IA-32 Intel® Architecture Software Developer’s Manual

Documentation Changes

March 27, 2006

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: 252046-016

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Intel, Pentium, Intel Xeon and the Intel logo, and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Copyright © 2002–2006, Intel Corporation. All rights reserved.

Contents

Revision History 4

Preface 5

Summary Table of Changes 6

Documentation Changes 7

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
-015	• Removed Documentation Changes 1-21. • Added Documentation Changes 1-20.	March 9, 2006
-016	• Added Documentation changes 21-23.	March 27, 2006

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 3A, System Programming Guide	253668
IA-32 Intel® Architecture Software Developer's Manual: Volume 3B, System Programming Guide	253669

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	Note added on use of spurious vectors for LVT entries
2	Documentation of settings for FPU flags updated
3	PSUBUSB/PSUBUSW compiler intrinsics corrected
4	Updates made to WRMSR description
5	Documentation on enabling/disabling APIC updated
6	Updated information on MOV CR8
7	Updated data on semaphores and the WC memory type
8	Updates made to VMCALL information
9	Thermal monitor information updated
10	Error in example corrected
11	Documentation for guest error handling updated
12	Missing text and footnote addressed
13	Sentence corrected (where change impacts sense)
14	Text updated to a address a numbered list problem
15	More APIC documentation clarifications (for C6 support)
16	Register reference corrected
17	Appendix A updated to include 3-byte opcode information
18	Segment reference corrected
19	Figure callout corrected
20	Error in table corrected
21	Encodings added for Multi-Byte No Operation
22	Update RMDPMC documentation adding more family-specific data
23	Sections covering variable range MTRRs updated

Documentation Changes

1. Note added on use of spurious vectors for LVT entries

In Section 8.9, "Spurious Interrupt," of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3A.* A note has been added to cover a usage restriction. See the change bar.

8.9 SPURIOUS INTERRUPT

A special situation may occur when a processor raises its task priority to be greater than or equal to the level of the interrupt for which the processor INTR signal is currently being asserted. If at the time the INTA cycle is issued, the interrupt that was to be dispensed has become masked (programmed by software), the local APIC will deliver a spurious-interrupt vector. Dispensing the spurious-interrupt vector does not affect the ISR, so the handler for this vector should return without an EOI.

The vector number for the spurious-interrupt vector is specified in the spurious-interrupt vector register (see Figure 8-23). The functions of the fields in this register are as follows:

Spurious Vector Determines the vector number to be delivered to the processor when the local APIC generates a spurious vector.

(Pentium 4 and Intel Xeon processors.) Bits 0 through 7 of the this field are programmable by software.

(P6 family and Pentium processors). Bits 4 through 7 of the this field are programmable by software, and bits 0 through 3 are hardwired to logical ones. Software writes to bits 0 through 3 have no effect.

APIC Software Enable/Disable Allows software to temporarily enable (1) or disable (0) the local APIC (see Section 8.4.3, "Enabling or Disabling the Local APIC").

Focus Processor Checking Determines if focus processor checking is enabled (0) or disabled (1) when using the lowest-priority delivery mode. In Pentium 4 and Intel Xeon processors, this bit is reserved and should be cleared to 0.

NOTE

Do not program an LVT or IOAPIC RTE with a spurious vector even if you set the mask bit. A spurious vector ISR does not do an EOI. If for some reason an interrupt is generated by an LVT or RTE entry, the bit in the in-service register will be left set for the spurious vector. This will mask all interrupts at the same or lower priority

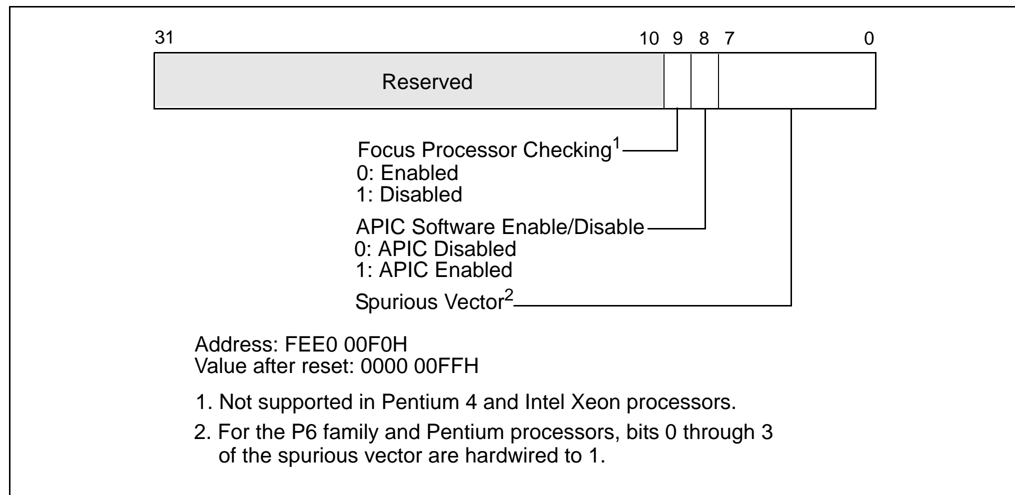

Figure 8-23. Spurious-Interrupt Vector Register (SVR)

2. Documentation of settings for FPU flags updated

In Chapter 3, "Instruction Set Reference, A-M," of the *IA-32 Intel Architecture Software Developer's Manual, Volume 2A*, see the "FXCH—Exchange Register Contents" section. Flag settings have been corrected in the discussion of FXCH. Corrected settings are marked with a change bar.

FXCH—Exchange Register Contents

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 C8+i	FXCH ST(i)	Valid	Valid	Exchange the contents of ST(0) and ST(i).
D9 C9	FXCH	Valid	Valid	Exchange the contents of ST(0) and ST(1).

Description

Exchanges the contents of registers ST(0) and ST(i). If no source operand is specified, the contents of ST(0) and ST(1) are exchanged.

This instruction provides a simple means of moving values in the FPU register stack to the top of the stack [ST(0)], so that they can be operated on by those floating-point instructions that can only operate on values in ST(0). For example, the following instruction sequence takes the square root of the third register from the top of the register stack:

```
FXCH ST(3);
FSQRT;
FXCH ST(3);
```

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
IF (Number-of-operands) is 1
    THEN
        temp ← ST(0);
        ST(0) ← SRC;
        SRC ← temp;
    ELSE
        temp ← ST(0);
        ST(0) ← ST(1);
        ST(1) ← temp;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; otherwise, set to 1.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.

Real-Address Mode Exceptions

Same exceptions as in Protected Mode.

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.

3. PSUBUSB/PSUBUSW compiler intrinsics corrected

In Chapter 4, "Instruction Set Reference, N-Z," of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 2B,* see the "PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation" section. Compiler instrinsics were corrected. The same correction was made to Appendix C of the same volume (not listed below).

PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F D8 /*r*	PSUBUSB *mm, mm/m64*	Valid	Valid	Subtract unsigned packed bytes in *mm/m64* from unsigned packed bytes in *mm* and saturate result.
66 0F D8 /*r*	PSUBUSB *xmm1*, *xmm2/m128*	Valid	Valid	Subtract packed unsigned byte integers in *xmm2/m128* from packed unsigned byte integers in xmm1 and saturate result.
0F D9 /*r*	PSUBUSW *mm, mm/m64*	Valid	Valid	Subtract unsigned packed words in *mm/m64* from unsigned packed words in *mm* and saturate result.
66 0F D9 /*r*	PSUBUSW *xmm1*, *xmm2/m128*	Valid	Valid	Subtract packed unsigned word integers in *xmm2/m128* from packed unsigned word integers in xmm1 and saturate result.

Description

Performs an SIMD subtract of the packed unsigned integers of the source operand (second operand) from the packed unsigned integers of the destination operand (first operand), and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1,* for an illustration of an SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The PSUBUSW instruction subtracts packed unsigned word integers. When an individual word result is less than zero, the saturated value of 0000H is written to 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

```
PSUBUSB instruction with 64-bit operands:
    DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] – SRC (7:0] );
    (* Repeat add operation for 2nd through 7th bytes *)
    DEST[63:56] ← SaturateToUnsignedByte (DEST[63:56] – SRC[63:56];
PSUBUSB instruction with 128-bit operands:
    DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] – SRC[7:0]);
    (* Repeat add operation for 2nd through 14th bytes *)
    DEST[127:120] ← SaturateToUnSignedByte (DEST[127:120] – SRC[127:120]);
PSUBUSW instruction with 64-bit operands:
    DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] – SRC[15:0] );
```

```
    (* Repeat add operation for 2nd and 3rd words *)
    DEST[63:48] ← SaturateToUnsignedWord (DEST[63:48] – SRC[63:48] );
PSUBUSW instruction with 128-bit operands:
    DEST[15:0]  ← SaturateToUnsignedWord (DEST[15:0] – SRC[15:0]);
    (* Repeat add operation for 2nd through 7th words *)
    DEST[127:112] ← SaturateToUnSignedWord (DEST[127:112] – SRC[127:112]);
```

Intel C/C++ Compiler Intrinsic Equivalents

PSUBUSB	__m64 _mm_subs_pu8(__m64 m1, __m64 m2)
PSUBUSB	__m128i _mm_subs_epu8(__m128i m1, __m128i m2)
PSUBUSW	__m64 _mm_subs_pu16(__m64 m1, __m64 m2)
PSUBUSW	__m128i _mm_subs_epu16(__m128i m1, __m128i m2)

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 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 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.

4. Updates made to WRMSR description

In Chapter 4, “Instruction Set Reference, N-Z,” of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 2B,* see the “WRMSR—Write to Model Specific Register” section.Changes have been made to the Description subsection. See the change bars.

WRMSR—Write to Model Specific Register

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 30	WRMSR	Valid	Valid	Write the value in EDX:EAX to MSR specified by ECX.
REX.W + 0F 30	WRMSR	Valid	N.E.	Write the value in RDX[31:0]: RAX[31:0] to MSR specified by RCX.

Description

In legacy and compatibility mode, writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified by the ECX register. The value loaded into the ECX register is the address of the MSR. The contents of the EDX register are copied to high-order 32 bits of the selected MSR and the contents of the EAX register are copied to low-order 32 bits of the MSR. Undefined or reserved bits in an MSR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write reserved bits in a MSR.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated. This includes global entries (see "Translation Lookaside Buffers (TLBs)" in Chapter 3 of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3A)*.

MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Appendix B, "Model-Specific Registers (MSRs)", in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3B,* lists all MSRs that can be read with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The WRMSR instruction is a serializing instruction (see "Serializing Instructions" in Chapter 7 of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3A)*.

The CPUID instruction should be used to determine whether MSRs are supported (EDX[5]=1) before using this instruction.

In 64-bit mode, operation is the same as legacy mode, except that targeted registers are updated by MSR[63:32] = RDX[31:0], MSR[31:0] = RAX[31:0].

IA-32 Architecture Compatibility

The MSRs and the ability to read them with the WRMSR instruction were introduced into the IA-32 architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

Operation

```
IF 64-BIt Mode and REX.W used
    THEN
        MSR[RCX] ← RDX:RAX;
    ELSE IF (Non-64-Bit Modes or Default 64-Bit Mode)
        MSR[ECX] ← EDX:EAX; FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the value in ECX specifies a reserved or unimplemented MSR address.
	If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.

Real-Address Mode Exceptions

#GP(0)	If the value in ECX specifies a reserved or unimplemented MSR address.
	If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.

Virtual-8086 Mode Exceptions

#GP(0)	The WRMSR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

Same exceptions as in Protected Mode.

5. Documentation on enabling/disabling APIC updated

In Section 8.4.3, "Enabling or Disabling the Local APIC," of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3A,* the change details APIC state during C6 transitions. See the change bars.

8.4.3 Enabling or Disabling the Local APIC

The local APIC can be enabled or disabled in either of two ways:

1. Using the APIC global enable/disable flag in the IA32_APIC_BASE MSR (MSR address 1BH; see Figure 8-5):
   - When IA32_APIC_BASE[11] is 0, the processor is functionally equivalent to an IA-32 processor without an on-chip APIC. The CPUID feature flag for the APIC (see Section 8.4.2, "Presence of the Local APIC") is also set to 0.
   - When IA32_APIC_BASE[11] is set to 0, processor APICs based on the 3-wire APIC bus cannot be generally re-enabled until a system hardware reset. The 3-wire bus looses track of arbitration that would be necessary for complete re-enabling. Certain APIC functionality can be enabled (for example: performance and thermal monitoring interrupt generation).
   - For processors that use Front Side Bus (FSB) delivery of interrupts, software may disable or enable the APIC by setting and resetting IA32_APIC_BASE[11]. A hardware reset is not required to re-start APIC functionality.

— When IA32_APIC_BASE[11] is set to 0, prior initialization to the APIC may be lost and the APIC may return to the state described in Section 8.4.7.1, "Local APIC State After Power-Up or Reset".

2. Using the APIC software enable/disable flag in the spurious-interrupt vector register (see Figure 8-23):

— If IA32_APIC_BASE[11] is 1, software can temporarily disable a local APIC at any time by clearing the APIC software enable/disable flag in the spurious-interrupt vector register (see Figure 8-23). The state of the local APIC when in this software-disabled state is described in Section 8.4.7.2, "Local APIC State After It Has Been Software Disabled".

— When the local APIC is in the software-disabled state, it can be re-enabled at any time by setting the APIC software enable/disable flag to 1.

For the Pentium processor, the APICEN pin (which is shared with the PICD1 pin) is used during power-up or RESET to disable the local APIC.

Note that each entry in the LVT has a mask bit that can be used to inhibit interrupts from being delivered to the processor from selected local interrupt sources (the LINT0 and LINT1 pins, the APIC timer, the performance-monitoring counters, the thermal sensor, and/or the internal APIC error detector).

6. Updated information on MOV CR8

In Chapter 3, "Instruction Set Reference, A-M," of the *IA-32 Intel Architecture Software Developer's Manual, Volume 2A,* see the "MOV—Move to/from Control Registers" section.

MOV CR8 is not defined in the architecture as being a serializing instruction. Existing text has been updated to reflect this. The same change was made in Chapter 7, "Multiple-Processor Management," in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3A* (not listed below).

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.*

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 20 /*r*	MOV *r32*,CR8	N.E.	N.E.	Move CR8 to *r32*.[1]
REX + 0F 20 /*r*	MOV *r64*,CR8	Valid	N.E.	Move extended CR8 to *r64*.

NOTE:

1. MOV CR* instructions, except for MOV CR8, are serializing instructions. MOV CR8 is not architecturally defined as a serializing instruction. For more information, see Chapter 7 in *IA-32 Intel® Architecture Software Developer's Manual, Volume 3A.*

7. Updated data on semaphores and the WC memory type

In Section 7.1.2.2, "Software Controlled Bus Locking," of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3A,* the additions document behavior of the WC memory type. See the change bars.

7.1.2.2 Software Controlled Bus Locking

To explicitly force the LOCK semantics, software can use the LOCK prefix with the following instructions when they are used to modify a memory location. An invalid-opcode exception (#UD) is generated when the LOCK prefix is used with any other instruction or when no write operation is made to memory (that is, when the destination operand is in a register).

- The bit test and modify instructions (BTS, BTR, and BTC).
- The exchange instructions (XADD, CMPXCHG, and CMPXCHG8B).
- The LOCK prefix is automatically assumed for XCHG instruction.
- The following single-operand arithmetic and logical instructions: INC, DEC, NOT, and NEG.
- The following two-operand arithmetic and logical instructions: ADD, ADC, SUB, SBB, AND, OR, and XOR.

A locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may be interpreted by the system as a lock for a larger memory area.

Software should access semaphores (shared memory used for signalling between multiple processors) using identical addresses and operand lengths. For example, if one processor accesses a semaphore using a word access, other processors should not access the semaphore using a byte access. Do not use semaphores on the WC memory type.

The integrity of a bus lock is not affected by the alignment of the memory field. The LOCK semantics are followed for as many bus cycles as necessary to update the entire operand. However, it is recommend that locked accesses be aligned on their natural boundaries for better system performance:

- Any boundary for an 8-bit access (locked or otherwise).
- 16-bit boundary for locked word accesses.
- 32-bit boundary for locked doubleword accesses.
- 64-bit boundary for locked quadword accesses.

Locked operations are atomic with respect to all other memory operations and all externally visible events. Only instruction fetch and page table accesses can pass locked instructions. Locked instructions can be used to synchronize data written by one processor and read by another processor.

For the P6 family processors, locked operations serialize all outstanding load and store operations (that is, wait for them to complete). This rule is also true for the Pentium 4 and Intel Xeon processors, with one exception. Load operations that reference weakly ordered memory types (such as the WC memory type) may not be serialized.

Locked instructions should not be used to insure that data written can be fetched as instructions.

NOTE

The locked instructions for the current versions of the Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors allow data written to be fetched as instructions. However, Intel recommends that developers who require the use of self-modifying code use a different synchronizing mechanism, described in the following sections.

8. Updates made to VMCALL information

In Chapter 5, "VMX Instruction Reference," of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 2B,* see the "VMCALL—Call to VM Monitor" section. Pseudocode in the "Operation" subsection has been updated. See the change bars.

VMCALL—Call to VM Monitor

Opcode	Instruction	Description
0F 01 C1	VMCALL	Call to VM monitor by causing VM exit.

Description

This instruction allows guest software can make a call for service into an underlying VM monitor. The details of the programming interface for such calls are VMM-specific; this instruction does nothing more than cause a VM exit, registering the appropriate exit reason.

Use of this instruction in VMX root operation invokes an SMM monitor (see Section 24.16.2 in *IA-32 Intel® Architecture Software Developer's Manual, Volume 3B*). This invocation will activate the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM) if it is not already active (see Section 24.16.6 in *IA-32 Intel® Architecture Software Developer's Manual, Volume 3B*).

Operation

