IA-32 Intel® Architecture Software Developer's Manual

Documentation Changes

January 2004

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 this specification update.

Document Number: 252046-007

Contents

Revision History

Version	Description	Date
-001	Initial Release	November 2002
-002	Added 1-10 Documentation Changes. Removed old Documentation Changes items that already have been incorporated in the published Software Developer's manual	December 2002
-003	Added 9 -17 Documentation Changes Removed Documenation Change #6 - References to bits Gen and Len Deleted Removed Documenation 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

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	245470-011
IA-32 Intel® Architecture Software Developer's Manual: Volume 2, Instruction Set Reference	245471-011
IA-32 Intel® Architecture Software Developer's Manual: Volume 3, System Programming Guide	245472-011

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 Maunal.

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.	IA32_THERM_CONTROL has Been Changed to IA32_CLOCK_MODULATION
2.	INTER-PRIVILEGE" was not Spelled Corretly in Pseudocode Entry
3.	Confusing Text Artifact Removed
4.	IA32_MISC_CTL has Been Removed From the List of Architectural MSRs
5.	Typo Corrected in Figure 8-24
6.	Typo Corrected in Figure 8-23
7.	Corrupted Text Corrected
8.	Corrected an Error in PACKSSDW Illustration
9.	SSM Corrected to SMM
10.	Exiting From SMM Text Updated
11.	L1 Data Cache Context Mode Description has Been Udpated
12.	#DE Should be #DB in Description of EFLAGS.RF
13.	There Have Been Revisions to the Table That States Priority Among Simultaneous Exceptions and Interrupts
14.	Corrections to Page-Directory-Pointer-Table Entry Desciption
15.	Behavior Notes on the Accessed (A) Flag and Dirty (D) Flag
16.	Interrupt 11 Discussion Concerning EXT Flag Functioning Has Been Updated
17.	Improved Information on Interpreting Machine-Check Error Codes
18.	More information on the Functioning of Debug BPs after POP SS/MOV SS Has Been Provided
19.	More Information on the LBR Stack Has Been Provided
20.	Limited Availability of Two MSRs Has Been Documented
21.	The Section On Microcode Update Facilities Has Been Refreshed
22.	A Mechanism for Determining Sync/Async SMIs Has Been Documented

Summary Table of Documentation Changes

Number	DOCUMENTATION CHANGES
23.	Omitted Debug Data Has Been Restored
24.	CLTS Exception Information Improved
25.	The MOVSS Description Have Been Updated
26.	An Instruction Listing (PULLHUW) Has Been Deleted
27.	Some Data Entry Errors in Table B-20 Have Been Corrected
28.	Figure Has Been Corrected
29.	The Description of Minimum Thermal Monitor Activation Time Has Been Updated
30.	Corrected Description of Exception- or Interrupt-Handler Procedures
31.	CMPSD and CMPSS Exception Information Updated
32.	PUNPCKHB*/PUNPCKLB* Exception Information Improved
33.	MOVHPD, MOVLPD, UNPCKHPS, UNPCKLPS Exception Information Improved.
34.	PEXTRW - PINSRW Exception Information Improved
35.	Restructuring of IA-32 manuals, Volume 2
36.	Statement on Setting the OXFXSR Flag Corrected
37.	Disclaimer Now Includes Stronger Warning
38.	Correction in the Description of SYSENTER and SYSEXIT
39.	Exception Lists for ANPD/ANPS Have Been Updated
40.	Cache descriptors for B0H, B3H Have Been Updated
41.	Exception List Corrections for unpcklps, unpckhps, unpcklpd, unpckhpd, shufps, shufpd, rsqrtss, rsqrtps, rcpss, rcpps, movups, movupd, movmskps, movmskpd, movaps, and movapd Have Been Made
42.	Correction to SMCCLEAR
43.	REP INS Syntax Has Been Corrected
44.	SGDT/SIDT Descriptions Are Now In Two Separate Sections
45.	Revision to Vol. 2B, Appendix A & Appendix B

Documentation Changes

1. IA32_THERM_CONTROL has been Changed to IA32_CLOCK_MODULATION

The name of the MSR IA32_THERM_CONTROL has been changed to IA32_CLOCK_MODULATION. This was done to avoid confusion about the MSR's function.

The following corrected table segment is from Appendix B, Table B-3, the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*. See the reproduced text below

Register Address		Register Name	Bit Description
Hex	Dec		
19AH	410	IA32_CLOCK_MODULATI ON	**Clock Modulation.** (R/W) Enables and disables on-demand clock modulation and allows the selection of the on-demand clock modulation duty cycle. (See Section 13.15.3., *Software Controlled Clock Modulation*. NOTE: IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.

2. INTER-PRIVILEGE" was not Spelled Correctly in Pseudocode Entry

The term inter-privilege was incorrectly spelled in pseudocode provided as part of the "INT n/INTO/INT 3—Call to Interrupt Procedure" section, Chapter 3, *IA-32 Intel Architecture Software Developer's Manual, Volume 2.*

The corrected text segment is reproduced below.

```
...INTER-PRIVILEGE-LEVEL-INTERRUPT
   (* PE=1, interrupt or trap gate, non-conforming code segment, DPL<CPL *)
   (* Check segment selector and descriptor for stack of new privilege level in current TSS *)
   IF current TSS is 32-bit TSS
       THEN
           TSSstackAddress ← (new code segment DPL ∗ 8) + 4.....
```

3. Confusing text Artifact Removed

There were some materials in the OPCODE table that should have been deleted. This error has been corrected. The corrected table segment (reproduced below) is in Appendix A, Table A-3, *IA-32 Intel Architecture Software Developer's Manual, Volume 2*. See address 0x0f0b

.

	8	9	A	B	C	D	E	F
0	INVD	WBINVD		**UD2**				

4. IA32_MISC_CTL has been Removed from the list of Architectural MSRs

We removed MSR IA32_MISC_CTL from the list of architectural MSRs. Note that this MSR is still listed in other locations.

The impacted segment (reproduced below) is from Appendix B, Table B-5, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*. The change bars show where the table row was deleted.

79H	121	IA32_BIOS_UPDT_TRIG	BIOS_UPDT_TRIG	P6 Family Processors
8BH	139	IA32_BIOS_SIGN_ID	BIOS_SIGN/BBL_CR_D3	P6 Family Processors
FEH	254	IA32_MTRRCAP	MTRRcap	P6 Family Processors
174H	372	IA32_SYSENTER_CS	SYSENTER_CS_MSR	P6 Family Processors
175H	373	IA32_SYSENTER_ESP	SYSENTER_ESP_MSR	P6 Family Processors

5. Typo Corrected in Figure 8-24

ExINT should be ExtINT in Figure 8-24, located in the “Message Data Register Format” section, Chapter 8, *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*. The corrected figure is reproduced below.

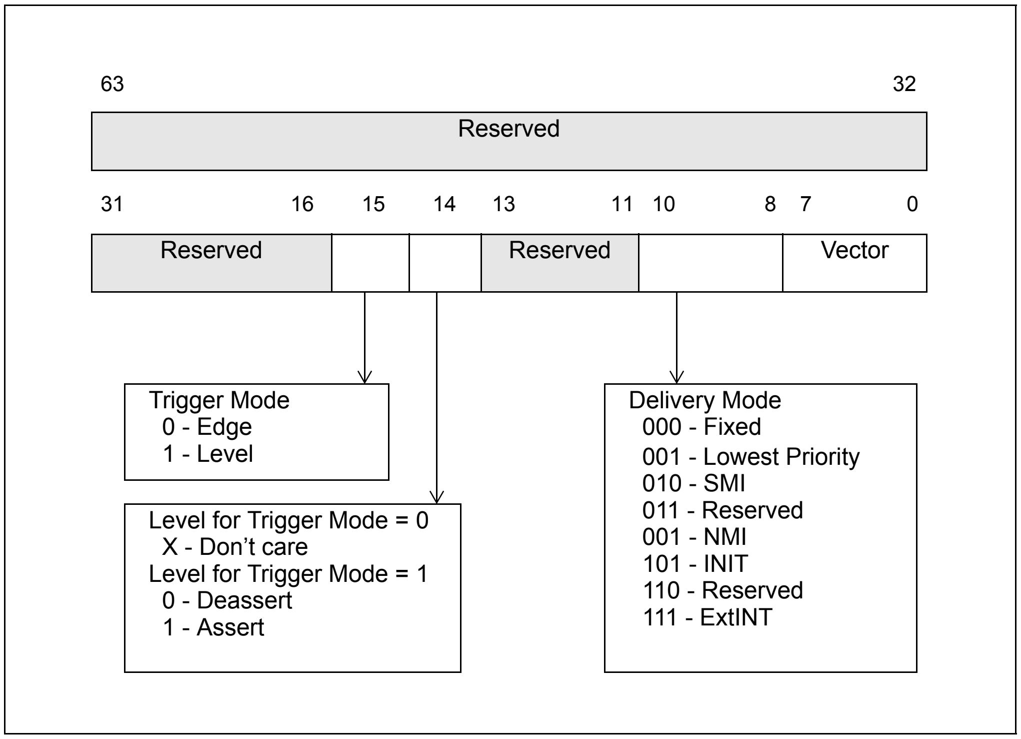


Figure 8-24. Layout of the MSI Message Data Register

6. Typo Corrected in Figure 8-23

0FEEH was incorrectly represented as 0FEEEH in Figure 8-23, located the “Message Address Register Format” section, Chapter 8, *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*.

The corrected figure is reproduced below.

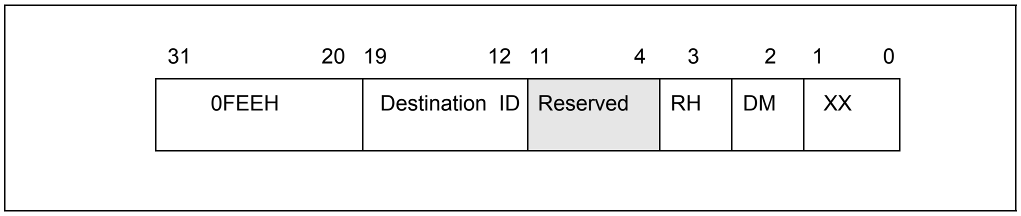


Figure 8-23. Layout of the MSI Message Address Register

7. Corrupted text Corrected

There was some corrupted text in the “State of the Logical Processors” section, Chapter 7, *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*.

The correction is shown in the segment below. See the changebar.

7.6.1.1 State of the Logical Processors

The following features are considered part of the architectural state of a logical processor with HT Technology. The features can be subdivided into three groups:

- Duplicated for each logical processor
- Shared by logical processors in a physical processor
- Shared or duplicated, depending on the implementation

The following features are duplicated for each logical processor:

- General purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, and EBP)

....

8. Corrected an error in PACKSSDW Illustration

Operation of the PACKSSDW instruction was incorrectly illustrated in Figure 3-6, the “PACKSSWB/PACKSSDW—Pack with Signed Saturation” section, Chapter 3, *IA-32 Intel Architecture Software Developer’s Manual, Volume 2*.

The corrected figure is reproduced below.

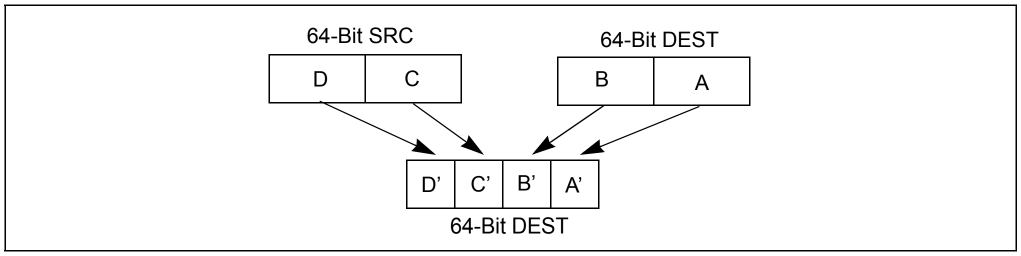


Figure 3.6. Operation of the PACKSSDW Instruction Using 64-bit Operands

9. SSM Corrected to SMM

In several places, SSM was still being used as an acronym for ‘system management mode.’ The correct usage is SMM. Corrections were made in the “Modes of Operation” section, Chapter 3, *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*. The updated paragraph is reproduced below.

....

System management mode (SMM). This mode provides a transparent mechanism for implementing platform-specific functions such as power management and system security. The processor enters SMM when the external SMM interrupt pin (SMI#) is activated or an SMI is received from the advanced programmable interrupt controller (APIC). In SMM, the processor switches to a separate address space while saving the basic context of the currently running program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is placed back into its state prior to the system management interrupt. SMM was introduced with the Intel386™ SL and Intel486™ SL processors and became a standard IA-32 feature with the Pentium processor family.

....

This change was also made in the “RSM—Resume from System Management Mode” section, Chapter 3, *IA-32 Intel Architecture Software Developer’s Manual, Volume 2*. The corrected segments are reproduced below.

....

Returns program control from system management mode (SMM) to the application program or operating-system procedure that was interrupted when the processor received an SMM interrupt. The processor’s state is restored from the dump created upon entering SMM. If the processor detects invalid state information during state restoration, it enters the shutdown state....

```
...
ReturnFromSMM;
ProcessorState ← Restore(SMMDump);
...
```

10. Exiting from SMM text Updated

A paragraph in the “Exiting from SMM” section, Chapter 13, *IA-32 Intel Architecture Software Developer’s Manual, Volume 3* has been updated. The information previously provided was not complete. The corrected text segment is reproduced below. See the change bar for location.

- (For the Pentium and Intel486 processors only.) If the address stored in the SMBASE register when an RSM instruction is executed is not aligned on a 32-KByte boundary. This restriction does not apply to the P6 family processors.

In the shutdown state, Intel processors stop executing instructions until a RESET#, INIT# or NMI# is asserted. Processors do recognize the FLUSH# signal in the shutdown state. While Pentium family processors recognize the SMI# signal in shutdown state, P6 family and Intel486 processors do not. Intel does not support using SMI# to recover from shutdown states for any processor family; the response of processors in this circumstance is not well defined. On Pentium 4 and later processors, shutdown will inhibit INTR and A20M but will not change any of the other inhibits. On these processors, NMIs will be inhibited if no action is taken in the SMM handler to uninhibit them (see Section 13.7.).

If the processor is in the HALT state when the SMI is received, the processor handles the return from SMM slightly differently (see Section 13.10., “Auto HALT Restart”). Also, the SMBASE address can be changed on a return from SMM (see Section 13.11., “SMBASE Relocation”).

11. L1 data Cache Context mode Description has been Udpated

In Appendix B, Table B-1, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*; the "L1 Data Cache Context Mode (RW)" table cell has been updated. Information about adaptive mode was clarified.

The updated table segment is reproduced below.

Register Address		Register Name Fields and Flags	Shared/ Unique[1]	Bit Description
Hex	Dec			
		24		**L1 Data Cache Context Mode (R/W).** When set to 1, this bit places the L1 Data Cache into shared mode. When set to 0 (the default), this bit places the L1 Data Cache into adaptive mode. When the L1 Data Cache is running in adaptive mode and the CR3s are identical, data in L1 is shared across logical processors. Otherwise, data in L1 is not shared and cache use is competitive. NOTE: If the Context ID feature flag, ECX[10], is not set to 1 after executing CPUID with EAX = 1; the ability to switch modes is not supported and the BIOS must not alter the contents of IA32_MISC_ENABLE[24].

12. #DE Should be #DB in Description of EFLAGS.RF

In the "System Flags and Fields in the EFLAGS Register" section, Chapter 2, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*; there was a sentence that began "When set, this flag temporarily disables debug exceptions (#DE)". Debug exceptions are noted as #DB, not #DE. This error has been corrected.

The corrected entry is reproduced below.

RF **Resume (bit 16).** Controls the processor's response to instruction-breakpoint conditions. When set, this flag temporarily disables debug exceptions (#DB) from being generated for instruction breakpoints; although, other exception conditions can cause an exception to be generated. When clear, instruction breakpoints will generate debug exceptions.

The primary function of the RF flag is to allow the restarting of an instruction following a debug exception that was caused by an instruction breakpoint condition. Here, debugger software must set this flag in the EFLAGS image on the stack just prior to returning to the interrupted program with the IRETD instruction, to prevent the instruction breakpoint from causing another debug exception. The processor then automatically clears this flag after the instruction returned to has been successfully executed, enabling instruction breakpoint faults again.

See Section 15.3.1.1., *Instruction-Breakpoint Exception Condition*, for more information on the use of this flag.

13. There have been Revisions to the Table that States Priority Among Simultaneous Exceptions and Interrupts

We have made a number of updates to Table 5-2, located in the “Priority Among Simultaneous Exceptions and Interrupts” section, Chapter 5, *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*.

The updated cells are reproduced below.

Priority	Descriptions (continued)...
5	External Interrupts - NMI Interrupts - Maskable Hardware Interrupts
6	Code Breakpoint Fault
7	Faults from Fetching Next Instruction - Code-Segment Limit Violation - Code Page Fault
8	Faults from Decoding the Next Instruction - Instruction length > 15 bytes - Invalid Opcode - Coprocessor Not Available
9 (Lowest)	Faults on Executing an Instruction - Overflow - Bound error - Invalid TSS - Segment Not Present - Stack fault - General Protection - Data Page Fault - Alignment Check - x87 FPU Floating-point exception - SIMD floating-point exception

14. Corrections to Page-Directory-Pointer-Table Entry Desciption

In Figure 3-20 and 3-21, located in the “Page-Directory and Page-Table Entries With Extended Addressing Enabled” section, Chapter 3, *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*; Bit 0 of both representations of the Page-Directory-Pointer-Table Entry now indicate P (showing the the location of the ‘present flag’ bit).

