
# Detecting Multi-Core Processor Topology in an IA-32 Platform

by Khang Nguyen and Shihjong Kuo

## Introduction

Intel® Architecture IA-32 platforms initiated hardware support for multi-processing by first using two or more sockets per system board, where each socket can be populated with one physical package. Prior to the introduction of Hyper-Threading Technology (HT Technology), each physical package of an IA-32 processor is capable of executing one thread at a time.

In 2002, IA-32 platform introduced Hyper-Threading Technology, where each physical package provides functionality logically equivalent to two discrete, single-thread-execution processors, but sharing the same processor core in a physical package. In 2005, Intel introduced IA-32 platforms with multi-core technology with the Intel® Pentium® processor Extreme Edition, which provides two processor cores, each supporting HT Technology. Thus, hardware support for multi-processing has evolved from multiple discrete sockets, to HT Technology, to multiple cores with HT Technology.

This paper discusses a robust algorithm to help application software enumerate the processor and cache topology in any single or multi-processor platform, using Intel processors. Enumerating processor topology correctly is essential for implementing licensing policy requirements. Understanding processor and cache topology information will allow multithreading software to make more efficient use of hardware multithreading resources and deliver optimal performance.

Software must recognize hardware multi-processing support in all of these combinations. For licensing purposes, Intel recommends a policy based on discrete physical packages. For performance optimization purposes, software may need to manage physical resources depending on the details of the sharing topology implemented in these various forms of hardware multiprocessing.

In this paper we show how to detect the topological relationships between physical package, processor core, and logical processors sharing the same core in a multi-processing platform with IA-32 processors. The algorithm described in this paper applies across many hardware multi-processing configurations, including single-socket and multi-socket platforms, IA-32 processors supporting Hyper-Threading Technology, dual-core and multiple cores.

## Hyper-Threading Technology, Multi-Core and Initial APIC ID

In a multithreading environment, using IA-32 processors with hardware multithreading support, each logical processor in the platform must have a unique identifier. This is established during platform power-up, and is referred to as “initial APIC[1] ID.” Each initial APIC ID’s value in a multiprocessor (MP) platform is assigned in an orderly manner such that software can extract a topological relationship between an initial APIC ID and the physical package and processor core. Software can also extract the topological relationships of sibling logical processors sharing the same core or sharing a particular cache level of the cache hierarchy.

1 APIC stands for Advanced Programmable Interrupt Controller

In general, each physical package can provide one or more processor cores. The number of logical processors sharing the same core must be derived from

information provided by the CPUID instruction. With respect to the cache-sharing topology of the same microarchitecture, the number of logical processors sharing a given cache level may vary for each cache level in the cache hierarchy. Between different microarchitectures, the cache sharing topology may also vary.

With HT Technology enabled processors, each cache level is shared by the all logical processors sharing a processor core. Software must use the CPUID instruction to query for relevant data for each cache level at runtime. (Details of using CPUID instruction to query initial APIC IDs, logical processor/core/cache configurations can be found in Chapter 3 of IA-32 Intel® Architecture Software Developer's Manual Vol. 2A.)

**CPUID Instruction**

Aside from the initial APIC IDs reported by CPUID instruction, there are three other essential parameters reported by CPUID. These parameters are used to determine the multithreading resource topology of an IA-32 platform:

- Logical Processors per Package (CPUID.1.EBX[23:16]) — Indicates the maximum number of logical processors in a physical package. This represents the hardware capability of the processor as manufactured, and does not necessarily equate to the number of logical processors enabled by the platform bios or operating system.

- Cores per Package[2] (CPUID.4.EAX[31:26] + 1) — The maximum number of cores in a physical package is indicated by one plus the decimal value represented by CPUID.4.EAX[31:26].

- Logical Processors Sharing a Cache (CPUID.4.EAX[25:14] + 1) — The maximum number of logical processors in a physical package sharing the target level cache is indicated by one plus the decimal value represented by CPUID.4.EAX[25:14].

