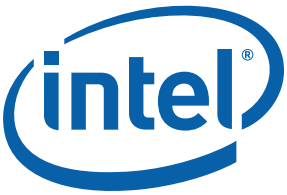


Intel® Processor Identification and the CPUID Instruction

Application Note 485

August 2009

Order Number: 241618-036

† Hyper-Threading Technology requires a computer system with an Intel® processor supporting Hyper-Threading Technology and an HT Technology enabled chipset, BIOS and operating system. Performance will vary depending on the specific hardware and software you use. See http://www.intel.com/products/ht/hyperthreading_more.htm for more information including details on which processors support HT Technology.

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

Contents

Figures

Tables

Examples

Revision History

Revision	Description	Date
-001	Original release.	05/93
-002	Modified Table 3-3 Intel486™ and Pentium® Processor Signatures.	10/93
-003	Updated to accommodate new processor versions. Program examples modified for ease of use, section added discussing BIOS recognition for OverDrive® processors and feature flag information updated.	09/94
-004	Updated with Pentium Pro and OverDrive processors information. Modified, Table 3-2, and Table 3-3. Inserted Table 3-5. Feature Flag Values Reported in the ECX Register. Inserted Sections 3.4. and 3.5.	12/95
-005	Added Figure 2-1 and Figure 3-2. Added Footnotes 1 and 2. Added Assembly code example in Section 4. Modified Tables 3, 5 and 7. Added two bullets in Section 5.0. Modified cpuid3b.ASM and cpuid3b.C programs to determine if processor features MMX™ technology. Modified Figure 6.0.	11/96
-006	Modified Table 3. Added reserved for future member of P6 family of processors entry. Modified table header to reflect Pentium II processor family. Modified Table 5. Added SEP bit definition. Added Section 3.5. Added Section 3.7 and Table 9. Corrected references of P6 family to reflect correct usage. Modified cpuid3a.asm, cpuid3b.asm and cpuid3.c example code sections to check for SEP feature bit and to check for, and identify, the Pentium II processor. Added additional disclaimer related to designers and errata.	03/97
-007	Modified Table 2. Added Pentium II processor, model 5 entry. Modified existing Pentium II processor entry to read “Pentium II processor, model 3”. Modified Table 5. Added additional feature bits, PAT and FXSR. Modified Table 7. Added entries 44h and 45h. Removed the note “Do not assume a value of 1 in a feature flag indicates that a given feature is present. For future feature flags, a value of 1 may indicate that the specific feature is not present” in section 4.0. Modified cpuid3b.asm and cpuid3.c example code section to check for, and identify, the Pentium II processor, model 5. Modified existing Pentium II processor code to print Pentium II processor, model 3.	01/98
-008	Added note to identify Intel® Celeron® processor, model 5 in section 3.2. Modified Table 2. Added Celeron processor and Pentium® OverDrive® processor with MMX™ technology entry. Modified Table 5. Added additional feature bit, PSE-36. Modified cpuid3b.asm and cpuid3.c example code to check for, and identify, the Celeron processor.	04/98
-009	Added note to identify Pentium II Xeon® processor in section 3.2. Modified Table 2. Added Pentium II Xeon processor entry. Modified cpuid3b.asm and cpuid3.c example code to check for, and identify, the Pentium II Xeon processor.	06/98
-010	No Changes	
-011	Modified Table 2. Added Celeron processor, model 6 entry. Modified cpuid3b.asm and cpuid3.c example code to check for, and identify, the Celeron processor, model 6.	12/98
-012	Modified Figure 1 to add the reserved information for the Intel386 processors. Modified Figure 2. Added the Processor serial number information returned when the CPUID instruction is executed with EAX=3. Modified Table 1. Added the Processor serial number parameter. Modified Table 2. Added the Pentium III processor and Pentium III Xeon processor. Added Section 4 “Processor serial number”. Modified cpuid3a.asm, cpuid3b.asm and cpuid3.c example code to check for and identify the Pentium III processor and the Pentium III Xeon processor.	12/98
-013	Modified Figure 2. Added the Brand ID information returned when the CPUID instruction is executed with EAX=1. Added section 5 “Brand ID”. Added Table 10 that shows the defined Brand ID values. Modified cpuid3a.asm, cpuid3b.asm and cpuid3.c example code to check for and identify the Pentium III processor, model 8 and the Pentium III Xeon processor, model 8.	10/99
-014	Modified Table 4. Added Celeron processor, model 8.	03/00
-015	Modified Table 4. Added Pentium III Xeon processor, model A. Added the 8-way set associative 1M, and 8-way set associative 2M cache descriptor entries.	05/00

Revision	Description	Date
-016	Revised Figure 2 to include the Extended Family and Extended Model when CPUID is executed with EAX=1. Added section 6 which describes the Brand String. Added section 10 Alternate Method of Detecting Features and sample code. Added the Pentium 4 processor signature to Table 4. Added new feature flags (SSE2, SS and TM) to Table 5. Added new cache descriptors to Table 3-7. Removed Pentium Pro cache descriptor example.	11/00
-017	Modified Figure 2 to include additional features reported by the Pentium 4 processors. Modified to include additional Cache and TLB descriptors defined by the Intel NetBurst® microarchitecture. Added Section 9 and program Example 5 which describes how to detect if a processor supports the DAZ feature. Added Section 10 and program Example 6 which describes a method of calculating the actual operating frequency of the processor.	02/01
-018	Changed the second 66h cache descriptor in Table 7 to 68h. Added the 83h cache descriptor to Table 7. Added the Pentium III processor, model B, processor signature and the Intel Xeon processor, processor signature to Table 4. Modified Table 4 to include the extended family and extended model fields. Modified Table 1 to include the information returned by the extended CPUID functions.	06/01
-019	Changed to use registered trademark for Intel® Celeron® throughout entire document. Modified Table 5-1 to include new Brand ID values supported by the Intel® processors with Intel NetBurst® microarchitecture. Added Hyper-Threading Technology Flag to Table 3-4 and Logical Processor Count to Figure 3-1. Modified cpuid3b.asm and cpuid3.c example code to check for and identify Intel® processors based on the updated Brand ID values contained in Table 5-1.	01/02
-020	Modified Table 3-7 to include new Cache Descriptor values supported by the Intel processors with Intel NetBurst microarchitecture. Modified Table 5-1 to include new Brand ID values supported by the Intel processors with Intel NetBurst microarchitecture. Modified cpuid3b.asm and cpuid3.c example code to check for and identify Intel® processors based on the updated Brand ID values contained in Table 5-1.	03/02
-021	Modified Table 3-3 to include additional processors that return a processor signature with a value in the family code equal to 0Fh. Modified Table 3-7 to include new Cache Descriptor values supported by various Intel processors. Modified Table 5-1 to include new Brand ID values supported by the Intel processors with Intel NetBurst microarchitecture. Modified cpuid3b.asm and cpuid3.c example code to check for and identify Intel processors based on the updated Brand ID values contained in Table 5-1.	05/02
-022	Modified Table 3-7 with correct Cache Descriptor descriptions. Modified Table 3-4 with new feature flags returned in EDX. Added Table 3-5. Feature Flag Values Reported in the ECX Register the feature flags returned in ECX. Modified Table 3-3, broke out the processors with family 'F' by model numbers.	11/02
-023	Modified Table 3-3, added the Intel® Pentium® M processor. Modified Table 3-4 with new feature flags returned in EDX. Modified Table 3-5. Feature Flag Values Reported in the ECX Register the feature flags returned in ECX. Modified Table 3-7 with correct Cache Descriptor descriptions.	03/03
-024	Corrected feature flag definitions in Table 3-5. Feature Flag Values Reported in the ECX Register for bits 7 and 8.	11/03

Revision	Description	Date
-025	Modified Table 1 to add Deterministic Cache Parameters function (CPUID executed with EAX=4), MONITOR/MWAIT function (CPUID instruction is executed with EAX=5), Extended L2 Cache Features function (CPUID executed with EAX=80000006), Extended Addresses Sizes function (CPUID is executed with EAX=80000008). Modified Table 1 and Table 5 to reinforce no PSN on Pentium® 4 family processors. Modified, added the Intel® Pentium® 4 processor and Intel® Celeron® processor on 90nm process. Modified Table 3-5. Feature Flag Values Reported in the ECX Register to add new feature flags returned in ECX. Modified Table 3-7 to include new Cache Descriptor values supported by various Intel processors. Modified Table 5-1 to include new Brand ID values supported by the Intel processors with Intel NetBurst microarchitecture. Modified cpuid3b.asm and cpuid3.c example code to check for and identify Intel processors based on the updated Brand ID values contained in Table 5-1. Modified features.cpp, cpuid3.c, and cpuid3a.asm to check for and identify new feature flags based on the updated values in Table 3-5. Feature Flag Values Reported in the ECX Register.	01/04
-026	Corrected the name of the feature flag returned in EDX[31] (PBE) when the CPUID instruction is executed with EAX set to a 1. Modified Table 3-15 to indicate CPUID function 80000001h now returns extended feature flags in the EAX register. Added the Intel® Pentium® M processor (family 6, model D) to Table 3-3. Added section 3.2.2. Modified Table 3-5. Feature Flag Values Reported in the ECX Register to add new feature flags returned in ECX. Modified Table 3-5. Feature Flag Values Reported in the ECX Register to include new Cache Descriptor values supported by various Intel processors. Modified Table 5-1 to include new Brand ID values supported by the Intel processors with P6 family microarchitecture. Modified cpuid3b.asm and cpuid3.c example code to check for and identify Intel processors based on the updated Brand ID values contained in Table 5-1. Modified features.cpp, cpuid3.c, and cpuid3a.asm to check for and identify new feature flags based on the updated values in Table 3-5. Feature Flag Values Reported in the ECX Register.	05/04
-027	Corrected the register used for Extended Feature Flags in Section 3.2.2	07/04
-028	Corrected bit field definitions for CPUID functions 80000001h and 80000006h. Added processor names for family 'F', model '4' to Table 3-3. Updated Table 3-5. Feature Flag Values Reported in the ECX Register to include the feature flag definition (ECX[13]) for the CMPXCHG16B instruction. Updated Table 3-15 to include extended feature flag definitions for (EDX[11]) SYSCALL / SYSRET and (EDX[20]) Execute Disable bit. Updated Example 1 to extract CPUID extended function information. Updated Example 2 and Example 3 to detect and display extended features identified by CPUID function 80000001h.	02/05
-029	Modified Table 3-7 to include new Cache Descriptor values supported by various Intel processors.	03/05
-030	Corrected Table 3-16. Extended Feature Flag Values Reported in the ECX Register. Added CPUID function 6, Power management Feature to Table 3-1. Updated Table 3-5 to include the feature flag definition (EDX[30]) for IA64 capabilities. Updated Table 3-10 to include new Cache Descriptor values supported by Intel Pentium 4 processors. Modified cpuid3b.asm and cpuid3.c example code to check for IA64 capabilities, CMPXCHG16B, LAHF/SAHF instructions.	01/06
-031	Update Intel® EM64T portions with new naming convention for Intel® 64 Instruction Set Architecture. Added section Composing the Family, Model and Stepping (FMS) values Added section Extended CPUID Functions Updated Table 3-4 to include the Intel Core 2 Duo processor family. Updated Table 3-6 to include the feature flag definitions for VMX, SSSE3 and DCA. Updated Table 3-10 to include new Cache Descriptor values supported by Intel Core 2 Duo processor. Update CPUFREQ.ASM with alternate method to determine frequency without using TSC.	09/06

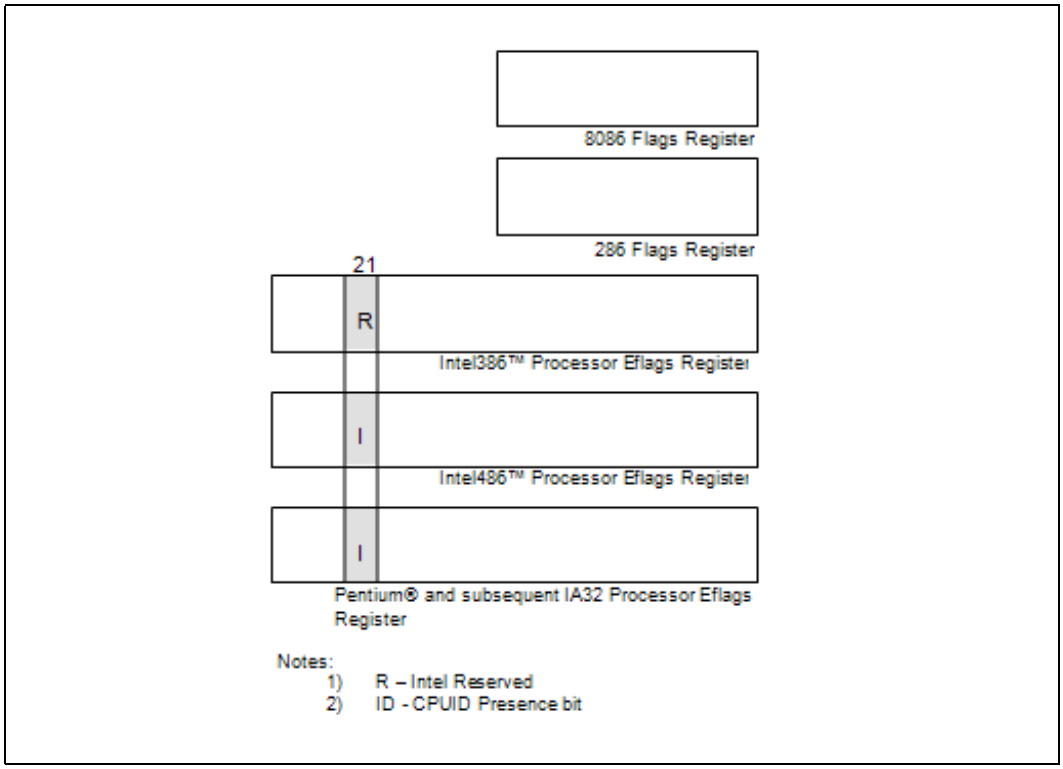
Revision	Description	Date
-032	Updated Table 3-10 to correct Cache Descriptor description. Updated trademarks for various processors. Added the Architectural Performance Monitor Features (Function Ah) section. Updated the supported processors in Table 3-3. Updated the feature flags reported by function 1 and reported in ECX, see Table 3-5. Updated the CPUID3A.ASM, CPUID3B.ASM and CPUID3.C sample code Removed the Alternate Method of Detecting Features chapter and sample code.	12/07
-033	Intel® Atom™ and Intel® Core™ i7 processors added to text and examples Updated Table 2-3 to include Intel Atom and Intel Core i7 processors Updated ECX and EDX feature flag definitions in Table 2-4 and Table 2-5 Updated cache and TLB descriptor values in Table 2-7 Updated Table 2-8 to feature Intel Core i7 processor Updated Section 2.1.3.1 and Table 2-8 to include Intel Core i7 processor Modified Table 2-10 to include new C-state definitions Updated Table 2-11 to include Intel® Turbo Boost Technology Added Section 2.1.13, "Reserved (Function 0Ch)" Combined Chapter 8: Usage Program Examples with Chapter 11: Program Examples into one chapter: Chapter 10: Program Examples	11/08
-034	Updated Table 3-3 processor signatures Corrected Intel® Core™ i7 processor Model No. data in Table 3-3 Updated Table 3-7 cache descriptor decode values Modified CPUID3A.ASM, CPUID3B.ASM and CPUID3.C: - Added detection of additional feature flags - Updated text output Minor updates to DAZDETCT.ASM & CPUFREQ.ASM	03/09
-035	Modified CPUID3A.ASM, CPUID3B.ASM and CPUID3.C: - Redesigned code so features are in tables for easier maintenance - Added Feature Flags per Software Developer Manual June 2009 - Added output for CPUID Function 04h, 05h, 0Ah, 0Bh, 0Dh. Modified FREQUENC.ASM, and added CPUFREQ.C: - Updated frequency code with APERF and MPERF MSR calculations	07/09
-036	Corrected CPUID Function 04h text to specify EAX[31:26] is APIC IDs reserved for the package.	08/09

§

1 Detecting the CPUID Instruction

The Intel486 family and subsequent Intel processors provide a straightforward method for determining whether the processor's internal architecture is able to execute the CPUID instruction. This method uses the ID flag in bit 21 of the EFLAGS register. If software can change the value of this flag, the CPUID instruction is executable[1] (see Figure 1-1).

Figure 1-1. Flag Register Evolution



The POPF, POPFD, PUSHF, and PUSHFD instructions are used to access the flags in EFLAGS register. The program examples at the end of this application note show how to use the PUSHFD instruction to read and the POPFD instruction to change the value of the ID flag.

§

1. Only in some Intel486™ and succeeding processors. Bit 21 in the Intel386™ processor's Eflag register cannot be changed by software, and the Intel386 processor cannot execute the CPUID instruction. Execution of CPUID on a processor that does not support this instruction will result in an invalid opcode exception.

1 Introduction

As the Intel® Architecture evolves with the addition of new generations and models of processors (8086, 8088, Intel286, Intel386™, Intel486™, Pentium® processors, Pentium® OverDrive® processors, Pentium® processors with MMX™ technology, Pentium® OverDrive® processors with MMX™ technology, Pentium® Pro processors, Pentium® II processors, Pentium® II Xeon® processors, Pentium® II Overdrive® processors, Intel® Celeron® processors, Mobile Intel® Celeron® processors, Intel® Celeron® D processors, Intel® Celeron® M processors, Pentium® III processors, Mobile Intel® Pentium® III processor - M, Pentium® III Xeon® processors, Pentium® 4 processors, Mobile Intel® Pentium® 4 processor – M, Intel® Pentium® M processor, Intel® Pentium® D processor, Pentium® processor Extreme Edition, Intel® Pentium® dual-core processor, Intel® Pentium® dual-core mobile processor, Intel® Core™ Solo processor, Intel® Core™ Duo processor, Intel® Core™ Duo mobile processor, Intel® Core™2 Duo processor, Intel® Core™2 Duo mobile processor, Intel® Core™2 Quad processor, Intel® Core™2 Extreme processor, Intel® Core™2 Extreme mobile processor, Intel® Xeon® processors, Intel® Xeon® processor MP, Intel® Atom™ processor, and Intel® Core™ i7 processor), it is essential that Intel provide an increasingly sophisticated means with which software can identify the features available on each processor. This identification mechanism has evolved in conjunction with the Intel Architecture as follows:

1. Originally, Intel published code sequences that could detect minor implementation or architectural differences to identify processor generations.
2. With the advent of the Intel386 processor, Intel implemented processor signature identification that provided the processor family, model, and stepping numbers to software, but only upon reset.
3. As the Intel Architecture evolved, Intel extended the processor signature identification into the CPUID instruction. The CPUID instruction not only provides the processor signature, but also provides information about the features supported by and implemented on the Intel processor.

This evolution of processor identification was necessary because, as the Intel Architecture proliferates, the computing market must be able to tune processor functionality across processor generations and models with differing sets of features. Anticipating that this trend will continue with future processor generations, the Intel Architecture implementation of the CPUID instruction is extensible.

This application note explains how to use the CPUID instruction in software applications, BIOS implementations, and various processor tools. By taking advantage of the CPUID instruction, software developers can create software applications and tools that can execute compatibly across the widest range of Intel processor generations and models, past, present, and future.

1.1 Update Support

Intel processor signature and feature bits information can be obtained from the developer's manual, programmer's reference manual and appropriate processor documentation. In addition, updated versions of the programming examples included in this application note are available through your Intel representative, or visit Intel's website at http://developer.intel.com/.

§

2 Output of the CPUID Instruction

The CPUID instruction supports two sets of functions. The first set returns basic processor information; the second set returns extended processor information. Figure 2-1 summarizes the basic processor information output by the CPUID instruction. The output from the CPUID instruction is fully dependent upon the contents of the EAX register. This means that, by placing different values in the EAX register and then executing CPUID, the CPUID instruction will perform a specific function dependent upon whatever value is resident in the EAX register. In order to determine the highest acceptable value for the EAX register input and CPUID functions that return the basic processor information, the program should set the EAX register parameter value to “0” and then execute the CPUID instruction as follows:

```
MOV EAX, 00h
CPUID
```

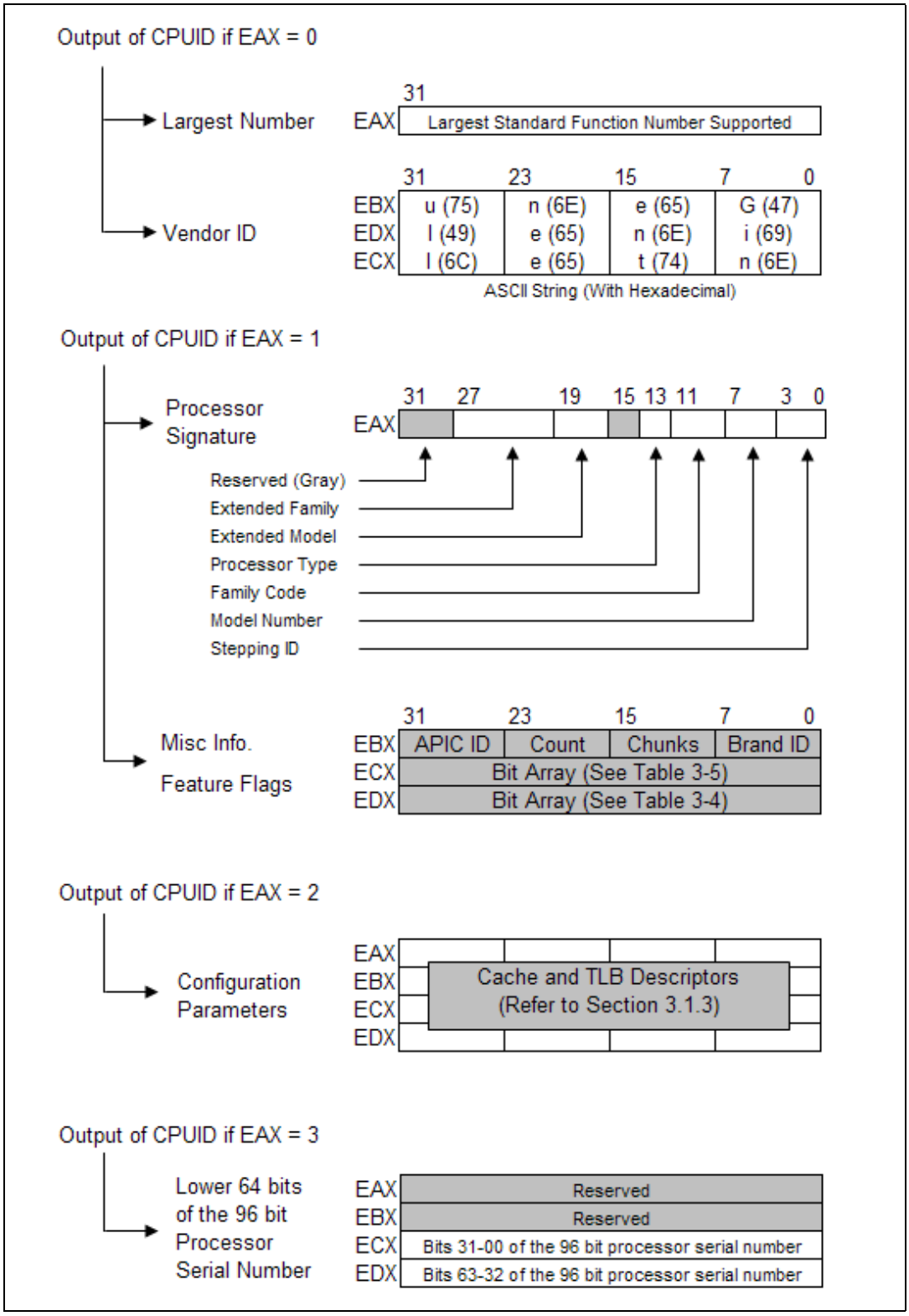
After the execution of the CPUID instruction, a return value will be present in the EAX register. Always use an EAX parameter value that is equal to or greater than zero and less than or equal to this highest EAX “returned” value.

In order to determine the highest acceptable value for the EAX register input and CPUID functions that return the extended processor information, the program should set the EAX register parameter value to “80000000h” and then execute the CPUID instruction as follows:

```
MOV EAX, 80000000h
CPUID
```

After the execution of the CPUID instruction, a return value will be present in the EAX register. Always use an EAX parameter value that is equal to or greater than 80000000h and less than or equal to this highest EAX “returned” value. On current and future IA-32 processors, bit 31 in the EAX register will be clear when CPUID is executed with an input parameter greater than the highest value for either set of functions, and when the extended functions are not supported. All other bit values returned by the processor in response to a CPUID instruction with EAX set to a value higher than appropriate for that processor are model specific and should not be relied upon.

Figure 2-1. CPUID Instruction Outputs



2.1 Standard CPUID Functions

2.1.1 Vendor-ID and Largest Standard Function (Function 0)

In addition to returning the largest standard function number in the EAX register, the Intel Vendor-ID string can be verified at the same time. If the EAX register contains an input value of 0, the CPUID instruction also returns the vendor identification string in the EBX, EDX, and ECX registers (see Figure 2-1). These registers contain the ASCII string:

GenuineIntel

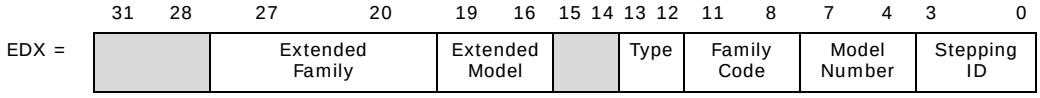
While any imitator of the Intel Architecture can provide the CPUID instruction, no imitator can legitimately claim that its part is a genuine Intel part. The presence of the "GenuineIntel" string is an assurance that the CPUID instruction and the processor signature are implemented as described in this document. If the "GenuineIntel" string is not returned after execution of the CPUID instruction, do not rely upon the information described in this document to interpret the information returned by the CPUID instruction.

2.1.2 Feature Information (Function 01h)

2.1.2.1 Processor Signature

Beginning with the Intel486 processor family, the EDX register contains the processor identification signature after RESET (see Figure 2-2). **The processor identification signature is a 32-bit value**. The processor signature is composed from eight different bit fields. The fields in gray represent reserved bits, and should be masked out when utilizing the processor signature. The remaining six fields form the processor identification signature.

Figure 2-2. EDX Register After RESET



Processors that implement the CPUID instruction also return the 32-bit processor identification signature after reset. However, the CPUID instruction gives you the flexibility of checking the processor signature at any time. Figure 2-2 shows the format of the 32-bit processor signature for the Intel486 and subsequent Intel processors. Note that the EDX processor signature value after reset is equivalent to the processor signature output value in the EAX register in Figure 2-1. Table 2-3 below shows the values returned in the EAX register currently defined for these processors.

The extended family, bit positions 20 through 27 are used in conjunction with the family code, specified in bit positions 8 through 11, to indicate whether the processor belongs to the Intel386, Intel486, Pentium, Pentium Pro or Pentium 4 family of processors. P6 family processors include all processors based on the Pentium Pro processor architecture and have an extended family equal to 00h and a family code equal to 06h. Pentium 4 family processors include all processors based on the Intel NetBurst® microarchitecture and have an extended family equal to 00h and a family code equal to 0Fh.

The extended model specified in bit positions 16 through 19, in conjunction with the model number specified in bits 4 though 7 are used to identify the model of the processor within the processor's family. The stepping ID in bits 0 through 3 indicates the revision number of that model.

The processor type values returned in bits 12 and 13 of the EAX register are specified in Table 2-1 below. These values indicate whether the processor is an original OEM processor, an OverDrive processor, or a dual processor (capable of being used in a dual processor system).

Table 2-1. Processor Type (Bit Positions 13 and 12)

Value	Description
00	Original OEM Processor
01	OverDrive® Processor
10	Dual Processor

The Pentium II processor, model 5, the Pentium II Xeon processor, model 5, and the Celeron processor, model 5 share the same extended family, family code, extended model and model number. To differentiate between the processors, software should check the cache descriptor values through executing CPUID instruction with EAX = 2. If no L2 cache is returned, the processor is identified as an Intel® Celeron® processor, model 5. If 1-MB or 2-MB L2 cache size is reported, the processor is the Pentium II Xeon processor otherwise it is a Pentium II processor, model 5 or a Pentium II Xeon processor with 512-KB L2 cache.

The Pentium III processor, model 7, and the Pentium III Xeon processor, model 7, share the same extended family, family code, extended model and model number. To differentiate between the processors, software should check the cache descriptor values through executing CPUID instruction with EAX = 2. If 1M or 2M L2 cache size is reported, the processor is the Pentium III Xeon processor otherwise it is a Pentium III processor or a Pentium III Xeon processor with 512 KB L2 cache.

The processor brand for the Pentium III processor, model 8, the Pentium III Xeon processor, model 8, and the Celeron processor, model 8, can be determined by using the Brand ID values returned by the CPUID instruction when executed with EAX equal to 01h. Further information regarding Brand ID and Brand String is detailed in Chapter 4 of this document.

Older versions of Intel486 SX, Intel486 DX and IntelDX2™ processors do not support the CPUID instruction, and return the processor signature only at reset.[1] Refer to Table 2-3 to determine which processors support the CPUID instruction.

Figure 2-3 shows the format of the processor signature for Intel386 processors. The Intel386 processor signature is different from the signature of other processors. Table 2-2 provides the processor signatures of Intel386™ processors.

1. All Intel486 SL-enhanced and Write-Back enhanced processors are capable of executing the CPUID instruction. See Table 2-3.