The corrected tables are reproduced below.

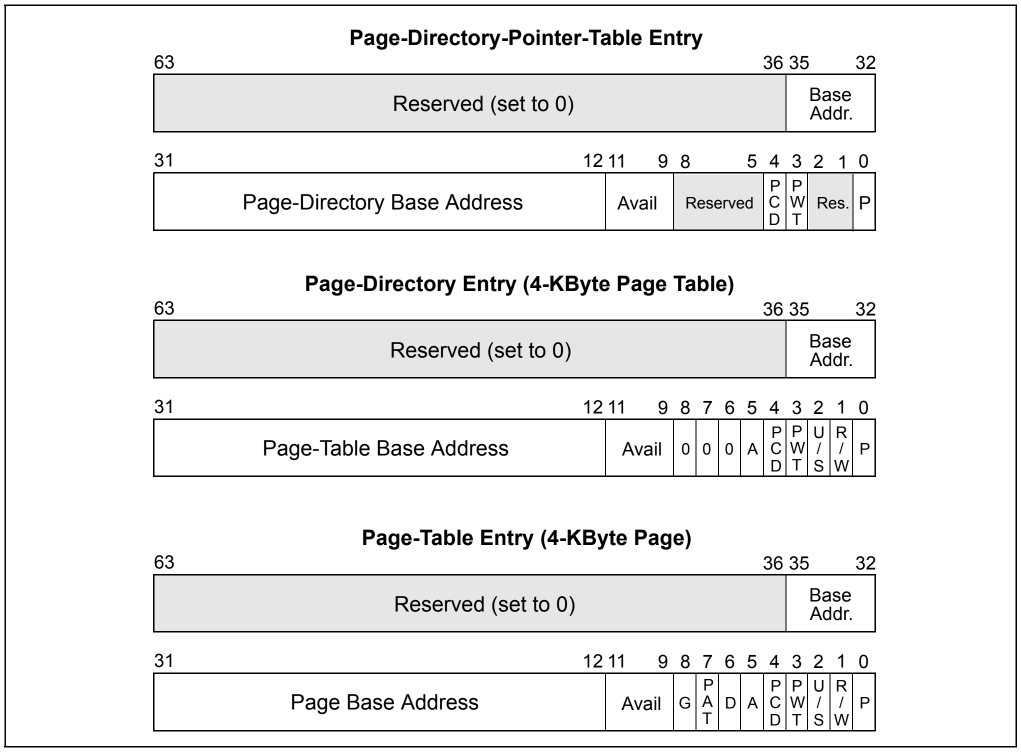


Figure 3-20. Format of Page-Directory-Pointer-Table, Page-Directory, and Page-Table Entries for 4-KByte Pages with PAE Enabled

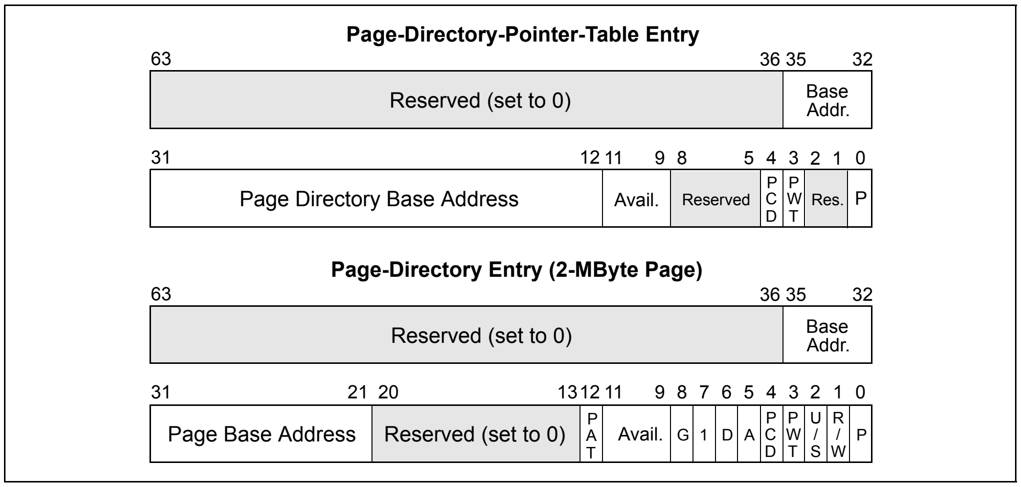


Figure 3-21. Format of Page-Directory-Pointer-Table and Page-Directory Entries for 2-MByte Pages with PAE Enabled

In addition, the paragraph discussing the present flag has been updated. this text is also located in the "Page-Directory and Page-Table Entries With Extended Addressing Enabled" section, Chapter 3, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*.

The applicable text segment is reproduced below. Note the change bar.

For all table entries (except for page-directory entries that point to 2-MByte pages), the bits in the page base address are interpreted as the 24 most-significant bits of a 36-bit physical address, which forces page tables and pages to be aligned on 4-KByte boundaries. When a page-directory entry points to a 2-MByte page, the base address is interpreted as the 15 most-significant bits of a 36-bit physical address, which forces pages to be aligned on 2-MByte boundaries.

The present flag (bit 0) in the page-directory-pointer-table entries can be set to 0 or 1. If the present flag is clear, the remaining bits in the page-directory-pointer-table entry are available to the operating system. If the present flag is set, the fields of the page-directory-pointer-table entry are defined in Figures 3-20 for 4KB pages and Figures 3-21 for 2MB pages.

The page size (PS) flag (bit 7) in a page-directory entry determines if the entry points to a page table or a 2-MByte page. When this flag is clear, the entry points to a page table; when the flag is set, the entry points to a 2-MByte page. This flag allows 4-KByte and 2-MByte pages to be mixed within one set of paging tables.

15. Behavior Notes on the Accessed (A) flag and Dirty (D) flag

Notes have been added to two sub-paragraphs of the "Page-Directory and Page-Table Entries" section, Chapter 3, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*. The notes clarify a limitation on the processor's Self-Modifying Code detection logic in the Accessed (A) flag and Dirty (D) flag context.

The applicable sections are reproduced below. See the change bars.

Accessed (A) flag, bit 5

Indicates whether a page or page table has been accessed (read from or written to) when set. Memory management software typically clears this flag when a page or page table is initially loaded into physical memory. The processor then sets this flag the first time a page or page table is accessed.

This flag is a "sticky" flag, meaning that once set, the processor does not implicitly clear it. Only software can clear this flag. The accessed and dirty flags are provided for use by memory management software to manage the transfer of pages and page tables into and out of physical memory.

NOTE: The accesses used by the processor to set this bit may or may not be exposed to the processor's Self-Modifying Code detection logic. If the processor is executing code from the same memory area that is being used for page table structures, the setting of the bit may or may not result in an immediate change to the executing code stream.

Dirty (D) flag, bit 6

Indicates whether a page has been written to when set. (This flag is not used in page-directory entries that point to page tables.) Memory management software typically clears this flag when a page is initially loaded into physical memory. The processor then sets this flag the first time a page is accessed for a write operation.

This flag is "sticky," meaning that once set, the processor does not implicitly clear it. Only software can clear this flag. The dirty and accessed flags are provided for use by memory management software to manage the transfer of pages and page tables into and out of physical memory.

NOTE: The accesses used by the processor to set this bit may or may not be exposed to the processor's Self-Modifying Code detection logic. If the processor is executing code from the same memory area that is being used for page table structures, the setting of the bit may or may not result in an immediate change to the executing code stream.

16. Interrupt 11 Discussion Concerning EXT Flag Functioning Has Been Updated

The Volume 3, Chapter 5, Interrupt 11: Error Code section has been updated. This is the second draft of this correction. The section now provides a more complete description of the EXT flag. The impacted text is reproduced below.

A not-present indication in a gate descriptor, however, does not indicate that a segment is not present (because gates do not correspond to segments). The operating system may use the present flag for gate descriptors to trigger exceptions of special significance to the operating system.

A contributory exception or page fault that subsequently referenced a not-present segment would cause a double fault (#DF) to be generated instead of #NP.

Exception Error Code

An error code containing the segment selector index for the segment descriptor that caused the violation is pushed onto the stack of the exception handler. If the EXT flag is set, it indicates that the exception resulted from either:

- an external event (NMI or INTR) that caused an interrupt, which subsequently referenced a not-present segment
- a benign exception that subsequently referenced a not-present segment

The IDT flag is set if the error code refers to an IDT entry. This occurs when the IDT entry for an interrupt being serviced references a not-present gate. Such an event could be generated by an INT instruction or a hardware interrupt.

17. Improved Information on Interpreting Machine-Check Error Codes

In Volume 3, Appendix E has been re-written to incorporate new IA32_MCi_STATUS data. This is the second draft of this correction. Impacted sections are reproduced below. Encoding of the model-specific and other information fields is different for the 06H and 0FH processor families. Differences are documented in the following sections.

E.1. DECODING FAMILY 06H SPECIFIC MACHINE ERROR CODES

Machine error code reporting by processor family 06H is based on values read from IA32_MC*i*_STATUS (Figure E-1).

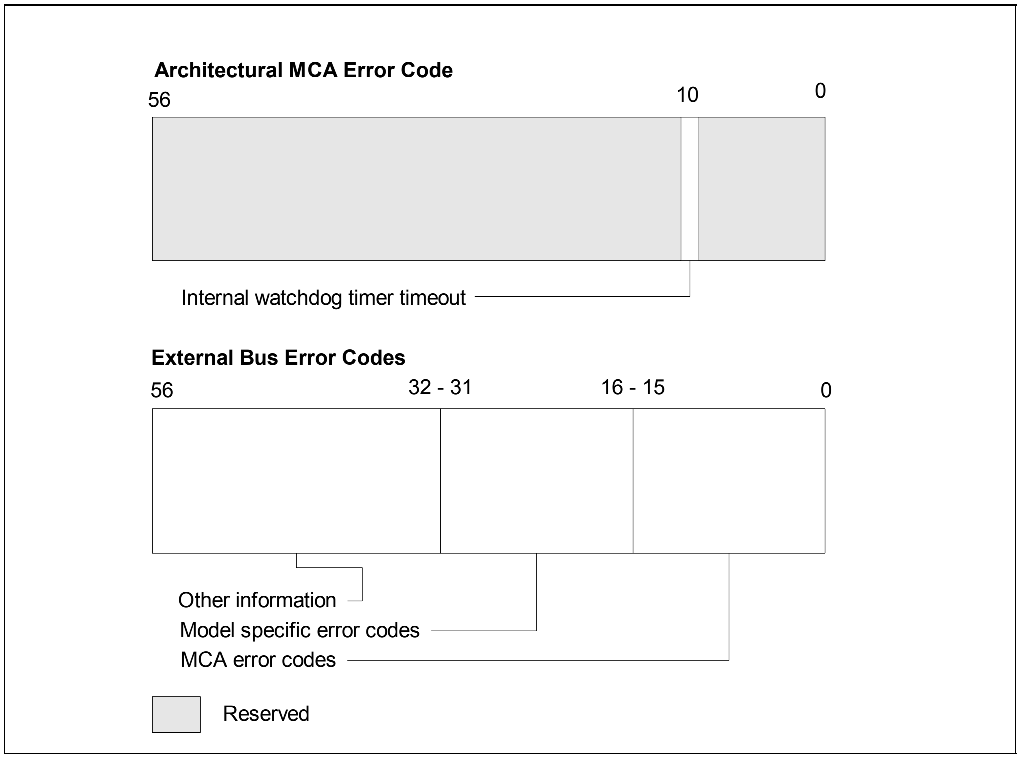


Figure E-1. IA32_MC*i*_STATUS Encoding for Family 06H

Table E-1 shows how to interpret internal watchdog timer timeout machine-check errors reported in IA32_MC*i*_STATUS for processor family 06H.

Table E-1. Family 06H Encoding of Internal Watchdog Timer Errors Reported in IA32_MC*i*_STATUS

	Bit No.	Bit Function	Bit Description
Architectural MCA error code	0-15	0000010000000000	Internal watchdog timer timeout. Note that a watch-dog timer time-out only occurs if the BINIT driver circuitry is enabled.
Model-specific error codes	16-31	Reserved	Reserved
Other information	32-56	Reserved	Reserved

Table E-2 shows how to interpret errors that occur on the external bus.

Table E-2. Family 06H Encoding 32_MC*i*_STATUS for External Bus Errors

Type	Bit No.	Bit Function	Bit Description
MCA error codes	0-1	Reserved	Reserved.
	2-3	For external bus errors: special cycle or I/O For internal timeout: Reserved	For external bus errors: • Bit 2 is set to 1 if the access was a special cycle. • Bit 3 is set to 1 if the access was a special cycle OR a I/O cycle. For internal timeout: Reserved

	4-7	For external bus errors: Read/Write
		
		For internal timeout: Reserved
			For external bus errors, 00WR:
			W = 1 for writes
			R = 1 for reads
			
			For internal timeout: Reserved

	8-9	Reserved	Reserved
	10-11	10 01	External bus errors Internal watchdog timer timeout
	12-15	Reserved	Reserved
Model specific errors	16-18	Reserved	Reserved

Table E-2. Family 06H Encoding 32_MC*i*_STATUS for External Bus Errors (Continued)

Type	Bit No.	Bit Function	Bit Description
Model specific errors	19-24	Bus queue request type	000000 for BQ_DCU_READ_TYPE error 000010 for BQ_IFU_DEMAND_TYPE error 000011 for BQ_IFU_DEMAND_NC_TYPE error 000100 for BQ_DCU_RFO_TYPE error 000101 for BQ_DCU_RFO_LOCK_TYPE error 000110 for BQ_DCU_ITOM_TYPE error 001000 for BQ_DCU_WB_TYPE error 001010 for BQ_DCU_WCEVICT_TYPE error 001011 for BQ_DCU_WCLINE_TYPE error 001100 for BQ_DCU_BTM_TYPE error 001101 for BQ_DCU_INTACK_TYPE error 001110 for BQ_DCU_INVALL2_TYPE error 001111 for BQ_DCU_FLUSHL2_TYPE error 010000 for BQ_DCU_PART_RD_TYPE error 010010 for BQ_DCU_PART_WR_TYPE error 010100 for BQ_DCU_SPEC_CYC_TYPE error 011000 for BQ_DCU_IO_RD_TYPE error 011001 for BQ_DCU_IO_WR_TYPE error 011100 for BQ_DCU_LOCK_RD_TYPE error 011110 for BQ_DCU_SPLOCK_RD_TYPE error 011101 for BQ_DCU_LOCK_WR_TYPE error 000010 for BQ_IFU_DEMAND_TYPE error 000011 for BQ_IFU_DEMAND_NC_TYPE error 000100 for BQ_DCU_RFO_TYPE error 000101 for BQ_DCU_RFO_LOCK_TYPE error 000110 for BQ_DCU_ITOM_TYPE error 001000 for BQ_DCU_WB_TYPE error 001010 for BQ_DCU_WCEVICT_TYPE error 001011 for BQ_DCU_WCLINE_TYPE error 001100 for BQ_DCU_BTM_TYPE error 001101 for BQ_DCU_INTACK_TYPE error 001110 for BQ_DCU_INVALL2_TYPE error 001111 for BQ_DCU_FLUSHL2_TYPE error 010000 for BQ_DCU_PART_RD_TYPE error 010010 for BQ_DCU_PART_WR_TYPE error 010100 for BQ_DCU_SPEC_CYC_TYPE error 011000 for BQ_DCU_IO_RD_TYPE error 011001 for BQ_DCU_IO_WR_TYPE error 011100 for BQ_DCU_LOCK_RD_TYPE error 011110 for BQ_DCU_SPLOCK_RD_TYPE error 011101 for BQ_DCU_LOCK_WR_TYPE error
Model specific errors	27-25	Bus queue error type	000 for BQ_ERR_HARD_TYPE error 001 for BQ_ERR_DOUBLE_TYPE error 010 for BQ_ERR_AERR2_TYPE error 100 for BQ_ERR_SINGLE_TYPE error 101 for BQ_ERR_AERR1_TYPE error

Table E-2. Family 06H Encoding 32_MC*i*_STATUS for External Bus Errors (Continued)

Type	Bit No.	Bit Function	Bit Description
Model specific errors	28	FRC error	1 if FRC error active
	29	BERR	1 if BERR is driven
	30	Internal BINIT	1 if BINIT driven for this processor
	31	Reserved	Reserved
Other information	32-34	Reserved	Reserved
	35	External BINIT	1 if BINIT is received from external bus.
	36	RESPONSE PARITY ERROR	This bit is asserted in IA32_MC*i*_STATUS if this component has received a parity error on the RS[2:0]# pins for a response transaction. The RS signals are checked by the RSP# external pin.
	37	BUS BINIT	This bit is asserted in IA32_MC*i*_STATUS if this component has received a hard error response on a split transaction (one access that has needed to be split across the 64-bit external bus interface into two accesses).
	38	TIMEOUT BINIT	This bit is asserted in IA32_MC*i*_STATUS if this component has experienced a ROB time-out, which indicates that no micro-instruction has been retired for a predetermined period of time. A ROB time-out occurs when the 15-bit ROB time-out counter carries a 1 out of its high order bit. The timer is cleared when a micro-instruction retires, an exception is detected by the core processor, RESET is asserted, or when a ROB BINIT occurs. The ROB time-out counter is prescaled by the 8-bit PIC timer which is a divide by 128 of the bus clock (the bus clock is 1:2, 1:3, 1:4 of the core clock). When a carry out of the 8-bit PIC timer occurs, the ROB counter counts up by one. While this bit is asserted, it cannot be overwritten by another error.
	39-41	Reserved	Reserved
	42	HARD ERROR	This bit is asserted in IA32_MC*i*_STATUS if this component has initiated a bus transactions which has received a hard error response. While this bit is asserted, it cannot be overwritten.
	43	IERR	This bit is asserted in IA32_MC*i*_STATUS if this component has experienced a failure that causes the IERR pin to be asserted. While this bit is asserted, it cannot be overwritten.