Intel supports only homogeneous MP systems. This means that in an MP system, any logical processor from any physical package must report the same values for the three parameters described above.

CPUID cannot be called directly from high-level languages such as C or C++. This must be done using assembly language. In this paper, we show sample code that executes the CPUID instruction from C/C++ source code using inline assembly.

**Three Level Topology and Initial APIC ID**

For a single-clustered MP system[3], the 8-bit value of an initial APIC ID decomposes into three bit fields. The order in which initial APIC IDs are assigned in an IA-32 MP system ensures that the right most bit field represents the maximum number of logical processors sharing the same core.

This sub-field is referred to as **SMT_ID**, and its width is related to the maximum number of unique IDs required to identify each logical processor sharing the same core. This width of this field can be zero, e.g., Pentium D processor does not support Hyper-Threading Technology. Software must use the algorithm described in this paper to determine the width dynamically.

Adjacent to the SMT_ID bit field is a bit field that represents the maximum number of processor cores in a package. This sub-field is referred to as **CORE_ID**. For a single-core processor, the width of the CORE_ID field is zero.

The remaining bit field can be used to identify a physical package in a non-clustered MP platform. The sub-field is referred to as **PACKAGE_ID**.

Figure 1 shows the layout out of the initial APIC ID (content of CPUID.1.EBX [31:24]). Note the width of each bit field depends on multithreading hardware configurations. Figures 2 through 4 depict the dependence of each bit field position on three different hardware configurations. These three examples illustrate the important point that software must not assume each bit field has a constant width, or exists at all. This paper does not discuss clustered systems.

2 Software must check CPUID for its support of leaf 4 when implementing support for multi-core. If CPUID leaf 4 is not available at runtime, software can handle the situation as if there is only one core per package

3 Typically, a clustered system is made up of many nodes of multi-processor systems. This paper applies to multi-processor systems within the same node in a clustered system. A multi-node clustered system will have a four-level topology and is beyond the scope of this paper.

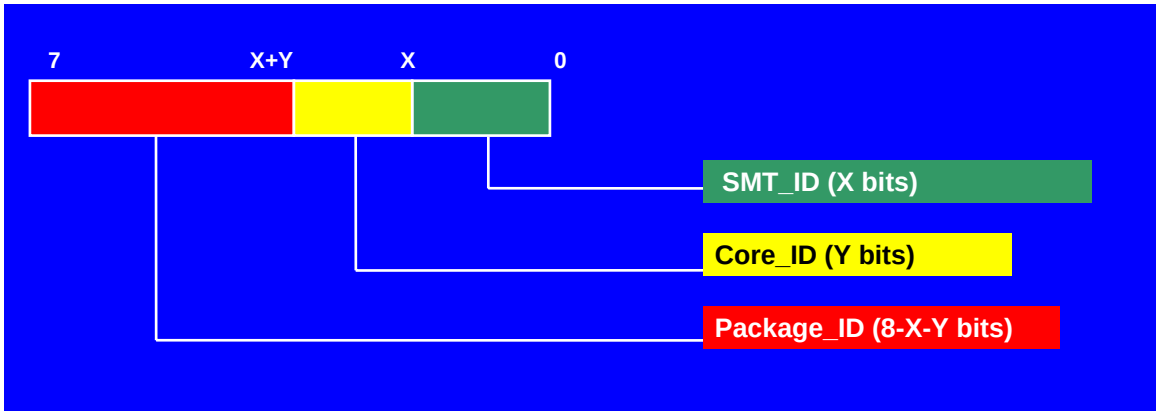


**Figure 1. Generalized bit position layout of initial APIC ID for a non-clustered system**

With a dual-core system that supports Hyper-Threading Technology, x will be equal to 1 and y will be equal to 1. This is shown in Figure 2.

