IA-32 Intel® Architecture and Intel® Extended Memory 64 Technology Software Developer's Manual

Documentation Changes

November 2004

Document Number: 252046-011

Contents

Revision History

Version	Description	Date
-001	• Initial Release	November 2002
-002	• Added 1-10 Documentation Changes. • Removed old Documentation Changes items that already have been incorporated in the published Software Developer's manual	December 2002
-003	• Added 9 -17 Documentation Changes. • Removed Documentation Change #6 - References to bits Gen and Len Deleted. • Removed Documentation Change #4 - VIF Information Added to CLI Discussion.	February 2003
-004	• Removed Documentation changes 1-17. • Added Documentation changes 1-24.	June 2003
-005	• Removed Documentation Changes 1-24. • Added Documentation Changes 1-15.	September 2003
-006	• Added Documentation Changes 16- 34.	November 2003
-007	• Updated Documentation changes 14, 16, 17, and 28. • Added Documentation Changes 35-45.	January 2004
-008	• Removed Documentation Changes 1-45. • Added Documentation Changes 1-5.	March 2004
-009	• Added Documentation Changes 7-27.	May 2004
-010	• Removed Documentation Changes 1-27. • Added Documentation Changes 1.	August 2004
-011	• Added Documentation Changes 2-28.	November 2004

Preface

This document is an update to the specifications contained in the Affected Documents/Related Documents table below. This document is a compilation of documentation changes. It is intended for hardware system manufacturers and software developers of applications, operating systems, or tools.

Affected Documents/Related Documents

Document Title	Document Number
IA-32 Intel® Architecture Software Developer's Manual: Volume 1, Basic Architecture	253665
IA-32 Intel® Architecture Software Developer's Manual: Volume 2A, Instruction Set Reference	253666
IA-32 Intel® Architecture Software Developer's Manual: Volume 2B, Instruction Set Reference	253667
IA-32 Intel® Architecture Software Developer's Manual: Volume 3, System Programming Guide	253668
Intel® Extended Memory 64 Technology Software Developer's Guide Volumes 1 and 2	300835

Nomenclature

Documentation Changes include errors or omissions from the current published specifications. These changes will be incorporated in the next release of the Software Development Manual.

Summary Table of Changes

The following table indicates documentation changes which apply to the IA-32 Intel Architecture. This table uses the following notations:

Codes Used in Summary Table

Change bar to left of table row indicates this erratum is either new or modified from the previous version of the document.

Summary Table of Documentation Changes

Number	Documentation Changes
1	The mechanism for handling the Extended Family ID and Extended Model ID fields has been updated
2	INT n/INTO/INT 3—Call to Interrupt Procedure Section Updated
3	Table 1-37 Corrected
4	Appendix E Corrected
5	Chapter 15 Corrected
6	IA-32e Updates for CVTSI2SD, CVTSI2SS, CVTTSD2SI
7	Table 9-6 Corrected
8	Figure 4-6 Corrected
9	Table B-1 Updated
10	POPF/POPFD—Pop Stack into EFLAGS Section Updated
11	LAR and LSL Sections Updated
12	Chapter 15 and Appendix B Updated
13	CPUID Leaf 4 Section Updated
14	Bit(BitBase, BitOffset) Subsection Updated
15	CPUID Section Updated
17	INVLPG Description Information Updated
18	LEA Summary Table Updated
19	IA32_MCi_CTL MSRs Section Updated
20	Section 7.7.5 Updated
21	CALL and JMP Summary Tables Updated
22	Memory Options Table Updated
23	LAHF/SAHF Section Updated
24	Chapter 3 Corrected
25	Appendix E Corrected
26	Section 7.1 Corrected
27	Table 10-5 Corrected
28	Section 10.3 Corrected

Documentation Changes

1. The mechanism for handling the Extended Family ID and Extended Model ID fields has been updated

In Chapter 3, *IA-32 Intel Architecture Software Developer's Manual, Volume 2B*; the CPUID section has been updated. Text defining the recommended mechanisms for handling the Extended Family ID and the Extended model ID fields have been updated. The updated text is reproduced below.

The Extended Family ID is to be examined only if the Family ID is 0FH; the Extended Model ID is to be examined only if the Family ID is 0FH or 06H. Often software displays processor information as a combination of family, model and stepping.

The recommended mechanism for integrating Family ID fields into a display follows:

Displayed_Family = (Extended_Family_ID) (8-bits) + Family_ID (4-bits zero extended to 8-bits)

The recommended mechanism for integrating Model ID fields into a display follows:

Displayed_Model = ((Extended_Model_ID (4-bits) << 4))(8-bits) + Model_ID (4-bits zero extended to 8-bits)

2. *INT n/INTO/INT 3—Call to Interrupt Procedure* Section Updated

IA-32 Intel Architecture Software Developer's Manual, Volume 2A; Chapter 3, *INT n/INTO/INT 3—Call to Interrupt Procedure* section: text has been re-written. The updated text is marked by a change bar.

When the processor is executing in virtual-8086 mode, the IOPL determines the action of the INT *n* instruction. If the IOPL is less than 3, the processor generates a #GP(selector) exception; if the IOPL is 3, the processor executes a protected mode interrupt to privilege level 0. The interrupt gate's DPL must be set to 3 and the target CPL of the interrupt handler procedure must be 0 to execute the protected mode interrupt to privilege level 0.

The interrupt descriptor table register (IDTR) specifies the base linear address and limit of the IDT. The initial base address value of the IDTR after the processor is powered up or reset is 0.

3. Table 1-37 Corrected

Table 1-37, *Intel® Extended Memory 64 Technology Software Developer's Guide, Volume 1* has been corrected. The updated table is reprinted below.

Table 1-37 TSS Format in IA-32e Mode

Byte Offset	31:16	15:0
+64H	I/O Map Base	Reserved
+60H	Reserved	
+5CH	Reserved	
+58H	IST7 (upper 32-bits)	
+54H	IST7 (lower 32-bits)	
+50H	IST6 (upper 32-bits)	
+4CH	IST6 (lower 32-bits)	
+48H	IST5 (upper 32-bits)	
+44H	IST5 (lower 32-bits)	
+40H	IST4 (upper 32-bits)	
+3CH	IST4 (lower 32-bits)	
+38H	IST3 (upper 32-bits)	
+34H	IST3 (lower 32-bits)	
+30H	IST2 (upper 32-bits)	
+2CH	IST2 (lower 32-bits)	
+28H	IST1 (upper 32-bits)	
+24H	IST1 (lower 32-bits)	
+20H	Reserved	
+1CH	Reserved	
+18H	RSP2 (upper 32-bits)	
+14H	RSP2 (lower 32-bits)	
+10H	RSP1 (upper 32-bits)	
+0CH	RSP1 (lower 32-bits)	
+08H	RSP0 (upper 32-bits)	
+04H	RSP0 (lower 32-bits)	
00H	Reserved	

4. Appendix E Corrected

Intel® Extended Memory 64 Technology Software Developer's Guide, Volume 2; Appendix e: a description of the SMRAM state save map has been added in table form. Change bars mark the additions.

E.1 SMRAM STATE SAVE MAP

When the processor initially enters SMM, it writes its state to the state save area of the SMRAM. The state save area begins at [SMBASE + 8000H + 7FFFH] and extends down to [SMBASE + 8000H + 7C00H]. If the processor reports CPUID.80000001.EDX[29] = 1, The layout of the SMRAM state save map is shown in Table E-1.

