.2.2 Algorithm Enumerating Topology Discovery Method

The algorithm to enumerate the topology has evolved over time, as mentioned in the introduction. Figure 1-1 illustrates the method of discovering the best method available on the current platform.

Figure 1-1. Choosing CPUID Leaf Information for System Topology Enumeration



The maximum value of supported CPUID leaf can be determined by setting EAX = 0, executing CPUID, and examining the returned value in EAX, i.e., CPUID.0:EAX. If CPUID.0:EAX ≥ 1FH, software can determine whether CPUID leaf 1FH exists by setting EAX=1FH, ECX=0, and executing CPUID to examine the non-zero value returned in EBX, i.e., CPUID.(EAX=1FH, ECX=0):EBX ≠ 0.

If leaf 1FH is not found, software should next check for CPUID leaf 0BH in the same manner. If CPUID.0:EAX ≥ 0BH, software can determine whether CPUID leaf 0BH exists by setting EAX=0BH, ECX=0, and executing CPUID to examine the non-zero value returned in EBX, i.e., CPUID. (EAX=0BH, ECX=0):EBX ≠ 0.

Fully functional hardware multi-threading requires full-reporting of CPUID leaves.

If software observes that CPUID.0:EAX < 4 on a newer Intel 64 or IA-32 processor (newer than 2004), it should examine the MSR IA32_MISC_ENABLES[bit 22].

If IA32_MISC_ENABLES[bit 22] was set to ‘1’ (by BIOS or other means), the user can restore the CPUID leaf function full reporting by having IA32_MISC_ENABLES[bit 22] set to ‘0’ (modify the BIOS CMOS setting or use the WRMSR instruction).

Some older IA-32 processors support only two topology domains (logical processor and physical package). The three-level system topology enumeration algorithm (using CPUID leaf 01H and leaf 04H) is fully compatible with these older processors. For processors that report CPUID.01H:EBX[23:16] as reserved (i.e., 0), the processor supports only one topology domain.

1.2.3 CPUID Leaf 1FH Supersedes CPUID Leaf 0BH

CPUID leaf 0BH was introduced in 2009. At that time, the system topology consisted of three domains; logical processor, core, and package. The software algorithm that was documented at the time was inadequate for legacy software to properly handle new domains that would be unknown to it.

The algorithm was updated, and to maintain legacy software compatibility, CPUID leaf 1FH was introduced to return all topology domain enumerations, three domains and beyond. CPUID leaf 0BH will only ever enumerate the three domains of logical processor, core, and package to maintain legacy software. All new software, especially written to use CPUID leaf 1FH, needs to implement topology detection based on the algorithm specified in this document. The new algorithm is backwards compatible and works with CPUID leaf 0BH as well.

1.2.4 Application Software and Topology Enumeration

System topology enumeration at the application level using CPUID involves executing the CPUID instruction on each logical processor in the system. This implies context switching using services provided by an operating system. On-demand context switching by user code generally relies on a thread affinity management API provided by the operating system. The capability and limitation of a thread affinity API by different operating systems vary.

It is recommended for application software to use operating system APIs that provide this enumeration when available rather than to implement this algorithm manually using the CPUID instruction.

1.3 SYSTEM TOPOLOGY ENUMERATION USING CPUID LEAF 1FH OR CPUID LEAF 0BH

The same algorithm can be used for both CPUID leaf 0BH and CPUID leaf 1FH; see Section 1.2.3 for an explanation of the difference between these two leaves. This algorithm has been updated since the original documentation of CPUID leaf 0BH, and any software using the old algorithm must change to adhere to the updates outlined in this section.

The algorithm of system topology enumeration can be summarized as three phases of operation:

- Derive "mask width" constants that will be used to extract each Sub-ID.
- Gather the unique APIC IDs of each logical processor in the system, and extract/decompose each APIC ID into three sets of Sub-IDs.
- Analyze the relationship of hierarchical Sub-IDs to establish mapping tables between the operating system's thread management services according to three hierarchical domains of processor topology.

The generated topology map can be used by software in a number of application-specific ways. Some of the more common usages include:

1. Determine the number of physical processors to implement a per-package licensing policy. As the example topology in Table 1-1 shows, each unique ID extracted for the PACAKAGE_ID column in Table 1-2 represents a physical package.
2. A thread-binding strategy may choose to favor binding each new task to a separate core in the system. This may require the software to know the relationships between the affinity mask of each logical processor relative to each distinct processor core.
3. An MP-scaling optimization strategy may wish to partition its data working set according to the size of the large last-level cache and allow multiple threads to process the data tile residing in each last level cache. This will require software to manage the affinity masks and thread binding relative to each Cache_ID and x2APIC ID in the system.

Table 1-3 provides a reference for CPUID leaf 1FH.