```
IF not in VMX operation
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VM exit;
ELSIF in SMM or if the valid bit in the IA32_SMM_MONITOR_CTL MSR is clear
    THEN VMfail(VMCALL executed in VMX root operation);
ELSIF (RFLAGS.VM = 1) OR (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF dual-monitor treatment of SMIs and SMM is active
```

```
    THEN perform an SMM VM exit (see Section 24.16.2
     of the IA-32 Intel® Architecture Software Developer's Manual, Volume 3B);
ELSIF current-VMCS pointer is not valid
    THEN VMfailInvalid;
ELSIF launch state of current VMCS is not clear
    THEN VMfailValid(VMCALL with non-clear VMCS);
ELSIF VM-exit control fields pertinent to saving state are not valid[1]
    THEN VMfailValid(VMCALL with invalid VM-exit control fields);
ELSE
    enter SMM;
    read revision identifier in MSEG;
    IF revision identifier does not match that supported by processor
        THEN
            leave SMM;
            VMfailValid(VMCALL with incorrect MSEG revision identifier);
        ELSE
            read SMM-monitor features field, MSEG (see Section 24.16.6.2,
            in the IA-32 Intel® Architecture Software Developer's Manual, Volume 3B);
            IF features field is invalid
                THEN
                    leave SMM;
                    VMfailValid(VMCALL with invalid SMM-monitor features);
                ELSE activate dual-monitor treatment of SMIs and SMM (see Section 24.16.6
                in the IA-32 Intel® Architecture Software Developer's Manual, Volume 3B);
            FI;
        FI;
FI;
```

Flags Affected

See the operation section and Section 5.2.

Use of Prefixes

LOCK	Causes #UD
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ)
Segment overrides	Ignored
Operand size	Causes #UD
Address size	Ignored
REX	Ignored

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0 and the logical processor is in VMX root operation.
#UD	If executed outside VMX operation.

1. This includes the “save” VM-exit controls and the VM-exit MSR-store address and count fields.

Real-Address Mode Exceptions

#UD A logical processor cannot be in real-address mode while in VMX operation and the VMCALL instruction is not recognized outside VMX operation.

Virtual-8086 Mode Exceptions

#UD If executed outside VMX non-root operation.

Compatibility Mode Exceptions

#UD If executed outside VMX non-root operation.

64-Bit Mode Exceptions

#UD If executed outside VMX operation.

9. Thermal monitor information updated

In Section 13.2.2, “Thermal Monitor,” in the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3A,* the thermal monitor section has been updated to clarify the data provided about enabling mechanisms.

13.2.2 Thermal Monitor

Pentium 4, Intel Xeon and Pentium M processors introduced a second temperature sensor that is factory-calibrated to trip when the processor’s core temperature crosses a level corresponding to the recommended thermal design envelop. The trip-temperature of the second sensor is calibrated below the temperature assigned to the catastrophic shutdown detector.

13.2.2.1 Thermal Monitor 1

The Pentium 4 processor uses the second temperature sensor in conjunction with a mechanism called TM1 (Thermal Monitor 1) to control the core temperature of the processor. TM1 controls the processor’s temperature by modulating the duty cycle of the processor clock. Modulation of duty cycles is processor model specific. Note that the processors STPCLK# pin is not used here; the stop-clock circuitry is controlled internally.

Support for TM1 is indicated by CPUID.1:EDX.TM[bit 29] = 1.

TM1 is enabled by setting the thermal-monitor enable flag (bit 3) in IA32_MISC_ENABLE [see Appendix B, “Model-Specific Registers (MSRs)”]. Following a power-up or reset, the flag is cleared, disabling TM1. BIOS is required to enable only one automatic thermal monitoring modes. Operating systems and applications must not disable the operation of these mechanisms.

13.2.2.2 Thermal Monitor 2

An additional automatic thermal protection mechanism, called Thermal Monitor 2 (TM2), was introduced in the Intel Pentium M processor and also incorporated in newer models of the Pentium 4 processor family. TM2 controls the core temperature of the processor by reducing the operating frequency and voltage of the processor and offers a higher performance level for a given level of power reduction than TM1.

TM2 is triggered by the same temperature sensor as TM1. The mechanism to enable TM2 may be implemented differently across various IA-32 processor families with different CPUID signatures in the family encoding value, but will be uniform within an IA-32 processor family.

Support for TM2 is indicated by CPUID.1:ECX.TM2[bit 8] = 1.

13.2.2.3 Two Methods for Enabling TM2

On processors with CPUID family/model/stepping signature encoded as 0x69n or 0x6Dn (early Pentium M processors), TM2 is enabled if the TM_SELECT flag (bit 16) of the MSR_THERM2_CTL register is set to 1 (Figure 13-2) and bit 3 of the IA32_MISC_ENABLE register is set to 1.

Following a power-up or reset, the TM_SELECT flag is cleared. BIOS is required to enable either TM1 or TM2. Operating systems and applications must not disable the mechanisms that enable TM1 or TM2. If bit 3 of the IA32_MISC_ENABLE register is set and TM_SELECT flag of the MSR_THERM2_CTL register is cleared, TM1 is enabled.

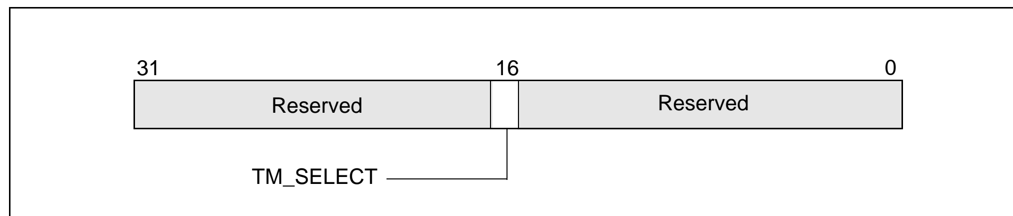

Figure 13-2. MSR_THERM2_CTL Register On Processors with CPUID Family/Model/Stepping Signature Encoded as 0x69n or 0x6Dn

On processors introduced after the Pentium 4 processor (this includes most Pentium M processors), the method used to enable TM2 is different. TM2 is enable by setting bit 13 of IA32_MISC_ENABLE register to 1.

The target operating frequency and voltage for the TM2 transition after TM2 is triggered is specified by the value written to MSR_THERM2_CTL, bits 15:0 (Figure 13-3). Following a power-up or reset, BIOS is required to enable at least one of these two thermal monitoring mechanisms. If both TM1 and TM2 are supported, BIOS may choose to enable TM2 instead of TM1. Operating systems and applications must not disable the mechanisms that enable TM1or TM2; and they must not alter the value in bits 15:0 of the MSR_THERM2_CTL register.

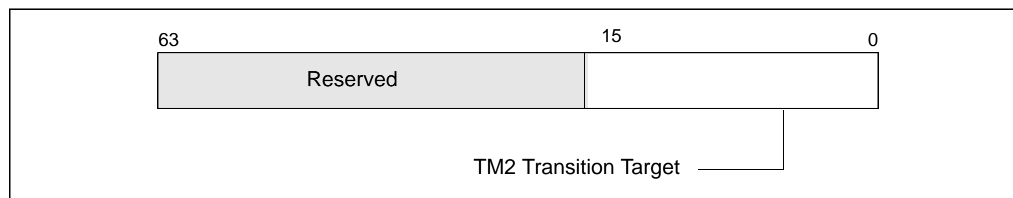

Figure 13-3. MSR_THERM2_CTL Register for Supporting TM2

10. Error in example corrected

In Chapter 3, “Instruction Set Reference, A-M,” of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 2B,* see the “CPUID—CPU Identification” subsection. Example 3-1 has been updated to address an error. See the change bar.

Example 3-1. Example of Cache and TLB Interpretation

The first member of the family of Pentium 4 processors returns the following information about caches and TLBs when the CPUID executes with an input value of 2:

