IA-32 Intel® Architecture Software Developer's Manual

Documentation Changes

September 2005

Notice: The IA-32 Intel® Architecture may contain design defects or errors known as errata that may cause the product to deviate from published specifications. Current characterized errata are documented in the specification updates.

Document Number: 252046-014

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
-012	• Removed Documentation Changes 1-28. • Added Documentation Changes 1-16.	March 2005
-013	• Updated title. • There are no Documentation Changes for this revision of the document.	July 2005
-014	• Added Documentation Changes 1-21.	September 2005

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

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	Table addressing CPUID return values has been updated
2	Changes made to clarify the impact on TLBs when code modifies CR4.PGE
3	More information provided on the handling override prefixes in 64-bit mode
4	IA32_MISC_ENABLE information updated for clarity
5	Opcode map updated
6	Count operand usage issue corrected
7	Note on interrupt delivery added
8	Error corrected in Table 9-1, Volume 3
9	Updated CPUID operation description
10	Information on IA32_CLOCK_MODULATION MSR updated
11	MOVUPS/MOVUPD inconsistencies corrected
12	Information on IRET treatment of EFLAGS.NT updated
13	Incorrect diagram of page directory entry corrected
14	MOV to/from control registers updated
15	LGDT/LIDT exceptions updated
16	Underflow description corrected
17	IN/OUT virtual-8086 mode exceptions updated
18	SYSENTER and SYSEXIT sections updated
19	LTR section updated
20	Table updated
21	Corrections to Jcc summary table

Documentation Changes

1. Table addressing CPUID return values has been updated

In Table 3-12, “CPUID—CPU Identification” section, *IA-32 Intel Architecture Software Developer’s Manual, Volume 2A*; a number of issues have been addressed:

- Associativity field values have been corrected.
- The relationship between CPUID.EAX = 04H, ECX, and specific return values has been clarified using an extended note.
- Formatting issues have been addressed.

Table 3-12 has been reproduced below. See the change bars for the impacted area.

Table 3-12. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
	Basic CPUID Information	
0H	EAX EBX ECX EDX	Maximum Input Value for Basic CPUID Information (see Table 3-13) “Genu” “ntel” “ineI”
01H	EAX	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-5)
	EBX	Bits 7-0: Brand Index Bits 15-8: CLFLUSH line size (Value ∗ 8 = cache line size in bytes) Bits 23-16: Maximum number of logical processors in this physical package. Bits 31-24: Initial APIC ID
	ECX EDX	Extended Feature Information (see Figure 3-6 and Table 3-15) Feature Information (see Figure 3-7 and Table 3-16)
02H	EAX EBX ECX EDX	Cache and TLB Information (see Table 3-17) Cache and TLB Information Cache and TLB Information Cache and TLB Information
03H	EAX EBX	Reserved. Reserved.
	ECX	Bits 00-31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
	EDX	Bits 32-63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) **NOTE:** Processor serial number (PSN) is not supported 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. See AP-485, *Intel Processor Identification and the CPUID Instruction* (Order Number 241618) for more information on PSN.

Table 3-12. Information Returned by CPUID Instruction (Continued)

Initial EAX Value	Information Provided about the Processor	
	CPUID leaves > 3 < 80000000 are visible only when IA32_MISC_ENABLES.BOOT_NT4[bit 22] = 0 (default).	
	Deterministic Cache Parameters Leaf	
04H		NOTE: 04H output also depends on the inital value in ECX. See also: "INPUT EAX = 4: Returns Deterministic Cache Parameters for Each Level" on page 3-172.
	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: Maximum number of threads sharing this cache in a physical package (see note)** Bits 31-26: Maximum number of processor cores in this physical package**
	EBX	Bits 11-00: L = System Coherency Line Size** Bits 21-12: P = Physical Line partitions** Bits 31-22: W = Ways of associativity**
	ECX	Bits 31-00: S = Number of Sets**
	EDX	Reserved = 0
		MORE NOTES: * 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.
	MONITOR/MWAIT Leaf	
5H	EAX	Bits 15-00: Smallest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0
	EBX	Bits 15-00: Largest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0
	ECX EDX	Reserved = 0 Reserved = 0

Table 3-12. Information Returned by CPUID Instruction (Continued)

Initial EAX Value	Information Provided about the Processor	
	Extended Function CPUID Information	
80000000H	EAX	Maximum Input Value for Extended Function CPUID Information (see Table 3-13).
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
80000001H	EAX	Extended Processor Signature and Extended Feature Bits.
	EBX	Reserved
	ECX	Bit 0: LAHF/SAHF available in 64-bit mode Bits 31-1 Reserved
	EDX	Bits 10-0: Reserved Bit 11: SYSCALL/SYSRET available (when in 64-bit mode) Bits 19-12: Reserved = 0 Bit 20: Execute Disable Bit available Bits 28-21: Reserved = 0 Bit 29: Intel EM64T available = 1 Bits 31-30: Reserved = 0
80000002H	EAX	Processor Brand String
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
80000003H	EAX	Processor Brand String Continued
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
80000004H	EAX	Processor Brand String Continued
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued

Table 3-12. Information Returned by CPUID Instruction (Continued)

Initial EAX Value		Information Provided about the Processor
80000005H	EAX EBX ECX EDX	Reserved = 0 Reserved = 0 Reserved = 0 Reserved = 0
80000006H	EAX EBX	Reserved = 0 Reserved = 0
	ECX	Bits 7-0: Cache Line size Bits 15-12: L2 Associativity field * Bits 31-16: Cache size in 1K units
	EDX	Reserved = 0
		NOTES: * L2 associativity field encodings: 00H - Disabled 01H - Direct mapped 02H - 2-way 04H - 4-way 06H - 8-way 08H - 16-way 0FH - Fully associative
80000007H	EAX EBX ECX EDX	Reserved = 0 Reserved = 0 Reserved = 0 Reserved = 0

2. Changes made to clarify the impact on TLBs when code modifies CR4.PGE

In Section 3.12, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*; updates that clarify the relationship between CR4.PGE and TLB have been made. The text has been reproduced below (with changes introduced in context). See the change bars for the impacted area.

... ...

(Introduced in the Pentium Pro processor.) The page global enable (PGE) flag in register CR4 and the global (G) flag of a page-directory or page-table entry (bit 8) can be used to prevent frequently used pages from being automatically invalidated in the TLBs on a task switch or a load of register CR3. (See Section 3.7.6, "Page-Directory and Page-Table Entries", for more information about the global flag.) When the processor loads a page-directory or page-table entry for a global page into a TLB, the entry will remain in the TLB indefinitely. The only ways to deterministically invalidate global page entries are as follows:

- Clear the PGE flag; this will invalidate the TLBs.
- Execute the INVLPG instruction to invalidate individual page-directory or page-table entries in the TLBs.

For additional information about invalidation of the TLBs, see Section 10.9, "Invalidating the Translation Lookaside Buffers (TLBs)".

.. ..

Sections 10.11.7.2 and 10.11.8, *IA-32 Intel Architecture Software Developer's Manual, Volume 3;* changes have also been made to to address the same issue. The text has been reproduced below (with changes introduced in context). See the change bars for the impacted area.

10.11.7.2 MemTypeSet() Function

The MemTypeSet() function in Example 10-6 sets a MTRR for the physical memory range specified by the parameters base and size to the type specified by type. The base address and size are multiples of 4 KBytes and the size is not 0.

Example 10-6. MemTypeSet Pseudocode

```
IF CPU_FEATURES.MTRR (* processor supports MTRRs *)
    THEN
        IF BASE and SIZE are not 4-KByte aligned or size is 0
            THEN return INVALID;
        FI;
        IF (BASE + SIZE) wrap 4-GByte address space
            THEN return INVALID;
        FI;
        IF TYPE is invalid for Pentium 4, Intel Xeon, and P6 family processors
            THEN return UNSUPPORTED;
        FI;
        IF TYPE is WC and not supported
            THEN return UNSUPPORTED;
        FI;
        IF IA32_MTRRCAP.FIX is set AND range can be mapped using a fixed-range MTRR
            THEN
                pre_mtrr_change();
                update affected MTRR;
                post_mtrr_change();
        FI;

    ELSE (* try to map using a variable MTRR pair *)
        IF IA32_MTRRCAP.VCNT = 0
            THEN return UNSUPPORTED;
        FI;
        IF conflicts with current variable ranges
            THEN return RANGE_OVERLAP;
        FI;
        IF no MTRRs available
            THEN return VAR_NOT_AVAILABLE;
        FI;
        IF BASE and SIZE do not meet the power of 2 requirements for variable MTRRs
            THEN return INVALID_VAR_REQUEST;
        FI;
        pre_mtrr_change();
        Update affected MTRRs;
        post_mtrr_change();
FI;

pre_mtrr_change()
    BEGIN
```

```
            disable interrupts;
            Save current value of CR4;
            disable and flush caches;
            flush TLBs;
            disable MTRRs;
            IF multiprocessing
                  THEN maintain consistency through IPIs;
            FI;
      END
post_mtrr_change()
      BEGIN
            flush caches and TLBs;
            enable MTRRs;
            enable caches;
            restore value of CR4;
            enable interrupts;
      END
```

The physical address to variable range mapping algorithm in the MemTypeSet function detects conflicts with current variable range registers by cycling through them and determining whether the physical address in question matches any of the current ranges. During this scan, the algorithm can detect whether any current variable ranges overlap and can be concatenated into a single range.

The pre_mtrr_change() function disables interrupts prior to changing the MTRRs, to avoid executing code with a partially valid MTRR setup. The algorithm disables caching by setting the CD flag and clearing the NW flag in control register CR0. The caches are invalidated using the WBINVD instruction. The algorithm flushes all TLB entries either by clearing the page-global enable (PGE) flag in control register CR4 (if PGE was already set) or by updating control register CR3 (if PGE was already clear). Finally, it disables MTRRs by clearing the E flag in the IA32_MTRR_DEF_TYPE MSR.

After the memory type is updated, the post_mtrr_change() function re-enables the MTRRs and again invalidates the caches and TLBs. This second invalidation is required because of the processor's aggressive prefetch of both instructions and data. The algorithm restores interrupts and re-enables caching by setting the CD flag.

An operating system can batch multiple MTRR updates so that only a single pair of cache invalidations occur.

10.11.8 MTRR Considerations in MP Systems

In MP (multiple-processor) systems, the operating systems must maintain MTRR consistency between all the processors in the system. The Pentium 4, Intel Xeon, and P6 family processors provide no hardware support to maintain this consistency. In general, all processors must have the same MTRR values.

This requirement implies that when the operating system initializes an MP system, it must load the MTRRs of the boot processor while the E flag in register MTRRdefType is 0. The operating system then directs other processors to load their MTRRs with the same memory map. After all the processors have loaded their MTRRs, the operating system signals them to enable their MTRRs. Barrier synchronization is used to prevent further memory accesses until all processors indicate that the MTRRs are enabled. This synchronization is likely to be a shoot-down style algorithm, with shared variables and interprocessor interrupts.