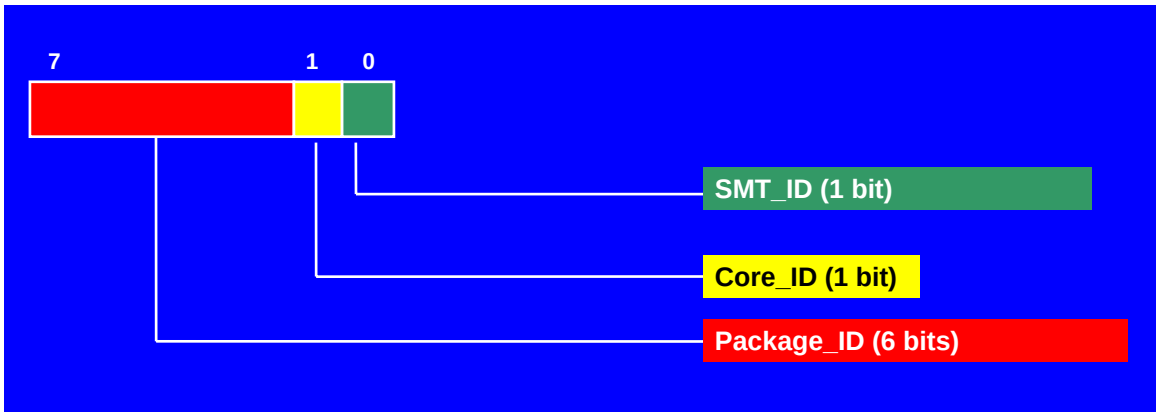


**Figure 2. Bit position layout for a dual-core system configuration supporting Hyper-Threading Technology.**

With a **dual-core** system that does not support Hyper-Threading Technology, x will be equal to 0 and y will be equal to 1. This is shown in Figure 3.

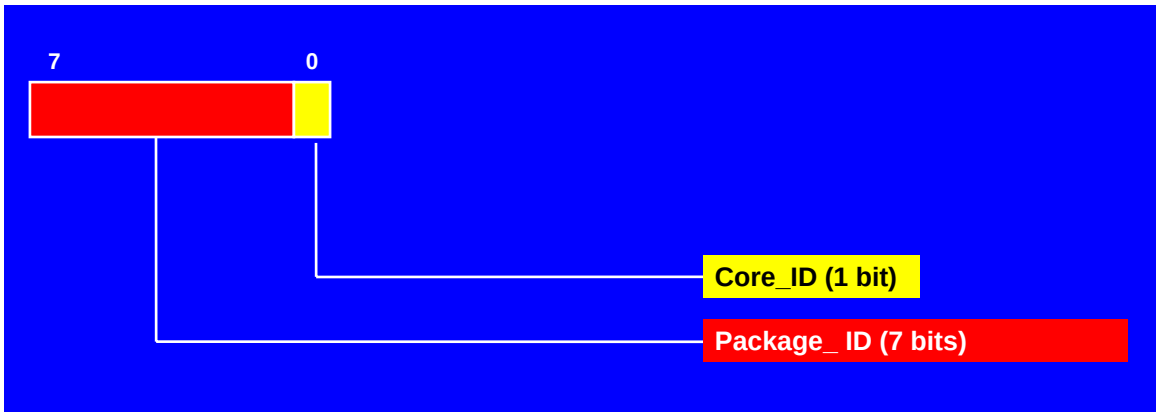


**Figure 3. Bit Position Layout for a dual-core system that does not support Hyper-Threading Technology**

With a **single-core-**system supporting Hyper-Threading Technology, x will be equal to 1 and y will be equal to 0. This is shown in Figure 4.