```
EAX    66 5B 50 01H
EBX    0H
ECX    0H
EDX    00 7A 70 00H
```

Which means:

- The least-significant byte (byte 0) of register EAX is set to 01H. This indicates that CPUID needs to be executed once with an input value of 2 to retrieve complete information about caches and TLBs.
- The most-significant bit of all four registers (EAX, EBX, ECX, and EDX) is set to 0, indicating that each register contains valid 1-byte descriptors.
- Bytes 1, 2, and 3 of register EAX indicate that the processor has:
  — 50H - a 64-entry instruction TLB, for mapping 4-KByte and 2-MByte or 4-MByte pages.
  — 5BH - a 64-entry data TLB, for mapping 4-KByte and 4-MByte pages.
  — 66H - an 8-KByte 1st level data cache, 4-way set associative, with a 64-Byte cache line size.
- The descriptors in registers EBX and ECX are valid, but contain NULL descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor has:
  — 00H - NULL descriptor.
  — 70H - Trace cache: 12 K-μop, 8-way set associative.
  — 7AH - a 256-KByte 2nd level cache, 8-way set associative, with a sectored, 64-byte cache line size.
  — 00H - NULL descriptor.

11. Documentation for guest error handling updated

Section 25.7.1.2, “Resuming Guest Software after Handling an Exception,” of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3B* has been re-structured for clarity. See the change bars.

25.7.1.2 Resuming Guest Software after Handling an Exception

If the VMM determines that a VM exit was caused by an exception due to a condition established by the VMM itself, it may choose to resume guest software after removing the condition. The approach for removing the condition may be specific to the VMM’s software architecture. and algorithms This section describes how guest software may be resumed after removing the condition.

In general, the VMM can resume guest software simply by executing VMRESUME. The following items provide details of cases that may require special handling:

- Bit 12 of the VM-exit interruption-information field indicates that the VM exit was due to a fault encountered during an execution of the IRET instruction that unblocked non-maskable interrupts (NMIs). In particular, it provides this indication if the following are all true:
  - — The "NMI exiting" VM-execution control is 0.
  - — Bit 31 (valid) in the IDT-vectoring information field is 0.
  - — The value of bits 7:0 (vector) of the VM-exit interruption-information field is not 8 (the VM exit is not due to a double-fault exception).

  If these are all true and bit 12 of the VM-exit interruption-information field is 1, NMIs were blocked before guest software executed the IRET instruction that caused the fault that caused the VM exit. The VMM should set bit 3 (blocking by NMI) in the interruptibility-state field (using VMREAD and VMWRITE) before resuming guest software.
- Bit 31 (valid) of the IDT-vectoring information field indicates, if set, that the exception causing the VM exit occurred while another event was being delivered to guest software. The VMM should ensure that the other event is delivered when guest software is resumed. It can do so using VM-entry event injection, as described in and below: It can do so using the VM-entry event injection described in Section 22.5 and detailed in the following paragraphs:
  - — The VMM can copy (using VMREAD and VMWRITE) the contents of the IDT-vectoring information field (which is presumed valid) to the VM-entry interruption-information field (which, if valid, will cause the exception to be delivered as part of the next VM entry). Care should be taken to ensure that reserved bits 30:12 in the VM-entry interruption-information field are 0. In particular, the value of bit 12 in the IDT-vectoring information field is undefined after all VM exits. If this bit is copied as 1 into the VM-entry interruption-information field, the next VM entry will fail because the bit should be 0.
  - — The VMM can also copy the contents of the IDT-vectoring error-code field to the VM-entry exception error-code field. This need not be done if bit 11 (error code valid) is clear in the IDT-vectoring information field.
  - — The VMM can also copy the contents of the VM-exit instruction-length field to the VM-entry instruction-length field. This need be done only if bits 10:8 (interruption type) in the IDT-vectoring information field indicate either software interrupt, privileged software exception, or software exception.

12. Missing text and footnote addressed

In Section 24.16.4.2, "Checks on VM-Execution Control Fields," of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3B,* information has been corrected and added. See the change bars.

24.16.4.2 Checks on VM-Execution Control Fields

VM entries that return from SMM differ from other VM entries with regard to the checks performed on the VM-execution control fields specified in Section 22.2.1.1. They do not apply the checks to the current VMCS. Instead, VM-entry behavior depends on whether the executive-VMCS pointer field contains the VMXON pointer: [4]

- If the executive-VMCS pointer field contains the VMXON pointer (the VM entry remains in VMX root operation), the checks are not performed at all.
- If the executive-VMCS pointer field does not contain the VMXON pointer (the VM entry enters VMX non-root operation), the checks are performed on the VM-execution control fields in the executive VMCS (the VMCS referenced by the executive-VMCS pointer field in the

current VMCS). These checks are performed after checking the executive-VMCS pointer field itself (for proper alignment).

24.16.4.3 Checks on Guest Non-Register State

For VM entries that return from SMM, the activity-state field must not indicate the wait-for-SIPI state if the executive-VMCS pointer field contains the VMXON pointer (the VM entry is to VMX root operation).[4]

4 An SMM monitor can determine the VMXON pointer by reading the executive-VMCS pointer field in the current VMCS after the SMM VM exit that activates the dual-monitor treatment.

13. Sentence corrected (where change impacts sense)

In Section 24.15.3, “Protection of CR4.VMXE in SMM,” of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3B,* a sentence has been corrected. See the change bar.

24.15.3 Protection of CR4.VMXE in SMM

Under the default treatment, CR4.VMXE is treated as a reserved bit while a logical processor is in SMM. Any attempt by software running in SMM to set this bit causes a general-protection exception. In addition, software cannot use VMX instructions or enter VMX operation while in SMM.

14. Text updated to a address a numbered list problem

In Section 22.7, “VM-Entry Failures During or After Loading Guest State,” of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3B,* a numbered list was corrected and text was edited. See the change bar.

22.7 VM-ENTRY FAILURES DURING OR AFTER LOADING GUEST STATE

VM-entry failures due to the checks identified in Section 22.3.1 and failures during the MSR loading identified in Section 22.4 are treated differently from those that occur earlier in VM entry. In these cases, the following steps take place:

1. Information about the VM-entry failure is recorded in the VM-exit information fields:
   — Exit reason.
      - Bits 15:0 of this field contain the basic exit reason. It is loaded with a number indicating the general cause of the VM-entry failure. The following numbers are used:
        33. VM-entry failure due to invalid guest state. A VM entry failed one of the checks identified in Section 22.3.1.
        34. VM-entry failure due to MSR loading. A VM entry failed in an attempt to load MSRs (see Section 22.4).
        41. VM-entry failure due to machine check. A machine check occurred during VM entry (see Section 22.8).

  - Bit 31 is set to 1 to indicate a VM-entry failure.
  - The remainder of the field (bits 30:16) is cleared.
- Exit qualification. This field is set based on the exit reason.
  - VM-entry failure due to invalid guest state. In most cases, the exit qualification is cleared to 0. The following non-zero values are used in the cases indicated:
    1. Not used.
    2. Failure was due to a problem loading the PDPTRs (see Section 22.3.1.6).
    3. Failure was due to an attempt to inject a non-maskable interrupt (NMI) into a guest that is blocking events through the STI blocking bit in the interruptibility-state field. Such failures are implementation-specific (see Section 22.3.1.5).
    4. Failure was due to an invalid VMCS link pointer (see Section 22.3.1.5).

    Note that VM-entry checks on guest-state fields may be performed in any order. Thus, an indication by exit qualification of one cause does not imply that there are not also other errors. Different processors may give different exit qualifications for the same VMCS.
  - VM-entry failure due to MSR loading. The exit qualification is loaded to indicate which entry in the VM-entry MSR-load area caused the problem (1 for the first entry, 2 for the second, etc.).
- All other VM-exit information fields are unmodified.

2. Processor state is loaded as would be done on a VM exit (see Section 23.5). If this results in [CR4.PAE & CR0.PG & ~IA32_EFER.LMA] = 1, page-directory pointers (PDPTRS) may be checked and loaded (see Section 23.5.4).
3. The state of blocking by NMI is what it was before VM entry.
4. MSRs are loaded as specified in the VM-exit MSR-load area (see Section 23.6).

Although this process resembles that of a VM exit, many steps taken during a VM exit do not occur for these VM-entry failures:

- Most VM-exit information fields are not updated (see step 1 above).
- The valid bit in the VM-entry interruption-information field is not cleared.
- The guest-state area is not modified.
- No MSRs are saved into the VM-exit MSR-store area.

15. More APIC documentation clarifications (for C6 support)

In Section 8.4.1, “The Local APIC Block Diagram,” and Section 8.6.1, “Interrupt Command Register (ICR),” of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3A*, additional APIC documentation clarifications have been included. See the change bars and bold text.

8.4.1 The Local APIC Block Diagram

Figure 8-4 gives a functional block diagram for the local APIC. Software interacts with the local APIC by reading and writing its registers. APIC registers are memory-mapped to a 4-KByte region of the processor’s physical address space with an initial starting address of FEE00000H. For correct

APIC operation, this address space must be mapped to an area of memory that has been designated as strong uncacheable (UC). See Section 10.3, “Methods of Caching Available.”

In MP system configurations, the APIC registers for IA-32 processors on the system bus are initially mapped to the same 4-KByte region of the physical address space. Software has the option of changing initial mapping to a different 4-KByte region for all the local APICs or of mapping the APIC registers for each local APIC to its own 4-KByte region. Section 8.4.5, “Relocating the Local APIC Registers,” describes how to relocate the base address for APIC registers.

NOTE

For P6 family, Pentium 4, and Intel Xeon processors, the APIC handles all memory accesses to addresses within the 4-KByte APIC register space internally and no external bus cycles are produced. For the Pentium processors with an on-chip APIC, bus cycles are produced for accesses to the APIC register space. Thus, for software intended to run on Pentium processors, system software should explicitly not map the APIC register space to regular system memory. Doing so can result in an invalid opcode exception (#UD) being generated or unpredictable execution.

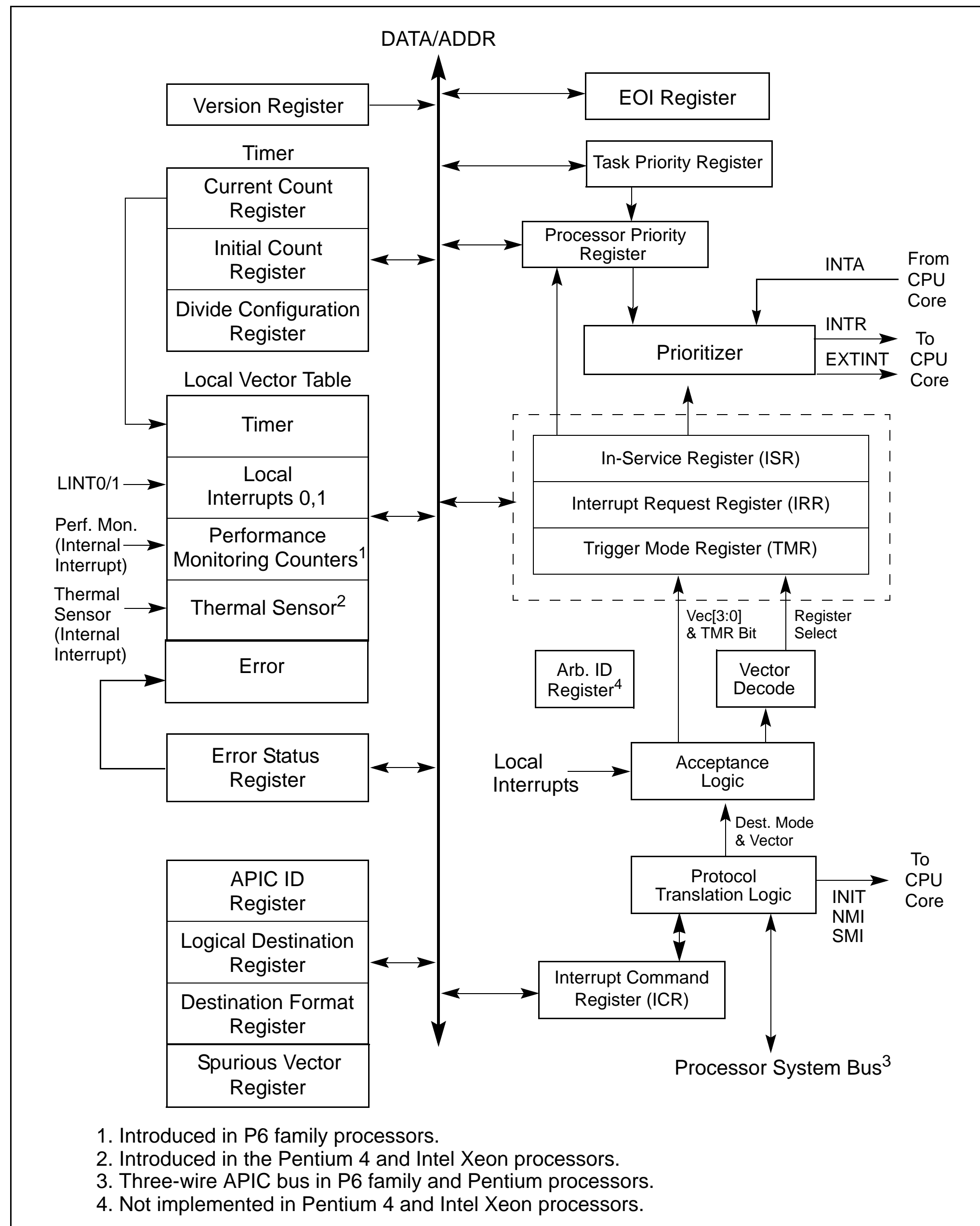

Figure 8-4. Local APIC Structure

Table 8-1 shows how the APIC registers are mapped into the 4-KByte APIC register space. Registers are 32 bits, 64 bits, or 256 bits in width; all are aligned on 128-bit boundaries. All 32-bit registers should be accessed using 128-bit aligned 32-bit loads or stores. **Some processors may support loads and stores of less than 32 bits to some of the APIC registers.** This is model specific behavior and is not guaranteed to work on all processors. Wider registers (64-bit or 256-bit) must be accessed using multiple 32-bit loads or stores, with the first access being 128-bit aligned. If a LOCK prefix is used with a MOV instruction that accesses the APIC address space, the prefix is

ignored. The locking operation does not take place. All the registers listed in Table 8-1 are described in the following sections.

The local APIC registers listed in Table 8-1 are not MSRs. The only MSR associated with the programming of the local APIC is the IA32_APIC_BASE MSR (see Section 8.4.3, “Enabling or Disabling the Local APIC”).

Table 8-1. Local APIC Register Address Map

Address	Register Name	Software Read/Write
FEE0 0000H	Reserved	
FEE0 0010H	Reserved	
FEE0 0020H	Local APIC ID Register	Read/Write.
FEE0 0030H	Local APIC Version Register	Read Only.
FEE0 0040H	Reserved	
FEE0 0050H	Reserved	
FEE0 0060H	Reserved	
FEE0 0070H	Reserved	
FEE0 0080H	Task Priority Register (TPR)	Read/Write.
FEE0 0090H	Arbitration Priority Register[1] (APR)	Read Only.
FEE0 00A0H	Processor Priority Register (PPR)	Read Only.
FEE0 00B0H	EOI Register	Write Only.
FEE0 00C0H	Reserved	
FEE0 00D0H	Logical Destination Register	Read/Write.
FEE0 00E0H	Destination Format Register	Bits 0-27 Read only; bits 28-31 Read/Write.
FEE0 00F0H	Spurious Interrupt Vector Register	Bits 0-8 Read/Write; bits 9-31 Read Only.
FEE0 0100H through FEE0 0170H	In-Service Register (ISR)	Read Only.
FEE0 0180H through FEE0 01F0H	Trigger Mode Register (TMR)	Read Only.
FEE0 0200H through FEE0 0270H	Interrupt Request Register (IRR)	Read Only.
FEE0 0280H	Error Status Register	Read Only.
FEE0 0290H through FEE0 02F0H	Reserved	
FEE0 0300H	Interrupt Command Register (ICR) [0-31]	Read/Write.
FEE0 0310H	Interrupt Command Register (ICR) [32-63]	Read/Write.
FEE0 0320H	LVT Timer Register	Read/Write.
FEE0 0330H	LVT Thermal Sensor Register[2]	Read/Write.
FEE0 0340H	LVT Performance Monitoring Counters Register[3]	Read/Write.
FEE0 0350H	LVT LINT0 Register	Read/Write.
FEE0 0360H	LVT LINT1 Register	Read/Write.
FEE0 0370H	LVT Error Register	Read/Write.
FEE0 0380H	Initial Count Register (for Timer)	Read/Write.
FEE0 0390H	Current Count Register (for Timer)	Read Only.

Table 8-1. Local APIC Register Address Map (Continued)

Address	Register Name	Software Read/Write
FEE0 03A0H through FEE0 03D0H	Reserved	
FEE0 03E0H	Divide Configuration Register (for Timer)	Read/Write.
FEE0 03F0H	Reserved	

NOTES:

1. Not supported in the Pentium 4 and Intel Xeon processors.
2. Introduced in the Pentium 4 and Intel Xeon processors. This APIC register and its associated function are implementation dependent and may not be present in future IA-32 processors.
3. Introduced in the Pentium Pro processor. This APIC register and its associated function are implementation dependent and may not be present in future IA-32 processors.

------------------------------------- *Second correction* --

8.6.1 Interrupt Command Register (ICR)

The interrupt command register (ICR) is a 64-bit local APIC register (see Figure 8-12) that allows software running on the processor to specify and send interprocessor interrupts (IPIs) to other IA-32 processors in the system.

To send an IPI, software must set up the ICR to indicate the type of IPI message to be sent and the destination processor or processors. (All fields of the ICR are read-write by software with the exception of the delivery status field, which is read-only.) The act of writing to the low doubleword of the ICR causes the IPI to be sent.

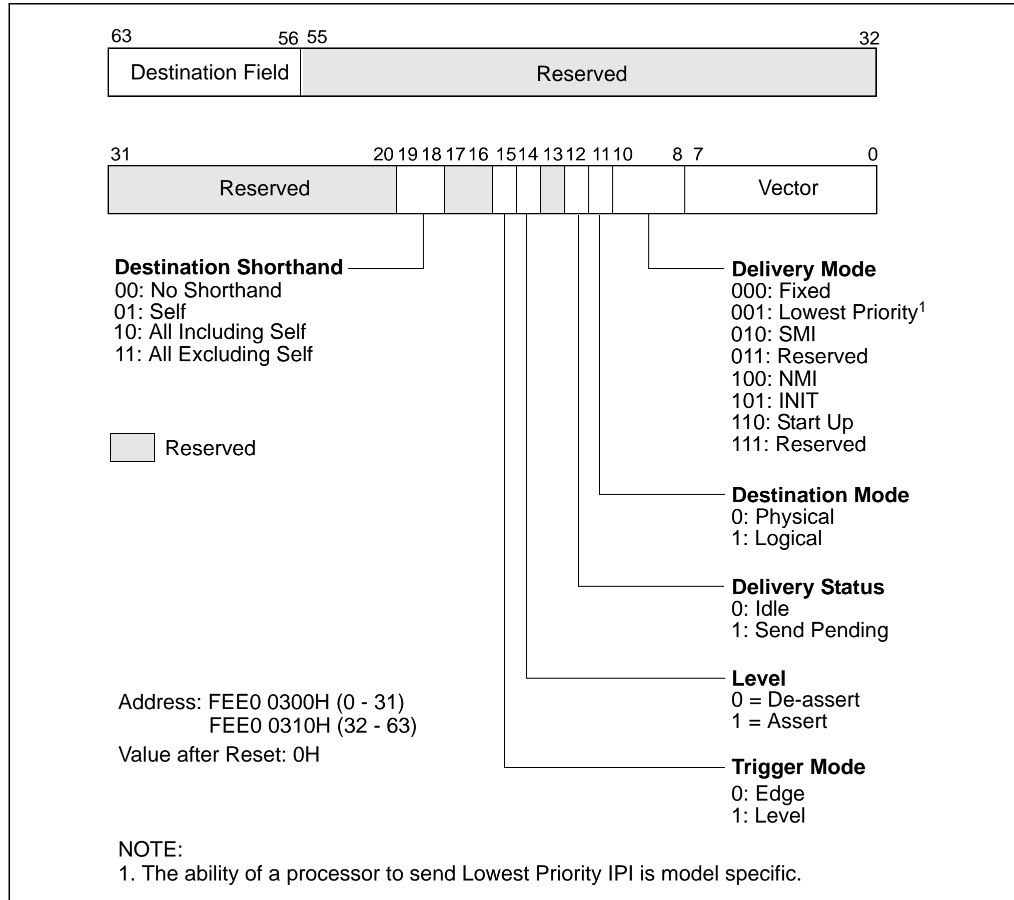

Figure 8-12. Interrupt Command Register (ICR)

The ICR consists of the following fields.

Vector The vector number of the interrupt being sent.

Delivery Mode Specifies the type of IPI to be sent. This field is also know as the IPI message type field.

000 (Fixed) Delivers the interrupt specified in the vector field to the target processor or processors.

001 (Lowest Priority)
Same as fixed mode, except that the interrupt is delivered to the processor executing at the lowest priority among the set of processors specified in the destination field. The ability for a processor to send a lowest priority IPI is model specific and should be avoided by BIOS and operating system software.

010 (SMI) Delivers an SMI interrupt to the target processor or processors. The vector field must be programmed to 00H for future compatibility.

011 (Reserved)

100 (NMI) Delivers an NMI interrupt to the target processor or processors. The vector information is ignored.

101 (INIT) Delivers an INIT request to the target processor or processors, which causes them to perform an INIT. As a result of this IPI message, all the target processors perform an INIT. The vector field must be programmed to 00H for future compatibility.

101 (INIT Level De-assert)
(Not supported in the Pentium 4 and Intel Xeon processors.) Sends a synchronization message to all the local APICs in the system to set their arbitration IDs (stored in their Arb ID registers) to the values of their APIC IDs (see Section 8.7, "System and APIC Bus Arbitration"). For this delivery mode, the level flag must be set to 0 and trigger mode flag to 1. This IPI is sent to all processors, regardless of the value in the destination field or the destination shorthand field; however, software should specify the "all including self" shorthand.

110 (Start-Up) Sends a special "start-up" IPI (called a SIPI) to the target processor or processors. The vector typically points to a start-up routine that is part of the BIOS boot-strap code (see Section 7.5, "Multiple-Processor (MP) Initialization"). IPIs sent with this delivery mode are not automatically retried if the source APIC is unable to deliver it. It is up to the software to determine if the SIPI was not successfully delivered and to reissue the SIPI if necessary.

Destination Mode Selects either physical (0) or logical (1) destination mode (see Section 8.6.2, "Determining IPI Destination").

Delivery Status (Read Only)
Indicates the IPI delivery status, as follows:

0 (Idle) There is currently no IPI activity for this local APIC, or the previous IPI sent from this local APIC was delivered and accepted by the target processor or processors.

1 (Send Pending)
Indicates that the last IPI sent from this local APIC has not yet been accepted by the target processor or processors.

Level For the INIT level de-assert delivery mode this flag must be set to 0; for all other delivery modes it must be set to 1. (This flag has no meaning in Pentium 4 and Intel Xeon processors, and will always be issued as a 1.)

Trigger Mode Selects the trigger mode when using the INIT level de-assert delivery mode: edge (0) or level (1). It is ignored for all other delivery modes. (This flag has no meaning in Pentium 4 and Intel Xeon processors, and will always be issued as a 0.)

Destination Shorthand Indicates whether a shorthand notation is used to specify the destination of the interrupt and, if so, which shorthand is used. Destination shorthands are used in place of the 8-bit destination field, and can be sent by software using a single write to the low doubleword of the ICR. Shorthands are defined for the following cases: software self interrupt, IPIs to all processors in the system including the sender, IPIs to all processors in the system excluding the sender.

00: (No Shorthand)
The destination is specified in the destination field.

01: (Self) The issuing APIC is the one and only destination of the IPI. This destination shorthand allows software to interrupt the processor on which it is executing. An APIC implementation is free to deliver the self-interrupt message internally or to issue the message to the bus and “snoop” it as with any other IPI message.

10: (All Including Self)
The IPI is sent to all processors in the system including the processor sending the IPI. The APIC will broadcast an IPI message with the destination field set to FH for Pentium and P6 family processors and to FFH for Pentium 4 and Intel Xeon processors.

11: (All Excluding Self)