Any change to the value of the MTRRs in an MP system requires the operating system to repeat the loading and enabling process to maintain consistency, using the following procedure:

1. Broadcast to all processors to execute the following code sequence.
2. Disable interrupts.
3. Wait for all processors to reach this point.
4. Enter the no-fill cache mode. (Set the CD flag in control register CR0 to 1 and the NW flag to 0.)
5. Flush all caches using the WBINVD instructions. Note on a processor that supports self-snooping, CPUID feature flag bit 27, this step is unnecessary.
6. If the PGE flag is set in control register CR4, flush all TLBs by clearing that flag.
7. If the PGE flag is clear in control register CR4, flush all TLBs by executing a MOV from control register CR3 to another register and then a MOV from that register back to CR3.
8. Disable all range registers (by clearing the E flag in register MTRRdefType). If only variable ranges are being modified, software may clear the valid bits for the affected register pairs instead.
9. Update the MTRRs.
10. Enable all range registers (by setting the E flag in register MTRRdefType). If only variable-range registers were modified and their individual valid bits were cleared, then set the valid bits for the affected ranges instead.
11. Flush all caches and all TLBs a second time. (The TLB flush is required for Pentium 4, Intel Xeon, and P6 family processors. Executing the WBINVD instruction is not needed when using Pentium 4, Intel Xeon, and P6 family processors, but it may be needed in future systems.)
12. Enter the normal cache mode to re-enable caching. (Set the CD and NW flags in control register CR0 to 0.)
13. Set PGE flag in control register CR4, if cleared in Step 6 (above).
14. Wait for all processors to reach this point.
15. Enable interrupts.

3. More information provided on the handling override prefixes in 64-bit mode

Section 3.3.7.1, *IA-32 Intel Architecture Software Developer's Manual, Volume 1;* information on the handling of override prefixes has been added. The added text has been reproduced below (with changes reproduced in context). See the change bars for the impacted area.

3.3.7.1 Canonical Addressing

In 64-bit mode, an address is considered to be in canonical form if address bits 63 through to the most-significant implemented bit by the microarchitecture are set to either all ones or all zeros.

Intel EM64T defines a 64-bit linear address. Implementations can support less. The first implementation of IA-32 processors with Intel EM64T supports a 48-bit linear address. This means a canonical address must have bits 63 through 48 set to zeros or ones (depending on whether bit 47 is a zero or one).

Although implementations may not use all 64 bits of the linear address, they should check bits 63 through the most-significant implemented bit to see if the address is in canonical form. If a linear-memory reference is not in canonical form, the implementation should generate an exception. In most cases, a general-protection exception (#GP) is generated. However, in the case of explicit or implied stack references, a stack fault (#SS) is generated.

Instructions that have implied stack references, by default, use the SS segment register. These include PUSH/POP-related instructions and instructions using RSP/RBP as base registers. In these cases, the canonical fault is #SF.

If an instruction uses base registers RSP/RBP and uses a segment override prefix to specify a non-SS segment, a canonical fault generates a #GP (instead of an #SF). In 64-bit mode, only FS and GS segment-overrides are applicable in this situation. Other segment override prefixes (CS, DS, ES and SS) are ignored. Note that this also means that an SS segment-override applied to a “non-stack” register reference is ignored. Such a sequence still produces a #GP for a canonical fault (and not an #SF).

4. IA32_MISC_ENABLE information updated for clarity

In Table B-1, *IA-32 Intel Architecture Software Developer’s Manual*, *Volume 3;* corrections have been made to IA32_MISC_ENABLE data. The applicable table cells are reproduced below (with the changes marked in context). See the change bars for the impacted area.

Table B-1 MSRs in the Pentium 4 and Intel Xeon Processors

Register Address		**Register Name Fields and Flags**	**Model Avail-ability**	**Shared/ Unique**[1]	**Bit Description**
Hex	**Dec**				
1A0H	416	IA32_MISC_ENABLE	0, 1, 2, 3, 4	Shared	**Enable Miscellaneous Processor Features.** (R/W) Allows a variety of processor functions to be enabled and disabled.
		0			**Fast-Strings Enable.** When set, the fast-strings feature on the Pentium 4 processor is enabled (default); when clear, fast-strings are disabled.
		1			**Reserved.**
		2			**x87 FPU Fopcode Compatibility Mode Enable.** When set, fopcode compatibility mode is enabled; when clear (default), mode is disabled. See “Fopcode Compatibility Mode“ in Chapter 8 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*.
		3			**Thermal Monitor 1 Enable.** When set, clock modulation controlled by the processor’s internal thermal sensor is enabled; when clear (default), automatic clock modulation is disabled. See Section 13.16.2, “Thermal Monitor”.

Table B-1 MSRs in the Pentium 4 and Intel Xeon Processors

		4			**Split-Lock Disable.** This debug feature is specific to the Pentium 4 processor. When set, the bit causes an #AC exception to be issued instead of a split-lock cycle. Operating systems that set this bit must align system structures to avoid split-lock scenarios. When the bit is clear (default), normal split-locks are issued to the bus.
		5			**Reserved.**
		6			**Third-Level Cache Disable.** (R/W) When set, the third-level cache is disabled; when clear (default) the third-level cache is enabled. This flag is reserved for processors that do not have a third-level cache. Note that the bit controls only the third-level cache; and only if overall caching is enabled through the CD flag of control register CR0, the page-level cache controls, and/or the MTRRs. See Section 10.5.4, “Disabling and Enabling the L3 Cache”.
		7			**Performance Monitoring Available.** (R) When set, performance monitoring is enabled; when clear, performance monitoring is disabled.
		8			**Suppress Lock Enable.** When set, assertion of LOCK on the bus is suppressed during a Split Lock access. When clear (default), LOCK is not suppressed.
		9			**Prefetch Queue Disable.** When set, disables the prefetch queue. When clear (default), enables the prefetch queue.

Table B-1 MSRs in the Pentium 4 and Intel Xeon Processors

		10			**FERR# Interrupt Reporting Enable.** (R/W) When set, interrupt reporting through the FERR# pin is enabled; when clear, this interrupt reporting function is disabled. When this flag is set and the processor is in the stop-clock state (STPCLK# is asserted), asserting the FERR# pin signals to the processor that an interrupt (such as, INIT#, BINIT#, INTR, NMI, SMI#, or RESET#) is pending and that the processor should return to normal operation to handle the interrupt. This flag does not affect the normal operation of the FERR# pin (to indicate an unmasked floating-point error) when the STPCLK# pin is not asserted.
		11			**Branch Trace Storage Unavailable (BTS_UNAVILABLE).** (R) When set, the processor does not support branch trace storage (BTS); when clear, BTS is supported.
		12			**Precise Event Based Sampling Unavailable (PEBS_UNAVILABLE).** (R) When set, the processor does not support precise event-based sampling (PEBS); when clear, PEBS is supported.
		13	**3**		**TM2 Enable.** (R/W) When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0. When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermal managed state. **NOTE:** If the TM2 feature flag (ECX[8]) is not set to 1 after executing CPUID with EAX = 1, then this feature is not supported and BIOS must not alter the contents of this bit location. The processor is operating out of spec if both this bit and the TM1 bit are set to disabled states.
		17:14			**Reserved.**

Table B-1 MSRs in the Pentium 4 and Intel Xeon Processors

		18	**3**		**ENABLE MONITOR FSM.** (R/W) When set (default), the MONITOR and MWAIT instructions are enabled. When clear, these instructions are disabled and attempting to execute them results in an invalid opcode exception. **NOTE:** CPUID.1:EAX.MONITOR[bit 3] indicates the setting of the Enable Monitor FSM bit. If CPUID.1:ECX.SSE3[bit 0] is not set, then the operating system must not attempt to alter the setting of the Enable Monitor FSM bit. BIOS should leave this bit in the default state.
		19			**Adjacent Cache Line Prefetch Disable.** (R/W) When set to 1, the processor fetches the cache line of the 128-byte sector containing currently required data. When set to 0, the processor fetches both cache lines in the sector. Single processor platforms should not set this bit. Server platforms should set or clear this bit based on platform performance observed in validation and testing. BIOS may contain a setup option that controls the setting of this bit.
		21:20			**Reserved.**
		22	**3**		**Limit CPUID MAXVAL.** (R/W) When set to 1, CPUID with EAX = 0 returns a maximum value in EAX[7:0] of 3. When set to a 0 (default), CPUID with EAX = 0 returns the number corresponding to the maximum standard function supported. **NOTE:** Some older OS's cannot handle a MAXVAL greater than 3. BIOS should contain a setup question that allows the user to specify such an OS is installed. Before setting this bit, BIOS must execute the CPUID instruction with EAX = 0 and examine the maximum value returned in EAX[7:0]. If the maximum value is greater than 3, then this bit is supported. Otherwise, this bit is not supported and BIOS must not alter the contents of this bit location.
		23			**Reserved.**

Table B-1 MSRs in the Pentium 4 and Intel Xeon Processors

		24			**L1 Data Cache Context Mode.** (R/W) When set, the L1 data cache is placed in shared mode; when clear (default), the cache is placed in adaptive mode. This bit is only enabled for IA-32 processors that support Intel Hyper-Threading Technology. See Section 10.5.6, “L1 Data Cache Context Mode” for additional information about the use of this flag. When L1 is running in adaptive mode and CR3s are identical, data in L1 is shared across logical processors. Otherwise, L1 is not shared and cache use is competitive. **NOTE:** If the Context ID feature flag (ECX[10]) is set to 0 after executing CPUID with EAX = 1, the ability to switch modes is not supported. BIOS must not alter the contents of IA32_MISC_ENABLE[24].

Changes to support IA32_MISC_ENABLE have also been made to Section 10.5.6 through 10.5.6.2, *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*. The text has been reproduced below (with changes reproduced in context). See the change bars for the impacted area.

10.5.6 L1 Data Cache Context Mode

L1 data cache context mode is a feature of IA-32 processors that support Hyper-Threading Technology. When CPUID.1:ECX[bit 10] = 1, the processor supports setting L1 data cache context mode using the L1 data cache context mode flag (IA32_MISC_ENABLE[bit 24]). Selectable modes are adaptive mode (default) and shared mode.

The BIOS is responsible for configuring the L1 data cache context mode.

10.5.6.1 Adaptive Mode

Adaptive mode facilitates L1 data cache sharing between logical processors. When running in adaptive mode, the L1 data cache is shared across logical processors in the same core if:

- CR3 control registers for logical processors sharing the cache are identical.
- The same paging mode is used by logical processors sharing the cache.

In this situation, the entire L1 data cache is available to each logical processor (instead of being competitively shared).

If CR3 values are different for the logical processors sharing an L1 data cache or the logical processors use different paging modes, processors compete for cache resources. This reduces the effective size of the cache for each logical processor. Aliasing of the cache is not allowed (which prevents data thrashing).

10.5.6.2 Shared Mode

In shared mode, the L1 data cache is competitively shared between logical processors. This is true even if the logical processors use identical CR3 registers and paging modes.

In shared mode, linear addresses in the L1 data cache can be aliased, meaning that one linear address in the cache can point to different physical locations. The mechanism for resolving aliasing can lead to thrashing. For this reason, IA32_MISC_ENABLE[bit 24] = 0 is the preferred configuration for IA-32 processors that support Hyper-Threading Technology.

5. Opcode map updated

In Table A-4, *IA-32 Intel Architecture Software Developer's Manual, Volume 2B*; two table cells have been updated to reflect the operation of modern processors. The impacted rows and columns are reproduced below. See the shaded cells.

Table A-4. Two-Byte Opcode Map for Non-64-Bit Mode (First Byte is 0FH)

	8	9	A	B	C	D	E	F
0	INVD^{1D}	WBINVD^{1D}		UD2		**NOP Ev**		
1	PREFETCH^{1C} (Grp 16^{1A})							**NOP Ev**

6. Count operand usage issue corrected

See the "PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical," "PSRAW/PSRAD—Shift Packed Data Right Arithmetic," and "PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical" sections of the *IA-32 Intel Architecture Software Developer's Manual, Volume 2B.*

The PRM did not explicitly state how much of the count operand is used to determine whether the count exceeds the datatype. One could have interpreted that the full 128-bit operand (for the SSE-2 integer xmm/m128 versions) is parsed for the count. Actually, only 64 bits of xmm/m128 are checked.