Offset (Added to SMBASE + 8000H)	Register
7FF8H	CR0
7FF0H	CR3
7FFE8	RFLAGS
7FE0H	IA32_EFER
7FD8H	RIP
7FD0H	DR6
7FC8H	DR7
7FC4H	TR SEL
7FC0H	LDTR SEL
7FBCH	GS SEL
7FB8H	FS SEL
7FB4H	DS SEL
7FB0H	SS SEL
7FACH	CS SEL
7FA8H	ES SEL
7FA4H	IO_MISC
7F9CH	IO_MEM_ADDR
7F94H	RDI
7F8CH	RSI
7F84H	RBP
7F7CH	RSP
7F74H	RBX
7F6CH	RDX
7F64H	RCX
7F5CH	RAX
7F54H	R8
7F4CH	R9
7F44H	R10
7F3CH	R11
7F34H	R12
7F2CH	R13

Offset (Added to SMBASE + 8000H)	Register
7F24H	R14
7F1CH	R15
7F08H-7F1BH	Reserved
7F04H	IEDBASE
7F02H	I/O Instruction Restart Field (Word)
7F00H	Auto HALT Restart Field (Word)
7EFCH	SMM Revision Identifier Field (Doubleword)
7EF8H	SMBASE Field (Doubleword)
7EF7H - 7EA8H	Reserved
7EA4H	LDT Info
7EA0H	LDT Limit
7E9CH	LDT Base (Lower 32 bits)
7E98H	IDT Limit
7E94H	IDT Base (Lower 32 bits)
7E90H	GDT Limit
7E8CH	GDT Base (Lower 32 bits)
7E8BH - 7E44H	Reserved
7E40H	CR4
7E3FH - 7DF0H	Reserved
7DE8H	IO_EIP
7DE7H - 7DDCH	Reserved
7DD8H	IDT Base (Upper 32 bits)
7DD4H	LDT Base (Upper 32 bits)
7DD0H	GDT Base (Upper 32 bits)
7DCFH - 7C00H	Reserved

5. Chapter 15 Corrected

IA-32 Intel Architecture Software Developer's Manual, Volume 3, Chapter 15: text has been added to clarify PMI mask conditions. Changes bars mark the additions (with context paragraphs also included).

Section 15.5.7.3 (text is at the end of the section):

- The ISR must clear the mask bit in the performance counter LVT entry.
- The ISR must re-enable the CCCR's ENABLE bit if it is servicing an overflow PMI due to PEBS.
- The Pentium 4 Processor and Intel Xeon Processor mask PMIs upon receiving an interrupt. Clear this condition before leaving the interrupt handler.

Section 15.10.6.9 (text is in the middle of section):

To enable an interrupt on counter overflow, the OVR_PMI flag in the counter's associated CCCR MSR must be set. When overflow occurs, a PMI is generated through the local APIC. (Here, the performance counter entry in the local vector table [LVT] is set up to deliver the interrupt generated by the PMI to the processor.)

The PMI service routine can use the OVF flag to determine which counter overflowed when multiple counters have been configured to generate PMIs. Also, note that these processors mask PMIs upon receiving an interrupt. Clear this condition before leaving the interrupt handler.

When generating interrupts on overflow, the performance counter being used should be preset to value that will cause an overflow after a specified number of events are counted plus 1. The simplest way to select the preset value is to write a negative number into the counter, as described in Section 15.10.6.6., "Cascading Counters". Here, however, if an interrupt is to be generated after 100 event counts, the counter should be preset to minus 100 plus 1 (-100 + 1), or -99. The counter will then overflow after it counts 99 events and generate an interrupt on the next (100th) event counted. The difference of 1 for this count enables the interrupt to be generated immediately after the selected event count has been reached, instead of waiting for the overflow to be propagation through the counter.

6. IA-32e Updates for CVTSI2SD, CVTSI2SS, CVTTSD2SI

For CVTS12SD - CVTTSD2SI; Chapter 2 in the *Intel® Extended Memory 64 Technology Software Developer's Guide, Volume 1*; IA-32e Mode Operation section have been updated. Change bars mark the corrections.

CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F2 0F 2A /r	CVTSI2SD *xmm*, r/ m32	Valid	Valid	Convert one signed doubleword integer from *r/m32* to one double-precision floating-point value in *xmm*.
REX.W + F2 0F 2A /r	CVTSI2SD *xmm*, r/ m64	Valid	N.E.	Convert one signed quadword integer from *r/m64* to one double-precision floating-point value in *xmm*.

IA-32e Mode Operation

Enables access to XMM8-XMM15.
Promoted to 64 bits.
Enables access to new registers R8-R15.

CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F 2A /r	CVTSI2SS *xmm*, *r/ m32*	Valid	Valid	Convert one signed doubleword integer from *r/m32* to one single-precision floating-point value in *xmm*.
REX.W + F3 0F 2A /r	CVTSI2SS *xmm*, *r/ m64*	Valid	N.E.	Convert one signed quadword integer from *r/m64* to one single-precision floating-point value in *xmm*.

IA-32e Mode Operation

Enables access to XMM8-XMM15. XMMn[31:0] = CVT(reg/mem64), XMMn[127:32] = unchanged.

Promoted to 64 bits.

Enables access to new registers R8-R15

CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F 2A /r	CVTSI2SS *xmm*, *r/ m32*	Valid	Valid	Convert one signed doubleword integer from *r/m32* to one single-precision floating-point value in *xmm*.
REX.W + F3 0F 2A /r	CVTSI2SS *xmm*, *r/ m64*	Valid	N.E.	Convert one signed quadword integer from *r/m64* to one single-precision floating-point value in *xmm*.

IA-32e Mode Operation

Enables access to XMM8-XMM15. XMMn[31:0] = CVT(reg/mem64), XMMn[127:32] = unchanged.

Promoted to 64 bits.

Enables access to new registers R8-R15.

....

7. Table 9-6 Corrected

In Table 9-6, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*; the microcode update checksum has been corrected. The corrected cells are marked by change bars.

Field Name	Offset (bytes)	Length (bytes)	Description
Header Version	0	4	Version number of the update header.
Update Revision	4	4	Unique version number for the update, the basis for the update signature provided by the processor to indicate the current update functioning within the processor. Used by the BIOS to authenticate the update and verify that the processor loads successfully. The value in this field cannot be used for processor stepping identification alone. This is a signed 32-bit number.
Date	8	4	Date of the update creation in binary format: mmddyyyy (e.g. 07/18/98 is 07181998H).
Processor Signature	12	4	*Extended family, extended model, type, family, model*, and *stepping* of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, *type, family, model*, and *stepping* of the processor. The BIOS uses the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.
Checksum	16	4	Checksum of Update Data and Header. Used to verify the integrity of the update header and data. Checksum is correct when the summation of all DWORDs that comprise the microcode update result in 00000000H. Note that extended processor signature table, by itself, should checksum to zero.
Loader Revision	20	4	Version number of the loader program needed to correctly load this update. The initial version is 00000001H.
Processor Flags	24	4	Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. The BIOS uses the processor flags field in conjunction with the platform Id bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.
Data Size	28	4	Specifies the size of the encrypted data in bytes, and must be a multiple of DWORDs. If this value is 00000000H, then the microcode update encrypted data is 2000 bytes (or 500 DWORDs).
Total Size	32	4	Specifies the total size of the microcode update in bytes. It is the summation of the header size, the encrypted data size and the size of the optional extended signature table.
Reserved	36	12	Reserved fields for future expansion
Update Data	48	Data Size or 2000	Update data
Extended Signature Count	Data Size + 48	4	Specifies the number of extended signature structures (Processor Signature[n], processor flags[n] and checksum[n]) that exist in this microcode update.

Field Name	Offset (bytes)	Length (bytes)	Description
Extended Checksum	Data Size + 52	4	Checksum of update extended processor signature table. Used to verify the integrity of the extended processor signature table. Checksum is correct when the summation of the DWORDs that comprise the extended processor signature table results in 00000000H.
Reserved	Data Size + 56	12	Reserved fields
Processor Signature[n]	Data Size + 68 + (n * 12)	4	*Extended family, extended model, type, family, model*, and *stepping* of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, *type, family, model*, and *stepping* of the processor. The BIOS uses the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.
Processor Flags[n]	Data Size + 72 + (n * 12)	4	Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. The BIOS uses the processor flags field in conjunction with the platform Id bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.
Checksum[n]	Data Size + 76 + (n * 12)	4	Used by utility software to decompose a microcode update into multiple microcode updates where each of the new updates is constructed without the optional Extended Processor Signature Table. To calculate the Checksum, substitute the Primary Processor Signature entry and the Processor Flags entry with the corresponding Extended Patch entry. Delete the Extended Processor Signature Table entries. The Checksum is correct when the summation of all DWORDs that comprise the created Extended Processor Patch results in 00000000H.