Figure 2-3. Processor Signature Format on Intel386™ Processors

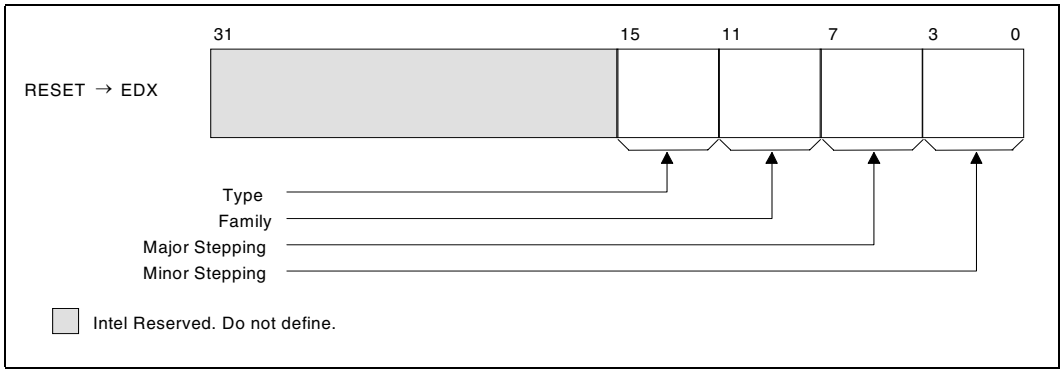


Table 2-2. Intel386™ Processor Signatures

Type	Family	Major Stepping	Minor Stepping	Description
0000	0011	0000	xxxx	Intel386™ DX processor
0010	0011	0000	xxxx	Intel386 SX processor
0010	0011	0000	xxxx	Intel386 CX processor
0010	0011	0000	xxxx	Intel386 EX processor
0100	0011	0000 and 0001	xxxx	Intel386 SL processor
0000	0011	0100	xxxx	RapidCAD* coprocessor

Table 2-3. Intel486™ and Subsequent Processor Signatures (Sheet 1 of 4)

Extended Family	Extended Model	Type	Family Code	Model No.	Stepping ID	Description
00000000	0000	00	0100	000x	xxxx (1)	Intel486™ DX processors
00000000	0000	00	0100	0010	xxxx (1)	Intel486 SX processors
00000000	0000	00	0100	0011	xxxx (1)	Intel487™ processors
00000000	0000	00	0100	0011	xxxx (1)	IntelDX2™ processors
00000000	0000	00	0100	0011	xxxx (1)	IntelDX2 OverDrive® processors
00000000	0000	00	0100	0100	xxxx (3)	Intel486 SL processor
00000000	0000	00	0100	0101	xxxx (1)	IntelSX2™ processors
00000000	0000	00	0100	0111	xxxx (3)	Write-Back Enhanced IntelDX2 processors
00000000	0000	00	0100	1000	xxxx (3)	IntelDX4™ processors
00000000	0000	0x	0100	1000	xxxx (3)	IntelDX4 OverDrive processors
00000000	0000	00	0101	0001	xxxx (2)	Pentium® processors (60, 66)
00000000	0000	00	0101	0010	xxxx (2)	Pentium processors (75, 90, 100, 120, 133, 150, 166, 200)

Table 2-3. Intel486™ and Subsequent Processor Signatures (Sheet 2 of 4)

Extended Family	Extended Model	Type	Family Code	Model No.	Stepping ID	Description
00000000	0000	01(4)	0101	0001	xxxx (2)	Pentium OverDrive processor for Pentium processor (60, 66)
00000000	0000	01(4)	0101	0010	xxxx (2)	Pentium OverDrive processor for Pentium processor (75, 90, 100, 120, 133)
00000000	0000	01	0101	0011	xxxx (2)	Pentium OverDrive processors for Intel486 processor-based systems
00000000	0000	00	0101	0100	xxxx (2)	Pentium processor with MMX™ technology (166, 200)
00000000	0000	01	0101	0100	xxxx (2)	Pentium OverDrive processor with MMX™ technology for Pentium processor (75, 90, 100, 120, 133)
00000000	0000	00	0110	0001	xxxx (2)	Pentium Pro processor
00000000	0000	00	0110	0011	xxxx (2)	Pentium II processor, model 03
00000000	0000	00	0110	0101 (5)	xxxx (2)	Pentium II processor, model 05, Pentium II Xeon processor, model 05, and Intel® Celeron® processor, model 05
00000000	0001	00	0110	0101	xxxx (2)	Intel EP80579 Integrated Processor and Intel EP80579 Integrated Processor with Intel QuickAssist Technology
00000000	0000	00	0110	0110	xxxx (2)	Celeron processor, model 06
00000000	0000	00	0110	0111 (6)	xxxx (2)	Pentium III processor, model 07, and Pentium III Xeon processor, model 07
00000000	0000	00	0110	1000 (7)	xxxx (2)	Pentium III processor, model 08, Pentium III Xeon processor, model 08, and Celeron processor, model 08
00000000	0000	00	0110	1001	xxxx (2)	Intel Pentium M processor, Intel Celeron M processor model 09.
00000000	0000	00	0110	1010	xxxx (2)	Pentium III Xeon processor, model 0Ah
00000000	0000	00	0110	1011	xxxx (2)	Pentium III processor, model 0Bh
00000000	0000	00	0110	1101	xxxx (2)	Intel Pentium M processor, Intel Celeron M processor, model 0Dh. All processors are manufactured using the 90 nm process.
00000000	0000	00	0110	1110	xxxx (2)	Intel Core™ Duo processor, Intel Core™ Solo processor, model 0Eh. All processors are manufactured using the 65 nm process.

Table 2-3. Intel486™ and Subsequent Processor Signatures (Sheet 3 of 4)

Extended Family	Extended Model	Type	Family Code	Model No.	Stepping ID	Description
00000000	0000	00	0110	1111	xxxx (2)	Intel Core™2 Duo processor, Intel Core™2 Duo mobile processor, Intel Core™2 Quad processor, Intel Core™2 Quad mobile processor, Intel Core™2 Extreme processor, Intel Pentium Dual-Core processor, Intel Xeon processor, model 0Fh. All processors are manufactured using the 65 nm process.
00000000	0001	00	0110	0110	xxxx (2)	Intel Celeron processor model 16h. All processors are manufactured using the 65 nm process
00000000	0001	00	0110	0111	xxxx (2)	Intel Core™2 Extreme processor, Intel Xeon processor, model 17h. All processors are manufactured using the 45 nm process.
00000000	0000	01	0110	0011	xxxx (2)	Intel Pentium II OverDrive processor
00000000	0000	00	1111	0000	xxxx (2)	Pentium 4 processor, Intel Xeon processor. All processors are model 00h and manufactured using the 0.18 micron process.
00000000	0000	00	1111	0001	xxxx (2)	Pentium 4 processor, Intel Xeon processor, Intel Xeon processor MP, and Intel Celeron processor. All processors are model 01h and manufactured using the 0.18 micron process.
00000000	0000	00	1111	0010	xxxx (2)	Pentium 4 processor, Mobile Intel Pentium 4 processor – M, Intel Xeon processor, Intel Xeon processor MP, Intel Celeron processor, and Mobile Intel Celeron processor. All processors are model 02h and manufactured using the 0.13 micron process.
00000000	0000	00	1111	0011	xxxx (2)	Pentium 4 processor, Intel Xeon processor, Intel Celeron D processor. All processors are model 03h and manufactured using the 90 nm process.
00000000	0000	00	1111	0100	xxxx (2)	Pentium 4 processor, Pentium 4 processor Extreme Edition, Pentium D processor, Intel Xeon processor, Intel Xeon processor MP, Intel Celeron D processor. All processors are model 04h and manufactured using the 90 nm process.

Table 2-3. Intel486™ and Subsequent Processor Signatures (Sheet 4 of 4)

Extended Family	Extended Model	Type	Family Code	Model No.	Stepping ID	Description
00000000	0000	00	1111	0110	xxxx (2)	Pentium 4 processor, Pentium D processor, Pentium processor Extreme Edition, Intel Xeon processor, Intel Xeon processor MP, Intel Celeron D processor. All processors are model 06h and manufactured using the 65 nm process.
00000000	0001	00	0110	1100	xxxx (2)	Intel Atom processor. All processors are manufactured using the 45 nm process
00000000	0001	00	0110	1010	xxxx (2)	Intel Core i7 processor and Intel Xeon processor. All processors are manufactured using the 45 nm process.
00000000	0001	00	0110	1101	xxxx(2)	Intel Xeon processor MP. All processors are manufactured using the 45 nm process.

Notes:
1. This processor does not implement the CPUID instruction.
2. Refer to the Intel486™ documentation, the Pentium® Processor Specification Update (Document Number 242480), the Pentium® Pro Processor Specification Update (Document Number 242689), the Pentium® II Processor Specification Update (Document Number 243337), the Pentium® II Xeon Processor Specification Update (Document Number 243776), the Intel® Celeron® Processor Specification Update (Document Number 243748), the Pentium ® III Processor Specification Update (Document Number 244453), the Pentium® III Xeon® Processor Specification Update (Document Number 244460), the Pentium® 4 Processor Specification Update (Document Number 249199), the Intel® Xeon® Processor Specification Update (Document Number 249678) or the Intel® Xeon® Processor MP Specification Update (Document Number 290741) for the latest list of stepping numbers.
3. Stepping 3 implements the CPUID instruction.
4. The definition of the type field for the OverDrive processor is 01h. An erratum on the Pentium OverDrive processor will always return 00h as the type.
5. To differentiate between the Pentium II processor, model 5, Pentium II Xeon processor and the Celeron processor, model 5, software should check the cache descriptor values through executing CPUID instruction with EAX = 2. If no L2 cache is returned, the processor is identified as an Celeron processor, model 5. If 1M or 2M L2 cache size is reported, the processor is the Pentium II Xeon processor otherwise it is a Pentium II processor, model 5 or a Pentium II Xeon processor with 512-KB L2 cache size.
6. To differentiate between the Pentium III processor, model 7 and the Pentium III Xeon processor, model 7, software should check the cache descriptor values through executing CPUID instruction with EAX = 2. If 1M or 2M L2 cache size is reported, the processor is the Pentium III Xeon processor otherwise it is a Pentium III processor or a Pentium III Xeon processor with 512-KB L2 cache size.
7. To differentiate between the Pentium III processor, model 8 and the Pentium III Xeon processor, model 8, software should check the Brand ID values through executing CPUID instruction with EAX = 1.
8. To differentiate between the processors with the same processor Vendor ID, software should execute the Brand String functions and parse the Brand String.

2.1.2.2 Composing the Family, Model and Stepping (FMS) values

The processor family is an 8-bit value obtained by adding the Extended Family field of the processor signature returned by CPUID Function 1 with the Family field.

Equation 2-1.Calculated Family Value

```
F = Extended Family + Family
F = CPUID(1).EAX[27:20] + CPUID(1).EAX[11:8]
```

The processor model is an 8-bit value obtained by shifting left 4 the Extended Model field of the processor signature returned by CPUID Function 1 then adding the Model field.

Equation 2-2.Calculated Model Value

```
M = (Extended Model << 4) + Model
M = (CPUID(1).EAX[19:16] << 4) + CPUID(1).EAX[7:4]
```

The processor stepping is a 4-bit value obtained by copying the *Stepping* field of the processor signature returned by CPUID function 1.

Equation 2-3.Calculated Stepping Value

```
S = Stepping
S = CPUID(1).EAX[3:0]
```

Recommendations for Testing Compliance

New and existing software should be inspected to ensure code always uses:

1. The full 32-bit value when comparing processor signatures;
2. The full 8-bit value when comparing processor families, the full 8-bit value when comparing processor models; and
3. The 4-bit value when comparing processor steppings.

2.1.2.3 Feature Flags

When the EAX register contains a value of 1, the CPUID instruction (in addition to loading the processor signature in the EAX register) loads the EDX and ECX register with the feature flags. The feature flags (when a Flag = 1) indicate what features the processor supports. Table 2-4 and Table 2-5 detail the currently-defined feature flag values.

For future processors, refer to the programmer's reference manual, user's manual, or the appropriate documentation for the latest feature flag values.

Use the feature flags in applications to determine which processor features are supported. By using the CPUID feature flags to determine processor features, software can detect and avoid incompatibilities introduced by the addition or removal of processor features.

Table 2-4. Feature Flag Values Reported in the EDX Register (Sheet 1 of 3)

Bit	Name	Description when Flag = 1	Comments
0	FPU	Floating-point Unit On-Chip	The processor contains an FPU that supports the Intel387 floating-point instruction set.
1	VME	Virtual Mode Extension	The processor supports extensions to virtual-8086 mode.
2	DE	Debugging Extension	The processor supports I/O breakpoints, including the CR4.DE bit for enabling debug extensions and optional trapping of access to the DR4 and DR5 registers.
3	PSE	Page Size Extension	The processor supports 4-MB pages.
4	TSC	Time Stamp Counter	The RDTSC instruction is supported including the CR4.TSD bit for access/privilege control.
5	MSR	Model Specific Registers	Model Specific Registers are implemented with the RDMSR, WRMSR instructions
6	PAE	Physical Address Extension	Physical addresses greater than 32 bits are supported.
7	MCE	Machine-Check Exception	Machine-Check Exception, INT18, and the CR4.MCE enable bit are supported.

Table 2-4. Feature Flag Values Reported in the EDX Register (Sheet 2 of 3)

Bit	Name	Description when Flag = 1	Comments
8	CX8	CMPXCHG8 Instruction	The compare and exchange 8-bytes instruction is supported.
9	APIC	On-chip APIC Hardware	The processor contains a software-accessible local APIC.
10		Reserved	Do not count on their value.
11	SEP	Fast System Call	Indicates whether the processor supports the Fast System Call instructions, SYSENTER and SYSEXIT. NOTE: Refer to Section 2.1.2.4 for further information regarding SYSENTER/SYSEXIT feature and SEP feature bit.
12	MTRR	Memory Type Range Registers	The processor supports the Memory Type Range Registers specifically the MTRR_CAP register.
13	PGE	Page Global Enable	The global bit in the page directory entries (PDEs) and page table entries (PTEs) is supported, 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 is supported, specifically the MCG_CAP register.
15	CMOV	Conditional Move Instruction	The processor supports CMOVcc, and if the FPU feature flag (bit 0) is also set, supports the FCMOVCC and FCOMI instructions.
16	PAT	Page Attribute Table	Indicates whether the processor supports the Page Attribute Table. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory on 4K granularity through a linear address.
17	PSE-36	36-bit Page Size Extension	Indicates whether the processor supports 4-MB pages that are capable of addressing physical memory beyond 4-GB. This feature indicates that the upper four bits of the physical address of the 4-MB page is encoded by bits 13-16 of the page directory entry.
18	PSN	Processor serial number is present and enabled	The processor supports the 96-bit processor serial number feature, and the feature is enabled. Note: The Pentium 4 and subsequent processor families do not support this feature.
19	CLFSH	CLFLUSH Instruction	Indicates that the processor supports the CLFLUSH instruction.
20		Reserved	Do not count on their value.
21	DS	Debug Store	Indicates that 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.
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	MMX technology	The processor supports the MMX technology instruction set extensions to Intel Architecture.
24	FXSR	FXSAVE and FXSTOR Instructions	The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions.
25	SSE	Streaming SIMD Extensions	The processor supports the Streaming SIMD Extensions to the Intel Architecture.

Table 2-4. Feature Flag Values Reported in the EDX Register (Sheet 3 of 3)

Bit	Name	Description when Flag = 1	Comments
26	SSE2	Streaming SIMD Extensions 2	Indicates the processor supports the Streaming SIMD Extensions 2 Instructions.
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	Multi-Threading	The physical processor package is capable of supporting more than one logical processor.
			This field does not indicate that Hyper-Threading Technology or Core Multi-Processing (CMP) has been enabled for this specific processor. To determine if Hyper-Threading Technology or CMP is supported, compare value returned in EBX[23:16] after executing CPUID with EAX=1. If the resulting value is > 1, then the processor supports Multi-Threading.
			IF (CPUID(1).EBX[23:16] > 1)
			{
			Multi-Threading = TRUE
			}
			ELSE
			{
			Multi-Threading = FALSE
			}

29	TM	Thermal Monitor	The processor implements the Thermal Monitor automatic thermal control circuitry (TCC).
30		Reserved	Do not count on their value.
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.

Table 2-5. Feature Flag Values Reported in the ECX Register (Sheet 1 of 2)

Bit	Name	Description when Flag = 1	Comments
0	SSE3	Streaming SIMD Extensions 3	The processor supports the Streaming SIMD Extensions 3 instructions.
1	PCLMULDQ	PCLMULDQ Instruction	The processor supports PCLMULDQ instruction.
2	DTES64	64-Bit Debug Store	Indicates that the processor has the ability to write a history of the 64-bit branch to and from addresses into a memory buffer.
3	MONITOR	MONITOR/MWAIT	The processor supports the MONITOR and MWAIT instructions.
4	DS-CPL	CPL Qualified Debug Store	The processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5	VMX	Virtual Machine Extensions	The processor supports Intel® Virtualization Technology
6	SMX	Safer Mode Extensions	The processor supports Intel® Trusted Execution Technology
7	EST	Enhanced Intel SpeedStep® Technology	The processor supports Enhanced Intel SpeedStep Technology and implements the IA32_PERF_STS and IA32_PERF_CTL registers.
8	TM2	Thermal Monitor 2	The processor implements the Thermal Monitor 2 thermal control circuit (TCC).
9	SSSE3	Supplemental Streaming SIMD Extensions 3	The processor supports the Supplemental Streaming SIMD Extensions 3 instructions.
10	CNXT-ID	L1 Context ID	The L1 data cache mode can be set to either adaptive mode or shared mode by the BIOS.
12:11		Reserved	Do not count on their value.
13	CX16	CMPXCHG16B	This processor supports the CMPXCHG16B instruction.
14	xTPR	xTPR Update Control	The processor supports the ability to disable sending Task Priority messages. When this feature flag is set, Task Priority messages may be disabled. Bit 23 (Echo TPR disable) in the IA32_MISC_ENABLE MSR controls the sending of Task Priority messages.
15	PDCM	Perfmon and Debug Capability	The processor supports the Performance Capabilities MSR. IA32_PERF_CAPABILITIES register is MSR 345h.
17:16		Reserved	Do not count on their value.
18	DCA	Direct Cache Access	The processor supports the ability to prefetch data from a memory mapped device.
19	SSE4.1	Streaming SIMD Extensions 4.1	The processor supports the Streaming SIMD Extensions 4.1 instructions.
20	SSE4.2	Streaming SIMD Extensions 4.2	The processor supports the Streaming SIMD Extensions 4.2 instructions.
21	x2APIC	Extended xAPIC Support	The processor supports x2APIC feature.
22	MOVBE	MOVBE Instruction	The processor supports MOVBE instruction.
23	POPCNT	POPCNT Instruction	The processor supports the POPCNT instruction.
24		Reserved	Do not count on their value.
25	AES	AES Instruction	The processor supports AES instruction.

Table 2-5. Feature Flag Values Reported in the ECX Register (Sheet 2 of 2)

Bit	Name	Description when Flag = 1	Comments
26	XSAVE	XSAVE/XSTOR States	The processor supports the XSAVE/XRSTOR processor extended states feature, the XSETBV/ XGETBV instructions, and the XFEATURE_ENABLED_MASK register (XCR0)
27	OSXSAVE	OS-Enabled Extended State Management	A value of 1 indicates that the OS has enabled XSETBV/XGETBV instructions to access the XFEATURE_ENABLED_MASK register (XCR0), and support for processor extended state management using XSAVE/XRSTOR.
31:28		Reserved	Do not count on their value.

2.1.2.4 SYSENTER/SYSEXIT – SEP Features Bit

The SYSENTER Present (SEP) Feature bit (returned in EDX bit 11 after execution of CPUID Function 1) indicates support for SYSENTER/SYSEXIT instructions. An operating system that detects the presence of the SEP Feature bit must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present:

```
IF (CPUID SEP Feature bit is set, i.e. CPUID (1).EDX[11] == 1)
{
       IF ((Processor Signature & 0x0FFF3FFF) < 0x00000633)
           Fast System Call is NOT supported
       ELSE
           Fast System Call is supported
}
```

The Pentium Pro processor (Model = 1) returns a set SEP CPUID feature bit, but should not be used by software.

2.1.3 Cache Descriptors (Function 02h)

When the EAX register contains a value of 2, the CPUID instruction loads the EAX, EBX, ECX and EDX registers with descriptors that indicate the processor's cache and TLB characteristics. The lower 8 bits of the EAX register (AL) contain a value that identifies the number of times the CPUID must be executed in order to obtain a complete image of the processor's caching systems. For example, the Intel® Core™ i7 processor returns a value of 01h in the lower 8 bits of the EAX register to indicate that the CPUID instruction need only be executed once (with EAX = 2) to obtain a complete image of the processor configuration.

The remainder of the EAX register, the EBX, ECX and EDX registers, contain the cache and Translation Lookaside Buffer (TLB) descriptors. Table 2-6 shows that when bit 31 in a given register is zero, that register contains valid 8-bit descriptors. To decode descriptors, move sequentially from the most significant byte of the register down through the least significant byte of the register. Assuming bit 31 is 0, then that register contains valid cache or TLB descriptors in bits 24 through 31, bits 16 through 23, bits 8 through 15 and bits 0 through 7. Software must compare the value contained in each of the descriptor bit fields with the values found in Table 2-7 to determine the cache and TLB features of a processor.

Table 2-7 lists the current cache and TLB descriptor values and their respective characteristics. This list will be extended in the future as necessary. Between models and steppings of processors the cache and TLB information may change bit field locations, therefore it is important that software not assume fixed locations when parsing the cache and TLB descriptors.

Table 2-6. Descriptor Formats

Register bit 31	Descriptor Type	Description
1	Reserved	Reserved for future use.
0	8-bit descriptors	Descriptors point to a parameter table to identify cache characteristics. The descriptor is null if it has a 0 value.

Table 2-7. Descriptor Decode Values (Sheet 1 of 3)

Value	Cache or TLB Descriptor Description
00h	Null
01h	Instruction TLB: 4-KB Pages, 4-way set associative, 32 entries
02h	Instruction TLB: 4-MB Pages, fully associative, 2 entries
03h	Data TLB: 4-KB Pages, 4-way set associative, 64 entries
04h	Data TLB: 4-MB Pages, 4-way set associative, 8 entries
05h	Data TLB: 4-MB Pages, 4-way set associative, 32 entries
06h	1st-level instruction cache: 8-KB, 4-way set associative, 32-byte line size
08h	1st-level instruction cache: 16-KB, 4-way set associative, 32-byte line size
09h	1st-level Instruction Cache: 32-KB, 4-way set associative, 64-byte line size
0Ah	1st-level data cache: 8-KB, 2-way set associative, 32-byte line size
0Ch	1st-level data cache: 16-KB, 4-way set associative, 32-byte line size
0Dh	1st-level Data Cache: 16-KB, 4-way set associative, 64-byte line size, ECC
21h	256-KB L2 (MLC), 8-way set associative, 64-byte line size
22h	3rd-level cache: 512-KB, 4-way set associative, sectored cache, 64-byte line size
23h	3rd-level cache: 1-MB, 8-way set associative, sectored cache, 64-byte line size
25h	3rd-level cache: 2-MB, 8-way set associative, sectored cache, 64-byte line size
29h	3rd-level cache: 4-MB, 8-way set associative, sectored cache, 64-byte line size
2Ch	1st-level data cache: 32-KB, 8-way set associative, 64-byte line size
30h	1st-level instruction cache: 32-KB, 8-way set associative, 64-byte line size
39h	2nd-level cache: 128-KB, 4-way set associative, sectored cache, 64-byte line size
3Ah	2nd-level cache: 192-KB, 6-way set associative, sectored cache, 64-byte line size
3Bh	2nd-level cache: 128-KB, 2-way set associative, sectored cache, 64-byte line size
3Ch	2nd-level cache: 256-KB, 4-way set associative, sectored cache, 64-byte line size
3Dh	2nd-level cache: 384-KB, 6-way set associative, sectored cache, 64-byte line size
3Eh	2nd-level cache: 512-KB, 4-way set associative, sectored cache, 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-KB, 4-way set associative, 32-byte line size
42h	2nd-level cache: 256-KB, 4-way set associative, 32-byte line size
43h	2nd-level cache: 512-KB, 4-way set associative, 32-byte line size

Table 2-7. Descriptor Decode Values (Sheet 2 of 3)

Value	Cache or TLB Descriptor Description
44h	2nd-level cache: 1-MB, 4-way set associative, 32-byte line size
45h	2nd-level cache: 2-MB, 4-way set associative, 32-byte line size
46h	3rd-level cache: 4-MB, 4-way set associative, 64-byte line size
47h	3rd-level cache: 8-MB, 8-way set associative, 64-byte line size
48h	2nd-level cache: 3-MB, 12-way set associative, 64-byte line size, unified on-die
49h	3rd-level cache: 4-MB, 16-way set associative, 64-byte line size (Intel Xeon processor MP, Family 0Fh, Model 06h) 2nd-level cache: 4-MB, 16-way set associative, 64-byte line size
4Ah	3rd-level cache: 6-MB, 12-way set associative, 64-byte line size
4Bh	3rd-level cache: 8-MB, 16-way set associative, 64-byte line size
4Ch	3rd-level cache: 12-MB, 12-way set associative, 64-byte line size
4Dh	3rd-level cache: 16-MB, 16-way set associative, 64-byte line size
4Eh	2nd-level cache: 6-MB, 24-way set associative, 64-byte line size
50h	Instruction TLB: 4-KB, 2-MB or 4-MB pages, fully associative, 64 entries
51h	Instruction TLB: 4-KB, 2-MB or 4-MB pages, fully associative, 128 entries
52h	Instruction TLB: 4-KB, 2-MB or 4-MB pages, fully associative, 256 entries
55h	Instruction TLB: 2-MB or 4-MB pages, fully associative, 7 entries
56h	L1 Data TLB: 4-MB pages, 4-way set associative, 16 entries
57h	L1 Data TLB: 4-KB pages, 4-way set associative, 16 entries
5Ah	Data TLB0: 2-MB or 4-MB pages, 4-way associative, 32 entries
5Bh	Data TLB: 4-KB or 4-MB pages, fully associative, 64 entries
5Ch	Data TLB: 4-KB or 4-MB pages, fully associative, 128 entries
5Dh	Data TLB: 4-KB or 4-MB pages, fully associative, 256 entries
60h	1st-level data cache: 16-KB, 8-way set associative, sectored cache, 64-byte line size
66h	1st-level data cache: 8-KB, 4-way set associative, sectored cache, 64-byte line size
67h	1st-level data cache: 16-KB, 4-way set associative, sectored cache, 64-byte line size
68h	1st-level data cache: 32-KB, 4 way set associative, sectored cache, 64-byte line size
70h	Trace cache: 12K-uops, 8-way set associative
71h	Trace cache: 16K-uops, 8-way set associative
72h	Trace cache: 32K-uops, 8-way set associative
73h	Trace cache: 64K-uops, 8-way set associative
78h	2nd-level cache: 1-MB, 4-way set associative, 64-byte line size
79h	2nd-level cache: 128-KB, 8-way set associative, sectored cache, 64-byte line size
7Ah	2nd-level cache: 256-KB, 8-way set associative, sectored cache, 64-byte line size
7Bh	2nd-level cache: 512-KB, 8-way set associative, sectored cache, 64-byte line size
7Ch	2nd-level cache: 1-MB, 8-way set associative, sectored cache, 64-byte line size
7Dh	2nd-level cache: 2-MB, 8-way set associative, 64-byte line size
7Fh	2nd-level cache: 512-KB, 2-way set associative, 64-byte line size
82h	2nd-level cache: 256-KB, 8-way set associative, 32-byte line size
83h	2nd-level cache: 512-KB, 8-way set associative, 32-byte line size
84h	2nd-level cache: 1-MB, 8-way set associative, 32-byte line size
85h	2nd-level cache: 2-MB, 8-way set associative, 32-byte line size

Table 2-7. Descriptor Decode Values (Sheet 3 of 3)

Value	Cache or TLB Descriptor Description
86h	2nd-level cache: 512-KB, 4-way set associative, 64-byte line size
87h	2nd-level cache: 1-MB, 8-way set associative, 64-byte line size
B0h	Instruction TLB: 4-KB Pages, 4-way set associative, 128 entries
B1h	Instruction TLB: 2-MB pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries
B2h	Instruction TLB: 4-KB pages, 4-way set associative, 64 entries
B3h	Data TLB: 4-KB Pages, 4-way set associative, 128 entries
B4h	Data TLB: 4-KB Pages, 4-way set associative, 256 entries
CAh	Shared 2nd-level TLB: 4 KB pages, 4-way set associative, 512 entries
D0h	512KB L3 Cache, 4-way set associative, 64-byte line size
D1h	1-MB L3 Cache, 4-way set associative, 64-byte line size
D2h	2-MB L3 Cache, 4-way set associative, 64-byte line size
D6h	1-MB L3 Cache, 8-way set associative, 64-byte line size
D7h	2-MB L3 Cache, 8-way set associative, 64-byte line size
D8h	4-MB L3 Cache, 8-way set associative, 64-byte line size
DCh	1.5-MB L3 Cache, 12-way set associative, 64-byte line size
DDh	3-MB L3 Cache, 12-way set associative, 64-byte line size
DEh	6-MB L3 Cache, 12-way set associative, 64-byte line size
E2h	2-MB L3 Cache, 16-way set associative, 64-byte line size
E3h	4-MB L3 Cache, 16-way set associative, 64-byte line size
E4h	8-MB L3 Cache, 16-way set associative, 64-byte line size
EAh	12-MB L3 Cache, 24-way set associative, 64-byte line size
EBh	18-MB L3 Cache, 24-way set associative, 64-byte line size
ECh	24-MB L3 Cache, 24-way set associative, 64-byte line size
F0h	64-byte Prefetching
F1h	128-byte Prefetching

2.1.3.1 Intel® Core™ i7 Processor, Model 1Ah Output Example

The Core i7 processor, model 1Ah returns the values shown in Table 2-8. Since the value of AL=1, it is valid to interpret the remainder of the registers. Table 2-8 also shows the MSB (bit 31) of all the registers are 0 which indicates that each register contains valid 8-bit descriptor.