Table E-2. Family 06H Encoding 32_MC*i*_STATUS for External Bus Errors (Continued)

Type	Bit No.	Bit Function	Bit Description
Other information	44	AERR	This bit is asserted in IA32_MC*i*_STATUS if this component has initiated 2 failing bus transactions which have failed due to Address Parity Errors (AERR asserted). While this bit is asserted, it cannot be overwritten.
	45	UECC	The Uncorrectable ECC error bit is asserted in IA32_MC*i*_STATUS for uncorrected ECC errors. While this bit is asserted, the ECC syndrome field will not be overwritten.
	46	CECC	The correctable ECC error bit is asserted in IA32_MC*i*_STATUS for corrected ECC errors.
	47-54	ECC syndrome	The ECC syndrome field in IA32_MC*i*_STATUS contains the 8-bit ECC syndrome only if the error was a correctable/uncorrectable ECC error and there wasn't a previous valid ECC error syndrome logged in IA32_MC*i*_STATUS. A previous valid ECC error in IA32_MC*i*_STATUS is indicated by IA32_MC*i*_STATUS.bit45 (uncorrectable error occurred) being asserted. After processing an ECC error, machine-check handling software should clear IA32_MC*i*_STATUS.bit45 so that future ECC error syndromes can be logged.
	55-56	Reserved	Reserved.

E.2. DECODING FAMILY 0FH SPECIFIC MACHINE ERROR CODES

Machine error code reporting by processor family 0FH is also based on values read from IA32_MC*i*_STATUS (Figure E-2).

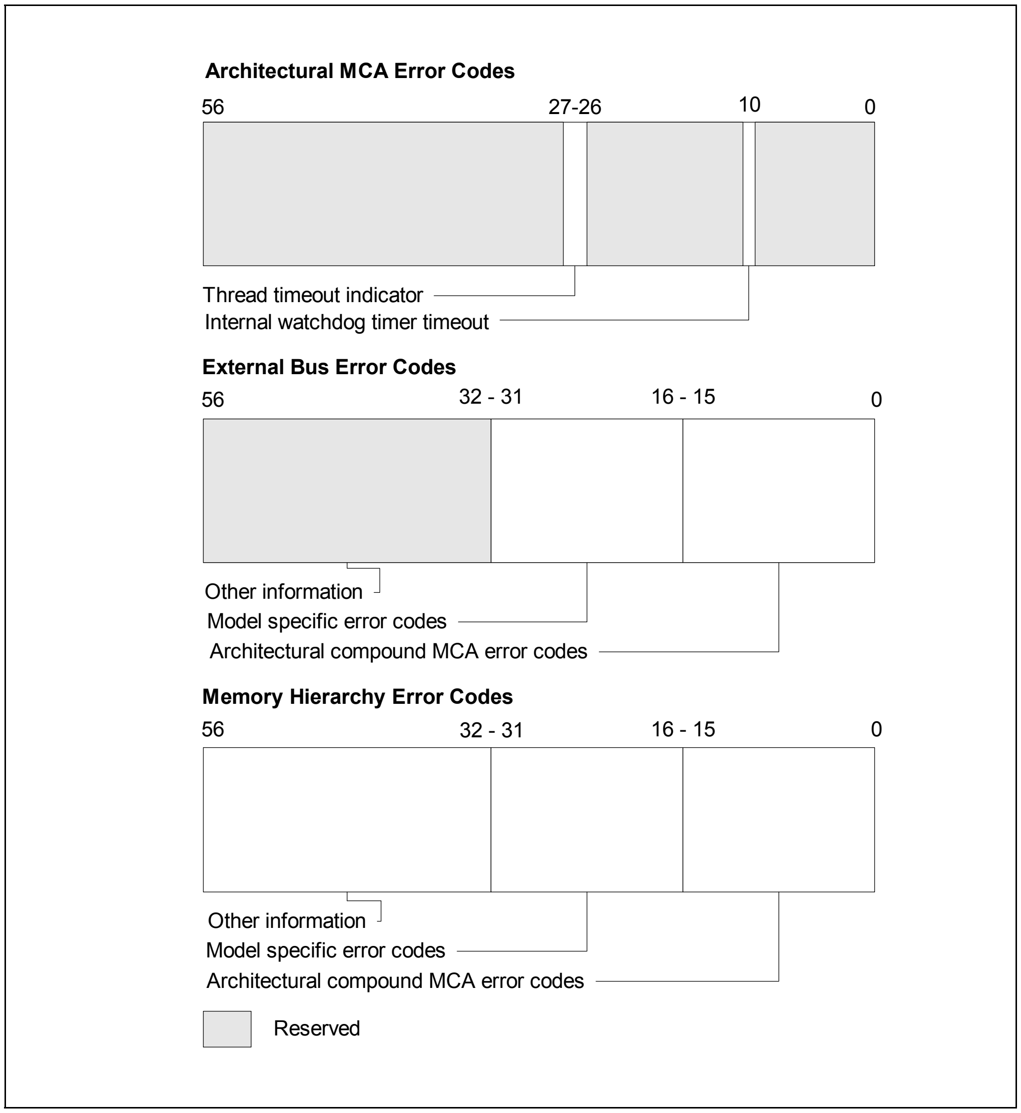

Architectural MCA Error Codes
56
27-26
10
0
Thread timeout indicator
Internal watchdog timer timeout
External Bus Error Codes
56
32 - 31
16 - 15
0
Other information
Model specific error codes
Architectural compound MCA error codes
Memory Hierarchy Error Codes
56
32 - 31
16 - 15
0
Other information
Model specific error codes
Architectural compound MCA error codes
Reserved

Figure E-2. IA32_MC*i*_STATUS Encoding for Family 0FH

Table E-3 provides information on how to interpret processor family 0FH error code fields for internal watchdog timer timeout machine-checks.

Table E-3. Family 0FH Encoding of IA32_MC*i*_STATUS for Internal Watchdog Timer Errors

	Bit No.	Bit Function	Bit Description
Architectural MCA error code	0-15	0000010000000000	Internal watchdog timer timeout. Note that a watch-dog timer time-out only occurs if the BINIT driver circuitry is enabled.
Model-specific error code	16-25	Reserved	Reserved
	26-27	Thread timeout indicator (TT)	Contains the indication of the thread which timed out: 01 - Thread 0 timed out 10 - Thread 1 timed out 11 - Both threads timed out
	28-31	Reserved	Reserved
Other information	32-56	Reserved	Reserved

Table E-4 provides the information to interpret errors that occur on the external bus. Note that processor family 0FH uses the compound MCA code format for external bus errors. Refer to *Chapter 14, Machine-Check Architecture* for more information.

Table E-4. Family 0FH Encoding of IA32_MC*i*_STATUS for External Bus Errors

	Bit No.	Bit Function	Bit Description
Architectural compound MCA error codes	0-1	Memory hierarchy level (LL)	Refer to Table 14-5 for detailed decoding of the memory hierarchy level (LL) sub-field.
	2-3	Memory and I/O (II)	Refer to Table 14-7 for a detailed decoding of the memory or IO (II) sub-field.
	4-7	Request (RRRR)	Refer to Table 14-6 for a detailed decoding of the request (RRRR) sub-field.
	8	Timeout (T)	Refer to Table 14-7 for a detailed decoding of the Timeout (T) Sub-Field.
	9-10	Participation (PP)	Refer to Table 14-7 for a detailed decoding of the participation (PP) sub-field.
	11-15	00001	Bus and interconnect errors
Model-specific error codes	16	FSB address parity	Address parity error detected: 1 = Address parity error detected 0 = No address parity error
	17	Response hard fail	Hardware failure detected on response
	18	Response parity	Parity error detected on response
	19	PIC and FSB data parity	Data Parity detected on either PIC or FSB access
	20	Processor Signature = 00000F04H: Invalid PIC request All other processors: Reserved	Processor Signature = 00000F04H. Indicates error due to an invalid PIC request (access was made to PIC space with WB memory): 1 = Invalid PIC request error 0 = No Invalid PIC request error Reserved
	21	Pad state machine	The state machine that tracks P and N data-strobe relative timing has become unsynchronized or a glitch has been detected.
	22	Pad strobe glitch	Data strobe glitch
	23	Pad address glitch	Address strobe glitch
	24-31	Reserved	Reserved
Other Information	32-56	Reserved	Reserved

Table E-5 provides information on how to interpret errors that occur within the memory hierarchy.

Table E-5. Family 0FH Encoding of IA32_MC*i*_STATUS for Memory Hierarchy Errors

	Bit No.	**Bit Function**	**Bit Description**
Architectural compound MCA error code	0-1	Memory Hierarchy Level (LL)	Refer to Table 14-5 for a detailed decoding of the memory hierarchy level (LL) sub-field.
	2-3	Transaction Type (TT)	Refer to Table 14-5 for a detailed decoding of the transaction type (TT) sub-field.
	4-7	Request (RRRR)	Refer to Table 14-6 for a detailed decoding of the request type (RRRR) sub-field.
	8-15	00000001	Memory hierarchy error format
Model specific error codes	16-17	Tag Error Code	Contains the tag error code for this machine check error: 00 = No error detected 01 = Parity error on tag miss with a clean line 10 = Parity error/multiple tag match on tag hit 11 = Parity error/multiple tag match on tag miss
	18-19	Data Error Code	Contains the data error code for this machine check error: 00 = No error detected 01 = Single bit error 10 = Double bit error on a clean line 11 = Double bit error on a modified line
	20	L3 Error	This bit is set if the machine check error originated in the L3 (it can be ignored for invalid PIC request errors): 1 = L3 error 0 = L2 error
	21	Invalid PIC Request	Indicates error due to invalid PIC request (access was made to PIC space with WB memory): 1 = Invalid PIC request error 0 = No invalid PIC request error
	22-31	Reserved	Reserved
Other Information	32-39	8-bit Error Count	Holds a count of the number of errors since reset. The counter begins at 0 for the first error and saturates at a count of 254.
	40-56	Reserved	Reserved

18. More information on the Functioning of Debug BPs after POP SS/MOV SS Has Been Provided

In Volume 3, Section 15.3.1.1; more information has been provided on the functioning of code instruction breakpoints immediately after POP SS/MOV SS instructions. This data is reprinted below (in context). Footnotes have been added to the POP and MOV sections in Volume 2 of the IA-32 Intel Architecture Software Developer's Manual which contain the same information for POP SS/ MOV SS (the footnotes are not reproduced here).

15.3.1.1. INSTRUCTION-BREAKPOINT EXCEPTION CONDITION

The processor reports an instruction breakpoint when it attempts to execute an instruction at an address specified in a breakpoint-address register (DB0 through DR3) that has been set up to detect instruction execution (R/W flag is set to 0). Upon reporting the instruction breakpoint, the processor generates a fault-class, debug exception (#DB) before it executes the target instruction for the breakpoint.

Instruction breakpoints are the highest priority debug exceptions. They are serviced before any other exceptions detected during the decoding or execution of an instruction. Note, however, that if a code instruction breakpoint is placed on an instruction located immediately after a POP SS/MOV SS instruction, it may not be triggered. In most situations, POP SS/MOV SS will inhibit such interrupts (see "MOV-Move" and "POP-Pop a Value from the Stack" in the *IA-32 Intel Architecture Software Developer's Manual, Volume 2*).

19. More Information on the LBR Stack Has Been Provided

The following information has been added to Volume 3 of the IA-32 Intel Architecture Software Developer's Manual, Section 15.5. This information describes the LBR stack and MSR_LASTBRANCH_TOS

- Last Branch Record (LBR) Stack — The LBR stack is a circular stack that consists of four MSRs (MSR_LASTBRANCH_0 through MSR_LASTBRANCH_3) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, models 0H-02H]. The LBR stack consists of 16 MSR pairs (MSR_LASTBRANCH_0_FROM_LIP through MSR_LASTBRANCH_15_FROM_LIP and MSR_LASTBRANCH_0_TO_LIP through MSR_LASTBRANCH_15_TO_LIP) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, model 03H].
- Last Branch Record Top-of-Stack (TOS) Pointer — The TOS Pointer MSR contains a 2-bit pointer (0-3) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, models 0H-02H]. This pointer becomes a 4-bit pointer (0-15) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, model 03H].

See also: Table 15-2, Figure 15-3, and Figure 15-4 below.

Table 15-2. LBR MSR Stack Structure for the Pentium 4 and Intel Xeon Processor Family

LBR MSRs for Family 0FH, Models 0H-02H; MSRs at locations 1DBH-1DEH.	**Decimal Value of TOS Pointer in MSR_LASTBRANCH_TOS (bits 0-1)**
MSR_LASTBRANCH_0	0
MSR_LASTBRANCH_1	1
MSR_LASTBRANCH_2	2
MSR_LASTBRANCH_3	3
LBR MSRs for Family 0FH, Models; MSRs at locations 680H-68FH.	**Decimal Value of TOS Pointer in MSR_LASTBRANCH_TOS (bits 0-3)**
MSR_LASTBRANCH_0_FROM_LIP	0
MSR_LASTBRANCH_1_FROM_LIP	1
MSR_LASTBRANCH_2_FROM_LIP	2
MSR_LASTBRANCH_3_FROM_LIP	3
MSR_LASTBRANCH_4_FROM_LIP	4
MSR_LASTBRANCH_5_FROM_LIP	5
MSR_LASTBRANCH_6_FROM_LIP	6
MSR_LASTBRANCH_7_FROM_LIP	7
MSR_LASTBRANCH_8_FROM_LIP	8
MSR_LASTBRANCH_9_FROM_LIP	9
MSR_LASTBRANCH_10_FROM_LIP	10
MSR_LASTBRANCH_11_FROM_LIP	11
MSR_LASTBRANCH_12_FROM_LIP	12
MSR_LASTBRANCH_13_FROM_LIP	13
MSR_LASTBRANCH_14_FROM_LIP	14
MSR_LASTBRANCH_15_FROM_LIP	15
LBR MSRs for Family 0FH, Model 03H; MSRs at locations 6C0H-6CFH.	
MSR_LASTBRANCH_0_TO_LIP	0
MSR_LASTBRANCH_1_TO_LIP	1
MSR_LASTBRANCH_2_TO_LIP	2
MSR_LASTBRANCH_3_TO_LIP	3
MSR_LASTBRANCH_4_TO_LIP	4
MSR_LASTBRANCH_5_TO_LIP	5
MSR_LASTBRANCH_6_TO_LIP	6
MSR_LASTBRANCH_7_TO_LIP	7
MSR_LASTBRANCH_8_TO_LIP	8
MSR_LASTBRANCH_9_TO_LIP	9
MSR_LASTBRANCH_10_TO_LIP	10
MSR_LASTBRANCH_11_TO_LIP	11
MSR_LASTBRANCH_12_TO_LIP	12
MSR_LASTBRANCH_13_TO_LIP	13
MSR_LASTBRANCH_14_TO_LIP	14
MSR_LASTBRANCH_15_TO_LIP	15

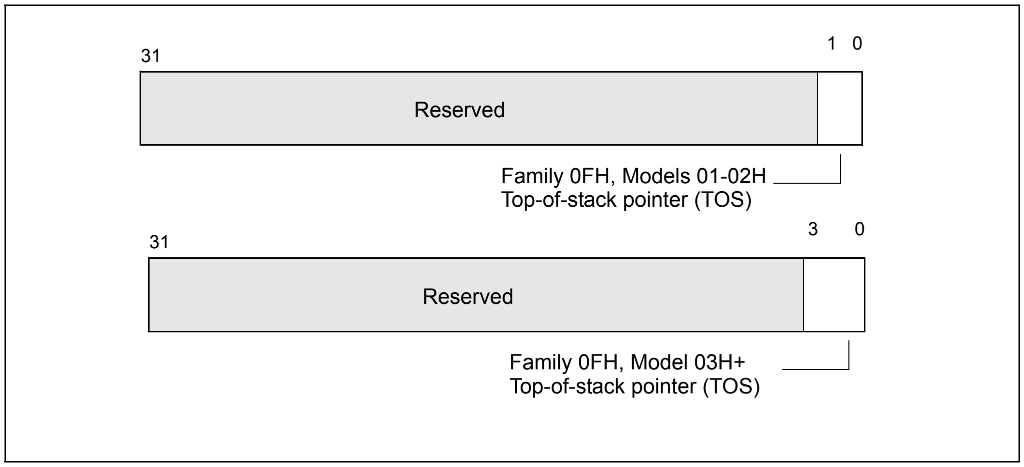

31
1 0
Reserved
Family 0FH, Models 01-02H
Top-of-stack pointer (TOS)
31
3 0
Reserved
Family 0FH, Model 03H+
Top-of-stack pointer (TOS)