Table 1-3. Reference for CPUID Leaf 1FH

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 using leaf 1FH when available rather than leaf 0BH and ensuring that any leaf 0BH algorithms are updated to support leaf 1FH.* The sub-leaves of CPUID leaf 1FH describe an ordered hierarchy of logical processors starting from the smallest-scoped domain of a Logical Processor (sub-leaf index 0) to the Core domain (sub-leaf index 1) to the largest-scoped domain (the last valid sub-leaf index) that is implicitly subordinate to the unenumerated highest-scoped domain of the processor package (socket). The details of each valid domain is enumerated by a corresponding sub-leaf. Details for a domain include its type and how all instances of that domain determine the number of logical processors and x2 APIC ID partitioning at the next higher-scoped domain. The ordering of domains within the hierarchy is fixed architecturally as shown below. For a given processor, not all domains may be relevant or enumerated; however, the logical processor and core domains are always enumerated. As an example, a processor may report an ordered hierarchy consisting only of "Logical Processor," "Core," and "Die." For two valid sub-leaves N and N+1, sub-leaf N+1 represents the next immediate higher-scoped domain with respect to the domain of sub-leaf N for the given processor. If sub-leaf index "N" returns an invalid domain type in ECX[15:08] (00H), then all sub-leaves with an index greater than "N" shall also return an invalid domain type. A sub-leaf returning an invalid domain always returns 0 in EAX and EBX.
	EAX	Bits 04-00: The number of bits that the x2APIC ID must be shifted to the right to address instances of the next higher-scoped domain. When logical processor is not supported by the processor, the value of this field at the Logical Processor domain sub-leaf may be returned as either 0 (no allocated bits in the x2APIC ID) or 1 (one allocated bit in the x2APIC ID); software should plan accordingly. Bits 31-05: Reserved.
	EBX	Bits 15-00: The number of logical processors across all instances of this domain within the next higher-scoped domain relative to this current logical processor. (For example, in a processor socket/package symmetric topology comprising "M" dies of "N" cores each, where each core has "L" logical processors, the "die" domain sub-leaf value of this field would be M*N*L. In an asymmetric topology this would be the summation of the value across the lower domain level instances to create each upper domain level instance.) This number reflects configuration as shipped by Intel. Note, software must not use this field to enumerate processor topology*. Bits 31-16: Reserved.

Table 1-3. Reference for CPUID Leaf 1FH (Continued)

Initial EAX Value	Information Provided about the Processor	
	ECX	Bits 07-00: The input ECX sub-leaf index. Bits 15-08: Domain Type: This field provides an identification value which indicates the domain as shown below. Although domains are ordered, as also shown below, their assigned identification values are not and software should not depend on it. (For example, if a new domain between core and module is specified, it will have an identification value higher than 5.) Hierarchy / Domain / Domain Type Identification Value Lowest / Logical Processor / 1 ... / Core / 2 ... / Module / 3 ... / Tile / 4 ... / Die / 5 ... / DieGrp / 6 Highest / Package/Socket / (implied) (Note that enumeration values of 0 and 7-255 are reserved.) Bits 31-16: Reserved.
	EDX	Bits 31-00: x2APIC ID of the current logical processor. It is always valid and does not vary with the sub-leaf index in ECX. **NOTES:** * Software must not use the value of EBX[15:0] to enumerate processor topology of the system. The value is only intended for display and 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 following sections outline how to obtain the shift values for the topology expected by software.

1.3.1 Implementing a Three-Domain Topology

The three-domain topology is implemented by creating the LOGICAL_PROCESSOR_ID Mask, CORE_ID Mask, and PACKAGE_ID Mask. Modern platforms can include more topology domains than these three, which would be unknown to software written prior to their introduction. These unknown domains cannot be ignored as the original algorithm suggested for CPUID leaf 0BH in 2009, as it will lead to incorrect topology generation. These unknown domains cannot be used by software either as software is written to understand a certain hierarchy. Instead, they must be absorbed into the known lowest domain directly below.

Table 1-4 illustrates an example of a topology containing more than three domains. Since the software only understands three domains, it must convert this into a usable three-domain topology.

Table 1-4. Example B: A Topology Bit Layout With More Than Three Domains

Domain	Bit Layout	Description
LOGICAL_PROCESSOR_ID	[0]	Two logical processors per core.
CORE_ID	[2:1]	Four cores per unknown domain.
Unknown Domain	[3]	Two unknown domains per package.

To convert this example into a three-domain topology, the last enumerated known domain simply merges with the unknown domains above it. This creates a known topology to software while still maintaining the true relationship between the lower and upper domains.

The PACKAGE_ID Mask is always above the highest enumeration. If CORE_ID is not enumerated or enumerates a shift of 0, then the package contains a single core. The package additionally may only have one logical processor if LOGICAL_PROCESSOR_ID also enumerates a shift of zero.

Table 1-5. Example B Converted to a Three-Domain Topology

Domain	Bit Layout	Description
LOGICAL_PROCESSOR_ID	[0]	Two logical processors per core.
CORE_ID	[3:1]	Eight cores per package.

There are eight cores in this package, in which each unknown domain maintains four cores. The problem with ignoring an unknown domain is that it will create gaps and incorrect topologies. In this case, the unknown domain would be enumerated then as the package domain. This would result in software seeing two packages where it should only see one.

