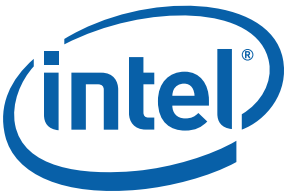


Intel® Processor Identification and the CPUID Instruction

Application Note 485

November 2008

Order Number: 241618-033

† 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

Revision	Description	Date
-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
-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 3-3 to include Intel Atom and Intel Core i7 processors Updated ECX and EDX feature flag definitions in Table 3-4 and Table 3-5 Updated cache and TLB descriptor values in Table 3-7 Updated Table 3-8 to feature Intel Core i7 processor Updated Section 3.1.3.1 and Table 3-8 to include Intel Core i7 processor Modified Table 3-10 to include new C-state definitions Updated Table 3-11 to include Intel® Turbo Boost Technology Added Section 3.1.13, “Reserved (Function 0Ch)” Combined Chapter 8: Usage Program Examples with Chapter 11: Program Examples into one chapter: Chapter 10: Program Examples	11/08

§

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

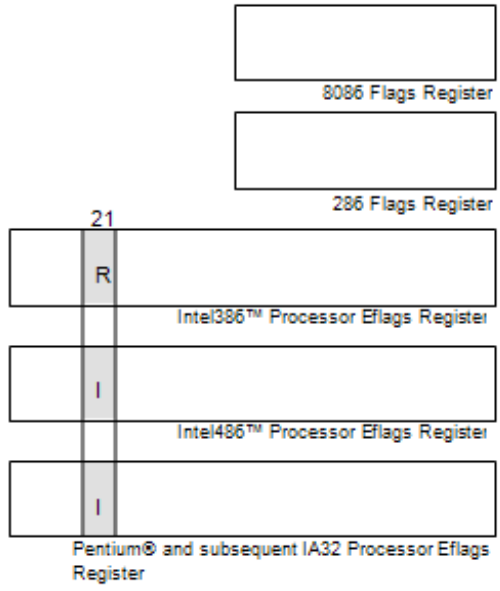
Intel processor signature and feature bits information can be obtained from the developer's manual, programmer's reference manual and appropriate processor documentation. In addition, updated versions of the programming examples included in this application note are available through your Intel representative,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

§

2 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 2-1).

Figure 2-1. 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.

3 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 3-1 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.

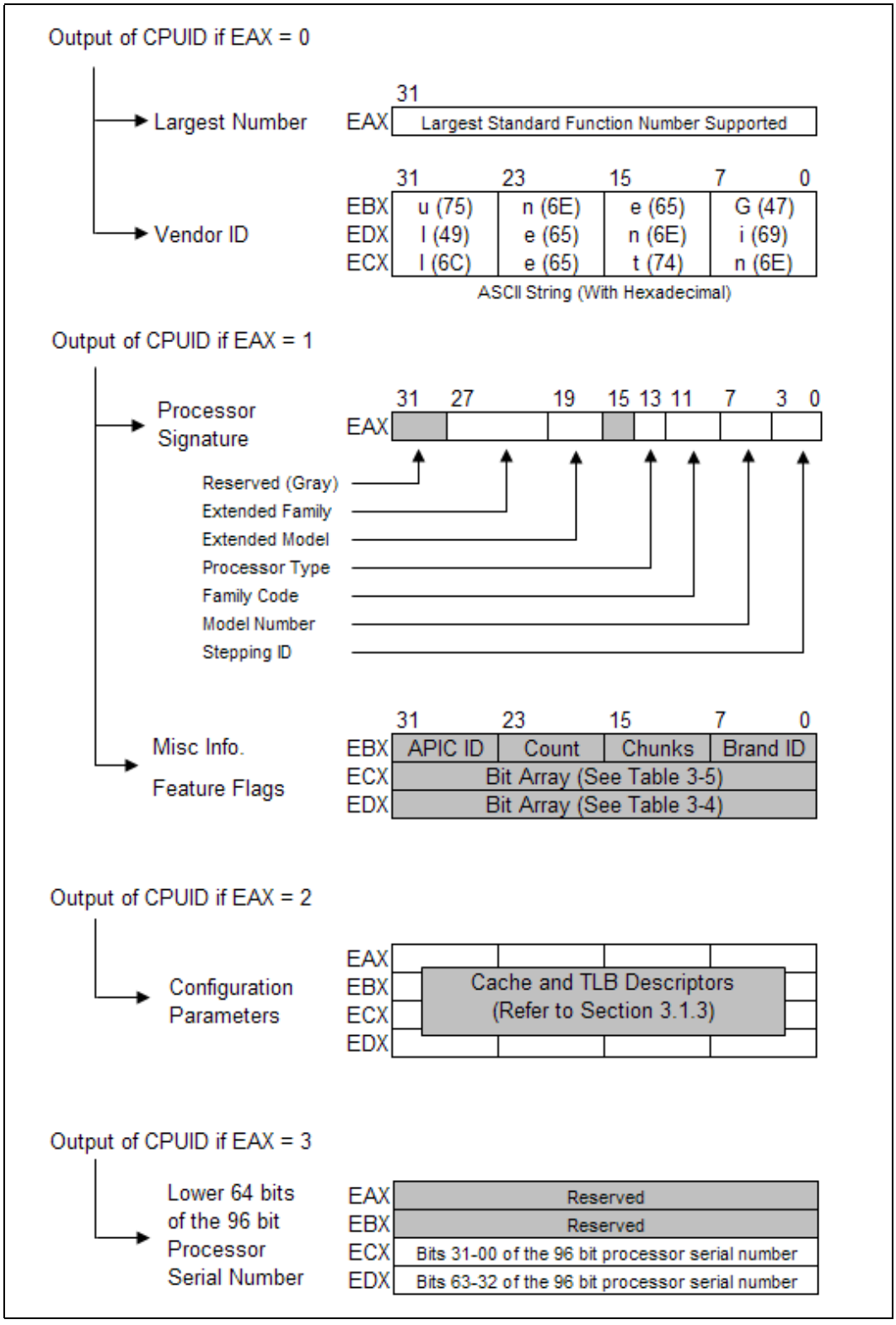
In order to determine the highest acceptable value for the EAX register input and CPUID functions that return the extended processor information, the program should set the EAX register parameter value to “80000000h” and then execute the CPUID instruction as follows:

```
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 3-1. CPUID Instruction Outputs



3.1 Standard CPUID Functions

3.1.1 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 3-1). These registers contain the ASCII string:

GenuineIntel

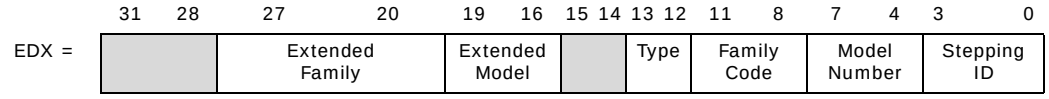
While any imitator of the Intel Architecture can provide the CPUID instruction, no imitator can legitimately claim that its part is a genuine Intel part. The presence of the "GenuineIntel" string is an assurance that the CPUID instruction and the processor signature are implemented as described in this 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.

3.1.2 Feature Information (Function 1)

3.1.2.1 Processor Signature

Beginning with the Intel486 processor family, the EDX register contains the processor identification signature after RESET (see Figure 3-2). **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 3-2. 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 3-2 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 3-1. Table 3-3 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 3-1 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 3-1. 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 5 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 3-3 to determine which processors support the CPUID instruction.

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

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

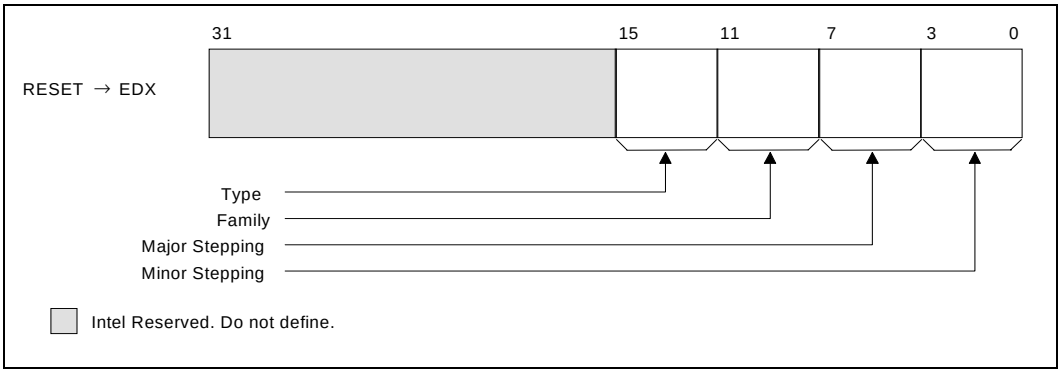


Table 3-2. Intel386™ Processor Signatures

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 3-3. 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 3-3. 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	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 3-3. 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 3-3. 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	1100	xxxx (2)	Intel Core i7 processor. 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.

3.1.2.2 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 3-1.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 3-2.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 3-3.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.

3.1.2.3 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 3-4 and Table 3-5 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 3-4. 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.
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.

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

Bit	Name	Description when Flag = 1	Comments
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 3.1.2.4 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.
26	SSE2	Streaming SIMD Extensions 2	Indicates the processor supports the Streaming SIMD Extensions 2 Instructions.

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

Bit	Name	Description when Flag = 1	Comments
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	IA64	IA64 Capabilities	The processor is a member of the Intel® Itanium® processor family and currently operating in IA-32 emulation mode.
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 3-5. 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		Reserved	Do not count on their value.
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.
25:24		Reserved	Do not count on their value.

Table 3-5. 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		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.

3.1.2.4 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.

3.1.3 Cache Descriptors (Function 2)

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 3-6 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 3-7 to determine the cache and TLB features of a processor.

Table 3-7 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 3-6. 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 3-7. 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 3-7. 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 3-7. 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	2-MB L3 Cache, 12-way set associative, 64-byte line size
DDh	4-MB L3 Cache, 12-way set associative, 64-byte line size
DEh	8-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
F0h	64-byte Prefetching
F1h	128-byte Prefetching

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

The Core i7 processor, model 1Ah returns the values shown in Table 3-8. Since the value of AL=1, it is valid to interpret the remainder of the registers. Table 3-8 also shows the MSB (bit 31) of all the registers are 0 which indicates that each register contains valid 8-bit descriptor.

Table 3-8. 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 3-8 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

3.1.4 Processor Serial Number (Function 3)

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 4 for more details.

3.1.5 Deterministic Cache Parameters (Function 4)

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 cores implemented in 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. After executing the CPUID instruction, (EAX[31:26] + 1) contains the number of cores.

Table 3-9. Deterministic Cache Parameters (Sheet 1 of 2)

Register Bits	Description
EAX[31:26]	Maximum number of processor cores per package. Encoded with a "plus 1" encoding. Add one to the value in the register field to get the number.
EAX[25:14]	Maximum number of threads sharing this cache. Encoded with a "plus 1" encoding. Add one to the value in the register field to get the number.
EAX[13:10]	**Reserved**.
EAX[09]	Fully Associative Cache
EAX[08]	Self Initializing cache level
EAX[07:05]	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 the register field to get the number.
EBX[21:12]	Physical Line partitions Encoded with a "plus 1" encoding. Add one to the value in the register field to get the number.

Table 3-9. Deterministic Cache Parameters (Sheet 2 of 2)

Register Bits	Description
EBX[11:0]	System Coherency Line Size Encoded with a “plus 1” encoding. Add one to the value in the register field to get the number.
ECX[31:0]	Number of Sets Encoded with a “plus 1” encoding. Add one to the value in the register field to get the number.
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 3-4.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)

3.1.5.1 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 cores per physical 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*.

3.1.6 MONITOR / MWAIT Parameters (Function 5)

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 3-10. MONITOR / MWAIT Parameters

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

3.1.7 Digital Thermal Sensor and Power Management Parameters (Function 6)

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 3-11. 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 ACNT, MCNT MSRs)
EDX[31:0]	**Reserved**

3.1.8 Reserved (Function 7)

This function is reserved.

3.1.9 Reserved (Function 8)

This function is reserved.

3.1.10 Direct Cache Access (DCA) Parameters (Function 9)

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

Table 3-12. 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**

3.1.11 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 3-13 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 3-13. Performance Monitor Features (Sheet 1 of 2)

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**

Table 3-13. Performance Monitor Features (Sheet 2 of 2)

Register Bits	Description
EDX[31:13]	**Reserved**
EDX[12:5]	Number of Bits in the Fixed Counters (width)
EDX[4:0]	Number of Fixed Counters

3.1.12 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, 1, 2, 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 3-14. 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 3-17). CPUID Function Bh with ECX=0 provides topology information for the thread level (refer to Table 3-15). And CPUID Function Bh with ECX=1 provides topology information for the Core level (refer to Table 3-16). 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 3-15. 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 3-16. 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 3-17. Core Level Processor Topology (CPUID Function 0Bh with ECX>=2) (Sheet 1 of 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)

Table 3-17. Core Level Processor Topology (CPUID Function 0Bh with ECX>=2) (Sheet 2 of 2)

Register Bits	Description	Value with ECX>=2 as Input
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.

3.1.13 Reserved (Function 0Ch)

This function is reserved.

3.1.14 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 3-18. 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.

3.2 Extended CPUID Functions

3.2.1 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 3-19. 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**

3.2.2 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 3-4 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 3-4. Extended Feature Flag Values Reported in the EDX Register

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.
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 3-20. 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

3.2.3 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.

3.2.3.1 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 3-1. 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
       MOB    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
```