Table 2-8. Intel® Core™ i7 Processor, Model 1Ah with 8-MB L3 Cache CPUID (EAX=2)

	31	23	15	7 0
EAX	55h	03h	5Ah	01h
EBX	00h	F0h	B2h	E4h
ECX	00h	00h	00h	00h
EDX	09h	CAh	21h	2Ch

The register values in Table 2-8 show that this Core i7 processor has the following cache and TLB characteristics:

- (55h) Instruction TLB: 2-MB or 4-MB pages, fully associative, 7 entries
- (03h) Data TLB: 4-KB Pages, 4-way set associative, 64 entries

- (5Ah) Data TLB0: 2-MB or 4-MB pages, 4-way associative, 32 entries
- (01h) Instruction TLB: 4-KB Pages, 4-way set associative, 32 entries
- (F0h) 64-byte Prefetching
- (B2h) Instruction TLB: 4-KB pages, 4-way set associative, 64 entries
- (E4h) 8-MB L3 Cache, 16-way set associative, 64-byte line size
- (09h) 1st-level Instruction Cache: 32-KB, 4-way set associative, 64-byte line size
- (CAh) Shared 2nd-level TLB: 4-KB pages, 4-way set associative, 512 entries
- (21h) 256KB L2 (MLC), 8-way set associative, 64-byte line size
- (2Ch) 1st-level data cache: 32-KB, 8-way set associative, 64-byte line size

2.1.4 Processor Serial Number (Function 03h)

Processor serial number (PSN) is available in Pentium III processor only. The value in this register is reserved 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. Refer to Section 3 for more details.

2.1.5 Deterministic Cache Parameters (Function 04h)

When EAX is initialized to a value of 4, the CPUID instruction returns deterministic cache information in the EAX, EBX, ECX and EDX registers. This function requires ECX be initialized with an index which indicates which cache to return information about. The OS is expected to call this function (CPUID.4) with ECX = 0, 1, 2, until EAX[4:0] == 0, indicating no more caches. The order in which the caches are returned is not specified and may change at Intel's discretion.

Note: The BIOS will use this function to determine the number of APIC IDs reserved for a specific physical processor package. To do this the BIOS must initially set the EAX register to 4 and the ECX register to 0 prior to executing the CPUID instruction. EAX[31:16] will contain the value.

Table 2-9. Deterministic Cache Parameters (Sheet 1 of 2)

Register Bits	Description
EAX[31:26]	Number of APIC IDs reserved for the package; this value is not the number of threads implemented for the package. Encoded with a "plus 1" encoding. Add one to the value in this register field.
EAX[25:14]	Maximum number of threads sharing this cache. Encoded with a "plus 1" encoding. Add one to the value in this register field.
EAX[13:10]	**Reserved**.
EAX[9]	Fully Associative Cache
EAX[8]	Self Initializing cache level
EAX[7:5]	Cache Level (Starts at 1)
EAX[4:0]	Cache Type 0 = Null, no more caches 1 = Data Cache 2 = Instruction Cache 3 = Unified Cache 4-31 = Reserved
EBX[31:22]	Ways of Associativity Encoded with a "plus 1" encoding. Add one to the value in this register field.

Table 2-9. Deterministic Cache Parameters (Sheet 2 of 2)

Register Bits	Description
EBX[21:12]	Physical Line partitions Encoded with a “plus 1” encoding. Add one to the value in this register field.
EBX[11:0]	System Coherency Line Size Encoded with a “plus 1” encoding. Add one to the value in this register field.
ECX[31:0]	Number of Sets Encoded with a “plus 1” encoding. Add one to the value in this register field.
EDX[31:2]	**Reserved**
EDX[1]	Cache is inclusive to lower cache levels. A value of ‘0’ means that WBINVD/INVD from any thread sharing this cache acts upon all lower caches for threads sharing this cache. A value of ‘1’ means that WBINVD/ INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache.
EDX[0]	WBINVD/INVD behavior on lower level caches. A value of ‘0’ means WBINVD/INVD from any thread sharing this cache acts upon all lower level caches for threads sharing this cache; A value of ‘1’ means WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads.

Equation 2-4.Calculating the Cache Size

```
Cache Size in Bytes    = (Ways +1) × (Partitions +1) × (Line Size +1) × (Sets +1)
                       = (EBX[31:22] +1) × (EBX[21:12] +1) × (EBX[11:0] +1 × (ECX + 1)
```

2.1.5.1 Cache Sharing Among Cores and Threads

The multi-core and threads fields give information about cache sharing. By comparing the following three numbers:

1. Number of logical processors per physical processor package (CPUID.1.EBX[23:16])
2. Number of APIC IDs reserved per package (CPUID.4.EAX[31:26] + 1)
3. Total number of threads serviced by this cache (CPUID.4.EAX[25:14] + 1)

Software can determine whether this cache is shared between cores, or specific to one core, or even specific to one thread or a subset of threads. This feature is very important with regard to logical processors since it is a means of differentiating a Hyper-Threading technology processor from a multi-core processor or a multi-core processor with Hyper-Threading Technology. Note that the sharing information was not available using the cache descriptors returned by CPUID function 2. Refer to section 7.10.3 of the *Intel@ 64 and IA-32 Software Developer’s Manual, Volume 3A: System Programming Guide*.

2.1.6 MONITOR / MWAIT Parameters (Function 05h)

When EAX is initialized to a value of 5, the CPUID instruction returns MONITOR / MWAIT parameters in the EAX, EBX, ECX and EDX registers if the MONITOR and MWAIT instructions are supported by the processor.

Table 2-10. MONITOR / MWAIT Parameters (Sheet 1 of 2)

Register Bits	Description
EAX[31:16]	**Reserved.**
EAX[15:0]	Smallest monitor line size in bytes.
EBX[31:16]	**Reserved.**

Table 2-10. MONITOR / MWAIT Parameters (Sheet 2 of 2)

Register Bits	Description
EBX[15:0]	Largest monitor line size in bytes.
ECX[31:2]	**Reserved**
ECX[1]	Support for treating interrupts as break-events for MWAIT.
ECX[0]	MONITOR / MWAIT Extensions supported
EDX[31:20]	**Reserved**
EDX[19:16]	Number of C7 sub-states supported using MONITOR / MWAIT* Number of C4 sub-states supported using MONITOR / MWAIT†
EDX[15:12]	Number of C6 sub-states supported using MONITOR / MWAIT‡ Number of C3 sub-states supported using MONITOR / MWAIT§
EDX[11:8]	Number of C2 sub-states supported using MONITOR / MWAIT**
EDX[7:4]	Number of C1 sub-states supported using MONITOR / MWAIT
EDX[3:0]	Number of C0 sub-states supported using MONITOR / MWAIT

Notes:
* EDX[19:16] C7 sub-states supported on Core i7 and subsequent processors
† EDX[19:16] C4 sub-states supported on processors prior to the Core i7 generation
‡ EDX[15:12] C6 sub-states supported on Core i7 and subsequent processors
§ EDX[15:12] C3 sub-states supported on Core i7 and prior generation processors
** EDX[11:8] C2 sub-states supported on processors prior to the Core i7 generation

2.1.7 Digital Thermal Sensor and Power Management Parameters (Function 06h)

When EAX is initialized to a value of 6, the CPUID instruction returns Digital Thermal Sensor and Power Management parameters in the EAX, EBX, ECX and EDX registers.

Table 2-11. Digital Sensor and Power Management Parameters

Register Bits	Description
EAX[31:2]	**Reserved**
EAX[1]	Intel® Turbo Boost Technology
EAX[0]	Digital Thermal Sensor Capability
EBX[31:4]	**Reserved**
EBX[3:0]	Number of Interrupt Thresholds
EBX[31:1]	**Reserved**
ECX[0]	Hardware Coordination Feedback Capability (Presence of IA32_APERF, IA32_MPERF MSRs)
EDX[31:0]	**Reserved**

2.1.8 Reserved (Function 07h)

This function is reserved.

2.1.9 Reserved (Function 08h)

This function is reserved.

2.1.10 Direct Cache Access (DCA) Parameters (Function 09h)

When EAX is initialized to a value of 9, the CPUID instruction returns DCA information in the EAX, EBX, ECX and EDX registers.

Table 2-12. DCA Parameters

Register Bits	Description
EAX[31:0]	Value of PLATFORM_DCA_CAP MSR Bits [31:0] (Offset 1F8h)
EBX[31:0]	**Reserved**
ECX[31:0]	**Reserved**
EDX[31:0]	**Reserved**

2.1.11 Architectural Performance Monitor Features (Function 0Ah)

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 is greater than Pn 0. See Table 2-13 below.

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, “Debugging and Performance Monitoring,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.*

Table 2-13. Performance Monitor Features

Register Bits	Description
EAX[31:24]	Number of arch events supported per logical processor
EAX[23:16]	Number of bits per programmable counter (width)
EAX[15:8]	Number of counters per logical processor
EAX[7:0]	Architectural PerfMon Version
EBX[31:7]	**Reserved**
EBX[6]	Branch Mispredicts Retired 0 = supported
EBX[5]	Branch Instructions Retired 0 = supported
EBX[4]	Last Level Cache Misses 0 = supported
EBX[3]	Last Level Cache References 0 = supported
EBX[2]	EBX[2] Reference Cycles 0 = supported
EBX[1]	Instructions Retired 0 = supported
EBX[0]	Core Cycles 0 = supported
ECX[31:0]	**Reserved**
EDX[31:13]	**Reserved**
EDX[12:5]	Number of Bits in the Fixed Counters (width)
EDX[4:0]	Number of Fixed Counters

2.1.12 x2APIC Features / Processor Topology (Function 0Bh)

When EAX is initialized to a value of 0Bh, the CPUID instruction returns core/logical processor topology information in EAX, EBX, ECX, and EDX registers. This function requires ECX be initialized with an index which indicates which core or logical processor level to return information about. The BIOS or OS is expected to call this function

(CPUID.EAX=0Bh) with ECX = 0 (Thread), 1 (Core), 2..n (Package), until EAX=0 and EBX=0, indicating no more levels. The order in which the processor topology levels are returned is specific since each level reports some cumulative data and thus some information is dependent on information retrieved from a previous level.

Table 2-14. Core / Logical Processor Topology Overview

Register Bits	Description
EAX[31:5]	**Reserved**
EAX[4:0]	Number of bits to shift right APIC ID to get next level APIC ID. Note: All logical processors with same topology ID map to same core or package at this level.
EBX[31:16]	**Reserved**
EBX[15:0]	Number of factory-configured logical processors at this level. This value does NOT change based on Intel HT Technology disable and core disables.
ECX[31:16]	**Reserved**
ECX[15:8]	Level Type (0=Invalid, 1=Thread, 2=Core)
ECX[7:0]	Level Number (same as ECX input)
EDX[31:0]	Extended APIC ID -- Lower 8 bits identical to the legacy APIC ID

BIOS is expected to iterate through the core / logical processor hierarchy using CPUID Function Bh with ECX using input values from 0-N. In turn, the CPUID function Bh provides topology information in terms of levels. Level 0 is lowest level (reserved for thread), level 1 is core, and the last level is package. All logical processors with same topology ID map to same core/package at this level.

BIOS enumeration occurs via iterative CPUID calls with input of level numbers in ECX starting from 0 and incrementing by 1. BIOS should continue until EAX = EBX = 0 returned indicating no more levels are available (refer to Table 2-17). CPUID Function Bh with ECX=0 provides topology information for the thread level (refer to Table 2-15). And CPUID Function Bh with ECX=1 provides topology information for the Core level (refer to Table 2-16). Note that at each level, all logical processors with same topology ID map to same core or package which is specified for that level.

Table 2-15. Thread Level Processor Topology (CPUID Function 0Bh with ECX=0)

Register Bits	Description	Value with ECX=0 as Input
EAX[31:5]	**Reserved**	
EAX[4:0]	Number of bits to shift right APIC ID to get next level APIC ID. Note: All logical processors with same topology ID map to same core at this level.	1
EBX[31:16]	**Reserved**	
EBX[15:0]	Number of factory-configured logical processors at this level. This value does NOT change based on Intel HT Technology disable and core disables.	1-2 (see Note 1)
ECX[31:16]	**Reserved**	
ECX[15:8]	Level Type (0=Invalid, 1=Thread, 2=Core)	1
ECX[7:0]	Level Number (same as ECX input)	0
EDX[31:0]	Extended APIC ID -- Lower 8 bits identical to the legacy APIC ID	Varies

Note:

1. One logical processor per core if Intel HT Technology is factory-configured as disabled and two logical processors per core if Intel HT Technology is factory-configured as enabled.

Table 2-16. Core Level Processor Topology (CPUID Function 0Bh with ECX=1)

Register Bits	Description	Value with ECX=1 as Input
EAX[31:5]	**Reserved**	
EAX[4:0]	Number of bits to shift right APIC ID to get next level APIC ID. Note: All logical processors with same topology ID map to same package at this level.	4 5 – Nehalem-EX
EBX[31:16]	**Reserved**	
EBX[15:0]	Number of factory-configured logical processors at this level. This value does NOT change based on Intel HT Technology disable and core disables.	1-8 (see Note 1)
ECX[31:16]	**Reserved**	
ECX[15:8]	Level Type (0=Invalid, 1=Thread, 2=Core)	2
ECX[7:0]	Level Number (same as ECX input)	1
EDX[31:0]	Extended APIC ID -- Lower 8 bits identical to the legacy APIC ID	Varies

Note:
1. One logical processor per core if Intel HT Technology is factory-configured as disabled and two logical processors per core if Intel HT Technology is factory-configured as enabled.

Table 2-17. Core Level Processor Topology (CPUID Function 0Bh with ECX>=2)

Register Bits	Description	Value with ECX>=2 as Input
EAX[31:5]	**Reserved**	
EAX[4:0]	Number of bits to shift right APIC ID to get next level APIC ID. Note: All logical processors with same topology ID map to same package at this level.	0 (see Note 1)
EBX[31:16]	**Reserved**	
EBX[15:0]	Number of factory-configured logical processors at this level. This value does NOT change based on Intel HT Technology disable and core disables.	0 (see Note 1)
ECX[31:16]	**Reserved**	
ECX[15:8]	Level Type (0=Invalid, 1=Thread, 2=Core)	0
ECX[7:0]	Level Number (same as ECX input)	Varies (same as ECX input value)
EDX[31:0]	Extended APIC ID -- Lower 8 bits identical to the legacy APIC ID	Varies

Note:
1. One logical processor per core if Intel HT Technology is factory-configured as disabled and two logical processors per core if Intel HT Technology is factory-configured as enabled.

2.1.13 Reserved (Function 0Ch)

This function is reserved.

2.1.14 XSAVE Features (Function 0Dh)

When EAX is initialized to a value of 0Dh and ECX is initialized to a value of 0 (EAX=0Dh AND ECX =0h), the CPUID instruction returns the Processor Extended State Enumeration in the EAX, EBX, ECX and EDX registers.

Note: An initial value greater than ‘0’ in ECX is invalid, therefore, EAX/EBX/ECX/EDX return 0.

Table 2-18. Processor Extended State Enumeration

Register Bits	Description
EAX[31:0]	Reports the valid bit fields of the lower 32 bits of the XFEATURE_ENABLED_MASK register (XCR0). If a bit is 0, the corresponding bit field in XCR0 is reserved.
EBX[31:0]	Maximum size (bytes) required by enabled features in XFEATURE_ENABLED_MASK (XCR0). May be different than ECX when features at the end of the save area are not enabled.
ECX[31:0]	Maximum size (bytes) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e all the valid bit fields in XFEATURE_ENABLED_MASK. This includes the size needed for the XSAVE.HEADER.
EDX[31:0]	Reports the valid bit fields of the upper 32 bits of the XFEATURE_ENABLED_MASK register (XCR0). If a bit is 0, the corresponding bit field in XCR0 is reserved.

2.2 Extended CPUID Functions

2.2.1 Largest Extended Function # (Function 8000_0000h)

When EAX is initialized to a value of 8000_0000h, the CPUID instruction returns the largest extended function number supported by the processor in register EAX.

Table 2-19. Largest Extended Function

	31 23 15 70
EAX[31:0]	Largest extended function number supported
EBX[31:0]	**Reserved**
EDX[31:0]	**Reserved**
ECX[31:0]	**Reserved**

2.2.2 Extended Feature Bits (Function 8000_0001h)

When the EAX register contains a value of 80000001h, the CPUID instruction loads the EDX register with the extended feature flags. The feature flags (when a Flag = 1) indicate what extended features the processor supports. Table 2-4 lists the currently defined extended feature flag values.

For future processors, refer to the programmer's reference manual, user's manual, or the appropriate documentation for the latest extended feature flag values.

Note: By using CPUID feature flags to determine processor features, software can detect and avoid incompatibilities introduced by the addition or removal of processor features.

Figure 2-4. Extended Feature Flag Values Reported in the EDX Register (Sheet 1 of 2)

Bit	Name	Description when Flag = 1	Comments
10:0		Reserved	Do not count on their value.
11	SYSCALL	SYSCALL/SYSRET	The processor supports the SYSCALL and SYSRET instructions.
19:12		Reserved	Do not count on their value.
20	XD Bit	Execution Disable Bit	The processor supports the XD Bit when PAE mode paging is enabled.

Figure 2-4. Extended Feature Flag Values Reported in the EDX Register (Sheet 2 of 2)

Bit	Name	Description when Flag = 1	Comments
28:21		Reserved	Do not count on their value.
29	Intel® 64	Intel® 64 Instruction Set Architecture	The processor supports Intel® 64 Architecture extensions to the IA-32 Architecture. For additional information refer to http://developer.intel.com/technology/architecture-silicon/intel64/index.htm
31:30		Reserved	Do not count on their value.

Table 2-20. Extended Feature Flag Values Reported in the ECX Register

Bit	Name	Description when Flag = 1	Comments
0	LAHF	LAHF / SAHF	A value of 1 indicates the LAHF and SAHF instructions are available when the IA-32e mode is enabled and the processor is operating in the 64-bit sub-mode.
31:1		Reserved	Do not count on their value

2.2.3 Processor Name / Brand String (Function 8000_0002h, 8000_0003h, 8000_0004h)

Functions 8000_0002h, 8000_0003h, and 8000_0004h each return up to 16 ASCII bytes of the processor name in the EAX, EBX, ECX, and EDX registers. The processor name is constructed by concatenating each 16-byte ASCII string returned by the three functions. The processor name is right justified with leading space characters. It is returned in little-endian format and NULL terminated. The processor name can be a maximum of 48 bytes including the NULL terminator character. In addition to the processor name, these functions return the maximum supported speed of the processor in ASCII.

2.2.3.1 Building the Processor Name

BIOS must reserve enough space in a byte array to concatenate the three 16 byte ASCII strings that comprise the processor name. BIOS must execute each function in sequence. After sequentially executing each CPUID Brand String function, BIOS must concatenate EAX, EBX, ECX, and EDX to create the resulting Processor Brand String.

Example 2-1. Building the Processor Brand String

```
Processor_Name   DB 48 dup(0)

       MOV    EAX, 80000000h
       CPUID
       CMP    EAX, 80000004h            ; Check if extended
                                        ; functions are
                                        ; supported
       JB Not_Supported

       MOV    EAX, 80000002h
       MOV    DI, OFFSET Processor_Name
       CPUID                            ; Get the first 16
                                        ; bytes of the
                                        ; processor name

       CALL   Save_String
       MOV    EAX, 80000003h
       CPUID                            ; Get the second 16
                                        ; bytes of the
                                        ; processor name

       CALL   Save_String
       MOV    EAX, 80000004h
       CPUID                            ; Get the last 16
                                        ; bytes of the
                                        ; processor name

       CALL   Save_String


Not_Supported
       RET

Save_String:
       MOV    Dword Ptr[DI],EAX
       MOV    Dword Ptr [DI+4],EBX
       MOV    Dword Ptr [DI+8],ECX
       MOV    Dword Ptr [DI+12],EDX
       ADD    DI, 16
       RET
```

2.2.3.2 Displaying the Processor Brand String

The processor name may be a right justified string padded with leading space characters. When displaying the processor name string, the display software must skip the leading space characters and discontinue printing characters when the NULL character is encountered.

Example 2-2. Displaying the Processor Brand String

```
        CLD
        MOV    SI, OFFSET Processor_Name
                                        ; Point SI to the
                                        ; name string
Spaces:

        LODSB
        CMP    AL, ' '                  ; Skip leading space chars
        JE Spaces
        CMP    AL, 0                    ; Exit if NULL byte
                                        ; encounterd
        JE Done

Display_Char:

        CALL Display_Character          ; Put a char on the
                                        ; output device
        LODSB
        CMP    AL, 0                    ; Exit if NULL byte
                                        ; encountered
        JNE Display_Char

Done:
```

2.2.4 Reserved (Function 8000_0005h)

This function is reserved.

2.2.5 Extended L2 Cache Features (Function 8000_0006h)

Functions 8000_0006h returns details of the L2 cache in the ECX register. The details returned are the line size, associativity, and the cache size described in 1024-byte units (see Table 2-5).

Figure 2-5. L2 Cache Details

Register Bits	Description
EAX[31:0]	**Reserved**
EBX[31:0]	**Reserved**
ECX[31:16]	L2 Cache size described in 1024-byte units.
ECX[15:12]	L2 Cache Associativity Encodings 00h Disabled 01h Direct mapped 02h 2-Way 04h 4-Way 06h 8-Way 08h 16-Way 0Fh Fully associative
ECX[11:8]	**Reserved**
ECX[7:0]	L2 Cache Line Size in bytes.
EDX[31:0]	**Reserved**

2.2.6 Advanced Power Management (Function 8000_0007h)

In the Core i7 and future processor generations, the TSC will continue to run in the deepest C-states. Therefore, the TSC will run at a constant rate in all ACPI P-, C-. and T-states. Support for this feature is indicated by CPUID.0x8000_0007.EDX[8]. On processors with invariant TSC support, the OS may use the TSC for wall clock timer services (instead of ACPI or HPET timers). TSC reads are much more efficient and do not incur the overhead associated with a ring transition or access to a platform resource.

Table 2-21. Power Management Details

Register Bits	Description
EAX[31:0]	**Reserved**
EBX[31:0]	**Reserved**
ECX[31:0]	**Reserved**
EDX[31:9]	**Reserved**
EDX[8]	TSC Invariance (1 = Available, 0 = Not available) TSC will run at a constant rate in all ACPI P-states, C-states and T-states.
EDX[7:0]	**Reserved**

2.2.7 Virtual and Physical address Sizes (Function 8000_0008h)

On the Core Solo, Core Duo, Core2 Duo processor families, when EAX is initialized to a value of 8000_0008h, the CPUID instruction will return the supported virtual and physical address sizes in EAX. Values in other general registers are reserved. This information is useful for BIOS to determine processor support for Intel® 64 Instruction Set Architecture (Intel® 64).

If this function is supported, the Physical Address Size returned in EAX[7:0] should be used to determine the number of bits to configure MTRRn_PhysMask values with. Software must determine the MTRR PhysMask for each execution thread based on this function and not assume all execution threads in a platform have the same number of physical address bits.

Table 2-22. Virtual and Physical Address Size Definitions

Register Bits	Description
EAX[31:16]	**Reserved**
EAX[15:8]	Virtual Address Size: Number of address bits supported by the processor for a virtual address.
EAX[7:0]	Physical Address Size: Number of address bits supported by the processor for a physical address.
EBX[31:0]	**Reserved**
ECX[31:0]	**Reserved**
EDX[31:0]	**Reserved**

§

3 Processor Serial Number

The processor serial number extends the concept of processor identification. Processor serial number is a 96-bit number accessible through the CPUID instruction. Processor serial number can be used by applications to identify a processor, and by extension, its system.

The processor serial number creates a software accessible identity for an individual processor. The processor serial number, combined with other qualifiers, could be applied to user identification. Applications include membership authentication, data backup/ restore protection, removable storage data protection, managed access to files, or to confirm document exchange between appropriate users.

Processor serial number is another tool for use in asset management, product tracking, remote systems load and configuration, or to aid in boot-up configuration. In the case of system service, processor serial number could be used to differentiate users during help desk access, or track error reporting. Processor serial number provides an identifier for the processor, but should not be assumed to be unique in itself. There are potential modes in which erroneous processor serial numbers may be reported. For example, in the event a processor is operated outside its recommended operating specifications, (e.g., voltage, frequency, etc.) the processor serial number may not be correctly read from the processor. Improper BIOS or software operations could yield an inaccurate processor serial number. These events could lead to possible erroneous or duplicate processor serial numbers being reported. System manufacturers can strengthen the robustness of the feature by including redundancy features, or other fault tolerant methods.

Processor serial number used as a qualifier for another independent number could be used to create an electrically accessible number that is likely to be distinct. Processor serial number is one building block useful for the purpose of enabling the trusted, connected PC.

3.1 Presence of Processor Serial Number

To determine if the processor serial number feature is supported, the program should set the EAX register parameter value to “1” and then execute the CPUID instruction as follows:

```
MOV EAX, 01H
CPUID
```

After execution of the CPUID instruction, the ECX and EDX register contains the Feature Flags. If the PSN Feature Flags, (EDX register, bit 18) equals “1”, the processor serial number feature is supported, and enabled. **If the PSN Feature Flags equals “0”, the processor serial number feature is either not supported, or disabled in a Pentium III processor.**

3.2 Forming the 96-bit Processor Serial Number

The 96-bit processor serial number is the concatenation of three 32-bit entities.

To access the most significant 32-bits of the processor serial number the program should set the EAX register parameter value to “1” and then execute the CPUID instruction as follows:

```
MOV EAX, 01H
CPUID
```

After execution of the CPUID instruction, the EAX register contains the Processor Signature. The Processor Signature comprises the most significant 32-bits of the processor serial number. The value in EAX should be saved prior to gathering the remaining 64-bits of the processor serial number.

To access the remaining 64-bits of the processor serial number the program should set the EAX register parameter value to “3” and then execute the CPUID instruction as follows:

```
MOV EAX, 03H
CPUID
```

After execution of the CPUID instruction, the EDX register contains the middle 32-bits, and the ECX register contains the least significant 32-bits of the processor serial number. Software may then concatenate the saved Processor Signature, EDX, and ECX before returning the complete 96-bit processor serial number.

Processor serial number should be displayed as 6 groups of 4 hex nibbles (e.g. XXXX-XXXX-XXXX-XXXX-XXXX-XXXX where X represents a hex digit). Alpha hex characters should be displayed as capital letters.

§

4 Brand ID and Brand String

4.1 Brand ID

Beginning with the Pentium III processors, model 8, the Pentium III Xeon processors, model 8, and Celeron processor, model 8, the concept of processor identification is further extended with the addition of Brand ID. Brand ID is an 8-bit number accessible through the CPUID instruction. Brand ID may be used by applications to assist in identifying the processor.

Processors that implement the Brand ID feature return the Brand ID in bits 7 through 0 of the EBX register when the CPUID instruction is executed with EAX=1 (see Table 4-1). Processors that do not support the feature return a value of 0 in EBX bits 7 through 0.

To differentiate previous models of the Pentium II processor, Pentium II Xeon processor, Celeron processor, Pentium III processor and Pentium III Xeon processor, application software relied on the L2 cache descriptors. In certain cases, the results were ambiguous. For example, software could not accurately differentiate a Pentium II processor from a Pentium II Xeon processor with a 512-KB L2 cache. Brand ID eliminates this ambiguity by providing a software-accessible value unique to each processor brand. Table 4-1 shows the values defined for each processor.

Table 4-1. Brand ID (EAX=1) Return Values in EBX (Bits 7 through 9) (Sheet 1 of 2)

Value	Description
00h	**Unsupported**
01h	Intel® Celeron® processor
02h	Intel® Pentium® III processor
03h	Intel® Pentium® III Xeon® processor If processor signature = 000006B1h, then “Intel® Celeron® processor”
04h	Intel® Pentium® III processor
06h	Mobile Intel® Pentium® III Processor-M
07h	Mobile Intel® Celeron® processor
08h	Intel® Pentium® 4 processor If processor signature is >=00000F13h, then “Intel® Genuine processor”
09h	Intel® Pentium® 4 processor
0Ah	Intel® Celeron® Processor
0Bh	Intel® Xeon® processor If processor signature is <00000F13h, then “Intel® Xeon® processor MP”
0Ch	Intel® Xeon® processor MP
0Eh	Mobile Intel® Pentium® 4 processor–M If processor signature is <00000F13h, then “Intel® Xeon® processor”
0Fh	Mobile Intel® Celeron® processor
11h	Mobile Genuine Intel® processor
12h	Intel® Celeron® M processor
13h	Mobile Intel® Celeron® processor
14h	Intel® Celeron® Processor
15h	Mobile Genuine Intel® processor

Table 4-1. Brand ID (EAX=1) Return Values in EBX (Bits 7 through 9) (Sheet 2 of 2)

Value	Description
16h	Intel® Pentium® M processor
17h	Mobile Intel® Celeron® processor
All other values	**Reserved**

4.2 Brand String

The Brand string is an extension to the CPUID instruction implemented in some Intel IA-32 processors, including the Pentium 4 processor. Using the brand string feature, future IA-32 architecture based processors will return their ASCII brand identification string and maximum operating frequency via an extended CPUID instruction. Note that the frequency returned is the maximum operating frequency that the processor has been qualified for and not the current operating frequency of the processor.

When CPUID is executed with EAX set to the values listed in Table 4-2, the processor will return an ASCII brand string in the general-purpose registers as detailed in this document.

The brand/frequency string is defined to be 48 characters long, 47 bytes will contain characters and the 48th byte is defined to be NULL (0). A processor may return less than the 47 ASCII characters as long as the string is null terminated and the processor returns valid data when CPUID is executed with EAX = 80000002h, 80000003h and 80000004h.