Figure 15-3. MSR_LASTBRANCH_TOS MSR Layout for the Pentium 4 and Intel Xeon Processor Family

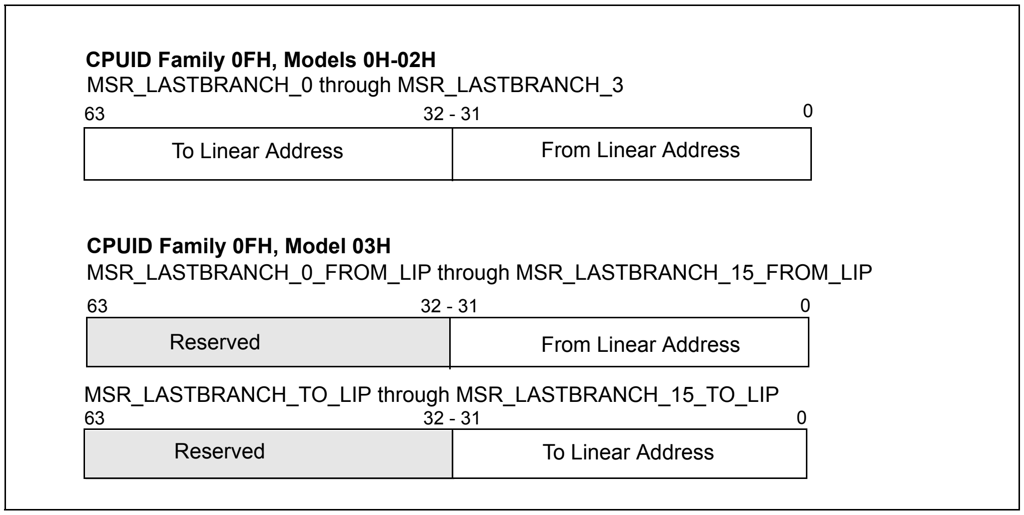

CPUID Family 0FH, Models 0H-02H
MSR_LASTBRANCH_0 through MSR_LASTBRANCH_3
63
32 - 31
0
To Linear Address
From Linear Address
CPUID Family 0FH, Model 03H
MSR_LASTBRANCH_0_FROM_LIP through MSR_LASTBRANCH_15_FROM_LIP
63
32 - 31
0
Reserved
From Linear Address
MSR_LASTBRANCH_TO_LIP through MSR_LASTBRANCH_15_TO_LIP
63
32 - 31
0
Reserved
To Linear Address

Figure 15-4. LBR MSR Branch Record Layout for the Pentium 4 and Intel Xeon Processor Family

Volume 3, Appendix B, Table B-1 has also been updated to reflect new LBR stack information. Impacted cells are reproduced below.

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

Register Address		Register Name Fields and Flags	Model Availability	Shared/ Unique[1]	Bit Description
Hex	Dec				
1DAH	474	MSR_LASTBRANCH _TOS	0, 1, 2, 3	Unique	**Last Branch Record Stack TOS.** (R) Contains an index (0-3 or 0-15) that points to the top of the last branch record stack (that is, that points the index of the MSR containing the most recent branch record.
1DBH	475	MSR_LASTBRANCH _0	0, 1, 2	Unique	**Last Branch Record 0.** (R/W) One of four last branch record registers on the last branch record stack. It contains pointers to the source and destination instruction for one of the last four branches, exceptions, or interrupts that the processor took. NOTE: MSR_LASTBRANCH_0 through MSR_LASTBRANCH_3 at 1DBH-1DEH are available only on family 0FH, models 0H-02H. They have been replaced by the MSRs at 680H-68FH and 6C0H-6CFH..
1DCH	476	MSR_LASTBRANCH _1	0, 1, 2	Unique	**Last Branch Record 1.** See description of the MSR_LASTBRANCH_0 MSR at 1DBH.
1DDH	477	MSR_LASTBRANCH _2	0, 1, 2	Unique	**Last Branch Record 2.** See description of the MSR_LASTBRANCH_0 MSR at 1DBH.
1DEH	478	MSR_LASTBRANCH _3	0, 1, 2	Unique	**Last Branch Record 3.** See description of the MSR_LASTBRANCH_0 MSR at 1DBH.
680H	1664	MSR_LASTBRANCH _0_FROM_LIP	3	Unique	**Last Branch Record 0.** (R/W) One of 16 pairs of last branch record registers on the last branch record stack (680H-68FH). This part of the stack contains pointers to the **source instruction** for one of the last 16 branches, exceptions, or interrupts taken by the processor. NOTE: The MSRs at 680H-68FH, 6C0H-6CfH are not available in processor releases before family 0FH, model 03H. These MSRs replace MSRs previously located at 1DBH-1DEH.which performed the same function for early releases.
681H	1665	MSR_LASTBRANCH _1_FROM_LIP	3	Unique	**Last Branch Record 1.** See description of MSR_LASTBRANCH_0 at 680H.
682H	1666	MSR_LASTBRANCH _2_FROM_LIP	3	Unique	**Last Branch Record 2.** See description of MSR_LASTBRANCH_0 at 680H.

Table B-1. MSRs in the Pentium 4 and Intel Xeon Processors (Continued)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
683H	1667	MSR_LASTBRANCH _3_FROM_LIP	3	Unique	**Last Branch Record 3.** See description of MSR_LASTBRANCH_0 at 680H.
684H	1668	MSR_LASTBRANCH _4_FROM_LIP	3	Unique	**Last Branch Record 4.** See description of MSR_LASTBRANCH_0 at 680H.
685H	1669	MSR_LASTBRANCH _5_FROM_LIP	3	Unique	**Last Branch Record 5.** See description of MSR_LASTBRANCH_0 at 680H.
686H	1670	MSR_LASTBRANCH _6_FROM_LIP	3	Unique	**Last Branch Record 6.** See description of MSR_LASTBRANCH_0 at 680H.
687H	1671	MSR_LASTBRANCH _7_FROM_LIP	3	Unique	**Last Branch Record 7.** See description of MSR_LASTBRANCH_0 at 680H.
688H	1672	MSR_LASTBRANCH _8_FROM_LIP	3	Unique	**Last Branch Record 8.** See description of MSR_LASTBRANCH_0 at 680H.
689H	1673	MSR_LASTBRANCH _9_FROM_LIP	3	Unique	**Last Branch Record 9.** See description of MSR_LASTBRANCH_0 at 680H.
68AH	1674	MSR_LASTBRANCH _10_FROM_LIP	3	Unique	**Last Branch Record 10.** See description of MSR_LASTBRANCH_0 at 680H.
68BH	1675	MSR_LASTBRANCH _11_FROM_LIP	3	Unique	**Last Branch Record 11.** See description of MSR_LASTBRANCH_0 at 680H.
68CH	1676	MSR_LASTBRANCH _12_FROM_LIP	3	Unique	**Last Branch Record 12.** See description of MSR_LASTBRANCH_0 at 680H.
68DH	1677	MSR_LASTBRANCH _13_FROM_LIP	3	Unique	**Last Branch Record 13.** See description of MSR_LASTBRANCH_0 at 680H.
68EH	1678	MSR_LASTBRANCH _14_FROM_LIP	3	Unique	**Last Branch Record 14.** See description of MSR_LASTBRANCH_0 at 680H.
68FH	1679	MSR_LASTBRANCH _15_FROM_LIP	3	Unique	**Last Branch Record 15.** See description of MSR_LASTBRANCH_0 at 680H.
6C0H	1728	MSR_LASTBRANCH _0_TO_LIP	3	Unique	**Last Branch Record 0.** (R/W) One of 16 pairs of last branch record registers on the last branch record stack (6C0H-6CFH). This part of the stack contains pointers to the **destination instruction** for one of the last 16 branches, exceptions, or interrupts that the processor took.

Table B-1. MSRs in the Pentium 4 and Intel Xeon Processors (Continued)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
6C1H	1729	MSR_LASTBRANCH _1_TO_LIP	3	Unique	**Last Branch Record 1.** See description of MSR_LASTBRANCH_0 at 6C0H.
6C2H	1730	MSR_LASTBRANCH _2_TO_LIP	3	Unique	**Last Branch Record 2.** See description of MSR_LASTBRANCH_0 at 6C0H.
6C3H	1731	MSR_LASTBRANCH _3_TO_LIP	3	Unique	**Last Branch Record 3.** See description of MSR_LASTBRANCH_0 at 6C0H.
6C4H	1732	MSR_LASTBRANCH _4_TO_LIP	3	Unique	**Last Branch Record 4.** See description of MSR_LASTBRANCH_0 at 6C0H.
6C5H	1733	MSR_LASTBRANCH _5_TO_LIP	3	Unique	**Last Branch Record 5.** See description of MSR_LASTBRANCH_0 at 6C0H.
6C6H	1734	MSR_LASTBRANCH _6_TO_LIP	3	Unique	**Last Branch Record 6.** See description of MSR_LASTBRANCH_0 at 6C0H.
6C7H	1735	MSR_LASTBRANCH _7_TO_LIP	3	Unique	**Last Branch Record 7.** See description of MSR_LASTBRANCH_0 at 6C0H.
6C8H	1736	MSR_LASTBRANCH _8_TO_LIP	3	Unique	**Last Branch Record 8.** See description of MSR_LASTBRANCH_0 at 6C0H.
6C9H	1737	MSR_LASTBRANCH _9_TO_LIP	3	Unique	**Last Branch Record 9.** See description of MSR_LASTBRANCH_0 at 6C0H.
6CAH	1738	MSR_LASTBRANCH _10_TO_LIP	3	Unique	**Last Branch Record 10.** See description of MSR_LASTBRANCH_0 at 6C0H.
6CBH	1739	MSR_LASTBRANCH _11_TO_LIP	3	Unique	**Last Branch Record 11.** See description of MSR_LASTBRANCH_0 at 6C0H.
6CCH	1740	MSR_LASTBRANCH _12_TO_LIP	3	Unique	**Last Branch Record 12.** See description of MSR_LASTBRANCH_0 at 6C0H.
6CDH	1741	MSR_LASTBRANCH _13_TO_LIP	3	Unique	**Last Branch Record 13.** See description of MSR_LASTBRANCH_0 at 6C0H.
6CEH	1742	MSR_LASTBRANCH _14_TO_LIP	3	Unique	**Last Branch Record 14.** See description of MSR_LASTBRANCH_0 at 6C0H.
6CFH	1743	MSR_LASTBRANCH _15_TO_LIP	3	Unique	**Last Branch Record 15.** See description of MSR_LASTBRANCH_0 at 6C0H.

20. Limited Availability of Two MSR's Have Been Documented

A note has been added to Volume 3, Chapter 15, Table 15-4. The note indicates the availability of MSR_IQ_ESCR0 and MSR_IQ_ESCR1. The impacted table cells are reproduced below.

Counter			CCCR		ESCR		
Name	**No.**	**Addr**	**Name**	**Addr**	**Name**	**No.**	**Addr**
MSR_IQ_COUNTER0	12	30CH	MSR_IQ_CCCR0	36CH	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH
MSR_IQ_COUNTER1	13	30DH	MSR_IQ_CCCR1	36DH	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH
MSR_IQ_COUNTER2	14	30EH	MSR_IQ_CCCR2	36EH	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH
MSR_IQ_COUNTER3	15	30FH	MSR_IQ_CCCR3	36FH	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH
MSR_IQ_COUNTER4	16	310H	MSR_IQ_CCCR4	370H	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH
MSR_IQ_COUNTER5	17	311H	MSR_IQ_CCCR5	371H	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH

[1] MSR_IQ_ESCR0 and MSR_IQ_ESCR1 are available only on early processor builds (family 0FH, models 01H-02H). These MSRs are not available on later versions.

21. The Microcode Update Facilities Section Has Been Updated

Volume 3 of the IA-32 Intel Architecture Software Developer's Manual, Section 9.11 has been updated. The new information has been added that documents the microcode update facilities added on new processors.

9.11. MICROCODE UPDATE FACILITIES

The Pentium 4, Intel Xeon, and P6 family processors have the capability to correct errata by loading an Intel-supplied data block into the processor. The data block is called a microcode update. This section describes the mechanisms the BIOS needs to provide in order to use this feature during system initialization. It also describes a specification that permits the incorporation of future updates into a system BIOS.

Intel considers the release of a microcode update for a silicon revision to be the equivalent of a processor stepping and completes a full-stepping level validation for releases of microcode updates.

A microcode update is used to correct errata in the processor. The BIOS, which has an update loader, is responsible for loading the update on processors during system initialization (Figure 9-7). There are two steps to this process: the first is to incorporate the necessary update data blocks into the BIOS; the second is to load update data blocks into the processor.

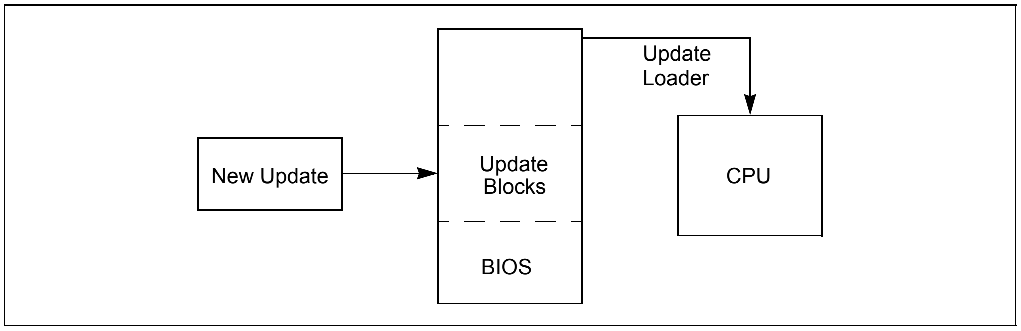


Figure 9-7. Applying Microcode Updates

9.11.1. Microcode Update

A microcode update consists of an Intel-supplied binary that contains a descriptive header and data. No executable code resides within the update. Each microcode update is tailored for a specific list of processor signatures. A mismatch of the processor's signature with the signature contained in the update will result in a failure to load. A processor signature includes the extended family, extended model, type, family, model, and stepping of the processor (starting with processor family 0fH, model 03H, a given microcode update may be associated with one of multiple processor signatures; see Section 9.11.2. for detail).

Microcode updates are composed of a multi-byte header, followed by encrypted data and then by an optional extended signature table. Table 9-1 provides a definition of the fields; Table 9-2 shows the format of an update.

The header is 48 bytes. The first 4 bytes of the header contain the header version. The update header and its reserved fields are interpreted by software based upon the header version. An encoding scheme guards against tampering and provides a means for determining the authenticity of any given update. For microcode updates with a data size field equal to 00000000H, the size of the microcode update is 2048 bytes. The first 48 bytes contain the microcode update header. The remaining 2000 bytes contain encrypted data.

For microcode updates with a data size not equal to 00000000H, the total size field specifies the size of the microcode update. The first 48 bytes contain the microcode update header. The second part of the microcode update is the encrypted data. The data size field of the microcode update header specifies the encrypted data size, its value must be a multiple of the size of DWORD. The optional extended signature table if implemented follows the encrypted data, and its size is calculated by (Total Size – (Data Size + 48)).

NOTE

The optional extended signature table is supported starting with processor family 0FH, model 03H.

.

Table 9-1. Microcode Update Field Definitions

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

Table 9-1. Microcode Update Field Definitions (Continued)

Field Name	Offset (bytes)	Length (bytes)	Description
Reserved	36	12	Reserved fields for future expansion
Update Data	48	Data Size or 2000	Update data
Extended Signature Count	Data Size + 48	4	Specifies the number of extended signature structures (Processor Signature[n], processor flags[n] and checksum[n]) that exist in this microcode update.
Extended Checksum	Data Size + 52	4	Checksum of update extended processor signature table. Used to verify the integrity of the extended processor signature table. Checksum is correct when the summation of the DWORDs that comprise the extended processor signature table results in 00000000H.
Reserved	Data Size + 56	12	Reserved fields
Processor Signature[n]	Data Size + 68 + (n * 12)	4	*Extended family, extended model, type, family, model*, and *stepping* of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, *type, family, model*, and *stepping* of the processor. The BIOS uses the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.
Processor Flags[n]	Data Size + 72 + (n * 12)	4	Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. The BIOS uses the processor flags field in conjunction with the platform Id bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.
Checksum[n]	Data Size + 76 + (n * 12)	4	Used by utility software to decompose a microcode update into multiple microcode updates where each of the new microcode updates is constructed without the optional extended processor signature table.

Table 9-2. Microcode Update Format

31	24	16				8		0	Bytes
Header Version									0
Update Revision									4
Month: 8	Day: 8	Year: 16							8
Processor Signature (CPUID)									12
Res: 4	Extended Family: 8	Extended Mode: 4	Reserved: 2	Type: 2	Family: 4	Model: 4		Stepping: 4	
Checksum									16

Table 9-2. Microcode Update Format (Continued)

31 24 16 8 0	Bytes
Loader Revision	20
Processor Flags	24
Reserved (24 bits) | P7 | P6 | P5 | P4 | P3 | P2 | P1 | P0	
Data Size	28
Total Size	32
Reserved (12 Bytes)	36
Update Data (Data Size bytes, or 2000 Bytes if Data Size = 00000000H)	48
Extended Signature Count 'n'	Data Size + 48
Extended Processor Signature Table Checksum	Data Size + 52
Reserved (12 Bytes)	Data Size + 56
Processor Signature[n]	Data Size + 68 + (n * 12)
Processor Flags[n]	Data Size + 72 + (n * 12)
Checksum[n]	Data Size + 76 + (n * 12)

9.11.2. Optional Extended Signature Table

The extended signature table is a structure that may be appended to the end of the encrypted data when the encrypted data only supports a single processor signature (optional case). The extended signature table will always be present when the encrypted data supports multiple processor steppings and/or models (required case).