3.2.3.2 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 3-2. 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:

        CALLDisplay_Character; Put a char on the
                        ; output device
        LODSB
        CMP AL, 0       ; Exit if NULL byte
```

```
                                ; encountered
        JNE Display_Char

Done:
```

3.2.4 Reserved (Function 8000_0005h)

This function is reserved.

3.2.5 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 3-5).

Figure 3-5. 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**

3.2.6 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 3-21. Power Management Details

Register Bits	Description
EAX[31:0]	**Reserved**
EBX[31:0]	**Reserved**

Table 3-21. Power Management Details

Register Bits	Description
ECX[31:0]	**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[31:0]	**Reserved**

3.2.7 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 3-22. 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**

§

4 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.

4.1 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.**

4.2 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.

§

5 Brand ID and Brand String

5.1 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 5-1). 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 5-1 shows the values defined for each processor.

Table 5-1. 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 5-1. 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**

5.2 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 5-2, 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.

The cpuid3a.asm program shows how software forms the brand string (see Chapter 10, Building the Processor Brand String). To determine if the brand string is supported on a processor, software must follow the step 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 5-2. 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

§

6 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.

§

7 Proper Identification Sequence

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

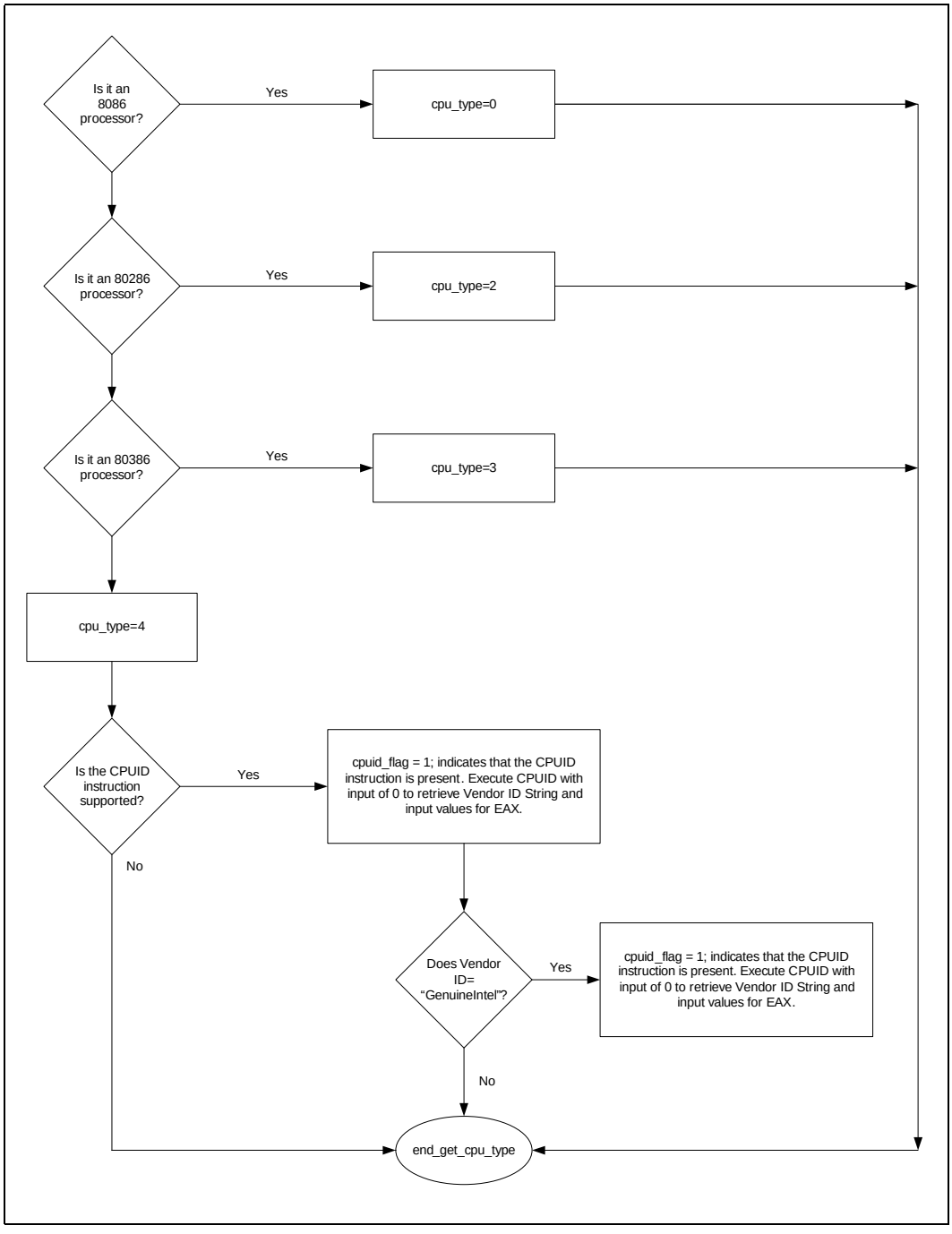
1. Determine if the CPUID instruction is supported by modifying the ID flag in the EFLAGS register. If the ID flag cannot be modified, the processor cannot be identified using the CPUID instruction.
2. Execute the CPUID instruction with EAX equal to 80000000h. CPUID function 80000000h is used to determine 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 5-15-1) to identify the processor.
4. If the Brand ID feature is not supported, software should use the processor signature (see Table 3-23-2 and Table 3-33-3) in conjunction with the cache descriptors (see Section 3.1.3) to identify the processor.

The cpuid3a.asm program example demonstrates the correct use of the CPUID instruction (see Example 1). 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 7-1). This program example contains the following two procedures:

- get_cpu_type identifies the processor type. Figure 7-1 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 7-1. Flow of Processor get_cpu_type Procedure



§

8 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 XMM feature flag (EDX bit 25) or the EMM 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.

§

9 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) 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 C0_MCNT (C0 maximum frequency clock count) register improve on the ability to calculate the C0 state frequency by provide a resetable free running counter. To use the C0_MCNT 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 C0_MCNT register. The actual frequency is calculated by dividing the C0_MCNT register value by the reference period. Reference the following steps as well as the assembly language program frequenc.asm (Frequency Detection) in Chapter 10 of this document.

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 is belonging to Intel Core 2 Duo processors family. This indicates the processor supports the Maximum Frequency Clock Count.
4. Clear the C0_MCNT register at the beginning of the reference period.
5. Read the C0_MCNT register at the end of the reference period.

6. Compute the actual frequency by dividing the C0_MCNT delta by the reference period.

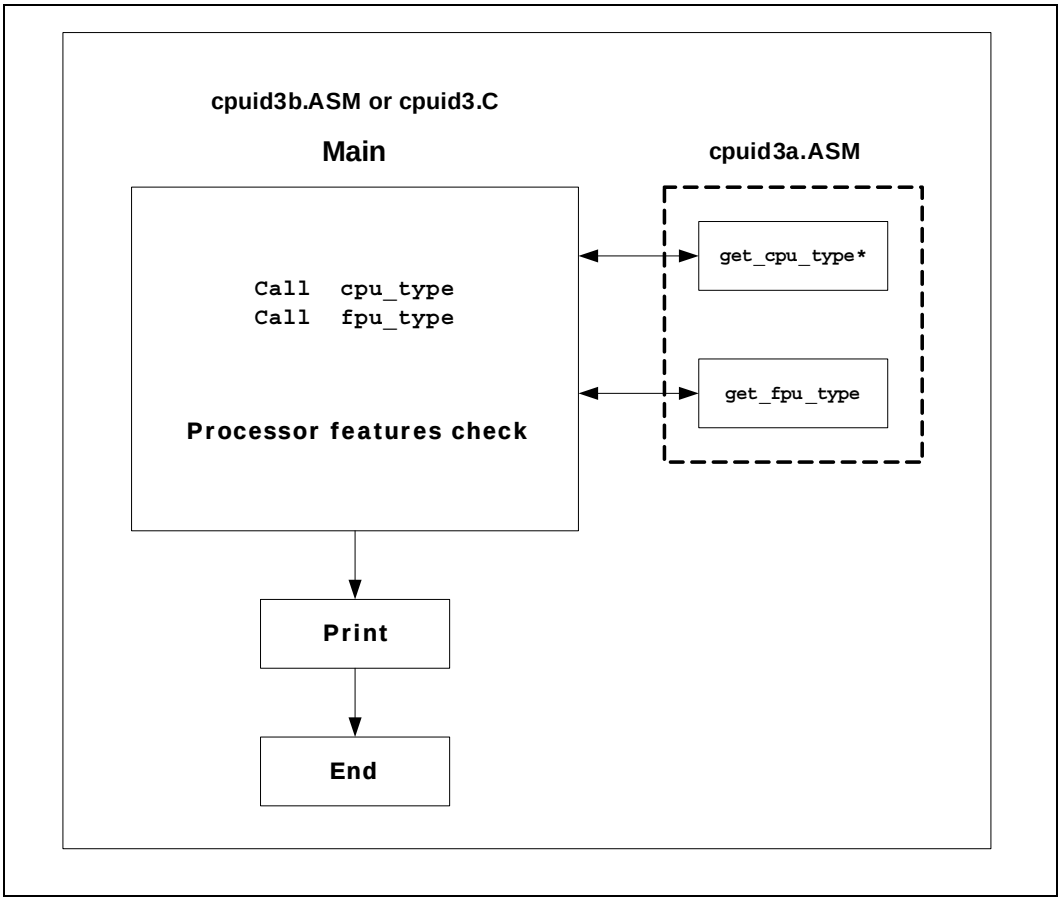
Actual frequency = Ending C0_MCNT value / reference period

§

10 Program Examples

As noted in Chapter 5, the cpuid3a.asm program shows how software forms the brand string (see Processor Identification Extraction Procedure). The cpuid3b.asm and cpuid3.c program examples demonstrate applications that call get_cpu_type and get_fpu_type procedures and interpret the returned information. This code is shown in Example 10-2 and Example 10-3. The results are displayed on the monitor and identify the installed processor and features. The cpuid3b.asm example is written in assembly language and demonstrates an application that displays the returned information in the DOS environment. The cpuid3.c example is written in the C language. Figure 10-1 presents an overview of the relationship between the three program examples.

Figure 10-1. Flow of Processor Identification Extraction Procedure



Example 10-1.Processor Identification Extraction Procedure