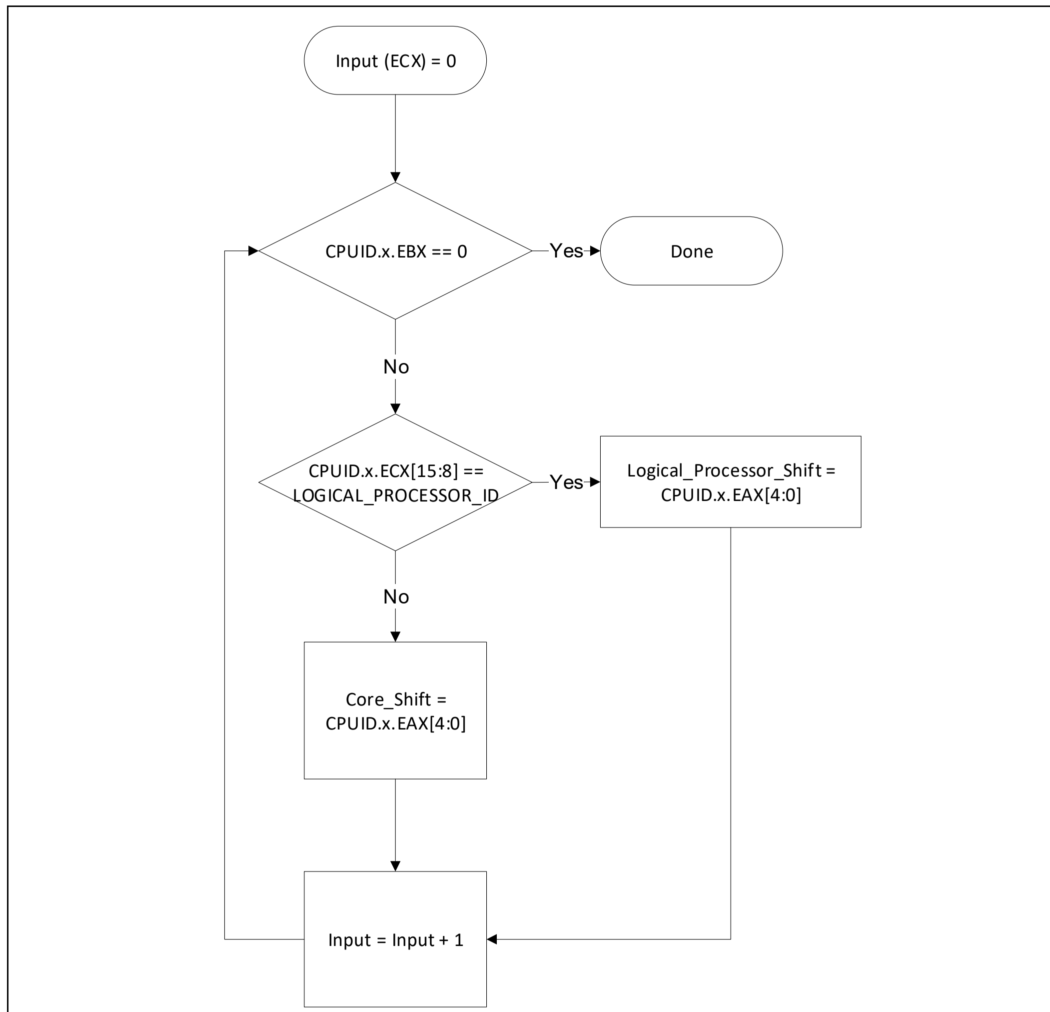
The solution requires that CORE_ID change to use the unknown domain shift enumeration, i.e., this becomes CORE_ID[3:1]. This will now correctly describe a three-domain topology when an unknown topology appears.

The simplest algorithm is the following:

1. Query the right-shift value for the logical processor domain of the topology using CPUID leaf 1FH (or CPUID leaf 0BH) where ECX[15:8] is LOGICAL_PROCESSOR_ID (which would occur when the ECX input value is 0). The number of bits to shift-right on x2APIC ID (EAX[4:0]) can distinguish different higher-domain entities above logical processor (e.g., processor cores) in the same physical package. This is also the width of the bit mask to extract the LOGICAL_PROCESSOR_ID.
2. In the enumeration loop, it can simply update the Core_Shift value (EAX[4:0]) at each enumeration domain where ECX[15:8] ≠ LOGICAL_PROCESSOR_ID and EBX ≠ 0. Do not check for CORE_ID explicitly; the CORE_ID Mask should end up as the highest domain returned from CPUID leaf 1FH or CPUID leaf 0BH.

Figure 1-2 provides a flow chart demonstrating this algorithm, where x can be CPUID leaf 1FH or CPUID leaf 0BH.