The extended signature table consists of a 20-byte extended signature header structure, which contains the extended signature count, the extended processor signature table checksum, and 12 reserved bytes (Table 9-3). Following the extended signature header structure, the extended signature table contains 0-to-n extended processor signature structures.

Each processor signature structure consist of the processor signature, processor flags, and a checksum (Table 9-4).

The extended signature count in the extended signature header structure indicates the number of processor signature structures that exist in the extended signature table.

The extended processor signature table checksum is a checksum of all DWORDs that comprise the extended signature table. That includes the extended signature count, extended processor signature table checksum, 12 reserved bytes and the n processor signature structures. A valid extended signature table exists when the result of a DWORD checksum is 00000000H.

Table 9-3. Extended Processor Signature Table Header Structure

Extended Signature Count ‘n’	Data Size + 48
Extended Processor Signature Table Checksum	Data Size + 52
Reserved (12 Bytes)	Data Size + 56

Table 9-4. Processor Signature Structure

Processor Signature[n]	Data Size + 68 + (n * 12)
Processor Flags[n]	Data Size + 72 + (n * 12)
Checksum[n]	Data Size + 76 + (n * 12)

9.11.3. Processor Identification

Each microcode update is designed to for a specific processor or set of processors. To determine the correct microcode update to load, software must ensure that one of the processor signatures embedded in the microcode update matches the 32-bit processor signature returned by the CPUID instruction when executed by the target processor with EAX = 1. Attempting to load a microcode update that does not match a processor signature embedded in the microcode update with the processor signature returned by CPUID will cause the processor to reject the update.

Example 9-1 shows how to check for a valid processor signature match between the processor and microcode update.

Example 9-1. Pseudo Code to Validate the Processor Signature

```
ProcessorSignature ← CPUID(1):EAX

If (Update.HeaderVersion == 00000001h)
{
        // first check the ProcessorSignature field
    If (ProcessorSignature == Update.ProcessorSignature)
        Success

        // if extended signature is present
    Else If (Update.TotalSize > (Update.DataSize + 48))
    {

        //
        // Assume the Data Size has been used to calculate the
        // location of Update.ProcessorSignature[0].
        //

        For (N ← 0; ((N < Update.ExtendedSignatureCount) AND
            (ProcessorSignature != Update.ProcessorSignature[N])); N++);

            // if the loops ended when the iteration count is
            // less than the number of processor signatures in
            // the table, we have a match
        If (N < Update.ExtendedSignatureCount)
            Success
        Else
            Fail
        }
        Else
            Fail
    Else
        Fail
```

9.11.4. Platform Identification

In addition to verifying the processor signature, the intended processor platform type must be determined to properly target the microcode update. The intended processor platform type is determined by reading the IA32_PLATFORM_ID register, (MSR 17H). This 64-bit register must be read using the RDMSR instruction.

The three platform ID bits, when read as a binary coded decimal (BCD) number, indicate the bit position in the microcode update header's processor flags field associated with the installed processor. The processor flags in the 48-byte header and the processor flags field associated with the extended processor signature structures may have multiple bits set. Each set bit represents a different platform ID that the update supports.

Register Name:	**IA32_PLATFORM_ID**
MSR Address:	017H
Access:	Read Only

IA32_PLATFORM_ID is a 64-bit register accessed only when referenced as a Qword through a RDMSR instruction.

Table 9-5. Processor Flags

Bit	Descriptions
63:53	Reserved
52:50	Platform Id Bits (RO). The field gives information concerning the intended platform for the processor. See also Table 9-2. **52** **51** **50** 0 0 0 Processor Flag 0 0 0 1 Processor Flag 1 0 1 0 Processor Flag 2 0 1 1 Processor Flag 3 1 0 0 Processor Flag 4 1 0 1 Processor Flag 5 1 1 0 Processor Flag 6 1 1 1 Processor Flag 7
49:0	Reserved

To validate the platform information, software may implement an algorithm similar to the algorithms in Example 9-2.

Example 9-2. Pseudo Code Example of Processor Flags Test

```
Flag ← 1 << IA32_PLATFORM_ID[52:50]

If (Update.HeaderVersion == 00000001h)
{
    If (Update.ProcessorFlags & Flag)
    {
        Load Update
    }
    Else
    {

        //
        // Assume the Data Size has been used to calculate the
        // location of Update.ProcessorSignature[N] and a match
        // on Update.ProcessorSignature[N] has already succeeded
        //

        If (Update.ProcessorFlags[n] & Flag)
        {
            Load Update
        }
    }
}
```

9.11.5. Microcode Update Checksum

Each microcode update contains a DWORD checksum located in the update header. It is software's responsibility to ensure that a microcode update is not corrupt. To check for a corrupt microcode update, software must perform a unsigned DWORD (32-bit) checksum of the microcode update. Even though some fields are signed, the checksum procedure treats all DWORDs as unsigned. Microcode updates with a header version equal to 00000001H must sum all DWORDs that comprise the microcode update. A valid checksum check will yield a value of 00000000H. Any other value indicates the microcode update is corrupt and should not be loaded.

The checksum algorithm shown by the pseudo code in Example 9-3 treats the microcode update as an array of unsigned DWORDs. If the data size DWORD field at byte offset 32 equals 00000000H, the size of the encrypted data is 2000 bytes, resulting in 500 DWORDs. Otherwise the microcode update size in DWORDs = (*Total Size / 4*).

Example 9-3. Pseudo Code Example of Checksum Test

```
N ← 512

If (Update.DataSize != 00000000H)
    N ← Update.TotalSize / 4

ChkSum ← 0
For (I ← 0; I < N; I++)
{
    ChkSum ← ChkSum + MicrocodeUpdate[I]
}

If (ChkSum == 00000000H)
    Success
Else
    Fail
```

9.11.6. Microcode Update Loader

This section describes an update loader used to load an update into a Pentium 4, Intel Xeon, or P6 family processor. It also discusses the requirements placed on the BIOS to ensure proper loading. The update loader described contains the minimal instructions needed to load an update. The specific instruction sequence that is required to load an update is dependent upon the loader revision field contained within the update header. This revision is expected to change infrequently (potentially, only when new processor models are introduced).

Example 9-4 below represents the update loader with a loader revision of 00000001H. Note that the microcode update must be aligned on a 16-byte boundary.

Example 9-4. Assembly Code Example of Simple Microcode Update Loader

```
mov ecx,79h     ; MSR to read in ECX
xor eax,eax     ; clear EAX
xor ebx,ebx     ; clear EBX
mov ax,cs       ; Segment of microcode update
shl eax,4
mov bx,offset Update; Offset of microcode update
add eax,ebx     ; Linear Address of Update in EAX
add eax,48d     ; Offset of the Update Data within the Update
xor edx,edx     ; Zero in EDX
WRMSR           ; microcode update trigger
```

The loader shown in Example 9-4 assumes that *update* is the address of a microcode update (header and data) embedded within the code segment of the BIOS. It also assumes that the processor is operating in real mode. The data may reside anywhere in memory, aligned on a 16-byte boundary, that is accessible by the processor within its current operating mode (real, protected).

Before the BIOS executes the microcode update trigger (WRMSR) instruction, the following must be true:

- EAX contains the linear address of the start of the update data
- EDX contains zero
- ECX contains 79H (address of IA32_BIOS_UPDT_TRIG)

Other requirements are:

- If the update is loaded while the processor is in real mode, then the update data may not cross a segment boundary.
- If the update is loaded while the processor is in real mode, then the update data may not exceed a segment limit.
- If paging is enabled, pages that are currently present must map the update data.
- The microcode update data requires a 16-byte boundary alignment.

9.11.6.1. Hard Resets in Update Loading

The effects of a loaded update are cleared from the processor upon a hard reset. Therefore, each time a hard reset is asserted during the BIOS POST, the update must be reloaded on all processors that observed the reset. The effects of a loaded update are, however, maintained across a processor INIT. There are no side effects caused by loading an update into a processor multiple times.

9.11.6.2. Update in a Multiprocessor System

A multiprocessor (MP) system requires loading each processor with update data appropriate for its CPUID and platform ID bits. The BIOS is responsible for ensuring that this requirement is met and that the loader is located in a module executed by all processors in the system. If a system design permits multiple steppings of Pentium 4, Intel Xeon, and P6 family processors to exist concurrently; then the BIOS must verify individual processors against the update header information to ensure appropriate loading. Given these considerations, it is most practical to load the update during MP initialization.

9.11.6.3. Update in a System with Intel HT Technology

Intel Hyper-Threading Technology (HT Technology) has implications on the loading of the microcode update. The update must be loaded for each core in a physical processor. Thus, for a processor with HT Technology, only one logical processor per core is required to load the microcode update. Each individual logical processor can independently load the update. However, MP initialization must provide some mechanism (e.g. a software semaphore) to force serialization of microcode update loads and to prevent simultaneous load attempts to the same core.

9.11.6.4. Update Loader Enhancements

The update loader presented in Section 9.11.6., *Microcode Update Loader* is a minimal implementation that can be enhanced to provide additional functionality. Potential enhancements are described below:

- BIOS can incorporate multiple updates to support multiple steppings of the Pentium 4, Intel Xeon, and P6 family processors. This feature provides for operating in a mixed stepping environment on an MP system and enables a user to upgrade to a later version of the processor. In this case, modify the loader to check the CPUID and platform ID bits of the processor that it is running on against the available headers before loading a particular update. The number of updates is only limited by available BIOS space.
- A loader can load the update and test the processor to determine if the update was loaded correctly. See Section 9.11.7., *Update Signature and Verification*.
- A loader can verify the integrity of the update data by performing a checksum on the double words of the update summing to zero. See Section 9.11.5., *Microcode Update Checksum*.
- A loader can provide power-on messages indicating successful loading of an update.

9.11.7. Update Signature and Verification

The Pentium 4, Intel Xeon, and P6 family processors provide capabilities to verify the authenticity of a particular update and to identify the current update revision. This section describes the model-specific extensions of processors that support this feature. The update verification method below assumes that the BIOS will only verify an update that is more recent than the revision currently loaded in the processor.

CPUID returns a value in a model specific register in addition to its usual register return values. The semantics of CPUID cause it to deposit an update ID value in the 64-bit model-specific register at address 08BH (IA32_BIOS_SIGN_ID). If no update is present in the processor, the value in the MSR remains unmodified. The BIOS must pre-load a zero into the MSR before executing CPUID. If a read of the MSR at 8BH still returns zero after executing CPUID, this indicates that no update is present.

The update ID value returned in the EDX register after RDMSR executes indicates the revision of the update loaded in the processor. This value, in combination with the CPUID value returned in the EAX register, uniquely identifies a particular update. The signature ID can be directly compared with the update revision field in a microcode update header for verification of a correct load. No consecutive updates released for a given stepping of a processor may share the same signature. The processor signature returned by CPUID differentiates updates for different steppings.

9.11.7.1. Determining the Signature

An update that is successfully loaded into the processor provides a signature that matches the update revision of the currently functioning revision. This signature is available any time after the actual update has been loaded. Requesting the signature does not have a negative impact upon a loaded update.

The procedure for determining this signature shown in Example 9-5.

Example 9-5. Assembly Code to Retrieve the Update Revision

```
MOV    ECX, 08BH;IA32_BIOS_SIGN_ID
XOR    EAX, EAX;clear EAX
XOR    EDX, EDX;clear EDX
WRMSR  ;Load 0 to MSR at 8BH
MOV    EAX, 1
cpuid
MOV    ECX, 08BH;IA32_BIOS_SIGN_ID
rdmsr  ;Read Model Specific Register
```

If there is an update active in the processor, its revision is returned in the EDX register after the RDMSR instruction executes.

IA32_BIOS_SIGN_ID	**Microcode Update Signature Register**
MSR Address:	08BH Accessed as a Qword
Default Value:	XXXX XXXX XXXX XXXXh
Access:	Read/Write

The IA32_BIOS_SIGN_ID register is used to report the microcode update signature when CPUID executes. The signature is returned in the upper DWORD (Table 9-6).

Table 9-6. Microcode Update Signature

Bit	Description
63:32	Microcode update signature. This field contains the signature of the currently loaded microcode update when read following the execution of the CPUID instruction, function 1. It is required that this register field be pre-loaded with zero prior to executing the CPUID, function 1. If the field remains equal to zero, then there is no microcode update loaded. Another non-zero value will be the signature.
31:0	Reserved.

9.11.7.2. Authenticating the Update

An update may be authenticated by the BIOS using the signature primitive, described above, and the algorithm in Example 9-6.

Example 9-6. Pseudo Code to Authenticate the Update

```
Z ← Obtain Update Revision from the Update Header to be authenticated;
X ← Obtain Current Update Signature from MSR 8BH;

If (Z > X)
{
    Load Update that is to be authenticated;
    Y ← Obtain New Signature from MSR 8BH;

    If (Z == Y)
        Success
    Else
        Fail
}
Else
    Fail
```

Example 9-6 requires that the BIOS only authenticate updates that contain a numerically larger revision than the currently loaded revision, where Current Signature (X) < New Update Revision (Z). A processor with no loaded update is considered to have a revision equal to zero.

This authentication procedure relies upon the decoding provided by the processor to verify an update from a potentially hostile source. As an example, this mechanism in conjunction with other safeguards provides security for dynamically incorporating field updates into the BIOS.

9.11.8. Pentium 4, Intel Xeon, and P6 Family Processor Microcode Update Specifications

This section describes the interface that an application can use to dynamically integrate processor-specific updates into the system BIOS. In this discussion, the application is referred to as the calling program or caller.

The real mode INT15 call specification described here is an Intel extension to an OEM BIOS. This extension allows an application to read and modify the contents of the microcode update data in NVRAM. The update loader, which is part of the system BIOS, cannot be updated by the interface. All of the functions defined in the specification must be implemented for a system to be considered compliant with the specification. The INT15 functions are accessible only from real mode.

9.11.8.1. Responsibilities of the BIOS

If a BIOS passes the presence test (INT 15H, AX = 0D042H, BL = 0H), it must implement all of the sub-functions defined in the INT 15H, AX = 0D042H specification. There are no optional functions. BIOS must load the appropriate update for each processor during system initialization.

A Header Version of an update block containing the value 0FFFFFFFFH indicates that the update block is unused and available for storing a new update.

The BIOS is responsible for providing a region of non-volatile storage (NVRAM) for each potential processor stepping within a system. This storage unit consists of one or more update blocks. An update block is a contiguous 2048-byte block of memory. The BIOS for a single processor system need only provide update blocks to store one microcode update. If the BIOS for a multiple processor system is intended to support mixed processor steppings, then the BIOS needs to provide enough update blocks to store each unique microcode update or for each processor socket on the OEM's system board.

The BIOS is responsible for managing the NVRAM update blocks. This includes garbage collection, such as removing microcode updates that exist in NVRAM for which a corresponding processor does not exist in the system. This specification only provides the mechanism for ensuring security, the uniqueness of an entry, and that stale entries are not loaded. The actual update block management is implementation specific on a per-BIOS basis.

As an example, the BIOS may use update blocks sequentially in ascending order with CPU signatures sorted versus the first available block. In addition, garbage collection may be implemented as a setup option to clear all NVRAM slots or as BIOS code that searches and eliminates unused entries during boot.

NOTE

For IA-32 processors starting with family 0FH and model 03H, the microcode update may be as large as 16 KBytes. Thus, BIOS must allocate 8 update blocks for each microcode update. In a MP system, a common microcode update may be sufficient for each socket in the system.

For IA-32 processors earlier than family 0FH and model 03H, the microcode update is 2 KBytes. An MP-capable BIOS that supports multiple steppings must allocate a block for each socket in the system.

A single-processor BIOS that supports variable-sized microcode update and fixed-sized microcode update must allocate one 16 KByte region and a second region of at least 2 KBytes.

The following algorithm (Example 9-7) describes the steps performed during BIOS initialization used to load the updates into the processor(s). The algorithm assumes:

- The BIOS ensures that no update contained within NVRAM has a header version or loader version that does not match one currently supported by the BIOS.
- The update contains a correct checksum.
- The BIOS ensures that (at most) one update exists for each processor stepping.
- Older update revisions are not allowed to overwrite more recent ones.

These requirements are checked by the BIOS during the execution of the write update function of this interface. The BIOS sequentially scans through all of the update blocks in NVRAM starting with index 0. The BIOS scans until it finds an update where the processor fields in the header match the processor signature (extended family, extended model, type, family, model, and stepping) as well as the platform bits of the current processor.