To determine if the brand string is supported on a processor, software must follow the steps below:

1. Execute the CPUID instruction with EAX=80000000h
2. If ((returned value in EAX) > 80000000h) then the processor supports the extended CPUID functions and EAX contains the largest extended function supported.
3. The processor brand string feature is supported if EAX >= 80000004h

Table 4-2. Processor Brand String Feature

EAX Input Value	Function	Return Value
80000000h	Largest Extended Function Supported	EAX=Largest supported extended function number, EBX = ECX = EDX = Reserved
80000001h	Extended Processor Signature and Extended Feature Bits	EDX and ECX contain Extended Feature Flags EAX = EBX = Reserved
80000002h	Processor Brand String	EAX, EBX, ECX, EDX contain ASCII brand string
80000003h	Processor Brand String	EAX, EBX, ECX, EDX contain ASCII brand string
80000004h	Processor Brand String	EAX, EBX, ECX, EDX contain ASCII brand string

§

5 Usage Guidelines

This document presents Intel-recommended feature-detection methods. Software should not try to identify features by exploiting programming tricks, undocumented features, or otherwise deviating from the guidelines presented in this application note.

The following guidelines are intended to help programmers maintain the widest range of compatibility for their software.

- Do not depend on the absence of an invalid opcode trap on the CPUID opcode to detect the CPUID instruction. Do not depend on the absence of an invalid opcode trap on the PUSHFD opcode to detect a 32-bit processor. Test the ID flag, as described in Section 2 and shown in Section 7.
- Do not assume that a given family or model has any specific feature. For example, do not assume the family value 5 (Pentium processor) means there is a floating-point unit on-chip. Use the feature flags for this determination.
- Do not assume processors with higher family or model numbers have all the features of a processor with a lower family or model number. For example, a processor with a family value of 6 (P6 family processor) may not necessarily have all the features of a processor with a family value of 5.
- Do not assume that the features in the OverDrive processors are the same as those in the OEM version of the processor. Internal caches and instruction execution might vary.
- Do not use undocumented features of a processor to identify steppings or features. For example, the Intel386 processor A-step had bit instructions that were withdrawn with the B-step. Some software attempted to execute these instructions and depended on the invalid-opcode exception as a signal that it was not running on the A-step part. The software failed to work correctly when the Intel486 processor used the same opcodes for different instructions. The software should have used the stepping information in the processor signature.
- Test feature flags individually and do not make assumptions about undefined bits. For example, it would be a mistake to test the FPU bit by comparing the feature register to a binary 1 with a compare instruction.
- Do not assume the clock of a given family or model runs at a specific frequency, and do not write processor speed-dependent code, such as timing loops. For instance, an OverDrive Processor could operate at a higher internal frequency and still report the same family and/or model. Instead, use a combination of the system's timers to measure elapsed time and the Time-Stamp Counter (TSC) to measure processor core clocks to allow direct calibration of the processor core. See Section 10 and Example 6 for details.
- Processor model-specific registers may differ among processors, including in various models of the Pentium processor. Do not use these registers unless identified for the installed processor. This is particularly important for systems upgradeable with an OverDrive processor. Only use Model Specific registers that are defined in the BIOS writers guide for that processor.
- Do not rely on the result of the CPUID algorithm when executed in virtual 8086 mode.
- Do not assume any ordering of model and/or stepping numbers. They are assigned arbitrarily.

- Do not assume processor serial number is a unique number without further qualifiers.
- Display processor serial number as 6 groups of 4 hex nibbles (e.g. XXXX-XXXX-XXXX-XXXX-XXXX-XXXX where X represents a hex digit).
- Display alpha hex characters as capital letters.
- A zero in the lower 64 bits of the processor serial number indicate the processor serial number is invalid, not supported, or disabled on this processor.

§

6 Proper Identification Sequence

To identify the processor using the CPUID instructions, software should follow the following steps.

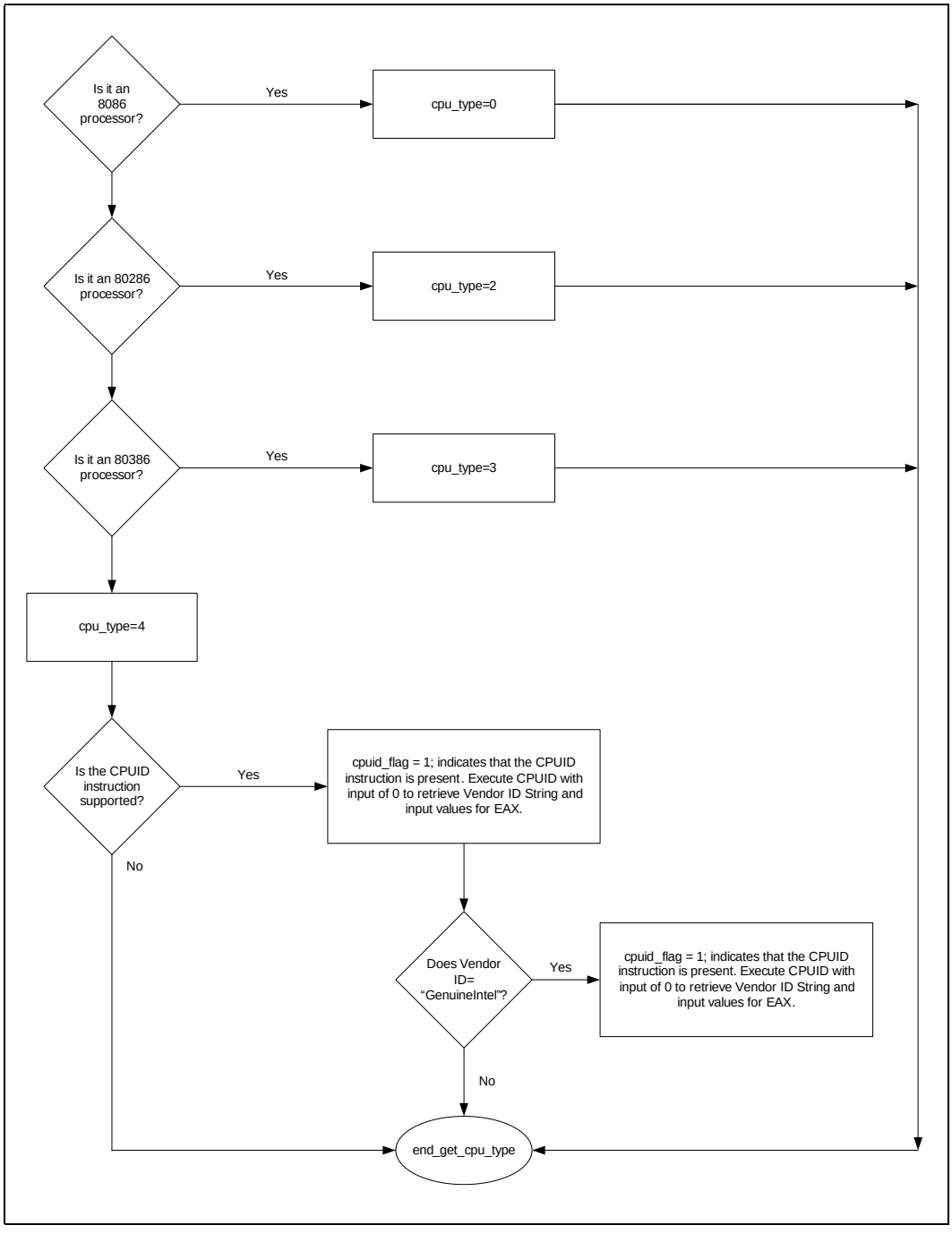
1. Determine if the CPUID instruction is supported by modifying the ID flag in the EFLAGS register. If the ID flag cannot be modified, the processor cannot be identified using the CPUID instruction.
2. Execute the CPUID instruction with EAX equal to 80000000h. CPUID function 80000000h is used to determine if Brand String is supported. If the CPUID function 80000000h returns a value in EAX greater than or equal to 80000004h the Brand String feature is supported and software should use CPUID functions 80000002h through 80000004h to identify the processor.
3. If the Brand String feature is not supported, execute CPUID with EAX equal to 1. CPUID function 1 returns the processor signature in the EAX register, and the Brand ID in the EBX register bits 0 through 7. If the EBX register bits 0 through 7 contain a non-zero value, the Brand ID is supported. Software should scan the list of Brand IDs (see Table 4-1) to identify the processor.
4. If the Brand ID feature is not supported, software should use the processor signature (see Table 2-2 and Table 2-3) in conjunction with the cache descriptors (see Section 2.1.3) to identify the processor.

The cpuid3a.asm program example demonstrates the correct use of the CPUID instruction. It also shows how to identify earlier processor generations that do not implement the Brand String, Brand ID, processor signature or CPUID instruction (see Figure 6-1). This program example contains the following two procedures:

- get_cpu_type identifies the processor type. Figure 6-1 illustrates the flow of this procedure.
- get_fpu_type determines the type of floating-point unit (FPU) or math coprocessor (MCP).

This assembly language program example is suitable for inclusion in a run-time library, or as system calls in operating systems.

Figure 6-1. Flow of Processor get_cpu_type Procedure



§

7 Denormals Are Zero

With the introduction of the SSE2 extensions, some Intel Architecture processors have the ability to convert SSE and SSE2 source operand denormal numbers to zero. This feature is referred to as Denormals-Are-Zero (DAZ). The DAZ mode is not compatible with IEEE Standard 754. The DAZ mode is provided to improve processor performance for applications such as streaming media processing, where rounding a denormal operand to zero does not appreciably affect the quality of the processed data.

Some processor steppings support SSE2 but do not support the DAZ mode. To determine if a processor supports the DAZ mode, software must perform the following steps:

1. Execute the CPUID instruction with an input value of EAX=0 and ensure the vendor-ID string returned is “GenuineIntel”.
2. Execute the CPUID instruction with EAX=1. This will load the EDX register with the feature flags.
3. Ensure that the FXSR feature flag (EDX bit 24) is set. This indicates the processor supports the FXSAVE and FXRSTOR instructions.
4. Ensure that the SSE feature flag (EDX bit 25) or the SSE2 feature flag (EDX bit 26) is set. This indicates that the processor supports at least one of the SSE/SSE2 instruction sets and its MXCSR control register.
5. Zero a 16-byte aligned, 512-byte area of memory. This is necessary since some implementations of FXSAVE do not modify reserved areas within the image.
6. Execute an FXSAVE into the cleared area.
7. Bytes 28-31 of the FXSAVE image are defined to contain the MXCSR_MASK. If this value is 0, then the processor's MXCSR_MASK is 0xFFBF, otherwise MXCSR_MASK is the value of this dword.
8. If bit 6 of the MXCSR_MASK is set, then DAZ is supported.

After completing this algorithm, if DAZ is supported, software can enable DAZ mode by setting bit 6 in the MXCSR register save area and executing the FXRSTOR instruction. Alternately software can enable DAZ mode by setting bit 6 in the MXCSR by executing the LDMXCSR instruction. Refer to the chapter titled “Programming with the Streaming SIMD Extensions (SSE)” in the Intel Architecture Software Developer’s Manual volume 1: Basic Architecture.

The assembly language program dazdtect.asm (see Detecting Denormals-Are-Zero Support) demonstrates this DAZ detection algorithm.

§

8 Operating Frequency

With the introduction of the Time-Stamp Counter, it is possible for software operating in real mode or protected mode with ring 0 privilege to calculate the actual operating frequency of the processor. To calculate the operating frequency, the software needs a reference period. The reference period can be a periodic interrupt, or another timer that is based on time, and not based on a system clock. Software needs to read the Time-Stamp Counter (TSC) at the beginning and ending of the reference period. Software can read the TSC by executing the RDTSC instruction, or by setting the ECX register to 10h and executing the RDMSR instruction. Both instructions copy the current 64-bit TSC into the EDX:EAX register pair.

To determine the operating frequency of the processor, software performs the following steps. The assembly language program frequenc.asm (see Frequency Detection Procedure) demonstrates the us of the frequency detection algorithm.

1. Execute the CPUID instruction with an input value of EAX=0 and ensure the vendor-ID string returned is “GenuineIntel”.
2. Execute the CPUID instruction with EAX=1 to load the EDX register with the feature flags.
3. Ensure that the TSC feature flag (EDX bit 4) is set. This indicates the processor supports the Time-Stamp Counter and RDTSC instruction.
4. Read the TSC at the beginning of the reference period.
5. Read the TSC at the end of the reference period.
6. Compute the TSC delta from the beginning and ending of the reference period.
7. Compute the actual frequency by dividing the TSC delta by the reference period.

Actual frequency = (Ending TSC value – Beginning TSC value) / reference period.

Note: **The measured accuracy is dependent on the accuracy of the reference period. A longer reference period produces a more accurate result. In addition, repeating the calculation multiple times may also improve accuracy.**

Intel processors that support the IA32_MPERF (C0 maximum frequency clock count) register improve on the ability to calculate the C0 state frequency by providing a resetable free running counter. To use the IA32_MPERF register to determine frequency, software should clear the register by a write of ‘0’ while the core is in a C0 state. Subsequently, at the end of a reference period read the IA32_MPERF register. The actual frequency is calculated by dividing the IA32_MPERF register value by the reference period.

1. Execute the CPUID instruction with an input value of EAX=0 and ensure the vendor-ID string returned is “GenuineIntel”.
2. Execute the CPUID instruction with EAX=1 to load the EAX register with the Processor Signature value.
3. Ensure that the processor belongs to the Intel Core 2 Duo processor family. This indicates the processor supports the Maximum Frequency Clock Count.
4. Clear the IA32_MPERF register at the beginning of the reference period.
5. Read the IA32_MPERF register at the end of the reference period.

6. Compute the actual frequency by dividing the IA32_MPERF delta by the reference period.

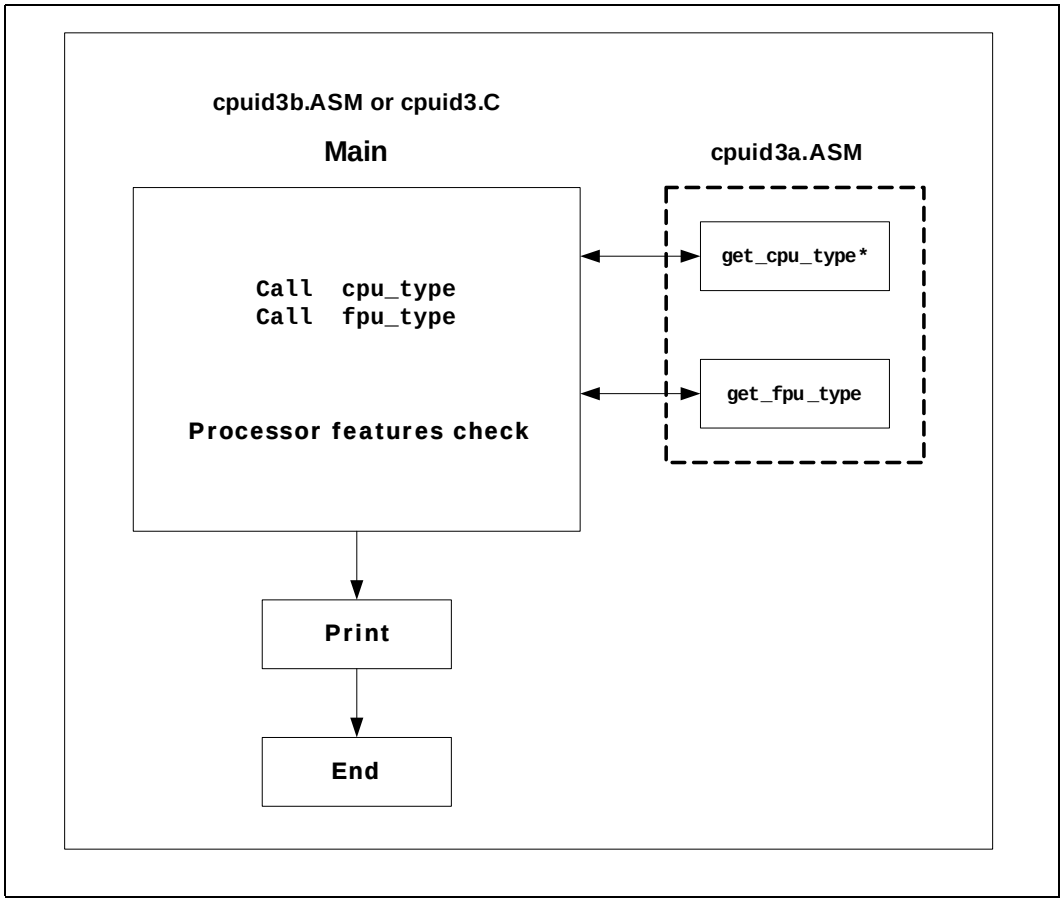
Actual frequency = Ending IA32_MPERF value / reference period

§

9 Program Examples

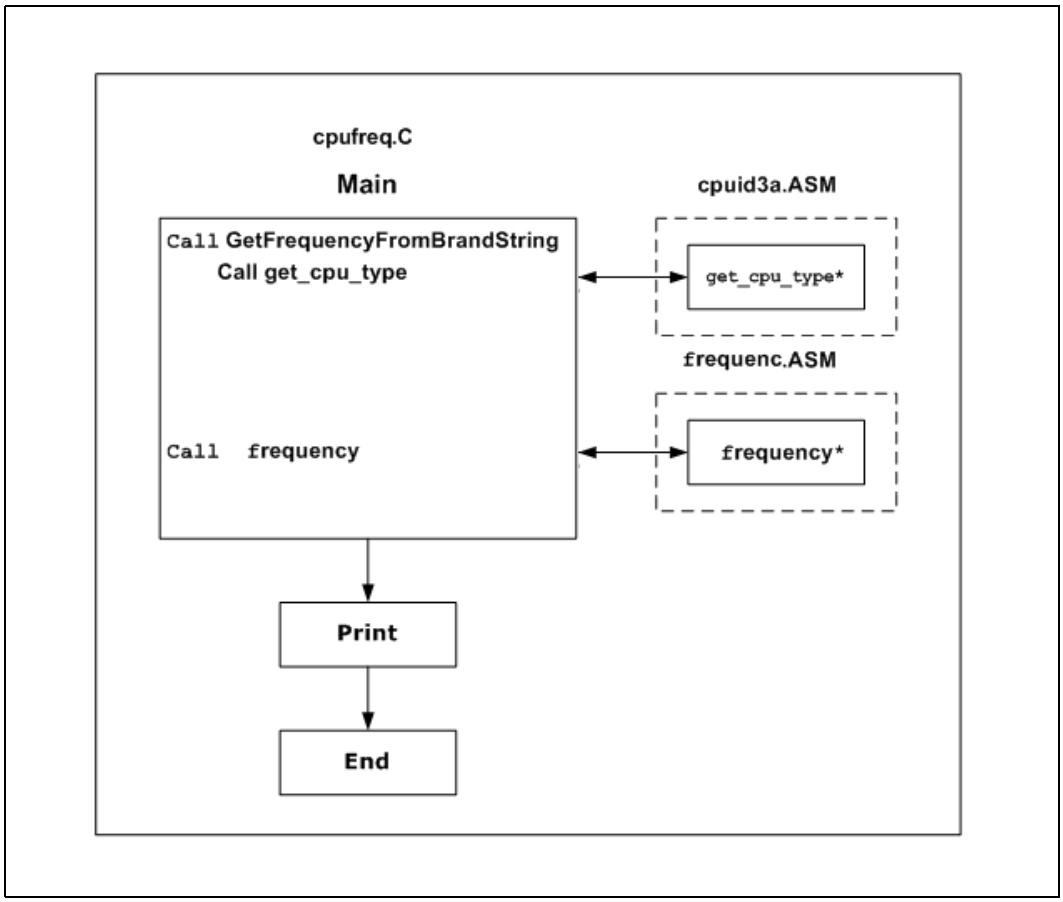
As noted in Chapter 4, the cpuid3a.asm program shows how software forms the brand string; the code is shown in Example 9-1. The cpuid3b.asm and cpuid3.c programs demonstrate applications that call get_cpu_type and get_fpu_type procedures, interpret the returned information, and display the information in the DOS environment; this code is shown in Example 9-2 and Example 9-3. cpuid3a.asm and cpuid3b.asm are written in x86 Assembly language, and cpuid3.c is written in C language. Figure 9-1 presents an overview of the relationship between the three programs.

Figure 9-1. Flow of Processor Identification Extraction Procedure



The cpufreq.c program demonstrates an application that calls get_cpu_type and frequency to calculate the processor frequency; this code is shown in Example 9-6. The frequenc.asm program shows how software calculates frequency from the Time Stamp Counter and brand string; this code is shown in Example 9-5. cpuid3a.asm and frequenc.asm are written in x86 Assembly language, and cpufreq.c is written in C language. Figure 9-2 presents an overview of the relationship between the three programs.