8. Figure 4-6 Corrected

IA-32 Intel Architecture Software Developer's Manual, Volume 3; Figure 4-6: a line that incorrectly indicated privilege access was removed. The corrected figure is reprinted below.

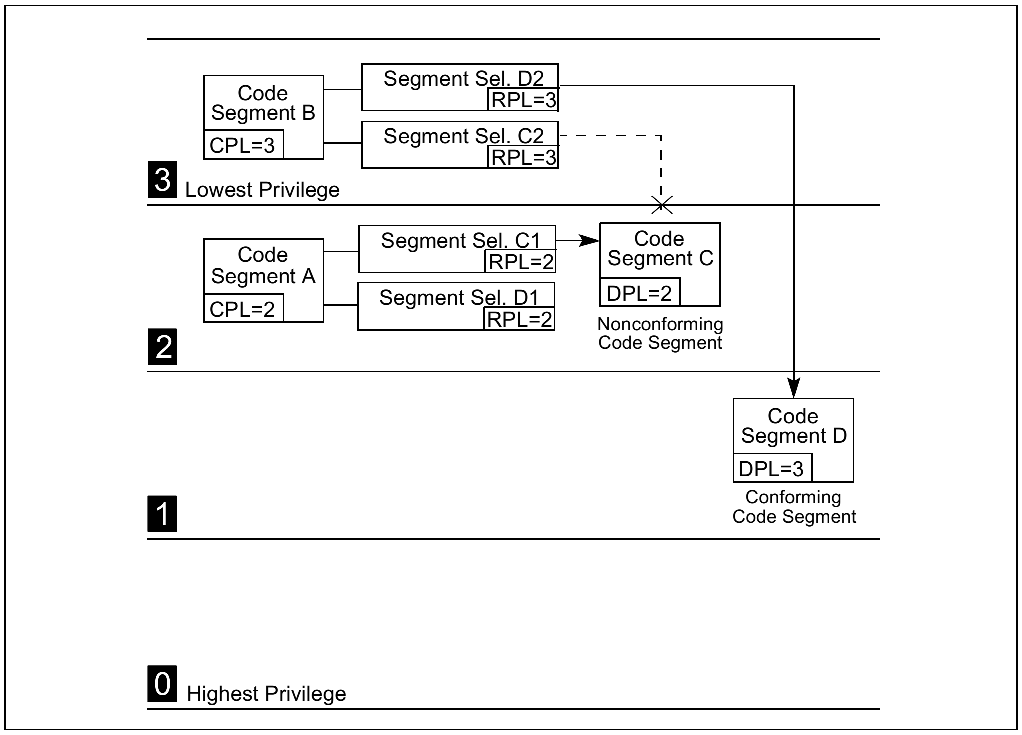

Figure 4-6. Examples of Accessing Conforming and Nonconforming Code Segments From Various Privilege Levels

9. Table B-1 Updated

In Table B-1, MSR_EBC_FREQUENCY_ID has been updated to address the Core Clock Frequency to System Bus Frequency Ratio entry (see bits 19-63). Updated text are marked by change bars.

Register Address		Register Name Fields and Flags	Model Availability	Shared/ Unique[1]	Bit Description
Hex	Dec				
....					
2CH	44	MSR_EBC_ FREQUENCY_ID	2,3	Shared	**Processor Frequency Configuration.** The bit field layout of this MSR varies according to the MODEL value in the CPUID version information. The following bit field layout applies to Pentium 4 and Xeon Processors with MODEL encoding equal or greater than 2. (R) The field Indicates the current processor frequency configuration.

		15:0			**Reserved.**
		18:16			**Scalable Bus Speed.** (R/W) Indicates the intended scalable bus speed: Encoding Scalable Bus Speed 000B 100 MHz 001B 133 MHz 010B 200 MHz 011B 166 MHz 133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B All Others Reserved
		23:19			**Reserved**
		31:24			**Core Clock Frequency to System Bus Frequency Ratio.** (R) The processor core clock frequency to system bus frequency ratio observed at the de-assertion of the reset pin.
		63:25			**Reserved.**
....					

10. POPF/POPFD—Pop Stack into EFLAGS Section Updated

IA-32 Intel Architecture Software Developer's Manual, Volume 2B; Chapter 4, *POPF/POPFD—Pop Stack into EFLAGS Register* section: text has been updated to clarify the instructions' impact on the EFLAGS register. Updated text are marked by change bars (additional material is included to show the context).

Description

Pops a doubleword (POPFD) from the top of the stack (if the current operand-size attribute is 32) and stores the value in the EFLAGS register, or pops a word from the top of the stack (if the operand-size attribute is 16) and stores it in the lower 16 bits of the EFLAGS register (that is, the FLAGS register). These instructions reverse the operation of the PUSHF/PUSHFD instructions.

The POPF (pop flags) and POPFD (pop flags double) mnemonics reference the same opcode. The POPF instruction is intended for use when the operand-size attribute is 16 and the POPFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when POPF is used and to 32 when POPFD is used. Others may treat these mnemonics as synonyms (POPF/POPFD) and use the current setting of the operand-size attribute to determine the size of values to be popped from the stack, regardless of the mnemonic used.

The effect of the POPF/POPFD instructions on the EFLAGS register changes slightly, depending on the mode of operation of the processor. When the processor is operating in protected mode at privilege level 0 (or in real-address mode, which is equivalent to privilege level 0), all the nonreserved flags in the EFLAGS register except the VIP, VIF, and VM flags can be modified. The VIP, VIF and VM flags remain unaffected.

When operating in protected mode, with a privilege level greater than 0, but less than or equal to IOPL, all the flags can be modified except the IOPL field and the VIP, VIF, and VM flags. Here, the IOPL flags are unaffected, the VIP and VIF flags are cleared, and the VM flag is unaffected. The interrupt flag (IF) is altered only when executing at a level at least as privileged as the IOPL. If a POPF/POPFD instruction is executed with insufficient privilege, an exception does not occur but the privileged bits do not change.

When operating in virtual-8086 mode, the I/O privilege level (IOPL) must be equal to 3 to use POPF/POPFD instructions and the VM, RF, IOPL, VIP, and VIF flags are unaffected. If the IOPL is less than 3, the POPF/POPFD instructions cause a general-protection exception (#GP).

11. LAR and LSL Sections Updated

IA-32 Intel Architecture Software Developer's Manual, Volume 2A; Chapter 3, LAR and LSL sections: summary data has been updated. Updated text are marked by change bars.

From the LAR section in Chapter 3:

LAR—Load Access Rights Byte

Opcode	**Instruction**	**64-Bit Mode**	**Compat/ Leg Mode**	**Description**
0F 02 /*r*	LAR *r16, r16/m16*	Valid	Valid	*r16* ← *r16/m16* masked by FF00H.
0F 02 /*r*	LAR *r32, r32/m16*[1]	Valid	Valid	*r32* ← *r32/m16* masked by 00FxFF00H
REX + 0F 02 /*r*	LAR *r64, r32/m16*[1]	Valid	N.E.	*r64* ← *r32/m16* masked by 00FxFF00H

NOTES:
1. For all loads (regardless of source or destination sizing), only bits 16-0 are used. Other bits are ignored.

From the LSL section in Chapter 3:

LSL—Load Segment Limit

Opcode	**Instruction**	**64-Bit Mode**	**Compat/ Leg Mode**	**Description**
0F 03 /*r*	LSL *r16, r16/m16*	Valid	Valid	Load: *r16* ← segment limit, selector *r16/m16.*
0F 03 /*r*	LSL *r32, r32/m16*[1]	Valid	Valid	Load: *r32* ← segment limit, selector *r32/m16.*
REX.W + 0F 03 /*r*	LSL *r64, r32/m16*[1]	Valid	Valid	Load: *r64* ← segment limit, selector *r32/m64*

NOTES:
1. For all loads (regardless of destination sizing), only bits 16-0 are used. Other bits are ignored.

12. Chapter 15 and Appendix B Updated

IA-32 Intel Architecture Software Developer's Manual, Volume 3; Chapter 15 and Appendix B, the IA32_TIME_STAMP_COUNTER information has been updated. Updated text is marked by change bars.