Example 9-7. Pseudo Code, Checks Required Prior to Loading an Update

```
For each processor in the system
{
    Determine the Processor Signature via CPUID function 1;
    Determine the Platform Bits ← 1 << IA32_PLATFORM_ID[52:50];

    For (I ← UpdateBlock 0, I < NumOfBlocks; I++)
    {
        If (Update.Header_Version == 0x00000001)
        {
            If ((Update.ProcessorSignature == Processor Signature) &&
                 (Update.ProcessorFlags & Platform Bits))
            {
                Load Update.UpdateData into the Processor;
                Verify update was correctly loaded into the processor
                Go on to next processor
                    Break;
            }
            Else If (Update.TotalSize > (Update.DataSize + 48))
            {
                N ← 0
                While (N < Update.ExtendedSignatureCount)
                {
                    If ((Update.ProcessorSignature[N] ==
                         Processor Signature) &&
                         (Update.ProcessorFlags[N] & Platform Bits))
                    {
                        Load Update.UpdateData into the Processor;
                        Verify update was correctly loaded into the processor
                        Go on to next processor
                            Break;
                    }
                    N ← N + 1
                }
                I ← I + (Update.TotalSize / 2048)
                If ((Update.TotalSize MOD 2048) == 0)
                    I ← I + 1
            }
        }
    }
}
```

NOTE

The platform Id bits in IA32_PLATFORM_ID are encoded as a three-bit binary coded decimal field. The platform bits in the microcode update header are individually bit encoded. The algorithm must do a translation from one format to the other prior to doing a check.

When performing the INT 15H, 0D042H functions, the BIOS must assume that the caller has no knowledge of platform specific requirements. It is the responsibility of BIOS calls to manage all chipset and platform specific prerequisites for managing the NVRAM device. When writing the update data using the Write Update sub-function, the BIOS must maintain implementation specific data requirements (such as the update of NVRAM checksum). The BIOS should also attempt to verify the success of write operations on the storage device used to record the update.

9.11.8.2. Responsibilities of the Calling Program

This section of the document lists the responsibilities of acalling program using the interface specifications to load microcode update(s) into BIOS NVRAM.

- The calling program should call the INT 15H, 0D042H functions from a pure real mode program and should be executing on a system that is running in pure real mode.
- The caller should issue the presence test function (sub function 0) and verify the signature and return codes of that function.
- It is important that the calling program provides the required scratch RAM buffers for the BIOS and the proper stack size as specified in the interface definition.
- The calling program should read any update data that already exists in the BIOS in order to make decisions about the appropriateness of loading the update. The BIOS must refuse to overwrite a newer update with an older version. The update header contains information about version and processor specifics for the calling program to make an intelligent decision about loading.
- There can be no ambiguous updates. The BIOS must refuse to allow multiple updates for the same CPU to exist at the same time; it also must refuse to load updates for processors that don't exist on the system.
- The calling application should implement a verify function that is run after the update write function successfully completes. This function reads back the update and verifies that the BIOS returned an image identical to the one that was written.

Example 9-8 represents a calling program.

Example 9-8. INT 15 DO42 Calling Program Pseudo-code

```
//
// We must be in real mode
//
If the system is not in Real mode exit
//
// Detect the presence of Genuine Intel processor(s) that can be updated
// using(CPUID)
//
If no Intel processors exist that can be updated exit
```

```
//
// Detect the presence of the Intel microcode update extensions
//
If the BIOS fails the PresenceTestexit
//
// If the APIC is enabled, see if any other processors are out there
//
Read IA32_APICBASE
If APIC enabled
{
    Send Broadcast Message to all processors except self via APIC
    Have all processors execute CPUID and record the Processor Signature
    (i.e.,Extended Family, Extended Model, Type, Family, Model, Stepping)
    Have all processors read IA32_PLATFORM_ID[52:50] and record Platform
     Id Bits

    If current processor cannot be updated
        exit
}
//
// Determine the number of unique update blocks needed for this system
//
NumBlocks = 0
For each processor
{
    If ((this is a unique processor stepping) AND
        (we have a unique update in the database for this processor))
    {
        Checksum the update from the database;
        If Checksum fails
            exit
        NumBlocks ← NumBlocks + size of microcode update / 2048
    }
}

//
// Do we have enough update slots for all CPUs?
//
If there are more blocks required to support the unique processor steppings
than update blocks provided by the BIOS
    exit
//
// Do we need any update blocks at all?  If not, we are done
//
If (NumBlocks == 0)
    exit
//
// Record updates for processors in NVRAM.
//
For (I=0; I<NumBlocks; I++)
{
    //
    // Load each Update
    //
    Issue the WriteUpdate function

    If (STORAGE_FULL) returned
    {
       Display Error -- BIOS is not managing NVRAM appropriately
       exit
    }

    If (INVALID_REVISION) returned
    {
       Display Message: More recent update already loaded in NVRAM for
        this stepping
       continue
    }

    If any other error returned
    {
       Display Diagnostic
       exit
    }

    //
    //  Verify the update was loaded correctly
    //
    Issue the ReadUpdate function

    If an error occurred
    {
       Display Diagnostic
       exit
    }
    //
    //  Compare the Update read to that written
    //
    If (Update read != Update written)
    {
       Display Diagnostic
       exit
    }

    I ← I + (size of microcode update / 2048)
}
//
//  Enable Update Loading, and inform user
//
Issue the Update Control function with Task = Enable.
```

9.11.8.3. Microcode Update Functions

Table 9-7 defines current Pentium 4, Intel Xeon, and P6 family processor microcode update functions.

Table 9-7. Microcode Update Functions

Microcode Update Function	Function Number	Description	Required/Optional
Presence test	00H	Returns information about the supported functions.	Required
Write update data	01H	Writes one of the update data areas (slots).	Required

Table 9-7. Microcode Update Functions

Microcode Update Function	Function Number	Description	Required/Optional
Update control	02H	Globally controls the loading of updates.	Required
Read update data	03H	Reads one of the update data areas (slots).	Required

9.11.8.4. INT 15H-based Interface

Intel recommends that a BIOS interface be provided that allows additional microcode updates to be added to system flash. The INT15H interface is the Intel-defined method for doing this.

The program that calls this interface is responsible for providing three 64-kilobyte RAM areas for BIOS use during calls to the read and write functions. These RAM scratch pads can be used by the BIOS for any purpose, but only for the duration of the function call. The calling routine places real mode segments pointing to the RAM blocks in the CX, DX and SI registers. Calls to functions in this interface must be made with a minimum of 32 kilobytes of stack available to the BIOS.

In general, each function returns with CF cleared and AH contains the returned status. The general return codes and other constant definitions are listed in Section 9.11.8.9., *Return Codes*.

The OEM error field (AL) is provided for the OEM to return additional error information specific to the platform. If the BIOS provides no additional information about the error, OEM error must be set to SUCCESS. The OEM error field is undefined if AH contains either SUCCESS (00H) or NOT_IMPLEMENTED (86H). In all other cases, it must be set with either SUCCESS or a value meaningful to the OEM.

The following sections describe functions provided by the INT15H-based interface.

9.11.8.5. Function 00H—Presence Test

This function verifies that the BIOS has implemented required microcode update functions. Table 9-8 lists the parameters and return codes for the function.

Table 9-8. Parameters for the Presence Test

Input		
AX	Function Code	0D042H
BL	Sub-function	00H - Presence test
Output		
CF	Carry Flag	Carry Set - Failure - AH contains status Carry Clear - All return values valid
AH	Return Code	
AL	OEM Error	Additional OEM information.
EBX	Signature Part 1	'INTE' - Part one of the signature
ECX	Signature Part 2	'LPEP'- Part two of the signature
EDX	Loader Version	Version number of the microcode update loader
SI	Update Count	Number of 2048 update blocks in NVRAM the BIOS allocated to storing microcode updates
Return Codes (see Table 9-13 for code definitions)		
SUCCESS		The function completed successfully.
NOT_IMPLEMENTED		The function is not implemented.

Description

In order to assure that the BIOS function is present, the caller must verify the carry flag, the return code, and the 64-bit signature. The update count reflects the number of 2048-byte blocks available for storage within one non-volatile RAM.

The loader version number refers to the revision of the update loader program that is included in the system BIOS image.

9.11.8.6. Function 01H—Write Microcode Update Data

This function integrates a new microcode update into the BIOS storage device. Table 9-9 lists the parameters and return codes for the function.

Table 9-9. Parameters for the Write Update Data Function

Input		
AX	Function Code	0D042H
BL	Sub-function	01H - Write update
ES:DI	Update Address	Real Mode pointer to the Intel Update structure. This buffer is 2048 bytes in length if the processor supports only fixed-size microcode update or... Real Mode pointer to the Intel Update structure. This buffer is 64K-bytes in length if the processor supports a variable-size microcode update.
CX	Scratch Pad1	Real mode segment address of 64 kilobytes of RAM block
DX	Scratch Pad2	Real mode segment address of 64 kilobytes of RAM block
SI	Scratch Pad3	Real mode segment address of 64 kilobytes of RAM block
SS:SP	Stack pointer	32 kilobytes of stack minimum
Output		
CF	Carry Flag	Carry Set - Failure - AH Contains status Carry Clear - All return values valid
AH	Return Code	Status of the call
AL	OEM Error	Additional OEM information
Return Codes (see Table 9-13 for code definitions)		
SUCCESS		The function completed successfully.
WRITE_FAILURE		A failure occurred because of the inability to write the storage device.
ERASE_FAILURE		A failure occurred because of the inability to erase the storage device.
READ_FAILURE		A failure occured because of the inability to read the storage device.
STORAGE_FULL		The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.
CPU_NOT_PRESENT		The processor stepping does not currently exist in the system.
INVALID_HEADER		The update header contains a header or loader version that is not recognized by the BIOS.
INVALID_HEADER_CS		The update does not checksum correctly.
SECURITY_FAILURE		The processor rejected the update.
INVALID_REVISION		The same or more recent revision of the update exists in the storage device.

Description

The BIOS is responsible for selecting an appropriate update block in the non-volatile storage for storing the new update. This BIOS is also responsible for ensuring the integrity of the information provided by the caller, including authenticating the proposed update before incorporating it into storage.

Before writing the update block into NVRAM, the BIOS should ensure that the update structure meets the following criteria in the following order:

1. The update header version should be equal to an update header version recognized by the BIOS.
2. The update loader version in the update header should be equal to the update loader version contained within the BIOS image.
3. The update block must checksum. This checksum is computed as a 32-bit summation of all double words in the structure, including the header, data, and processor signature table.

The BIOS selects update block(s) in non-volatile storage for storing the candidate update. The BIOS can select any available update block as long as it guarantees that only a single update exists for any given processor stepping in non-volatile storage. If the update block selected already contains an update, the following additional criteria apply to overwrite it:

- The processor signature in the proposed update must be equal to the processor signature in the header of the current update in NVRAM (Processor Signature + platform ID bits).
- The update revision in the proposed update should be greater than the update revision in the header of the current update in NVRAM.

If no unused update blocks are available and the above criteria are not met, the BIOS can overwrite update block(s) for a processor stepping that is no longer present in the system. This can be done by scanning the update blocks and comparing the processor steppings, identified in the MP Specification table, to the processor steppings that currently exist in the system.

Finally, before storing the proposed update in NVRAM, the BIOS must verify the authenticity of the update via the mechanism described in Section 9.11.6., *Microcode Update Loader*. This includes loading the update into the current processor, executing the CPUID instruction, reading MSR 08Bh, and comparing a calculated value with the update revision in the proposed update header for equality.

When performing the write update function, the BIOS must record the entire update, including the header, the update data, and the extended processor signature table (if applicable). When writing an update, the original contents may be overwritten, assuming the above criteria have been met. It is the responsibility of the BIOS to ensure that more recent updates are not overwritten through the use of this BIOS call, and that only a single update exists within the NVRAM for any processor stepping and platform ID.

Figure 9-8 and Figure 9-9 show the process the BIOS follows to choose an update block and ensure the integrity of the data when it stores the new microcode update.

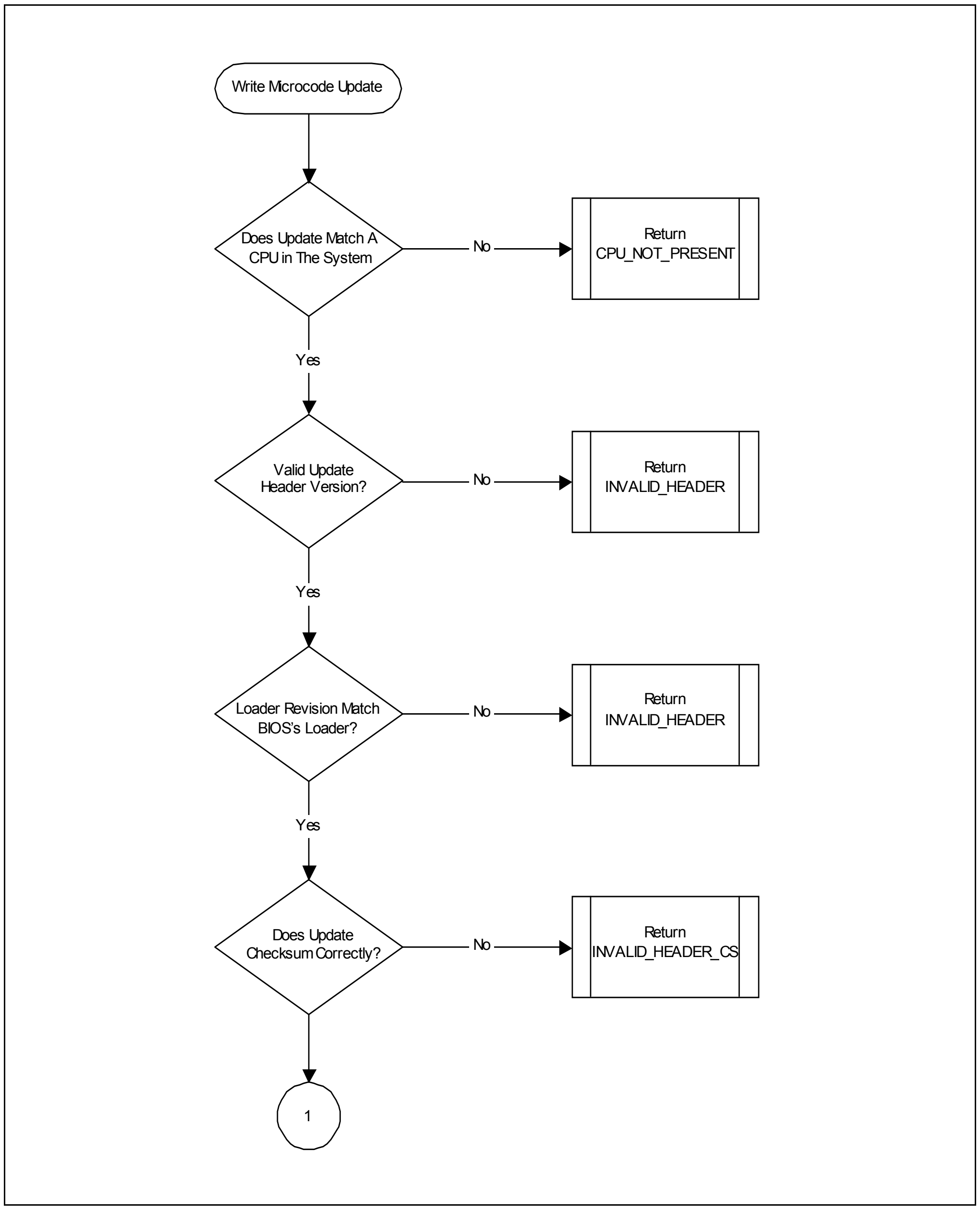

Write Microcode Update
Does Update Match A CPU in The System
No
Return CPU_NOT_PRESENT
Yes
Valid Update Header Version?
No
Return INVALID_HEADER
Yes
Loader Revision Match BIOS's Loader?
No
Return INVALID_HEADER
Yes
Does Update Checksum Correctly?
No
Return INVALID_HEADER_CS
1

Figure 9-8. Microcode Update Write Operation Flow [1]