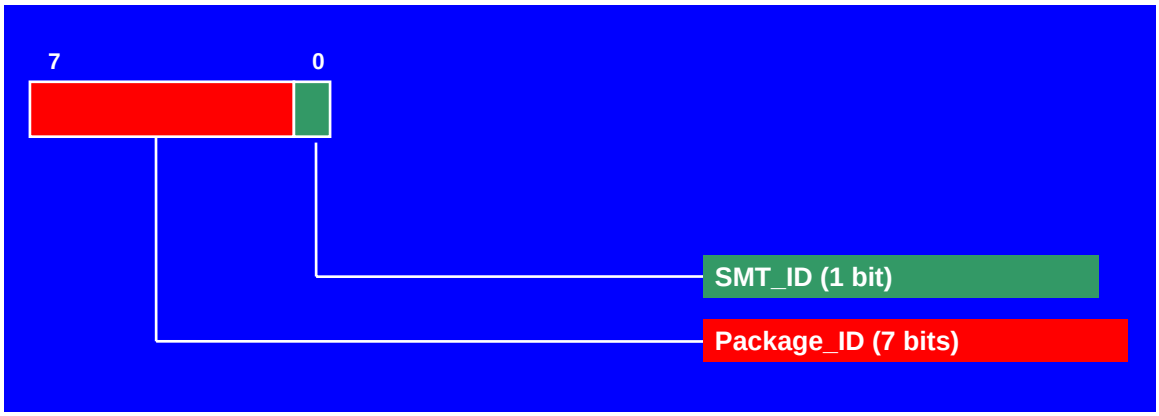


**Figure 4. Bit position layout for a single-core system supporting Hyper-Threading Technology**

An algorithm to map initial APIC ID to three-level topological IDs

The following algorithm demonstrates the method by which the initial APIC ID is decomposed into its three topological identifiers. Several support routines used in this algorithm are described in the following text.

## Detecting Hardware Support for Multi-Threading

The first support routine determines if the current logical processor is contained in a physical package with hardware multi-threading support. This is accomplished by executing the CPUID instruction with register EAX set equal to one. If bit 28 of the returned value in register EDX is set, the associated physical package supports hardware multithreading. Note that this only asserts that a physical package is capable of hardware multithreading. It does not imply that hardware multithreading is enabled, or indicate the number of logical processors in the package enabled by system software.

```
//
// The function returns 0 when the hardware multi-threaded bit is
// not set.
//
#define HWD_MT_BIT (1 << 28) // more descriptive
unsigned int is_HWMT_Supported(void)
{
unsigned int Regedx = 0;
if ((CpuIDSupported() >= 1) && GenuineIntel())
{
__asm
{
mov eax, 1
cpuid
mov Regedx, edx
}
}
return (Regedx & HWD_MT_BIT);
}
//
// CpuIDSupported will return 0 if CPUID instruction is
// unavailable.
```

```
// Otherwise, it will return
// the maximum supported standard function.
//
unsigned int CpuIDSupported(void)
{
unsigned int MaxInputValue =0;
__try // If CPUID instruction is supported
{
__asm
{
xor eax, eax // call cpuid with eax = 0
cpuid
mov MaxInputValue, eax
}
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
return(0); // cpuid instruction is unavailable
}
return MaxInputValue;
}
//
// GenuineIntel will return 0 if the processor is not a Genuine
// Intel Processor
//
unsigned int GenuineIntel(void)
{
unsigned int VendorID[3] = {0, 0, 0};
__try // If CPUID instruction is supported
{
__asm
{
xor eax, eax // call cpuid with eax = 0
cpuid // Get vendor id string
mov VendorID, ebx
mov VendorID + 4, edx
mov VendorID + 8, ecx
}
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
return(0); // cpuid instruction is unavailable
}
return ( (VendorID[0] == 'uneG') &&
(VendorID[1] == 'Ieni') &&
(VendorID[2] == 'letn'));
}
```