The IPI is sent to all processors in a system with the exception of the processor sending the IPI. The APIC broadcasts a message with the physical destination mode and destination field set to 0xFH for Pentium and P6 family processors and to 0xFFH for Pentium 4 and Intel Xeon processors. Support for this destination shorthand in conjunction with the lowest-priority delivery mode is model specific. For Pentium 4 and Intel Xeon processors, when this shorthand is used together with lowest priority delivery mode, the IPI may be redirected back to the issuing processor.

Destination Specifies the target processor or processors. This field is only used when the destination shorthand field is set to 00B. If the destination mode is set to physical, then bits 56 through 59 contain the APIC ID of the target processor for Pentium and P6 family processors and bits 56 through 63 contain the APIC ID of the target processor the for Pentium 4 and Intel Xeon processors. If the destination mode is set to logical, the interpretation of the 8-bit destination field depends on the settings of the DFR and LDR registers of the local APICs in all the processors in the system (see Section 8.6.2, “Determining IPI Destination”).

Not all combinations of options for the ICR are valid. Table 8-3 shows the valid combinations for the fields in the ICR for the Pentium 4 and Intel Xeon processors; Table 8-4 shows the valid combinations for the fields in the ICR for the P6 family processors. **Also note that the lower half of the ICR may not be preserved over transitions to the deepest C-States.**

Table 8-3. Valid Combinations for the Pentium 4 and Intel Xeon Processors' Local xAPIC Interrupt Command Register

Destination Shorthand	Valid/ Invalid	Trigger Mode	Delivery Mode	Destination Mode
No Shorthand	Valid	Edge	All Modes[1]	Physical or Logical
No Shorthand	Invalid[2]	Level	All Modes	Physical or Logical
Self	Valid	Edge	Fixed	X[3]
Self	Invalid[2]	Level	Fixed	X
Self	Invalid	X	Lowest Priority, NMI, INIT, SMI, Start-Up	X
All Including Self	Valid	Edge	Fixed	X
All Including Self	Invalid[2]	Level	Fixed	X
All Including Self	Invalid	X	Lowest Priority, NMI, INIT, SMI, Start-Up	X
All Excluding Self	Valid	Edge	Fixed, Lowest Priority[1,4], NMI, INIT, SMI, Start-Up	X
All Excluding Self	Invalid[2]	Level	FIxed, Lowest Priority[4], NMI, INIT, SMI, Start-Up	X

NOTES:
1. The ability of a processor to send a lowest priority IPI is model specific.
2. For these interrupts, if the trigger mode bit is 1 (Level), the local xAPIC will override the bit setting and issue the interrupt as an edge triggered interrupt.
3. X means the setting is ignored.
4. When using the "lowest priority" delivery mode and the "all excluding self" destination, the IPI can be redirected back to the issuing APIC, which is essentially the same as the "all including self" destination mode.

Table 8-4. Valid Combinations for the P6 Family Processors' Local APIC Interrupt Command Register

Destination Shorthand	Valid/ Invalid	Trigger Mode	Delivery Mode	Destination Mode
No Shorthand	Valid	Edge	All Modes[1]	Physical or Logical
No Shorthand	Valid[2]	Level	Fixed, Lowest Priority[1], NMI	Physical or Logical
No Shorthand	Valid[3]	Level	INIT	Physical or Logical
Self	Valid	Edge	Fixed	X[4]
Self	1	Level	Fixed	X
Self	Invalid[5]	X	Lowest Priority, NMI, INIT, SMI, Start-Up	X
All including Self	Valid	Edge	Fixed	X
All including Self	Valid[2]	Level	Fixed	X
All including Self	Invalid[5]	X	Lowest Priority, NMI, INIT, SMI, Start-Up	X
All excluding Self	Valid	Edge	All Modes[1]	X
All excluding Self	Valid[2]	Level	Fixed, Lowest Priority[1], NMI	X
All excluding Self	Invalid[5]	Level	SMI, Start-Up	X
All excluding Self	Valid[3]	Level	INIT	X
X	Invalid[5]	Level	SMI, Start-Up	X

Table 8-4. Valid Combinations for the P6 Family Processors' Local APIC Interrupt Command Register (Continued)

Destination Shorthand	Valid/ Invalid	Trigger Mode	Delivery Mode	Destination Mode

NOTES:
1. The ability of a processor to send a lowest priority IPI is model specific.
2. Treated as edge triggered if level bit is set to 1, otherwise ignored.
3. Treated as edge triggered when Level bit is set to 1; treated as "INIT Level Deassert" message when level bit is set to 0 (deassert). Only INIT level deassert messages are allowed to have the level bit set to 0. For all other messages the level bit must be set to 1.
4. X means the setting is ignored.
5. The behavior of the APIC is undefined.

16. Register reference corrected

In Section 6.3.4, "Saving Procedure State Information," of the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, a register name has been corrected. See the bold type and the change bar.

6.3.4 Saving Procedure State Information

The processor does not save the contents of the general-purpose registers, segment registers, or the EFLAGS register on a procedure call. A calling procedure should explicitly save the values in any of the general-purpose registers that it will need when it resumes execution after a return. These values can be saved on the stack or in memory in one of the data segments.

The PUSHA and POPA instructions facilitate saving and restoring the contents of the general-purpose registers. PUSHA pushes the values in all the general-purpose registers on the stack in the following order: EAX, ECX, EDX, EBX, ESP (the value prior to executing the PUSHA instruction), EBP, ESI, and EDI. The POPA instruction pops all the register values saved with a PUSHA instruction (except the **ESP** value) from the stack to their respective registers.

If a called procedure changes the state of any of the segment registers explicitly, it should restore them to their former values before executing a return to the calling procedure.

If a calling procedure needs to maintain the state of the EFLAGS register, it can save and restore all or part of the register using the PUSHF/PUSHFD and POPF/POPFD instructions. The PUSHF instruction pushes the lower word of the EFLAGS register on the stack, while the PUSHFD instruction pushes the entire register. The POPF instruction pops a word from the stack into the lower word of the EFLAGS register, while the POPFD instruction pops a double word from the stack into the register.

17. Appendix A updated to include 3-byte opcode information

In Appendix A, "Opcode Map," of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 2B*, has been updated to include 3-byte opcodes. The appendix is reproduced below. See the change bars.

APPENDIX A
OPCODE MAP

Use the opcode tables in this chapter to interpret IA-32 architecture object code. Instructions are divided into encoding groups:

- 1-byte, 2-byte and 3-byte opcode encodings are used to encode integer, system, MMX technology, SSE/SSE2/SSE3, and VMX instructions. Maps for these instructions are given in Table A-2 through Table A-6.
- Escape opcodes (in the format: ESC character, opcode, ModR/M byte) are used for floating-point instructions. The maps for these instructions are provided in Table A-7 through Table A-22.

NOTE

All blanks in opcode maps are reserved and must not be used. Do not depend on the operation of undefined or blank opcodes.

A.1 USING OPCODE TABLES

Tables in this appendix list opcodes of instructions (including required instruction prefixes, opcode extensions in associated ModR/M byte). Blank cells in the tables indicate opcodes that are reserved or undefined.

The opcode map tables are organized by hex values of the upper and lower 4 bits of an opcode byte. For 1-byte encodings (Table A-2), use the four high-order bits of an opcode to index a row of the opcode table; use the four low-order bits to index a column of the table. For 2-byte opcodes beginning with 0FH (Table A-3), skip any instruction prefixes, the 0FH byte (0FH may be preceded by 66H, F2H, or F3H) and use the upper and lower 4-bit values of the next opcode byte to index table rows and columns. Similarly, for 3-byte opcodes beginning with 0F38H or 0F3AH (Table A-4), skip any instruction prefixes, 0F38H or 0F3AH and use the upper and lower 4-bit values of the third opcode byte to index table rows and columns. See Section A.2.4, “Opcode Look-up Examples for One, Two, and Three-Byte Opcodes.”

When a ModR/M byte provides opcode extensions, this information qualifies opcode execution. For information on how an opcode extension in the ModR/M byte modifies the opcode map in Table A-2 and Table A-3, see Section A.4.

The escape (ESC) opcode tables for floating point instructions identify the eight high order bits of opcodes at the top of each page. See Section A.5. If the accompanying ModR/M byte is in the range of 00H-BFH, bits 3-5 (the top row of the third table on each page) along with the reg bits of ModR/M determine the opcode. ModR/M bytes outside the range of 00H-BFH are mapped by the bottom two tables on each page of the section.

A.2 KEY TO ABBREVIATIONS

Operands are identified by a two-character code of the form Zz. The first character, an uppercase letter, specifies the addressing method; the second character, a lowercase letter, specifies the type of operand.

A.2.1 Codes for Addressing Method

The following abbreviations are used to document addressing methods:

A Direct address: the instruction has no ModR/M byte; the address of the operand is encoded in the instruction. No base register, index register, or scaling factor can be applied (for example, far JMP (EA)).

C The reg field of the ModR/M byte selects a control register (for example, MOV (0F20, 0F22)).

D The reg field of the ModR/M byte selects a debug register (for example, MOV (0F21,0F23)).

E A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.

F EFLAGS/RFLAGS Register.

G The reg field of the ModR/M byte selects a general register (for example, AX (000)).

I Immediate data: the operand value is encoded in subsequent bytes of the instruction.

J The instruction contains a relative offset to be added to the instruction pointer register (for example, JMP (0E9), LOOP).

M The ModR/M byte may refer only to memory (for example, BOUND, LES, LDS, LSS, LFS, LGS, CMPXCHG8B).

N The R/M field of the ModR/M byte selects a packed-quadword, MMX technology register.

O The instruction has no ModR/M byte. The offset of the operand is coded as a word or double word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied (for example, MOV (A0–A3)).

P The reg field of the ModR/M byte selects a packed quadword MMX technology register.

Q A ModR/M byte follows the opcode and specifies the operand. The operand is either an MMX technology register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.

R The R/M field of the ModR/M byte may refer only to a general register (for example, MOV (0F20-0F23)).

S The reg field of the ModR/M byte selects a segment register (for example, MOV (8C,8E)).

U The R/M field of the ModR/M byte selects a 128-bit XMM register.

V The reg field of the ModR/M byte selects a 128-bit XMM register.

W A ModR/M byte follows the opcode and specifies the operand. The operand is either a 128-bit XMM register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.

X Memory addressed by the DS:rSI register pair (for example, MOVS, CMPS, OUTS, or LODS).

Y Memory addressed by the ES:rDI register pair (for example, MOVS, CMPS, INS, STOS, or SCAS).

A.2.2 Codes for Operand Type

The following abbreviations are used to document operand types:

a Two one-word operands in memory or two double-word operands in memory, depending on operand-size attribute (used only by the BOUND instruction).

b Byte, regardless of operand-size attribute.

c Byte or word, depending on operand-size attribute.

d Doubleword, regardless of operand-size attribute.

dq Double-quadword, regardless of operand-size attribute.

p 32-bit or 48-bit pointer, depending on operand-size attribute.

pi Quadword MMX technology register (for example: mm0).

ps 128-bit packed single-precision floating-point data.

q Quadword, regardless of operand-size attribute.

s 6-byte or 10-byte pseudo-descriptor.

ss Scalar element of a 128-bit packed single-precision floating data.

si Doubleword integer register (for example: eax).

v Word, doubleword or quadword (in 64-bit mode), depending on operand-size attribute.

w Word, regardless of operand-size attribute.

z Word for 16-bit operand-size or doubleword for 32 or 64-bit operand-size.

A.2.3 Register Codes

When an opcode requires a specific register as an operand, the register is identified by name (for example, AX, CL, or ESI). The name indicates whether the register is 64, 32, 16, or 8 bits wide.

A register identifier of the form eXX or rXX is used when register width depends on the operand-size attribute. eXX is used when 16 or 32-bit sizes are possible; rXX is used when 16, 32, or 64-bit sizes are possible. For example: eAX indicates that the AX register is used when the operand-size attribute is 16 and the EAX register is used when the operand-size attribute is 32. rAX can indicate AX, EAX or RAX.

When the REX.B bit is used to modify the register specified in the reg field of the opcode, this fact is indicated by adding “/x” to the register name to indicate the additional possibility. For example, rCX/r9 is used to indicate that the register could either be rCX or r9. Note that the size of r9 in this case is determined by the operand size attribute (just as for rCX).

A.2.4 Opcode Look-up Examples for One, Two, and Three-Byte Opcodes

This section provides examples that demonstrate how opcode maps are used.

A.2.4.1 One-Byte Opcode Instructions

The opcode map for 1-byte opcodes is shown in Table A-2. The opcode map for 1-byte opcodes is arranged by row (the least-significant 4 bits of the hexadecimal value) and column (the most-

significant 4 bits of the hexadecimal value). Each entry in the table lists one of the following types of opcodes:

- Instruction mnemonics and operand types using the notations listed in Section A.2
- Opcodes used as an instruction prefix

For each entry in the opcode map that corresponds to an instruction, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, "Instruction Format," of the *IA-32 Intel Architecture Software Developer's Manual, Volume 2A*. Operand types are listed according to notations listed in Section A.2.
- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-6 when interpreting the ModR/M byte.
- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction prefix or entries for instructions without operands that use ModR/M (for example: 60H, PUSHA; 06H, PUSH ES).

Example A-1. Look-up Example for 1-Byte Opcodes

Opcode 030500000000H for an ADD instruction is interpreted using the 1-byte opcode map (Table A-2) as follows:

- The first digit (0) of the opcode indicates the table row and the second digit (3) indicates the table column. This locates an opcode for ADD with two operands.
- The first operand (type Gv) indicates a general register that is a word or doubleword depending on the operand-size attribute. The second operand (type Ev) indicates a ModR/M byte follows that specifies whether the operand is a word or doubleword general-purpose register or a memory address.
- The ModR/M byte for this instruction is 05H, indicating that a 32-bit displacement follows (00000000H). The reg/opcode portion of the ModR/M byte (bits 3-5) is 000, indicating the EAX register.

The instruction for this opcode is ADD EAX, mem_op, and the offset of mem_op is 00000000H.

Some 1- and 2-byte opcodes point to group numbers (shaded entries in the opcode map table). Group numbers indicate that the instruction uses the reg/opcode bits in the ModR/M byte as an opcode extension (refer to Section A.4).

A.2.4.2 Two-Byte Opcode Instructions

The two-byte opcode map shown in Table A-3 includes primary opcodes that are either two bytes or three bytes in length. Primary opcodes that are 2 bytes in length begin with an escape opcode 0FH. The upper and lower four bits of the second opcode byte are used to index a particular row and column in Table A-3.

Two-byte opcodes that are 3 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and the escape opcode (OFH). The upper and lower four bits of the third byte are used to index a particular row and column in Table A-3 (except when the second opcode byte is the 3-byte escape opcodes 38H or 3AH; in this situation refer to Section A.2.4.3).

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into one of the following cases:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, "Instruction Format," of the *IA-32 Intel Architecture Software Developer's Manual, Volume 2A*. The operand types are listed according to notations listed in Section A.2.
- A ModR/M byte is required and includes an opcode extension in the reg field in the ModR/M byte. Use Table A-6 when interpreting the ModR/M byte.
- Use of the ModR/M byte is reserved or undefined. This applies to entries that represent an instruction without operands that are encoded using ModR/M (for example: 0F77H, EMMS).

Example A-2. Look-up Example for 2-Byte Opcodes

Look-up opcode 0FA4050000000003H for a SHLD instruction using Table A-3.

- The opcode is located in row A, column 4. The location indicates a SHLD instruction with operands Ev, Gv, and Ib. Interpret the operands as follows:
  - — Ev: The ModR/M byte follows the opcode to specify a word or doubleword operand.
  - — Gv: The reg field of the ModR/M byte selects a general-purpose register.
  - — Ib: Immediate data is encoded in the subsequent byte of the instruction.
- The third byte is the ModR/M byte (05H). The mod and opcode/reg fields of ModR/M indicate that a 32-bit displacement is used to locate the first operand in memory and eAX as the second operand.
- The next part of the opcode is the 32-bit displacement for the destination memory operand (00000000H). The last byte stores immediate byte that provides the count of the shift (03H).
- By this breakdown, it has been shown that this opcode represents the instruction: SHLD DS:00000000H, EAX, 3.

A.2.4.3 Three-Byte Opcode Instructions

The three-byte opcode maps shown in Table A-4 and Table A-5 includes primary opcodes that are either 3 or 4 bytes in length. Primary opcodes that are 3 bytes in length begin with two escape bytes 0F38H or 0F3A. The upper and lower four bits of the third opcode byte are used to index a particular row and column in Table A-4 or Table A-5.

Three-byte opcodes that are 4 bytes in length begin with a mandatory prefix (66H, F2H, or F3H) and two escape bytes (0F38H or 0F3AH). The upper and lower four bits of the fourth byte are used to index a particular row and column in Table A-4 or Table A-5.

For each entry in the opcode map, the rules for interpreting the byte following the primary opcode fall into the following case:

- A ModR/M byte is required and is interpreted according to the abbreviations listed in Section A.1 and Chapter 2, "Instruction Format," of the *IA-32 Intel Architecture Software Developer's Manual, Volume 2A*. The operand types are listed according to notations listed in Section A.2.

Example A-3. Look-up Example for 3-Byte Opcodes

Look-up opcode 660F3A0FC108H for a PALIGNR instruction using Table A-5.

- 66H is a prefix and 0F3AH indicate to use Table A-5. The opcode is located in row 0, column F indicating a PALIGNR instruction with operands Vdq, Wdq, and Ib. Interpret the operands as follows:
  - — Vdq: The reg field of the ModR/M byte selects a 128-bit XMM register.

— Wdq: The R/M field of the ModR/M byte selects either a 128-bit XMM register or memory location.

— Ib: Immediate data is encoded in the subsequent byte of the instruction.

- The next byte is the ModR/M byte (C1H). The reg field indicates that the first operand is XMM0. The mod shows that the R/M field specifies a register and the R/M indicates that the second operand is XMM1.
- The last byte is the immediate byte (08H).
- By this breakdown, it has been shown that this opcode represents the instruction: PALIGNR XMM0, XMM1, 8.

A.2.5 Superscripts Utilized in Opcode Tables

Table A-1 contains notes on particular encodings. These notes are indicated in the following opcode maps by superscripts. Gray cells indicate instruction groupings.

Table A-1. Superscripts Utilized in Opcode Tables

Superscript Symbol	Meaning of Symbol
1A	Bits 5, 4, and 3 of ModR/M byte used as an opcode extension (refer to Section A.4, “Opcode Extensions For One-Byte And Two-byte Opcodes”).
1B	Use the 0F0B opcode (UD2 instruction) or the 0FB9H opcode when deliberately trying to generate an invalid opcode exception (#UD).