```
;       Filename:  cpuid3a.asm
;       Copyright (c) Intel Corporation 1993-2008
;
;       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 code is assumed to correctly detect the current Intel 8086/8088,
;       80286, 80386, 80486, Pentium(R) processor, Pentium(R) Pro
;       processor, Pentium(R) II processor, Pentium II Xeon(R) processor,
;       Pentium II Overdrive(R), Intel Celeron processor, Pentium III processor,
;       Pentium III Xeon processor, Pentium 4 processors, Intel(R) Xeon(R)
processors,
;       Intel(R) Core(TM) processors, Intel(R) Core(TM)2 processors, and Intel(R)
Core(TM) i7
;       processors.

;       NOTE: When using this code with C program cpuid3.c, 32-bit
;           segments are recommended.

;       To assemble this code with TASM, add the JUMPS directive.
;       jumps   ; Uncomment this line for TASM


        TITLEcpuid3a
;
;       comment this line for 32-bit segments
;
DOSSEG
;
;       uncomment the following 2 lines for 32-bit segments
;
;       .386
;       .modelflat
```

```
;
;       comment this line for 32-bit segments
;
        .modelsmall

CPU_ID  MACRO
        db  0fh ; Hardcoded CPUID instruction
        db  0a2h
ENDM

.data
        public_cpu_type
        public_fpu_type
        public_v86_flag
        public_cpuid_flag
        public_intel_CPU
        public_vendor_id
        public_cpu_signature
        public_features_ebx
        public_features_ecx
        public_features_edx
        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_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_sep_flag
        public_brand_string

        _cpu_typedb0
        _fpu_typedb0
        _v86_flagdb0
        _cpuid_flagdb0
        _intel_CPUdb0
        _sep_flagdb0
        _max_functdd0; Maximum function from CPUID.0.EAX
        _vendor_iddb"------------"
        intel_iddb"GenuineIntel"
        _cpu_signaturedd0; CPUID.1.EAX
        _features_ebxdd0; CPUID.1.EBX
        _features_ecxdd0; CPUID.1.ECX - feature flags
        _features_edxdd0; CPUID.1.EDX - feature flags
        _ext_max_functdd0; Max ext leaf from CPUID.80000000h.EAX
        _ext_funct_1_eaxdd0
        _ext_funct_1_ebxdd0
        _ext_funct_1_ecxdd0
        _ext_funct_1_edxdd0
```

```
        _ext_funct_6_eaxdd0
        _ext_funct_6_ebxdd0
        _ext_funct_6_ecxdd0
        _ext_funct_6_edxdd0
        _ext_funct_8_eaxdd0
        _ext_funct_8_ebxdd0
        _ext_funct_8_ecxdd0
        _ext_funct_8_edxdd0
        _cache_eaxdd0
        _cache_ebxdd0
        _cache_ecxdd0
        _cache_edxdd0
        _dcp_cache_eaxdd0
        _dcp_cache_ebxdd0
        _dcp_cache_ecxdd0
        _dcp_cache_edxdd0

        fp_statusdw0
        _brand_stringdb48 dup (0)

.code
;
;       comment this line for 32-bit segments
;
.8086
;
;       uncomment this line for 32-bit segments
;
;       .386

;************************************************************************
        public_get_cpu_type
        _get_cpu_typeproc

;       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
        pushax          ; 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
```

```
        pushsp          ; 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:
        smswax          ; 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
        pushcx          ; 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
        pusheax         ; 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
        pushecx
        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
```

```
        pusheax         ; 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.

        mov _cpuid_flag, 1; flag indicating use of CPUID inst.
        pushebx         ; save registers
        pushesi
        pushedi

        mov eax, 0      ; set up for CPUID instruction
        CPU_ID          ; get and save vendor ID

        mov dword ptr _vendor_id, ebx
        mov dword ptr _vendor_id[+4], edx
        mov dword ptr _vendor_id[+8], ecx
        mov dword ptr _max_funct, eax

        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      ; make sure 1 is valid input for CPUID
        jl  ext_functions; if not, jump to end

        mov eax, 1
        CPU_ID          ; get family/model/stepping/features

        mov _cpu_signature, eax
        mov _features_ebx, ebx
        mov _features_edx, edx
        mov _features_ecx, ecx

        shr eax, 8      ; isolate family
        and eax, 0fh
        mov _cpu_type, al; set _cpu_type with family

;       Execute CPUID.2 instruction to determine the cache descriptor
;       information.