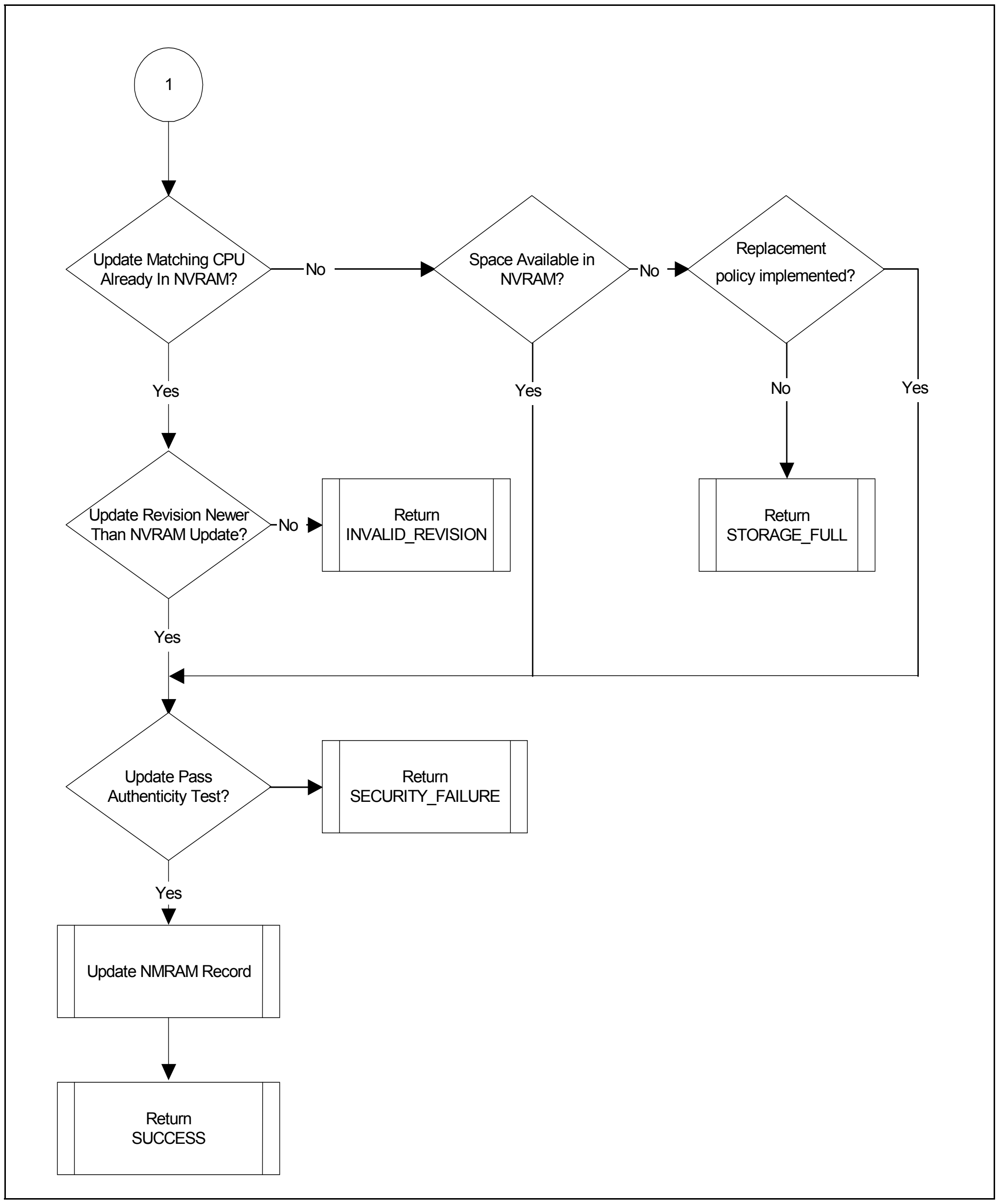

1
Update Matching CPU Already In NVRAM?
No
Space Available in NVRAM?
No
Replacement policy implemented?
Yes
Yes
No
Yes
Update Revision Newer Than NVRAM Update?
No
Return INVALID_REVISION
Return STORAGE_FULL
Yes
Update Pass Authenticity Test?
Return SECURITY_FAILURE
Yes
Update NMRAM Record
Return SUCCESS

Figure 9-9. Microcode Update Write Operation Flow [2]

9.11.8.7. Function 02H—Microcode Update Control

This function enables loading of binary updates into the processor. Table 9-10 lists the parameters and return codes for the function.

Table 9-10. Parameters for the Control Update Sub-function

Input		
AX	Function Code	0D042H
BL	Sub-function	02H - Control update
BH	Task	See the description below.
CX	Scratch Pad1	Real mode segment of 64 kilobytes of RAM block
DX	Scratch Pad2	Real mode segment of 64 kilobytes of RAM block
SI	Scratch Pad3	Real mode segment of 64 kilobytes of RAM block
SS:SP	Stack pointer	32 kilobytes of stack minimum
Output		
CF	Carry Flag	Carry Set - Failure - AH contains status Carry Clear - All return values valid.
AH	Return Code	Status of the call
AL	OEM Error	Additional OEM Information.
BL	Update Status	Either enable or disable indicator
Return Codes (see Table 9-13 for code definitions)		
SUCCESS		Function completed successfully.
READ_FAILURE		A failure occurred because of the inability to read the storage device.

Description

This control is provided on a global basis for all updates and processors. The caller can determine the current status of update loading (enabled or disabled) without changing the state. The function does not allow the caller to disable loading of binary updates, as this poses a security risk.

The caller specifies the requested operation by placing one of the values from Table 9-11 in the BH register. After successfully completing this function, the BL register contains either the enable or the disable designator. Note that if the function fails, the update status return value is undefined.

Table 9-11. Mnemonic Values

Mnemonic	Value	Meaning
Enable	1	Enable the Update loading at initialization time.
Query	2	Determine the current state of the update control without changing its status.

The READ_FAILURE error code returned by this function has meaning only if the control function is implemented in the BIOS NVRAM. The state of this feature (enabled/disabled) can also be implemented using CMOS RAM bits where READ failure errors cannot occur.

9.11.8.8. Function 03H—Read Microcode Update Data

This function reads a currently installed microcode update from the BIOS storage into a caller-provided RAM buffer. Table 9-12 lists the parameters and return codes for the function.

Table 9-12. Parameters for the Read Microcode Update Data Function

Input		
AX	Function Code	0D042H
BL	Sub-function	03H - Read Update
ES:DI	Buffer Address	Real Mode pointer to the Intel Update structure that will be written with the binary data
ECX	Scratch Pad1	Real Mode Segment address of 64 kilobytes of RAM Block (lower 16 bits)
ECX	Scratch Pad2	Real Mode Segment address of 64 kilobytes of RAM Block (upper 16 bits)
DX	Scratch Pad3	Real Mode Segment address of 64 kilobytes of RAM Block
SS:SP	Stack pointer	32 kilobytes of Stack Minimum
SI	Update Number	This is the index number of the update block to be read. This value is zero based and must be less than the update count returned from the presence test function.
Output		
CF	Carry Flag	Carry Set - Failure - AH contains Status
Carry Clear - All return values are valid.		
AH	Return Code	Status of the Call
AL	OEM Error	Additional OEM Information
Return Codes (see Table 9-13 for code definitions)		
SUCCESS		The function completed successfully.
READ_FAILURE		There was a failure because of the inability to read the storage device.
UPDATE_NUM_INVALID		Update number exceeds the maximum number of update blocks implemented by the BIOS.
NOT_EMPTY		The specified update block is a subsequent block in use to store a valid microcode update that spans multiple blocks. The specified block is not a header block and is not empty.

Description

The read function enables the caller to read any microcode update data that already exists in a BIOS and make decisions about the addition of new updates. As a result of a successful call, the BIOS copies the microcode update into the location pointed to by ES:DI, with the contents of all Update block(s) that are used to store the specified microcode update.

If the specified block is not a header block, but does contain valid data from a microcode update that spans multiple update blocks, then the BIOS must return Failure with the NOT_EMPTY error code in AH.

An update block is considered unused and available for storing a new update if its Header Version contains the value 0FFFFFFFFH after return from this function call. The actual implementation of

NVRAM storage management is not specified here and is BIOS dependent. As an example, the actual data value used to represent an empty block by the BIOS may be zero, rather than 0FFFFFFFFH. The BIOS is responsible for translating this information into the header provided by this function.

9.11.8.9. Return Codes

After the call has been made, the return codes listed in Table 9-13 are available in the AH register.

Table 9-13. Return Code Definitions

Return Code	Value	Description
SUCCESS	00H	The function completed successfully.
NOT_IMPLEMENTED	86H	The function is not implemented
ERASE_FAILURE	90H	A failure because of the inability to erase the storage device.
WRITE_FAILURE	91H	A failure because of the inability to write the storage device.
READ_FAILURE	92H	A failure because of the inability to read the storage device.
STORAGE_FULL	93H	The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.
CPU_NOT_PRESENT	94H	The processor stepping does not currently exist in the system.
INVALID_HEADER	95H	The update header contains a header or loader version that is not recognized by the BIOS.
INVALID_HEADER_CS	96H	The update does not checksum correctly.
SECURITY_FAILURE	97H	The update was rejected by the processor.
INVALID_REVISION	98H	The same or more recent revision of the update exists in the storage device.
UPDATE_NUM_INVALID	99H	The update number exceeds the maximum number of update blocks implemented by the BIOS.
NOT_EMPTY	9AH	The specified update block is a subsequent block in use to store a valid microcode update that spans multiple blocks. The specified block is not a header block and is not empty.

22. A Mechanism for Determining Sync/Async SMIs Has Been Documented

In Volume 3, a new section has been added. The section provides information about two new CRs that provide an effective means to determine whether an SMI is synchronous or asynchronous. The new section is reproduced below.

13.7. MANAGING SYNCHRONOUS AND ASYNCHRONOUS SYSTEM MANAGEMENT INTERRUPTS

Particularly when coding for a multiprocessor system or a system with Intel HT Technology, it was not always possible for an SMI handler to distinguish between a synchronous SMI (triggered during an I/O instruction) and an asynchronous SMI. To facilitate the discrimination of these two events, incremental state information has been added to the SMM state save map.

Processors that have an SMM revision ID of 30004H or higher have the incremental state information described below.

13.7.1. I/O State Implementation

Within the extended SMM state save map, a bit (IO_SMI) is provided that is set only when an SMI is either taken immediately after a *successful* I/O instruction or is taken after a *successful* iteration of a REP I/O instruction (note that the *successful* notion pertains to the processor point of view; not necessarily to the corresponding platform function). When set, the IO_SMI bit provides a strong indication that the corresponding SMI was synchronous. In this case, the SMM State Save Map also supplies the port address of the I/O operation. The IO_SMI bit and the I/O Port Address may be used in conjunction with the information logged by the platform to confirm that the SMI was indeed synchronous.

Note that the IO_SMI bit by itself is a strong indication, not a guarantee, that the SMI is synchronous. This is because an asynchronous SMI might coincidentally be taken after an I/O instruction. In such a case, the IO_SMI bit would still be set in the SMM state save map.

Information characterizing the I/O instruction is saved in two locations in the SMM State Save Map (Table 13-14). Note that the IO_SMI bit also serves as a valid bit for the rest of the I/O information fields. The contents of these I/O information fields are not defined when the IO_SMI bit is not set.

Table 13-14. I/O Instruction Information in the SMM State Save Map

State (SMM Rev. ID: 30004H or higher)	**Format**				
	31 16	**15 8**	**7 4**	**3 1**	**0**
I/0 State Field SMRAM offset 7FA4	I/O Port	Reserved	I/O Type	I/O Length	IO_SMI
	31 0				
I/O Memory Address Field SMRAM offset 7FA0	I/O Memory Address				

IO_SMI is set if an SMI was taken during or immediately following an I/O instruction. When IO_SMI is set, the other fields may be interpreted as follows:

- I/O Length can be:
    - 001 – Byte
    - 010 – Word
    - 100 – Dword
- I/O Instruction Type:

Table 13-15. I/O Instruction Type Encodings

Instruction	**Encoding**
IN Immediate	1001
IN DX	0001
OUT Immediate	1000
OUT DX	0000
INS	0011
OUTS	0010
REP INS	0111
REP OUTS	0110

- I/O Memory Address is:
    - for OUTS/REP_OUTS: Segment_base + eSI
    - for INS/REP_INS: ES_base + eDI
    - for IN/OUT: 0x0

The SMRAM save map (Table 13-1) has also been updated. It now indicates the location of the relevant CRs. Impacted table cells are reproduced below.

Table 13-1. SMRAM State Save Map

Offset (Added to SMBASE + 8000H)	Register	Writable?
7FA4H	I/O State Field, see Section 13.7.	No
7FA0H	I/O Memory Address Field, see Section 13.7.	No

23. Omitted Debug Data Has Been Restored

In the Volume 3, Chapter 5, Interrupt 1 section; data deleted by mistake has been restored. This material is reproduced below.

Interrupt 1—Debug Exception (#DB)

Exception Class Trap or Fault. The exception handler can distinguish between traps or faults by examining the contents of DR6 and the other debug registers.

Description

Indicates that one or more of several debug-exception conditions has been detected. Whether the exception is a fault or a trap depends on the condition (see Table 13-2). See Chapter 15, *Debugging and Performance Monitoring*, for detailed information about the debug exceptions.

Table 13-2. Debug Exception Conditions and Corresponding Exception Classes

Exception Condition	Exception Class
Instruction fetch breakpoint	Fault
Data read or write breakpoint	Trap
I/O read or write breakpoint	Trap
General detect condition (in conjunction with in-circuit emulation)	Fault
Single-step	Trap
Task-switch	Trap

Exception Error Code

None. An exception handler can examine the debug registers to determine which condition caused the exception.

Saved Instruction Pointer

Fault—Saved contents of CS and EIP registers point to the instruction that generated the exception.

Trap—Saved contents of CS and EIP registers point to the instruction following the instruction that generated the exception.

Program State Change

Fault—A program-state change does not accompany the debug exception, because the exception occurs before the faulting instruction is executed. The program can resume normal execution upon returning from the debug exception handler.

Trap—A program-state change does accompany the debug exception, because the instruction or task switch being executed is allowed to complete before the exception is generated. However, the new state of the program is not corrupted and execution of the program can continue reliably.

24. CLTS Exception Information Updated

In the Volume 2, Chapter 3, CLTS—Clear Task-Switched Flag in CR0 section; missing exception data has been added. The corrected area is reproduced below.

CLTS—Clear Task-Switched Flag in CR0: only updated exception sections reproduced...

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

#GP(0)	CLTS is not recognized in virtual-8086 mode.

25. The MOVSS Description Has Been Updated

In the Volume 2, Chapter 3, MOVSS—Move Scalar Single--Precision Floating-Point Values section; incorrect data has been updated. The corrected area is reprinted below. In the incorrect version, the description block below indicated 64-bit memory locations.

MOVSS—Move Scalar Single--Precision Floating-Point Values

Opcode	Instruction	Description
F3 0F 10 /r	MOVSS *xmm1*, *xmm2/m32*	Move scalar single-precision floating-point value from *xmm2/m32* to *xmm1* register.
F3 0F 11 /r	MOVSS *xmm2/m32*, *xmm1*	Move scalar single-precision floating-point value from *xmm1* register to *xmm2/m32*.

Description

Moves a scalar single-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 32-bit memory locations. This instruction can be used to move a single-precision floating-point value to and from the low doubleword of an XMM register and a 32-bit memory location, or to move a single-precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

When the source and destination operands are XMM registers, the three high-order doublewords of the destination operand remain unchanged. When the source operand is a memory location and destination operand is an XMM registers, the three high-order doublewords of the destination operand are cleared to all 0s.

----

26. An Instruction Listing (PULLHUW) Has Been Deleted

In Volume 2of the IA-32 Intel Architecture Software Developer's Manual, Appendix B, Table B-13; there was a listing for a 'PULLHUW' instruction. This appears to have been an old corruption of PMULHUW entry (see the SSE/SSE2 entries by that name). The PULLHUW table entry has been deleted. The reproduced table cells (below) indicate the point of deletion.

PMADDWD - Packed multiply add	
mmxreg2 to mmxreg1	0000 1111:11110101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11110101: mod mmxreg r/m
PMULHW - Packed multiplication, store high word	
mmxreg2 to mmxreg1	0000 1111:11100101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11100101: mod mmxreg r/m

27. Data Entry Errors in Table B-20 Have Been Corrected

In Volume 2 of the IA-32 Intel Architecture Software Developer's Manual, Appendix B, Table B-20. MOVQ encoding information has been changed. The impacted table cells are reproduced below.

Table B-20. (continued)

MOVQ - Move Quadword	
xmmreg2 to xmmreg1	**11110011**:00001111:01111110: 11 xmmreg1 xmmreg2
xmmreg2 from xmmreg1	01100110:00001111:11010110: 11 xmmreg1 xmmreg2
mem to xmmreg	**11110011**:00001111:01111110: mod xmmreg r/m
mem from xmmreg	01100110:00001111:11010110: mod xmmreg r/m

28. Figure Has Been Corrected

In Volume 3, Chapter 8, Figure 8-20; an incorrect address has been corrected. This is the second draft of this correction. The corrected figure and impacted text have been reproduced below.

8.8.4 Interrupt Acceptance for Fixed Interrupts

The local APIC queues the fixed interrupts that it accepts in one of two interrupt pending registers: the interrupt request register (IRR) or in-service register (ISR). These two 256-bit read-only registers are shown in Figure 8-20). The 256 bits in these registers represent the 256 possible vectors; vectors 0 through 15 are reserved by the APIC (see also: Section 8.5.2.).

NOTE

All interrupts with an NMI, SMI, INIT, ExtINT, start-up, or INIT-deassert delivery mode bypass the IRR and ISR registers and are sent directly to the processor core for servicing.

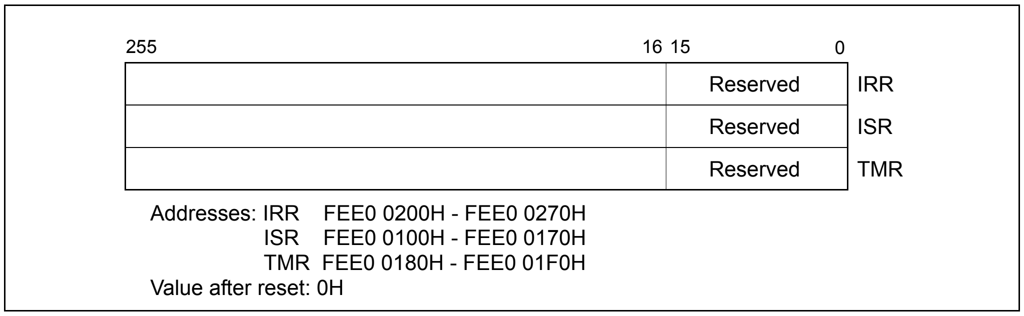

255
16 15
0
Reserved
Reserved
Reserved
IRR
ISR
TMR
Addresses: IRR FEE0 0200H - FEE0 0270H
ISR FEE0 0100H - FEE0 0170H
TMR FEE0 0180H - FEE0 01F0H
Value after reset: 0H

Figure 8-20. IRR, ISR and TMR Registers

29. The Description of Minimum Thermal Monitor Activation Time Has Been Updated

In Volume 3of the IA-32 Intel Architecture Software Developer's Manual, Section 13.15.2.4; a paragraph describing TM1/TM2 has been re-written to provide a more accurate description. The applicable text is reproduced below.