**Determining Maximum Logical Processors per Physical Package (Socket)**

The second support routine determines the maximum number of logical processors in a physical package. This parameter is obtained by executing CPUID with the register EAX set equal to one and storing bits 16 to 23 of register EBX on return. If an IA-32 processor does not have hardware multithreading support, the value in CPUID.1.EBX[23:16] is reserved. However, in this case, one can treat this as a discrete processor containing a maximum of one logical processor per package. Note that the number of logical processors obtained here is the maximum number of

logical processors supported by this physical package. The actual number of logical processors enabled by system software and made available to applications may be less.

```
#define NUM_LOGICAL_BITS 0xFF0000 //(or (0xFF << 16))
unsigned GetMaxNumLPperPackage(void)
{
unsigned int reg_ebx = 0;
if (!is_HWMT_supported()) return (unsigned char) 1;
__asm {
mov eax, 1
cpuid
mov reg_ebx, ebx
}
return (unsigned) ((reg_ebx & NUM_LOGICAL_BITS) >> 16);
}
```

**Determining Maximum Number of Cores in a Package**

The third support routine determines the maximum number of cores per physical package. This parameter is obtained by executing the CPUID instruction with the input value of 4 in EAX and 0 in ECX, followed by storing the returned decimal value of EAX[31:26] and incrementing it by one.

If a processor does not support theCPUID.4 leaf, then software must assume this is a single-core processor[4]. Note that if the maximum number of cores in a physical package is greater than one, it only indicates that the physical package provides more than one core. The actual number of cores enabled by system software and made available to application may be less.

```
#define NUM_CORE_BITS 0xfc000000 //(or (0xFC << 26) )
unsigned GetMaxNumCoresPerPackage(void)
{
unsigned int reg_eax = 0;
if (!is_HWMT_supported())
{
// must be single-core
return (unsigned) 1;
```

4 Software must check for the availability of CPUID leaf 4 before querying for information using CPUID leaf 4. If a BIOS allows the end user to configure a multi-core processor to work with an older operating system that is not compatible with the presence of CPUID leaf 4, the user must ensure that the proper configuration of the platform is enforced by restoring the BIOS setting to default such that CPUID leaf 4 is available. The proper configuration and optimal performance of modern OS supporting multi-core processors relies on CPUID leaf 4 to be available.

```
}
__asm {
mov eax, 0 // how many leaves does cpuid support?
cpuid
cmp eax, 4 // does cpuid support leaf 4
jl single_core // if not, must be single core
mov eax, 4 // call cpuid with eax = 4
mov ecx, 0 // start with 1st level using index = 0
cpuid
mov reg_eax, eax // Has info on number of cores
jmp multi_core
```

```
single_core:
mov reg_eax, 0 // must be single core
multi_core:
}
return (unsigned ) ((reg_eax & NUM_CORE_BITS) >> 26)+1;
}
```

**Determining the Width of a Bit Field**

An important part of the algorithm is to determine the width of each bit field. This can be accomplished using a general purpose support routine that determines the width of a bit field based on the maximum number of unique identifiers that bit field can represent. The input value to determine the width of the SMT_ID bit field is the "maximum number of logical processors sharing the same core," and the input value to determine the width of the CORE_ID bit field is the "maximum number of cores per physical package." One should not assume that the number of available threads or cores will be a power of two.

```
unsigned find_maskwidth(unsigned count_item)
{
unsigned int mask_width, cnt = count_item;
__asm
{
mov eax, cnt
mov ecx, 0
mov mask_width, ecx
dec eax
bsr cx, ax
jz next
inc cx
mov mask_width, ecx
next:
mov eax, mask_width
}
return mask_width;
}
```

**Collecting the Initial APIC IDs of Each Logical Processor**

Before putting the algorithm together, we must retrieve the initial APIC ID of each logical processor in the platform. This requires using an OS-specific processor affinity service and using an affinity mask to bind the current execution thread to a specific logical processor. **Once the current thread has successfully affinitized to the specific logical processor**, the following code will retrieve the initial APIC ID of the logical processor that this code is currently running on:

```
#define INITIAL_APIC_ID_BITS 0xff000000 //(or (0xFF << 24) )
unsigned char GetInitialApicId (void)
{
unsigned int reg_ebx = 0;
__asm {
mov eax, 1 // call cpuid with eax = 1
cpuid
mov reg_ebx, ebx // Has APIC ID info
}
return (unsigned char) ((reg_ebx & INITIAL_APIC_ID_BITS) >> 24);
}
```