1C	Some instructions added in the Pentium III processor may use the same two-byte opcode. If the instruction has variations, or the opcode represents different instructions, the ModR/M byte will be used to differentiate the instruction. For the value of the ModR/M byte needed to decode the instruction, see Table A-6. These instructions include SFENCE, STMXCSR, LDMXCSR, FXRSTOR, and FXSAVE, as well as PREFETCH and its variations.
i64	The instruction is invalid or not encodable in 64-bit mode. 40 through 4F (single-byte INC and DEC) are REX prefix combinations when in 64-bit mode (use FE/FF Grp 4 and 5 for INC and DEC).
o64	Instruction is only available when in 64-bit mode.
d64	When in 64-bit mode, instruction defaults to 64-bit operand size and cannot encode 32-bit operand size.
f64	The operand size is forced to a 64-bit operand size when in 64-bit mode (prefixes that change operand size are ignored for this instruction in 64-bit mode).

A.3 ONE, TWO, AND THREE-BYTE OPCODE MAPS

See Table A-2 through Table A-5 below. The tables are multiple page presentations. Rows and columns with sequential relationships are placed on facing pages to make look-up tasks easier. Note that table footnotes are not presented on each page. Table footnotes for each table are presented on the last page of the table.

Table A-2. One-byte Opcode Map: (00H — F7H)*

	0	1	2	3	4	5	6	7
0	ADD						PUSH ES[i64]	POP ES[i64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
1	ADC						PUSH SS[i64]	POP SS[i64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
2	AND						SEG=ES (Prefix)	DAA[i64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
3	XOR						SEG=SS (Prefix)	AAA[i64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
4	INC[i64] general register / REX[o64] Prefixes							
	eAX REX	eCX REX.B	eDX REX.X	eBX REX.XB	eSP REX.R	eBP REX.RB	eSI REX.RX	eDI REX.RXB
5	PUSH[d64] general register							
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA[i64]/ PUSHAD[i64]	POPA[i64]/ POPAD[i64]	BOUND[i64] Gv, Ma	ARPL[i64] Ew, Gw MOVSXD[o64] Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc[f64], Jb - Short-displacement jump on condition							
	O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1[1A]				TEST		XCHG	
	Eb, Ib	Ev, Iz	Eb, Ib[i64]	Ev, Ib	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv
9	NOP PAUSE(F3) XCHG r8, rAX	XCHG word, double-word or quad-word register with rAX						
		rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
A	MOV				MOVS/B Xb, Yb	MOVS/W/D/Q Xv, Yv	CMPS/B Xb, Yb	CMPS/W/D Xv, Yv
	AL, Ob	rAX, Ov	Ob, AL	Ov, rAX				
B	MOV immediate byte into byte register							
	AL/R8L, Ib	CL/R9L, Ib	DL/R10L, Ib	BL/R11L, Ib	AH/R12L, Ib	CH/R13L, Ib	DH/R14L, Ib	BH/R15L, Ib
C	Shift Grp 2[1A]		RETN[f64] Iw	RETN[f64]	LES[i64] Gz, Mp	LDS[i64] Gz, Mp	Grp 11[1A] - MOV	
	Eb, Ib	Ev, Ib					Eb, Ib	Ev, Iz
D	Shift Grp 2[1A]				AAM[i64] Ib	AAD[i64] Ib		XLAT/ XLATB
	Eb, 1	Ev, 1	Eb, CL	Ev, CL				
E	LOOPNE[f64]/ LOOPNZ[f64] Jb	LOOPE[f64]/ LOOPZ[f64] Jb	LOOP[f64] Jb	JrCXZ[f64]/ Jb	IN		OUT	
					AL, Ib	eAX, Ib	Ib, AL	Ib, eAX
F	LOCK (Prefix)		REPNE (Prefix)	REP/ REPE (Prefix)	HLT	CMC	Unary Grp 3[1A]	
							Eb	Ev

Table A-2. One-byte Opcode Map: (08H — FFH)*

	8	9	A	B	C	D	E	F
0	OR						PUSH CS[i64]	2-byte escape (Table A-3)
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
1	SBB						PUSH DS[i64]	POP DS[i64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
2	SUB						SEG=CS (Prefix)	DAS[i64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
3	CMP						SEG=DS (Prefix)	AAS[i64]
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
4	DEC[i64] general register / REX[o64] Prefixes							
	eAX REX.W	eCX REX.WB	eDX REX.WX	eBX REX.WXB	eSP REX.WR	eBP REX.WRB	eSI REX.WRX	eDI REX.WRXB
5	POP[d64] into general register							
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSH[d64] Iz	IMUL Gv, Ev, Iz	PUSH[d64] Ib	IMUL Gv, Ev, Ib	INS/ INSB Yb, DX	INS/ INSW/ INSD Yz, DX	OUTS/ OUTSB DX, Xb	OUTS/ OUTSW/ OUTSD DX, Xz
7	Jcc[f64], Jb- Short displacement jump on condition							
	S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
8	MOV				MOV Ev, Sw	LEA Gv, M	MOV Sw, Ew	Grp 1A[1A] POP[d64] Ev
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev				
9	CBW/ CWDE/ CDQE	CWD/ CDQ/ CQO	CALLF[i64] Ap	FWAIT/ WAIT	PUSHF/D/Q [d64]/ Fv	POPF/D/Q [d64]/ Fv	SAHF	LAHF
A	TEST		STOS/B Yb, AL	STOS/W/D/Q Yv, rAX	LODS/B AL, Xb	LODS/W/D/Q rAX, Xv	SCAS/B AL, Yb	SCAS/W/D/Q rAX, Xv
	AL, Ib	rAX, Iz						
B	MOV immediate word or double into word, double, or quad register							
	rAX/r8, Iv	rCX/r9, Iv	rDX/r10, Iv	rBX/r11, Iv	rSP/r12, Iv	rBP/r13, Iv	rSI/r14, Iv	rDI/r15, Iv
C	ENTER Iw, Ib	LEAVE[d64]	RETF Iw	RETF	INT 3	INT Ib	INTO[i64]	IRET/D/Q
D	ESC (Escape to coprocessor instruction set)							
E	CALL[f64]	JMP			IN		OUT	
	Jz	near[f64] Jz	far[i64] AP	short[f64] Jb	AL, DX	eAX, DX	DX, AL	DX, eAX
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp 4[1A]	INC/DEC Grp 5[1A]

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-3. Two-byte Opcode Map: 00H — 77H (First Byte is 0FH)*

	0	1	2	3	4	5	6	7
0	Grp 6[1A]	Grp 7[1A]	LAR Gv, Ew	LSL Gv, Ew		SYSCALL[o64]	CLTS	SYSRET[o64]
1	movups Vps, Wps movss (F3) Vss, Wss movupd (66) Vpd, Wpd movsd (F2) Vsd, Wsd	movups Wps, Vps movss (F3) Wss, Vss movupd (66) Wpd, Vpd movsd (F2) Vsd, Wsd	movlps Vq, Mq movlpd (66) Vq, Mq movhlps Vq, Uq movddup(F2) Vq, Wq movsldup(F3) Vq, Wq	movlps Mq, Vq movlpd (66) Mq, Vq	unpcklps Vps, Wq unpcklpd (66) Vpd, Wq	unpckhps Vps, Wq unpckhpd (66) Vpd, Wq	movhps Vq, Mq movhpd (66) Vq, Mq movlhps Vq, Uq movshdup(F3) Vq, Wq	movhps Mq, Vq movhpd(66) Mq, Vq
2	MOV Rd, Cd	MOV Rd, Dd	MOV Cd, Rd	MOV Dd, Rd				
3	WRMSR	RDTSC	RDMSR	RDPMC	SYSENTER	SYSEXIT		
4	CMOVcc, (Gv, Ev) - Conditional Move							
	O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
5	movmskps Gd, Ups movmskpd (66) Gd, Upd	sqrtps Vps, Wps sqrtss (F3) Vss, Wss sqrtpd (66) Vpd, Wpd sqrtsd (F2) Vsd, Wsd	rsqrtps Vps, Wps rsqrtss (F3) Vss, Wss	rcpps Vps, Wps rcpss (F3) Vss, Wss	andps Vps, Wps andpd (66) Vpd, Wpd	andnps Vps, Wps andnpd (66) Vpd, Wpd	orps Vps, Wps orpd (66) Vpd, Wpd	xorps Vps, Wps xorpd (66) Vpd, Wpd
6	punpcklbw Pq, Qd punpcklbw (66) Vdq, Wdq	punpcklwd Pq, Qd punpcklwd (66) Vdq, Wdq	punpckldq Pq, Qd punpckldq (66) Vdq, Wdq	packsswb Pq, Qq packsswb (66) Vdq, Wdq	pcmpgtb Pq, Qq pcmpgtb (66) Vdq, Wdq	pcmpgtw Pq, Qq pcmpgtw (66) Vdq, Wdq	pcmpgtd Pq, Qq pcmpgtd (66) Vdq, Wdq	packuswb Pq, Qq packuswb (66) Vdq, Wdq
7	pshufw Pq, Qq, Ib pshufd (66) Vdq,Wdq,Ib pshufhw(F3) Vdq,Wdq,Ib pshuflw (F2) Vdq Wdq,Ib	(Grp 12[1A])	(Grp 13[1A])	(Grp 14[1A])	pcmpeqb Pq, Qq pcmpeqb (66) Vdq, Wdq	pcmpeqw Pq, Qq pcmpeqw (66) Vdq, Wdq	pcmpeqd Pq, Qq pcmpeqd (66) Vdq, Wdq	emms

Table A-3. Two-byte Opcode Map: 08H — 7FH (First Byte is 0FH)*

	8	9	A	B	C	D	E	F
0	INVD	WBINVD		2-byte Illegal Opcodes UD2[1B]		NOP Ev		
1	Prefetch[1C] (Grp 16[1A])							NOP Ev
2	movaps Vps, Wps movapd (66) Vpd, Wpd	movaps Wps, Vps movapd (66) Wpd, Vpd	cvtpi2ps Vps, Qq cvtsi2ss (F3) Vss, Ed/q cvtpi2pd (66) Vpd, Qq cvtsi2sd (F2) Vsd, Ed/q	movntps Mps, Vps movntpd (66) Mpd, Vpd	cvttps2pi Qq, Wps cvttss2si (F3) Gd, Wss cvttpd2pi (66) Qdq, Wpd cvttsd2si (F2) Gd, Wsd	cvtps2pi Qq, Wps cvtss2si (F3) Gd/q, Wss cvtpd2pi (66) Qdq, Wpd cvtsd2si (F2) Gd/q, Wsd	ucomiss Vss, Wss ucomisd (66) Vsd, Wsd	comiss Vps, Wps comisd (66) Vsd, Wsd
3	3-byte escape (Table A-4)		3-byte escape (Table A-5)					
4	CMOVcc(Gv, Ev) - Conditional Move							
	S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
5	addps Vps, Wps addss (F3) Vss, Wss addpd (66) Vpd, Wpd addsd (F2) Vsd, Wsd	mulps Vps, Wps mulss (F3) Vss, Wss mulpd (66) Vpd, Wpd mulsd (F2) Vsd, Wsd	cvtps2pd Vpd, Wps cvtss2sd (F3) Vss, Wss cvtpd2ps (66) Vps, Wpd cvtsd2ss (F2) Vsd, Wsd	cvtdq2ps Vps, Wdq cvtps2dq (66) Vdq, Wps cvttps2dq (F3) Vdq, Wps	subps Vps, Wps subss (F3) Vss, Wss subpd (66) Vpd, Wpd subsd (F2) Vsd, Wsd	minps Vps, Wps minss (F3) Vss, Wss minpd (66) Vpd, Wpd minsd (F2) Vsd, Wsd	divps Vps, Wps divss (F3) Vss, Wss divpd (66) Vpd, Wpd divsd (F2) Vsd, Wsd	maxps Vps, Wps maxss (F3) Vss, Wss maxpd (66) Vpd, Wpd maxsd (F2) Vsd, Wsd
6	punpckhbw Pq, Qd punpckhbw (66) Pdq, Qdq	punpckhwd Pq, Qd punpckhwd (66) Pdq, Qdq	punpckhdq Pq, Qd punpckhdq (66) Pdq, Qdq	packssdw Pq, Qd packssdw (66) Pdq, Qdq	punpcklqdq (66) Vdq, Wdq	punpckhqdq (66) Vdq, Wdq	movd/q/ Pd, Ed/q movd/q (66) Vdq, Ed/q	movq Pq, Qq movdqa (66) Vdq, Wdq movdqu (F3) Vdq, Wdq
7	VMREAD Ed/q, Gd/q	VMWRITE Gd/q, Ed/q			haddps(F2) Vps, Wps haddpd(66) Vpd, Wpd	hsubps(F2) Vps, Wps hsubpd(66) Vpd, Wpd	movd/q Ed/q, Pd movd/q (66) Ed/q, Vdq movq (F3) Vq, Wq	movq Qq, Pq movdqa (66) Wdq, Vdq movdqu (F3) Wdq, Vdq

Table A-3. Two-byte Opcode Map: 80H — F7H (First Byte is 0FH)*

	0	1	2	3	4	5	6	7
8	Jcc[f64], Jz - Long-displacement jump on condition							
	O	NO	B/CNAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
9	SETcc, Eb - Byte Set on condition							
	O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
A	PUSH[d64] FS	POP[d64] FS	CPUID	BT Ev, Gv	SHLD Ev, Gv, Ib	SHLD Ev, Gv, CL		
B	CMPXCHG		LSS Gv, Mp	BTR Ev, Gv	LFS Gv, Mp	LGS Gv, Mp	MOVZX	
	Eb, Gb	Ev, Gv					Gv, Eb	Gv, Ew
C	XADD Eb, Gb	XADD Ev, Gv	cmpps Vps, Wps, Ib cmpss (F3) Vss, Wss, Ib cmppd (66) Vpd, Wpd, Ib cmpsd (F2) Vsd, Wsd, Ib	movnti Md/q, Gd/q	pinsrw Pq, Ew, Ib pinsrw (66) Vdq, Ew, Ib	pextrw Gd, Nq, Ib pextrw (66) Gd, Udq, Ib	shufps Vps, Wps, Ib shufpd (66) Vpd, Wpd, Ib	Grp 9[1A]
D	addsubps(F2) Vps, Wps addsubpd(66) Vpd, Wpd	psrlw Pq, Qq psrlw (66) Vdq, Wdq	psrld Pq, Qq psrld (66) Vdq, Wdq	psrlq Pq, Qq psrlq (66) Vdq, Wdq	paddq Pq, Qq paddq (66) Vdq, Wdq	pmullw Pq, Qq pmullw (66) Vdq, Wdq	movq (66) Wq, Vq movq2dq (F3) Vdq, Nq movdq2q (F2) Pq, Uq	pmovmskb Gd, Nq pmovmksb (66) Gd, Udq
E	pavgb Pq, Qq pavgb (66) Vdq, Wdq	psraw Pq, Qq psraw (66) Vdq, Wdq	psrad Pq, Qq psrad (66) Vdq, Wdq	pavgw Pq, Qq pavgw (66) Vdq, Wdq	pmulhuw Pq, Qq pmulhuw (66) Vdq, Wdq	pmulhw Pq, Qq pmulhw (66) Vdq, Wdq	cvtpd2dq (F2) Vdq, Wpd cvttpd2dq (66) Vdq, Wpd cvtdq2pd (F3) Vpd, Wdq	movntq Mq, Pq movntdq (66) Mdq, Vdq
F	lddqu (F2) Vdq, Mdq	psllw Pq, Qq psllw (66) Vdq, Wdq	pslld Pq, Qq pslld (66) Vdq, Wdq	psllq Pq, Qq psllq (66) Vdq, Wdq	pmuludq Pq, Qq pmuludq (66) Vdq, Wdq	pmaddwd Pq, Qq pmaddwd (66) Vdq, Wdq	psadbw Pq, Qq psadbw (66) Vdq, Wdq	maskmovq Pq, Nq maskmovdqu (66) Vdq, Udq

Table A-3. Two-byte Opcode Map: 88H — FFH (First Byte is 0FH)*

	8	9	A	B	C	D	E	F
8	Jcc[f64], Jz - Long-displacement jump on condition							
	S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
9	SETcc, Eb - Byte Set on condition							
	S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
A	PUSH[d64] GS	POP[d64] GS	RSM	BTS Ev, Gv	SHRD Ev, Gv, Ib	SHRD Ev, Gv, CL	(Grp 15[1A])[1C]	IMUL Gv, Ev
B		Grp 10[1A] Invalid Opcode[1B]	Grp 8[1A] Ev, Ib	BTC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOVSX Gv, Eb	MOVSX Gv, Ew
C	BSWAP RAX/EAX/ R8/R8D	RCX/ECX/ R9/R9D	RDX/EDX/ R10/R10D	RBX/EBX/ R11/R11D	RSP/ESP/ R12/R12D	RBP/EBP/ R13/R13D	RSI/ESI/ R14/R14D	RDI/EDI/ R15/R15D
D	psubusb Pq, Qq psubusb (66) Vdq, Wdq	psubusw Pq, Qq psubusw (66) Vdq, Wdq	pminub Pq, Qq pminub (66) Vdq, Wdq	pand Pq, Qq pand (66) Vdq, Wdq	paddusb Pq, Qq paddusb (66) Vdq, Wdq	paddusw Pq, Qq paddusw (66) Vdq, Wdq	pmaxub Pq, Qq pmaxub (66) Vdq, Wdq	pandn Pq, Qq pandn (66) Vdq, Wdq
E	psubsb Pq, Qq psubsb (66) Vdq, Wdq	psubsw Pq, Qq psubsw (66) Vdq, Wdq	pminsw Pq, Qq pminsw (66) Vdq, Wdq	por Pq, Qq por (66) Vdq, Wdq	paddsb Pq, Qq paddsb (66) Vdq, Wdq	paddsw Pq, Qq paddsw (66) Vdq, Wdq	pmaxsw Pq, Qq pmaxsw (66) Vdq, Wdq	pxor Pq, Qq pxor (66) Vdq, Wdq
F	psubb Pq, Qq psubb (66) Vdq, Wdq	psubw Pq, Qq psubw (66) Vdq, Wdq	psubd Pq, Qq psubd (66) Vdq, Wdq	psubq Pq, Qq psubq (66) Vdq, Wdq	paddb Pq, Qq paddb (66) Vdq, Wdq	paddw Pq, Qq paddw (66) Vdq, Wdq	paddd Pq, Qq paddd (66) Vdq, Wdq	

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-4. Three-byte Opcode Map: 00H — F7H (First Two Bytes are 0F 38H)*

	0	1	2	3	4	5	6	7
0	pshufb Pq, Qq pshufb (66) Vdq, Wdq	phaddw Pq, Qq phaddw (66) Vdq, Wdq	phaddd Pq, Qq phaddd (66) Vdq, Wdq	phaddsw Pq, Qq phaddsw (66) Vdq, Wdq	pmaddubsw Pq, Qq pmaddubsw (66) Vdq, Wdq	phsubw Pq, Qq phsubw (66) Vdq, Wdq	phsubd Pq, Qq phsubd (66) Vdq, Wdq	phsubsw Pq, Qq phsubsw (66) Vdq, Wdq
1								
2								
3								
4								
5								
6								
7								
8								
9								
A								
B								
C								
D								
E								
F								

Table A-4. Three-byte Opcode Map: 08H — FFH (First Two Bytes are 0F 38H)*

	8	9	A	B	C	D	E	F
0	psignb Pq, Qq psignb (66) Vdq, Wdq	psignw Pq, Qq psignw (66) Vdq, Wdq	psignd Pq, Qq psignd (66) Vdq, Wdq	pmulhrsw Pq, Qq pmulhrsw (66) Vdq, Wdq				
1					pabsb Pq, Qq pabsb (66) Vdq, Wdq	pabsw Pq, Qq pabsw (66) Vdq, Wdq	pabsd Pq, Qq pabsd (66) Vdq, Wdq	
2								
3								
4								
5								
6								
7								
8								
9								
A								
B								
C								
D								
E								
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-5. Three-byte Opcode Map: 00H — F7H (First two bytes are 0F 3AH)*

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								
A								
B								
C								
D								
E								
F								

Table A-5. Three-byte Opcode Map: 08H — FFH (First Two Bytes are 0F 3AH)*

	8	9	A	B	C	D	E	F
0								palignr Pq, Qq, Ib palignr(66) Vdq, Wdq, Ib
1								
2								
3								
4								
5								
6								
7								
8								
9								
A								
B								
C								
D								
E								
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.4 OPCODE EXTENSIONS FOR ONE-BYTE AND TWO-BYTE OPCODES

Some 1-byte and 2-byte opcodes use bits 3-5 of the ModR/M byte (the nnn field in Figure A-1) as an extension of the opcode.

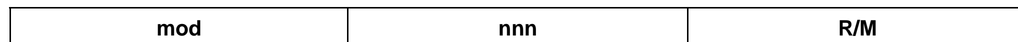

Figure A-1. ModR/M Byte nnn Field (Bits 5, 4, and 3)

Opcodes that have opcode extensions are indicated in Table A-6 and organized by group number. Group numbers (from 1 to 16, second column) provide a table entry point. The encoding for the r/m field for each instruction can be established using the third column of the table.

A.4.1 Opcode Look-up Examples Using Opcode Extensions

An Example is provided below.

Example A-3. Interpreting an ADD Instruction

An ADD instruction with a 1-byte opcode of 80H is a Group 1 instruction:

- Table A-6 indicates that the opcode extension field encoded in the ModR/M byte for this instruction is 000B.
- The r/m field can be encoded to access a register (11B) or a memory address using a specified addressing mode (for example: mem = 00B, 01B, 10B).

Example A-4. Looking Up 0F01C3H

Look up opcode 0F01C3 for a VMRESUME instruction by using Table A-2, Table A-3 and Table A-6:

- 0F tells us that this instruction is in the 2-byte opcode map.
- 01 (row 0, column 1 in Table A-3) reveals that this opcode is in Group 7 of Table A-6.
- C3 is the ModR/M byte. The first two bits of C3 are 11B. This tells us to look at the second of the Group 7 rows in Table A-6.
- The Op/Reg bits [5,4,3] are 000B. This tells us to look in the 000 column for Group 7.
- Finally, the R/M bits [2,1,0] are 011B. This identifies the opcode as the VMRESUME instruction.

A.4.2 Opcode Extension Tables

See Table A-6 below.

Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number*

Opcode	Group	Mod 7,6	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
			000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
80-83	1	mem, 11B	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
8F	1A	mem, 11B	POP							
C0, C1 reg, imm D0, D1 reg, 1 D2, D3 reg, CL	2	mem, 11B	ROL	ROR	RCL	RCR	SHL/SAL	SHR		SAR
F6, F7	3	mem, 11B	TEST Ib/Iz		NOT	NEG	MUL AL/rAX	IMUL AL/rAX	DIV AL/rAX	IDIV AL/rAX
FE	4	mem, 11B	INC Eb	DEC Eb						
FF	5	mem, 11B	INC Ev	DEC Ev	CALLN^{f64} Ev	CALLF Ep	JMPN^{f64} Ev	JMPF Ep	PUSH^{d64} Ev	
0F 00	6	mem, 11B	SLDT Rv/Mw	STR Rv/Mw	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
0F 01	7	mem	SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Mw/Rv		LMSW Ew	INVLPG Mb
		11B	VMCALL (001) VMLAUNCH (010) VMRESUME (011) VMXOFF (100)	MONITOR (000) MWAIT (001)						SWAPGS ^{o64}(000)
0F BA	8	mem, 11B					BT	BTS	BTR	BTC
0F C7	9	mem		CMPXCH8B Mq CMPXCHG16B Mdq					VMPTRLD Mq VMCLEAR (66) Mq VMXON (F3) Mq	VMPTRST Mq
		11B								
0F B9	10	mem								
		11B								
C6	11	mem, 11B	MOV Eb, Ib							
C7		mem	MOV Ev, Iz							
		11B								

Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number* (Continued)

Opcode	Group	Mod 7,6	Encoding of Bits 5,4,3 of the ModR/M Byte (bits 2,1,0 in parenthesis)							
			000	001	010	011	100	101	110	111
0F 71	12	mem								
		11B			psrlw Nq, Ib psrlw (66) Udq, Ib		psraw Nq, Ib psraw (66) Udq, Ib		psllw Nq, Ib psllw (66) Udq, Ib	
0F 72	13	mem								
		11B			psrld Nq, Ib psrld (66) Udq, Ib		psrad Nq, Ib psrad (66) Udq, Ib		pslld Nq, Ib pslld (66) Udq, Ib	
0F 73	14	mem								
		11B			psrlq Nq, Ib psrlq (66) Udq, Ib	psrldq (66) Udq, Ib			psllq Nq, Ib psllq (66) Udq, Ib	pslldq (66) Udq, Ib
0F AE	15	mem	fxsave	fxrstor	ldmxcsr	stmxcsr				clflush
		11B						lfence	mfence	sfence
0F 18	16	mem	prefetch NTA	prefetch T0	prefetch T1	prefetch T2				
		11B								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5 ESCAPE OPCODE INSTRUCTIONS

Opcode maps for coprocessor escape instruction opcodes (x87 floating-point instruction opcodes) are in Table A-7 through Table A-22. These maps are grouped by the first byte of the opcode, from D8-DF. Each of these opcodes has a ModR/M byte. If the ModR/M byte is within the range of 00H-BFH, bits 3-5 of the ModR/M byte are used as an opcode extension, similar to the technique used for 1-and 2-byte opcodes (see Section A.4). If the ModR/M byte is outside the range of 00H through BFH, the entire ModR/M byte is used as an opcode extension.

A.5.1 Opcode Look-up Examples for Escape Instruction Opcodes

Examples are provided below.

Example A-5. Opcode with ModR/M Byte in the 00H through BFH Range

DD0504000000H can be interpreted as follows:

- The instruction encoded with this opcode can be located in Section. Since the ModR/M byte (05H) is within the 00H through BFH range, bits 3 through 5 (000) of this byte indicate the opcode for an FLD double-real instruction (see Table A-9).
- The double-real value to be loaded is at 00000004H (the 32-bit displacement that follows and belongs to this opcode).

Example A-6. Opcode with ModR/M Byte outside the 00H through BFH Range

D8C1H can be interpreted as follows:

- This example illustrates an opcode with a ModR/M byte outside the range of 00H through BFH. The instruction can be located in Section A.4.

- In Table A-8, the ModR/M byte C1H indicates row C, column 1 (the FADD instruction using ST(0), ST(1) as operands).

A.5.2 Escape Opcode Instruction Tables

Tables are listed below.

A.5.2.1 Escape Opcodes with D8 as First Byte

Table A-7 and A-8 contain maps for the escape instruction opcodes that begin with D8H. Table A-7 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-7. D8 Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte (refer to Figure A.4)							
000B	001B	010B	011B	100B	101B	110B	111B
FADD single-real	FMUL single-real	FCOM single-real	FCOMP single-real	FSUB single-real	FSUBR single-real	FDIV single-real	FDIVR single-real

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-8 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-8. D8 Opcode Map When ModR/M Byte is Outside 00H to BFH*

	0	1	2	3	4	5	6	7
C	FADD							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCOM							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),T(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FSUB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F	FDIV							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C	FMUL							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCOMP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),T(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FSUBR							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F	FDIVR							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.2 Escape Opcodes with D9 as First Byte

Table A-9 and A-10 contain maps for escape instruction opcodes that begin with D9H. Table A-9 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-9. D9 Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FLD single-real		FST single-real	FSTP single-real	FLDENV 14/28 bytes	FLDCW 2 bytes	FSTENV 14/28 bytes	FSTCW 2 bytes

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-10 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-10. D9 Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FLD							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FNOP							
E	FCHS	FABS			FTST	FXAM		
F	F2XM1	FYL2X	FPTAN	FPATAN	FXTRACT	FPREM1	FDECSTP	FINCSTP

	8	9	A	B	C	D	E	F