        cmp _max_funct, 2
        jl  ext_functions

        mov eax, 2      ; set up to read cache descriptor
        CPU_ID
        cmp al, 1       ; Is one iteration enough to obtain
        jne ext_functions; cache information?
                        ; This code supports one iteration
                        ; only.
        mov _cache_eax, eax; store cache information
        mov _cache_ebx, ebx; NOTE: for future processors, CPUID
        mov _cache_ecx, ecx; instruction may need to be run more
```

```
                mov _cache_edx, edx; than once to get complete cache
                                   ; information

                cmp _max_funct, 4; Deterministic cache parameters supported?
                jl  ext_functions

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

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

        ext_functions:
                mov eax, 80000000h; check if brand string is supported
                CPU_ID
        mov         _ext_max_funct, eax
                cmp eax, 80000004h
                jb  end_cpuid_type; take jump if not supported


                mov eax, 80000001h; Get the Extended Feature Flags
                CPU_ID

                mov _ext_funct_1_eax, eax
                mov _ext_funct_1_ebx, ebx
                mov _ext_funct_1_ecx, ecx
                mov _ext_funct_1_edx, edx

                mov di, offset _brand_string

                mov eax, 80000002h; get first 16 bytes of brand string
                CPU_ID
                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
                CPU_ID
                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
                CPU_ID
                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; are L2 Cache Features supported
```

```
        jb  end_cpuid_type

        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, 80000008h; is Address Size function supported
        jb  end_cpuid_type

        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 this line for 32-bit segments
;
.8086
end_cpu_type:
        ret
_get_cpu_typeendp

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

        public_get_fpu_type
        _get_fpu_typeproc

;       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
        fnstswfp_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:
        fnstcwfp_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
        fstswfp_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_typeendp

        end
```

Example 10-2.Processor Identification Procedure in Assembly Language

```
;       Filename:  cpuid3b.asm
;       Copyright (c) Intel Corporation 1993-2008
;
;       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 message.  This part is
;           specific to the DOS environment and uses the DOS
;           system calls to print out the messages.
;
;       If this code is assembled with no options specified and linked
;       with the cpuid3a module, it is assumed to correctly identify the current
;       Intel 8086/8088, 80286, 80386, 80486, Pentium(R), Pentium(R) Pro,
;       Pentium(R) II processors, Pentium(R) II Xeon(R) processors, Pentium(R) II
;       Overdrive(R) processors,  Intel(R) Celeron(R) processors, Pentium(R) III
;       processors, Pentium(R) III Xeon(R) processors, Pentium(R) 4 processors,
;       Intel(R) Xeon(R) processors DP and MP, Intel(R) Core(TM) processors,
;       Intel(R) Core(TM)2 processors, and Intel(R) Core(TM) i7 processors when
;       executed in the real-address mode.

;       NOTE: This code is written using 16-bit Segments

;       To assemble this code with TASM, add the JUMPS directive.
;       jumps           ; Uncomment this line for TASM

        TITLEcpuid3b

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_vendor_id: byte
        extrn_cpu_signature:dword
```

```
        extrn_features_ecx:dword
        extrn_features_edx:dword
        extrn_features_ebx: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_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
        callprint

        mov ax, 4c00h
        int 21h

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

        extrn_get_cpu_type: proc

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

        extrn_get_fpu_type: proc

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

FPU_FLAGequ 0001h
VME_FLAGequ 0002h
DE_FLAG     equ 0004h
PSE_FLAGequ 0008h
TSC_FLAGequ 0010h
```

```
MSR_FLAGequ 0020h
PAE_FLAGequ 0040h
MCE_FLAGequ 0080h
CX8_FLAGequ 0100h
APIC_FLAGequ 0200h
SEP_FLAGequ 0800h
MTRR_FLAGequ 1000h
PGE_FLAGequ 2000h
MCA_FLAGequ 4000h
CMOV_FLAGequ 8000h
PAT_FLAGequ 10000h
PSE36_FLAGequ 20000h
PSNUM_FLAGequ 40000h
CLFLUSH_FLAGequ 80000h
DTS_FLAGequ 200000h
ACPI_FLAGequ 400000h
MMX_FLAGequ 800000h
FXSR_FLAGequ 1000000h
SSE_FLAGequ 2000000h
SSE2_FLAGequ 4000000h
SS_FLAG    equ 8000000h
HTT_FLAGequ 10000000h
TM_FLAG    equ 20000000h
IA64_FLAGequ 40000000h
PBE_FLAGequ 80000000h

SSE3_FLAGequ 0001h
MONITOR_FLAGequ 0008h
DS_CPL_FLAG equ 0010h
VMX_FLAGequ 0020h
SMX_FLAGequ 0040h
EIST_FLAGequ 0080h
TM2_FLAGequ 0100h
SSSE3_FLAGequ 0200h
CID_FLAGequ 0400h
CX16_FLAGequ 02000h
XTPR_FLAGequ 04000h
PDCM_FLAGequ 08000h
DCA_FLAGequ 040000h
SSE4_1_FLAGequ 080000h
SSE4_2_FLAGque 0100000h

EM64T_FLAGequ 20000000h
XD_FLAG    equ 00100000h
SYSCALL_FLAGequ 00000800h
LAHF_FLAGequ 00000001h

.data
id_msg     db     "This system has a$"
cp_errordb     "n unknown processor$"
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$"
cp_486sxdb     "n 80486SX processor$"

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_msgdb " Genuine Intel486(TM) processor$"
intel486dx_msgdb" Genuine Intel486(TM) DX processor$"
intel486sx_msgdb" Genuine Intel486(TM) SX processor$"
inteldx2_msgdb " Genuine IntelDX2(TM) processor$"
intelsx2_msgdb " Genuine IntelSX2(TM) processor$"
inteldx4_msgdb " Genuine IntelDX4(TM) processor$"
inteldx2wb_msgdb" Genuine Write-Back Enhanced"
           db      " IntelDX2(TM) processor$"
pentium_msgdb  " Genuine Intel(R) Pentium(R) processor$"
pentiumpro_msgdb" Genuine Intel Pentium(R) Pro processor$"

pentiumiimodel3_msgdb" Genuine Intel(R) Pentium(R) II processor, model 3$"
pentiumiixeon_m5_msgdb" Genuine Intel(R) Pentium(R) II processor, model 5 or"
           db      " Intel(R) Pentium(R) II Xeon(R) processor$"
pentiumiixeon_msgdb" Genuine Intel(R) Pentium(R) II Xeon(R) processor$"
celeron_msgdb" Genuine Intel(R) Celeron(R) processor, model 5$"
celeronmodel6_msgdb" Genuine Intel(R) Celeron(R) processor, model 6$"
celeron_branddb" Genuine Intel(R) Celeron(R) processor$"
pentiumiii_msgdb" Genuine Intel(R) Pentium(R) III processor, model 7 or"
           db  " Intel Pentium(R) III Xeon(R) processor, model 7$"
pentiumiiixeon_msgdb" Genuine Intel(R) Pentium(R) III Xeon(R) processor, model 7$"
pentiumiiixeon_branddb" Genuine Intel(R) Pentium(R) III Xeon(R) processor$"
pentiumiii_branddb" Genuine Intel(R) Pentium(R) III processor$"
mobile_piii_branddb" Genuine Mobile Intel(R) Pentium(R) III Processor-M$"
mobile_icp_branddb" Genuine Mobile Intel(R) Celeron(R) processor$"
mobile_P4_branddb" Genuine Mobile Intel(R) Pentium(R) 4 processor - M$"
pentium4_branddb" Genuine Intel(R) Pentium(R) 4 processor$"
xeon_branddb" Genuine Intel(R) Xeon(R) processor$"
xeon_mp_branddb" Genuine Intel(R) Xeon(R) processor MP$"
mobile_icp_brand_2db" Genuine Mobile Intel(R) Celeron(R) processor$"
mobile_pentium_m_branddb" Genuine Intel(R) Pentium(R) M processor$"
mobile_genuine_branddb" Mobile Genuine Intel(R) processor$"
mobile_icp_m_branddb" Genuine Intel(R) Celeron(R) M processor$"
unknown_msgdb"n unknown Genuine Intel(R) processor$"

brand_entrystruct
       brand_valuedb?
       brand_stringdw?
brand_entryends

brand_tablebrand_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_sizeequ($ - offset brand_table) / (sizeof brand_entry)
```