Figure 9-2. Flow of Processor Frequency Calculation Procedure



Example 9-1.Processor Identification Extraction Procedure

```
; Filename: CPUID3A.ASM
; Copyright (c) Intel Corporation 1993-2009
;
; This program has been developed by Intel Corporation.  Intel
; has various intellectual property rights which it may assert
; under certain circumstances, such as if another
; manufacturer's processor mis-identifies itself as being
; "GenuineIntel" when the CPUID instruction is executed.
;
; Intel specifically disclaims all warranties, express or
; implied, and all liability, including consequential and other
; indirect damages, for the use of this program, including
; liability for infringement of any proprietary rights,
; and including the warranties of merchantability and fitness
; for a particular purpose.  Intel does not assume any
; responsibility for any errors which may appear in this program
; nor any responsibility to update it.
;
;*****************************************************************
;
; This code contains two procedures:
; _get_cpu_type: Identifies processor type in _cpu_type:
;     0=8086/8088 processor
;     2=Intel 286 processor
;     3=Intel386(TM) family processor
;     4=Intel486(TM) family processor
;     5=Pentium(R) family processor
;     6=P6 family of processors
;     F=Pentium 4 family of processors
;
; _get_fpu_type: Identifies FPU type in _fpu_type:
;     0=FPU not present
;     1=FPU present
;     2=287 present (only if cpu_type=3)
;     3=387 present (only if cpu_type=3)
;
;*****************************************************************
;
; This program has been compiled with Microsoft Macro Assembler
; 6.15. If this code is compiled with no options specified and
; linked with CPUID3.C or CPUID3B.ASM, it's assumed to correctly
; identify the current Intel 8086/8088, 80286, 80386, 80486,
; Pentium(R), Pentium(R) Pro, Pentium(R) II, Pentium(R) II
; Xeon(R), Pentium(R) II OverDrive(R), Intel(R) Celeron(R),
; Pentium(R) III, Pentium(R) III Xeon(R), Pentium(R) 4, Intel(R)
; Xeon(R) DP and MP, Intel(R) Core(TM), Intel(R) Core(TM) 2,
; Intel(R) Core(TM) i7, and Intel(R) Atom(TM) processors when
; executed in real-address mode.
;
; NOTE: When using this code with C program CPUID3.C, 32-bit
;       segments are recommended.
;
;*****************************************************************

TITLE CPUID3A

; comment the following line for 32-bit segments
.DOSSEG

; uncomment the following 2 lines for 32-bit segments
;.386
```


```
;.MODEL flat

; comment the following line for 32-bit segments
.MODEL small

CPU_ID          MACRO
        db 0Fh                          ; CPUID opcodes
        db 0A2h
ENDM

.DATA
public _cpu_type
public _fpu_type
public _v86_flag
public _cpuid_flag
public _intel_CPU
public _max_funct
public _vendor_id
public _cpu_signature
public _features_ebx
public _features_ecx
public _features_edx
public _funct_5_eax
public _funct_5_ebx
public _funct_5_ecx
public _funct_5_edx
public _funct_6_eax
public _funct_6_ebx
public _funct_6_ecx
public _funct_6_edx
public _funct_A_eax
public _funct_A_ebx
public _funct_A_ecx
public _funct_A_edx
public _ext_max_funct
public _ext_funct_1_eax
public _ext_funct_1_ebx
public _ext_funct_1_ecx
public _ext_funct_1_edx
public _ext_funct_6_eax
public _ext_funct_6_ebx
public _ext_funct_6_ecx
public _ext_funct_6_edx
public _ext_funct_7_eax
public _ext_funct_7_ebx
public _ext_funct_7_ecx
public _ext_funct_7_edx
public _ext_funct_8_eax
public _ext_funct_8_ebx
public _ext_funct_8_ecx
public _ext_funct_8_edx
public _cache_eax
public _cache_ebx
public _cache_ecx
public _cache_edx
public _dcp_cache_eax
public _dcp_cache_ebx
public _dcp_cache_ecx
public _dcp_cache_edx
public _brand_string

_cpu_type          db 0
```

```
_fpu_type          db 0
_v86_flag          db 0
_cpuid_flag        db 0
_intel_CPU         db 0
_max_funct         dd 0                    ; Maximum function from CPUID.0.EAX
_vendor_id         db "------------"
intel_id           db "GenuineIntel"
_cpu_signature     dd 0                    ; CPUID.1.EAX
_features_ebx      dd 0                    ; CPUID.1.EBX
_features_ecx      dd 0                    ; CPUID.1.ECX - feature flags
_features_edx      dd 0                    ; CPUID.1.EDX - feature flags
_funct_5_eax       dd 0                    ; CPUID.5.EAX - monitor/mwait leaf
_funct_5_ebx       dd 0                    ; CPUID.5.EBX - monitor/mwait leaf
_funct_5_ecx       dd 0                    ; CPUID.5.ECX - monitor/mwait leaf
_funct_5_edx       dd 0                    ; CPUID.5.EDX - monitor/mwait leaf
_funct_6_eax       dd 0                    ; CPUID.6.EAX - sensor & power mgmt. flags
_funct_6_ebx       dd 0                    ; CPUID.6.EBX - sensor & power mgmt. flags
_funct_6_ecx       dd 0                    ; CPUID.6.ECX - sensor & power mgmt. flags
_funct_6_edx       dd 0                    ; CPUID.6.EDX - sensor & power mgmt. flags
_funct_A_eax       dd 0                    ; CPUID.A.EAX
_funct_A_ebx       dd 0                    ; CPUID.A.EBX
_funct_A_ecx       dd 0                    ; CPUID.A.ECX
_funct_A_edx       dd 0                    ; CPUID.A.EDX
_ext_max_funct     dd 0                    ; Max ext leaf from CPUID.80000000h.EAX
_ext_funct_1_eax   dd 0
_ext_funct_1_ebx   dd 0
_ext_funct_1_ecx   dd 0
_ext_funct_1_edx   dd 0
_ext_funct_6_eax   dd 0
_ext_funct_6_ebx   dd 0
_ext_funct_6_ecx   dd 0
_ext_funct_6_edx   dd 0
_ext_funct_7_eax   dd 0
_ext_funct_7_ebx   dd 0
_ext_funct_7_ecx   dd 0
_ext_funct_7_edx   dd 0
_ext_funct_8_eax   dd 0
_ext_funct_8_ebx   dd 0
_ext_funct_8_ecx   dd 0
_ext_funct_8_edx   dd 0
_cache_eax         dd 0
_cache_ebx         dd 0
_cache_ecx         dd 0
_cache_edx         dd 0
_dcp_cache_eax     dd 0
_dcp_cache_ebx     dd 0
_dcp_cache_ecx     dd 0
_dcp_cache_edx     dd 0
_brand_string      db 48 dup (0)
fp_status          dw 0

CPUID_regs    struct
    regEax             dd 0
    regEbx             dd 0
    regEcx             dd 0
    regEdx             dd 0
CPUID_regs    ends

.CODE
.586

;*****************************************************************************
```

```
; _cpuidEx
;   Input: SP+2  – EAX value to set
;          SP+6  – ECX value to set
;          SP+10 – offset of CPUID_regs structure
;*************************************************************************
_cpuidEx proc public
    push    bp                          ; .small model causes proc call to push IP onto stack,
    mov     bp, sp                      ;  so parameters passed to proc start at SP+4 after PUSH
BP
    pushad
    mov     eax, [bp+4]                   ; get EAX from stack
    mov     ecx, [bp+8]                   ; get ECX from stack
    cpuid
    mov     si, [bp+12]                   ; get offset of CPUID_regs structure
    mov     dword ptr [si].CPUID_regs.regEax, eax
    mov     dword ptr [si].CPUID_regs.regEbx, ebx
    mov     dword ptr [si].CPUID_regs.regEcx, ecx
    mov     dword ptr [si].CPUID_regs.regEdx, edx
    popad
    pop     bp
    ret
_cpuidEx endp

; comment the following line for 32-bit segments
.8086

; uncomment the following line for 32-bit segments
;.386

;*************************************************************************
_get_cpu_type proc public

; This procedure determines the type of processor in a system
; and sets the _cpu_type variable with the appropriate
; value.  If the CPUID instruction is available, it is used
; to determine more specific details about the processor.
; All registers are used by this procedure, none are preserved.
; To avoid AC faults, the AM bit in CR0 must not be set.

; Intel 8086 processor check
; Bits 12-15 of the FLAGS register are always set on the
; 8086 processor.

;
; For 32-bit segments comment the following lines down to the next
; comment line that says “STOP”
;
check_8086:
    pushf                                 ; push original FLAGS
    pop     ax                            ; get original FLAGS
    mov     cx, ax                        ; save original FLAGS
    and     ax, 0FFFh                     ; clear bits 12-15 in FLAGS
    push    ax                            ; save new FLAGS value on stack
    popf                                  ; replace current FLAGS value
    pushf                                 ; get new FLAGS
    pop     ax                            ; store new FLAGS in AX
    and     ax, 0F000h                    ; if bits 12-15 are set, then
    cmp     ax, 0F000h                    ;  processor is an 8086/8088
    mov     _cpu_type, 0                  ; turn on 8086/8088 flag
    jne     check_80286                   ; go check for 80286
    push    sp                            ; double check with push sp
    pop     dx                            ; if value pushed was different
```

```
    cmp     dx, sp                          ; means it's really an 8086
    jne     end_cpu_type                    ; jump if processor is 8086/8088
    mov     _cpu_type, 10h                  ; indicate unknown processor
    jmp     end_cpu_type

; Intel 286 processor check
; Bits 12-15 of the FLAGS register are always clear on the
; Intel 286 processor in real-address mode.

.286
check_80286:
    smsw    ax                              ; save machine status word
    and     ax, 1                           ; isolate PE bit of MSW
    mov     _v86_flag, al                   ; save PE bit to indicate V86
    or      cx, 0F000h                      ; try to set bits 12-15
    push    cx                              ; save new FLAGS value on stack
    popf                                    ; replace current FLAGS value
    pushf                                   ; get new FLAGS
    pop     ax                              ; store new FLAGS in AX
    and     ax, 0F000h                      ; if bits 12-15 are clear
    mov     _cpu_type, 2                    ; processor=80286, turn on 80286 flag
    jz      end_cpu_type                    ; jump if processor is 80286

; Intel386 processor check
; The AC bit, bit #18, is a new bit introduced in the EFLAGS
; register on the Intel486 processor to generate alignment
; faults.
; This bit cannot be set on the Intel386 processor.

.386
; "STOP"

;                                           ; it is safe to use 386 instructions
check_80386:
    pushfd                                  ; push original EFLAGS
    pop     eax                             ; get original EFLAGS
    mov     ecx, eax                        ; save original EFLAGS
    xor     eax, 40000h                     ; flip AC bit in EFLAGS
    push    eax                             ; save new EFLAGS value on stack
    popfd                                   ; replace current EFLAGS value
    pushfd                                  ; get new EFLAGS
    pop     eax                             ; store new EFLAGS in EAX
    xor     eax, ecx                        ; can't toggle AC bit processor=80386
    mov     _cpu_type, 3                    ; turn on 80386 processor flag
    jz      end_cpu_type                    ; jump if 80386 processor
    push    ecx
    popfd                                   ; restore AC bit in EFLAGS first

; Intel486 processor check
; Checking for ability to set/clear ID flag (Bit 21) in EFLAGS
; which indicates the presence of a processor with the CPUID
; instruction.

.486
check_80486:
    mov     _cpu_type, 4                    ; turn on 80486 processor flag
    mov     eax, ecx                        ; get original EFLAGS
    xor     eax, 200000h                    ; flip ID bit in EFLAGS
    push    eax                             ; save new EFLAGS value on stack
    popfd                                   ; replace current EFLAGS value
    pushfd                                  ; get new EFLAGS
    pop     eax                             ; store new EFLAGS in EAX
```


```
    xor     eax, ecx                        ; can't toggle ID bit,
    je      end_cpu_type                    ; processor=80486

; Execute CPUID instruction to determine vendor, family,
; model, stepping and features.  For the purpose of this
; code, only the initial set of CPUID information is saved.
.586
    mov     _cpuid_flag, 1                  ; flag indicating use of CPUID inst.
    push    ebx                             ; save registers
    push    esi
    push    edi

    xor     eax, eax                        ; get Vendor ID
    cpuid
    mov     _max_funct, eax
    mov     dword ptr _vendor_id, ebx
    mov     dword ptr _vendor_id[4], edx
    mov     dword ptr _vendor_id[8], ecx

    cmp     dword ptr intel_id, ebx
    jne     end_cpuid_type
    cmp     dword ptr intel_id[4], edx
    jne     end_cpuid_type
    cmp     dword ptr intel_id[8], ecx
    jne     end_cpuid_type                  ; if not equal, not an Intel processor

    mov     _intel_CPU, 1                   ; indicate an Intel processor
    cmp     eax, 1                          ; is 1 valid input for CPUID?
    jb      ext_functions                   ; jmp if no

    mov     eax, 1                          ; get family/model/stepping/features
    cpuid
    mov     _cpu_signature, eax
    mov     _features_ebx, ebx
    mov     _features_ecx, ecx
    mov     _features_edx, edx
    shr     eax, 8                          ; isolate family
    and     eax, 0Fh
    mov     _cpu_type, al                   ; set _cpu_type with family

    cmp     _max_funct, 2                   ; is 2 valid input for CPUID?
    jb      ext_functions                   ; jmp if no

    mov     eax, 2                          ; set up to read cache descriptor
    cpuid                                   ; This code supports one iteration only
    mov     _cache_eax, eax
    mov     _cache_ebx, ebx
    mov     _cache_ecx, ecx
    mov     _cache_edx, edx

    cmp     _max_funct, 4                   ; is 4 valid input for CPUID?
    jb      ext_functions                   ; jmp if no

    mov     eax, 4                          ; set up to read deterministic cache params
    xor     ecx, ecx
    cpuid
    push    eax
    and     al, 1Fh                         ; determine if valid cache parameters read
    cmp     al, 00h                         ; EAX[4:0] = 0 indicates invalid cache
    pop     eax
    je      cpuid_5
```

```
    mov     _dcp_cache_eax, eax             ; store deterministic cache information
    mov     _dcp_cache_ebx, ebx
    mov     _dcp_cache_ecx, ecx
    mov     _dcp_cache_edx, edx

cpuid_5:
    cmp     _max_funct, 5                   ; is 5 valid input for CPUID?
    jb      ext_functions                   ; jmp if no

    mov     eax, 5
    cpuid
    mov     _funct_5_eax, eax
    mov     _funct_5_ebx, ebx
    mov     _funct_5_ecx, ecx
    mov     _funct_5_edx, edx

    cmp     _max_funct, 6                   ; is 6 valid input for CPUID?
    jb      ext_functions                   ; jmp if no

    mov     eax, 6
    cpuid
    mov     _funct_6_eax, eax
    mov     _funct_6_ebx, ebx
    mov     _funct_6_ecx, ecx
    mov     _funct_6_edx, edx

    cmp     _max_funct, 0Ah                 ; is 0Ah valid input for CPUID?
    jb      ext_functions                   ; jmp if no

    mov     eax, 0Ah
    cpuid
    mov     _funct_A_eax, eax
    mov     _funct_A_ebx, ebx
    mov     _funct_A_ecx, ecx
    mov     _funct_A_edx, edx

ext_functions:
    mov     eax, 80000000h                  ; check if brand string is supported
    cpuid
    mov     _ext_max_funct, eax

    cmp     _ext_max_funct, 80000001h       ; is 80000001h valid input for CPUID?
    jb      end_cpuid_type                  ; jmp if no

    mov     eax, 80000001h                  ; Get the Extended Feature Flags
    cpuid
    mov     _ext_funct_1_eax, eax
    mov     _ext_funct_1_ebx, ebx
    mov     _ext_funct_1_ecx, ecx
    mov     _ext_funct_1_edx, edx

    cmp     _ext_max_funct, 80000002h       ; is 80000002h valid input for CPUID?
    jb      end_cpuid_type                  ; jmp if no

    mov     di, offset _brand_string
    mov     eax, 80000002h                  ; get first 16 bytes of brand string
    cpuid
    mov     dword ptr [di], eax             ; save bytes 0-15
    mov     dword ptr [di+4], ebx
    mov     dword ptr [di+8], ecx
    mov     dword ptr [di+12], edx
    add     di, 16
```

```
    mov     eax, 80000003h
    cpuid
    mov     dword ptr [di], eax             ; save bytes 16-31
    mov     dword ptr [di+4], ebx
    mov     dword ptr [di+8], ecx
    mov     dword ptr [di+12], edx
    add     di, 16

    mov     eax, 80000004h
    cpuid
    mov     dword ptr [di], eax             ; save bytes 32-47
    mov     dword ptr [di+4], ebx
    mov     dword ptr [di+8], ecx
    mov     dword ptr [di+12], edx

    cmp     _ext_max_funct, 80000006h       ; is 80000006h valid input for CPUID?
    jb      end_cpuid_type                  ; jmp if no

    mov     eax, 80000006h
    cpuid
    mov     _ext_funct_6_eax, eax
    mov     _ext_funct_6_ebx, ebx
    mov     _ext_funct_6_ecx, ecx
    mov     _ext_funct_6_edx, edx

    cmp     _ext_max_funct, 80000007h       ; is 80000007h valid input for CPUID?
    jb      end_cpuid_type                  ; jmp if no

    mov     eax, 80000007h
    cpuid
    mov     _ext_funct_7_eax, eax
    mov     _ext_funct_7_ebx, ebx
    mov     _ext_funct_7_ecx, ecx
    mov     _ext_funct_7_edx, edx

    cmp     _ext_max_funct, 80000008h       ; is 80000008h valid input for CPUID?
    jb      end_cpuid_type                  ; jmp if no

    mov     eax, 80000008h
    cpuid
    mov     _ext_funct_8_eax, eax
    mov     _ext_funct_8_ebx, ebx
    mov     _ext_funct_8_ecx, ecx
    mov     _ext_funct_8_edx, edx

end_cpuid_type:
    pop     edi                             ; restore registers
    pop     esi
    pop     ebx

; comment the following line for 32-bit segments
.8086
end_cpu_type:
    ret
_get_cpu_type        endp

;*************************************************************************

_get_fpu_type proc public

; This procedure determines the type of FPU in a system
```

```
; and sets the _fpu_type variable with the appropriate value.
; All registers are used by this procedure, none are preserved.

; Coprocessor check
; The algorithm is to determine whether the floating-point
; status and control words are present.  If not, no
; coprocessor exists.  If the status and control words can
; be saved, the correct coprocessor is then determined
; depending on the processor type.  The Intel386 processor can
; work with either an Intel287 NDP or an Intel387 NDP.
; The infinity of the coprocessor must be checked to determine
; the correct coprocessor type.

    fninit                                  ; reset FP status word
    mov     fp_status, 5A5Ah                ; initialize temp word to non-zero
    fnstsw  fp_status                       ; save FP status word
    mov     ax, fp_status                   ; check FP status word
    cmp     al, 0                           ; was correct status written
    mov     _fpu_type, 0                    ; no FPU present
    jne     end_fpu_type

check_control_word:
    fnstcw  fp_status                       ; save FP control word
    mov     ax, fp_status                   ; check FP control word
    and     ax, 103fh                       ; selected parts to examine
    cmp     ax, 3fh                         ; was control word correct
    mov     _fpu_type, 0
    jne     end_fpu_type                    ; incorrect control word, no FPU
    mov     _fpu_type, 1

; 80287/80387 check for the Intel386 processor

check_infinity:
    cmp     _cpu_type, 3
    jne     end_fpu_type
    fld1                                    ; must use default control from FNINIT
    fldz                                    ; form infinity
    fdiv                                    ; 8087/Intel287 NDP say +inf = -inf
    fld     st                              ; form negative infinity
    fchs                                    ; Intel387 NDP says +inf <> -inf
    fcompp                                  ; see if they are the same
    fstsw   fp_status                       ; look at status from FCOMPP
    mov     ax, fp_status
    mov     _fpu_type, 2                    ; store Intel287 NDP for FPU type
    sahf                                    ; see if infinities matched
    jz      end_fpu_type                    ; jump if 8087 or Intel287 is present
    mov     _fpu_type, 3                    ; store Intel387 NDP for FPU type

end_fpu_type:
    ret
_get_fpu_type        endp

    end
```