C	FXCH							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D								
E	FLD1	FLDL2T	FLDL2E	FLDPI	FLDLG2	FLDLN2	FLDZ	
F	FPREM	FYL2XP1	FSQRT	FSINCOS	FRNDINT	FSCALE	FSIN	FCOS

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.3 Escape Opcodes with DA as First Byte

Table A-11 and A-12 contain maps for escape instruction opcodes that begin with DAH. Table A-11 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-11. DA Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FIADD dword-integer	FIMUL dword-integer	FICOM dword-integer	FICOMP dword-integer	FISUB dword-integer	FISUBR dword-integer	FIDIV dword-integer	FIDIVR dword-integer

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-11 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-12. DA Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FCMOVB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVBE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E								
F								

	8	9	A	B	C	D	E	F
C	FCMOVE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVU							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E		FUCOMPP						
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.4 Escape Opcodes with DB as First Byte

Table A-13 and A-14 contain maps for escape instruction opcodes that begin with DBH. Table A-13 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-13. DB Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FILD dword-integer	FISTTP dword-integer	FIST dword-integer	FISTP dword-integer		FLD extended-real		FSTP extended-real

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-14 shows the map if the ModR/M byte is outside the range of 00H-BFH. Here, the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-14. DB Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FCMOVNB							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVNBE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E			FCLEX	FINIT				
F	FCOMI							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C	FCMOVNE							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
D	FCMOVNU							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
E	FUCOMI							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.5 Escape Opcodes with DC as First Byte

Table A-15 and A-16 contain maps for escape instruction opcodes that begin with DCH. Table A-15 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-15. DC Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte (refer to Figure A-1)							
000B	001B	010B	011B	100B	101B	110B	111B
FADD double-real	FMUL double-real	FCOM double-real	FCOMP double-real	FSUB double-real	FSUBR double-real	FDIV double-real	FDIVR double-real

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-16 shows the map if the ModR/M byte is outside the range of 00H-BFH. In this case the first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-16. DC Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FADD							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUBR							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVR							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

	8	9	A	B	C	D	E	F
C	FMUL							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUB							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIV							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.6 Escape Opcodes with DD as First Byte

Table A-17 and A-18 contain maps for escape instruction opcodes that begin with DDH. Table A-17 shows the map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-17. DD Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FLD double-real	FISTTP integer64	FST double-real	FSTP double-real	FRSTOR 98/108bytes		FSAVE 98/108bytes	FSTSW 2 bytes

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-18 shows the map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-18. DD Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FFREE							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
D	FST							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
E	FUCOM							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F								

	8	9	A	B	C	D	E	F
C								
D	FSTP							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
E	FUCOMP							
	ST(0)	ST(1)	ST(2)	ST(3)	ST(4)	ST(5)	ST(6)	ST(7)
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.7 Escape Opcodes with DE as First Byte

Table A-19 and A-20 contain opcode maps for escape instruction opcodes that begin with DEH. Table A-19 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. In this case, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-19. DE Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FIADD word-integer	FIMUL word-integer	FICOM word-integer	FICOMP word-integer	FISUB word-integer	FISUBR word-integer	FIDIV word-integer	FIDIVR word-integer

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-20 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-20. DE Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C	FADDP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D								
E	FSUBRP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVRP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

	8	9	A	B	C	D	E	F
C	FMULP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
D		FCOMPP						
E	FSUBP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0)	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)
F	FDIVP							
	ST(0),ST(0)	ST(1),ST(0)	ST(2),ST(0).	ST(3),ST(0)	ST(4),ST(0)	ST(5),ST(0)	ST(6),ST(0)	ST(7),ST(0)

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

A.5.2.8 Escape Opcodes with DF As First Byte

Table A-21 and A-22 contain the opcode maps for escape instruction opcodes that begin with DFH. Table A-21 shows the opcode map if the ModR/M byte is in the range of 00H-BFH. Here, the value of bits 3-5 (the nnn field in Figure A-1) selects the instruction.

Table A-21. DF Opcode Map When ModR/M Byte is Within 00H to BFH *

nnn Field of ModR/M Byte							
000B	001B	010B	011B	100B	101B	110B	111B
FILD word-integer	FISTTP word-integer	FIST word-integer	FISTP word-integer	FBLD packed-BCD	FILD qword-integer	FBSTP packed-BCD	FISTP qword-integer

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

Table A-22 shows the opcode map if the ModR/M byte is outside the range of 00H-BFH. The first digit of the ModR/M byte selects the table row and the second digit selects the column.

Table A-22. DF Opcode Map When ModR/M Byte is Outside 00H to BFH *

	0	1	2	3	4	5	6	7
C								
D								
E	FSTSW AX							
F	FCOMIP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)

	8	9	A	B	C	D	E	F
C								
D								
E	FUCOMIP							
	ST(0),ST(0)	ST(0),ST(1)	ST(0),ST(2)	ST(0),ST(3)	ST(0),ST(4)	ST(0),ST(5)	ST(0),ST(6)	ST(0),ST(7)
F								

NOTES:

* All blanks in all opcode maps are reserved and must not be used. Do not depend on the operation of undefined or reserved locations.

18. Segment reference corrected

In Section 4.6, "Privilege Level Checking When Accessing Data Segments," of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3A*, the wrong segment indicated in a statement has been corrected. See the bold type and the change bar.

4.6 PRIVILEGE LEVEL CHECKING WHEN ACCESSING DATA SEGMENTS

To access operands in a data segment, the segment selector for the data segment must be loaded into the data-segment registers (DS, ES, FS, or GS) or into the stack-segment register (SS). (Segment registers can be loaded with the MOV, POP, LDS, LES, LFS, LGS, and LSS instructions.) Before the processor loads a segment selector into a segment register, it performs a privilege check (see Figure 4-4) by comparing the privilege levels of the currently running program or task (the CPL), the RPL of the segment selector, and the DPL of the segment's segment descriptor. The processor loads the segment selector into the segment register if the DPL is numerically greater than or equal to both the CPL and the RPL. Otherwise, a general-protection fault is generated and the segment register is not loaded.

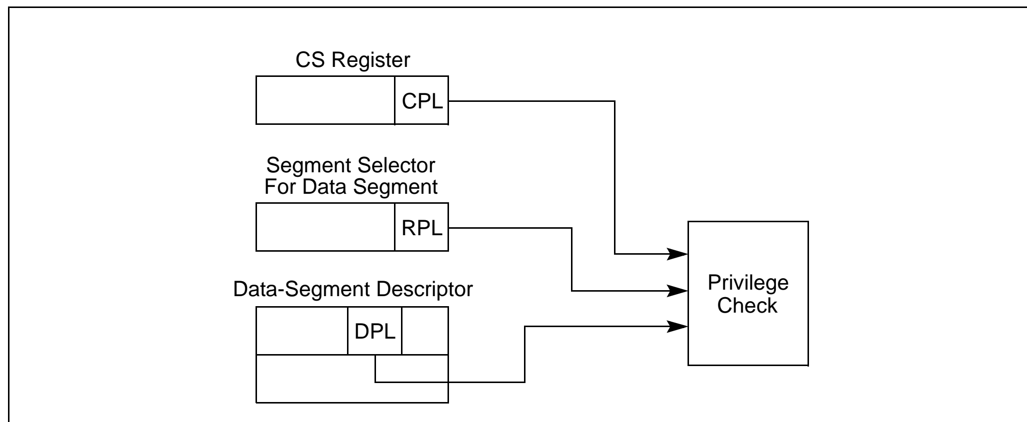

Figure 4-4. Privilege Check for Data Access

Figure 4-5 shows four procedures (located in codes segments A, B, C, and D), each running at different privilege levels and each attempting to access the same data segment.

1. The procedure in code segment A is able to access data segment E using segment selector E1, because the CPL of code segment A and the RPL of segment selector E1 are equal to the DPL of data segment E.
2. The procedure in code segment B is able to access data segment E using segment selector E2, because the **CPL of code segment B** and the RPL of segment selector E2 are both numerically lower than (more privileged) than the DPL of data segment E. A code segment B procedure can also access data segment E using segment selector E1.
3. The procedure in code segment C is not able to access data segment E using segment selector E3 (dotted line), because the CPL of code segment C and the RPL of segment selector E3 are both numerically greater than (less privileged) than the DPL of data segment E. Even if a code segment C procedure were to use segment selector E1 or E2, such that the RPL would be acceptable, it still could not access data segment E because its CPL is not privileged enough.
4. The procedure in code segment D should be able to access data segment E because code segment D's CPL is numerically less than the DPL of data segment E. However, the RPL of segment selector E3 (which the code segment D procedure is using to access data segment E) is numerically greater than the DPL of data segment E, so access is not allowed. If the code segment D procedure were to use segment selector E1 or E2 to access the data segment, access would be allowed.

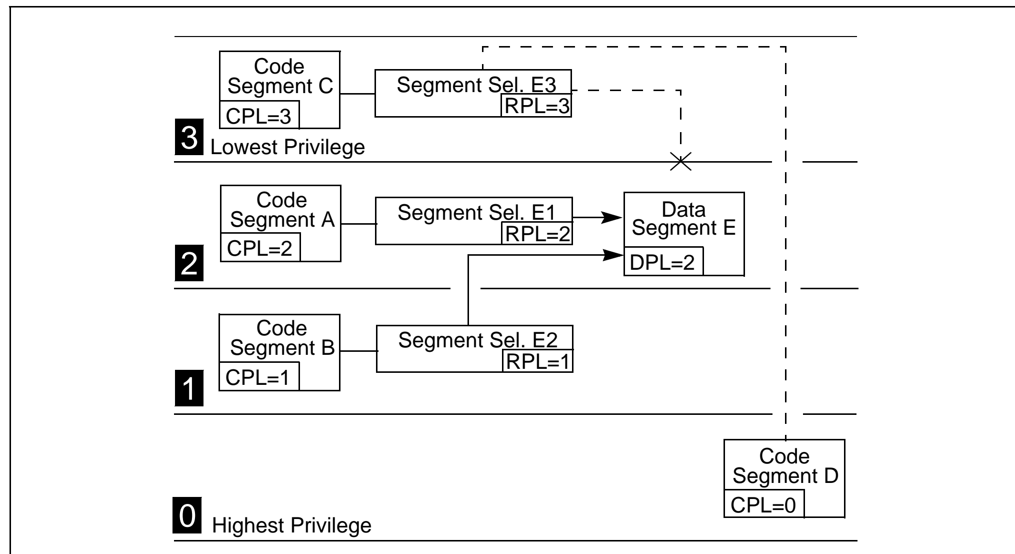

Figure 4-5. Examples of Accessing Data Segments From Various Privilege Levels

As demonstrated in the previous examples, the addressable domain of a program or task varies as its CPL changes. When the CPL is 0, data segments at all privilege levels are accessible; when the CPL is 1, only data segments at privilege levels 1 through 3 are accessible; when the CPL is 3, only data segments at privilege level 3 are accessible.

The RPL of a segment selector can always override the addressable domain of a program or task. When properly used, RPLs can prevent problems caused by accidental (or intensional) use of segment selectors for privileged data segments by less privileged programs or procedures.

It is important to note that the RPL of a segment selector for a data segment is under software control. For example, an application program running at a CPL of 3 can set the RPL for a data-segment selector to 0. With the RPL set to 0, only the CPL checks, not the RPL checks, will provide protection against deliberate, direct attempts to violate privilege-level security for the data segment. To prevent these types of privilege-level-check violations, a program or procedure can check access privileges whenever it receives a data-segment selector from another procedure (see Section 4.10.4, “Checking Caller Access Privileges (ARPL Instruction)”).

19. Figure callout corrected

In Figure 4-7 in the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3A*, a box indicating DPL has been corrected. See the bold text and the change bar.

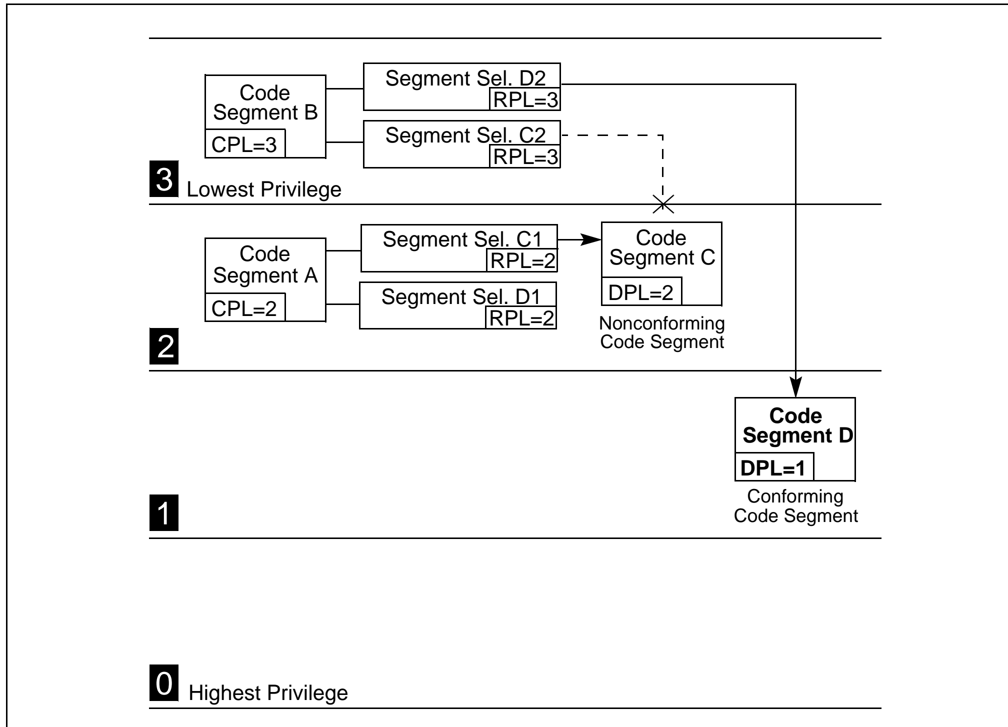

Figure 4-7. Examples of Accessing Conforming and Nonconforming Code Segments From Various Privilege Levels

20. Error in table corrected

In Table 2-2 of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3A*, a typing error has been corrected in a summary table entry. See the change bar and the bold text below.

Table 2-2. Summary of System Instructions

Instruction	Description	Useful to Application?	Protected from Application?
LLDT	Load LDT Register	No	Yes
SLDT	Store LDT Register	No	No
LGDT	Load GDT Register	No	Yes
SGDT	Store GDT Register	No	No
LTR	Load Task Register	No	Yes
STR	Store Task Register	No	No
LIDT	Load IDT Register	No	Yes
SIDT	Store IDT Register	No	No
MOV CR*n*	Load and store control registers	No	Yes
SMSW	Store MSW	Yes	No
LMSW	Load MSW	No	Yes
CLTS	Clear TS flag in CR0	No	Yes

Table 2-2. Summary of System Instructions (Continued)

Instruction	Description	Useful to Application?	Protected from Application?
ARPL	Adjust RPL	Yes[1, 5]	No
LAR	Load Access Rights	Yes	No
LSL	Load Segment Limit	Yes	No
VERR	Verify for Reading	Yes	No
VERW	Verify for Writing	Yes	No
MOV DR*n*	Load and store debug registers	No	Yes
INVD	Invalidate cache, no writeback	No	Yes
WBINVD	Invalidate cache, with writeback	No	Yes
INVLPG	Invalidate TLB entry	No	Yes
HLT	Halt Processor	No	Yes
LOCK (Prefix)	Bus Lock	Yes	No
RSM	Return from system management mode	No	Yes
RDMSR[3]	Read Model-Specific Registers	No	Yes
WRMSR[3]	Write Model-Specific Registers	No	Yes
RDPMC[4]	Read Performance-Monitoring Counter	Yes	Yes[2]
RDTSC[3]	Read Time-Stamp Counter	Yes	Yes[2]

NOTES:
1. Useful to application programs running at a CPL of 1 or 2.
2. The TSD and PCE flags in control register CR4 control access to these instructions by application programs running at a CPL of 3.
3. These instructions were introduced into the IA-32 Architecture with the Pentium processor.
4. This instruction was introduced into the IA-32 Architecture with the Pentium Pro processor and the Pentium processor with MMX technology.
5. This instruction is not supported in 64-bit mode.

21. Encodings added for Multi-Byte No Operation

See Table B-13 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 2B*. Information has been added about the Multi-Byte No Operation. This information is reprinted below. Note that only new cells from the table are provided. Other parts of the table have not changed.

Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes

NOP – Multi-byte No Operation[1]	
register	0000 1111 0001 1111 : 11 000 reg
memory	0000 1111 0001 1111 : mod 000 r/m

NOTES:
1. The multi-byte NOP instruction does not alter the content of the register and will not issue a memory operation.

Information about multi-byte no operations has also been added to the "NOP—No Operation" section of Chapter 4 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 2B*. This section is reprinted below.

NOP—No Operation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
90	NOP	Valid	Valid	One byte no-operation instruction.
0F 1F /0	NOP r/m16	Valid	Valid	Multi-byte no-operation instruction.
0F 1F /0	NOP r/m32	Valid	Valid	Multi-byte no-operation instruction.

Description

This instruction performs no operation. It is a one-byte or multi-byte NOP that takes up space in the instruction stream but does not impact machine context, except for the EIP register.

The multi-byte form of NOP is available on processors with model encoding:

- CPUID.01H.EAX[Bytes 11:8] = 0110B or 1111B

The multi-byte NOP instruction does not alter the content of the register and will not issue a memory operation. The instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

The multi-byte NOP instruction performs no operation on supported processors and generates undefined opcode exception on processors that do not support the multi-byte NOP instruction.

The memory operand form of the instruction allows software to create a byte sequence of "no operation" as one instruction. For situations where multiple-byte NOPs are needed, the recommended operations (32-bit mode and 64-bit mode) are:

Table 4-1. Recommended Multi-Byte Sequence of NOP Instruction

Length	Assembly	Byte Sequence
2 bytes	66 NOP	66 90H
3 bytes	NOP DWORD ptr [EAX]	0F 1F 00H
4 bytes	NOP DWORD ptr [EAX + 00H]	0F 1F 40 00H
5 bytes	NOP DWORD ptr [EAX + EAX*1 + 00H]	0F 1F 44 00 00H
6 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
7 bytes	NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
8 bytes	NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
9 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H

Flags Affected

None.

Exceptions (All Operating Modes)

None.

22. Update RMDPMC documentation adding more family-specific data

See the “RDPMC—Read Performance-Monitoring Counters” subsection in Chapter 4 of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 2B*. The section has been updated to provide more information on processor family specific behavior. The section has been reproduced below.

RDPMC—Read Performance-Monitoring Counters

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 33	RDPMC	Valid	Valid	Read performance-monitoring counter specified by ECX into EDX:EAX.

Description

Loads the 40-bit performance-monitoring counter specified in the ECX register into registers EDX:EAX. The EDX register is loaded with the high-order 8 bits of the counter and the EAX register is loaded with the low-order 32 bits. The counter to be read is specified with an unsigned integer placed in the ECX register.

The indices used to specify performance counters are model-specific and may vary by processor implementations. See Table 4-2 for valid indices for each processor family.

Table 4-2. Valid Performance Counter Index Range for RDPMC

Processor Family	CPUID Family/Model/ Other Signatures	Valid PMC Index Range	40-bit Counters
P6	Family 06H	0, 1	0, 1
Pentium 4, Intel Xeon processors	Family 0FH; Model 00H, 01H, 02H	≥ 0 and ≤ 17	≥ 0 and ≤ 17
Pentium 4, Intel Xeon processors	(Family 0FH; Model 03H, 04H, 06H) and (L3 is absent)	≥ 0 and ≤ 17	≥ 0 and ≤ 17
Pentium M processors	Family 06H, Model 09H, 0DH	0, 1	0, 1
64-bit Intel Xeon processors with L3 (see Chapter 18of the *IA-32 Intel® Architecture Software Developer’s Manual, Volume 3B*)	(Family 0FH; Model 03H, 04H) and (L3 is present)	≥ 0 and ≤ 25	≥ 0 and ≤ 17
Intel Core Solo and Core Duo processors	Family 06H, Model 0EH	0, 1	0, 1

The Pentium 4 and Intel Xeon processors also support “fast” (32-bit) and “slow” (40-bit) reads on the first 18 performance counters. Selected this option using ECX[bit 31]. If bit 31 is set, RDPMC reads only the low 32 bits of the selected performance counter. If bit 31 is clear, all 40 bits are read. A 32-bit result is returned in EAX and EDX is set to 0. A 32-bit read executes faster on Pentium 4 processors and Intel Xeon processors than a full 40-bit read.

On 64-bit Intel Xeon processors with L3, performance counters with indices 18-25 are 32-bit counters. EDX is cleared after executing RDPMC for these counters.

When in protected or virtual 8086 mode, the performance-monitoring counters enabled (PCE) flag in register CR4 restricts the use of the RDPMC instruction as follows. When the PCE flag is set, the RDPMC instruction can be executed at any privilege level; when the flag is clear, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDPMC instruction is always enabled.)

The performance-monitoring counters can also be read with the RDMSR instruction, when executing at privilege level 0.

The performance-monitoring counters are event counters that can be programmed to count events such as the number of instructions decoded, number of interrupts received, or number of cache loads. Appendix A, "Performance Monitoring Events," in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3B*, lists the events that can be counted for the Pentium 4, Intel Xeon, and earlier IA-32 processors.

The RDPMC instruction is not a serializing instruction; that is, it does not imply that all the events caused by the preceding instructions have been completed or that events caused by subsequent instructions have not begun. If an exact event count is desired, software must insert a serializing instruction (such as the CPUID instruction) before and/or after the RDPCM instruction.

In the Pentium 4 and Intel Xeon processors, performing back-to-back fast reads are not guaranteed to be monotonic. To guarantee monotonicity on back-to-back reads, a serializing instruction must be placed between the tow RDPMC instructions.

The RDPMC instruction can execute in 16-bit addressing mode or virtual-8086 mode; however, the full contents of the ECX register are used to select the counter, and the event count is stored in the full EAX and EDX registers. The RDPMC instruction was introduced into the IA-32 Architecture in the Pentium Pro processor and the Pentium processor with MMX technology. The earlier Pentium processors have performance-monitoring counters, but they must be read with the RDMSR instruction.

In 64-bit mode, RDPMC behavior is unchanged from 32-bit mode. The upper 32 bits of RAX and RDX are cleared.

Operation