```
; The following 16 entries must stay intact as an array
intel_486_0dw  offset intel486dx_msg
intel_486_1dw  offset intel486dx_msg
intel_486_2dw  offset intel486sx_msg
intel_486_3dw  offset inteldx2_msg
intel_486_4dw  offset intel486_msg
intel_486_5dw  offset intelsx2_msg
intel_486_6dw  offset intel486_msg
intel_486_7dw  offset inteldx2wb_msg
intel_486_8dw  offset inteldx4_msg
intel_486_9dw  offset intel486_msg
intel_486_adw  offset intel486_msg
intel_486_bdw  offset intel486_msg
intel_486_cdw  offset intel486_msg
intel_486_ddw  offset intel486_msg
intel_486_edw  offset intel486_msg
intel_486_fdw  offset intel486_msg
; end of array

signature_msgdb 13, 10,"Processor Signature / Version Information: $"
family_msgdb13,10,"Processor Family:  $"
model_msgdb13,10,"Model:             $"
stepping_msgdb13,10,"Stepping:          $"
ext_fam_msgdb13,10," Extended Family: $"
ext_mod_msgdb13,10," Extended Model:  $"
cr_lf      db  13,10,"$"
turbo_msgdb13,10,"The processor is an OverDrive(R)"
           db  "  processor$"
dp_msg     db  13,10,"The processor is the upgrade"
           db  " processor in a dual processor system$"
fpu_msg    db  13,10,"The processor contains an on-chip"
           db  " FPU$"
vme_msg    db  13,10,"The processor supports Virtual"
           db  " Mode Extensions$"
de_msg     db  13,10,"The processor supports Debugging"
           db  " Extensions$"
pse_msg    db  13,10,"The processor supports Page Size"
           db  " Extensions$"
tsc_msg    db  13,10,"The processor supports Time Stamp"
           db  " Counter$"
msr_msg    db  13,10,"The processor supports Model"
           db  " Specific Registers$"
pae_msg    db  13,10,"The processor supports Physical"
           db  " Address Extensions$"
mce_msg    db  13,10,"The processor supports Machine"
           db  " Check Exceptions$"
cx8_msg    db  13,10,"The processor supports the"
           db  " CMPXCHG8B instruction$"
apic_msgdb 13,10,"The processor contains an on-chip"
           db  " APIC$"
sep_msg    db  13,10,"The processor supports Fast System"
           db  " Call$"
no_sep_msgdb13,10,"The processor does not support Fast"
           db  " System Call$"
mtrr_msgdb 13,10,"The processor supports Memory Type"
           db  " Range Registers$"
pge_msg    db  13,10,"The processor supports Page Global"
           db  " Enable$"
mca_msg    db  13,10,"The processor supports Machine"
           db  " Check Architecture$"
cmov_msgdb 13,10,"The processor supports Conditional"
```

```
            db  " Move Instruction$"
pat_msg     db  13,10,"The processor supports Page  Attribute"
            db  " Table$"
pse36_msgdb 13,10,"The processor supports 36-bit Page"
            db  " Size Extension$"
psnum_msgdb 13,10,"The processor supports the"
            db  " processor serial number$"
clflush_msgdb13,10,"The processor supports the"
            db  " CLFLUSH instruction$"
dts_msg     db  13,10,"The processor supports the"
            db  " Debug Trace Store feature$"
acpi_msgdb  13,10,"The processor supports the"
            db  " ACPI registers in MSR space$"
mmx_msg     db  13,10,"The processor supports Intel Architecture"
            db  " MMX(TM) Technology$"
fxsr_msgdb  13,10,"The processor supports Fast floating point"
            db  " save and restore$"
sse_msg     db  13,10,"The processor supports the Streaming"
            db  " SIMD extensions$"
sse2_msgdb  13,10,"The processor supports the Streaming"
            db  " SIMD extensions 2 instructions$"
ss_msg      db  13,10,"The processor supports Self-Snoop$"
htt_msg     db  13,10,"The processor supports Hyper-Threading Technology$"
tm_msg      db  13,10,"The processor supports the"
            db  " Thermal Monitor$"
ia64_msgdb  13,10,"The processor is a member of the"
            db  "Intel(R) Itanium(R) processor family executing in IA32 emulation
            db  " mode$"
pbe_msg     db  13,10,"The processor supports the Pending Break Event$"
sse3_msgdb  13,10,"The processor supports the Streaming SIMD"
            db  " Extensions 3 instructions$"
monitor_msgdb13,10,"The processor supports the MONITOR and MWAIT"
            db  " instructions$"
ds_cpl_msgdb13,10,"The processor supports Debug Store extensions for"
            db  " branch message storage by CPL$"
vmx_msg     db  13,10,"The processor supports Intel(R) Virtualization
            db  " Technology$"
smx_msg     db  13,10,"The processor supports Intel(R) Trusted Execution "
            db  "Technology$"
eist_msgdb  13,10,"The processor supports"
            db  " Enhanced SpeedStep(R) Technology$"
tm2_msg     db  13,10,"The processor supports the"
            db  " Thermal Monitor 2$"
ssse3_msgdb 13,10, "The processor supports the Supplemental Streaming SIMD"
            db  " Extensions 3 instructions$"
cid_msg     db  13,10,"The processor supports L1 Data Cache Context ID$"
cx16_msgdb  13,10,"The processor supports CMPXCHG16B instruction$"
xtpr_msgdb  13,10,"The processor supports transmitting TPR messages$"
pdcm_msgdb  13,10,"The processor supports the Performance Capabilities MSR$"
dca_msg     db  13,10,"The processor supports the Direct Cache Access feature$"
sse4_1_msgdb13,10,"The processor supports the Supplemental Streaming SIMD "
            db  "Extensions 4.1 instructions$"
sse4_2_msgdb13,10,"The processor supports the Supplemental Streaming SIMD "
            db  "Extensions 4.2 instructions$"

em64t_msgdb 13,10, "The processor supports Intel(R) Extended Memory 64
            db  " Technology$"
xd_bit_msgdb13,10, "The processor supports the Execute Disable Bit$"
syscall_msgdb13,10, "The processor supports the SYSCALL & SYSRET
            db  " instructions$"
lahf_msgdb  13,10, "The processor supports the LAHF & SAHF instructions$"
```

```
not_inteldb"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 at this time.$"

ASC_MSG MACROmsg
        LOCALascii_done; local label
        add al, 30h
        cmp al, 39h     ; is it 0-9?
        jle ascii_done
        add al, 07h
ascii_done:
        mov byte ptr msg[20], al
        mov dx, offset msg
        mov ah, 9h
        int 21h
ENDM

.code
.8086

print   proc

;       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.

        mov dx, offset cr_lf
        mov ah, 9h
        int 21h

        mov dx, offset id_msg; print initial message
        mov ah, 9h
        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 dx, offset cp_8086
        mov ah, 9h
        int 21h
        cmp _fpu_type, 0
        je  end_print
        mov dx, offset fp_8087
        mov ah, 9h
        int 21h
        jmp end_print

print_286:
        cmp _cpu_type, 2
        jne print_386
        mov dx, offset cp_286
        mov ah, 9h
        int 21h
```

```
        cmp _fpu_type, 0
        je  end_print

print_287:
        mov dx, offset fp_287
        mov ah, 9h
        int 21h
        jmp end_print

print_386:
        cmp _cpu_type, 3
        jne print_486
        mov dx, offset cp_386
        mov ah, 9h
        int 21h
        cmp _fpu_type, 0
        je  end_print
        cmp _fpu_type, 2
        je  print_287
        mov dx, offset fp_387
        mov ah, 9h
        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, 9h
        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 dl, '       '; Print a space (' ') character
        mov ah, 2
        int 21h

print_brand_char:
        mov dl, [di]    ; print upto the max chars
        mov ah, 2
        int 21h

        inc di
        cmp byte ptr [di], 0
        je  print_family
        loopprint_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
        jae 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, print 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
        loopceleron_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
        loopceleron_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
        loopceleron_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
        loopceleron_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
        cmp al, 0       ; Is brand_id supported?
        je  print_unknown_type

        mov di, offset brand_table; Setup pointer to brand_id table
        mov cx, brand_table_size; 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
        loopnext_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, 9h
        int 21h

; print family, model, and stepping
print_family:
        mov dx, offset cr_lf
        mov ah, 9h
        int 21h

        mov dx, offset signature_msg
        mov ah, 9h
        int 21h

        mov eax, dword ptr _cpu_signature
        mov cx, 8

print_signature:

; print all 8 digits of the processor signature EAX[31:0]

        rol eax, 4      ; Moving EAX[31:28] into EAX[3:0]
        mov dl, al
        and dl, 0fh
        add dl, '0      '; Convert the nibble to ASCII number
        cmp dl, '9'
        jle @f

        add dl, 7

@@:
        pusheax
        mov ah, 2
        int 21h         ; print lower nibble of ext family
        pop eax

        loopprint_signature
```


```
        pusheax
        mov dx, offset family_msg
        mov ah, 9h
        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

@@:
        cmp al, '9'
        jl  @f
        add al, 7

@@:
        pusheax
        mov dl, ah      ;  Print upper nibble Family[7:4]
        mov ah, 2
        int 21h
        pop eax

        mov dl, al      ; Print lower nibble Family[3:0]
        mov ah, 2
        int 21h

print_model:

        mov dx, offset model_msg
        mov ah, 9h
        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

@@:
        cmp al, '9'
        jl  @f
        add al, 7

@@:
        pusheax
        mov dl, ah      ; print upper nibble Model[7:4]
        mov ah, 2
        int 21h
        pop eax

        mov dl, al      ; print lower nibble Model[3:0]
        mov ah, 2
        int 21h

print_stepping:

        mov dx, offset stepping_msg
        mov ah, 9h
        int 21h

        mov eax, dword ptr _cpu_signature
        mov dl, al
        and dl, 0Fh
        add dl, '0'
        cmp dl, '9'
        jle @f
        add dl, 7

@@:
        mov ah, 2
        int 21h

        mov dx, offset cr_lf
        mov ah, 9h
        int 21h

print_upgrade:
        mov eax, dword ptr _cpu_signature
        testax, 1000h  ; check for turbo upgrade
        jz  check_dp
        mov dx, offset turbo_msg
        mov ah, 9h
        int 21h
        jmp print_features

check_dp:
        testax, 2000h  ; check for dual processor
        jz  print_features
```