Example 9-2.Processor Identification Procedure in Assembly Language

```
; Filename: CPUID3B.ASM
; Copyright (c) Intel Corporation 1993-2009
;
; This program has been developed by Intel Corporation.  Intel
; has various intellectual property rights which it may assert
; under certain circumstances, such as if another
; manufacturer's processor mis-identifies itself as being
; "GenuineIntel" when the CPUID instruction is executed.
;
; Intel specifically disclaims all warranties, express or
; implied, and all liability, including consequential and
; other indirect damages, for the use of this program,
; including liability for infringement of any proprietary
; rights, and including the warranties of merchantability and
; fitness for a particular purpose.  Intel does not assume any
; responsibility for any errors which may appear in this
; program nor any responsibility to update it.
;
;*****************************************************************
;
; This program contains three parts:
; Part 1: Identifies processor type in the variable
;         _cpu_type:
;
; Part 2: Identifies FPU type in the variable
;         _fpu_type:
;
; Part 3: Prints out the appropriate messages.  This part is
;         specific to the DOS environment and uses the DOS
;         system calls to print out the messages.
;
;*****************************************************************
;
; This program has been compiled with Microsoft Macro Assembler
; 6.15. If this code is compiled with no options specified and
; linked with CPUID3A.ASM, it's assumed to correctly identify
; the current Intel 8086/8088, 80286, 80386, 80486, Pentium(R),
; Pentium(R) Pro, Pentium(R) II, Pentium(R) II Xeon(R),
; Pentium(R) II OverDrive(R), Intel(R) Celeron(R),
; Pentium(R) III, Pentium(R) III Xeon(R), Pentium(R) 4, Intel(R)
; Xeon(R) DP and MP, Intel(R) Core(TM), Intel(R) Core(TM) 2,
; Intel(R) Core(TM) i7, and Intel(R) Atom(TM) processors when
; executed in real-address mode.
;
; NOTE: This code is written using 16-bit Segments.
;       This module is the application; CPUID3A.ASM is linked as
;       a support module.
;
;*****************************************************************

TITLE CPUID3B

.DOSSEG
.MODEL small

.STACK 100h

OP_O        MACRO
    db      66h                             ; hardcoded operand override
ENDM
```

```
.DATA
extrn   _cpu_type:          byte
extrn   _fpu_type:          byte
extrn   _cpuid_flag:        byte
extrn   _intel_CPU:         byte
extrn   _max_funct:         dword
extrn   _cpu_signature:     dword
extrn   _features_ecx:      dword
extrn   _features_edx:      dword
extrn   _features_ebx:      dword
extrn   _funct_5_eax:       dword
extrn   _funct_5_ebx:       dword
extrn   _funct_5_ecx:       dword
extrn   _funct_5_edx:       dword
extrn   _funct_6_eax:       dword
extrn   _funct_6_ebx:       dword
extrn   _funct_6_ecx:       dword
extrn   _funct_6_edx:       dword
extrn   _funct_A_eax:       dword
extrn   _funct_A_ebx:       dword
extrn   _funct_A_ecx:       dword
extrn   _funct_A_edx:       dword
extrn   _ext_max_funct:     dword
extrn   _ext_funct_1_eax:   dword
extrn   _ext_funct_1_ebx:   dword
extrn   _ext_funct_1_ecx:   dword
extrn   _ext_funct_1_edx:   dword
extrn   _ext_funct_6_eax:   dword
extrn   _ext_funct_6_ebx:   dword
extrn   _ext_funct_6_ecx:   dword
extrn   _ext_funct_6_edx:   dword
extrn   _ext_funct_7_eax:   dword
extrn   _ext_funct_7_ebx:   dword
extrn   _ext_funct_7_ecx:   dword
extrn   _ext_funct_7_edx:   dword
extrn   _ext_funct_8_eax:   dword
extrn   _ext_funct_8_ebx:   dword
extrn   _ext_funct_8_ecx:   dword
extrn   _ext_funct_8_edx:   dword
extrn   _cache_eax:         dword
extrn   _cache_ebx:         dword
extrn   _cache_ecx:         dword
extrn   _cache_edx:         dword
extrn   _dcp_cache_eax:     dword
extrn   _dcp_cache_ebx:     dword
extrn   _dcp_cache_ecx:     dword
extrn   _dcp_cache_edx:     dword
extrn   _brand_string:      byte

; The purpose of this code is to identify the processor and
; coprocessor that is currently in the system.  The program
; first determines the processor type.  Then it determines
; whether a coprocessor exists in the system.  If a
; coprocessor or integrated coprocessor exists, the program
; identifies the coprocessor type.  The program then prints
; the processor and floating point processors present and type.

.CODE
.8086
start:
    mov     ax, @data
    mov     ds, ax                          ; set segment register
```

```
    mov     es, ax                          ; set segment register
    and     sp, not 3                       ; align stack to avoid AC fault
    call    _get_cpu_type                   ; determine processor type
    call    _get_fpu_type
    call    print

    mov     ax, 4C00h
    int     21h

extrn   _get_cpu_type:      proc
extrn   _get_fpu_type:      proc

.DATA
id_msg                  db “This processor:$”
cp_error                db “n unknown processor$”
cp_8086                 db “n 8086/8088 processor$”
cp_286                  db “n 80286 processor$”
cp_386                  db “n 80386 processor$”
cp_486                  db “n 80486DX, 80486DX2 processor or”
                        db “ 80487SX math coprocessor$”
cp_486sx                db “n 80486SX processor$”

fp_8087                 db “ and an 8087 math coprocessor$”
fp_287                  db “ and an 80287 math coprocessor$”
fp_387                  db “ and an 80387 math coprocessor$”

intel486_msg            db “ Genuine Intel486(TM) processor$”
intel486dx_msg          db “ Genuine Intel486(TM) DX processor$”
intel486sx_msg          db “ Genuine Intel486(TM) SX processor$”
inteldx2_msg            db “ Genuine IntelDX2(TM) processor$”
intelsx2_msg            db “ Genuine IntelSX2(TM) processor$”
inteldx4_msg            db “ Genuine IntelDX4(TM) processor$”
inteldx2wb_msg          db “ Genuine Write-Back Enhanced”
                        db “ IntelDX2(TM) processor$”
pentium_msg             db “ Genuine Intel(R) Pentium(R) processor$”
pentiumpro_msg          db “ Genuine Intel Pentium(R) Pro processor$”
pentiumiimodel3_msg     db “ Genuine Intel(R) Pentium(R) II processor, model 3$”
pentiumiixeon_m5_msg    db “ Genuine Intel(R) Pentium(R) II processor, model 5 or”
                        db “ Intel(R) Pentium(R) II Xeon(R) processor$”
pentiumiixeon_msg       db “ Genuine Intel(R) Pentium(R) II Xeon(R) processor$”
celeron_msg             db “ Genuine Intel(R) Celeron(R) processor, model 5$”
celeronmodel6_msg       db “ Genuine Intel(R) Celeron(R) processor, model 6$”
celeron_brand           db “ Genuine Intel(R) Celeron(R) processor$”
pentiumiii_msg          db “ Genuine Intel(R) Pentium(R) III processor, model 7 or”
                        db “ Intel Pentium(R) III Xeon(R) processor, model 7$”
pentiumiiixeon_msg      db “ Genuine Intel(R) Pentium(R) III Xeon(R) processor, model 7$”
pentiumiiixeon_brand    db “ Genuine Intel(R) Pentium(R) III Xeon(R) processor$”
pentiumiii_brand        db “ Genuine Intel(R) Pentium(R) III processor$”
mobile_piii_brand       db “ Genuine Mobile Intel(R) Pentium(R) III Processor-M$”
mobile_icp_brand        db “ Genuine Mobile Intel(R) Celeron(R) processor$”
mobile_P4_brand         db “ Genuine Mobile Intel(R) Pentium(R) 4 processor - M$”
pentium4_brand          db “ Genuine Intel(R) Pentium(R) 4 processor$”
xeon_brand              db “ Genuine Intel(R) Xeon(R) processor$”
xeon_mp_brand           db “ Genuine Intel(R) Xeon(R) processor MP$”
mobile_icp_brand_2      db “ Genuine Mobile Intel(R) Celeron(R) processor$”
mobile_pentium_m_brand  db “ Genuine Intel(R) Pentium(R) M processor$”
mobile_genuine_brand    db “ Mobile Genuine Intel(R) processor$”
mobile_icp_m_brand      db “ Genuine Intel(R) Celeron(R) M processor$”
unknown_msg             db “n unknown Genuine Intel(R) processor$”

brand_entry        struct
    brand_value     dd ?
```

```
    brand_string    dw ?
brand_entry        ends

brand_table LABEL BYTE
    brand_entry         <01h, offset celeron_brand>
    brand_entry         <02h, offset pentiumiii_brand>
    brand_entry         <03h, offset pentiumiiixeon_brand>
    brand_entry         <04h, offset pentiumiii_brand>
    brand_entry         <06h, offset mobile_piii_brand>
    brand_entry         <07h, offset mobile_icp_brand>
    brand_entry         <08h, offset pentium4_brand>
    brand_entry         <09h, offset pentium4_brand>
    brand_entry         <0Ah, offset celeron_brand>
    brand_entry         <0Bh, offset xeon_brand>
    brand_entry         <0Ch, offset xeon_mp_brand>
    brand_entry         <0Eh, offset mobile_p4_brand>
    brand_entry         <0Fh, offset mobile_icp_brand>
    brand_entry         <11h, offset mobile_genuine_brand>
    brand_entry         <12h, offset mobile_icp_m_brand>
    brand_entry         <13h, offset mobile_icp_brand_2>
    brand_entry         <14h, offset celeron_brand>
    brand_entry         <15h, offset mobile_genuine_brand>
    brand_entry         <16h, offset mobile_pentium_m_brand>
    brand_entry         <17h, offset mobile_icp_brand_2>

brand_table_count       equ ($ - offset brand_table) / (sizeof brand_entry)

; The following 16 entries must stay intact as an array
intel_486_0             dw offset intel486dx_msg
intel_486_1             dw offset intel486dx_msg
intel_486_2             dw offset intel486sx_msg
intel_486_3             dw offset inteldx2_msg
intel_486_4             dw offset intel486_msg
intel_486_5             dw offset intelsx2_msg
intel_486_6             dw offset intel486_msg
intel_486_7             dw offset inteldx2wb_msg
intel_486_8             dw offset inteldx4_msg
intel_486_9             dw offset intel486_msg
intel_486_a             dw offset intel486_msg
intel_486_b             dw offset intel486_msg
intel_486_c             dw offset intel486_msg
intel_486_d             dw offset intel486_msg
intel_486_e             dw offset intel486_msg
intel_486_f             dw offset intel486_msg
; end of array

not_intel               db “t least an 80486 processor.”
                        db 13,10,”It does not contain a Genuine”
                        db “Intel part and as a result,”
                        db “the”,13,10,”CPUID”
                        db “ detection information cannot be”
                        db “ determined.$”

signature_msg           db 13,10,”Processor Signature / Version Information: $”
family_msg              db 13,10,”Processor Family: $”
model_msg               db 13,10,”Model:            $”
stepping_msg            db 13,10,”Stepping:         $”
ext_fam_msg             db 13,10,”Extended Family:  $”
ext_mod_msg             db 13,10,”Extended Model:   $”
cr_lf                   db 13,10,”$”
turbo_msg               db 13,10,”The processor is an OverDrive(R) processor$”
dp_msg                  db 13,10,”The processor is the upgrade”
```

```
                        db " processor in a dual processor system$"
; CPUID features documented per Software Developers Manual June 2009
document_msg            db 13,10,"CPUID features documented in the Intel(R) 64 and IA-32 Software
Developer"
                       db 13,10,"Manual Volume 2A Instruction Set A-M [doc #253666 on intel.com]$"
delimiter_msg           db "; $"
colon_msg               db ": $"
supported_msg           db "Supported$"
unsupported_msg         db "Unsupported$"
reserved_msg            db "Reserved$"
invalid_msg             db "Invalid$"
not_available_msg       db "Not Available$"
available_msg           db "Available$"
bytes_msg               db " bytes$"
kbytes_msg              db " KB$"
mbytes_msg              db " MB$"

feature_entry    struct
    feature_mask    dd ?
    feature_msg     db 16 dup(?)            ; 16 characters max including $ terminator
feature_entry    ends

; CPUID.1.ECX features
feature_1_ecx_msg       db 13,10,"CPUID.1.ECX Features: $"
feature_1_ecx_table LABEL BYTE
    feature_entry       <00000001h, "SSE3$">            ; [0]
    feature_entry       <00000002h, "PCLMULQDQ$">       ; [1]
    feature_entry       <00000004h, "DTES64$">          ; [2]
    feature_entry       <00000008h, "MONITOR$">         ; [3]
    feature_entry       <00000010h, "DS-CPL$">          ; [4]
    feature_entry       <00000020h, "VMX$">             ; [5]
    feature_entry       <00000040h, "SMX$">             ; [6]
    feature_entry       <00000080h, "EIST$">            ; [7]
    feature_entry       <00000100h, "TM2$">             ; [8]
    feature_entry       <00000200h, "SSSE3$">           ; [9]
    feature_entry       <00000400h, "CNXT-ID$">         ; [10]
    feature_entry       <00001000h, "FMA$">             ; [12]
    feature_entry       <00002000h, "CMPXCHG16B$">      ; [13]
    feature_entry       <00004000h, "XTPR$">            ; [14]
    feature_entry       <00008000h, "PDCM$">            ; [15]
    feature_entry       <00040000h, "DCA$">             ; [18]
    feature_entry       <00080000h, "SSE4.1$">          ; [19]
    feature_entry       <00100000h, "SSE4.2$">          ; [20]
    feature_entry       <00200000h, "x2APIC$">          ; [21]
    feature_entry       <00400000h, "MOVBE$">           ; [22]
    feature_entry       <00800000h, "POPCNT$">          ; [23]
    feature_entry       <02000000h, "AES$">             ; [25]
    feature_entry       <04000000h, "XSAVE$">           ; [26]
    feature_entry       <08000000h, "OSXSAVE$">         ; [27]
    feature_entry       <10000000h, "AVX$">             ; [28]

feature_1_ecx_table_count equ ($ - offset feature_1_ecx_table) / (sizeof feature_entry)

; CPUID.1.EDX features
feature_1_edx_msg       db 13,10,"CPUID.1.EDX Features: $"
feature_1_edx_table LABEL BYTE
    feature_entry       <00000001h, "FPU$">             ; [0]
    feature_entry       <00000002h, "VME$">             ; [1]
    feature_entry       <00000004h, "DE$">              ; [2]
    feature_entry       <00000008h, "PSE$">             ; [3]
    feature_entry       <00000010h, "TSC$">             ; [4]
    feature_entry       <00000020h, "MSR$">             ; [5]
```

```
    feature_entry       <00000040h, “PAE$”>             ; [6]
    feature_entry       <00000080h, “MCE$”>             ; [7]
    feature_entry       <00000100h, “CX8$”>             ; [8]
    feature_entry       <00000200h, “APIC$”>            ; [9]
    feature_entry       <00000800h, “SEP$”>             ; [11]
    feature_entry       <00001000h, “MTRR$”>            ; [12]
    feature_entry       <00002000h, “PGE$”>             ; [13]
    feature_entry       <00004000h, “MCA$”>             ; [14]
    feature_entry       <00008000h, “CMOV$”>            ; [15]
    feature_entry       <00010000h, “PAT$”>             ; [16]
    feature_entry       <00020000h, “PSE36$”>           ; [17]
    feature_entry       <00040000h, “PSN$”>             ; [18]
    feature_entry       <00080000h, “CLFSH$”>           ; [19]
    feature_entry       <00200000h, “DS$”>              ; [21]
    feature_entry       <00400000h, “ACPI$”>            ; [22]
    feature_entry       <00800000h, “MMX$”>             ; [23]
    feature_entry       <01000000h, “FXSR$”>            ; [24]
    feature_entry       <02000000h, “SSE$”>             ; [25]
    feature_entry       <04000000h, “SSE2$”>            ; [26]
    feature_entry       <08000000h, “SS$”>              ; [27]
    feature_entry       <10000000h, “HTT$”>             ; [28]
    feature_entry       <20000000h, “TM$”>              ; [29]
    feature_entry       <80000000h, “PBE$”>             ; [31]

feature_1_edx_table_count equ ($ - offset feature_1_edx_table) / (sizeof feature_entry)

; CPUID.6.EAX features
feature_6_eax_msg       db 13,10,”CPUID.6.EAX Features: $”
feature_6_eax_table LABEL BYTE
    feature_entry       <00000001h, “DIGTEMP$”>         ; [0]
    feature_entry       <00000002h, “TRBOBST$”>         ; [1]
    feature_entry       <00000004h, “ARAT$”>            ; [2]

feature_6_eax_table_count equ ($ - offset feature_6_eax_table) / (sizeof feature_entry)

; CPUID.6.ECX features
feature_6_ecx_msg       db 13,10,”CPUID.6.ECX Features: $”
feature_6_ecx_table LABEL BYTE
    feature_entry       <00000001h, “MPERF-APERF-MSR$”> ; [0]

feature_6_ecx_table_count equ ($ - offset feature_6_ecx_table) / (sizeof feature_entry)

; CPUID.80000001h.ECX features
feature_ext1_ecx_msg    db 13,10,”CPUID.80000001h.ECX Features: $”
feature_ext1_ecx_table LABEL BYTE
    feature_entry       <00000001h, “LAHF-SAHF$”>       ; [0]

feature_ext1_ecx_table_count equ ($ - offset feature_ext1_ecx_table) / (sizeof feature_entry)

; CPUID.80000001h.EDX features
feature_ext1_edx_msg    db 13,10,”CPUID.80000001h.EDX Features: $”
feature_ext1_edx_table LABEL BYTE
    feature_entry       <00000800h, “SYSCALL$”>         ; [11]
    feature_entry       <00100000h, “XD$”>              ; [20]
    feature_entry       <08000000h, “RDTSCP$”>          ; [27]
    feature_entry       <20000000h, “EM64T$”>           ; [29]

feature_ext1_edx_table_count equ ($ - offset feature_ext1_edx_table) / (sizeof feature_entry)

; CPUID.80000007h.EDX features
feature_ext7_edx_msg    db 13,10,”CPUID.80000007h.EDX Features: $”
feature_ext7_edx_table LABEL BYTE
```

```
    feature_entry       <00000100h, “INVTSC$”>          ; [8]

feature_ext7_edx_table_count equ ($ - offset feature_ext7_edx_table) / (sizeof feature_entry)

; CPUID.4
func_4_msg              db 13,10,13,10,”CPUID.4 Deterministic Cache Parameters (DCP) Leaf:$”
dcp_entry    struct
    cache_type_msg      db 16 dup(?)            ; 16 characters max including $ terminator
dcp_entry    ends

cache_msg               db “ Cache$”
level_msg               db “Level $”
size_msg                db “Size $”
dcp_table LABEL BYTE
    dcp_entry           <“Null$”>
    dcp_entry           <“       Data$”>
    dcp_entry           <“Instruction$”>
    dcp_entry           <“    Unified$”>
dcp_table_count equ ($ - offset dcp_table) / (sizeof dcp_entry)

; CPUID.5
func_5_msg              db 13,10,13,10,”CPUID.5 Monitor/MWAIT Leaf:$”
func_5_eax_msg          db 13,10,”  Smallest monitor line size: $”
func_5_ebx_msg          db 13,10,”  Largest monitor line size: $”
func_5_mwait_msg        db 13,10,”  Enumeration of Monitor-MWAIT extensions: $”
func_5_mwait_intr_msg   db 13,10,”  Interrupts as break-event for MWAIT: $”
func_5_mwait_Cx_msg1    db 13,10,”  Number of C$”
func_5_mwait_Cx_msg2    db “ sub C-States supported using MWAIT: $”

; CPUID.A
func_A_msg              db 13,10,13,10,”CPUID.A Architecture Performance Monitoring Leaf:$”
archPerfMon_ver_msg     db 13,10,”  Version ID: $”
gp_perfMon_counter_msg  db 13,10,”  Number of General Purpose Counters per Logical Processor: $”
bits_gp_counter_msg     db 13,10,”  Bit Width of General Purpose Counters: $”
ebx_bit_vector_len_msg  db 13,10,”  Length of EBX bit vector to enumerate events: $”
core_cycle_msg          db 13,10,”  Core Cycle event                 : $”
inst_retired_msg        db 13,10,”  Instruction Retired event       : $”
reference_cycles_msg    db 13,10,”  Reference Cycles event          : $”
lastlevelcache_ref_msg  db 13,10,”  Last-Level Cache Reference event: $”
lastlevelcache_miss_msg db 13,10,”  Last-Level Cache Misses event   : $”
branch_inst_retired_msg db 13,10,”  Branch Instruction Retired event: $”
branch_mispredict_msg   db 13,10,”  Branch Mispredict Retired event : $”
fixed_func_counter_msg  db 13,10,”  Number of Fixed-Function Performance Counters: $”
bits_fixed_func_counter db 13,10,”  Bit Width of Fixed-Function Performance Counters: $”

archPerfMon_table LABEL BYTE
    brand_entry         <0, offset core_cycle_msg>
    brand_entry         <1, offset inst_retired_msg>
    brand_entry         <2, offset reference_cycles_msg>
    brand_entry         <3, offset lastlevelcache_ref_msg>
    brand_entry         <4, offset lastlevelcache_miss_msg>
    brand_entry         <5, offset branch_inst_retired_msg>
    brand_entry         <6, offset branch_mispredict_msg>

archPerfMon_table_count equ ($ - offset archPerfMon_table) / (sizeof brand_entry)

; CPUID.B
func_B_msg              db 13,10,”CPUID.B Extended Topology Leaf n=$”
thread_msg              db “Thread$”        ; Input ECX=0
core_msg                db “Core$”          ; Input ECX=1
package_msg             db “Package$”       ; Input ECX=2..n
smt_msg                 db “SMT$”
```

```
x2apic_id_shr_msg       db "  x2APIC ID bits to shift right to get unique topology ID: $"
logical_proc_level_msg  db "  Logical processors at this level type: $"
level_number_msg        db "  Level Number: $"
level_type_msg          db "  Level Type  : $"
x2apic_id_msg           db "  x2APIC ID   : $"

; CPUID.D
func_D_mainLeaf_msg     db 13,10,13,10,"CPUID.D Processor Extended State Enumeration Main Leaf
n=0:$"
func_D_mainLeaf_eax_msg db 13,10,"  Valid bit fields of XCR0[31: 0]: $"
func_D_mainLeaf_ebx_msg db 13,10,"  Max size required by enabled features in XCR0: $"
func_D_mainLeaf_ecx_msg db 13,10,"  Max size required by XSAVE/XRSTOR for supported features: $"
func_D_mainLeaf_edx_msg db 13,10,"  Valid bit fields of XCR0[63:32]: $"
func_D_subLeaf_msg      db 13,10,"CPUID.D Processor Extended State Enumeration Sub-Leaf n=$"
func_D_subLeaf_eax_msg  db 13,10,"  Size required for feature associated with sub-leaf: $"
func_D_subLeaf_ebx_msg  db 13,10,"  Offset of save area from start of XSAVE/XRSTOR area: $"
func_D_subLeaf_ecx_msg  db 13,10,"  Reserved: $"
func_D_subLeaf_edx_msg  db 13,10,"  Reserved: $"

.CODE
.486

;*****************************************************************************
; printDecimal
;   Input: eax – value to print
;          bl - # of bytes in value (1=byte, 2=word, 4=dword)
;*****************************************************************************
printDecimal    proc
    pushad

checkDecByte:
    mov     edx, 0FFh
    cmp     bl, 1
    je      startDec
checkDecWord:
    mov     edx, 0FFFFh
    cmp     bl, 2
    je      startDec
checkDecDword:
    mov     edx, 0FFFFFFFFh
    cmp     bl, 4
    jne     exit_PrintDecimal

startDec:
    and     eax, edx
    mov     ebp, 10                         ; set destination base to 10 (decimal)
    xor     cx, cx                          ; initialize saved digit counter

doDivDec:
    xor     edx, edx                        ; clear high portion of dividend
    div     ebp

    push    dx                              ; save remainder
    inc     cx                              ; increment saved digit counter

    or      eax, eax                        ; is quotient 0 (need more div)?
    jnz     doDivDec                        ; jmp if no

    mov     ah, 2                           ; print char service
printDec:
    pop     dx                              ; pop digit
    add     dl, '0'                         ; convert digit to ASCII 0-9
```

```
    int     21h                             ; print it
    loop    printDec                        ; decrement counter and loop if > 0

exit_PrintDecimal:
    popad
    ret
printDecimal    endp

;*************************************************************************
; printBinary - prints value in binary, assumes lowest bit in [0]
;   Input: eax – value to print
;          bl - # of bits in value (e.g. 4=nibble, 8=byte, 16=word, 32=dword)
;*************************************************************************
printBinary    proc
    pushad

    cmp     bl, 32                          ; bit count > 32?
    ja      exit_PrintBinary                ; jmp if yes

startBin:
    mov     cl, 32                          ; max 32 bits
    sub     cl, bl                          ; get # of unused bits in value
    mov     esi, eax
    shl     esi, cl                         ; shift bits so highest used bit is in [31]
    movzx   cx, bl                          ; initialize bit counter
    mov     ah, 2                           ; print char service
printBin:
    mov     dl, ‘0’
    shl     esi, 1                          ; shift bit to print into CF
    adc     dl, 0                           ; add CF to convert bit to ASCII 0 or 1
    int     21h                             ; print char
skipPrintBin:
    loop    printBin

    mov     dl, ‘b’                         ; binary indicator
    int     21h

exit_PrintBinary:
    popad
    ret
printBinary    endp

;*************************************************************************
; printDwordHex
;   Input: eax – dword to print
;*************************************************************************
printDwordHex    proc
    pushad

    mov     ebx, eax
    mov     cx, 8                           ; print the 8 digits of EAX[31:0]
    mov     ah, 2                           ; print char service

printNibble:
    rol     ebx, 4                          ; moving EAX[31:28] into EAX[3:0]
    mov     dl, bl
    and     dl, 0Fh
    add     dl, ‘0’                         ; convert nibble to ASCII number
    cmp     dl, ‘9’
    jle     @f
    add     dl, 7                           ; convert to ‘A’-’F’
```

```
@@:
    int     21h                             ; print lower nibble of ext family
    loop    printNibble

    mov     dl, 'h'                         ; hex indicator
    int     21h

    popad
    ret
printDwordHex    endp

.586

;*****************************************************************************
; printFeaturesTable
;   Input: esi – 32-bit features value
;          di  – offset of features table
;          cl  – count of features table
;          dx  - offset of features msg
;*****************************************************************************
printFeaturesTable    proc
    or      esi, esi                        ; Are any features flags present (>0)?
    jz      exit_PrintFeatures              ; jmp if no

    mov     ah, 9                           ; print string $ terminated service
    int     21h

print_features_loop:
    push    esi
    and     esi, dword ptr [di].feature_entry.feature_mask
    pop     esi
    jz      check_next_feature

print_features_msg:
    mov     ah, 9                           ; print string $ terminated service
    lea     dx, [di].feature_entry.feature_msg
    int     21h
    mov     ah, 2                           ; print char service
    mov     dl, ' '
    int     21h

check_next_feature:
    add     di, sizeof feature_entry
    dec     cl                              ; decrement feature count
    jnz     print_features_loop

exit_PrintFeatures:
    ret
printFeaturesTable    endp

;*****************************************************************************
; printFeatures - print CPU features reported by CPUID
;*****************************************************************************
printFeatures proc
    pushad

    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset document_msg
    int     21h
    mov     dx, offset cr_lf
    int     21h
; print CPUID.1.ECX features
```


```
    mov     esi, _features_ecx
    mov     dx, offset feature_1_ecx_msg
    mov     di, offset feature_1_ecx_table
    mov     cl, feature_1_ecx_table_count
    call    printFeaturesTable
; print CPUID.1.EDX features
    mov     esi, dword ptr _features_edx
; Fast System Call had a different meaning on some processors,
; so check CPU signature.
check_sep:
    bt      esi, 11                         ; Is SEP bit set?
    jnc     check_htt                       ; jmp if no
    cmp     _cpu_type, 6                    ; Is CPU type != 6?
    jne     check_htt                       ; jmp if yes
    cmp     byte ptr _cpu_signature, 33h    ; Is CPU signature >= 33h?
    jae     check_htt                       ; jmp if yes
    btr     esi, 11                         ; SEP not truly present
                                            ;  so clear feature bit
; HTT can be fused off even when feature flag reports HTT supported,
; so perform additional checks.
check_htt:
    bt      esi, 28                         ; Is HTT bit set?
    jnc     print_edx_features              ; jmp if no
    mov     eax, dword ptr _features_ebx
    shr     eax, 16                         ; Place the logical processor count in AL
    xor     ah, ah                          ; clear AH.
    mov     ebx, dword ptr _dcp_cache_eax
    shr     ebx, 26                         ; Place core count in BL (originally in EAX[31:26])
    and     bx, 3Fh                         ; clear BL preserving the core count
    inc     bl
    div     bl
    cmp     al, 2                           ; >= 2 cores?
    jae     print_edx_features              ; jmp if yes
    btr     esi, 28                         ; HTT not truly present
                                            ;  so clear feature bit
print_edx_features:
    mov     dx, offset feature_1_edx_msg
    mov     di, offset feature_1_edx_table
    mov     cl, feature_1_edx_table_count
    call    printFeaturesTable
; print CPUID.6.EAX features
    mov     esi, _funct_6_eax
    mov     dx, offset feature_6_eax_msg
    mov     di, offset feature_6_eax_table
    mov     cl, feature_6_eax_table_count
    call    printFeaturesTable
; print CPUID.6.ECX features
    mov     esi, _funct_6_ecx
    mov     dx, offset feature_6_ecx_msg
    mov     di, offset feature_6_ecx_table
    mov     cl, feature_6_ecx_table_count
    call    printFeaturesTable
; print CPUID.80000001h.ECX features
    mov     esi, _ext_funct_1_ecx
    mov     dx, offset feature_ext1_ecx_msg
    mov     di, offset feature_ext1_ecx_table
    mov     cl, feature_ext1_ecx_table_count
    call    printFeaturesTable
; print CPUID.80000001h.EDX features
    mov     esi, _ext_funct_1_edx
    mov     dx, offset feature_ext1_edx_msg
    mov     di, offset feature_ext1_edx_table
```

```
    mov     cl, feature_ext1_edx_table_count
    call    printFeaturesTable
; print CPUID.80000007h.EDX features
    mov     esi, _ext_funct_7_edx
    mov     dx, offset feature_ext7_edx_msg
    mov     di, offset feature_ext7_edx_table
    mov     cl, feature_ext7_edx_table_count
    call    printFeaturesTable

    popad
    ret
printFeatures endp

;*****************************************************************************
; printDCPLeaf - print Deterministic Cache Parameters Leaf
;*****************************************************************************
printDCPLeaf proc
    pushad
    cmp     _max_funct, 4
    jb      exit_dcp

    mov     eax, 4
    xor     ecx, ecx                    ; set Index to 0
    cpuid
    and     ax, 1Fh                     ; Cache Type=0 (NULL)?
    jz      exit_dcp                    ; jmp if yes

    mov     ah, 9                       ; print string $ terminated service
    mov     dx, offset func_4_msg
    int     21h

    xor     ebp, ebp                    ; start at Index 0
loop_dcp:
    mov     eax, 4
    mov     ecx, ebp                    ; set Index
    cpuid
    mov     si, ax
    and     si, 1Fh                     ; Cache Type=0 (NULL)?
    jz      exit_dcp                    ; jmp if yes

    inc     ebp                         ; next Index
    cmp     si, dcp_table_count
    ja      loop_dcp

    push    eax
    mov     ah, 9                       ; print string $ terminated service
    mov     dx, offset cr_lf
    int     21h
    pop     eax
    shr     al, 5                       ; Get Cache Level
    add     al, '0'                     ; Convert digit to ASCII
    push    ebx
    push    ecx
    push    eax
    mov     ah, 2                       ; print char service
    mov     dl, ' '
    int     21h
    int     21h
    mov     ax, sizeof dcp_entry
    mul     si
    mov     si, ax
    mov     ah, 9                       ; print string $ terminated service
```

```
    lea     dx, dcp_table[si].dcp_entry
    int     21h
    mov     dx, offset cache_msg
    int     21h
    mov     dx, offset delimiter_msg
    int     21h
    mov     dx, offset level_msg
    int     21h
    pop     edx
    mov     ah, 2                           ; print char service
    int     21h                             ; print Cache Level
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset delimiter_msg
    int     21h
    mov     dx, offset size_msg
    int     21h
    pop     ecx
    pop     ebx                             ; Cache Size =
(Ways+1)*(Partitions+1)*(LineSize+1)*(Sets+1)
    mov     eax, ebx
    and     eax, 0FFFh                      ; EAX = LineSize
    inc     eax                             ; EAX = LineSize+1
    inc     ecx                             ; ECX = Sets+1
    mul     ecx                             ; EAX = (LineSize+1)*(Sets+1)
    mov     ecx, ebx
    shr     ecx, 12
    and     ecx, 03FFh                      ; ECX = Partitions
    inc     ecx                             ; ECX = Partitions+1
    mul     ecx                             ; EAX = (Partitions+1)*(LineSize+1)*(Sets+1)
    shr     ebx, 22
    inc     ebx                             ; EBX = Ways+1
    mul     ebx                             ; EAX = (Ways+1)*(Partitions+1)*(LineSize+1)*(Sets+1)
    shr     eax, 10                         ; convert bytes to KB
    mov     bl, 4
    call    printDecimal
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset kbytes_msg
    int     21h
    jmp     loop_dcp

exit_dcp:
    popad
    ret
printDCPLeaf endp

;******************************************************************************
; printMwaitLeaf - print Monitor/Mwait Leaf
;******************************************************************************
printMwaitLeaf proc
    pushad
    cmp     _max_funct, 5
    jb      exit_mwait

    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset func_5_msg
    int     21h
    mov     dx, offset func_5_eax_msg
    int     21h
    mov     eax, _funct_5_eax
    mov     bl, 2                           ; output word
    call    printDecimal
    mov     ah, 9                           ; print string $ terminated service
```

```
    mov     dx, offset bytes_msg
    int     21h
    mov     dx, offset func_5_ebx_msg
    int     21h
    mov     eax, _funct_5_ebx
    mov     bl, 2                           ; output word
    call    printDecimal
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset bytes_msg
    int     21h
    mov     dx, offset func_5_mwait_msg
    int     21h
    mov     ecx, _funct_5_ecx
    mov     dx, offset supported_msg        ; output supported string
    bt      ecx, 0                          ; is enumeration of MWAIT extensions supported?
    jc      printMwaitSupport               ; jmp if yes
    mov     dx, offset unsupported_msg      ; output unsupported string
printMwaitSupport:
    int     21h
    mov     dx, offset func_5_mwait_intr_msg
    int     21h
    mov     dx, offset supported_msg        ; output supported string
    bt      ecx, 0                          ; is intr as break-event for MWAIT supported?
    jc      printMwaitIntr                  ; jmp if yes
    mov     dx, offset unsupported_msg      ; output unsupported string
printMwaitIntr:
    int     21h

    mov     esi, _funct_5_edx
    xor     cx, cx                          ; CX is counter, start at 0 to match starting at C0
loop_mwait_Cx:
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset func_5_mwait_Cx_msg1
    int     21h
    mov     ax, cx                          ; output counter, which matches Cx being output
    mov     bl, 1                           ; output byte
    call    printDecimal
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset func_5_mwait_Cx_msg2
    int     21h
    mov     ax, si                          ; get Cx value
    and     ax, 0Fh                         ; keep [3:0]
    mov     bl, 1                           ; output byte
    call    printDecimal

    shr     esi, 4                          ; put next 4 bits into SI[3:0]
    inc     cx                              ; increment counter
    cmp     cx, 5                           ; have 5 iterations been completed?
    jb      loop_mwait_Cx                   ; jmp if no

exit_mwait:
    popad
    ret
printMwaitLeaf endp

;*****************************************************************************
; printArchPerfMonLeaf - print Architecture Performance Monitoring Leaf
;*****************************************************************************
printArchPerfMonLeaf proc
    pushad
    cmp     _max_funct, 0Ah
    jb      exit_ArchPerfMon
```

```
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset func_A_msg
    int     21h
    mov     dx, offset archPerfMon_ver_msg
    int     21h
    mov     eax, _funct_A_eax
    mov     ecx, eax                        ; copy value so bits can be rotated
    mov     bl, 1                           ; output byte
    call    printDecimal
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset gp_perfMon_counter_msg
    int     21h
    ror     ecx, 8                          ; next 8 bits
    mov     al, cl
    call    printDecimal
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset bits_gp_counter_msg
    int     21h
    ror     ecx, 8                          ; next 8 bits
    mov     al, cl
    call    printDecimal
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset ebx_bit_vector_len_msg
    int     21h
    ror     ecx, 8                          ; next 8 bits
    mov     al, cl
    call    printDecimal

    push    ecx
    mov     si, offset archPerfMon_table
    movzx   cx, cl                          ; use Length of EBX Vector to Enumerate Events
loop_ArchPerfMon:
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, word ptr [si].brand_entry.brand_string
    int     21h
    movzx    ebp, byte ptr [si].brand_entry.brand_value
    mov     dx, offset available_msg
    bt      _funct_A_ebx, ebp
    jnc     @f
    mov     dx, offset not_available_msg
@@:
    int     21h
    add     si, sizeof brand_entry
    loop    loop_ArchPerfMon

    pop     ecx
    ror     ecx, 8                          ; next 8 bits
    cmp     cl, 2                           ; is Version ID < 2?
    jb      exit_ArchPerfMon                ; jmp if yes

    mov     dx, offset fixed_func_counter_msg
    int     21h
    mov     eax, _funct_A_edx
    and     ax, 1Fh
    mov     bl, 1                           ; output byte
    call    printDecimal
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset bits_fixed_func_counter
    int     21h
    mov     eax, _funct_A_edx
    shr     eax, 5
```

```
    call    printDecimal

exit_ArchPerfMon:
    popad
    ret
printArchPerfMonLeaf endp

;*****************************************************************************
; printExtendedTopologyLeaf - print Extended Topology Leaf
;*****************************************************************************
printExtendedTopologyLeaf proc
    pushad
    cmp     _max_funct, 0Bh
    jb      exit_extTopology

    xor     ebp, ebp                        ; start at Index 0

loop_extTopology:
    mov     eax, 0Bh
    mov     ecx, ebp                        ; set Index
    cpuid
    mov     esi, eax
    or      esi, ebx                        ; is EAX=EBX=0?
    jz      exit_extTopology                ; jmp if yes

    or      ebp, ebp                        ; is Index 0?
    jnz     saveRegs_extTopology
    push    eax
    push    edx
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset cr_lf
    int     21h
    pop     edx
    pop     eax

saveRegs_extTopology:
    push    edx
    push    ecx
    push    ebx
    push    eax
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset func_B_msg
    int     21h
    mov     eax, ebp
    mov     bl, 4
    call    printDecimal                    ; print leaf number
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset colon_msg
    int     21h
    mov     dx, offset cr_lf
    int     21h
    mov     dx, offset x2apic_id_shr_msg
    int     21h
    pop     eax                             ; restore EAX to print
    and     al, 1Fh
    mov     bl, 1
    call    printDecimal                    ; print x2APIC ID bits
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset cr_lf
    int     21h
    mov     dx, offset logical_proc_level_msg
    int     21h
```

```
    pop     eax                             ; restore EBX to print
    and     al, 1Fh
    mov     bl, 1
    call    printDecimal                    ; print number of logical processors at level
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset cr_lf
    int     21h
    mov     dx, offset level_number_msg
    int     21h
    pop     ecx                             ; restore ECX to print
    mov     al, cl
    mov     bl, 1
    call    printDecimal                    ; print Level Number
    mov     ah, 2                           ; print char service
    mov     dl, ' '
    int     21h
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset thread_msg
    or      cl, cl                          ; is it level 0 (thread)?
    jz      printLevelNumber                ; jmp if yes
    mov     dx, offset core_msg
    cmp     cl, 1                           ; is it level 1 (core)?
    je      printLevelNumber                ; jmp if yes
    mov     dx, offset package_msg          ; level 2..n (package)
printLevelNumber:
    int     21h
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset cr_lf
    int     21h
    mov     dx, offset level_type_msg
    int     21h
    mov     al, ch
    mov     bl, 1
    call    printDecimal                    ; print Level Type
    mov     ah, 2                           ; print char service
    mov     dl, ' '
    int     21h
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset invalid_msg
    or      ch, ch
    jz      printLevelType
    mov     dx, offset smt_msg
    cmp     ch, 1
    je      printLevelType
    mov     dx, offset core_msg
    cmp     ch, 2
    je      printLevelType
    mov     dx, offset reserved_msg
printLevelType:
    int     21h
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset cr_lf
    int     21h
    mov     dx, offset x2apic_id_msg
    int     21h
    pop     eax                             ; restore EDX to print
    mov     bl, 4
    call    printDecimal                    ; print x2APIC ID for current logical processor

    inc     ebp                             ; next Index
    jmp     loop_extTopology
```

```
exit_extTopology:
    popad
    ret
printExtendedTopologyLeaf endp

;*****************************************************************************
; printCpuExtendedStateLeaf - print CPU Extended State Leaf
;*****************************************************************************
printCpuExtendedStateLeaf proc
    pushad
    cmp     _max_funct, 0Dh
    jb      exit_cpuExtState

    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset func_D_mainLeaf_msg
    int     21h
    mov     dx, offset func_D_mainLeaf_eax_msg
    int     21h

    mov     eax, 0Dh
    xor     ecx, ecx                        ; set to Index 0
    cpuid
    push    ecx
    push    ebx
    push    edx
    mov     bl, 32
    call    printBinary                     ; print returned EAX
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset func_D_mainLeaf_edx_msg
    int     21h
    pop     eax                             ; restore EDX to print
    call    printBinary                     ; print returned EDX
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset func_D_mainLeaf_ebx_msg
    int     21h
    mov     bl, 4
    pop     eax                             ; restore EBX to print
    call    printDecimal                    ; print returned EBX
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset bytes_msg
    int     21h
    mov     dx, offset func_D_mainLeaf_ecx_msg
    int     21h
    pop     eax                             ; restore ECX to print
    call    printDecimal                    ; print returned ECX
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset bytes_msg
    int     21h

    mov     ebp, 2                          ; start at Index 2 (Index 1 is reserved)

loop_cpuExtState:
    mov     eax, 0Dh
    mov     ecx, ebp                        ; set Index
    cpuid
    or      eax, eax                        ; is leaf invalid (0)?
    jz      exit_cpuExtState                ; jmp if yes

    push    ebx
    push    eax
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset func_D_subLeaf_msg
```

```
    int     21h
    mov     eax, ebp
    mov     bl, 4
    call    printDecimal
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset colon_msg
    int     21h
    mov     dx, offset func_D_subLeaf_eax_msg
    int     21h
    pop     eax
    call    printDecimal
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset bytes_msg
    int     21h
    mov     dx, offset func_D_subLeaf_ebx_msg
    int     21h
    pop     eax
    call    printDecimal
    inc     ebp                             ; next Index
    jmp     loop_cpuExtState

exit_cpuExtState:
    popad
    ret
printCpuExtendedStateLeaf endp

.8086

;*****************************************************************************
; print
;   Input: none
;
; This procedure prints the appropriate CPUID string and
; numeric processor presence status. If the CPUID instruction
; was used, this procedure prints out the CPUID info.
; All registers are used by this procedure, none are preserved.
;*****************************************************************************
print    proc
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset cr_lf
    int     21h
    mov     dx, offset id_msg               ; print initial message
    int     21h

    cmp     _cpuid_flag, 1                  ; if set to 1, processor
                                            ; supports CPUID instruction
    je      print_cpuid_data                ; print detailed CPUID info

print_86:
    cmp     _cpu_type, 0
    jne     print_286
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset cp_8086
    int     21h
    cmp     _fpu_type, 0
    je      end_print
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset fp_8087
    int     21h
    jmp     end_print

print_286:
```

```
    cmp     _cpu_type, 2
    jne     print_386
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset cp_286
    int     21h
    cmp     _fpu_type, 0
    je      end_print

print_287:
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset fp_287
    int     21h
    jmp     end_print

print_386:
    cmp     _cpu_type, 3
    jne     print_486
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset cp_386
    int     21h
    cmp     _fpu_type, 0
    je      end_print
    cmp     _fpu_type, 2
    je      print_287
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset fp_387
    int     21h
    jmp     end_print

print_486:
    cmp     _cpu_type, 4
    jne     print_unknown                   ; Intel processors will have
    mov     dx, offset cp_486sx             ; CPUID instruction
    cmp     _fpu_type, 0
    je      print_486sx
    mov     dx, offset cp_486

print_486sx:
    mov     ah, 9                           ; print string $ terminated service
    int     21h
    jmp     end_print

print_unknown:
    mov     dx, offset cp_error
    jmp     print_486sx

print_cpuid_data:
.486
    cmp     _intel_CPU, 1                   ; check for genuine Intel
    jne     not_GenuineIntel                ; processor

    mov     di, offset _brand_string        ; brand string supported?
    cmp     byte ptr [di], 0
    je      print_brand_id

    mov     cx, 47                          ; max brand string length

skip_spaces:
    cmp     byte ptr [di], ' '              ; skip leading space chars
    jne     print_brand_string

    inc     di
```

```
    loop    skip_spaces

print_brand_string:
    cmp     cx, 0                       ; Nothing to print
    je      print_brand_id
    cmp     byte ptr [di], 0
    je      print_brand_id

    mov     ah, 2                       ; print char service
    mov     dl, ' '
    int     21h

print_brand_char:
    mov     dl, [di]                    ; print up to the max chars
    int     21h

    inc     di
    cmp     byte ptr [di], 0
    je      print_family
    loop    print_brand_char
    jmp     print_family

print_brand_id:
    cmp     _cpu_type, 6
    jb      print_486_type
    ja      print_pentiumiiimodel8_type
    mov     eax, dword ptr _cpu_signature
    shr     eax, 4
    and     al, 0Fh
    cmp     al, 8
    jne     print_pentiumiiimodel8_type

print_486_type:
    cmp     _cpu_type, 4                ; if 4, print 80486 processor
    jne     print_pentium_type
    mov     eax, dword ptr _cpu_signature
    shr     eax, 4
    and     eax, 0Fh                    ; isolate model
    mov     dx, intel_486_0[eax*2]
    jmp     print_common

print_pentium_type:
    cmp     _cpu_type, 5                ; if 5, print Pentium processor
    jne     print_pentiumpro_type
    mov     dx, offset pentium_msg
    jmp     print_common

print_pentiumpro_type:
    cmp     _cpu_type, 6                ; if 6 & model 1, print Pentium Pro processor
    jne     print_unknown_type
    mov     eax, dword ptr _cpu_signature
    shr     eax, 4
    and     eax, 0Fh                    ; isolate model
    cmp     eax, 3
    jge     print_pentiumiimodel3_type
    cmp     eax, 1
    jne     print_unknown_type          ; incorrect model number = 2
    mov     dx, offset pentiumpro_msg
    jmp     print_common

print_pentiumiimodel3_type:
    cmp     eax, 3                      ; if 6 & model 3,Pentium II processor, model 3
```

```
    jne     print_pentiumiimodel5_type
    mov     dx, offset pentiumiimodel3_msg
    jmp     print_common

print_pentiumiimodel5_type:
    cmp     eax, 5                          ; if 6 & model 5, either Pentium
                                            ; II processor, model 5, Pentium II
                                            ; Xeon processor or Intel Celeron
                                            ; processor, model 5
    je      celeron_xeon_detect

    cmp      eax, 7                         ; If model 7 check cache descriptors
                                            ; to determine Pentium III or Pentium III Xeon
    jne      print_celeronmodel6_type

celeron_xeon_detect:
; Is it Pentium II processor, model 5, Pentium II Xeon processor, Intel Celeron processor,
; Pentium III processor or Pentium III Xeon processor.

    mov     eax, dword ptr _cache_eax
    rol     eax, 8
    mov     cx, 3

celeron_detect_eax:
    cmp     al, 40h                         ; Is it no L2
    je      print_celeron_type
    cmp     al, 44h                         ; Is L2 >= 1M
    jae     print_pentiumiixeon_type

    rol     eax, 8
    loop    celeron_detect_eax

    mov     eax, dword ptr _cache_ebx
    mov     cx, 4

celeron_detect_ebx:
    cmp     al, 40h                         ; Is it no L2
    je      print_celeron_type
    cmp     al, 44h                         ; Is L2 >= 1M
    jae     print_pentiumiixeon_type

    rol     eax, 8
    loop    celeron_detect_ebx

    mov     eax, dword ptr _cache_ecx
    mov     cx, 4

celeron_detect_ecx:
    cmp     al, 40h                         ; Is it no L2
    je      print_celeron_type
    cmp     al, 44h                         ; Is L2 >= 1M
    jae     print_pentiumiixeon_type

    rol     eax, 8
    loop    celeron_detect_ecx

    mov     eax, dword ptr _cache_edx
    mov     cx, 4

celeron_detect_edx:
    cmp     al, 40h                         ; Is it no L2
    je      print_celeron_type
```

```
    cmp     al, 44h                         ; Is L2 >= 1M
    jae     print_pentiumiixeon_type

    rol     eax, 8
    loop    celeron_detect_edx

    mov     dx, offset pentiumiixeon_m5_msg
    mov     eax, dword ptr _cpu_signature
    shr     eax, 4
    and     eax, 0Fh                        ; isolate model
    cmp     eax, 5
    je      print_common
    mov     dx, offset pentiumiii_msg
    jmp     print_common

print_celeron_type:
    mov     dx, offset celeron_msg
    jmp     print_common

print_pentiumiixeon_type:
    mov     dx, offset pentiumiixeon_msg
    mov     ax, word ptr _cpu_signature
    shr     ax, 4
    and     eax, 0Fh                        ; isolate model
    cmp     eax, 5
    je      print_common
    mov     dx, offset pentiumiiixeon_msg
    jmp     print_common

print_celeronmodel6_type:
    cmp     eax, 6                          ; if 6 & model 6, print Intel
                                            ; Celeron processor, model 6
    jne     print_pentiumiiimodel8_type
    mov     dx, offset celeronmodel6_msg
    jmp     print_common

print_pentiumiiimodel8_type:
    cmp     eax, 8                          ; Pentium III processor, model 8, or
                                            ; Pentium III Xeon processor, model 8
    jb      print_unknown_type

    mov     eax, dword ptr _features_ebx
    or      al, al                          ; Is brand_id supported?
    jz      print_unknown_type

    mov     di, offset brand_table          ; Setup pointer to brand_id table
    mov     cx, brand_table_count           ; Get maximum entry count

next_brand:
    cmp     al, byte ptr [di]               ; Is this the brand reported by the processor
    je      brand_found

    add     di, sizeof brand_entry          ; Point to next Brand Defined
    loop    next_brand                      ; Check next brand if the table is not exhausted
    jmp     print_unknown_type

brand_found:
    mov     eax, dword ptr _cpu_signature
    cmp     eax, 06B1h                      ; Check for Pentium III, model B, stepping 1
    jne     not_b1_celeron

    mov     dx, offset celeron_brand        ; Assume this is a the special case (see Table 9)
```

```
    cmp     byte ptr[di], 3                 ; Is this a B1 Celeron?
    je      print_common

not_b1_celeron:
    cmp     eax, 0F13h
    jae     not_xeon_mp

    mov     dx, offset xeon_mp_brand        ; Early “Intel(R) Xeon(R) processor MP”?
    cmp     byte ptr [di], 0Bh
    je      print_common

    mov     dx, offset xeon_brand           ; Early “Intel(R) Xeon(R) processor”?
    cmp     byte ptr[di], 0Eh
    je      print_common

not_xeon_mp:
    mov     dx, word ptr [di+1]             ; Load DX with the offset of the brand string
    jmp     print_common

print_unknown_type:
    mov     dx, offset unknown_msg          ; if neither, print unknown

print_common:
    mov     ah, 9                           ; print string $ terminated service
    int     21h

; print family, model, and stepping
print_family:
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset signature_msg
    int     21h
    mov     eax, dword ptr _cpu_signature
    call    printDwordHex

    push    eax
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset family_msg
    int     21h
    pop     eax

    and     ah, 0Fh                         ; Check if Ext Family ID must be added
    cmp     ah, 0Fh                         ; to the Family ID to get the true 8-bit
                                            ; Family value to print.
    jne     @f

    mov     dl, ah
    ror     eax, 12                         ; Place ext family in AH
    add     ah, dl

@@:
    mov     al, ah
    ror     ah, 4
    and     ax, 0F0Fh
    add     ax, 3030h                       ; convert AH and AL to ASCII digits

    cmp     ah, ‘9’
    jl      @f
    add     ah, 7                           ; convert to ‘A’-’F’

@@:
    cmp     al, ‘9’
    jl      @f
```

```
    add     al, 7                           ; convert to ‘A’-’F’

@@:
    mov     cx, ax
    mov     ah, 2                           ; print char service
    mov     dl, ch                          ; print upper nibble Family[7:4]
    int     21h
    mov     dl, cl                          ; print lower nibble Family[3:0]
    int     21h

print_model:
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset model_msg
    int     21h

    mov     eax, dword ptr _cpu_signature

;
; If the Family_ID = 06h or Family_ID = 0Fh (Family_ID is EAX[11:8])
; then we must shift and add the Extended_Model_ID to Model_ID
;
    and     ah, 0Fh
    cmp     ah, 0Fh
    je      @f
    cmp     ah, 06h
    jne     no_ext_model

@@:
    shr     eax, 4                          ; ext model into AH[7:4], model into AL[3:0]
    and     ax, 0F00Fh
    add     ah, al

no_ext_model:
    mov     al, ah
    shr     ah, 4
    and     ax, 0F0Fh
    add     ax, 3030h                       ; convert AH and AL to ASCII digits

    cmp     ah, ‘9’
    jl      @f
    add     ah, 7                           ; convert to ‘A’-’F’

@@:
    cmp     al, ‘9’
    jl      @f
    add     al, 7                           ; convert to ‘A’-’F’

@@:
    mov     cx, ax
    mov     ah, 2                           ; print char service
    mov     dl, ch                          ; print upper nibble Model[7:4]
    int     21h
    mov     dl, cl                          ; print lower nibble Model[3:0]
    int     21h

print_stepping:
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset stepping_msg
    int     21h

    mov     edx, dword ptr _cpu_signature
    and     dl, 0Fh
```

```
    add     dl, '0'
    cmp     dl, '9'
    jle     @f
    add     dl, 7                           ; convert to 'A'-'F'

@@:
    mov     ah, 2                           ; print char service
    int     21h

    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset cr_lf
    int     21h

print_upgrade:
    mov     eax, dword ptr _cpu_signature
    test    ax, 1000h                       ; check for turbo upgrade
    jz      check_dp
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset turbo_msg
    int     21h
    jmp     print_features

check_dp:
    test    ax, 2000h                       ; check for dual processor
    jz      print_features
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset dp_msg
    int     21h

print_features:
    call    printFeatures
    call    printDCPLeaf
    call    printMwaitLeaf
    call    printArchPerfMonLeaf
    call    printExtendedTopologyLeaf
    call    printCpuExtendedStateLeaf
    jmp     end_print

not_GenuineIntel:
    mov     ah, 9                           ; print string $ terminated service
    mov     dx, offset not_intel
    int     21h

end_print:
    ret
print        endp

    end        start
```