**Extracting a Bit Field from an 8-bit ID**

The next routine provides the finishing touch to complete the support routines needed to decompose an 8-bit initial APIC ID into three topological identifiers. A subset of bits in an 8-bit initial APIC ID can be extracted using the appropriate bit mask and shift value. The code below allows one to extract from an 8-bit "Full_ID" a subset of bits using two other input parameters. The input parameter "MaxSubIDvalue" determines the width of bit field to extract from the "Full_ID." The input parameter "Shift_Count" specifies an offset from the right most bit of the 8-bit "Full_ID."

```
//
// This routine extracts a subset of bit fields from the 8-bit Full_ID
// The return value, or subID, is a non-zero-based, 8-bit value
//
unsigned char GetNzbSubID(unsigned char Full_ID,
unsigned char MaxSubIDvalue,
unsigned char Shift_Count)
{
unsigned MaskWidth;
unsigned char SubID, MaskBits;
MaskWidth = find_maskwidth((unsigned ) MaxSubIDvalue);
MaskBits = ((unsigned char) (0xff << Shift_Count)) ^
((unsigned char) (0xff << (Shift_Count + MaskWidth))) ;
SubID = Full_ID & MaskBits;
return SubID;
}
```

## Putting Everything Together to Extract IDs for the Three-Level Topology

Under an MP-aware OS, an application can assemble a list of all the initial APIC IDs of logical processors that are enabled by OS and made available to applications. Although there is a one-to-one mapping between each bit in an affinity mask to each unique APIC ID value, the ordering of affinity mask bits and the numerical values of initial APIC IDs are platform-specific. Because the OS allows applications to manage logical processors via affinity masks, the algorithm we describe here will construct two tables:

- One table provides a list of affinity masks corresponding to each unique PACKAGE_ID, each affinity mask includes logical processors residing in the same physical package.

- The second table provides a list of affinity masks corresponding to each unique CORE_ID; each affinity mask includes logical processors residing in the same core.

These two tables are built by first extracting the three-level identifiers from the initial APIC IDs of each logical processor. The code below illustrates how to use the support routines described previously to assemble individual tables that stores the sub-field ID of each level and for each logical processor.

To enumerate the processor and cache topology of all logical processors visible to an application, application software must rely on OS-specific services, such as affinity

APIs, to bind the current execution context to each logical processor. The code example below uses Win32 APIs to manage processor affinity. The tables below demonstrate the relationships between processor/cache topology information with OS-specific affinity constructs.

```
AFFINITY_MASK dwProcessAffinity;
AFFINITY_MASK dwSystemAffinity;
int j, numLP_enabled, MaxLPPerCore;
unsigned char apicId;
unsigned char PackageIDMask;
unsigned char tblPkg_ID[256];
unsigned char tblCore_ID[256];
unsigned char tblSMT_ID[256];
GetProcessAffinityMask(
GetCurrentProcess(),
&dwProcessAffinity,
&dwSystemAffinity);
if (dwProcessAffinity != dwSystemAffinity) {
printf ("Not all logical processors in the platform
are enabled for this process. \n");
}
j = 0;
dwAffinityMask = 1;
numLP_enabled = 0;
//
// This algorithm assumes that core within a package has the
// same number of logical processors.
// It does not assume that the value returned by
// MaxLPPerPackage()or MaxCoresPerPackage() is a power of 2.
//
MaxLPPerCore =
GetMaxNumLPperPackage()/GetMaxNumCoresPerPackage();
while (dwAffinityMask && dwAffinityMask <= dwSystemAffinity) {
if (SetThreadAffinityMask(GetCurrentThread(), dwAffinityMask)) {
Sleep(0); // Ensure this thread is on the affinitized CPU
apicId = GetInitialApicId();
//
// store SMT_ID and Core_ID of each logical processor
// Shift value for SMT_ID is 0
// Shift value for Core_ID is the mask width for maximum
// logical processors per core
//
tblSMT_ID[j] = GetNzbSubID(apicId, MaxLPPerCore, 0);
tblCore_ID[j] = GetNzbSubID(
apicId,
GetMaxNumCoresPerPackage(),
find_maskwidth(MaxLPPerCore));
//
// Extract PACKAGE_ID:
// Assume single cluster.
// Shift value is mask width for maximum logical processors
// per package
//
PackageIDMask =
((unsigned char) (0xff <<
find_maskwidth(GetMaxNumLPperPackage())));
```

```
tblPkg_ID[j] = apicId & PackageIDMask;
numLP_enabled ++;
}
j++;
dwAffinityMask = 1 << j;
}
//
// numLP_enabled contains the number logical processors in the platform
// that are enabled by system software and available for applications
//
```