This problem has been corrected for all of the above sections. Only the "PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical" section is reproduced below. See the change bars for the impacted area.

PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F F1 /*r*	PSLLW *mm, mm/m64*	Valid	Valid	Shift words in *mm* left *mm/m64* while shifting in 0s.
66 0F F1 /*r*	PSLLW *xmm1*, *xmm2/m128*	Valid	Valid	Shift words in *xmm1* left by *xmm2/m128* while shifting in 0s.
0F 71 /6 ib	PSLLW *xmm1*, *imm8*	Valid	Valid	Shift words in *mm* left by *imm8* while shifting in 0s.
66 0F 71 /6 ib	PSLLW *xmm1*, *imm8*	Valid	Valid	Shift words in *xmm1* left by *imm8* while shifting in 0s.
0F F2 /*r*	PSLLD *mm, mm/m64*	Valid	Valid	Shift doublewords in *mm* left by *mm/m64* while shifting in 0s.
66 0F F2 /*r*	PSLLD *xmm1*, *xmm2/m128*	Valid	Valid	Shift doublewords in *xmm1* left by *xmm2/m128* while shifting in 0s.
0F 72 /6 ib	PSLLD *mm, imm8*	Valid	Valid	Shift doublewords in *mm* left by *imm8* while shifting in 0s.
66 0F 72 /6 ib	PSLLD *xmm1*, *imm8*	Valid	Valid	Shift doublewords in *xmm1* left by *imm8* while shifting in 0s.
0F F3 /*r*	PSLLQ *mm, mm/m64*	Valid	Valid	Shift quadword in *mm* left by *mm/m64* while shifting in 0s.
66 0F F3 /*r*	PSLLQ *xmm1*, *xmm2/m128*	Valid	Valid	Shift quadwords in *xmm1* left by *xmm2/m128* while shifting in 0s.
0F 73 /6 ib	PSLLQ *mm, imm8*	Valid	Valid	Shift quadword in *mm* left by *imm8* while shifting in 0s.
66 0F 73 /6 ib	PSLLQ *xmm1*, *imm8*	Valid	Valid	Shift quadwords in *xmm1* left by *imm8* while shifting in 0s.

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 10-1 gives an example of shifting words in a 64-bit operand.

The destination operand may be an MMX technology register or an XMM register; the count operand can be either an MMX technology register or an 64-bit memory location, an XMM register or a 128-bit memory location, or an 8-bit immediate. Note that only the first 64-bits of a 128-bit count operand are checked to compute the count.

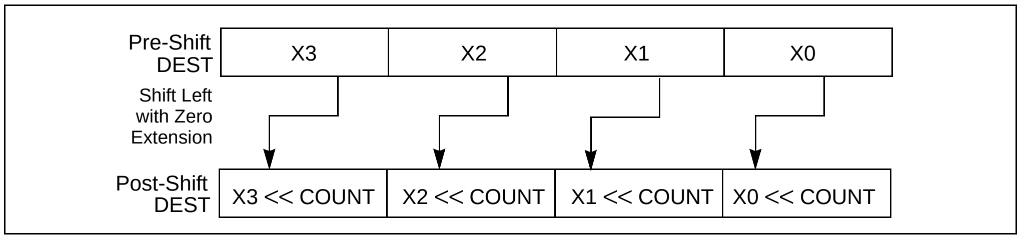

Figure 10-1. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand

The PSLLW instruction shifts each of the words in the destination operand to the left by the number of bits specified in the count operand; the PSLLD instruction shifts each of the doublewords in the destination operand; and the PSLLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode, using an REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
PSLLW instruction with 64-bit operand:
    IF (COUNT > 15)
    THEN
        DEST[64:0] ←0000000000000000H;
    ELSE
        DEST[15:0] ←ZeroExtend(DEST[15:0] << COUNT);
        (* Repeat shift operation for 2nd and 3rd words *)
        DEST[63:48] ←ZeroExtend(DEST[63:48] << COUNT);
    FI;
PSLLD instruction with 64-bit operand:
    IF (COUNT > 31)
    THEN
        DEST[64:0] ←0000000000000000H;
    ELSE
        DEST[31:0] ←ZeroExtend(DEST[31:0] << COUNT);
        DEST[63:32] ←ZeroExtend(DEST[63:32] << COUNT);
    FI;
PSLLQ instruction with 64-bit operand:
    IF (COUNT > 63)
    THEN
        DEST[64:0] ←0000000000000000H;
    ELSE
        DEST ←ZeroExtend(DEST << COUNT);
    FI;
PSLLW instruction with 128-bit operand:
    COUNT ←COUNT_SOURCE[63:0];
    IF (COUNT > 15)
    THEN
        DEST[128:0] ←00000000000000000000000000000000H;
    ELSE
        DEST[15:0]  ←ZeroExtend(DEST[15:0] << COUNT);
        (* Repeat shift operation for 2nd through 7th words *)
        DEST[127:112] ←ZeroExtend(DEST[127:112] << COUNT);
    FI;
PSLLD instruction with 128-bit operand:
    COUNT ←COUNT_SOURCE[63:0];
    IF (COUNT > 31)
    THEN
        DEST[128:0] ←00000000000000000000000000000000H;
    ELSE
        DEST[31:0]  ←ZeroExtend(DEST[31:0] << COUNT);
        (* Repeat shift operation for 2nd and 3rd doublewords *)
        DEST[127:96] ←ZeroExtend(DEST[127:96] << COUNT);
    FI;
```

```
PSLLQ instruction with 128-bit operand:
    COUNT ←COUNT_SOURCE[63:0];
    IF (COUNT > 63)
    THEN
         DEST[128:0] ←00000000000000000000000000000000H;
    ELSE
         DEST[63:0]  ←ZeroExtend(DEST[63:0] << COUNT);
         DEST[127:64] ←ZeroExtend(DEST[127:64] << COUNT);
    FI;
```

Intel C/C++Compiler Intrinsic Equivalents

PSLLW	__m64 _mm_slli_pi16 (__m64 m, int count)
PSLLW	__m64 _mm_sll_pi16(__m64 m, __m64 count)
PSLLW	__m128i _mm_slli_pi16(__m64 m, int count)
PSLLW	__m128i _mm_slli_pi16(__m128i m, __m128i count)
PSLLD	__m64 _mm_slli_pi32(__m64 m, int count)
PSLLD	__m64 _mm_sll_pi32(__m64 m, __m64 count)
PSLLD	__m128i _mm_slli_epi32(__m128i m, int count)
PSLLD	__m128i _mm_sll_epi32(__m128i m, __m128i count)
PSLLQ	__m64 _mm_slli_si64(__m64 m, int count)
PSLLQ	__m64 _mm_sll_si64(__m64 m, __m64 count)
PSLLQ	__m128i _mm_slli_si64(__m128i m, int count)
PSLLQ	__m128i _mm_sll_si64(__m128i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CR0.EM[bit 2] = 1.
	(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.
	(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.

#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	(128-bit operations only) If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If CR0.EM[bit 2] = 1.
	(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.
	(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

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 memory address is in a non-canonical form.
	(128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If CR0.EM[bit 2] = 1.
	(128-bit operations only) If CR4.OSFXSR[bit 9] = 0.
	(128-bit operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0.
#NM	If CR0.TS[bit 3] = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

7. Note on interrupt delivery added

In Section 5-10, *IA-32 Intel Architecture Software Developer's Manual*, *Volume 3*; a note has been added. The note clarifies the relationship between interrupts and the IDT configuration task. Text from the applicable section has been reproduced below (with the note reproduced in context). See the change bars for the impacted area.

10.5 INTERRUPT DESCRIPTOR TABLE (IDT)

The interrupt descriptor table (IDT) associates each exception or interrupt vector with a gate descriptor for the procedure or task used to service the associated exception or interrupt. Like the GDT and LDTs, the IDT is an array of 8-byte descriptors (in protected mode). Unlike the GDT, the first entry of the IDT may contain a descriptor. To form an index into the IDT, the processor scales the exception or interrupt vector by eight (the number of bytes in a gate descriptor). Because there are only 256 interrupt or exception vectors, the IDT need not contain more than 256 descriptors. It can contain fewer than 256 descriptors, because descriptors are required only for the interrupt and exception vectors that may occur. All empty descriptor slots in the IDT should have the present flag for the descriptor set to 0.

The base addresses of the IDT should be aligned on an 8-byte boundary to maximize performance of cache line fills. The limit value is expressed in bytes and is added to the base address to get the address of the last valid byte. A limit value of 0 results in exactly 1 valid byte. Because IDT entries are always eight bytes long, the limit should always be one less than an integral multiple of eight (that is, 8N – 1).

The IDT may reside anywhere in the linear address space. As shown in Figure 5-1, the processor locates the IDT using the IDTR register. This register holds both a 32-bit base address and 16-bit limit for the IDT.

The LIDT (load IDT register) and SIDT (store IDT register) instructions load and store the contents of the IDTR register, respectively. The LIDT instruction loads the IDTR register with the base address and limit held in a memory operand. This instruction can be executed only when the CPL is 0. It normally is used by the initialization code of an operating system when creating an IDT. An operating system also may use it to change from one IDT to another. The SIDT instruction copies the base and limit value stored in IDTR to memory. This instruction can be executed at any privilege level.

If a vector references a descriptor beyond the limit of the IDT, a general-protection exception (#GP) is generated.

NOTE

Because interrupts are delivered to the processor core only once, an incorrectly configured IDT could result in incomplete interrupt handling and/or the blocking of interrupt delivery. IA-32 architecture rules need to be followed for setting up IDTR base/limit/access fields and each field in the gate descriptors. This includes the implicit referencing of the destination code segment through the GDT or LDT, and the accessing of the stack.

8. Error corrected in Table 9-1, Volume 3

In Table 9-1, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*; a change has been made to correct an error. Cells from the table have been reproduced below (with the change reproduced in context). There is a change to one cell and a change to notes that follow the table. See the change bars for the impacted area.

Table 9-1. IA-32 Processor States Following Power-up, Reset, or INIT

Register	Pentium 4 and Intel Xeon Processor	P6 Family Processor	Pentium Processor
EFLAGS[1]	00000002H	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H	0000FFF0H
CR0	60000010H[2]	60000010H[2]	60000010H[2]
CR2, CR3, CR4	00000000H	00000000H	00000000H

CS	Selector = F000H
	Base = FFFF0000H
	Limit = FFFFH
	AR = Present, R/W, Accessed
		Selector = F000H
		Base = FFFF0000H
		Limit = FFFFH
		AR = Present, R/W, Accessed
			Selector = F000H
			Base = FFFF0000H
			Limit = FFFFH
			AR = Present, R/W, Accessed

SS, DS, ES, FS, GS	Selector = 0000H
	Base = 00000000H
	Limit = FFFFH
	AR = Present, R/W, Accessed
		Selector = 0000H
		Base = 00000000H
		Limit = FFFFH
		AR = Present, R/W, Accessed
			Selector = 0000H
			Base = 00000000H
			Limit = FFFFH
			AR = Present, R/W, Accessed

EDX	00000FxxH	**000n06xxH**[3]	000005xxH
EAX	0[4]	0[4]	0[4]
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H	00000000H
ST0 through ST7[5] x87 FPU Control Word[5] x87 FPU Status Word[5] x87 FPU Tag Word[5] x87 FPU Data Operand and CS Seg. Selectors[5] x87 FPU Data Operand and Inst. Pointers[5]	Pwr up or Reset: +0.0 FINIT/FNINIT: Unchanged Pwr up or Reset: 0040H FINIT/FNINIT: 037FH Pwr up or Reset: 0000H FINIT/FNINIT: 0000H Pwr up or Reset: 5555H FINIT/FNINIT: FFFFH Pwr up or Reset: 0000H FINIT/FNINIT: 0000H Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H	Pwr up or Reset: +0.0 FINIT/FNINIT: Unchanged Pwr up or Reset: 0040H FINIT/FNINIT: 037FH Pwr up or Reset: 0000H FINIT/FNINIT: 0000H Pwr up or Reset: 5555H FINIT/FNINIT: FFFFH Pwr up or Reset: 0000H FINIT/FNINIT: 0000H Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H	Pwr up or Reset: +0.0 FINIT/FNINIT: Unchanged Pwr up or Reset: 0040H FINIT/FNINIT: 037FH Pwr up or Reset: 0000H FINIT/FNINIT: 0000H Pwr up or Reset: 5555H FINIT/FNINIT: FFFFH Pwr up or Reset: 0000H FINIT/FNINIT: 0000H Pwr up or Reset: 00000000H FINIT/FNINIT: 00000000H
MM0 through MM7[5]	Pwr up or Reset: 0000000000000000H INIT or FINIT/FNINIT: Unchanged	Pentium II and Pentium III Processors Only— Pwr up or Reset: 0000000000000000H INIT or FINIT/FNINIT: Unchanged	Pentium with MMX Technology Only— Pwr up or Reset: 0000000000000000H INIT or FINIT/FNINIT: Unchanged
XMM0 through XMM7	Pwr up or Reset: 0000000000000000H INIT: Unchanged	Pentium III processor Only— Pwr up or Reset: 0000000000000000H INIT: Unchanged	NA
MXCSR	Pwr up or Reset: 1F80H INIT: Unchanged	Pentium III processor only- Pwr up or Reset: 1F80H INIT: Unchanged	NA

Table 9-1. IA-32 Processor States Following Power-up, Reset, or INIT (Continued)

Register	Pentium 4 and Intel Xeon Processor	P6 Family Processor	Pentium Processor

GDTR, IDTR	Base = 00000000H
	Limit = FFFFH
	AR = Present, R/W
		Base = 00000000H
		Limit = FFFFH
		AR = Present, R/W
			Base = 00000000H
			Limit = FFFFH
			AR = Present, R/W

LDTR, Task Register	Selector = 0000H
	Base = 00000000H
	Limit = FFFFH
	AR = Present, R/W
		Selector = 0000H
		Base = 00000000H
		Limit = FFFFH
		AR = Present, R/W
			Selector = 0000H
			Base = 00000000H
			Limit = FFFFH
			AR = Present, R/W

DR0, DR1, DR2, DR3	00000000H	00000000H	00000000H
DR6	FFFF0FF0H	FFFF0FF0H	FFFF0FF0H
DR7	00000400H	00000400H	00000400H
Time-Stamp Counter	Power up or Reset: 0H INIT: Unchanged	Power up or Reset: 0H INIT: Unchanged	Power up or Reset: 0H INIT: Unchanged
Perf. Counters and Event Select	Power up or Reset: 0H INIT: Unchanged	Power up or Reset: 0H INIT: Unchanged	Power up or Reset: 0H INIT: Unchanged
All Other MSRs	Pwr up or Reset: Undefined INIT: Unchanged	Pwr up or Reset: Undefined INIT: Unchanged	Pwr up or Reset: Undefined INIT: Unchanged
Data and Code Cache, TLBs	Invalid	Invalid	Invalid
Fixed MTRRs	Pwr up or Reset: Disabled INIT: Unchanged	Pwr up or Reset: Disabled INIT: Unchanged	Not Implemented
Variable MTRRs	Pwr up or Reset: Disabled INIT: Unchanged	Pwr up or Reset: Disabled INIT: Unchanged	Not Implemented
Machine-Check Architecture	Pwr up or Reset: Undefined INIT: Unchanged	Pwr up or Reset: Undefined INIT: Unchanged	Not Implemented
APIC	Pwr up or Reset: Enabled INIT: Unchanged	Pwr up or Reset: Enabled INIT: Unchanged	Pwr up or Reset: Enabled INIT: Unchanged

NOTES:

1. The 10 most-significant bits of the EFLAGS register are undefined following a reset. Software should not depend on the states of any of these bits.
2. The CD and NW flags are unchanged, bit 4 is set to 1, all other bits are cleared.
3. **Where "n" is the Extended Model Value for the respective processor.**
4. If Built-In Self-Test (BIST) is invoked on power up or reset, EAX is 0 only if all tests passed. (BIST cannot be invoked during an INIT.)
5. The state of the x87 FPU and MMX registers is not changed by the execution of an INIT.

9. Updated CPUID operation description

In the "Description" subsection, "CPUID—CPU Identification" section, Chapter 3, *IA-32 Intel Architecture Software Developer's Manual, Volume 2A*; input value ranges have been updated. The paragraphs have been reproduced below so that changes can be displayed in context. See the change bars for the impacted area.

... ...

Description

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction. This instruction operates the same in non-64-bit modes and 64-bit mode.

CPUID returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers. The instruction's output is dependent on the contents of the EAX register upon execution. For example, the following pseudocode loads EAX with 00H and causes CPUID to return a Maximum Return Value and the Vendor Identification String in the appropriate registers:

```
MOV EAX, 00H
CPUID
```

Table 3-12 shows information returned, depending on the initial value loaded into the EAX register. Table 3-13 shows the maximum CPUID input value recognized for each family of IA-32 processors on which CPUID is implemented.

Two types of information are returned: basic and extended function information. If a value is entered for CPUID.EAX is invalid for a particular processor, the data for the highest basic information leaf is returned. For example, using the Intel Pentium 4 Processor Extreme Edition, the following is true:

```
CPUID.EAX = 05H (* Returns MONITOR/MWAIT leaf. *)
CPUID.EAX = 09H (* INVALID: Returns the same information as CPUID.EAX = 05H. *)
CPUID.EAX = 80000008H (* Returns virtual/physical address size data. *)
CPUID.EAX = 8000000AH (* INVALID: Returns same information as CPUID.EAX = 05H. *)
```

CPUID can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.

... ...

10. Information on IA32_CLOCK_MODULATION MSR updated

Section 13.16.3, *IA-32 Intel Architecture Software Developer's Manual, Volume 3;* has been updated. Text from section has been reproduced below (with changes reproduced in context). See the change bars for the impacted area.

.--

13.16.3 Software Controlled Clock Modulation

Pentium 4, Intel Xeon and Pentium M processors also support software-controlled clock modulation. This provides a means for operating systems to implement a power management policy to reduce the power consumption of the processor. Here, the stop-clock duty cycle is controlled by software through the IA32_CLOCK_MODULATION MSR (see Figure 13-11).

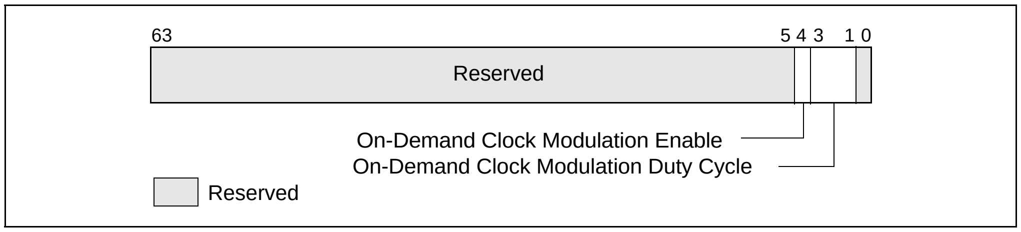
63
5 4 3
1 0
Reserved
On-Demand Clock Modulation Enable
On-Demand Clock Modulation Duty Cycle
Reserved

Figure 13-11. IA32_CLOCK_MODULATION MSR

The IA32_CLOCK_MODULATION MSR contains the following flag and field used to enable software-controlled clock modulation and to select the clock modulation duty cycle:

- **On-Demand Clock Modulation Enable, bit 4** — Enables on-demand software controlled clock modulation when set; disables software-controlled clock modulation when clear.
- **On-Demand Clock Modulation Duty Cycle, bits 1 through 3** — Selects the on-demand clock modulation duty cycle (see Table 13-8). This field is only active when the on-demand clock modulation enable flag is set.

Note that the on-demand clock modulation mechanism (like the thermal monitor) controls the processor's stop-clock circuitry internally to modulate the clock signal. The STPCLK# pin is not used in this mechanism.

Table 13-8. On-Demand Clock Modulation Duty Cycle Field Encoding

Duty Cycle Field Encoding	Duty Cycle
000B	Reserved
001B	12.5% (Default)
010B	25.0%
011B	37.5%
100B	50.0%
101B	63.5%
110B	75%
111B	87.5%

The on-demand clock modulation mechanism can be used to control processor power consumption. Power management software can write to the IA32_CLOCK_MODULATION MSR to enable clock modulation and to select a modulation duty cycle. If on-demand clock modulation and TM1 are both enabled and the thermal status of the processor is hot (bit 0 of the IA32_THERM_STATUS MSR is set), clock modulation at the duty cycle specified by TM1 takes precedence, regardless of the setting of the on-demand clock modulation duty cycle.

For Hyper-Threading Technology enabled processors, the IA32_CLOCK_MODULATION register is duplicated for each logical processor. In order for the On-demand clock modulation feature to work properly, the feature must be enabled on all the logical processors within a physical processor. If the programmed duty cycle is not identical for all the logical processors, the processor clock will modulate to the highest duty cycle programmed.

For the P6 family processors, on-demand clock modulation was implemented through the chipset, which controlled clock modulation through the processor's STPCLK# pin.

11. MOVUPS/MOVUPD inconsistencies corrected

In the "Interrupt 17—Alignment Check Exception (#AC)" section, Chapter 5, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*; text has been added to correct an issue with the MOVUPS and MOVUPD description. The updated text is reproduced below. See the change bars for the impacted area.

Interrupt 17—Alignment Check Exception (#AC)

Exception Class Fault.

Description

Indicates that the processor detected an unaligned memory operand when alignment checking was enabled. Alignment checks are only carried out in data (or stack) segments (not in code or system segments). An example of an alignment-check violation is a word stored at an odd byte address, or a doubleword stored at an address that is not an integer multiple of 4. Table 13-9 lists the alignment requirements various data types recognized by the processor.

Table 13-9. Alignment Requirements by Data Type

Data Type	Address Must Be Divisible By
Word	2
Doubleword	4
Single-precision floating-point (32-bits)	4
Double-precision floating-point (64-bits)	8
Double extended-precision floating-point (80-bits)	8
Quadword	8
Double quadword	16
Segment Selector	2
32-bit Far Pointer	2
48-bit Far Pointer	4
32-bit Pointer	4
GDTR, IDTR, LDTR, or Task Register Contents	4
FSTENV/FLDENV Save Area	4 or 2, depending on operand size
FSAVE/FRSTOR Save Area	4 or 2, depending on operand size
Bit String	2 or 4 depending on the operand-size attribute.

Note that the alignment check exception (#AC) is generated only for data types that must be aligned on word, doubleword, and quadword boundaries. A general-protection exception (#GP) is generated 128-bit data types that are not aligned on a 16-byte boundary.

To enable alignment checking, the following conditions must be true:

- AM flag in CR0 register is set.
- AC flag in the EFLAGS register is set.
- The CPL is 3 (protected mode or virtual-8086 mode).

Alignment-check exceptions (#AC) are generated only when operating at privilege level 3 (user mode). Memory references that default to privilege level 0, such as segment descriptor loads, do not generate alignment-check exceptions, even when caused by a memory reference made from privilege level 3.

Storing the contents of the GDTR, IDTR, LDTR, or task register in memory while at privilege level 3 can generate an alignment-check exception. Although application programs do not normally store these registers, the fault can be avoided by aligning the information stored on an even word-address.

The FXSAVE and FXRSTOR instructions save and restore a 512-byte data structure, the first byte of which must be aligned on a 16-byte boundary. If the alignment-check exception (#AC) is enabled when executing these instructions (and CPL is 3), a misaligned memory operand can cause either an alignment-check exception or a general-protection exception (#GP) depending on the IA-32 processor implementation (see “FXSAVE-Save x87 FPU, MMX, SSE, and SSE2 State” and “FXRSTOR-Restore x87 FPU, MMX, SSE, and SSE2 State” in Chapter 3 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 2*.

The MOVUPS and MOVUPD instructions perform 128-bit unaligned loads or stores. They do not generate general-protection exceptions (#GP) when operands are not aligned on a 16-byte boundary. If alignment checking is enabled, alignment-check exceptions (#AC) are generated when instructions are not aligned on an 8-byte boundary.

FSAVE and FRSTOR instructions generate unaligned references, which can cause alignment-check faults. These instructions are rarely needed by application programs.

Exception Error Code

Yes (always zero).

Saved Instruction Pointer

The saved contents of CS and EIP registers point to the instruction that generated the exception.

Program State Change

A program-state change does not accompany an alignment-check fault, because the instruction is not executed.

In addition, exception information in the “MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values” and “MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values” sections of Chapter 3, *IA-32 Intel Architecture Software Developer’s Manual, Volume 2A*, have been updated. Information on alignment check exceptions has been added. Exception data for both instructions is reprinted below, with changes marked in context. See the change bars for the impacted area.

MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values

Protected Mode Exceptions

#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.

#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE2[bit 26] = 0.

Real-Address Mode Exceptions

#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE2[bit 26] = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE2[bit 26] = 0.

MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values

Protected Mode Exceptions

#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.

#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE[bit 25] = 0.

Real-Address Mode Exceptions

#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE[bit 25] = 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE[bit 25] = 0.

12. Information on IRET treatment of EFLAGS.NT updated

In the “IRET/IRETD—Interrupt Return” section, Chapter 3, *IA-32 Intel Architecture Software Developer’s Manual, Volume 2A*; EFLAGS.NT data has been updated. The section is reprinted below. See the change bars for the impacted area.

IRET/IRETD—Interrupt Return

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
CF	IRET	Valid	Valid	Interrupt return (16-bit operand size).
CF	IRETD	Valid	Valid	Interrupt return (32-bit operand size).
REX.W + CF	IRETQ	Valid	N.E.	Interrupt return (64-bit operand size).

Description

Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions are also used to perform a return from a nested task. (A nested task is created when a CALL instruction is used to initiate a task switch or when an interrupt or exception causes a task switch to an interrupt or exception handler.) See the section titled "Task Linking" in Chapter 6 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*.

IRET and IRETD are mnemonics for the same opcode. The IRETD mnemonic (interrupt return double) is intended for use when returning from an interrupt when using the 32-bit operand size; however, most assemblers use the IRET mnemonic interchangeably for both operand sizes.

In Real-Address Mode, the IRET instruction preforms a far return to the interrupted program or procedure. During this operation, the processor pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

In Protected Mode, the action of the IRET instruction depends on the settings of the NT (nested task) and VM flags in the EFLAGS register and the VM flag in the EFLAGS image stored on the current stack. Depending on the setting of these flags, the processor performs the following types of interrupt returns:

- Return from virtual-8086 mode.
- Return to virtual-8086 mode.
- Intra-privilege level return.
- Inter-privilege level return.
- Return from nested task (task switch).

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack). As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