```
        mov dx, offset dp_msg
        mov ah, 9h
        int 21h

print_features:
        test dword ptr _features_edx, FPU_FLAG; check for FPU
        jz  check_VME
        mov dx, offset fpu_msg
        mov ah, 9h
        int 21h

check_VME:
        test dword ptr _features_edx, VME_FLAG; check for VME
        jz  check_DE
        mov dx, offset vme_msg
        mov ah, 9h
        int 21h

check_DE:
        test dword ptr _features_edx, DE_FLAG; check for DE
        jz  check_PSE
        mov dx, offset de_msg
        mov ah, 9h
        int 21h

check_PSE:
        test dword ptr _features_edx, PSE_FLAG; check for PSE
        jz  check_TSC
        mov dx, offset pse_msg
        mov ah, 9h
        int 21h

check_TSC:
        test dword ptr _features_edx, TSC_FLAG; check for TSC
        jz  check_MSR
        mov dx, offset tsc_msg
        mov ah, 9h
        int 21h

check_MSR:
        test dword ptr _features_edx, MSR_FLAG; check for MSR
        jz  check_PAE
        mov dx, offset msr_msg
        mov ah, 9h
        int 21h

check_PAE:
        test dword ptr _features_edx, PAE_FLAG; check for PAE
        jz  check_MCE
        mov dx, offset pae_msg
        mov ah, 9h
        int 21h

check_MCE:
        test dword ptr _features_edx, MCE_FLAG; check for MCE
        jz  check_CX8
        mov dx, offset mce_msg
        mov ah, 9h
        int 21h

check_CX8:
        test dword ptr _features_edx, CX8_FLAG; check for CMPXCHG8B
```

```
        jz  check_APIC
        mov dx, offset cx8_msg
        mov ah, 9h
        int 21h

check_APIC:
        test dword ptr _features_edx, APIC_FLAG; check for APIC
        jz  check_SEP
        mov dx, offset apic_msg
        mov ah, 9h
        int 21h

check_SEP:
        test dword ptr _features_edx, SEP_FLAG; Check for Fast System Call
         jz check_MTRR

        cmp _cpu_type, 6; Determine if Fast System
        jne print_sep   ;  Calls are supported.

        mov eax, dword ptr _cpu_signature
        cmp al, 33h
        jb  print_no_sep

print_sep:
        mov dx, offset sep_msg
        mov ah, 9h
        int 21h
        jmp check_MTRR

print_no_sep:
        mov dx, offset no_sep_msg
        mov ah, 9h
        int 21h

check_MTRR:
        test dword ptr _features_edx, MTRR_FLAG; check for MTRR
        jz  check_PGE
        mov dx, offset mtrr_msg
        mov ah, 9h
        int 21h

check_PGE:
        test dword ptr _features_edx, PGE_FLAG; check for PGE
        jz  check_MCA
        mov dx, offset pge_msg
        mov ah, 9h
        int 21h

check_MCA:
        test dword ptr _features_edx, MCA_FLAG; check for MCA
        jz  check_CMOV
        mov dx, offset mca_msg
        mov ah, 9h
        int 21h

check_CMOV:
        test dword ptr _features_edx, CMOV_FLAG; check for CMOV
        jz  check_PAT
        mov dx, offset cmov_msg
        mov ah, 9h
        int 21h
```

```
check_PAT:
        test dword ptr _features_edx, PAT_FLAG; Page Attribute Table?
        jz  check_PSE36
        mov dx, offset pat_msg
        mov ah, 9h
        int 21h

check_PSE36:
        test dword ptr _features_edx, PSE36_FLAG; Page Size Extensions?
        jz  check_PSNUM
        mov dx, offset pse36_msg
        mov ah, 9h
        int 21h

check_PSNUM:
        test dword ptr _features_edx, PSNUM_FLAG; check for processor serial number
        jz  check_CLFLUSH
        mov dx, offset psnum_msg
        mov ah, 9h
        int 21h

check_CLFLUSH:
        test dword ptr _features_edx, CLFLUSH_FLAG; check for Cache Line Flush
        jz  check_DTS
        mov dx, offset clflush_msg
        mov ah, 9h
        int 21h

check_DTS:
        test dword ptr _features_edx, DTS_FLAG; check for Debug Trace Store
        jz  check_ACPI
        mov dx, offset dts_msg
        mov ah, 9h
        int 21h

check_ACPI:
        test dword ptr _features_edx, ACPI_FLAG; check for processor serial number
        jz  check_MMX
        mov dx, offset acpi_msg
        mov ah, 9h
        int 21h

check_MMX:
        test dword ptr _features_edx, MMX_FLAG; check for MMX technology
        jz  check_FXSR
        mov dx, offset mmx_msg
        mov ah, 9h
        int 21h

check_FXSR:
        test dword ptr _features_edx, FXSR_FLAG; check for FXSR
        jz  check_SSE
        mov dx, offset fxsr_msg
        mov ah, 9h
        int 21h

check_SSE:
        test dword ptr _features_edx, SSE_FLAG; check for Streaming SIMD
         jz check_SSE2;   Extensions
        mov dx, offset sse_msg
        mov ah, 9h
        int 21h
```

```
check_SSE2:
        test dword ptr _features_edx, SSE2_FLAG; check for Streaming SIMD
        jz  check_SS    ;   Extensions 2
        mov dx, offset sse2_msg
        mov ah, 9h
        int 21h

check_SS:
        test dword ptr _features_edx, SS_FLAG; check for Self Snoop
        jz  check_HTT
        mov dx, offset ss_msg
        mov ah, 9h
        int 21h

check_HTT:
        test dword ptr _features_edx, HTT_FLAG; check for Hyper-Thread Technology
        jz  check_IA64

        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
        jl  check_IA64

        mov dx, offset htt_msg; Supports HTT
        mov ah, 9h
        int 21h

check_IA64:
        test dword ptr _features_edx, IA64_FLAG; check for IA64 capabilites
        jz  check_TM
        mov dx, offset ia64_msg
        mov ah, 9h
        int 21h

check_TM:
        test dword ptr _features_edx, TM_FLAG; check for Thermal Monitor
        jz  check_PBE
        mov dx, offset tm_msg
        mov ah, 9h
        int 21h

check_PBE:
        test dword ptr _features_edx, PBE_FLAG; check for Pending Break Event
        jz  check_sse3
        mov dx, offset pbe_msg
        mov ah, 9h
        int 21h

check_sse3:
        test dword ptr _features_ecx, SSE3_FLAG; check for SSE3 instructions
        jz  check_monitor
        mov dx, offset sse3_msg
        mov ah, 9h
        int 21h
```

```
check_monitor:
        test dword ptr _features_ecx, MONITOR_FLAG; check for monitor/mwait
instructions
        jz  check_ds_cpl
        mov dx, offset monitor_msg
        mov ah, 9h
        int 21h

check_ds_cpl:
        test dword ptr _features_ecx, DS_CPL_FLAG; Check for DS_CPL
        jz  check_VMX
        mov dx, offset ds_cpl_msg
        mov ah, 9h
        int 21h

check_VMX:
        test dword ptr _features_ecx, VMX_FLAG; Virtualization Technology?
        jz  check_SMX
        mov dx, offset vmx_msg
        mov ah, 9h
        int 21h

check_SMX:
        test dword ptr _features_ecx, SMX_FLAG; Trusted Execution Technology?
        jz  check_EIST
        mov dx, offset smx_msg
        mov ah, 9h
        int 21h

check_EIST:        ; check for Enhanced Intel SpeedStep Technology
        test dword ptr _features_ecx, EIST_FLAG
        jz  check_TM2
        mov dx, offset eist_msg
        mov ah, 9h
        int 21h

check_TM2:
        test dword ptr _features_ecx, TM2_FLAG; check for Thermal Monitor 2
        jz  check_SSSE3
        mov dx, offset tm2_msg
        mov ah, 9h
        int 21h

check_SSSE3:
        test dword ptr _features_ecx, SSSE3_FLAG; check for  SSSE3
        jz  check_CID
        mov dx, offset ssse3_msg
        mov ah, 9h
        int 21h

check_CID:
        test dword ptr _features_ecx, CID_FLAG; check for L1 Context ID
        jz  check_CX16
        mov dx, offset cid_msg
        mov ah, 9h
        int 21h

check_CX16:
        test dword ptr _features_ecx, CX16_FLAG; check for CMPXCHG16B
        jz  check_XTPR
        mov dx, offset cx16_msg
        mov ah, 9h
```

```
        int 21h

check_XTPR:
        test dword ptr _features_ecx, XTPR_FLAG; check for echo Task Priority
        jz  check_PDCM
        mov dx, offset xtpr_msg
        mov ah, 9h
        int 21h

check_PDCM:
        test dword ptr _features_ecx, PDCM_FLAG; check for echo Task Priority
        jz  check_DCA
        mov dx, offset pdcm_msg
        mov ah, 9h
        int 21h

check_DCA:
        test dword ptr _features_ecx, DCA_FLAG; Direct Cache Access?
        jz  check_SSE4_1
        mov dx, offset dca_msg
        mov ah, 9h
        int 21h

check_SSE4_1:
        test dword ptr _features_ecx, SSE4_1_FLAG; SSE4.1 Instructions?
        jz  check_SSE4_2
        mov dx, offset sse4_1_msg
        mov ah, 9h
        int 21h

check_SSE4_1:
        test dword ptr _features_ecx, SSE4_2_FLAG; SSE4.2 Instructions?
        jz  check_LAHF
        mov dx, offset sse4_2_msg
        mov ah, 9h
        int 21h

check_LAHF:
        test dword ptr _ext_funct_1_ecx, LAHF_FLAG; check for LAHF/SAHF
instructions
        jz  check_SYSCALL
        mov dx, offset LAHF_msg
        mov ah, 9h
        int 21h

check_SYSCALL:     ; check for SYSCALL/SYSRET instructions
        test dword ptr _ext_funct_1_edx, SYSCALL_FLAG
        jz  check_XD
        mov dx, offset syscall_msg
        mov ah, 9h
        int 21h

check_XD:
        test dword ptr _ext_funct_1_edx, XD_FLAG; Check for Execute Disable
        jz  check_EM64T
        mov dx, offset xd_bit_msg
        mov ah, 9h
        int 21h

check_EM64T:
        test dword ptr _ext_funct_1_edx, EM64T_FLAG  ; check for Intel EM64T
        jz  end_print
```