```
(* P6 family processors and Pentium processor with MMX technology *)
IF (ECX = 0 or 1) and ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN
        EAX ← PMC(ECX)[31:0];
        EDX ← PMC(ECX)[39:32];
    ELSE (* ECX is not 0 or 1 or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0);
FI;
(* Processors with CPUID family 15 *)
IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN IF (ECX[30:0] = 0:17)
        THEN IF ECX[31] = 0
            THEN IF 64-Bit Mode
                THEN
```

```
                RAX[31:0] ← PMC(ECX[30:0])[31:0]; (* 40-bit read *)
                RAX[63:32] ← 0;
                RDX[31:0] ← PMC(ECX[30:0])[39:32];
                RDX[63:32] ← 0;
            ELSE
                EAX ← PMC(ECX[30:0])[31:0]; (* 40-bit read *)
                EDX ← PMC(ECX[30:0])[39:32];
            FI;
    ELSE IF ECX[31] = 1
        THEN IF 64-Bit Mode
            THEN
                RAX[31:0] ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
                RAX[63:32] ← 0;
                RDX ← 0;
            ELSE
                EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
                EDX ← 0;
        FI;
    FI;
  ELSE IF (*64-bit Intel Xeon processor with L3 *)
    THEN IF (ECX[30:0] = 18:25
        EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
        EDX ← 0;
    FI;
  ELSE  (* Invalid PMC index in ECX[30:0], see Table 4-4. *)
    GP(0);
  FI;
ELSE  (* CR4.PCE = 0 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
  #GP(0);
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.