If the NT flag is set, the IRET instruction performs a task switch (return) from a nested task (a task called with a CALL instruction, an interrupt, or an exception) back to the calling or interrupted task. The updated state of the task executing the IRET instruction is saved in its TSS. If the task is re-entered later, the code that follows the IRET instruction is executed.

If the NT flag is set and the processor is in IA-32e mode, the IRET instruction causes a general protection exception.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.W prefix promotes operation to 64 bits (IRETQ). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF PE = 0
   THEN
       GOTO REAL-ADDRESS-MODE;
   ELSE
       IF (IA32_EFER.LMA = 0)
               THEN (* Protected mode *)
                   GOTO PROTECTED-MODE;
               ELSE (* IA-32e mode *)
                   GOTO IA-32e-MODE;
       FI;
FI;

REAL-ADDRESS-MODE;
   IF OperandSize = 32
       THEN
           IF top 12 bytes of stack not within stack limits
               THEN #SS; FI;
           tempEIP ←4 bytes at end of stack
           IF tempEIP[31:16] is not zero THEN #GP(0); FI;
           EIP ←Pop();
           CS ←Pop(); (* 32-bit pop, high-order 16 bits discarded *)
           tempEFLAGS ←Pop();
           EFLAGS ←(tempEFLAGS AND 257FD5H) OR (EFLAGS AND 1A0000H);
       ELSE (* OperandSize = 16 *)
           IF top 6 bytes of stack are not within stack limits
               THEN #SS; FI;
           EIP ←Pop(); (* 16-bit pop; clear upper 16 bits *)
           CS ←Pop(); (* 16-bit pop *)
           EFLAGS[15:0] ←Pop();
   FI;
   END;

PROTECTED-MODE:
   IF VM = 1 (* Virtual-8086 mode: PE = 1, VM = 1 *)
       THEN
           GOTO RETURN-FROM-VIRTUAL-8086-MODE; (* PE = 1, VM = 1 *)
   FI;
   IF NT = 1
       THEN
           GOTO TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
   FI;
   IF OperandSize = 32
       THEN
           IF top 12 bytes of stack not within stack limits
               THEN #SS(0); FI;
           tempEIP ←Pop();
```

```
            tempCS ← Pop();
            tempEFLAGS ← Pop();
        ELSE (* OperandSize = 16 *)
            IF top 6 bytes of stack are not within stack limits
                        THEN #SS(0); FI;
            tempEIP ← Pop();
            tempCS ← Pop();
            tempEFLAGS ← Pop();
            tempEIP ← tempEIP AND FFFFH;
            tempEFLAGS ← tempEFLAGS AND FFFFH;
    FI;
    IF tempEFLAGS(VM) = 1 and CPL = 0
        THEN
            GOTO RETURN-TO-VIRTUAL-8086-MODE;
            (* PE = 1, VM = 1 in EFLAGS image *)
        ELSE
            GOTO PROTECTED-MODE-RETURN;
            (* PE = 1, VM = 0 in EFLAGS image *)
    FI;
IA-32e-MODE:
    IF NT = 1
        THEN #GP(0);
    ELSE IF OperandSize = 32
        THEN
            IF top 12 bytes of stack not within stack limits
                THEN #SS(0); FI;
            tempEIP ← Pop();
            tempCS ← Pop();
            tempEFLAGS ← Pop();
        ELSE IF OperandSize = 16
            THEN
                IF top 6 bytes of stack are not within stack limits
                        THEN #SS(0); FI;
                tempEIP ← Pop();
                tempCS ← Pop();
                tempEFLAGS ← Pop();
                tempEIP ← tempEIP AND FFFFH;
                tempEFLAGS ← tempEFLAGS AND FFFFH;
            FI;
        ELSE (* OperandSize = 64 *)
            THEN
                    tempRIP ← Pop();
                    tempCS ← Pop();
                    tempEFLAGS ← Pop();
                    tempRSP ← Pop();
                    tempSS ← Pop();
    FI;
    GOTO IA-32e-MODE-RETURN;
```

```
RETURN-FROM-VIRTUAL-8086-MODE:
(* Processor is in virtual-8086 mode when IRET is executed and stays in virtual-8086 mode *)
   IF IOPL = 3 (* Virtual mode: PE = 1, VM = 1, IOPL = 3 *)
       THEN IF OperandSize = 32
           THEN
               IF top 12 bytes of stack not within stack limits
                   THEN #SS(0); FI;
               IF instruction pointer not within code segment limits
                   THEN #GP(0); FI;
               EIP ←Pop();
               CS ←Pop(); (* 32-bit pop, high-order 16 bits discarded *)
               EFLAGS ←Pop();
               (* VM, IOPL,VIP and VIF EFLAG bits not modified by pop *)
           ELSE (* OperandSize = 16 *)
               IF top 6 bytes of stack are not within stack limits
                   THEN #SS(0); FI;
               IF instruction pointer not within code segment limits
                   THEN #GP(0); FI;
               EIP ←Pop();
               EIP ←EIP AND 0000FFFFH;
               CS ←Pop(); (* 16-bit pop *)
               EFLAGS[15:0] ←Pop(); (* IOPL in EFLAGS not modified by pop *)
           FI;
       ELSE
           #GP(0); (* Trap to virtual-8086 monitor: PE = 1, VM = 1, IOPL < 3 *)
   FI;
END;

RETURN-TO-VIRTUAL-8086-MODE:
   (* Interrupted procedure was in virtual-8086 mode: PE = 1, VM = 1 in flag image *)
   IF top 24 bytes of stack are not within stack segment limits
       THEN #SS(0); FI;
   IF instruction pointer not within code segment limits
       THEN #GP(0); FI;
   CS ←tempCS;
   EIP ←tempEIP;
   EFLAGS ←tempEFLAGS;
   TempESP ←Pop();
   TempSS ←Pop();
   ES ←Pop(); (* Pop 2 words; throw away high-order word *)
   DS ←Pop(); (* Pop 2 words; throw away high-order word *)
   FS ←Pop(); (* Pop 2 words; throw away high-order word *)
   GS ←Pop(); (* Pop 2 words; throw away high-order word *)
   SS:ESP ←TempSS:TempESP;
   CPL ←3;
   (* Resume execution in Virtual-8086 mode *)
END;

TASK-RETURN: (* PE = 1, VM = 1, NT = 1 *)
   Read segment selector in link field of current TSS;
```

```
    IF local/global bit is set to local
    or index not within GDT limits
          THEN #TS (TSS selector); FI;
    Access TSS for task specified in link field of current TSS;
    IF TSS descriptor type is not TSS or if the TSS is marked not busy
          THEN #TS (TSS selector); FI;
    IF TSS not present
          THEN #NP(TSS selector); FI;
    SWITCH-TASKS (without nesting) to TSS specified in link field of current TSS;
    Mark the task just abandoned as NOT BUSY;
    IF EIP is not within code segment limit
          THEN #GP(0); FI;
END;


PROTECTED-MODE-RETURN: (* PE = 1, VM = 0 in flags image *)
    IF return code segment selector is NULL
          THEN GP(0); FI;
    IF return code segment selector addrsses descriptor beyond descriptor table limit
          THEN GP(selector); FI;
    Read segment descriptor pointed to by the return code segment selector;
    IF return code segment descriptor is not a code segment
          THEN #GP(selector); FI;
    IF return code segment selector RPL < CPL
          THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
    and return code segment DPL > return code segment selector RPL
          THEN #GP(selector); FI;
    IF return code segment descriptor is not present
          THEN #NP(selector); FI;
    IF return code segment selector RPL > CPL
          THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
          ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;

RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE = 1, VM = 0 in flags image, RPL = CPL *)
    IF new mode ≠ 64-Bit Mode
          THEN
                IF tempEIP is not within code segment limits
                      THEN #GP(0); FI;
                EIP ← tempEIP;
          ELSE (* new mode = 64-bit mode *)
                IF tempRIP is non-canonical
                            THEN #GP(0); FI;
                RIP ← tempRIP;
    FI;
    CS ← tempCS; (* Segment descriptor information also loaded *)
    EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
    IF OperandSize = 32 or OperandSize = 64
          THEN EFLAGS(RF, AC, ID) ← tempEFLAGS; FI;
    IF CPL ≤IOPL
        THEN EFLAGS(IF) ←tempEFLAGS; FI;
    IF CPL = 0
        THEN EFLAGS(IOPL) ←tempEFLAGS;
        IF OperandSize = 32
            THEN EFLAGS(VM, VIF, VIP) ←tempEFLAGS; FI;
        IF OperandSize = 64
            THEN EFLAGS( VIF, VIP) ←tempEFLAGS; FI;
    FI;
END;

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
    IF OperandSize = 32
        THEN
            IF top 8 bytes on stack are not within limits
                THEN #SS(0); FI;
        ELSE (* OperandSize = 16 *)
            IF top 4 bytes on stack are not within limits
                THEN #SS(0); FI;
    FI;
    Read return segment selector;
    IF stack segment selector is NULL
        THEN #GP(0); FI;
    IF return stack segment selector index is not within its descriptor table limits
        THEN #GP(SSselector); FI;
    Read segment descriptor pointed to by return segment selector;
    IF stack segment selector RPL ≠ RPL of the return code segment selector
    or the stack segment descriptor does not indicate a a writable data segment;
    or the stack segment DPL ≠ RPL of the return code segment selector
        THEN #GP(SS selector); FI;
    IF stack segment is not present
        THEN #SS(SS selector); FI;
    IF new mode ≠ 64-Bit Mode
        THEN
            IF tempEIP is not within code segment limits
                THEN #GP(0); FI;
            EIP ←tempEIP;
        ELSE (* new mode = 64-bit mode *)
            IF tempRIP is non-canonical
                    THEN #GP(0); FI;
            RIP ←tempRIP;
    FI;
    CS ←tempCS;
    EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ←tempEFLAGS;
    IF OperandSize = 32
        THEN EFLAGS(RF, AC, ID) ←tempEFLAGS; FI;
    IF CPL ≤IOPL
        THEN EFLAGS(IF) ←tempEFLAGS; FI;
    IF CPL = 0
        THEN
            EFLAGS(IOPL) ← tempEFLAGS;
            IF OperandSize = 32
                THEN EFLAGS(VM, VIF, VIP) ← tempEFLAGS; FI;
            IF OperandSize = 64
                THEN EFLAGS( VIF, VIP) ← tempEFLAGS; FI;
    FI;
    CPL ← RPL of the return code segment selector;
    FOR each of segment register (ES, FS, GS, and DS)
        DO
            IF segment register points to data or non-conforming code segment
            and CPL > segment descriptor DPL (* Stored in hidden part of segment register *)
                THEN (* Segment register invalid *)
                    SegmentSelector ← 0; (* NULL segment selector *)
            FI;
        OD;
END;

IA-32e-MODE-RETURN: (* IA32_EFER.LMA = 1, PE = 1, VM = 0 in flags image *)
    IF ( (return code segment selector is NULL) or (return RIP is non-canonical) or
            (SS selector is NULL going back to compatibility mode) or
            (SS selector is NULL going back to CPL3 64-bit mode) or
            (RPL <> CPL going back to non-CPL3 64-bit mode for a NULL SS selector) )
        THEN GP(0); FI;
    IF return code segment selector addrsses descriptor beyond descriptor table limit
        THEN GP(selector); FI;
    Read segment descriptor pointed to by the return code segment selector;
    IF return code segment descriptor is not a code segment
        THEN #GP(selector); FI;
    IF return code segment selector RPL < CPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
    and return code segment DPL > return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is not present
        THEN #NP(selector); FI;
    IF return code segment selector RPL > CPL
        THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;
```