The information details differences in operation between new and older processors.

From *IA-32 Intel Architecture Software Developer's Manual, Volume 3, Chapter 15,*

15.1 Time-Stamp Counter

The IA-32 architecture (beginning with the Pentium processor) defines a time-stamp counter mechanism that can be used to monitor and identify the relative time of occurrence of processor events. The time-stamp counter architecture includes the time-stamp counter (IA32_TIME_STAMP_COUNTER MSR [called the TSC MSR in the P6 family and Pentium processors]), an instruction for reading the time-stamp counter (RDTSC), a feature bit (TCS flag) that can be read with the CPUID instruction, and a time-stamp counter disable bit (TSD flag) in control register CR4.

Following execution of the CPUID instruction, the TSC flag in register EDX (bit 4) indicates (when set) that the time-stamp counter is present in a particular IA-32 processor implementation. (See "CPUID—CPU Identification" in Chapter 3 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 2*.)

The time-stamp counter (as implemented in the Pentium 4, Intel Xeon, P6 family, and Pentium processors) is a 64-bit counter that is set to 0 following the hardware reset of the processor. Following reset, the counter is incremented every processor clock cycle, even when the processor is halted by the HLT instruction or the external STPCLK# pin. However, the assertion of the external DPSLP# pin may cause the time-stamp counter to stop and Intel® SpeedStep® technology transitions may cause the frequency at which the time-stamp counter increments to change in accordance with the processor's internal clock frequency.

The RDTSC instruction reads the time-stamp counter and is guaranteed to return a monotonically increasing unique value whenever executed, except for 64-bit counter wraparound. Intel guarantees, architecturally, that the time-stamp counter frequency and configuration will be such that it will not wraparound within 10 years after being reset to 0. The period for counter wrap is much longer for Pentium 4, Intel Xeon, P6 family, and Pentium processors.

Normally, the RDTSC instruction can be executed by programs and procedures running at any privilege level and in virtual-8086 mode. The TSD flag in control register CR4 (bit 2) allows use of this instruction to be restricted to only programs and procedures running at privilege level 0. A secure operating system would set the TSD flag during system initialization to disable user access to the time-stamp counter. An operating system that disables user access to the time-stamp counter should emulate the instruction through a user-accessible programming interface.

The RDTSC instruction is not serializing or ordered with other instructions. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDTSC instruction operation is performed.

The RDMSR and WRMSR instructions can read and write the time-stamp counter, respectively, as an MSR (MSR address 10H). In the Pentium 4, Intel Xeon, and P6 family processors, all 64-bits of the time-stamp counter can be read with the RDMSR instruction (just as with the RDTSC instruction). When the WRMSR instruction is used to write to the time-stamp counter on processors before family 0FH, models 3 and 4: only the low order 32-bits of the time-stamp counter can be written (the high-order 32 bits are cleared to 0). For family 0FH, models 3 and 4: all bits are writeable.

From *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, *Table B-1,*

Register Address		**Register Name Fields and Flags**	**Model Avail-ability**	**Shared/ Unique**[1]	**Bit Description**
Hex	**Dec**				
...					
10H	16	IA32_TIME_STAMP_ COUNTER	0, 1, 2, 3	Unique	Time Stamp Counter. See Section 15.8.
		63:0			**Timestamp Count Value.** A 64-bit register accessed when referenced as a Qword through a RDMSR, WRMSR or RDTSC instruction. Returns the current time stamp count value. All 64 bits are readable. On earlier processors, only the lower 32 bits are writeable. On any write to the lower 32 bits, the upper 32 bits are cleared. For processor family 0FH, models 3 and 4: all 64 bits are writeable.
....					

13. CPUID Leaf 4 Section Updated

IA-32 Intel Architecture Software Developer's Manual, Volume 2A; Chapter 3, CPUID—CPU Identification section: the footnotes associated with Table 3-12, under *Deterministic Cache Parameters Leaf* heading, will be re-positioned and re-numbered for clarity. The affected cells in the table are marked by change bars.

Initial EAX Value		Information Provided about the Processor
.. ..		
04H		*Deterministic Cache Parameters Leaf*
	EAX	Bits 4-0: Cache Type* Bits 7-5: Cache Level (starts at 1) Bits 8: Self Initializing cache level (does not need SW initialization) Bits 9: Fully Associative cache Bits 13-10: Reserved Bits 25-14: Number of threads sharing this cache (see note)** Bits 31-26: Number of processor cores on this die**
	EBX	Bits 11-00: L = System Coherency Line Size** Bits 21-12: P = Physical Line partitions** Bits 31-22: W = Ways of associativity**
	ECX	Bits 31-00: S = Number of Sets** * **Cache Type fields:** **0 = Null - No more caches** **3 = Unified Cache** **1 = Data Cache** **4-31 = Reserved** **2 = Instruction Cache** ** **Add one to the value in the register to get the number. For example, the number of processor cores is EAX[31:26]+1.**
	EDX	Reserved = 0 **NOTE:** CPUID leaves > 3 < 80000000 are only visible when IA32_MISC_ENABLES.BOOT_NT4 (bit 22) is clear (default)
... ...		

14. Bit(BitBase, BitOffset) Subsection Updated

IA-32 Intel Architecture Software Developer's Manual, Volume 2A; Section 3.1.1.7: text in the Bit(BitBase, BitOffset) subsection will be updated to accommodate 64 bits. Data in this section impacts the following instructions: BT, BTC, BTR and BCS. All affected areas are marked with change bars.

- **Bit(BitBase, BitOffset)** — Returns the value of a bit within a bit string. The bit string is a sequence of bits in memory or a register. Bits are numbered from low-order to high-order within registers and within memory bytes. If the BitBase is a register, the BitOffset can be in the range 0 to [15, 31, 63] depending on the mode and register size. See Figure 3-1: the function Bit[RAX, 21] is illustrated.

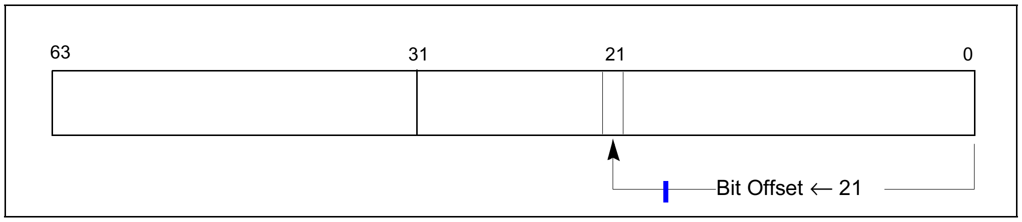

Figure 3-1. Bit Offset for BIT[RAX, 21]

If BitBase is a memory address, the BitOffset can range has different ranges depending on the operand size (see Table 3-7).

Operand Size	Immediate BitOffset	Register BitOffset
16	0 to 15	-2^{15} to $2^{15} - 1$
32	0 to 31	-2^{31} to $2^{31} - 1$
64	0 to 63	-2^{63} to $2^{63} - 1$

The addressed bit is numbered (Offset MOD 8) within the byte at address (BitBase + (BitOffset DIV 8)) where DIV is signed division with rounding towards negative infinity and MOD returns a positive number (see Figure 3-2).