Figure 1-2. Flow Chart Demonstrating How to Implement a Three-Domain Topology



1.3.2 Implementing Topology Domains Beyond Three

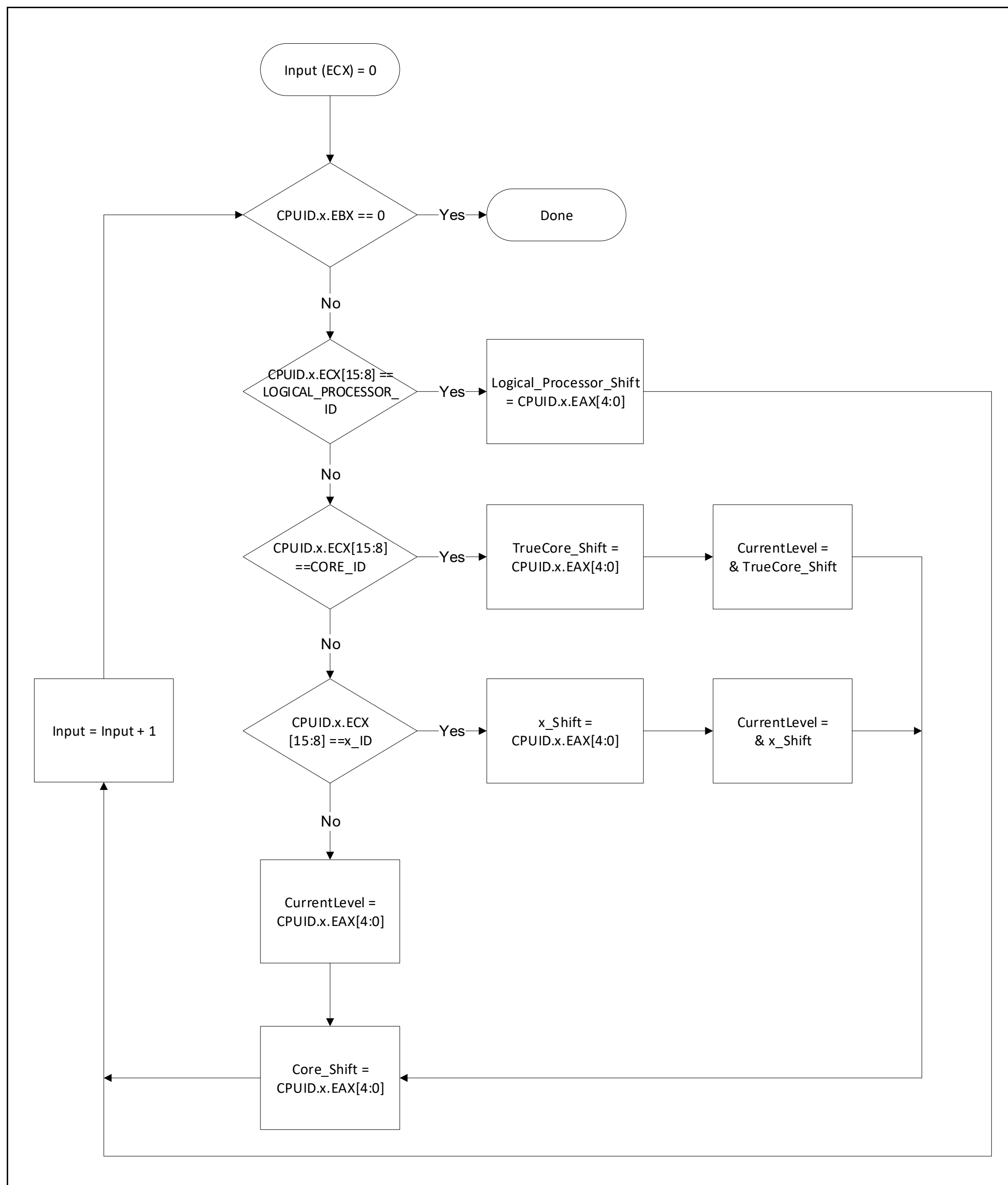
System software that wants to understand more than three domains or has a need to see multiple views (three domains in some places and other domain types elsewhere) must enumerate them as such. This includes handling unknown domains beyond what is understood. The base three domains can be implemented as shown in Section 1.3.1 in the simplest form, but to juggle multiple domains with unknown domains above and below, there is more work to do.

1. Query the right-shift value for the logical processor domain of the topology using CPUID leaf 1FH (or CPUID leaf 0BH) where ECX[15:8] is LOGICAL_PROCESSOR_ID (which would occur when ECX input value is 0). The number of bits to shift-right on x2APIC ID (EAX[4:0]) can distinguish different higher-level entities above logical processor (e.g., processor cores) in the same physical package. This is also the width of the bit mask to extract the LOGICAL_PROCESSOR_ID.
2. Enumerate the next domain, which would be (ECX[15:8] == CORE_ID), and set its shift value (EAX[4:0]). This would then be known as the current domain being enumerated such that if any unknown domains are encountered before the next known domain, the CORE_ID shift can be updated with that unknown domain.

3. Enumerate the next domain, and if ECX[15:8] is a known domain, then update the shift value (EAX[4:0]) and set this as the current known domain. If this domain is unknown, then update the current known domain with its shift value (EAX[4:0]). Repeat Step 3 until CPUID.[0BH or 1FH].EBX == 0.

Figure 1-3 provides a flow chart where x can be CPUID leaf 1FH or CPUID leaf 0BH. "x_ID" is some other domain topology that this software knows about. There could be "n" number of these. "TrueCore_Shift" and "Core_Shift" are separated such that "Core_Shift" can be used for a three-domain topology and generating the PACKAGE_ID Mask, where "TrueCore_Shift" is relative to the other "x_ID" topologies that this software is implementing.

Figure 1-3. Flow Chart Demonstrating How to Implement a Topology Greater Than Three Domains



There is an additional possible enumeration used to track the previous shift and when you find a new "x_ID" you create the lower shift using that previous shift value. This allows the code to isolate that particular ID relative to the lower ID regardless of what it is rather than hard coding the possibilities.

Tables 1-6 through 1-13 provide examples of topology conversions for multi-level topologies with unknown domains to three-domain topologies with known domains.

Table 1-6. Example C: A Five-Domain Topology Bit Layout

Domain	Bit Layout	Description
LOGICAL_PROCESSOR_ID	[0]	Two logical processors per core.
CORE_ID	[2:1]	Four cores per unknown domain.
Unknown Domain	[3]	Two unknown domains per die.
DIE_ID	[5:4]	Four dies per package.

Table 1-6 provides an example of a five-domain topology where there is an unknown domain between CORE_ID and DIE_ID. CORE_ID will need to merge with the unknown domain; this is represented by Table 1-7.

Table 1-7. Example C Converted to a Four-Domain Topology

Domain	Bit Layout	Description
LOGICAL_PROCESSOR_ID	[0]	Two logical processors per core.
CORE_ID	[3:1]	Eight cores per die.
DIE_ID	[5:4]	Four dies per package.

Table 1-8 represents the final step in moving to a three-domain topology; CORE_ID simply merges with DIE_ID.