Flags Affected

All the flags and fields in the EFLAGS register are potentially modified, depending on the mode of operation of the processor. If performing a return from a nested task to a previous task, the EFLAGS register will be modified according to the EFLAGS image stored in the previous task's TSS.

Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector is NULL.
	If the return instruction pointer is not within the return code segment limit.
#GP(selector)	If a segment selector index is outside its descriptor table limits.
	If the return code segment selector RPL is greater than the CPL.
	If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
	If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
	If the stack segment is not a writable data segment.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
	If the segment descriptor for a code segment does not indicate it is a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is not busy.
	If a TSS segment descriptor specifies that the TSS is not available.
#SS(0)	If the top bytes of stack are not within stack limits.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.

Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit.
#SS	If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit.
	IF IOPL not equal to 3.
#PF(fault-code)	If a page fault occurs.
#SS(0)	If the top bytes of stack are not within stack limits.
#AC(0)	If an unaligned memory reference occurs and alignment checking is enabled.

Compatibility Mode Exceptions

#GP(0)	If EFLAGS.NT[bit 14] = 1.

Other exceptions same as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If EFLAGS.NT[bit 14] = 1.
	If the return code segment selector is NULL.
	If the stack segment selector is NULL going back to compatibility mode.
	If the stack segment selector is NULL going back to CPL3 64-bit mode.
	If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.
	If the return instruction pointer is not within the return code segment limit.
	If the return instruction pointer is non-canonical.
#GP(Selector)	If a segment selector index is outside its descriptor table limits.
	If a segment descriptor memory address is non-canonical.
	If the segment descriptor for a code segment does not indicate it is a code segment.
	If the proposed new code segment descriptor has both the D-bit and L-bit set.
	If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
	If CPL is greater than the RPL of the code segment selector.
	If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
	If the stack segment is not a writable data segment.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
#SS(0)	If an attempt to pop a value off the stack violates the SS limit.
	If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.

13. Incorrect diagram of page directory entry corrected

In Figure 3-27, Chapter 3, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*; the page directory entry was incorrect. The corrected figure is reprinted below. See the change bars for the impacted area.

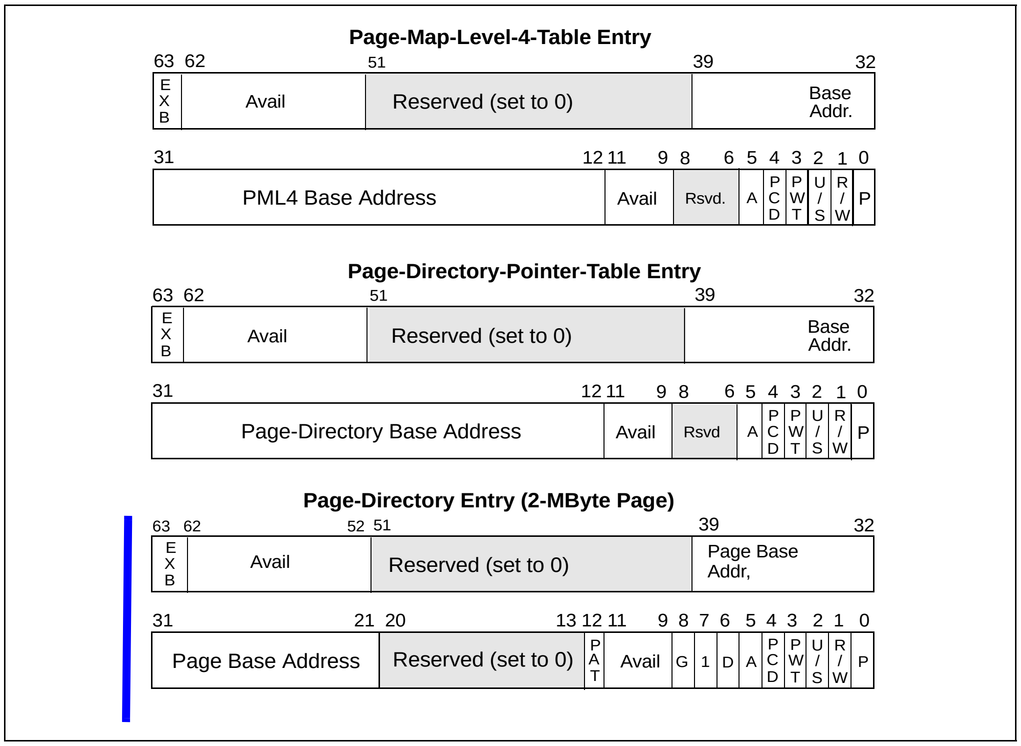

Figure 3-27. Format of Paging Structure Entries for 2-MByte Pages in IA-32e Mode

14. MOV to/from control registers updated

In the "MOV—Move to/from Control Registers" section, *IA-32 Intel Architecture Software Developer's Manual, Volume 2A;* the summary table and paragraph have been updated. The updates correct omissions in the REX.W information. The section is reprinted below. See the change bars for the impacted area.

MOV—Move to/from Control Registers

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 22 /r	MOV CR0,*r32*	N.E.	Valid	Move *r32* to CR0.
REX + 0F 22 /r	MOV CR0,*r64*	Valid	N.E.	Move *r64* to extended CR0.
0F 22 /r	MOV CR2,*r32*	N.E.	Valid	Move *r32* to CR2.
REX + 0F 22 /r	MOV CR2,*r64*	Valid	N.E.	Move *r64* to extended CR2.
0F 22 /r	MOV CR3,*r32*	N.E.	Valid	Move *r32* to CR3.
REX + 0F 22 /r	MOV CR3,*r64*	Valid	N.E.	Move *r64* to extended CR3.
0F 22 /r	MOV CR4,*r32*	N.E.	Valid	Move *r32* to CR4.
REX + 0F 22 /r	MOV CR4,*r64*	Valid	N.E.	Move *r64* to extended CR4.
0F 20 /r	MOV *r32*,CR0	N.E.	Valid	Move CR0 to *r32.*
REX + 0F 20 /r	MOV *r64*,CR0	Valid	N.E.	Move extended CR0 to *r64*.
0F 20 /r	MOV *r32*,CR2	N.E.	Valid	Move CR2 to *r32.*

REX + 0F 20 /r	MOV *r64*,CR2	Valid	N.E.	Move extended CR2 to *r64*.
0F 20 /r	MOV *r32*,CR3	N.E.	Valid	Move CR3 to *r32.*
REX + 0F 20 /r	MOV *r64*,CR3	Valid	N.E.	Move extended CR3 to *r64*.
0F 20 /r	MOV *r32*,CR4	N.E.	Valid	Move CR4 to *r32.*
REX + 0F 20 /r	MOV *r64*,CR4	Valid	N.E.	Move extended CR4 to *r64*.
0F 20 /r	MOV *r32*,CR8	N.E.	N.E.	Move CR8 to *r32.*
REX + 0F 20 /r	MOV *r64*,CR8	Valid	N.E.	Move extended CR8 to *r64*.

Description

Moves the contents of a control register (CR0, CR2, CR3, or CR4) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See “Control Registers” in Chapter 2 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*, for a detailed description of the flags and fields in the control registers.) This instruction can be executed only when the current privilege level is 0.