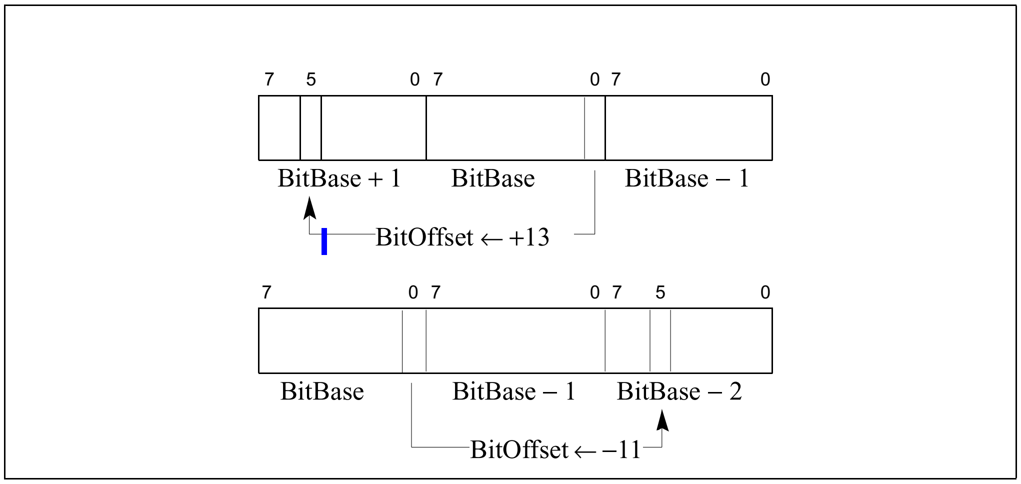

Figure 3-2. Memory Bit Indexing

15. CPUID Section Updated

IA-32 Intel Architecture Software Developer's Manual, Volume 2A; Chapter 3, CPUID section: the subsection on family and model information has been updated. The updated text has been marked by change bars (in context).

INPUT EAX = 1: Returns Model, Family, Stepping Information

\When CPUID executes with EAX set to 1, version information is returned in EAX (see Figure 3-5). For example: model, family, and processor type for the first processor in the Intel Pentium 4 family is returned as follows:

- Model—0000B
- Family—1111B
- Processor Type—00B

See Table 3-14 for available processor type values. Stepping IDs are provided as needed.

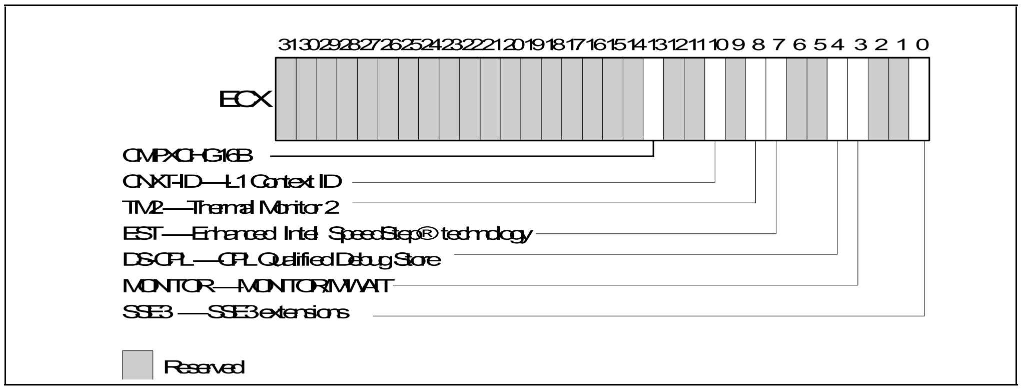

Figure 3-5. Version Information Returned by CPUID in EAX

Table 3-14. Processor Type Field

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

Note:

See AP-485, *Intel Processor Identification and the CPUID Instruction* (Order Number 241618) and Chapter 13 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for information on identifying earlier IA-32 processors.

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

```
IF Family_ID ≠ 0FH
    THEN Displayed_Family = Family_ID;
    ELSE Displayed Family = Extended_Family_ID + Family_ID;
    (* Right justify and zero-extend 4-bit field; display Family ID as HEX Field. *)
FI;
```

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

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

16. Section 8.4.6 Corrected

IA-32 Intel Architecture Software Developer's Manual, Volume 3; Section 8.4.6: text has been added to clarify the APIC ID value returned when CPUID is executed with a value of 1 in EAX. The additional text has been marked by a change bar.

8.4.6. Local APIC ID

In MP systems, the local APIC ID is also used as a processor ID by the BIOS and the operating system. Some processors permit software to modify the APIC ID. However, the ability of software to modify the APIC ID is processor model specific. Because of this, operating system software should avoid writing to the local APIC ID register. **The value returned to bits 31-24 of the EBX register (when the CPUID instruction is executed with a source operand value of 1 in the EAX register) is always the Initial APIC ID (determined by the platform initialization). This is true even if software has changed the value in the Local APIC ID register.**

17. INVLPG Description Information Updated

Intel® Extended Memory 64 Technology Software Developer's Guide, Volume 1; in the INVLPG-Invalidate TLB Entry section: the IA-32e Mode Operation and 64-Bit Mode Exceptions sections have been updated to document a situation in which INVLPG acts as a NOP. The updated text is marked by a change bar.

INVLPG—Invalidate TLB Entry

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 01/7	INVLPG m	Valid	Valid	Invalidate TLB Entry for page that contains *m*

Flags Affected

None.

IA-32e Mode Operation

Same as legacy mode with the following exception. In 64-bit mode if the memory address is in non-canonical form, INVLPG behaves the same as a NOP.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	Operand is a register.

Real-Address Mode Exceptions

#UD	Operand is a register.

Virtual-8086 Mode Exceptions

#GP(0)	The INVLPG instruction cannot be executed at the virtual-8086 mode.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

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

#GP(0)	If the current privilege level is not 0.
#UD	Operand is a register.

18. LEA Summary Table Updated

Intel® Extended Memory 64 Technology Software Developer's Guide, Volume 1; in the *LEA—Load Effective Address* section: the summary table needs to be updated. The affected table cells are marked by a change bar.

LEA—Load Effective Address

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
8D /*r*	LEA *r16,m*	**N.E**	Valid	Store effective address for *m* in register *r16*
8D /*r*	LEA *r32,m*	Valid	Valid	Store effective address for *m* in register *r32*
8D /*r*	LEA *r64,m*	Valid	N.E.	Store effective address for *m* in register *r64. Zero extended 32-bit register results to 64-bits.*

19. IA32_MCi_CTL MSRs Section Updated

IA-32 Intel Architecture Software Developer's Manual, Volume 3; Section 14.3.2.1: text has been updated to clarify MC Reporting/Logging. The updated text is marked by a change bar.

IA32_MCi_CTL MSRs

The IA32_MC*i*_CTL MSR (called MC*i*_CTL in P6 family processors) controls error reporting for errors produced by a particular hardware unit (or group of hardware units). Each of the 64 flags (EE*j*) represents a potential error. Setting an EE*j* flag enables reporting of the associated error and clearing it disables reporting of the error. The processor does not write changes to bits that are not implemented. Figure 14-5 shows the bit fields of IA32_MC*i*_CTL.

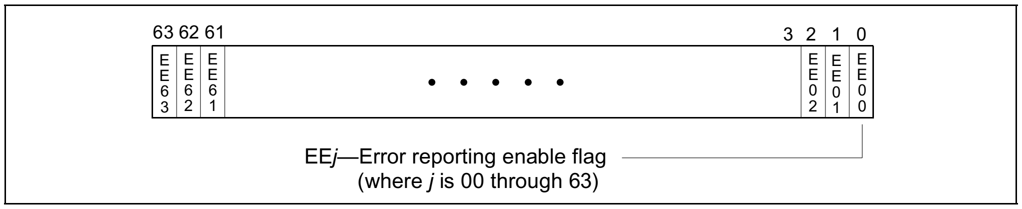
63 62 61
3 2 1 0
EE63
EE62
EE61
• • • • •
EE02
EE01
EE00
EE*j*—Error reporting enable flag
(where *j* is 00 through 63)

Figure 14-5. IA32_MC*i*_CTL Register

Note:

For P6 family processors only: the operating system or executive software must not modify the contents of the MC0_CTL MSR. This MSR is internally aliased to the EBL_CR_POWERON MSR and controls platform-specific error handling features. System specific firmware (the BIOS) is responsible for the appropriate initialization of the MC0_CTL MSR. P6 family processors only allow the writing of all 1s or all 0s to the MC*i*_CTL MSR.

20. Section 7.7.5 Updated

IA-32 Intel Architecture Software Developer's Manual, Volume 3; Section 7.7.5: text has been updated to correct inconsistencies in naming processor IDs and calculation of the ID bits. The affected areas have been marked by change bars.