Table 1-8. Example C Converted to a Three-Domain Topology

Domain	Bit Layout	Description
LOGICAL_PROCESSOR_ID	[0]	Two logical processors per core.
CORE_ID	[5:1]	32 cores per package.

Table 1-9 provides an example of a multi-level topology that contains a mix of domains, as well as multiple unknown domains in a row. This is an enumeration for 2,048 processors in a package.

Table 1-9. Example D: A Multi-Level Topology With Multiple Unknown Domains

Domain	Bit Layout	Description
LOGICAL_PROCESSOR_ID	[0]	Two logical processors per core.
CORE_ID	[2:1]	Four cores per unknown domain.
Unknown Domain	[3]	Two unknown domains per module.
MODULE_ID	[5:4]	Four modules per die.
DIE_ID	[7:6]	Four dies per unknown domain 2.
Unknown Domain 2	[8]	Two unknown domain 2 per unknown domain 3.
Unknown Domain 3	[10:9]	Four unknown domain 3 per package.

Tables 1-10 through 1-13 provide various conversion examples depending on the relative domains that software is seeking within a package.

Table 1-10. Example D Converted to a Five-Domain Topology

Domain	Bit Layout	Description
LOGICAL_PROCESSOR_ID	[0]	Two logical processors per core.
CORE_ID	[3:1]	Eight cores per module.
MODULE_ID	[5:4]	Four modules per die.
DIE_ID	[10:6]	32 dies per package.

Table 1-11. Example D Converted to a Four-Domain Topology

Domain	Bit Layout	Description
LOGICAL_PROCESSOR_ID	[0]	Two logical processors per core.
CORE_ID	[5:1]	32 cores per die.
DIE_ID	[10:6]	32 dies per package.

Table 1-12. Example D Converted to an Alternative Four-Domain Topology

Domain	Bit Layout	Description
LOGICAL_PROCESSOR_ID	[0]	Two logical processors per core.
CORE_ID	[3:1]	Eight cores per module.
MODULE_ID	[10:4]	128 modules per package.

Table 1-13. Example D Converted to a Three-Domain Topology

Domain	Bit Layout	Description
LOGICAL_PROCESSOR_ID	[0]	Two logical processors per core.
CORE_ID	[10:1]	1,024 cores per package.

1.3.3 Generating the Topology Masks from the Acquired Shift Values

Once the shift values are known, the topology is generated by creating bitmasks. On a three-domain topology, there is a Logical_Processor_Shift and a Core_Shift. Consider shift values of Logical_Processor_Shift = 1 and Core_Shift = 4. This correlates to a platform with the topology bit layout as shown in Table 1-14.

Table 1-14. A Three-Domain Topology With Logical_Processor_Shift = 1 and Core_Shift = 4

Domain	Bit Layout	Description
LOGICAL_PROCESSOR_ID	[0]	Two logical processors per core.
CORE_ID	[3:1]	Eight cores per package.

To generate the LOGICAL_PROCESSOR_ID Mask, perform the following ((1<<Logical_Processor_Shift)-1), which then provides a value of "0x1." This mask can determine which sibling is in the core, once a CORE_ID match between two logical processors is identified.

The CORE_ID (Relative to System) Mask can now be generated. This mask can determine which logical processors are in the same core. Globally, this is done by inverting the LOGICAL_PROCESSOR_ID Mask ~((1<<Logical_Processor_Shift)-1), which gives a value of "0xFFFFFFFE." This mask can then be used to mask and compare multiple x2APIC ID/APIC IDs to determine if they are in the same core.

The Package IDs and the Core IDs relative to the package can be obtained using the Core_Shift. The Package_ID Mask would be ~((1<<Core_Shift)-1), which gives a value of "0xFFFFFFF0." The shift would be a Core_Shift of 4 to retrieve the actual ID.

The CORE_ID Mask relative to the package is ((1<<Core_Shift)-1) & (~LOGICAL_PROCESSOR_ID Mask), which gives the mask value of "0x7." The shift value would be the Logical_Processor_Shift. The shift value is the lower edge of the bit mask, which is defined by the level directly below it.

These methods apply to multiple levels above core and below package. For example, the DIE Mask to create a DIE_ID relative to the package would be (((1<<Die_Shift)-1) & (~((1<<Core_Shift)-1))), which removes the complete lower bits. The shift would be Core_Shift since that is the direct lower-level shift value.

Once the masks are created, they can then be used to iterate through the x2APIC ID/APIC IDs of the platform to determine the topology relationship between each logical processor as outlined in Section 1.2.1.

1.4 CACHE TOPOLOGY ENUMERATION USING CPUID LEAF 04H

The physical package of an Intel 64 processor has a hierarchy of cache. A given level of the cache hierarchy may be shared by one or more logical processors. Some software may wish to optimize performance by taking advantage of the shared cache of a particular level of the cache hierarchy.

Performance tuning using cache topology can be accomplished by combining the system topology information with the addition of cache topology information.

The CACHE_IDs for each cache level can be extracted from the x2APIC ID for processors that report 32-bit x2APIC ID, or from the initial APIC ID for processors that do not report x2APIC ID. Each logical processor must enumerate and create a cache mask for each level and types of its cache. The list of CACHE_Masks[n] of all types and levels when a bitwise AND is performed against its own x2APIC ID/APIC ID will provide the CACHE_ID[n] value for that level and type that can be used to match other processors sharing the same cache by performing a bitwise AND against their x2APIC ID/APIC ID. The sub-leaf index number used to enumerate CPUID.04H has no relation to the cache level or cache type. The cache type must be read from CPUID.04H.n.EAX[4:0] and the cache level must be read from CPUID.04H.EAX[7:5]. The same information may be enumerated on different sub-leaf numbers between processors sharing the cache.