When loading control registers, programs should not attempt to change the reserved bits; that is, always set reserved bits to the value previously read. An attempt to change CR4's reserved bits will cause a general protection fault. Reserved bits in CR0 and CR3 remain clear after any load of those registers; attempts to set them have no impact. On Pentium 4, Intel Xeon and P6 family processors, CR0.ET remains set after any load of CR0; attempts to clear this bit have no impact.

At the opcode level, the *reg* field within the ModR/M byte specifies which of the control registers is loaded or read. The 2 bits in the *mod* field are always 11B. The *r/m* field specifies the general-purpose register loaded or read.

These instructions have the following side effect:

- When writing to control register CR3, all non-global TLB entries are flushed (see “Translation Lookaside Buffers (TLBs)” in Chapter 3 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*).

The following side effects are implementation specific for the Pentium 4, Intel Xeon, and P6 processor family. Software should not depend on this functionality in all IA-32 processors:

- When modifying any of the paging flags in the control registers (PE and PG in register CR0 and PGE, PSE, and PAE in register CR4), all TLB entries are flushed, including global entries.
- If the PG flag is set to 1 and control register CR4 is written to set the PAE flag to 1 (to enable the physical address extension mode), the pointers in the page-directory pointers table (PDPT) are loaded into the processor (into internal, non-architectural registers).
- If the PAE flag is set to 1 and the PG flag set to 1, writing to control register CR3 will cause the PDPTRs to be reloaded into the processor. If the PAE flag is set to 1 and control register CR0 is written to set the PG flag, the PDPTRs are reloaded into the processor.

In 64-bit mode, the instruction’s default operation size is 64 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W or 66H prefix is ignored. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST ←SRC;
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
	If an attempt is made to write a 1 to any reserved bit in CR4.
	If any of the reserved bits are set in the page-directory pointers table (PDPT) and the loading of a control register causes the PDPT to be loaded into the processor.

Real-Address Mode Exceptions

#GP	If an attempt is made to write a 1 to any reserved bit in CR4.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0).

Virtual-8086 Mode Exceptions

#GP(0)	These instructions cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
	If an attempt is made to write a 1 to any reserved bit in CR3.
	If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
	Attempting to clear CR0.PG[bit 32].
	If an attempt is made to write a 1 to any reserved bit in CR4.
	If an attempt is made to write a 1 to any reserved bit in CR8.
	If an attempt is made to write a 1 to any reserved bit in CR3.
	If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].

15. LGDT/LIDT exceptions updated

In the “LGDT/LIDT—Load Global/Interrupt Descriptor Table Register” section, *IA-32 Intel Architecture Software Developer’s Manual, Volume 2A*; virtual-8086 mode exceptions have been updated. The updated list is printed below. See the change bars for the impacted area.

Virtual-8086 Mode Exceptions

#UD	If source operand is not a memory location.
#GP(0)	The LGDT and LIDT instructions are not recognized in virtual-8086 mode.
#GP	If the current privilege level is not 0.

16. Underflow description corrected

In Section 4.9.1.5, *IA-32 Intel Architecture Software Developer's Manual, Volume 1*; the language has been updated to correct an ambiguity. The updated material is provided below (with changes shown in context). See the change bars for the impacted area.

4.9.1.5 Numeric Underflow Exception (#U)

The processor detects a floating-point numeric underflow condition whenever the result of rounding with unbounded exponent (taking into account precision control for x87) is tiny; that is, less than the smallest possible normalized, finite value that will fit into the destination operand. Table 4-10 shows the threshold range for numeric underflow for each of the floating-point formats (assuming normalized results); underflow occurs when a rounded result falls strictly within the threshold range. The ability to detect and handle underflow is provided to prevent a vary small result from propagating through a computation and causing another exception (such as overflow during division) to be generated at a later time.

Table 4-10. Numeric Underflow (Normalized) Thresholds

Floating-Point Format	Underflow Thresholds*
Single Precision	$\| x \| < 1.0 * 2^{-126}$
Double Precision	$\| x \| < 1.0 * 2^{-1022}$
Double Extended Precision	$\| x \| < 1.0 * 2^{-16382}$

* Where 'x' is the result rounded to destination precision with an unbounded exponent range.

How the processor handles an underflow condition, depends on two related conditions:

- creation of a tiny result
- creation of an inexact result; that is, a result that cannot be represented exactly in the destination format

Which of these events causes an underflow exception to be reported and how the processor responds to the exception condition depends on whether the underflow exception is masked:

- **Underflow exception masked —** The underflow exception is reported (the UE flag is set) only when the result is both tiny and inexact. The processor returns a denormalized result to the destination operand, regardless of inexactness.
- **Underflow exception not masked —** The underflow exception is reported when the result is tiny, regardless of inexactness. The processor leaves the source and destination operands unaltered or stores a biased result in the designating operand (depending whether the underflow exception was generated during an SSE/SSE2/SSE3 floating-point operation or an x87 FPU operation) and invokes a software exception handler.

See the following sections for information regarding the numeric underflow exception when detected while executing x87 FPU instructions or while executing SSE/SSE2/SSE3 instructions:

- x87 FPU; Section 8.5.5, "Numeric Underflow Exception (#U)"

SIMD floating-point exceptions; Section 11.5.2.5, “Numeric Underflow Exception (#U)”

17. IN/OUT virtual-8086 mode exceptions updated

In the “IN—Input from Port” section, *IA-32 Intel Architecture Software Developer’s Manual, Volume 2A* and the “OUT—Output to Port” section, *IA-32 Intel Architecture Software Developer’s Manual, Volume 2B*; the description of Virtual-8086 mode exceptions has been updated. The update for both sections is reprinted below. See the change bars for the impacted area.

Virtual-8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.

18. SYSENTER and SYSEXIT sections updated

In the “SYSENTER—Fast System Call” and “SYSEXIT—Fast Return from Fast System Call” sections, *IA-32 Intel Architecture Software Developer’s Manual, Volume 2B*; a number of updates have been made to address errors and omissions. The sections are reprinted below. See the change bars for the impacted area.

SYSENTER—Fast System Call

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 34	SYSENTER	Valid	Valid	Fast call to privilege level 0 system procedures.

Description

Executes a fast call to a level 0 system procedure or routine. SYSENTER is a companion instruction to SYSEXIT. The instruction is optimized to provide the maximum performance for system calls from user code running at privilege level 3 to operating system or executive procedures running at privilege level 0.

Prior to executing the SYSENTER instruction, software must specify the privilege level 0 code segment and code entry point, and the privilege level 0 stack segment and stack pointer by writing values to the following MSRs:

- IA32_SYSENTER_CS — Contains a 32-bit value, of which the lower 16 bits are the segment selector for the privilege level 0 code segment. This value is also used to compute the segment selector of the privilege level 0 stack segment.
- IA32_SYSENTER_EIP — Contains the 32-bit offset into the privilege level 0 code segment to the first instruction of the selected operating procedure or routine.
- IA32_SYSENTER_ESP — Contains the 32-bit stack pointer for the privilege level 0 stack.

These MSRs can be read from and written to using RDMSR/WRMSR. Register addresses are listed in Table 4-28. The addresses are defined to remain fixed for future IA-32 processors.

Table 4-28. MSRs Used By the SYSENTER and SYSEXIT Instructions

MSR	Address
IA32_SYSENTER_CS	174H
IA32_SYSENTER_ESP	175H
IA32_SYSENTER_EIP	176H

When SYSENTER is executed, the processor:

1. Loads the segment selector from the IA32_SYSENTER_CS into the CS register.
2. Loads the instruction pointer from the IA32_SYSENTER_EIP into the EIP register.
3. Adds 8 to the value in IA32_SYSENTER_CS and loads it into the SS register.
4. Loads the stack pointer from the IA32_SYSENTER_ESP into the ESP register.
5. Switches to privilege level 0.
6. Clears the VM flag in the EFLAGS register, if the flag is set.
7. Begins executing the selected system procedure.

The processor does not save a return IP or other state information for the calling procedure.

The SYSENTER instruction always transfers program control to a protected-mode code segment with a DPL of 0. The instruction requires that the following conditions are met by the operating system:

- The segment descriptor for the selected system code segment selects a flat, 32-bit code segment of up to 4 GBytes, with execute, read, accessed, and non-conforming permissions.
- The segment descriptor for selected system stack segment selects a flat 32-bit stack segment of up to 4 GBytes, with read, write, accessed, and expand-up permissions.

The SYSENTER can be invoked from all operating modes except real-address mode.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. When executing a SYSENTER instruction, the processor does not save state information for the user code, and neither the SYSENTER nor the SYSEXIT instruction supports passing parameters on the stack.

To use the SYSENTER and SYSEXIT instructions as companion instructions for transitions between privilege level 3 code and privilege level 0 operating system procedures, the following conventions must be followed:

- The segment descriptors for the privilege level 0 code and stack segments and for the privilege level 3 code and stack segments must be contiguous in the global descriptor table. This convention allows the processor to compute the segment selectors from the value entered in the SYSENTER_CS_MSR MSR.
- The fast system call "stub" routines executed by user code (typically in shared libraries or DLLs) must save the required return IP and processor state information if a return to the calling procedure is required. Likewise, the operating system or executive procedures called with SYSENTER instructions must have access to and use this saved return and state information when returning to the user code.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
    THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
        THEN
            SYSENTER/SYSEXIT_Not_Supported; FI;
        ELSE
            SYSENTER/SYSEXIT_Supported; FI;
FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

Operation

```
IF CR0.PE = 0 THEN #GP(0); FI;
IF SYSENTER_CS_MSR[15:2] = 0 THEN #GP(0); FI;
EFLAGS.VM ←0;                           (* Insures protected mode execution *)
EFLAGS.IF ←0;                           (* Mask interrupts *)
EFLAGS.RF ←0;

CS.SEL ←SYSENTER_CS_MSR                 (* Operating system provides CS *)
(* Set rest of CS to a fixed value *)
CS.BASE ←0;                             (* Flat segment *)
CS.LIMIT ←FFFFFH;                       (* 4-GByte limit *)
CS.ARbyte.G ←1;                         (* 4-KByte granularity *)
CS.ARbyte.S ←1;
CS.ARbyte.TYPE ←1011B;                  (* Execute +Read, Accessed *)
CS.ARbyte.D ←1;                     (* 32-bit code segment*)
CS.ARbyte.DPL ←0;
CS.SEL.RPL ←0;
CS.ARbyte.P ←1;
CPL ←0;

SS.SEL ←CS.SEL +8;
(* Set rest of SS to a fixed value *)
SS.BASE ←0;                             (* Flat segment *)
SS.LIMIT ←FFFFFH;                       (* 4-GByte limit *)
SS.ARbyte.G ←1;                         (* 4-KByte granularity *)
SS.ARbyte.S ←;
SS.ARbyte.TYPE ←0011B;                  (* Read/Write, Accessed *)
SS.ARbyte.D ←1;                         (* 32-bit stack segment*)
SS.ARbyte.DPL ←0;
SS.SEL.RPL ←0;
SS.ARbyte.P ←1;

ESP ←SYSENTER_ESP_MSR;
EIP ←SYSENTER_EIP_MSR;
```

IA-32e Mode Operation

In IA-32e mode, SYSENTER executes a fast system calls from user code running at privilege level 3 (in compatibility mode or 64-bit mode) to 64-bit executive procedures running at privilege level 0. This instruction is a companion instruction to the SYSEXIT instruction.

In IA-32e mode, the IA32_SYSENTER_EIP and IA32_SYSENTER_ESP MSRs hold 64-bit addresses and must be in canonical form; IA32_SYSENTER_CS must not contain a NULL selector.