## Counting Physical Packages Enabled in the Platform

Once we have the three-level topological IDs of each logical processors enabled in the platform, we can create an affinity mask to represent the sibling logical processors residing in the same physical package. The code below sorts out the initial APIC IDs in the platform and puts those initial APIC IDs with identical PACKAGE_ID into the same group, and updates the affinity mask associated with each distinct PACKAGE_ID.

```
//
// pPkgMask points to a buffer allocated by the caller
// NumStartedLPs is an integer supplied by the caller after the
// caller had assembled three tables of PKG_ID, CORE_ID and SMT_ID
//
DWORD pPkgMask[256];
unsigned char PackageIDBucket[256];
unsigned ProcessorMask;
int ProcessorNum;
int i, PkgNum = 1;
PackageIDBucket[0] = tblPkg_ID[0];
ProcessorMask = 1;
pPkgMask[0] = ProcessorMask;
for (ProcessorNum = 1; ProcessorNum < NumStartedLPs;
ProcessorNum++) {
ProcessorMask <<= 1;
for (i=0; i < PkgNum; i++) {
//
// we may be comparing bit-fields of logical processors
// residing in different packages, the code below assumes
// that the bit-masks are the same on all processors in
// the system
//
if ( tblPkg_ID[ProcessorNum] == PackageIDBucket[i]) {
pPkgMask[i] |= ProcessorMask;
break;
}
}
if (i == PkgNum) {
//
// Did not match any bucket, start new bucket
//
PackageIDBucket[i] = tblPkg_ID[ProcessorNum] ;
pPkgMask[i] = ProcessorMask;
PkgNum++;
}
}
```

```
//
// PkgNum has the actual number of physical packages enabled in
// the platform
// pPkgMask[i] has the affinity mask of the sibling logical processors
// for the i’th package
//
```

**Counting Processor Cores Enabled in the Platform**

The procedure to count OS-enabled processor cores in a platform is similar to the previous routine. Here, we compare both PACKAGE_ID and CORE_ID to distinguish whether more than one logical processors are siblings in the same core. The code below sorts out the initial APIC IDs in the platform and puts those initial APIC IDs with identical values of for PACKAGE_ID and CORE_ID into the same group and updates the affinity mask associated with each distinct core.

```
//
// pCoreProcessorMask points to a buffer allocated by the caller
// NumStartedLPs is an integer supplied by the caller after the
// caller had assembled three tables of PKG_ID, CORE_ID and SMT_ID
//
DWORD pCoreProcessorMask[256];
unsigned char CoreIDBucket [256];
unsigned ProcessorMask;
int ProcessorNum;
int i, CoreNum = 1;
CoreIDBucket[0] = tblPkg_ID[0] | tblCore_ID[0];
ProcessorMask = 1;
pCoreProcessorMask[0] = ProcessorMask;
for (ProcessorNum = 1; ProcessorNum < NumStartedLPs;
ProcessorNum++) {
ProcessorMask <<= 1;
for (i=0; i < CoreNum; i++) {
//
// we may be comparing bit-fields of logical processors
// residing in different packages, the code below assumes
// that the bit-masks are the same on all processors in
// the system
//
if (( tblPkg_ID[ProcessorNum] | tblCore_ID[ProcessorNum]) ==
CoreIDBucket[i]) {
pCoreProcessorMask[i] |= ProcessorMask;
break;
}
}
if (i == CoreNum) {
//
// Did not match any bucket, start new bucket
//
CoreIDBucket[i] = tblPkg_ID[ProcessorNum] |
tblCore_ID[ProcessorNum];
pCoreProcessorMask[i] = ProcessorMask;
CoreNum++;
}
}
//
// CoreNum has the actual number of cores enabled in the platform
// pCoreProcessorMask[i] has the affinity mask of the sibling
```

```
// logical processors for the i'th core
//
```