Table 1-15 provides a reference for CPUID leaf 04H.

Table 1-15. Reference for CPUID Leaf 04H

Initial EAX Value	Information Provided about the Processor
	Deterministic Cache Parameters Leaf
04H	**NOTES:** Leaf 04H output depends on the initial value in ECX.* See also: "INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level."

Table 1-15. Reference for CPUID Leaf 04H (Continued)

Initial EAX Value		Information Provided about the Processor
	EAX	Bits 04-00: Cache Type Field. 0 = Null - No more caches. 1 = Data Cache. 2 = Instruction Cache. 3 = Unified Cache. 4-31 = Reserved.
		Bits 07-05: Cache Level (starts at 1). Bit 08: Self Initializing cache level (does not need SW initialization). Bit 09: 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 00: 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 01: Cache Inclusiveness. 0 = Cache is not inclusive of lower cache levels. 1 = Cache is inclusive of lower cache levels. Bit 02: 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: * If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n+1 is invalid if sub-leaf n returns EAX[4:0] as 0. ** 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.

1.4.1 Cache ID Extraction Parameters

The enumeration of the cache levels is done by enumerating through CPUID.04H with ECX=0 and incrementally going through each sub-leaf until CPUID.04H.n.EAX[4:0] == 0. There is a lot of information contained in each sub-leaf for that identified cache. This section is only going to outline how to determine which logical processors are sharing that cache.

The algorithm for CPUID.04H must be executed on each logical processor as the caches may be asymmetric between them. This means that each logical processor will have its own set of CACHE_ID[n]s and CACHE_Mask[n]s.

The algorithm then is for each sub-leaf of “n” from 0 to k-1, where CPUID.04H.k.EAX[4:0] == 0:

1. Identify the cache level and cache type being described by this sub-leaf “n” by reading CPUID.04H.n.EAX[4:0] for the type and CPUID.04H.n.EAX[7:5] for the level.
2. Extract the number of logical processors sharing that cache:
   LogicalProcessorsSharingCache = CPUID.04H.n.EAX[25:14]+1.
3. Convert this to a power of 2, which can be done by (LogicalProcessorsSharingCache*2)-1, then taking the highest set bit and clearing all lower bits (LogicalProcessorsSharingCacheP2).
4. Create the cache mask using the following algorithm:
   CACHE_Mask[n] = ~(LogicalProcessorsSharingCacheP2-1) for this logical processor.
5. The CACHE_ID[n] for this logical processor is then determined by using the mask on the x2APIC ID/APIC ID of the current logical processor being enumerated:
   CACHE_ID[n] = CACHE_Mask[n] & x2APIC ID.
6. Find every logical processor “m” where (CACHE_Mask[n] & LogicalProcessor[m].x2APIC ID) == CACHE_ID[n].
7. Check if logical processor “m” enumerates this same cache. This is done by comparing the current processor's CPUID.04H.n registers (EAX, EBX, ECX, and EDX) for this cache against the CPUID.04H enumeration of logical processor “m”. Logical processor “m” does not need to enumerate this cache at the same sub-leaf of CPUID.04H. So, each sub-leaf of CPUID.04H of logical processor “m” must be compared against the current logical processor's CPUID.04H.n to determine if there is a match. The complete set of registers (EAX, EBX, ECX, and EDX) must match.

1.5 TLB TOPOLOGY ENUMERATION USING CPUID LEAF 18H

The logical processors within an Intel 64 processor package each have a hierarchy of TLBs. A given level of the TLB hierarchy may be shared by one or more logical processors. A use case, for example, would be for the memory manager to use this information for optimizing TLB shootdowns.

Table 1-16 provides a reference for CPUID leaf 18H.

Table 1-16. Reference for CPUID Leaf 18H

Initial EAX Value	Information Provided about the Processor	
	Deterministic Address Translation Parameters Main Leaf (EAX = 18H, ECX = 0)	
18H		**NOTES:** Each sub-leaf enumerates a different address translation 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 1-16. Reference for CPUID Leaf 18H (Continued)

Initial EAX Value	Information Provided about the Processor	
	Deterministic Address Translation Parameters Sub-leaf (EAX = 18H, ECX ≥ 1)	
18H		**NOTES:** Each sub-leaf enumerates a different address translation 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: 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.

1.5.1 TLB ID Extraction Parameters

The enumeration of the TLB levels is done by enumerating through CPUID.18H with ECX=0 and incrementally going through each sub-leaf until sub-leaf "n" == CPUID.18H.0.EAX. The sub-leaf number has no relation to the TLB level or type. The TLB Level is found at CPUID.18H.n.EDX[7:5] and the type is found at CPUID.18H.n.EDX[4:0]. The same information may be enumerated on different sub-leaf numbers between processors sharing the TLB. There is a lot of information contained in each sub-leaf for that identified TLB. This section only outlines how to determine which logical processors are sharing that TLB.

The algorithm for CPUID.18H must be executed on each logical processor as the caches may be asymmetric between them. This means that each logical processor will have its own set of TLB_ID[n]s and TLB_Mask[n]s.