When SYSENTER transfers control, the following fields are generated and bits set:

- **Target code segment** — Reads non-NULL selector from IA32_SYSENTER_CS.

- **New CS attributes** — L-bit = 1 (go to 64-bit mode); CS base = 0, CS limit = FFFFFFFFH.
- **Target instruction** — Reads 64-bit canonical address from IA32_SYSENTER_EIP.
- **Stack segment** — Computed by adding 8 to the value from IA32_SYSENTER_CS.
- **Stack pointer** — Reads 64-bit canonical address from IA32_SYSENTER_ESP.
- **New SS attributes** — SS base = 0, SS limit = FFFFFFFFH.

Flags Affected

VM, IF, RF (see Operation above)

Protected Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS[15:2] = 0.

Real-Address Mode Exceptions

#GP(0)	If protected mode is not enabled.

Virtual-8086 Mode Exceptions

Same exceptions as in Protected Mode.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

Same exceptions as in Protected Mode.

SYSEXIT—Fast Return from Fast System Call

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 35	SYSEXIT	Valid	Valid	Fast return to privilege level 3 user code.
REX.W + 0F 35	SYSEXIT	Valid	Valid	Fast return to 64-bit mode privilege level 3 user code.

Description

Executes a fast return to privilege level 3 user code. SYSEXIT is a companion instruction to the SYSENTER instruction. The instruction is optimized to provide the maximum performance for returns from system procedures executing at protections levels 0 to user procedures executing at protection level 3. It must be executed from code executing at privilege level 0.

Prior to executing SYSEXIT, software must specify the privilege level 3 code segment and code entry point, and the privilege level 3 stack segment and stack pointer by writing values into the following MSR and general-purpose registers:

- **IA32_SYSENTER_CS** — Contains a 32-bit value, of which the lower 16 bits are the segment selector for the privilege level 0 code segment in which the processor is currently executing. This value is used to compute the segment selectors for the privilege level 3 code and stack segments.

- **EDX** — Contains the 32-bit offset into the privilege level 3 code segment to the first instruction to be executed in the user code.
- **ECX** — Contains the 32-bit stack pointer for the privilege level 3 stack.

The IA32_SYSENTER_CS MSR can be read from and written to using RDMSR/WRMSR. The register address is listed in Table 4-28. This address is defined to remain fixed for future IA-32 processors.

When SYSEXIT is executed, the processor:

1. Adds 16 to the value in IA32_SYSENTER_CS and loads the sum into the CS selector register.
2. Loads the instruction pointer from the EDX register into the EIP register.
3. Adds 24 to the value in IA32_SYSENTER_CS and loads the sum into the SS selector register.
4. Loads the stack pointer from the ECX register into the ESP register.
5. Switches to privilege level 3.
6. Begins executing the user code at the EIP address.

See “SWAPGS—Swap GS Base Register” for information about using the SYSENTER and SYSEXIT instructions as companion call and return instructions.

The SYSEXIT instruction always transfers program control to a protected-mode code segment with a DPL of 3. The instruction requires that the following conditions are met by the operating system:

- The segment descriptor for the selected user code segment selects a flat, 32-bit code segment of up to 4 GBytes, with execute, read, accessed, and non-conforming permissions.
- The segment descriptor for selected user stack segment selects a flat, 32-bit stack segment of up to 4 GBytes, with expand-up, read, write, and accessed permissions.

The SYSENTER can be invoked from all operating modes except real-address mode and virtual 8086 mode.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
   THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
       THEN
           SYSENTER/SYSEXIT_Not_Supported; FI;
       ELSE
           SYSENTER/SYSEXIT_Supported; FI;
FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

Operation

```
IF SYSENTER_CS_MSR[15:2] = 0 THEN #GP(0); FI;
IF CR0.PE = 0 THEN #GP(0); FI;
IF CPL ≠ 0 THEN #GP(0); FI;
CS.SEL ←(SYSENTER_CS_MSR +16);      (* Segment selector for return CS *)
(* Set rest of CS to a fixed value *)
```

```
CS.BASE ←0;                          (* Flat segment *)
CS.LIMIT ←FFFFFH;                    (* 4-GByte limit *)
CS.ARbyte.G ←1;                      (* 4-KByte granularity *)
CS.ARbyte.S ←1;
CS.ARbyte.TYPE ←1011B;               (* Execute, Read, Non-Conforming Code *)
CS.ARbyte.D ←1;                      (* 32-bit code segment*)
CS.ARbyte.DPL ←3;
CS.SEL.RPL ←3;
CS.ARbyte.P ←1;
CPL ←3;

SS.SEL ←(SYSENTER_CS_MSR +24);       (* Segment selector for return SS *)
(* Set rest of SS to a fixed value *);
SS.BASE ←0;                          (* Flat segment *)
SS.LIMIT ←FFFFFH;                    (* 4-GByte limit *)
SS.ARbyte.G ←1;                      (* 4-KByte granularity *)
SS.ARbyte.S ←;
SS.ARbyte.TYPE ←0011B;               (* Expand Up, Read/Write, Data *)
SS.ARbyte.D ←1;                      (* 32-bit stack segment*)
SS.ARbyte.DPL ←3;
SS.SEL.RPL ←3;
SS.ARbyte.P ←1;

ESP ←ECX;
EIP←EDX;
```

IA-32e Mode Operation

In IA-32e mode, SYSEXIT executes a fast system calls from a 64-bit executive procedures running at privilege level 0 to user code running at privilege level 3 (in compatibility mode or 64-bit mode). This instruction is a companion instruction to the SYSENTER instruction.

In IA-32e mode, the IA32_SYSENTER_EIP and IA32_SYSENTER_ESP MSRs hold 64-bit addresses and must be in canonical form; IA32_SYSENTER_CS must not contain a NULL selector.

When the SYSEXIT instruction transfers control to 64-bit mode user code using REX.W, the following fields are generated and bits set:

- **Target code segment** — Computed by adding 32 to the value in the IA32_SYSENTER_CS.
- **New CS attributes** — L-bit = 1 (go to 64-bit mode).
- **Target instruction** — Reads 64-bit canonical address in RDX.
- **Stack segment** — Computed by adding 8 to the value of CS selector.
- **Stack pointer** — Update RSP using 64-bit canonical address in RCX.

When SYSEXIT transfers control to compatibility mode user code when the operand size attribute is 32 bits, the following fields are generated and bits set:

- **Target code segment** — Computed by adding 16 to the value in IA32_SYSENTER_CS.
- **New CS attributes** — L-bit = 0 (go to compatibility mode).
- **Target instruction** — Fetch the target instruction from 32-bit address in EDX.
- **Stack segment** — Computed by adding 24 to the value in IA32_SYSENTER_CS.
- **Stack pointer** — Update ESP from 32-bit address in ECX.

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS[15:2] = 0.
	If CPL ≠ 0.

Real-Address Mode Exceptions

#GP(0)	If protected mode is not enabled.

Virtual-8086 Mode Exceptions

#GP(0)	Always

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS = 0.
	If CPL ≠ 0.
	If ECX or EDX contains a non-canonical address.

19. LTR section updated

In the "LTR—Load Task Register" section, Chapter 3, *IA-32 Intel Architecture Software Developer's Manual, Volume 2A*; omissions have been corrected. The updated section is reproduced below. See the change bars for the impacted area.

LTR—Load Task Register

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 00 /3	LTR *r/m*16	Valid	Valid	Load *r/m*16 into task register.

Description

Loads the source operand into the segment selector field of the task register. The source operand (a general-purpose register or a memory location) contains a segment selector that points to a task state segment (TSS). After the segment selector is loaded in the task register, the processor uses the segment selector to locate the segment descriptor for the TSS in the global descriptor table (GDT). It then loads the segment limit and base address for the TSS from the segment descriptor into the task register. The task pointed to by the task register is marked busy, but a switch to the task does not occur.

The LTR instruction is provided for use in operating-system software; it should not be used in application programs. It can only be executed in protected mode when the CPL is 0. It is commonly used in initialization code to establish the first task to be executed.

The operand-size attribute has no effect on this instruction.

In 64-bit mode, the operand size is still fixed at 16 bits. The instruction references a 16-byte descriptor to load the 64-bit base.

Operation

```
IF SRC is a null selector
    THEN #GP(0);
IF SRC(Offset) > descriptor table limit OR IF SRC(type) ≠ global
    THEN #GP(segment selector); FI;
Read segment descriptor;
IF segment descriptor is not for an available TSS
    THEN #GP(segment selector); FI;
IF segment descriptor is not present
    THEN #NP(segment selector); FI;
TSSsegmentDescriptor(busy) ←1;
(* Locked read-modify-write operation on the entire descriptor when setting busy flag *)
TaskRegister(SegmentSelector) ←SRC;
TaskRegister(SegmentDescriptor) ←TSSSegmentDescriptor;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the source operand contains a NULL segment selector.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#GP(selector)	If the source selector points to a segment that is not a TSS or to one for a task that is already busy.
	If the selector points to LDT or is beyond the GDT limit.
#NP(selector)	If the TSS is marked not present.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	The LTR instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The LTR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

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.
	If the memory address is in a non-canonical form.
	If the source operand contains a NULL segment selector.
#GP(selector)	If the source selector points to a segment that is not a TSS or to one for a task that is already busy.
	If the selector points to LDT or is beyond the GDT limit.
	If the descriptor type of the upper 8-byte of the 16-byte descriptor is non-zero.
#NP(selector)	If the TSS is marked not present.
#PF(fault-code)	If a page fault occurs.

20. Table updated

In Table 4-3, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*; an earlier correction introduced confusion. Three table cells and the footnote have been updated to address the issue. The updated table is provided below. See the change bars for the impacted area.

Table 4-3. Combined Page-Directory and Page-Table Protection

Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	**Access Type**	**Privilege**	**Access Type**	**Privilege**	**Access Type**
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read-Write	User	Read-Only
User	Read-Write	User	Read-Only	User	Read-Only
User	Read-Write	User	Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

NOTE:

* If the CR0.WP = 1, the access type is determined by the R/W flags of the page-directory and page-table entries. IF CR0.WP = 0, supervisor privilege always permits read-write access.

21. Corrections to Jcc summary table

In the “Jcc—Jump if Condition Is Met” section, *IA-32 Intel Architecture Software Developer’s Manual*, *Volume 2A*; corrections have been made to the summary table at the beginning of the chapter. Applicable cells in the table are reprinted below in context. See the change bars for the impacted area.

Jcc—Jump if Condition Is Met

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
77 *cb*	JA *rel8*	Valid	Valid	Jump short if above (CF=0 and ZF=0).
73 *cb*	JAE *rel8*	Valid	Valid	Jump short if above or equal (CF=0).
72 *cb*	JB *rel8*	Valid	Valid	Jump short if below (CF=1).
76 *cb*	JBE *rel8*	Valid	Valid	Jump short if below or equal (CF=1 or ZF=1).
72 *cb*	JC *rel8*	Valid	Valid	Jump short if carry (CF=1).
E3 *cb*	JCXZ *rel8*	N.E.	Valid	Jump short if CX register is 0.
E3 *cb*	JECXZ *rel8*	Valid	Valid	Jump short if ECX register is 0.
E3 *cb*	JRCXZ *rel8*	Valid	N.E.	Jump short if RCX register is 0.
74 *cb*	JE *rel8*	Valid	Valid	Jump short if equal (ZF=1).
7F *cb*	JG *rel8*	Valid	Valid	Jump short if greater (ZF=0 and SF=OF).
7D *cb*	JGE *rel8*	Valid	Valid	Jump short if greater or equal (SF=OF).
...	...	...	...	...