Example 9-3.Processor Identification Procedure in C Language

```
/* Filename: CPUID3.C                                                    */
/* Copyright (c) Intel Corporation 1994-2009                             */
/*                                                                       */
/* This program has been developed by Intel Corporation.  Intel has      */
/* various intellectual property rights which it may assert under        */
/* certain circumstances, such as if another manufacturer's              */
/* processor mis-identifies itself as being "GenuineIntel" when          */
/* the CPUID instruction is executed.                                    */
/*                                                                       */
/* Intel specifically disclaims all warranties, express or implied,      */
/* and all liability, including consequential and other indirect         */
/* damages, for the use of this program, including liability for         */
/* infringement of any proprietary rights, and including the             */
/* warranties of merchantability and fitness for a particular            */
/* purpose.  Intel does not assume any responsibility for any            */
/* errors which may appear in this program nor any responsibility        */
/* to update it.                                                         */
/*                                                                       */
/*************************************************************************/
/*                                                                       */
/* This program contains three parts:                                    */
/* Part 1: Identifies CPU type in the variable _cpu_type:                */
/*                                                                       */
/* Part 2: Identifies FPU type in the variable _fpu_type:                */
/*                                                                       */
/* Part 3: Prints out the appropriate messages.                          */
/*                                                                       */
/*************************************************************************/
/*                                                                       */
/* If this code is compiled with no options specified and linked         */
/* with CPUID3A.ASM, it's assumed to correctly identify the              */
/* current Intel 8086/8088, 80286, 80386, 80486, Pentium(R),             */
/* Pentium(R) Pro, Pentium(R) II, Pentium(R) II Xeon(R),                 */
/* Pentium(R) II Overdrive(R), Intel(R) Celeron(R), Pentium(R) III,      */
/* Pentium(R) III Xeon(R), Pentium(R) 4, Intel(R) Xeon(R) DP and MP,     */
/* Intel(R) Core(TM), Intel(R) Core(TM)2, Intel(R) Core(TM) i7, and      */
/* Intel(R) Atom(TM) processors when executed in real-address mode.      */
/*                                                                       */
/* NOTE: This module is the application; CPUID3A.ASM is linked as        */
/*       a support module.                                               */
/*                                                                       */
/*************************************************************************/

#ifndef U8
typedef unsigned char         U8;
#endif
#ifndef U16
typedef unsigned short        U16;
#endif
#ifndef U32
typedef unsigned long         U32;
#endif

extern char cpu_type;
extern char fpu_type;
extern char cpuid_flag;
extern char intel_CPU;
extern U32  max_funct;
extern char vendor_id[12];
extern U32  ext_max_funct;
```

```
extern U32  cpu_signature;
extern U32  features_ecx;
extern U32  features_edx;
extern U32  features_ebx;

extern U32  funct_5_eax;
extern U32  funct_5_ebx;
extern U32  funct_5_ecx;
extern U32  funct_5_edx;

extern U32  funct_6_eax;
extern U32  funct_6_ebx;
extern U32  funct_6_ecx;
extern U32  funct_6_edx;

extern U32  funct_A_eax;
extern U32  funct_A_ebx;
extern U32  funct_A_ecx;
extern U32  funct_A_edx;

extern U32  cache_eax;
extern U32  cache_ebx;
extern U32  cache_ecx;
extern U32  cache_edx;

extern U32  dcp_cache_eax;
extern U32  dcp_cache_ebx;
extern U32  dcp_cache_ecx;
extern U32  dcp_cache_edx;

extern U32  ext_funct_1_eax;
extern U32  ext_funct_1_ebx;
extern U32  ext_funct_1_ecx;
extern U32  ext_funct_1_edx;

extern U32  ext_funct_6_eax;
extern U32  ext_funct_6_ebx;
extern U32  ext_funct_6_ecx;
extern U32  ext_funct_6_edx;

extern U32  ext_funct_7_eax;
extern U32  ext_funct_7_ebx;
extern U32  ext_funct_7_ecx;
extern U32  ext_funct_7_edx;

extern char brand_string[48];
extern int  brand_id;

long cache_temp;
long celeron_flag;
long pentiumxeon_flag;

typedef struct
{
    U32     eax;
    U32     ebx;
    U32     ecx;
    U32     edx;
} CPUID_regs;

struct brand_entry {
    U32     brand_value;
```

```
    char    *brand_string;
};

#define brand_table_count 20

struct brand_entry brand_table[brand_table_count] = {
    0x01, " Genuine Intel(R) Celeron(R) processor",
    0x02, " Genuine Intel(R) Pentium(R) III processor",
    0x03, " Genuine Intel(R) Pentium(R) III Xeon(R) processor",
    0x04, " Genuine Intel(R) Pentium(R) III processor",
    0x06, " Genuine Mobile Intel(R) Pentium(R) III Processor - M",
    0x07, " Genuine Mobile Intel(R) Celeron(R) processor",
    0x08, " Genuine Intel(R) Pentium(R) 4 processor",
    0x09, " Genuine Intel(R) Pentium(R) 4 processor",
    0x0A, " Genuine Intel(R) Celeron(R) processor",
    0x0B, " Genuine Intel(R) Xeon(R) processor",
    0x0C, " Genuine Intel(R) Xeon(R) Processor MP",
    0x0E, " Genuine Mobile Intel(R) Pentium(R) 4 Processor - M",
    0x0F, " Genuine Mobile Intel(R) Celeron(R) processor",
    0x11, " Mobile Genuine Intel(R) processor",
    0x12, " Genuine Mobile Intel(R) Celeron(R) M processor",
    0x13, " Genuine Mobile Intel(R) Celeron(R) processor",
    0x14, " Genuine Intel(R) Celeron(R) processor",
    0x15, " Mobile Genuine Intel(R) processor",
    0x16, " Genuine Intel(R) Pentium(R) M processor",
    0x17, " Genuine Mobile Intel(R) Celeron(R) processor",
};

// CPUID features documented per Software Developers Manual June 2009
char *document_msg = "\nCPUID features documented in the Intel(R) 64 and IA-32 Software Developer"
\
                     "\nManual Volume 2A Instruction Set A-M [doc #253666 on intel.com]";

struct feature_entry {
    U32     feature_mask;
    char    *feature_string;
};

// CPUID features documented per Software Developers Manual June 2009
// CPUID.1.ECX Features
char *feature_1_ecx_msg="\nCPUID.1.ECX Features: ";
#define feature_1_ecx_table_count 25
struct feature_entry feature_1_ecx_table[feature_1_ecx_table_count] = {
    0x00000001, "SSE3",                 // [0]
    0x00000002, "PCLMULQDQ",            // [1]
    0x00000004, "DTES64",               // [2]
    0x00000008, "MONITOR",              // [3]
    0x00000010, "DS-CPL",               // [4]
    0x00000020, "VMX",                  // [5]
    0x00000040, "SMX",                  // [6]
    0x00000080, "EIST",                 // [7]
    0x00000100, "TM2",                  // [8]
    0x00000200, "SSSE3",                // [9]
    0x00000400, "CNXT-ID",              // [10]
    0x00001000, "FMA",                  // [12]
    0x00002000, "CMPXCHG16B",           // [13]
    0x00004000, "XTPR",                 // [14]
    0x00008000, "PDCM",                 // [15]
    0x00040000, "DCA",                  // [18]
    0x00080000, "SSE4.1",               // [19]
    0x00100000, "SSE4.2",               // [20]
    0x00200000, "x2APIC",               // [21]
```

```
    0x00400000, “MOVBE”,                // [22]
    0x00800000, “POPCNT”,               // [23]
    0x02000000, “AES”,                  // [25]
    0x04000000, “XSAVE”,                // [26]
    0x08000000, “OSXSAVE”,              // [27]
    0x10000000, “AVX”,                  // [28]
};

// CPUID.1.EDX Features
char *feature_1_edx_msg=”\nCPUID.1.EDX Features: “;
#define feature_1_edx_table_count 29
struct feature_entry feature_1_edx_table[feature_1_edx_table_count] = {
    0x00000001, “FPU”,                  // [0]
    0x00000002, “VME”,                  // [1]
    0x00000004, “DE”,                   // [2]
    0x00000008, “PSE”,                  // [3]
    0x00000010, “TSC”,                  // [4]
    0x00000020, “MSR”,                  // [5]
    0x00000040, “PAE”,                  // [6]
    0x00000080, “MCE”,                  // [7]
    0x00000100, “CX8”,                  // [8]
    0x00000200, “APIC”,                 // [9]
    0x00000800, “SEP”,                  // [11]
    0x00001000, “MTRR”,                 // [12]
    0x00002000, “PGE”,                  // [13]
    0x00004000, “MCA”,                  // [14]
    0x00008000, “CMOV”,                 // [15]
    0x00010000, “PAT”,                  // [16]
    0x00020000, “PSE36”,                // [17]
    0x00040000, “PSN”,                  // [18]
    0x00080000, “CLFSH”,                // [19]
    0x00200000, “DS”,                   // [21]
    0x00400000, “ACPI”,                 // [22]
    0x00800000, “MMX”,                  // [23]
    0x01000000, “FXSR”,                 // [24]
    0x02000000, “SSE”,                  // [25]
    0x04000000, “SSE2”,                 // [26]
    0x08000000, “SS”,                   // [27]
    0x10000000, “HTT”,                  // [28]
    0x20000000, “TM”,                   // [29]
    0x80000000, “PBE”,                  // [31]
};

// CPUID.6.EAX Features
char *feature_6_eax_msg=”\nCPUID.6.EAX Features: “;
#define feature_6_eax_table_count 3
struct feature_entry feature_6_eax_table[feature_6_eax_table_count] = {
    0x00000001, “DIGTEMP”,              // [0]
    0x00000002, “TRBOBST”,              // [1]
    0x00000004, “ARAT”,                 // [2]
};

// CPUID.6.ECX Features
char *feature_6_ecx_msg=”\nCPUID.6.ECX Features: “;
#define feature_6_ecx_table_count 1
struct feature_entry feature_6_ecx_table[feature_6_ecx_table_count] = {
    0x00000001, “MPERF-APERF-MSR”,      // [0]
};

// CPUID.80000001h.ECX Features
char *feature_ext1_ecx_msg=”\nCPUID.80000001h.ECX Features: “;
#define feature_ext1_ecx_table_count 1
```

```
struct feature_entry feature_ext1_ecx_table[feature_ext1_ecx_table_count] = {
    0x00000001, “LAHF-SAHF”,            // [0]
};

// CPUID.80000001h.EDX Features
char *feature_ext1_edx_msg=”\nCPUID.80000001h.EDX Features: “;
#define feature_ext1_edx_table_count 4
struct feature_entry feature_ext1_edx_table[feature_ext1_edx_table_count] = {
    0x00000800, “SYSCALL”,              // [11]
    0x00100000, “XD”,                   // [20]
    0x08000000, “RDTSCP”,               // [27]
    0x20000000, “EM64T”,                // [29]
};

// CPUID.80000007h.EDX Features
char *feature_ext7_edx_msg=”\nCPUID.80000007h.EDX Features: “;
#define feature_ext7_edx_table_count 1
struct feature_entry feature_ext7_edx_table[feature_ext7_edx_table_count] = {
    0x00000100, “INVTSC”,               // [8]
};

#define archPerfMon_table_count 7

struct brand_entry archPerfMon_table[archPerfMon_table_count] = {
    0x00000001, “  Core Cycle event                 : “,
    0x00000002, “  Instruction Retired event        : “,
    0x00000004, “  Reference Cycles event           : “,
    0x00000008, “  Last-Level Cache Reference event: “,
    0x00000010, “  Last-Level Cache Misses event    : “,
    0x00000020, “  Branch Instruction Retired event: “,
    0x00000040, “  Branch Mispredict Retired event : “,
};

// extern functions
extern void    get_cpu_type();
extern void    get_fpu_type();
extern void    cpuidEx(U32 nEax, U32 nEcx, CPUID_regs* pCpuidRegs);

// forward declarations
int     print();

int main() {
    get_cpu_type();
    get_fpu_type();
    print();
    return(0);
}

//*************************************************************************
// printBinary
//   Input: nValue – value to print
//          nBits  - # of bits in value (e.g. 4=nibble, 8=byte, 16=word, 32=dword)
//*************************************************************************
void printBinary(U32 nValue, int nBits) {
    int i;
    U32 mask;

    if (nBits > 32) return;

    mask = (U32) 1 << (nBits - 1);
    for (i = 0; i < nBits; i++, mask >>= 1) {
        printf(“%c”, nValue & mask ? ‘1’ : ‘0’);
```

```
    }
    printf("b");
}

void printFeaturesTable(char *features_msg, struct feature_entry* pFeature, int nCount, U32
nFeatures) {
    int i;

    if (nFeatures == 0) return;

    printf("%s", features_msg);
    for (i = 0; i < nCount; i++, pFeature++) {
        if (nFeatures & pFeature->feature_mask) {
            printf("%s ", pFeature->feature_string);
        }
    }
}

void printFeatures() {
    printFeaturesTable(feature_1_ecx_msg, &feature_1_ecx_table[0], feature_1_ecx_table_count,
features_ecx);
    // Fast System Call had a different meaning on some processors,
    // so check CPU signature.
    if (features_edx & ((U32) 1 << 11)) {
        if ((cpu_type == 6) && ((cpu_signature & 0xff) < 0x33))
            features_edx &= ~((U32) 1 << 11);
    }
    // HTT can be fused off even when feature flag reports HTT supported,
    // so perform additional checks.
    if (features_edx & ((U32) 1 << 28)) {
        if ((((features_ebx >> 16) & 0x0FF) / (((dcp_cache_eax >> 26) & 0x3F) + 1) < 2))
            features_edx &= ~((U32) 1 << 28);
    }
    printFeaturesTable(feature_1_edx_msg, &feature_1_edx_table[0], feature_1_edx_table_count,
features_edx);
    printFeaturesTable(feature_6_eax_msg, &feature_6_eax_table[0], feature_6_eax_table_count,
funct_6_eax);
    printFeaturesTable(feature_6_ecx_msg, &feature_6_ecx_table[0], feature_6_ecx_table_count,
funct_6_ecx);
    printFeaturesTable(feature_ext1_ecx_msg, &feature_ext1_ecx_table[0],
feature_ext1_ecx_table_count, ext_funct_1_ecx);
    printFeaturesTable(feature_ext1_edx_msg, &feature_ext1_edx_table[0],
feature_ext1_edx_table_count, ext_funct_1_edx);
    printFeaturesTable(feature_ext7_edx_msg, &feature_ext7_edx_table[0],
feature_ext7_edx_table_count, ext_funct_7_edx);
}

void printDCPLeaf() {
    CPUID_regs regs;
    U32        cacheSize;
    U32        ecxIndex = 0;

    if (max_funct < 0x4) return;

    cpuidEx(0x4, ecxIndex, &regs);
    if ((regs.eax & 0x1F) == 0) return;
    printf("\n\nCPUID.4 Deterministic Cache Parameters (DCP) Leaf:");

    while (1) {
        cpuidEx(0x4, ecxIndex, &regs);
        if ((regs.eax & 0x1F) == 0) break;
        ecxIndex++;

        switch (regs.eax & 0x1F) {
```

```
            case 1: printf("\n         Data Cache; "); break;
            case 2: printf("\n  Instruction Cache; "); break;
            case 3: printf("\n      Unified Cache; "); break;
            default: continue;
        }
        printf("Level %d; ", (regs.eax >> 5) & 0x7);
        //          Sets           LineSize                 Partitions                      Ways
        cacheSize = (regs.ecx+1) * ((regs.ebx & 0xFFF)+1) * (((regs.ebx >> 12) & 0x3FF)+1) *
((regs.ebx >> 22)+1);
        printf("Size %lu KB", cacheSize >> 10);
    }
}

void printMwaitLeaf() {
    int i;

    if (max_funct < 0x5) return;

    printf("\n\nCPUID.5 Monitor/MWAIT Leaf:");
    printf("\n  Smallest monitor line size: %d bytes", funct_5_eax & 0xFFFF);
    printf("\n  Largest monitor line size: %d bytes", funct_5_ebx & 0xFFFF);
    printf("\n  Enumeration of Monitor-MWAIT extensions: %s", (funct_5_ecx & 0x1) ? "Supported" :
"Unsupported");
    printf("\n  Interrupts as break-event for MWAIT: %s", (funct_5_ecx & 0x2) ? "Supported" :
"Unsupported");
    for (i = 0; i < 5; i++) {
        printf("\n  Number of C%d sub C-States supported using MWAIT: %d", i, (funct_5_edx >>
(i*4)) & 0xF);
    }
}

void printArchPerfMonLeaf() {
    int i;

    if (max_funct < 0xA) return;

    printf("\n\nCPUID.A Architecture Performance Monitoring Leaf:");
    printf("\n  Version ID: %u", funct_A_eax & 0xFF);
    printf("\n  Number of General Purpose Counters per Logical Processor: %u", (funct_A_eax >> 8)
& 0xFF);
    printf("\n  Bit Width of General Purpose Counters: %u", (funct_A_eax >> 16) & 0xFF);
    printf("\n  Length of EBX bit vector to enumerate events: %u", (funct_A_eax >> 24) & 0xFF);

    for (i = 0; i < (funct_A_eax >> 24) & 0xFF; i++) {
        printf("\n%s%sAvailable", archPerfMon_table[i].brand_string,
(archPerfMon_table[i].brand_value & funct_A_ebx) ? "Not " : "");
    }

    if ((funct_A_eax & 0xFF) > 1) {
        printf("\n  Number of Fixed-Function Performance Counters: %u", funct_A_edx & 0x1F);
        printf("\n  Bit Width of Fixed-Function Performance Counters: %u", (funct_A_edx >> 5) &
0xFF);
    }
}

void printExtendedTopologyLeaf() {
    CPUID_regs regs;
    U32        ecxIndex = 0;

    if (max_funct < 0xB) return;

    while (1) {
        cpuidEx(0xB, ecxIndex, &regs);
```

```
        if (regs.eax == 0 && regs.ebx == 0) break;
        if (ecxIndex == 0) printf("\n");
        printf("\nCPUID.B Extended Topology Leaf n=%d:", ecxIndex);
        ecxIndex++;

        printf("\n  x2APIC ID bits to shift right to get unique topology ID: %d", regs.eax &
0xFFFF);
        printf("\n  Logical processors at this level type: %d", regs.ebx & 0xFFFF);
        printf("\n  Level Number: %d ", regs.ecx & 0xFF);
        switch (regs.ecx & 0xFF) {
            case 0: printf("Thread"); break;
            case 1: printf("Core"); break;
            default: printf("Package"); break;
        }
        printf("\n  Level Type  : %d ", (regs.ecx >> 8) & 0xFF);
        switch ((regs.ecx >> 8) & 0xFF) {
            case 0: printf("Invalid"); break;
            case 1: printf("SMT"); break;
            case 2: printf("Core"); break;
            default: printf("Reserved"); break;
        }
        printf("\n  x2APIC ID   : %lu", regs.edx);
    }
}

void printCpuExtendedStateLeaf() {
    CPUID_regs regs;
    U32        ecxIndex = 0;

    if (max_funct < 0xD) return;

    cpuidEx(0xD, ecxIndex, &regs);
    printf("\n\nCPUID.D Processor Extended State Enumeration Main Leaf n=0:");
    printf("\n  Valid bit fields of XCR0[31: 0]: ");
    printBinary(regs.eax, 32);
    printf("\n  Valid bit fields of XCR0[63:32]: ");
    printBinary(regs.edx, 32);
    printf("\n  Max size required by enabled features in XCR0: %lu bytes", regs.ebx);
    printf("\n  Max size required by XSAVE/XRSTOR for supported features: %lu bytes", regs.ecx);

    ecxIndex = 2;
    while (1) {
        cpuidEx(0xD, ecxIndex, &regs);
        if (regs.eax == 0) break;

        printf("\nCPUID.D Processor Extended State Enumeration Sub-Leaf n=%lu", ecxIndex);
        printf("\n  Size required for feature associated with sub-leaf: %lu bytes", regs.eax);
        printf("\n  Offset of save area from start of XSAVE/XRSTOR area: %lu", regs.ebx);
        ecxIndex++;
    }
}

int print() {
    int  brand_index = 0;
    int  family = 0;
    int  model = 0;

    printf("\nThis processor:");
    if (cpuid_flag == 0) {
        switch (cpu_type) {
        case 0:
            printf("n 8086/8088 processor");
```

```
        if (fpu_type) printf(“ and an 8087 math coprocessor”);
        break;
    case 2:
        printf(“n 80286 processor”);
        if (fpu_type) printf(“ and an 80287 math coprocessor”);
        break;
    case 3:
        printf(“n 80386 processor”);
        if (fpu_type == 2)
            printf(“ and an 80287 math coprocessor”);
        else if (fpu_type)
            printf(“ and an 80387 math coprocessor”);
        break;
    case 4:
        if (fpu_type)
            printf(“n 80486DX, 80486DX2 processor or 80487SX math coprocessor”);
        else
            printf(“n 80486SX processor”);
        break;
    default:
        printf(“n unknown processor”);
    }
} else {
// using cpuid instruction
    if (intel_CPU) {
        if (brand_string[0]) {
            brand_index = 0;
            while ((brand_string[brand_index] == ‘ ‘) && (brand_index < 48))
                brand_index++;
            if (brand_index != 48)
                printf(“ %s”, &brand_string[brand_index]);
        }
        else if (cpu_type == 4) {
            switch ((cpu_signature>>4) & 0xf) {
            case  0:
            case  1:
                printf(“ Genuine Intel486(TM) DX processor”);
                break;
            case  2:
                printf(“ Genuine Intel486(TM) SX processor”);
                break;
            case  3:
                printf(“ Genuine IntelDX2(TM) processor”);
                break;
            case  4:
                printf(“ Genuine Intel486(TM) processor”);
                break;
            case  5:
                printf(“ Genuine IntelSX2(TM) processor”);
                break;
            case  7:
                printf(“ Genuine Write-Back Enhanced \
                    IntelDX2(TM) processor”);
                break;
            case  8:
                printf(“ Genuine IntelDX4(TM) processor”);
                break;
            default:
                printf(“ Genuine Intel486(TM) processor”);
            }
        }
        else if (cpu_type == 5)
```

```
    printf(“ Genuine Intel Pentium(R) processor”);
else if ((cpu_type == 6) && (((cpu_signature >> 4) & 0xf) == 1))
    printf(“ Genuine Intel Pentium(R) Pro processor”);
else if ((cpu_type == 6) && (((cpu_signature >> 4) & 0xf) == 3))
    printf(“ Genuine Intel Pentium(R) II processor, model 3”);
else if (((cpu_type == 6) && (((cpu_signature >> 4) & 0xf) == 5)) ||
    ((cpu_type == 6) && (((cpu_signature >> 4) & 0xf) == 7)))
{
    celeron_flag = 0;
    pentiumxeon_flag = 0;
    cache_temp = cache_eax & 0xFF000000;
    if (cache_temp == 0x40000000)
        celeron_flag = 1;
    if ((cache_temp >= 0x44000000) && (cache_temp <= 0x45000000))
        pentiumxeon_flag = 1;

    cache_temp = cache_eax & 0xFF0000;
    if (cache_temp == 0x400000)
        celeron_flag = 1;
    if ((cache_temp >= 0x440000) && (cache_temp <= 0x450000))
        pentiumxeon_flag = 1;

    cache_temp = cache_eax & 0xFF00;
    if (cache_temp == 0x4000)
        celeron_flag = 1;

    if ((cache_temp >= 0x4400) && (cache_temp <= 0x4500))
        pentiumxeon_flag = 1;

    cache_temp = cache_ebx & 0xFF000000;
    if (cache_temp == 0x40000000)
        celeron_flag = 1;
    if ((cache_temp >= 0x44000000) && (cache_temp <=0x45000000))
        pentiumxeon_flag = 1;

    cache_temp = cache_ebx & 0xFF0000;
    if (cache_temp == 0x400000)
        celeron_flag = 1;
    if ((cache_temp >= 0x440000) && (cache_temp <= 0x450000))
        pentiumxeon_flag = 1;

    cache_temp = cache_ebx & 0xFF00;
    if (cache_temp == 0x4000)
        celeron_flag = 1;
    if ((cache_temp >= 0x4400) && (cache_temp <= 0x4500))
        pentiumxeon_flag = 1;

    cache_temp = cache_ebx & 0xFF;
    if (cache_temp == 0x40)
        celeron_flag = 1;
    if ((cache_temp >= 0x44) && (cache_temp <= 0x45))
        pentiumxeon_flag = 1;

    cache_temp = cache_ecx & 0xFF000000;
    if (cache_temp == 0x40000000)
        celeron_flag = 1;
    if ((cache_temp >= 0x44000000) && (cache_temp <= 0x45000000))
        pentiumxeon_flag = 1;

    cache_temp = cache_ecx & 0xFF0000;
    if (cache_temp == 0x400000)
        celeron_flag = 1;
```

```
    if ((cache_temp >= 0x440000) && (cache_temp <= 0x450000))
        pentiumxeon_flag = 1;

    cache_temp = cache_ecx & 0xFF00;
    if (cache_temp == 0x4000)
        celeron_flag = 1;
    if ((cache_temp >= 0x4400) && (cache_temp <= 0x4500))
        pentiumxeon_flag = 1;

    cache_temp = cache_ecx & 0xFF;
    if (cache_temp == 0x40)
        celeron_flag = 1;
    if ((cache_temp >= 0x44) && (cache_temp <= 0x45))
        pentiumxeon_flag = 1;

    cache_temp = cache_edx & 0xFF000000;
    if (cache_temp == 0x40000000)
        celeron_flag = 1;
    if ((cache_temp >= 0x44000000) && (cache_temp <= 0x45000000))
        pentiumxeon_flag = 1;

    cache_temp = cache_edx & 0xFF0000;
    if (cache_temp == 0x400000)
        celeron_flag = 1;
    if ((cache_temp >= 0x440000) && (cache_temp <= 0x450000))
        pentiumxeon_flag = 1;

    cache_temp = cache_edx & 0xFF00;
    if (cache_temp == 0x4000)
        celeron_flag = 1;
    if ((cache_temp >= 0x4400) && (cache_temp <= 0x4500))
        pentiumxeon_flag = 1;

    cache_temp = cache_edx & 0xFF;
    if (cache_temp == 0x40)
        celeron_flag = 1;
    if ((cache_temp >= 0x44) && (cache_temp <= 0x45))
        pentiumxeon_flag = 1;

    if (celeron_flag == 1) {
        printf(" Genuine Intel Celeron(R) processor, model 5");
    } else {
        if (pentiumxeon_flag == 1) {
            if (((cpu_signature >> 4) & 0x0f) == 5)
                printf(" Genuine Intel Pentium(R) II Xeon(R) processor");
            else
                printf(" Genuine Intel Pentium(R) III Xeon(R) processor,");
            printf(" model 7");
        } else {
            if (((cpu_signature >> 4) & 0x0f) == 5) {
                printf(" Genuine Intel Pentium(R) II processor, model 5 ");
                printf("or Intel Pentium(R) II Xeon(R) processor");
            } else {
                printf(" Genuine Intel Pentium(R) III processor, model 7");
                printf(" or Intel Pentium(R) III Xeon(R) processor,");
                printf(" model 7");
            }
        }
    }
}
else if ((cpu_type == 6) && (((cpu_signature >> 4) & 0xf) == 6))
    printf(" Genuine Intel Celeron(R) processor, model 6");
```

```
            else if ((features_ebx & 0xff) != 0) {
                while ((brand_index < brand_table_count) &&
                    ((features_ebx & 0xff) != brand_table[brand_index].brand_value))
                    brand_index++;
                if (brand_index < brand_table_count) {
                    if ((cpu_signature == 0x6B1) &&
                        (brand_table[brand_index].brand_value == 0x3))
                        printf(" Genuine Intel(R) Celeron(R) processor");
                    else if ((cpu_signature < 0xF13) &&
                        (brand_table[brand_index].brand_value == 0x0B))
                        printf(" Genuine Intel(R) Xeon(R) processor MP");
                    else if ((cpu_signature < 0xF13) &&
                        (brand_table[brand_index].brand_value == 0x0E))
                        printf(" Genuine Intel(R) Xeon(R) processor");
                    else
                        printf("%s", brand_table[brand_index].brand_string);
                } else {
                    printf("n unknown Genuine Intel processor");
                }
            } else {
                printf("n unknown Genuine Intel processor");
            }

            printf("\nProcessor Signature / Version Information: %08lXh", cpu_signature);

            if (cpu_type == 0x0f) {
                family = (int)((cpu_signature >> 20) & 0x0ff);
            }

            if ((cpu_type == 0x0f) || (cpu_type == 0x06)) {
                model = (int)((cpu_signature >> 12) & 0x0f0);
            }

            printf("\nProcessor Family: %02X", (family + (int)cpu_type));
            printf("\nModel:            %02X", (model + (int)((cpu_signature>>4)&0xf)));
            printf("\nStepping:         %X\n", (int)(cpu_signature&0xf));

            if (cpu_signature & 0x1000)
                printf("\nThe processor is an OverDrive(R) processor");
            else if (cpu_signature & 0x2000)
                printf("\nThe processor is the upgrade processor in a dual processor system");

            printf("%s\n", document_msg);
            printFeatures();
            printDCPLeaf();
            printMwaitLeaf();
            printArchPerfMonLeaf();
            printExtendedTopologyLeaf();
            printCpuExtendedStateLeaf();
        } else {
            printf("t least an 80486 processor. ");
            printf("\nIt does not contain a Genuine Intel part and as a result, the ");
            printf("\nCPUID detection information cannot be determined.");
        }
    }

    return(0);
}
```