If an invalid performance counter index is specified (see Table 4-2).

(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range.

Real-Address Mode Exceptions

#GP — If an invalid performance counter index is specified (see Table 4-2).

(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range.

Virtual-8086 Mode Exceptions

#GP(0) — If the PCE flag in the CR4 register is clear.

If an invalid performance counter index is specified (see Table 4-2).

(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0) — If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.

If an invalid performance counter index is specified in ECX[30:0] (see Table 4-2).

23. Sections covering variable range MTRRs updated

See Chapter 10 in the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3A*. Sections covering variable range MTRRs have been updated. These are reproduced below. See the change bars.

10.11.2.3 variable Range MTRRs

The Pentium 4, Intel Xeon, and P6 family processors permit software to specify the memory type for eight variable-size address ranges, using a pair of MTRRs for each range. The first entry in each pair (IA32_MTRR_PHYSBASE*n*) defines the base address and memory type for the range; the second entry (IA32_MTRR_PHYSMASK*n*) contains a mask used to determine the address range. The "*n*" suffix indicates register pairs 0 through 7.

For P6 family processors, the prefixes for these variable range MTRRs are MTRRphysBase and MTRRphysMask.

Table 10-9. Address Mapping for Fixed-Range MTRRs

Address Range (hexadecimal)								MTRR
63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0	
70000-7FFFF	60000-6FFFF	50000-5FFFF	40000-4FFFF	30000-3FFFF	20000-2FFFF	10000-1FFFF	00000-0FFFF	IA32_MTRR_FIX64K_00000
9C000 9FFFF	98000-98FFF	94000-97FFF	90000-93FFF	8C000-8FFFF	88000-8BFFF	84000-87FFF	80000-83FFF	IA32_MTRR_FIX16K_80000
BC000 BFFFF	B8000-BBFFF	B4000-B7FFF	B0000-B3FFF	AC000-AFFFF	A8000-ABFFF	A4000-A7FFF	A0000-A3FFF	IA32_MTRR_FIX16K_A0000

Table 10-9. Address Mapping for Fixed-Range MTRRs (Continued)

Address Range (hexadecimal)								MTRR
63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0	
C7000 C7FFF	C6000- C6FFF	C5000- C5FFF	C4000- C4FFF	C3000- C3FFF	C2000- C2FFF	C1000- C1FFF	C0000- C0FFF	IA32_MTRR_ FIX4K_C0000
CF000 CFFFF	CE000- CEFFF	CD000- CDFFF	CC000- CCFFF	CB000- CBFFF	CA000- CAFFF	C9000- C9FFF	C8000- C8FFF	IA32_MTRR_ FIX4K_C8000
D7000 D7FFF	D6000- D6FFF	D5000- D5FFF	D4000- D4FFF	D3000- D3FFF	D2000- D2FFF	D1000- D1FFF	D0000- D0FFF	IA32_MTRR_ FIX4K_D0000
DF000 DFFFF	DE000- DEFFF	DD000- DDFFF	DC000- DCFFF	DB000- DBFFF	DA000- DAFFF	D9000- D9FFF	D8000- D8FFF	IA32_MTRR_ FIX4K_D8000
E7000 E7FFF	E6000- E6FFF	E5000- E5FFF	E4000- E4FFF	E3000- E3FFF	E2000- E2FFF	E1000- E1FFF	E0000- E0FFF	IA32_MTRR_ FIX4K_E0000
EF000 EFFFF	EE000- EEFFF	ED000- EDFFF	EC000- ECFFF	EB000- EBFFF	EA000- EAFFF	E9000- E9FFF	E8000- E8FFF	IA32_MTRR_ FIX4K_E8000
F7000 F7FFF	F6000- F6FFF	F5000- F5FFF	F4000- F4FFF	F3000- F3FFF	F2000- F2FFF	F1000- F1FFF	F0000- F0FFF	IA32_MTRR_ FIX4K_F0000
FF000 FFFFF	FE000- FEFFF	FD000- FDFFF	FC000- FCFFF	FB000- FBFFF	FA000- FAFFF	F9000- F9FFF	F8000- F8FFF	IA32_MTRR_ FIX4K_F8000

Figure 10-6 shows flags and fields in these registers. The functions of these flags and fields are:

- **Type field, bits 0 through 7** — Specifies the memory type for the range (see Table 10-8 for the encoding of this field).
- **PhysBase field, bits 12 through (MAXPHYADDR-1)** — Specifies the base address of the address range. This 24-bit value, in the case where MAXPHYADDR is 36 bits, is extended by 12 bits at the low end to form the base address (this automatically aligns the address on a 4-KByte boundary).
- **PhysMask field, bits 12 through (MAXPHYADDR-1)** — Specifies a mask (24 bits if the maximum physical address size is 36 bits, 28 bits if the maximum physical address size is 40 bits). The mask determines the range of the region being mapped, according to the following relationships:
  - — Address_Within_Range AND PhysMask = PhysBase AND PhysMask
  - — This value is extended by 12 bits at the low end to form the mask value. For more information: see Section 10.11.3, “Example Base and Mask Calculations.”
  - — The width of the PhysMask field depends on the maximum physical address size supported by the processor.

    CPUID.80000008H reports the maximum physical address size supported by the processor. If CPUID.80000008H is not available, software may assume that the processor supports a 36-bit physical address size (then PhysMask is 24 bits wide and the upper 28 bits of IA32_MTRR_PHYSMASKn are reserved). See the Note below.
- **V (valid) flag, bit 11** — Enables the register pair when set; disables register pair when clear.

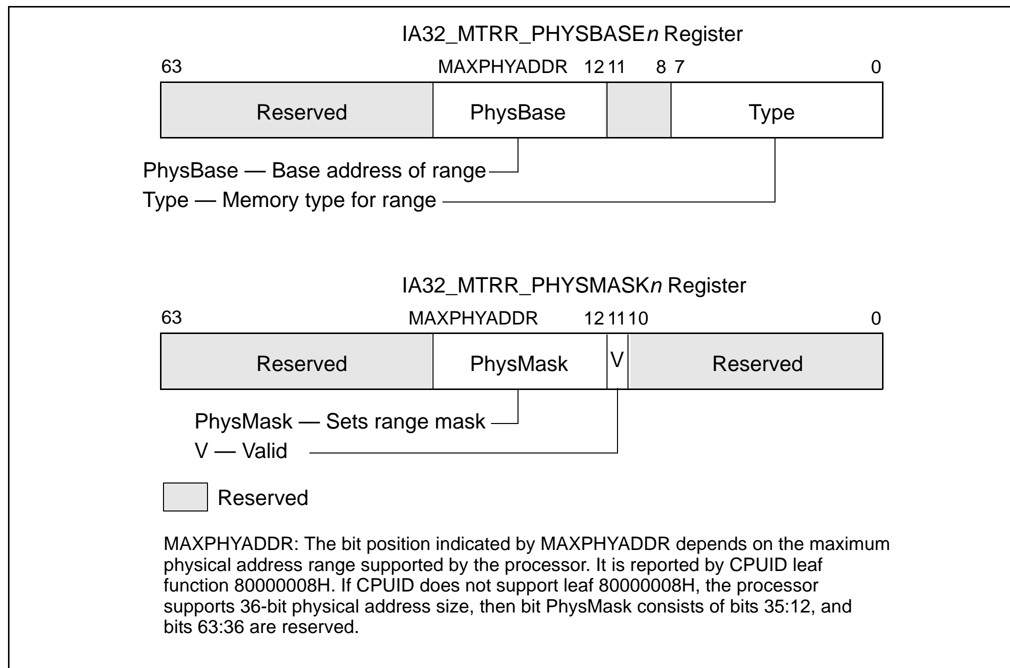

Figure 10-6. IA32_MTRR_PHYSBASE*n* and IA32_MTRR_PHYSMASK*n* Variable-Range Register Pair

All other bits in the IA32_MTRR_PHYSBASE*n* and IA32_MTRR_PHYSMASK*n* registers are reserved; the processor generates a general-protection exception (#GP) if software attempts to write to them.

Some mask values can result in ranges that are not continuous. In such ranges, the area not mapped by the mask value is set to the default memory type. Intel does not encourage the use of "discontinuous" ranges because they could require physical memory to be present throughout the entire 4-GByte physical memory map. If memory is not provided, the behaviour is undefined.

NOTE

It is possible for software to parse the memory descriptions that BIOS provides by using the ACPI/INT15 e820 interface mechanism. This information then can be used to determine how MTRRs are initialized (for example: allowing the BIOS to define valid memory ranges and the maximum memory range supported by the platform, including the processor).

See Section 10.11.4.1, "MTRR Precedences," for information on overlapping variable MTRR ranges.

10.11.3 Example Base and Mask Calculations

The examples in this section apply to processors that support a maximum physical address size of 36 bits. The base and mask values entered in variable-range MTRR pairs are 24-bit values that the processor extends to 36-bits.

For example, to enter a base address of 2 MBytes (200000H) in the IA32_MTRR_PHYSBASE3 register, the 12 least-significant bits are truncated and the value 000200H is entered in the PhysBase

field. The same operation must be performed on mask values. For example, to map the address range from 200000H to 3FFFFFH (2 MBytes to 4 MBytes), a mask value of FFFE00000H is required. Again, the 12 least-significant bits of this mask value are truncated, so that the value entered in the PhysMask field of IA32_MTRR_PHYSMASK3 is FFFE00H. This mask is chosen so that when any address in the 200000H to 3FFFFFH range is AND'd with the mask value, it will return the same value as when the base address is AND'd with the mask value (which is 200000H).

To map the address range from 400000H to 7FFFFFH (4 MBytes to 8 MBytes), a base value of 000400H is entered in the PhysBase field and a mask value of FFFC00H is entered in the PhysMask field.

Example 1-1. Setting-Up Memory for a System

Here is an example of setting up the MTRRs for an system. Assume that the system has the following characteristics:

- 96 MBytes of system memory is mapped as write-back memory (WB) for highest system performance.
- A custom 4-MByte I/O card is mapped to uncached memory (UC) at a base address of 64 MBytes. This restriction forces the 96 MBytes of system memory to be addressed from 0 to 64 MBytes and from 68 MBytes to 100 MBytes, leaving a 4-MByte hole for the I/O card.
- An 8-MByte graphics card is mapped to write-combining memory (WC) beginning at address A0000000H.
- The BIOS area from 15 MBytes to 16 MBytes is mapped to UC memory.

The following settings for the MTRRs will yield the proper mapping of the physical address space for this system configuration.

IA32_MTRR_PHYSBASE0 = 0000 0000 0000 0006H
IA32_MTRR_PHYSMASK0 = 0000 000F FC00 0800H
Caches 0-64 MByte as WB cache type.

IA32_MTRR_PHYSBASE1 = 0000 0000 0400 0006H
IA32_MTRR_PHYSMASK1 = 0000 000F FE00 0800H
Caches 64-96 MByte as WB cache type.

IA32_MTRR_PHYSBASE2 = 0000 0000 0600 0006H
IA32_MTRR_PHYSMASK2 = 0000 000F FFC0 0800H
Caches 96-100 MByte as WB cache type.

IA32_MTRR_PHYSBASE3 = 0000 0000 0400 0000H
IA32_MTRR_PHYSMASK3 = 0000 000F FFC0 0800H
Caches 64-68 MByte as UC cache type.

IA32_MTRR_PHYSBASE4 = 0000 0000 00F0 0000H
IA32_MTRR_PHYSMASK4 = 0000 000F FFF0 0800H
Caches 15-16 MByte as UC cache type.

IA32_MTRR_PHYSBASE5 = 0000 0000 A000 0001H
IA32_MTRR_PHYSMASK5 = 0000 000F FF80 0800H
Caches A0000000-A0800000 as WC type.

This MTRR setup uses the ability to overlap any two memory ranges (as long as the ranges are mapped to WB and UC memory types) to minimize the number of MTRR registers that are required to configure the memory environment. This setup also fulfills the requirement that two register pairs are left for operating system usage.

10.11.3.1 Base and Mask Calculations with Intel EM64T

For IA-32 processors that support greater than 36 bits of physical address size, software should query CPUID.80000008H to determine the maximum physical address.

Example 8-2. Setting-Up Memory for a System with a 40-Bit Address Size

If a processor supports 40-bits of physical address size, then the PhysMask field (in IA32_MTRR_PHYSMASK*n* registers) is 28 bits instead of 24 bits. For this situation, Example 8-1 should be modified as follows:

IA32_MTRR_PHYSBASE0 = 0000 0000 0000 0006H
IA32_MTRR_PHYSMASK0 = 0000 00FF FC00 0800H
Caches 0-64 MByte as WB cache type.

IA32_MTRR_PHYSBASE1 = 0000 0000 0400 0006H
IA32_MTRR_PHYSMASK1 = 0000 00FF FE00 0800H
Caches 64-96 MByte as WB cache type.

IA32_MTRR_PHYSBASE2 = 0000 0000 0600 0006H
IA32_MTRR_PHYSMASK2 = 0000 00FF FFC0 0800H
Caches 96-100 MByte as WB cache type.

IA32_MTRR_PHYSBASE3 = 0000 0000 0400 0000H
IA32_MTRR_PHYSMASK3 = 0000 00FF FFC0 0800H
Caches 64-68 MByte as UC cache type.

IA32_MTRR_PHYSBASE4 = 0000 0000 00F0 0000H
IA32_MTRR_PHYSMASK4 = 0000 00FF FFF0 0800H
Caches 15-16 MByte as UC cache type.

IA32_MTRR_PHYSBASE5 = 0000 0000 A000 0001H
IA32_MTRR_PHYSMASK5 = 0000 00FF FF80 0800H
Caches A0000000-A0800000 as WC type.

10.11.4 Range Size and Alignment Requirement

The range that is to be mapped to a variable-range MTRR must meet the following “power of 2” size and alignment rules:

1. The minimum range size is 4 KBytes, and the base address of this range must be on at least a 4-KByte boundary.
2. For ranges greater than 4 KBytes, each range must be of length 2^n and its base address must be aligned on a 2^n boundary, where n is a value equal to or greater than 12. The base-address alignment value cannot be less than its length. For example, an 8-KByte range cannot be aligned on a 4-KByte boundary. It must be aligned on at least an 8-KByte boundary.

10.11.4.1 MTRR Precedences

If the MTRRs are not enabled (by setting the E flag in the IA32_MTRR_DEF_TYPE MSR), then all memory accesses are of the UC memory type. If the MTRRs are enabled, then the memory type used for a memory access is determined as follows:

1. If the physical address falls within the first 1 MByte of physical memory and fixed MTRRs are enabled, the processor uses the memory type stored for the appropriate fixed-range MTRR.
2. Otherwise, the processor attempts to match the physical address with a memory type set by the variable-range MTRRs:
   a. If one variable memory range matches, the processor uses the memory type stored in the IA32_MTRR_PHYSBASE*n* register for that range.
   b. If two or more variable memory ranges match and the memory types are identical, then that memory type is used.
   c. If two or more variable memory ranges match and one of the memory types is UC, the UC memory type used.
   d. If two or more variable memory ranges match and the memory types are WT and WB, the WT memory type is used.
   e. For overlaps not defined by the above rules, processor behavior is undefined.
3. If no fixed or variable memory range matches, the processor uses the default memory type.