The algorithm then is for each sub-leaf of "n" from 0 to CPUID.18H.0.EAX:

1. Identify the TLB level and TLB type being described by this sub-leaf "n" by reading CPUID.18H.n.EDX[4:0] for the type, and CPUID.18H.n.EDX[7:5] for the level.
2. Extract the number of logical processors sharing that TLB:
   LogicalProcessorsSharingTlb = CPUID.18H.n.EDX[25:14]+1.
3. Convert this to a power of 2, which can be done by (LogicalProcessorsSharingTlb*2)-1, then taking the highest set bit and clearing all lower bits (LogicalProcessorsSharingTlbP2).
4. Create the TLB mask using the following algorithm:
   TLB_Mask[n] = ~(LogicalProcessorsSharingTlbP2-1) for this logical processor.
5. The TLB_ID[n] for this logical processor is then determined by using the mask on the x2APIC ID/APIC ID of the current logical processor being enumerated:
   TLB_ID[n] = TLB_Mask[n] & x2APIC ID.
6. Find every logical processor "m" where (TLB_Mask[n] & LogicalProcessor[m].x2APIC ID) == TLB_ID[n].
7. Check if logical processor "m" enumerates this same TLB. This is done by comparing the current processor's CPUID.18H.n registers (EBX, ECX, and EDX; do not compare EAX) for this TLB against the CPUID.18H enumeration of logical processor "m". Logical processor "m" does not need to enumerate this TLB at the same sub-leaf of CPUID.18H. So, each sub-leaf of CPUID.18H of logical processor "m" must be compared against the current logical processor's CPUID.18H.n to determine if there is a match. The compare needs to be done against EBX, ECX, and EDX. EAX on sub-leaf 0 contains the maximum sub-leaves, which does not need to match, and on subsequent processors it is reserved.

1.6 LEGACY ENUMERATION WITHOUT EXTENDED TOPOLOGY LEAVES

Older CPUs prior to the introduction of the x2APIC ID would not support either CPUID leaf 0BH or CPUID leaf 1FH, and then must use the older CPUID.01H and/or CPUID.04H methods for topology enumeration. These older methods use the 8-bit APIC ID and cannot be used with any platform that supports the x2APIC ID as the number of processors can exceed the 8-bit APIC ID.

For processors where the x2APIC ID exists, using this method with the APIC ID can result in an incorrect topology.

Figure 1-4 outlines the procedures of querying the initial APIC ID via CPUID leaf 01H and extracting sub-IDs corresponding to the logical processor, core, and physical package domains of the hierarchy using CPUID leaf 01H and CPUID leaf 04H. Extraction of a sub-ID from the initial APIC ID is based on querying CPUID leaf 01H and CPUID leaf 04H to derive the bit widths of three select masks (LOGICAL_PROCESSOR_ID Mask, CORE_ID Mask, and PACKAGE_ID Mask) that make up the 8-bit initial APIC ID field. The select masks allow software to extract the sub-IDs corresponding to "logical processor," "core," and "package" from the initial APIC ID of each logical processor.

Figure 1-4. Procedures to Extract Sub-IDs from the Initial APIC ID of Each Logical Processor