Example 9-4.Detecting Denormals-Are-Zero Support

```
; Filename: DAZDTECT.ASM
; Copyright (c) Intel Corporation 2001-2009
;
; This program has been developed by Intel Corporation.  Intel
; has various intellectual property rights which it may assert
; under certain circumstances, such as if another
; manufacturer's processor mis-identifies itself as being
; "GenuineIntel" when the CPUID instruction is executed.
;
; Intel specifically disclaims all warranties, express or
; implied, and all liability, including consequential and other
; indirect damages, for the use of this program, including
; liability for infringement of any proprietary rights,
; and including the warranties of merchantability and fitness
; for a particular purpose.  Intel does not assume any
; responsibility for any errors which may appear in this program
; nor any responsibility to update it.
;
; This example assumes the system has booted DOS.
; This program runs in real mode.
;
;*********************************************************************
;
; This program performs the following steps to determine if the
; processor supports the SSE/SSE2 DAZ mode.
;
; Step 1. Execute the CPUID instruction with an input value of EAX=0 and
;         ensure the vendor-ID string returned is "GenuineIntel".
;
; Step 2. Execute the CPUID instruction with EAX=1. This will load the
;         EDX register with the feature flags.
;
; Step 3. Ensure that the FXSR feature flag (EDX bit 24) is set. This
;         indicates the processor supports the FXSAVE and FXRSTOR
;         instructions.
;
; Step 4. Ensure that the SSE feature flag (EDX bit 25) or the SSE2 feature
;         flag (EDX bit 26) is set. This indicates that the processor supports
;         at least one of the SSE/SSE2 instruction sets and its MXCSR control
;         register.
;
; Step 5. Zero a 16-byte aligned, 512-byte area of memory.
;         This is necessary since some implementations of FXSAVE do not
;         modify reserved areas within the image.
;
; Step 6. Execute an FXSAVE into the cleared area.
;
; Step 7. Bytes 28-31 of the FXSAVE image are defined to contain the
;         MXCSR_MASK.  If this value is 0, then the processor's MXCSR_MASK
;         is 0xFFBF, otherwise MXCSR_MASK is the value of this dword.
;
; Step 8. If bit 6 of the MXCSR_MASK is set, then DAZ is supported.
;
;*********************************************************************

.DOSSEG
.MODEL small, c
.STACK

.DATA
```

```
buffer                  db 512+16 DUP (0)

not_intel               db “This is not an Genuine Intel processor.”,13,10,”$”
noSSEorSSE2             db “Neither SSE or SSE2 extensions are supported.”,13,10,”$”
no_FXSAVE               db “FXSAVE not supported.”,13,10,”$”
daz_mask_clear          db “DAZ bit in MXCSR_MASK is zero (clear).”,13,10,”$”
no_daz                  db “DAZ mode not supported.”,13,10,”$”
supports_daz            db “DAZ mode supported.”,13,10,”$”

.CODE
.686p
.XMM

dazdtect PROC NEAR
.STARTUP            ; Allow assembler to create code that
                    ; initializes stack and data segment
                    ; registers

; Step 1.
; Verify Genuine Intel processor by checking CPUID generated vendor ID
    mov     eax, 0
    cpuid
    cmp     ebx, ‘uneG’                     ; Compare first 4 letters of Vendor ID
    jne     notIntelprocessor               ; Jump if not Genuine Intel processor
    cmp     edx, ‘Ieni’                     ; Compare next 4 letters of Vendor ID
    jne     notIntelprocessor               ; Jump if not Genuine Intel processor
    cmp     ecx, ‘letn’                     ; Compare last 4 letters of Vendor ID
    jne     notIntelprocessor               ; Jump if not Genuine Intel processor

; Step 2. Get CPU feature flags
; Step 3. Verify FXSAVE and either SSE or
; Step 4. SSE2 are supported
    mov     eax, 1
    cpuid
    bt      edx, 24                         ; Feature Flags Bit 24 is FXSAVE support
    jnc     noFxsave                        ; jump if FXSAVE not supported
    bt      edx, 25                         ; Feature Flags Bit 25 is SSE support
    jc      sse_or_sse2_supported           ; jump if SSE is not supported
    bt      edx, 26                         ; Feature Flags Bit 26 is SSE2 support
    jnc     no_sse_sse2                     ; jump if SSE2 is not supported

sse_or_sse2_supported:
    ; FXSAVE requires a 16-byte aligned buffer so get offset into buffer
    mov     bx, offset buffer               ; Get offset of the buffer into bx
    and     bx, 0FFF0h
    add     bx, 16                          ; DI is aligned at 16-byte boundary

; Step 5. Clear the buffer that will be used for FXSAVE data
    push    ds
    pop     es
    mov     di,bx
    xor     ax, ax
    mov     cx, 512/2
    cld
    rep     stosw                           ; Fill at FXSAVE buffer with zeroes

; Step 6.
    fxsave  [bx]

; Step 7.
    mov     eax, DWORD PTR [bx][28t]        ; Get MXCSR_MASK
    cmp     eax, 0                          ; Check for valid mask
```


```
    jne     check_mxcsr_mask
    mov     eax, 0FFBFh                     ; Force use of default MXCSR_MASK

check_mxcsr_mask:
; EAX contains MXCSR_MASK from FXSAVE buffer or default mask

; Step 8.
    bt      eax, 6                          ; MXCSR_MASK Bit 6 is DAZ support
    jc      supported                       ; Jump if DAZ supported

    mov     dx, offset daz_mask_clear
    jmp     notSupported

supported:
    mov     dx, offset supports_daz         ; Indicate DAZ is supported.
    jmp     print

notIntelProcessor:
    mov     dx, offset not_intel            ; Assume not an Intel processor
    jmp     print

no_sse_sse2:
    mov     dx, offset noSSEorSSE2          ; Setup error message assuming no SSE/SSE2
    jmp     notSupported

noFxsave:
    mov     dx, offset no_FXSAVE

notSupported:
    mov     ah, 9                           ; Execute DOS print string function
    int     21h

    mov     dx, offset no_daz

print:
    mov     ah, 9                           ; Execute DOS print string function
    int     21h

exit:
    .EXIT           ; Allow assembler to generate code
                    ; that returns control to DOS
    ret

dazdtect    ENDP

    end
```

Example 9-5.Frequency Detection Procedure

```
; Filename: FREQUENC.ASM
; Copyright(c) 2003 - 2009 by Intel Corporation
;
; This program has been developed by Intel Corporation.
;
; Intel specifically disclaims all warranties, express or
; implied, and all liability, including consequential and other
; indirect damages, for the use of this program, including
; liability for infringement of any proprietary rights,
; and including the warranties of merchantability and fitness
; for a particular purpose. Intel does not assume any
; responsibility for any errors which may appear in this program
; nor any responsibility to update it.
;
;*****************************************************************
;
; This program performs the following steps to determine the
; processor frequency.
;
; Step  1. Execute the CPUID instruction with EAX=0 and ensure
;          the Vendor ID string returned is “GenuineIntel”.
; Step  2. Execute the CPUID instruction with EAX=1 to load EDX
;          with the feature flags.
; Step  3. Ensure that the TSC feature flag (EDX bit 4) is set.
;          This indicates the processor supports the Time Stamp
;          Counter and RDTSC instruction.
; Step  4. Verify that CPUID with EAX=6 is supported by checking
;          the maximum CPUID input returned with EAX=0 in EAX.
; Step  5. Execute the CPUID instruction with EAX=6 to load ECX
;          with the feature flags.
; Step  6. Ensure that the MPERF/APERF feature flag (ECX bit 0)
;          is set. This indicates the processor supports the
;          MPERF and APERF MSRs.
; Step  7. Zero the MPERF and APERF MSRs.
; Step  8. Read the TSC at the start of the reference period.
; Step  9. Read the MPERF and APERF MSRs at the end of the
;          reference period.
; Step 10. Read the TSC at the end of the reference period.
; Step 11. Compute the TSC delta from the start and end of the
;          reference period.
; Step 12. Compute the actual frequency by dividing the TSC
;          delta by the reference period.
; Step 13. Compute the MPERF and APERF frequency.
; Step 14. Compute the MPERF and APERF percentage frequency.
;
;*****************************************************************
;
; This program has been compiled with Microsoft Macro Assembler
; 6.15 with no options specified and linked with CPUFREQ.C and
; CPUID3A.ASM, and executes in real-address mode.
;
; NOTE: CPUFREQ.C is the application; FREQUENC.ASM and
;       CPUID3A.ASM are linked as support modules.
;
;*****************************************************************

TITLE FREQUENC

.DOSSEG

.MODEL small
```


```
SEG_BIOS_DATA_AREA      equ 40h
OFFSET_TICK_COUNT       equ 6Ch
INTERVAL_IN_TICKS       equ 10

IA32_MPERF_MSR          equ 0E7h
IA32_APERF_MSR          equ 0E8h

TSC_BIT                 equ 4               ; CPUID.1.EDX feature

.DATA

.CODE
.686p

_frequency       PROC NEAR PUBLIC
    push    bp
    mov     bp, sp                          ; save the stack pointer
    sub     sp, 28h                         ; create space on stack for local variables

    ; Using a local stack frame to simplify the changes to this routine
    addr_freq_in_mhz         TEXTEQU <word ptr [bp+4]>
    addr_mperf_freq          TEXTEQU <word ptr [bp+6]>
    addr_aperf_freq          TEXTEQU <word ptr [bp+8]>
    addr_aperf_mperf_percent TEXTEQU <word ptr [bp+10]>
    addr_mperf_tsc_percent   TEXTEQU <word ptr [bp+12]>
    addr_aperf_tsc_percent   TEXTEQU <word ptr [bp+14]>

    mperf_lo                 TEXTEQU <dword ptr [bp-4]>    ; dword local variable
    mperf_hi                 TEXTEQU <dword ptr [bp-8]>    ; dword local variable
    aperf_lo                 TEXTEQU <dword ptr [bp-12]>   ; dword local variable
    aperf_hi                 TEXTEQU <dword ptr [bp-16]>   ; dword local variable
    perf_msr_avail           TEXTEQU <dword ptr [bp-20]>   ; dword local variable
    tscDeltaLo               TEXTEQU <dword ptr [bp-24]>   ; dword local variable
    tscDeltaHi               TEXTEQU <dword ptr [bp-28]>   ; dword local variable
    tscLoDword               TEXTEQU <dword ptr [bp-32]>   ; dword local variable
    tscHiDword               TEXTEQU <dword ptr [bp-36]>   ; dword local variable
    maxCpuidInput            TEXTEQU <dword ptr [bp-40]>   ; dword local variable

    pushad
    push    es

; Step 1. Verify Genuine Intel processor by checking CPUID generated vendor ID
    xor     eax, eax
    cpuid
    cmp     ebx, 'uneG'                     ; Compare first 4 letters of Vendor ID
    jne     exit                            ; Jump if not Genuine Intel processor
    cmp     edx, 'Ieni'                     ; Compare next 4 letters of Vendor ID
    jne     exit                            ; Jump if not Genuine Intel processor
    cmp     ecx, 'letn'                     ; Compare last 4 letters of Vendor ID
    jne     exit                            ; Jump if not Genuine Intel processor

    mov     maxCpuidInput, eax              ; Save maximum CPUID input

; Step 2. Get CPU feature flags.
    mov     eax, 1
    cpuid
; Step 3. Verify TSC is supported.
    bt      edx, TSC_BIT                    ; Flags Bit 4 is TSC support
    jnc     exit                            ; jump if TSC not supported

    xor     eax, eax                        ; Initialize variables
```

```
    mov     perf_msr_avail, eax             ; Assume MPERF and APERF MSRs aren't available
    mov     bx, word ptr addr_mperf_freq
    mov     word ptr [bx], ax
    mov     bx, word ptr addr_aperf_freq
    mov     word ptr [bx], ax

; Step  4. Verify that CPUID with EAX=6 is supported.
    cmp     maxCpuidInput, 6                ; Is Power Management Parameters leaf supported?
    jb      @f                              ; Jump if not supported

; Step  5. Execute the CPUID instruction with EAX=6.
    mov     eax, 6                          ; Setup for Power Management Parameters leaf
    cpuid
; Step  6. Ensure that the MPERF/APERF feature flag (ECX bit 0) is set.
    bt      ecx, 0                          ; Check for MPERF/APERF MSR support
    jnc     @f                              ; Jump if not supported
    mov     perf_msr_avail, 1               ; MPERF and APERF MSRs are available

@@:
    push    SEG_BIOS_DATA_AREA
    pop     es
    mov     si, OFFSET_TICK_COUNT           ; The BIOS tick count updates ~18.2
    mov     ebx, dword ptr es:[si]          ;  times per second.

wait_for_new_tick:
    cmp     ebx, dword ptr es:[si]          ; Wait for tick count change
    je      wait_for_new_tick

; Step  7. Zero the MPERF and APERF MSRs.
    cmp     perf_msr_avail, 1
    jne     @f

    xor     eax, eax
    xor     edx, edx
    mov     ecx, IA32_MPERF_MSR
    wrmsr
    mov     ecx, IA32_APERF_MSR
    wrmsr

; Step  8. Read the TSC at the start of the reference period.
@@:
    ; Read CPU time stamp
    rdtsc                                   ; Read and save TSC immediately
    mov     tscLoDword, eax                 ;  after a tick
    mov     tscHiDword, edx

    add     ebx, INTERVAL_IN_TICKS + 1      ; Set time delay value ticks.

wait_for_elapsed_ticks:
    cmp     ebx, dword ptr es:[si]          ; Have we hit the delay?
    jne     wait_for_elapsed_ticks

; Step  9. Read the MPERF and APERF MSRs at the end of the reference period.
    cmp     perf_msr_avail, 1
    jne     @f

    mov     ecx, IA32_MPERF_MSR
    rdmsr
    mov     mperf_lo, eax
    mov     mperf_hi, edx
    mov     ecx, IA32_APERF_MSR
    rdmsr
```


```
    mov     aperf_lo, eax
    mov     aperf_hi, edx

; Step 10. Read the TSC at the end of the reference period.
@@:
    ; Read CPU time stamp immediately after tick delay reached.
    rdtsc

; Step 11. Compute the TSC delta from the start and end of the reference period.
    sub     eax, tscLoDword                 ; Calculate TSC delta from
    sbb     edx, tscHiDword                 ;  start to end of interval

    mov     tscDeltaLo, eax
    mov     tscDeltaHi, edx

; Step 12. Compute the actual frequency from TSC.
    ; 54945 = (1 / 18.2) * 1,000,000  This adjusts for MHz.
    ; 54945*INTERVAL_IN_TICKS adjusts for number of ticks in interval
    mov     ebx, 54945*INTERVAL_IN_TICKS
    div     ebx

    ; ax contains measured speed in MHz
    mov     bx, word ptr addr_freq_in_mhz
    mov     word ptr [bx], ax

    cmp     perf_msr_avail, 1
    jne     @f

    mov     eax, mperf_lo
    mov     edx, mperf_hi
    mov     ebx, 54945*INTERVAL_IN_TICKS
    div     ebx

; Step 13. Compute the MPERF and APERF frequency.
    ; ax contains measured speed in MHz
    mov     bx, word ptr addr_mperf_freq
    mov     word ptr [bx], ax

    mov     eax, aperf_lo
    mov     edx, aperf_hi

    mov     ebx, 54945*INTERVAL_IN_TICKS
    div     ebx

    ; ax contains measured speed in MHz
    mov     bx, word ptr addr_aperf_freq
    mov     word ptr [bx], ax

; Step 14. Compute the MPERF and APERF percentage frequency.
    mov     eax, aperf_lo
    mov     edx, aperf_hi
    mov     ebx, 100
    mul     ebx
    mov     ebx, mperf_lo
    div     ebx

    ; ax contains measured percentage AMCT/mperf
    mov     bx, word ptr addr_aperf_mperf_percent
    mov     word ptr [bx], ax

    mov     eax, aperf_lo
    mov     edx, aperf_hi
```

```
    mov     ebx, 100
    mul     ebx
    mov     ebx, tscDeltaLo
    div     ebx
    movzx   eax, ax

    ; ax contains measured percentage aperf/TSC * 100%
    mov     bx, word ptr addr_aperf_tsc_percent
    mov     word ptr [bx], ax

    mov     eax, mperf_lo
    mov     edx, mperf_hi
    mov     ebx, 100
    mul     ebx
    mov     ebx, tscDeltaLo
    div     ebx
    movzx   eax, ax

    ; ax contains measured percentage mperf/TSC * 100%
    mov     bx, word ptr addr_mperf_tsc_percent
    mov     word ptr [bx], ax

exit:
@@:
    pop     es
    popad
    mov     sp, bp
    pop     bp
    ret
_frequency       ENDP

    end
```

Example 9-6.Frequency Detection in C Language

```
/* Filename: CPUFREQ.C                                                  */
/* Copyright (c) Intel Corporation 2008-2009                            */
/*                                                                      */
/* This program has been developed by Intel Corporation.  Intel has     */
/* various intellectual property rights which it may assert under       */
/* certain circumstances, such as if another manufacturer's             */
/* processor mis-identifies itself as being "GenuineIntel" when         */
/* the CPUID instruction is executed.                                   */
/*                                                                      */
/* Intel specifically disclaims all warranties, express or implied,     */
/* and all liability, including consequential and other indirect        */
/* damages, for the use of this program, including liability for        */
/* infringement of any proprietary rights, and including the            */
/* warranties of merchantability and fitness for a particular           */
/* purpose.  Intel does not assume any responsibility for any           */
/* errors which may appear in this program nor any responsibility       */
/* to update it.                                                        */
/*                                                                      */
/************************************************************************/
/*                                                                      */
/* This program performs the following steps to determine the           */
/* processor actual frequency.                                          */
/*                                                                      */
/* Step 1. Call get_cpu_type() to get brand string.                     */
/* Step 2. Parse brand string looking for "xxxxyHz" or "x.xxyHz"        */
/*         for processor frequency, per Software Developer Manual       */
/*         Volume 2A, CPUID instruction, Figure "Algorithm for          */
/*         Extracting Maximum Processor Frequency".                     */
/* Step 3. Call frequency() to get frequency from multiple methods.     */
/*                                                                      */
/************************************************************************/
/*                                                                      */
/* NOTE: CPUFREQ.C is the application; FREQUENC.ASM and                 */
/*       CPUID3A.ASM are linked as support modules.                     */
/*                                                                      */
/************************************************************************/

#include <string.h>

#ifndef U8
typedef unsigned char         U8;
#endif
#ifndef U16
typedef unsigned short        U16;
#endif
#ifndef U32
typedef unsigned long         U32;
#endif

// extern variables
extern char brand_string[48];
extern long ext_max_funct;

// extern functions
extern void get_cpu_type();
extern void frequency(U16* pFreqMhz, U16* pFreqMperf, U16* pFreqAperf,
            U16* pFreqAperfMperfPercent, U16* pMperfTscPercent, U16* pAperfTscPercent);

U32 GetFrequencyFromBrandString(char *pFreqBuffer) {
    U32 multiplier = 0;
    U32 frequency = 0;
```

```
    U32 index = 0;

    get_cpu_type();

    pFreqBuffer[0] = 0; // empty string by setting 1st char to NULL

    // Verify CPUID brand string function is supported
    if (ext_max_funct < 0x80000004)
        return frequency;

    // -2 to prevent buffer overrun because looking for y in yHz, so z is +2 from y
    for (index=0; index<48-2; index++) {
        // format is either “x.xxyHz” or “xxxxyHz”, where y=M,G,T and x is digits
        // Search brand string for “yHz” where y is M, G, or T
        // Set multiplier so frequency is in MHz
        if (brand_string[index+1] == ‘H’ && brand_string[index+2] == ‘z’) {
            if (brand_string[index] == ‘M’)
                multiplier = 1;
            else if (brand_string[index] == ‘G’)
                multiplier = 1000;
            else if (brand_string[index] == ‘T’)
                multiplier = 1000000;
        }

        if (multiplier > 0) {
            // Copy 7 characters (length of “x.xxyHz”)
            // index is at position of y in “x.xxyHz”
            strncpy(pFreqBuffer, &brand_string[index-4], 7);
            pFreqBuffer[7] = 0; // null terminate the string

            // Compute frequency (in MHz) from brand string
            if (brand_string[index-3] == ‘.’) { // If format is “x.xx”
                frequency  = (U32)(brand_string[index-4] - ‘0’) * multiplier;
                frequency += (U32)(brand_string[index-2] - ‘0’) * (multiplier / 10);
                frequency += (U32)(brand_string[index-1] - ‘0’) * (multiplier / 100);
            } else {                            // If format is xxxx
                frequency  = (U32)(brand_string[index-4] - ‘0’) * 1000;
                frequency += (U32)(brand_string[index-3] - ‘0’) * 100;
                frequency += (U32)(brand_string[index-2] - ‘0’) * 10;
                frequency += (U32)(brand_string[index-1] - ‘0’);
                frequency *= multiplier;
            }

            break;
        }
    }

    // Return frequency obtained from CPUID brand string or return 0 indicating
    // CPUID brand string not supported.
    return frequency;
}

void main(int argc, char *argv[])
{
    U32 freqBrandString=0;
    U16 freq=0;
    U16 mperf=0;
    U16 aperf=0;
    U16 aperf_mperf_percent=0;
    U16 mperf_tsc_percent=0;
    U16 aperf_tsc_percent=0;
    char freqBuffer[16];
```

```
    freqBrandString = GetFrequencyFromBrandString(freqBuffer);
    if (freqBrandString == 0) {
        printf(“CPUID brand string frequency not supported\n”);
    } else {
        printf(“CPUID brand string frequency=%s (%u MHz)\n”, freqBuffer, freqBrandString);
    }

    frequency(&freq, &mperf, &aperf, &aperf_mperf_percent,
              &mperf_tsc_percent, &aperf_tsc_percent);

    printf(“Timestamp    frequency= %4d MHz TSC measured over time interval using RTC\n”, freq);
    if (!mperf)
        printf(“IA32_MPERF and IA32_APERF MSRs not available!\n”);
    else {
       printf(“MPERF        frequency= %4d MHz MCNT measured over time interval using RTC\n”,
mperf);
       if (aperf)
          printf(“APERF        frequency= %4d MHz ACNT measured over time interval using RTC\n”,
aperf);
       if (aperf_mperf_percent)
          printf(“APERF/MPERF percentage= %3d%% isolates P-state impact (100%%=max non-Turbo)\n”,
aperf_mperf_percent);
       if (mperf_tsc_percent)
          printf(“MPERF/TSC   percentage= %3d%% isolates T-state impact (100%%=no throttling)\n”,
mperf_tsc_percent);
       if (aperf_tsc_percent)
          printf(“APERF/TSC   percentage= %3d%% actual performance (100%%=max non-Turbo)\n”,
aperf_tsc_percent);
    }
}
```