```
        mov dx, offset em64t_msg
        mov ah, 9h
        int 21h

        jmp end_print

not_GenuineIntel:
        mov dx, offset not_intel
        mov ah, 9h
        int 21h

end_print:
        mov dx, offset cr_lf
        mov ah, 9h
        int 21h
        ret
print   endp

        end start
```

Example 10-3.Processor Identification Procedure in the C Language

```
/* FILENAME:  CPUID3.C  */
/* Copyright (c) Intel Corporation 1994-2008*/
/*     */
/* 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 message.  */
/*     */
/* If this code is assembled with no options specified and linked*/
/* with the cpuid3a module, it is assumed to correctly identify the current*/
/* Intel 8086/8088, 80286, 80386, 80486, Pentium(R), Pentium(R) Pro, */
/* Pentium(R) II processors, Pentium(R) II Xeon(R) processors, Pentium(R) II */
/* Overdrive(R) processors,  Intel(R) Celeron(R) processors, Pentium(R) III */
/* processors, Pentium(R) III Xeon(R) processors, Pentium(R) 4 processors, */
/* Intel(R) Xeon(R) processors DP and MP, Intel(R) Core(TM) processors,*/
/* Intel(R) Core(TM)2 processors and  Intel(R) Core(TM) i7 processors when*/
/* executed in the real-address mode. */

#define FPU_FLAG0x0001
#define VME_FLAG0x0002
#define DE_FLAG0x0004
#define PSE_FLAG0x0008
#define TSC_FLAG0x0010
#define MSR_FLAG0x0020
#define PAE_FLAG0x0040
#define MCE_FLAG0x0080
#define CX8_FLAG0x0100
#define APIC_FLAG0x0200
#define SEP_FLAG0x0800
#define MTRR_FLAG0x1000
#define PGE_FLAG0x2000
#define MCA_FLAG0x4000
#define CMOV_FLAG0x8000
#define PAT_FLAG0x10000
#define PSE36_FLAG0x20000
#define PSNUM_FLAG0x40000
#define CLFLUSH_FLAG0x80000
#define DTS_FLAG0x200000
#define ACPI_FLAG0x400000
#define MMX_FLAG0x800000
#define FXSR_FLAG0x1000000
#define SSE_FLAG0x2000000
```


```
#define SSE2_FLAG0x4000000
#define SS_FLAG0x8000000
#define HTT_FLAG0x10000000
#define TM_FLAG0x20000000
#define IA64_FLAG0x40000000
#define PBE_FLAG0x80000000

#define SSE3_FLAG0x0001
#define MONITOR_FLAG0x0008
#define DS_CPL_FLAG0x0010
#define VMX_FLAG0x0020
#define SMX_FLAG0x0040
#define EIST_FLAG0x0080
#define TM2_FLAG0x0100
#define SSSE3_FLAG0x0200
#define CID_FLAG0x0400
#define CX16_FLAG0x2000
#define XTPR_FLAG0x4000
#define PDCM_FLAG0x8000
#define DCA_FLAG0x40000
#define SSE4_1_FLAG0x80000
#define SSE4_2_FLAG0x100000

#define LAHF_FLAG0x00000001

#define SYSCALL_FLAG0x00000800
#define XD_FLAG0x00100000
#define EM64T_FLAG0x20000000


extern char cpu_type;
extern char fpu_type;
extern char cpuid_flag;
extern char intel_CPU;
extern char vendor_id[12];

extern long cpu_signature;
extern long features_ecx;
extern long features_edx;
extern long features_ebx;

extern long cache_eax;
extern long cache_ebx;
extern long cache_ecx;
extern long cache_edx;

extern long dcp_cache_eax;
extern long dcp_cache_ebx;
extern long dcp_cache_ecx;
extern long dcp_cache_edx;

extern long ext_funct_1_eax;
extern long ext_funct_1_ebx;
extern long ext_funct_1_ecx;
extern long ext_funct_1_edx;

extern char brand_string[48];
extern int  brand_id;

long cache_temp;
long celeron_flag;
long pentiumxeon_flag;
```

```
struct brand_entry {
    long    brand_value;
    char    *brand_string;
};

#define brand_table_size 20

struct brand_entry brand_table[brand_table_size] = {
        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",
};

void    get_cpu_type();
void    get_fpu_type();
int     print();

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

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

        printf("This system has a");
        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_size) &&
                        ((features_ebx & 0xff) !=
brand_table[brand_index].brand_value))
                        brand_index++;
                    if (brand_index < brand_table_size) {
                        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: %8X",
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: %2X", (family + (int)cpu_type));
                printf("\nModel:%2X", (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");

                if (features_edx & FPU_FLAG)
                    printf("\nThe processor contains an on-chip FPU");
                if (features_edx & VME_FLAG)
                    printf("\nThe processor supports Virtual Mode Extensions");
                if (features_edx & DE_FLAG)
                    printf("\nThe processor supports the Debugging Extensions");
                if (features_edx & PSE_FLAG)
                    printf("\nThe processor supports Page Size Extensions");
                if (features_edx & TSC_FLAG)
                    printf("\nThe processor supports Time Stamp Counter");
                if (features_edx & MSR_FLAG)
                    printf("\nThe processor supports Model Specific Registers");
                if (features_edx & PAE_FLAG)
                    printf("\nThe processor supports Physical Address Extension");
                if (features_edx & MCE_FLAG)
                    printf("\nThe processor supports Machine Check Exceptions");
                if (features_edx & CX8_FLAG)
                    printf("\nThe processor supports the CMPXCHG8B instruction");
```

```
                if (features_edx & APIC_FLAG)
                    printf("\nThe processor contains an on-chip APIC");
                if (features_edx & SEP_FLAG) {
                    if ((cpu_type == 6) && ((cpu_signature & 0xff) < 0x33))
                        printf("\nThe processor does not support the Fast System
Call");
                    else
                        printf("\nThe processor supports the Fast System Call");
                }
                if (features_edx & MTRR_FLAG)
                    printf("\nThe processor supports the Memory Type Range
Registers");
                if (features_edx & PGE_FLAG)
                    printf("\nThe processor supports Page Global Enable");
                if (features_edx & MCA_FLAG)
                    printf("\nThe processor supports the Machine Check
Architecture");
                if (features_edx & CMOV_FLAG)
                    printf("\nThe processor supports the Conditional Move
Instruction");
                if (features_edx & PAT_FLAG)
                    printf("\nThe processor supports the Page Attribute Table");
                if (features_edx & PSE36_FLAG)
                    printf("\nThe processor supports 36-bit Page Size Extension");
                if (features_edx & PSNUM_FLAG)
                    printf("\nThe processor supports the processor serial number");
                if (features_edx & CLFLUSH_FLAG)
                    printf("\nThe processor supports the CLFLUSH instruction");
                if (features_edx & DTS_FLAG)
                    printf("\nThe processor supports the Debug Trace Store
feature");
                if (features_edx & ACPI_FLAG)
                    printf("\nThe processor supports ACPI registers in MSR space");
                if (features_edx & MMX_FLAG)
                    printf("\nThe processor supports Intel Architecture MMX(TM)
technology");
                if (features_edx & FXSR_FLAG)
                    printf("\nThe processor supports the Fast floating point save
and restore");
                if (features_edx & SSE_FLAG)
                    printf("\nThe processor supports the Streaming SIMD extensions
instructions");
                if (features_edx & SSE2_FLAG)
                    printf("\nThe processor supports the Streaming SIMD extensions 2
instructions");
                if (features_edx & SS_FLAG)
                    printf("\nThe processor supports Self-Snoop");
                if ((features_edx & HTT_FLAG) &&
                    (((features_ebx >> 16) & 0x0FF) / (((dcp_cache_eax >> 26) &
0x3F) + 1) > 1))
                    printf("\nThe processor supports Hyper-Threading Technology");
                if (features_edx & TM_FLAG)
                    printf("\nThe processor supports the Thermal Monitor");
                if (features_edx & IA64_FLAG)
                    printf("\n%s\n%s", "The processor is a member of the Intel(R)
Itanium(R) ",
                           "processor family executing IA32 emulation mode");
                if (features_edx & PBE_FLAG)
                    printf("\nThe processor supports Pending Break Event
signaling");

                if (features_ecx & SSE3_FLAG)
                    printf("\nThe processor supports the Streaming SIMD extensions 3
instructions");
                if (features_ecx & MONITOR_FLAG)
```