7.7.5. Identifying Logical Processors in an MP System

For any IA-32 processor, the system hardware establishes an initial APIC ID for the processor during power-up or RESET (see Section 7.6.4.). For an IA-32 processor supporting Hyper-Threading Technology, system hardware assigns a unique APIC ID to each logical processor on the system bus.

The APIC ID for a logical processor is made up of three fields: logical processor number, physical package ID, and cluster ID. Figure 7-5 shows the layout of these fields. Bit 0 is used to identify the two logical processor within the package, bits 1-2 are the 2-bit package ID, and bits 3-4 are the 2-bit cluster ID.

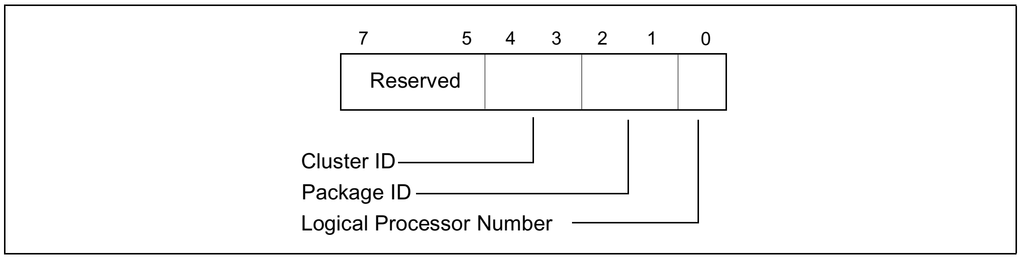

Figure 7-5. Interpretation of the APIC ID

Table 7-1 shows the APIC IDs that are generated for logical processors in a system with four MP-type Intel Xeon processors (a total of 8 logical processors). Of the two logical processors within a Intel Xeon processor MP, logical processor 0 is designated the "primary logical processor" and logical processor 1 is designated the "secondary logical processor."

Initial APIC ID of a Logical Processor	Package ID	Logical Processor Number
0H	0H	0H
1H	0H	1H
2H	1H	0H
3H	1H	1H
4H	2H	0H
5H	2H	1H
6H	3H	0H
7H	3H	1H

Software can determine the APIC IDs of the logical processor in the system in either of two ways described in Section 7.5.5. Note that only the APIC IDs of the primary logical processors in each physical package are included in the MP table created by the BIOS. All the logical processors in the system are included in the ACPI table, with the primary logical processors at the top of the table followed by the secondary logical processors.

In future IA-32 processors supporting Hyper-Threading Technology that implement more than two logical processors per physical processor, the logical processor bit shown in Figure 7-5 will be expanded to a 2- or 3-bit field to allow each of the logical processors to be identified. The package ID and cluster ID fields will be shifted left. Also, the package ID may be expanded to more than 2 bits, requiring the cluster ID field to be shifted left.

Operating system and application software can determine the layout of an APIC ID for a particular processor by interpreting the number of logical processors field and the local APIC physical ID field. These are returned to the EBX register when CPUID is executed with 1 in EAX.

As with IA-32 processors without HT Technology, software can assign a different APIC ID to a logical processor by writing a value into the local APIC ID register. However, the CPUID instruction will always report the processor's initial APIC ID (the value assigned during power-up or RESET).

Figure 7-5 depicts the layout of cluster ID, package ID and logical processor number bit fields of an APIC ID for current implementations of HT Technology (two logical processors per package). The number of bits that are reserved for the logical processor number can be computed by:

(1+((int)log$_2$(max(Logical_Per_Package-1,1))))

The content of an APIC ID (excluding cluster ID) for a logical processor in a package with a finite number of logical processors per package is given by:

(Package_ID<<(1+((int)log$_2$(max(Logical_Per_Package-1,1))))) | Logical_Processor_Number;

Use this formula to determine the association between logical processors and their physical packages for future implementations of HT Technology. The pseudo-code below (Examples 7-1 and 7-2) shows an algorithm to determine the relationship between logical and physical processors. This algorithm supports any number of logical processors per package. The algorithm is run on each logical processor in the system using an operating system specific affinity to accomplish binding. After running the algorithm, logical processors that have the same Processor ID exist within the same physical package. All processors present in the system must support the same number of logical processors per physical processor.

The algorithm for detecting support for HT Technology and identifying the relationships between a logical processor to the corresponding package ID consists of five steps:

1) Detect support for HT Technology in the processor.
2) Identify the number of logical processors available in a physical processor package.
3) Extract the initial APIC ID for this processor.
4) Compute a mask value and bit-shift value.
5) Compute a logical processor number and physical processor package ID.

See Example 7-2.

Example 7-2. Algorithm for Detecting Support for HT Technology and Identifying the Relationships Between a Logical Processor to the Corresponding Package ID

1. Detect support for Hyper-Threading Technology in a processor.

```
// Returns non-zero if Hyper-Threading Technology is supported on
// the processors and zero if not. This does not mean that
// Hyper-Threading Technology is enabled.

unsigned int HTSupported(void)
{
    try { // verify cpuid instruction is supported
        execute cpuid with eax = 0 to get vendor string
        execute cpuid with eax = 1 to get feature flag and signature
    }
    except (EXCEPTION_EXECUTE_HANDLER) {
    return 0 ; // CPUID is not supported and so Hyper-Threading
        // Technology is not supported
    }

    // Check to see if this a Genuine Intel Processor
    // and a member of the Pentium 4 processor family
    // supporting Hyper-Threading Technology

if (vendor string EQ GenuineIntel)
        if (family signature EQ Pentium4Family)
            return (feature_flag_edx & HTT_BIT);
    return 0;
}
```

2. Identify the number of logical processors per physical processor package.

```
#define NUM_LOGICAL_BITS 0x00FF0000 // EBX[23:16] indicate number of
                                    // logical processor per package

// Returns the number of logical processors per physical processor
package.

unsigned char LogicalProcessorsPerPackage(void)
{
    if (!HTSupported()) return (unsigned char) - 1;
    execute cpuid with eax = 1
    store returned value of ebx
    return (unsigned char) ((reg_ebx & NUM_LOGICAL_BITS) >> 16);
}
```

3. Extract the initial APIC ID of a processor.

```
#define INITIAL_APIC_ID_BITS 0xFF000000 // EBX[31:24] initial APIC ID

// Returns the 8-bit unique initial APIC ID for the processor that
// the code is running on. The default value returned is 0xFF if
// Hyper-Threading Technology is not supported.
```

```
unsigned char GetAPIC_ID (void)
{
    unsigned int reg_ebx = 0;
    if (!HTSupported()) return (unsigned char) -1;
    execute cpuid with eax = 1
    store returned value of ebx
    return (unsigned char) ((reg_ebx & INITIAL_APEIC_ID_BITS) >> 24;
}
```

4. **Compute the mask value and the bit-shift value, the logical processor number and physical processor package ID.**

```
    unsigned char i = 1;
    unsigned char PHY_ID_MASK = 0xFF;
    unsigned char PHY_ID_SHIFT = 0;
    unsigned char APIC_ID;
    unsigned char LOG_ID, PHY_ID;

    Logical_Per_Package = LogicalProcessorsPerPackage();
    While (i < Logical_Per_Package){
       i *= 2;
       PHY_ID_MASK <<= 1;
       PHY_ID_SHIFT++;
    }
    // Assume this thread is running on the logical processor from
    // which the logical processor number and its physical processor
    // package ID is extracted. If not, use the OS-specific affinity
    // service to bind this thread to the target logical processor.

    APIC_ID = GetAPIC_ID();
    LOT_ID = APIC_ID & ~PHY_ID_MASK;
    PHY_ID = APIC_ID >> PHY_ID_SHIFT;
```

5. **Compute the logical processor number and physical processor package ID.**

```
// The OS may limit the processor on which this process may run.

    hCurrentProcessHandle = GetCurrentProcess();
    GetProcessAffinityMask(hCurrentProcessHandle,
       &dwProcessAffinity, &dwSystemAffinity);

    // If the available process affinity mask does not equal the
    // available system affinity mask, determining if
    // Hyper-Threading Technology is enabled may not be possible.

    if (dwProcessAffinity != dwSystemAffinity)
       printf (“This process can not utilize all processors. \n”),

    dwAffinityMask = 1;
    while (dwAffinityMask != 0 &&
          dwAffinityMask <= dwProcessAffinity) {
       // Check to make sure we can utilize this processor first.
       if (dwAffinityMask & dwProcessAffinity){
          if (SetProcessAffinityMask(hCurrentProcessHandle,
                    dwAffinityMask)) {

            Sleep(0);     // May not be running on the logical processor
                          // on the affinity just set. Sleep gives the
                          // OS a chance to switch to the desired
                          // logical processor.

            // Retrieve APIC_ID for this logical processor;
            // Extract logical processor number and physical processor
            // package ID.

            }
        }
    }
```

21. CALL and JMP Summary Tables Updated

Intel® Extended Memory 64 Technology Software Developer's Guide, Volume 1; in the CALL and JMP sections: summary tables have been updated. The updated tables are reprinted below.