The appendix shows an example of how to tie all of the above code snippets together. The example will examine every logical processor visible to the running process and enumerate the processor topology and cache topology of these logical processors. The status multi-core and hyper-threading capability in the platform is also summarized. The source code of the example is listed and can be compiled in a Win32 environment as well as Linux* environment[5]. Some compilers do not support inline assembly; it is left as an exercise for the reader to complete compiler-specific customizations.

5 The source code listing can be compiled using Linux* kernel verison 2.6 or higher (e.g. RH 4AS-2.8 using GCC 3.4.4). Due to syntax variances of Linux affinity APIs with earlier kernel version and dependence on glibc library versions, compilation on Linux environment with older kernels and compilers may require kernel patches or compiler upgrades.

## Licensing Considerations

Multi-core and Hyper-Threading Technologies provide hardware multi-processing support within a physical package (or on each socket). These technologies are designed to enable mainstream adoption of the benefits of multi-processing environments. Intel recommends basing licensing policy on a per-physical-package scheme. A per-package based licensing scheme is expected to accelerate mainstream adoption of platforms with hardware multithreading support.

A per-package based licensing scheme will also accelerate end-user adoption of multithreaded applications that can best take advantage of the performance potentials of multi-cores. To implement such a licensing policy correctly, software must detect and count physical packages correctly.

Normally, using OS system calls alone will only count the total number of logical processors enabled in the platform, which is not adequate to implement a per-package licensing policy. This paper presents a more robust technique that combines data returned by the CPUID instruction regarding hardware multithreading support with some OS services to determine the three-level topology of PACKAGE_ID, CORE_ID, and SMT_ID. We have demonstrated how to use these topological identifiers to count the number of enabled physical packages and processor cores in the platform. Detecting shared cache topology can be done by adapting the algorithm presented; a follow-up paper will demonstrate such a technique.

The example in this paper also provides look-up tables of affinity masks for each package and each core, which can be used for performance tuning of multithreaded applications.

## Conclusion

Intel recommends licensing policy be based on a per-physical-package scheme. This paper presents a robust algorithm for licensing software to count the number of physical packages that are enabled for use by applications. Counting the number of physical packages correctly in a system was very important before the introduction of HT Technology, and is even more important with the introduction of multi-core processors. The techniques presented in this paper will work for past and present IA-32 processors, and are expected to work for future multi-core IA-32 processors.

## 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 **physical package**.

**Processor Core:** The circuitry that provides dedicated functionalities to decode, execute instructions, and transfer data between certain sub-systems 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.

**Hyper-Threading Technology (HT 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 core.

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

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

**Processor Topology:** Hierarchical relationships of "shared vs. dedicated" hardware resources within a computing platform using physical processors capable of one or more forms of hardware multithreading.

**Cache Topology:** Hierarchical relationships of "shared vs. dedicated" cache resources within a computing platform using physical processors capable of one or more forms of hardware multithreading.

## Appendix

To compile yourself, download the source code CpuCount.cpp

## About the Authors

Khang Nguyen is Applications Engineer working with Intel's Software and Solutions Group. He can be reached at khang.t.nguyen@intel.com.

Shihjong Kuo is Senior Technical Marketing Engineer in Intel’s Digital Enterprise Group. He can be reached at shihjong.kuo@intel.com