```
                    printf("\nThe processor supports the MONITOR and MWAIT
instructions");
                if (features_ecx & DS_CPL_FLAG)
                    printf("\nThe processor supports Debug Store extensions for
branch message ",                storage by CPL");
                if (features_ecx & VMX_FLAG)
                    printf("\nThe processor supports Intel(R) Virtualization
Technology");
                if (features_ecx & SMX_FLAG)
                    printf("\nThe processor supports Intel(R) Trusted Execution
Technology");
                if (features_ecx & EIST_FLAG)
                    printf("\nThe processor supports Enhanced SpeedStep(R)
Technology");
                if (features_ecx & TM2_FLAG)
                    printf("\nThe processor supports the Thermal Monitor 2");
                if (features_ecx & SSSE3_FLAG)
                    printf("\nThe processor supports the Supplemental Streaming SIMD
extensions 3 ",              instructions");
                if (features_ecx & CID_FLAG)
                    printf("\nThe processor supports L1 Data Cache Context ID");
                if (features_ecx & CX16_FLAG)
                    printf("\nThe processor supports the CMPXCHG16B instruction");
                if (features_ecx & XTPR_FLAG)
                    printf("\nThe processor supports transmitting TPR messages");
                if (features_ecx & PDCM_FLAG)
                    printf("\nThe processor supports the Performance Capabilities
MSR");
                if (features_ecx & DCA_FLAG)
                    printf("\nThe processor supports the Direct Cache Access
feature");
                if (features_ecx & SSE4_1_FLAG)
                    printf("\nThe processor supports the Streaming SIMD extensions
4.1 instructions");
                if (features_ecx & SSE4_2_FLAG)
                    printf("\nThe processor supports the Streaming SIMD extensions
4.2 instructions");

                if (ext_funct_1_ecx & LAHF_FLAG)
                    printf("\nThe processor supports the LAHF & SAHF instructions");
                if (ext_funct_1_edx & SYSCALL_FLAG)
                    printf("\nThe processor supports the SYSCALL & SYSRET
instructions");
                if (ext_funct_1_edx & XD_FLAG)
                    printf("\nThe processor supports the Execute Disable Bit ");
                if (ext_funct_1_edx & EM64T_FLAG)
                    printf("\nThe processor supports Intel(R) Extended Memory 64
Technology ");
            }
            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 at this
time.");
            }
        }
        printf("\n");
        return(0);
}
```

Example 10-4.Detecting Denormals-Are-Zero Support

```
;       Filename:  DAZDTECT.ASM
;       Copyright (c) Intel Corporation 2001-2008
;
;       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 8 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 XMM feature flag (EDX bit 25) or the EMM 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 segment

        .DATA

buffer  DB  512+16 DUP (0)

not_intelDB"This is not an Genuine Intel processor.", 0Dh, 0Ah, "$"
noSSEorSSE2DB"Neither SSE or SSE2 extensions are supported.", 0Dh, 0Ah, "$"
no_FXSAVEDB"FXSAVE not supported.", 0Dh, 0Ah, "$"
daz_mask_clearDB"DAZ bit in MXCSR_MASK is zero (clear).", 0Dh, 0Ah, "$"
no_daz  DB  "DAZ mode not supported.", 0Dh, 0Ah, "$"
supports_dazDB"DAZ mode supported.", 0Dh, 0Ah, "$"


; Code segment

        .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, 3, and 4

        ; Get CPU feature flags
        ; Verify FXSAVE and either SSE or
        ; SSE2 are supported

        mov eax, 1
        cpuid
        bt  edx, 24t; Feature Flags Bit 24 is FXSAVE support
        jnc noFxsave; jump if FXSAVE not supported

        bt  edx, 25t; Feature Flags Bit 25 is SSE support
        jc  sse_or_sse2_supported; jump if SSE is not supported

        bt  edx, 26t; 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, 16t; DI is aligned at 16-byte boundary

; Step 5.

        ; Clear the buffer that will be
        ; used for FXSAVE data

        pushds
        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, 6t ; 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, 09h; Execute DOS print string function
        int 21h

        mov dx, OFFSET no_daz

print:
        mov ah, 09h; Execute DOS print string function
```

```
        int 21h

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

dazdtectENDP

        END
```

Example 10-5.Frequency Detection

```
;       Filename:  CPUFREQ.ASM
;       Copyright(c) 2003 - 2008 by Intel Corporation
;
;       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 8 steps to determine the
;       processor actual frequency.
;
; 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 to load the
;               EDX register 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.Read the TSC at the beginning of the reference period
; Step 5.Read the TSC at the end of the reference period.
; Step 6.Compute the TSC delta from the beginning and ending
;               of the reference period.
; Step 7.Compute the actual frequency by dividing the TSC
;               delta by the reference period.
;
;*****************************************************************

        .DOSSEG
        .MODEL  small, pascal
        .STACK

include cpufreq.inc

SEG_BIOS_DATA_AREAEQU040h
OFFSET_TICK_COUNTEQU06ch
INTERVAL_IN_TICKSEQU091t; 18.2 * 5 seconds

PMG_PST_MCNTEQU0E7h

; Code segment

        .CODE
        .686p
```

```
;---------------------------------------------------------------
; Function cpufreq
;       This function calculates the Actual and Rounded frequency of
;       the processor.
;
; Input:    None
;
; Destroys: EAX, EBX, ECX, EDX
;
; Output:   AX = Measured Frequency
;                 BX = Reported Frequency
;
; Assumes:  Stack is available
;
;---------------------------------------------------------------
cpufreq PROC NEAR
        localtscLoDword:DWORD, \
                tscHiDword:DWORD, \
                mhz:WORD, \
                Nearest66Mhz:WORD,\
                Nearest50Mhz:WORD,\
                delta66Mhz:WORD

; Step 1.

;Verify Genuine Intel processor by checking CPUID generated
; vendor ID

        mov eax, 0
        cpuid

        cmp ebx, 'uneG'; Check VendorID = GenuineIntel
        jne exit                        ; not Genuine Intel processor
        cmp edx, 'Ieni'
        jne exit
        cmp ecx, 'letn'
        jne exit


; Step 2 and 3

        ; Get CPU feature flags
        ; Verify TSC is supported

        mov eax, 1
        cpuid
        bt  edx, 4t                     ; Flags Bit 4 is TSC support
        jnc exit                        ; jump if TSC not supported

        and eax, 0FFF3FF0h
        cmp eax, 000006F0h
        je  _4a_to_5a


        pushSEG_BIOS_DATA_AREA
        pop es
        mov si, OFFSET_TICK_COUNT; The BIOS tick count updates
        mov ebx, DWORD PTR es:[si]; ~ 18.2 times per second.

wait_for_new_tick:

        cmp ebx, DWORD PTR es:[si]; Wait for tick count change
```

```
        je  wait_for_new_tick

; Step 4
        ; **Timed interval starts**

        ; Read CPU time stamp
        rdtsc                           ; Read & save TSC immediately
        mov tscLoDword, eax; after a tick
        mov tscHiDword, edx

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

wait_for_elapsed_ticks:

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

; Step 5
        ; **Time interval ends**

        ; Read CPU time stamp immediately after tick delay reached.
        rdtsc

step_6:
; Step 6

        sub eax, tscLoDword; Calculate TSC delta from
        sbb edx, tscHiDword; beginning to end of
                    ; interval

        jmp step_7

_4a_to_5a:

        mov ecx, PMG_PST_MCNT; MSR containing C0_MCNT
        xor eax, eax
        xor edx, edx

        pushSEG_BIOS_DATA_AREA
        pop es
        mov si, OFFSET_TICK_COUNT; The BIOS tick count updates
        mov ebx, DWORD PTR es:[si]; ~ 18.2 times per second.


wait_for_new_tick_4a:
        cmp ebx, DWORD PTR es:[si]; Wait for tick count change
        je  wait_for_new_tick_4a

; Step 4a
        ; **Timed interval starts**
        ; Zero the C0_MCNT
        wrmsr       ; Write 0 to C0_MCNT timer

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

elapsed_ticks_4a:

        cmp ebx, DWORD PTR es:[si]; Have we hit the delay?
        jne elapsed_ticks_4a
```

```
; Step 5a
        ; **Time interval ends**
        ; Read C0_MCNT immediately after tick delay reached.
        rdmsr

; Step 7
        ;
        ; 54945 = (1 / 18.2) * 1,000,000  This adjusts for MHz.
        ; 54945*INTERVAL_IN_TICKS adjusts for number of ticks in
; interval
        ;
step_7:
        mov ebx, 54945*INTERVAL_IN_TICKS
        div ebx

; ax contains measured speed in MHz
        mov mhz, ax

; Find nearest full/half multiple of 66/133 MHz
        xor dx, dx
        mov ax, mhz
        mov bx, 3t
        mul bx
        add ax, 100t
        mov bx, 200t
        div bx
        mul bx
        xor dx, dx
        mov bx, 3
        div bx

; ax contains nearest full/half multiple of 66/100 MHz

        mov Nearest66Mhz, ax
        sub ax, mhz
        jge delta66
        neg ax              ; ax = abs(ax)

delta66:

        ; ax contains delta between actual and nearest 66/133 multiple
        mov Delta66Mhz, ax

        ; Find nearest full/half multiple of 100 MHz
        xor dx, dx
        mov ax, mhz
        add ax, 25t
        mov bx, 50t
        div bx
        mul bx

        ; ax contains nearest full/half multiple of 100 MHz

        mov Nearest50Mhz, ax
        sub ax, mhz
        jge delta50
        neg ax              ; ax = abs(ax)

delta50:

        ; ax contains delta between actual and nearest 50/100 MHz
; multiple
```

```
        mov bx, Nearest50Mhz
        cmp ax, Delta66Mhz
        jb  useNearest50Mhz
        mov bx, Nearest66Mhz

        ; Correction for 666 MHz (should be reported as 667 MHZ)

        cmp bx, 666
        jne correct666
        inc bx

correct666:
useNearest50MHz:

        ; bx contains nearest full/half multiple of 66/100/133 MHz

exit:

        mov ax, mhz         ; Return the measured
                            ; frequency in AX
        ret

cpufreq     ENDP

            END
```

§