CALL—Call Procedure

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
E8 *cw*	CALL *rel16*	N.S.	Valid	Call near, relative, displacement relative to next instruction.
E8 *cd*	CALL *rel32*	Valid	Valid	Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode.
FF /2	CALL *r/m16*	N.E.	Valid	Call near, absolute indirect, address given in *r/m16.*
FF /2	CALL *r/m32*	N.E.	Valid	Call near, absolute indirect, address given in *r/m32.* 32-bit displacement sign extended to 64-bits in 64-bit mode.
FF /2	CALL *r/m64*	Valid	N.E.	Call near, absolute indirect, address given in r/m64.
9A *cd*	CALL *ptr16:16*	Invalid	Valid	Call far, absolute, address given in operand.
9A *cp*	CALL *ptr16:32*	Invalid	Valid	Call far, absolute, address given in operand.

FF /3	CALL *m16:16*	Valid	Valid	Call far, absolute indirect address given in *m16:16.* In 32-bit mode: if selector points to a gate, then RIP = 32-bit zero extended displacement taken from gate; else RIP = zero extended 16-bit offset from far pointer referenced in the instruction.
FF /3	CALL *m16:32*	Valid	Valid	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = zero extended 32-bit offset from far pointer referenced in the instruction.
REX.W + FF /3	CALL *m16:64*	Valid	N.E.	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = 64-bit offset from far pointer referenced in the instruction.

JMP—Jump

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
EB *cb*	JMP *rel8*	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits
E9 *cw*	JMP *rel16*	N.S.	Valid	Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.
E9 *cd*	JMP *rel32*	Valid	Valid	Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits
FF /4	JMP *r/m16*	N.S.	Valid	Jump near, absolute indirect, address = sign-extended *r/m16.* Not supported in 64-bit mode.
FF /4	JMP *r/m32*	N.S.	Valid	Jump near, absolute indirect, address = sign-extended *r/m32.* Not supported in 64-bit mode.
FF /4	JMP *r/m64*	Valid	N.E.	Jump near, absolute indirect, RIP = 64-Bit offset from register or memory
EA *cd*	JMP *ptr16:16*	Inv.	Valid	Jump far, absolute, address given in operand
EA *cp*	JMP *ptr16:32*	Inv.	Valid	Jump far, absolute, address given in operand
FF /5	JMP *m16:16*	Valid	Valid	Jump far, absolute indirect, address given in *m16:16*
FF /5	JMP *m16:32*	Valid	Valid	Jump far, absolute indirect, address given in *m16:32.*
REX.W + FF /5	JMP *m16:64*	Valid	N.E.	Jump far, absolute indirect, address given in *m16:64.*

22. Memory Options Table Updated

Table 10-7 in *IA-32 Intel Architecture Software Developer's Manual, Volume 3* has been updated. Additional options are being documented. The affected table cells are marked by change bars.

Table 10-7. Effective Page-Level Memory Types for Pentium III, Pentium 4, and Intel Xeon Processors

MTRR Memory Type	PAT Entry Value	Effective Memory Type
UC	UC	UC[1]
	UC-	UC[1]
	WC	WC
	WT	UC[1]
	WB	UC[1]
	WP	UC[1]
WC	UC	UC[2]
	UC-	WC
	WC	WC
	WT	UC[2,3]
	WB	WC
	WP	UC[2,3]
WT	UC	UC[2]
	UC-	UC[2]
	WC	WC
	WT	WT
	WB	WT
	WP	WP[3]

Table 10-7. Effective Page-Level Memory Types for Pentium III, Pentium 4, and Intel Xeon Processors (Continued)

MTRR Memory Type	PAT Entry Value	Effective Memory Type
WB	UC	UC[2]
	UC-	UC[2]
	WC	WC
	WT	WT
	WB	WB
	WP	WP
WP	UC	UC[2]
	UC-	WC[3]
	WC	WC
	WT	WT[3]
	WB	WP
	WP	WP

NOTES:

1. The UC attribute comes from the MTRRs and the processors are not required to snoop their caches since the data could never have been cached. This attribute is preferred for performance reasons.
2. The UC attribute came from the page-table or page-directory entry and processors are required to check their caches because the data may be cached due to page aliasing, which is not recommended.
3. These combinations were specified as “undefined” in previous editions of the *IA-32 Intel Architecture Software Developer’s Manual*. However, all processors that support both the PAT and the MTRRs determine the effective page-level memory types for these combinations as given.

23. LAHF/SAHF Section Updated

Intel® Extended Memory 64 Technology Software Developer’s Guide, Volumes 1 & 2; in the LAHF and SAHF sections: text in the list has been edited to remove #UD from the list of 64-bit mode exceptions. No exceptions are listed for these instructions in this mode.

24. Chapter 3 Corrected

IA-32 Intel Architecture Software Developer’s Manual, Volume 2A; Chapter 3, CPUID section: information about the CMPXCHG16B feature flag has been added to the extended feature information returned when EAX =1. Added text has been marked by change bars.

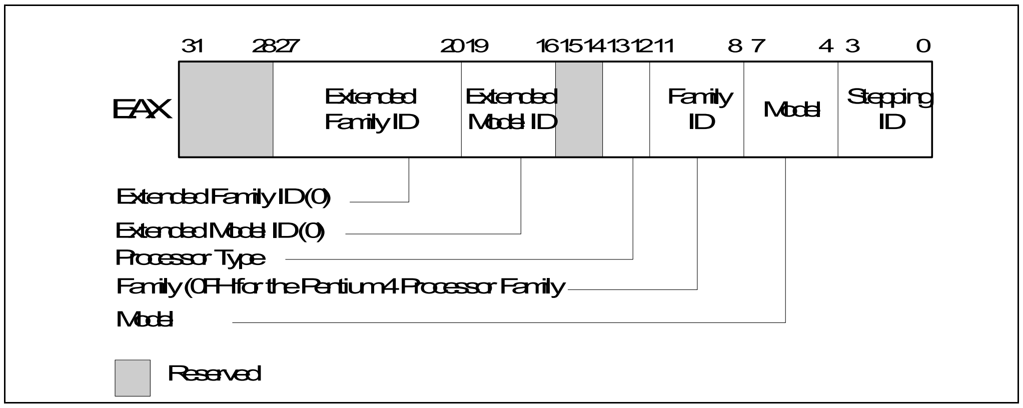

Figure 3-6. Extended Feature Information Returned in the ECX Register

Table 3-15. More on Extended Feature Information Returned in the ECX Register

Bit #	Mnemonic	Description
0	SSE3	**Streaming SIMD Extensions 3 (SSE3)**. A value of 1 indicates the processor supports this technology.
1-2	Reserved	Reserved
3	MONITOR	**MONITOR/MWAIT**. A value of 1 indicates the processor supports this feature.
4	DS-CPL	**CPL Qualified Debug Store**. A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5-6	Reserved	Reserved
7	EST	**Enhanced Intel SpeedStep® technology**. A value of 1 indicates that the processor supports this technology.
8	TM2	**Thermal Monitor 2**. A value of 1 indicates whether the processor supports this technology.
9	Reserved	Reserved
10	CNXT-ID	**L1 Context ID**. A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.
4-12	Reserved	Reserved
13	CMPXCHG16B	**CMPXCHG16B Available**. A value of 1 indicates that the feature is available. See the *CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes* section in this chapter for a description.
14-31	Reserved	Reserved

25. Appendix E Corrected

IA-32 Intel Architecture Software Developer's Manual, Volume 3; in Section E-2: 07H was incorrectly used instead of 0FH for a processor family designation. In addition (near the end of the table), 254 was used instead of 255 for error count. Areas of correction are marked by change bars.