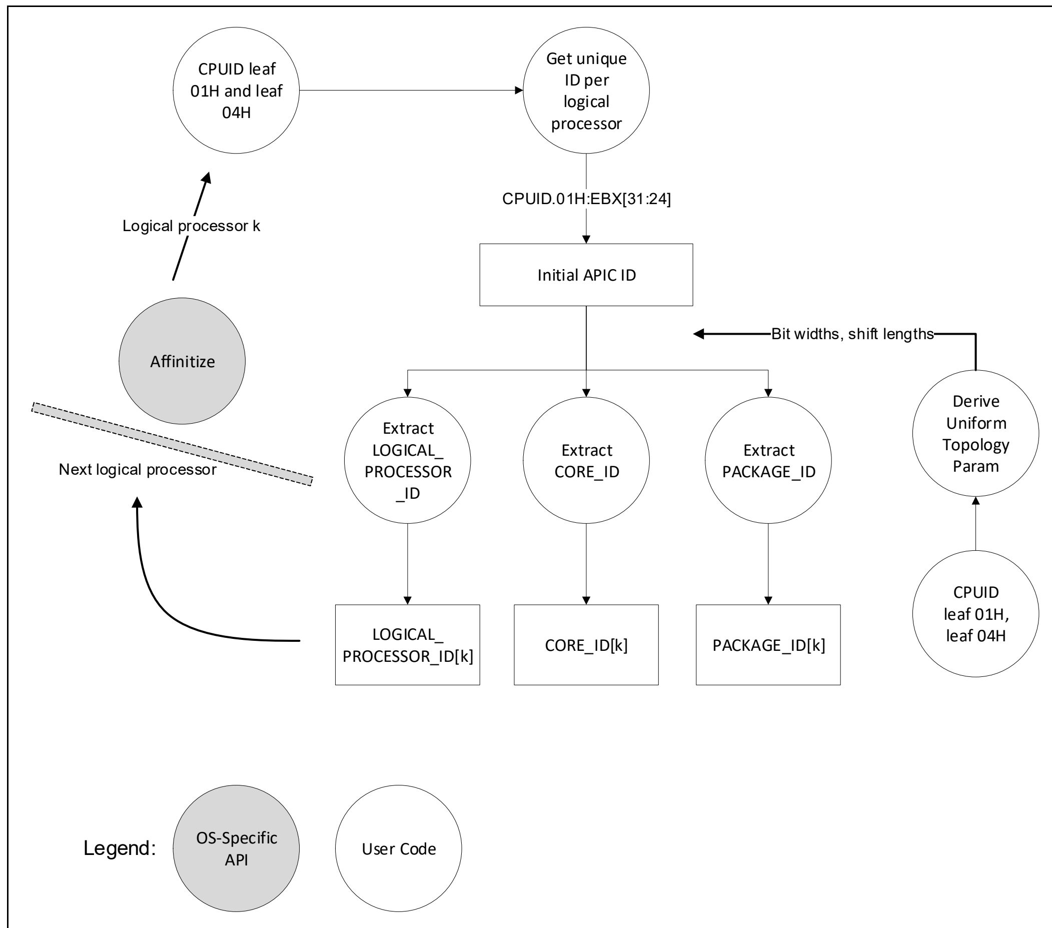


Table 1-17 provides a reference for CPUID leaf 01H. See Table 1-15 for a reference for CPUID 04H.

Table 1-17. Reference for CPUID Leaf 01H

Initial EAX Value	Information Provided about the Processor	
01H	EAX	Version Information: Type, Family, Model, and Stepping ID.
	EBX	Bits 07-00: Brand Index. Bits 15-08: CLFLUSH line size (Value ∗ 8 = cache line size in bytes; used also by CLFLUSHOPT). Bits 23-16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31-24: Initial APIC ID**.
	ECX	Feature Information.
	EDX	Feature Information.

Table 1-17. Reference for CPUID Leaf 01H (Continued)

Initial EAX Value	Information Provided about the Processor
	NOTES: * The nearest power-of-2 integer that is not smaller than EBX[23:16] is the number of unique initial APIC IDs reserved for addressing different logical processors in a physical package. This field is only valid if CPUID.1.EDX.HTT[bit 28]= 1. ** *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.*

1.6.1 Legacy Extraction Algorithm

The legacy extraction algorithm is simple in that the Maximum addressable IDs within a physical package at CPUID.01H.EBX[23:16] is rounded to the nearest power of 2. Rounding to the nearest power of two can be performed by the following steps:

1. Create the following value: "(CPUID.01H.EBX[23:16]*2)-1."
2. Find the highest bit set from the value output of that equation and then clear all lower bits to provide the "MaximumLogicalProcessorIDsPerPackage."

The second part of the algorithm is to extract the maximum addressable Core IDs in the package. This is done by reading (CPUID.04H.0.EAX[31:26]+1) and performing the same algorithm as above to round it to a power of 2. This value would then be the "MaximumCoreIDsPerPackage."

The PACKAGE_ID Mask can be created with ~(MaximumLogicalProcessorIDsPerPackage-1), with the bit location being the shift value.

If the MaximumLogicalProcessorIDsPerPackage value is not the same as the MaximumCoreIDsPerPackage value, then these values need to be divided to determine the CORE_ID Mask and the LOGICAL_PROCESSOR_ID Mask.

1. The number of "LogicalProcessorsPerCore" is created through this division:
   LogicalProcessorPerCore = MaximumLogicalProcessorIDsPerPackage / MaximumCoreIDsPerPackage
2. The LOGICAL_PROCESSOR_ID Mask can be created using the following equation:
   LOGICAL_PROCESSOR_ID Mask = (LogicalProcessorsPerCore - 1)
3. The CORE_ID Mask relative to the package can be created using the following equation:
   CORE_ID Mask = (MaximumCoreIDsPerPackage-1) & ~(LOGICAL_PROCESSOR_ID Mask)

1.7 ANALYZING TOPOLOGY ENUMERATION RESULT AND CUSTOMIZATION

How software uses topological information (in the form of hierarchical sub-IDs) depends on the needs and situations specific to each application. It may need adaptation due to differences of APIs provided by different operating systems. For the purpose of illustration, consider some examples of using sub-IDs to establish and manage affinity masks hierarchically.

The knowledge of sub-IDs of each topological hierarchy may be useful in several ways, for example:

1. Count the number of entities in a given hierarchical domain across the system.
2. Use OS threading management services (e.g., affinity masks) while adding topological insights (per-core, per-package, per-target-cache-level) to optimize application performance.

Affinity mask is a data structure that is defined within a specific operating system. Different operating systems may use the same concept but providing different means of application programming inter-

face. For example, Microsoft Windows provides affinity mask as a data type that can be directly manipulated via bit field by applications for affinity control. Linux implements a similar data structure internally but abstracts it so applications can manipulate affinity through an iterative interface that assigns zero-based numbers to each logical processor.

The affinity mask, or the equivalent numbering scheme provided by the operating system, does not carry attributes that can store hierarchical attributes of the system topology. The "affinity mask" terminology is used generically in this section, as the technique can be easily generalized to the numbered interface of affinity control.

In the reference code example, the sub-IDs are used to create an ordinal numbering scheme (zero-based) for each hierarchical domain. Different entities in the system topology (packages, cores) can be referenced by applications using a set of hierarchical ordinal numbers. Using the hierarchical ordinal number scheme and a look-up table to the corresponding affinity masks, software can easily control thread binding, optimize cache usage, etc.

1.7.1 Dynamic Software Visibility of Topology Enumeration

When application software examines/uses topology information, it must keep in mind the dynamic nature of software visibility. The hardware capability present at the platform level may be presented differently through BIOS setting, through OS boot option, and/or through OS-supported user interfaces.

Software must also not rely on symmetry as modern platforms may have an asymmetric topology. There can be some cores that support Intel Hyper-Threading Technology and some cores that do not. Software must be prepared to handle these cases and not assume symmetry after enumerating one core.