E.2. Incremental Decoding Information: Processor Family 0FH Machine Error Codes For Machine Check

Table E-2 provides information for interpreting additional family 0FH model-specific fields for external bus errors. These errors are reported in the IA32_MCi_STATUS MSRs. They are reported (architecturally) as compound errors with a general form of *0000 1PPT RRRR IILL* in the MCA error code field. See Chapter 14 for information on the interpretation of compound error codes.

Table E-2. Incremental Decoding Information: Processor Family 0FH Machine Error Codes For Machine Check

Type	Bit No.	Bit Function	Bit Description
MCA error codes[a]	0-15		
Model-specific error codes	16	FSB address parity	Address parity error detected: 1 = Address parity error detected 0 = No address parity error
	17	Response hard fail	Hardware failure detected on response
	18	Response parity	Parity error detected on response
	19	PIC and FSB data parity	Data Parity detected on either PIC or FSB access
	20	Processor Signature = 00000F04H: Invalid PIC request All other processors: Reserved	Processor Signature = 00000F04H. Indicates error due to an invalid PIC request (access was made to PIC space with WB memory): 1 = Invalid PIC request error 0 = No Invalid PIC request error Reserved
	21	Pad state machine	The state machine that tracks P and N data-strobe relative timing has become unsynchronized or a glitch has been detected.
	22	Pad strobe glitch	Data strobe glitch
	23	Pad address glitch	Address strobe glitch
Other Information	24-56	Reserved	Reserved
Status register validity indicators[1]	57-63		

a. These fields are architecturally defined. Refer to Chapter 14, *Machine-Check Architecture* for more information.

Table E-3 provides information on interpreting additional family 0FH, model specific fields for memory hierarchy errors. These errors are reported in one of the IA32_MCi_STATUS MSRs. These errors are reported, architecturally, as compound errors with a general form of *0000 0001 RRRR TTLL* in the MCA error code field. See Chapter 14 for how to interpret the compound error code.

Table E-3. Decoding Family 0FH Machine Check Codes for Memory Hierarchy Errors

Type	Bit No.	Bit Function	Bit Description
MCA error codes[a]	0-15		
Model specific error codes	16-17	Tag Error Code	Contains the tag error code for this machine check error: 00 = No error detected 01 = Parity error on tag miss with a clean line 10 = Parity error/multiple tag match on tag hit 11 = Parity error/multiple tag match on tag miss
	18-19	Data Error Code	Contains the data error code for this machine check error: 00 = No error detected 01 = Single bit error 10 = Double bit error on a clean line 11 = Double bit error on a modified line
	20	L3 Error	This bit is set if the machine check error originated in the L3 (it can be ignored for invalid PIC request errors): 1 = L3 error 0 = L2 error
	21	Invalid PIC Request	Indicates error due to invalid PIC request (access was made to PIC space with WB memory): 1 = Invalid PIC request error 0 = No invalid PIC request error
	22-31	Reserved	Reserved
Other Information	32-39	8-bit Error Count	Holds a count of the number of errors since reset. The counter begins at 0 for the first error and saturates at **a count of 255.**
	40-56	Reserved	Reserved
Status register validity indicators[1]	57-63		

a. These fields are architecturally defined. Refer to Chapter 14, *Machine-Check Architecture* for more information.

26. Section 7.1 Corrected

IA-32 Intel Architecture Software Developer's Manual, Volume 3; Section 7.1: a note has been added to address the subject of "lock starvation." Added text has been marked by a change bar.

7.1. Locked Atomic Operations

The 32-bit IA-32 processors support locked atomic operations on locations in system memory. These operations are typically used to manage shared data structures (such as semaphores, segment descriptors, system segments, or page tables) in which two or more processors may try simultaneously to modify the same field or flag. The processor uses three interdependent mechanisms for carrying out locked atomic operations:

- guaranteed atomic operations
- bus locking, using the LOCK# signal and the LOCK instruction prefix
- cache coherency protocols that insure that atomic operations can be carried out on cached data structures (cache lock); this mechanism is present in the Pentium 4, Intel Xeon, and P6 family processors

These mechanisms are interdependent in the following ways. Certain basic memory transactions (such as reading or writing a byte in system memory) are always guaranteed to be handled atomically. That is, once started, the processor guarantees that the operation will be completed before another processor or bus agent is allowed access to the memory location. The processor also supports bus locking for performing selected memory operations (such as a read-modify-write operation in a shared area of memory) that typically need to be handled atomically, but are not automatically handled this way. Because frequently used memory locations are often cached in a processor's L1 or L2 caches, atomic operations can often be carried out inside a processor's caches without asserting the bus lock. Here the processor's cache coherency protocols insure that other processors that are caching the same memory locations are managed properly while atomic operations are performed on cached memory locations.

Note:

Where there are contested lock accesses, software may need to implement algorithms that ensure fair access to resources in order to prevent lock starvation. The hardware provides no resource that guarantees fairness to participating agents. It is the responsibility of software to manage the fairness of semaphores and exclusive locking functions.

The mechanisms for handling locked atomic operations have evolved as the complexity of IA-32 processors has evolved. As such, more recent IA-32 processors (such as the Pentium 4, Intel Xeon, and P6 family processors) provide a more refined locking mechanism than earlier IA-32 processors. These are described in the following sections.

27. Table 10-5 Corrected

IA-32 Intel Architecture Software Developer's Manual, Volume 3; Table 10-5: an incorrect entry has been deleted. A change bar marks the correction (not all cells in the table have been included).

CD	NW	Caching and Read/Write Policy	L1	L2/L3[1]
... ...				
1	0	No-fill Cache Mode. Memory coherency is maintained.		
		- (Pentium 4 and Intel Xeon processors.) State of processor after a power up or reset.	Yes	Yes
		- Read hits access the cache; read misses do not cause replacement (see Pentium 4 and Intel Xeon processors reference below).	Yes	Yes
		- Write hits update the cache.	Yes	Yes
		- Only writes to shared lines and write misses update system memory.	Yes	Yes
		- Write misses access memory.	Yes	Yes
		- Write hits can change shared lines to exclusive under control of the MTRRs and with associated read invalidation cycle.	Yes	Yes
		- (Pentium processor only.) Write hits can change shared lines to exclusive under control of the WB/WT#.	Yes	
		- (Pentium 4, Intel Xeon, and P6 family processors only.) Strict memory ordering is not enforced unless the MTRRs are disabled and/or all memory is referenced as uncached (see Section 7.2.4., "Strengthening or Weakening the Memory Ordering Model").	Yes	Yes
		- Invalidation is allowed.	Yes	Yes
		- External snoop traffic is supported.	Yes	Yes
... ...				

28. Section 10.3 Corrected

A note will be added in section 10.3, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*; Section 10.3: a note has been added that clarifies UC memory behavior for FP and SSE/SSE2 operations. Added text is marked by a change bar.

10.3. Methods of Caching Available

The processor allows any area of system memory to be cached in the L1, L2, and L3 caches. In individual pages or regions of system memory, it allows the type of caching (also called **memory type**) to be specified (see Section 10.5). Memory types currently defined for the IA-32 architecture are as follows (see Table 10-2):

- Strong Uncacheable (UC)—System memory locations are not cached. All reads and writes appear on the system bus and are executed in program order without reordering. No speculative memory accesses, page-table walks, or prefetches of speculated branch targets are made. This type of cache-control is useful for memory-mapped I/O devices. When used with normal RAM, it greatly reduces processor performance.

Note:

The behavior of FP and SSE/SSE2 operations on operands in UC memory is implementation dependent. In some implementations, accesses to UC memory may occur more than once. To ensure predictable behavior, use loads and stores of general purpose registers to access UC memory that may have read or write side effects.

Additional bullets follow....