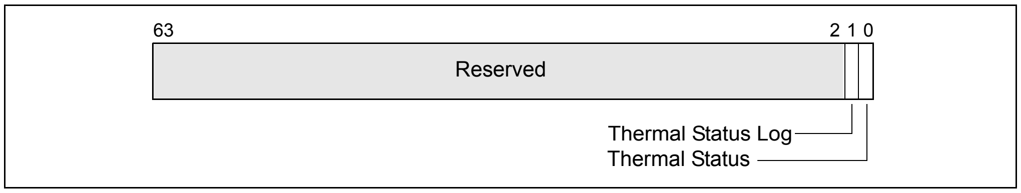


Figure 0-1. IA32_THERM_STATUS MSR

Thermal Status Log flag, bit 1

When set, indicates that the thermal sensor has tripped since the last power-up or reset or since the last time that software cleared this flag. This flag is a sticky bit; once set it remains set until cleared by software or until a power-up or reset of the processor. The default state is clear.

After the second temperature sensor has been tripped, the thermal monitor (TM1/TM2) will remain engaged for a minimum time period (on the order of 1 ms). The thermal monitor will remain engaged until the processor core temperature drops below the preset trip temperature of the temperature sensor, taking hysteresis into account.

30. Corrected Description of Exception- or Interrupt-Handler Procedures

In Volume 3, Section 5.12.1; text describing exception- or interrupt-handler procedures has been rewritten. The new text is reproduced below.

When the processor performs a call to the exception- or interrupt-handler procedure:

- If the handler procedure is going to be executed at a numerically lower privilege level, a stack switch occurs. When the stack switch occurs:
  a. The segment selector and stack pointer for the stack to be used by the handler are obtained from the TSS for the currently executing task. On this new stack, the processor pushes the stack segment selector and stack pointer of the interrupted procedure.
  b. The processor then saves the current state of the EFLAGS, CS, and EIP registers on the new stack (see Figure 5-4).
  c. If an exception causes an error code to be saved, it is pushed on the new stack after the EIP value.
- If the handler procedure is going to be executed at the same privilege level as the interrupted procedure:
  a. The processor saves the current state of the EFLAGS, CS, and EIP registers on the current stack (see Figure 5-4).
  b. If an exception causes an error code to be saved, it is pushed on the current stack after the EIP value.

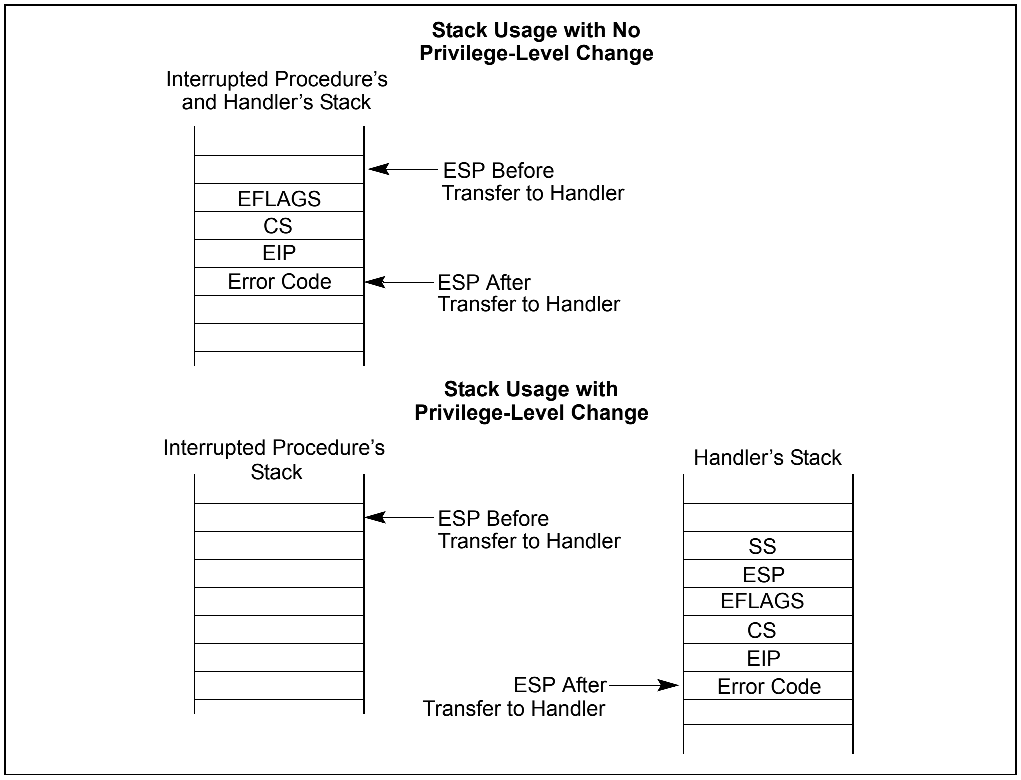

Stack Usage with No Privilege-Level Change
Interrupted Procedure's and Handler's Stack
EFLAGS
CS
EIP
Error Code
ESP Before Transfer to Handler
ESP After Transfer to Handler
Stack Usage with Privilege-Level Change
Interrupted Procedure's Stack
Handler's Stack
ESP Before Transfer to Handler
SS
ESP
EFLAGS
CS
EIP
Error Code
ESP After Transfer to Handler

Figure 5-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

31. CMPSD and CMPSS Exception Information Updated

Changes were made in the relevant sections of Volume 2of the IA-32 Intel Architecture Software Developer's Manual, Chapter 3. The updated exception sections are reproduced below.

CMPSD — Compare Scalar Double-Precision Floating-Point Values: only updated exception sections reproduced....

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

CMPSS — Compare Scalar Single-Precision Floating-Point Values: only updated exception sections reproduced....

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.
#AC(0)	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)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.

#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

32. PUNPCKHB*/PUNPCKLB* Exception Information Improved

Changes were made in the relevant sections of Volume 2, Chapter 3. The updated exception sections are reproduced below.

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ — Unpack High Data: only updated exception sections reproduced....

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 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 EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#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)	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
	(128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#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.

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ — Unpack Low Data: only updated exception sections reproduced...

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 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 EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#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)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
	(128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#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.

33. MOVHPD, MOVLPD, UNPCKHPS, UNPCKLPS Exception Information Updated

Changes were made in the relevant sections of Volume 2 of the *IA-32 Intel Architecture Software Developer's Manual*, Chapter 3. The updated exception sections are reproduced below.

MOVHPD—Move High Packed Double-Precision Floating-Point Value: only updated exception sections reproduced...

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.
#AC(0)	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)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.

#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

MOVHPS — Move High Packed Single-Precision Floating-Point Values: only updated exception sections reproduced...

Real-Address Mode Exceptions

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values: only updated exception sections reproduced...

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values: only updated exception sections reproduced...

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.

#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

34. PEXTRW - PINSRW Exception Information Updated

Changes were made in the relevant sections of Volume 2 *of the IA-32 Intel Architecture Software Developer's Manual*, Chapter 3. The updated exception sections are reproduced below.

PEXTRW — Extract Word: only updated exception sections reproduced...

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	#SS(0)If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#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)	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

35. Restructuring of IA-32 manuals, Volume 2

IA-32 Intel Architecture Software Developer's Manual, Volume 2 has been split into Volumes 2A-2B. This was done because the size of the growing size of the volume. The re-orgnanization required the updating of cross references and some new introductory information. All introductory chapters and all title pages were impacted. The technical content for impacted sections did not change.

36. Statement on Setting the OXFXSR Flag Corrected

IA-32 Intel Architecture Software Developer's Manual, Volume 3; Chapter 12, Section 12.1.4 has been updated. A statement that describes an exception condition associated with setting the OSFXSR flag has been corrected. See the text reproduced below.

12.1.4. Initialization of the SSE/SSE2/SSE3 Extensions

The operating system or executive should carry out the following steps to set up the SSE/SSE2/SSE3 extensions for use by application programs.

1. Set bit 9 of CR4 (the OSFXSR bit) to 1. Setting this flag assumes that the operating system provides facilities for saving and restoring SSE/SSE2/SSE3 states using FXSAVE and FXRSTOR instructions. These instructions are commonly used to save the SSE/SSE2/SSE3 state during task switches and when invoking the SIMD floating-point exception (#XF) handler (see Section 12.4., "Saving the SSE/SSE2/SSE3 State on Task or Context Switches" and Section 12.1.6., "Providing an Handler for the SIMD Floating-Point Exception (#XF)", respectively). If the processor does not support the FXSAVE and FXRSTOR instructions, attempting to set the OSFXSR flag will cause an exception (#GP) to be generated.
2. Set bit 10 of CR4 (the OSXMMEXCPT bit) to 1. Setting this flag assumes that the operating system provides an SIMD floating-point exception (#XF) handler (see Section 12.1.6., "Providing an Handler for the SIMD Floating-Point Exception (#XF)").

37. Disclaimer Now Includes Stronger Warning

For all IA-32 documentation: The new legal disclaimer includes a stronger warning on the subject of using reserved areas. The text is reproduced below.

Developers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Improper use of reserved or undefined features or instructions may cause unpredictable behavior or failure in developer's software code when running on an Intel processor. Intel reserves these features or instructions for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from their unauthorized use.

38. Correction in the Description of SYSENTER and SYSEXIT

IA-32 Intel Architecture Software Developer's Manual, Volumes 2A & 2B; Appendix B, Table B-15 (in the new edition),Table B-14 (in the old edition) ; encodings for SYSENTER and SYSEXIT have been corrected. See the table segment reproduced below for the change.

Instruction and Format	Encoding
SYSENTER—Fast System Call	00001111:00110100
SYSEXIT—Fast Return from Fast System Call	00001111:00110101

39. Exception Lists for ANPD/ANPS Have Been Updated

IA-32 Intel Architecture Software Developer's Manual, Volumes 2A & 2B; Chapter 3, The ANPD and ANPS exception lists have been updated to include corrections. See the material reproduced below for the corrections.

From 'ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values'

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

From: 'ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values'

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

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

40. Cache descriptors for B0H, B3H Have Been Updated

IA-32 Intel Architecture Software Developer's Manual, Volumes 2A & 2B; Chapter 3, CPUID—CPU Identification section, Table 3-12 (in the new edition), information on the B0H and B3H descriptors has been corrected. The relevant table segment is reproduced below.

B0H	Instruction TLB: 4K-Byte Pages, 4-way set associative, 128 entries
B3H	Data TLB: 4K-Byte Pages, 4-way set associative, 128 entries

41. Exception List Corrections for UNPCKLPS, UNPCKHPS, UNPCKLPD, UNPCKHPD, SHUFPS, SHUFPD, RSQRTSS, RSQRTPS, RCPSS, RCPPS, MOVUPS, MOVUPD, MOVMSKPS, MOVMSKPD, MOVAPS, and MOVAPD Have Been Made

----

IA-32 Intel Architecture Software Developer's Manual, Volumes 2A & 2B; Chapters 3 and 4: the exception lists for instructions listed above have been updated.

The following actions were taken:

- #XM line has been removed in all three modes.
- #UD (CR4 OSXCEPT) has been removed in all three modes.

42. Correction to SMCCLEAR

IA-32 Intel Architecture Software Developer's Manual, Volume 3; Appendix A, Table A-2, SMCCLEAR has been updated. See the table segment reproduced below for the corrections.

Event Name	Event Parameters	Parameter Value	Description
machine_clear			This event increments according to the mask bit specified while the entire pipeline of the machine is cleared. Specify one of the mask bit to select the cause.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	
	ESCR Event Select	02H	ESCR[31:25]
	ESCR Event Mask		ESCR[24:9]
		Bit 0: CLEAR	Counts for a portion of the many cycles while the machine is cleared for any cause. Use Edge triggering for this bit only to get a count of occurrence versus a duration.
		2: MOCLEAR	Increments each time the machine is cleared due to memory ordering issues.
		6: SMCCLEAR	Increments each time the machine is cleared due to self-modifying code issues.

43. REP INS Syntax Has Been Corrected

IA-32 Intel Architecture Software Developer's Manual, Volumes 2A & 2B; Chapter 4 (in the new edition), REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix section, the opening syntax statements have been corrected. See the table segment reproduced below for the data.

Opcode	Instruction	Description
F3 6C	REP INS *m8*, DX	Input (E)CX bytes from port DX into ES:[(E)DI]
F3 6D	REP INS *m16*, DX	Input (E)CX words from port DX into ES:[(E)DI]
F3 6D	REP INS *m32*, DX	Input (E)CX doublewords from port DX into ES:[(E)DI]

44. SGDT/SIDT Descriptions Are Now In Two Separate Sections

IA-32 Intel Architecture Software Developer's Manual, Volumes 2A & 2B; Chapter 4 (in the new edition): the SGDT and SIDT instructions are now described in separate sections. These are titled: 'SGDT—Store Global Descriptor Table Register' and 'SIDT—Store Interrupt Descriptor Table Register'respectively. The technical content for these sections remains the same.

45. Revision to Vol. 2B, Appendix A & Appendix B

In Volume 2B, Apendix A and Appendix B have been updated. SSE3 data has been added; in addition, more information has been added to the opcode map. Both apendixes are reproduced in this package. Areas of activity are indicated by change bars.

See APPENDIX A and APPENDIX B below.

APPENDIX A
OPCODE MAP

Opcode tables in this appendix are provided to aid in interpreting IA-32 object code. Instructions are divided into three encoding groups: 1-byte opcode encoding, 2-byte opcode encoding, and escape (floating-point) encoding.

One and 2-byte opcode encoding is used to encode integer, system, MMX technology, and SSE/SSE2/SSE3 instructions. The opcode maps for these instructions are given in Table A-2 and Table A-3. Section A.3.1., "One-Byte Opcode Instructions" through Section A.3.4., "Opcode Extensions For One- And Two-byte Opcodes" give instructions for interpreting 1- and 2-byte opcode maps.

Escape encoding is used to encode floating-point instructions. The opcode maps for these instructions are in Table A-5 through Table A-20. Section A.3.5., "Escape Opcode Instructions" provides instructions for interpreting the escape opcode maps.

A.1. NOTES ON USING OPCODE TABLES

Tables in this appendix define a primary opcode (including instruction prefix where appropriate) and the ModR/M byte. Blank cells in the tables indicate opcodes that are reserved or undefined. Use the four high-order bits of the primary opcode as an index to a row of the opcode table; use the four low-order bits as an index to a column of the table. If the first byte of the primary opcode is 0FH, or 0FH is preceded by either 66H, F2H, F3H; refer to the 2-byte opcode table and use the second byte of the opcode to index the rows and columns of that table.

When the ModR/M byte includes opcode extensions, this indicates that the instructions are an instruction group in Table A-2, Table A-3. More information about opcode extensions in the ModR/M byte are covered in Table A-4.

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

Refer to Chapter 2 in *IA-32 Intel Architecture Software Developer's Manual, Volume 2A* for more information on the ModR/M byte, register values, and addressing forms.

A.2. KEY TO ABBREVIATIONS

Operands are identified by a two-character code of the form Zz. The first character (Z) specifies the addressing method; the second character (z) specifies the type of operand.

A.2.1. Codes for Addressing Method

The following abbreviations are used for 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 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: mod != 11B (BOUND, LEA, LES, LDS, LSS, LFS, LGS, CMPXCHG8B, LDDQU).
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 mod field of the ModR/M byte may refer only to a general register (for example, MOV (0F20-0F24, 0F26)).
S	The reg field of the ModR/M byte selects a segment register (for example, MOV (8C,8E)).
T	The reg field of the ModR/M byte selects a test register (for example, MOV (0F24,0F26)).

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

A.2.2. Codes for Operand Type

The following abbreviations are used for 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 (e.g. mm0)
pd	128-bit packed double-precision floating-point data
ps	128-bit packed single-precision floating-point data.
q	Quadword, regardless of operand-size attribute.
s	6-byte pseudo-descriptor.
sd	Scalar element of a 128-bit packed double-precision floating data.
ss	Scalar element of a 128-bit packed single-precision floating data.
si	Doubleword integer register (e.g., eax)
v	Word or doubleword, depending on operand-size attribute.
w	Word, regardless of operand-size attribute.

A.2.3. Register Codes

When an operand is a specific register encoded in the opcode, the register is identified by its name (for example, AX, CL, or ESI). The name of the register indicates whether the register is

32, 16, or 8 bits wide. A register identifier of the form eXX is used when the width of the register depends on the operand-size attribute. 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.

A.3. OPCODE LOOK-UP EXAMPLES

This section provides several examples to demonstrate how the following opcode maps are used.

A.3.1. One-Byte Opcode Instructions

The opcode maps for 1-byte opcodes are shown in Table A-2. Looking at the 1-byte opcode maps, the instruction mnemonic and its operands can be determined from the hexadecimal value of the 1-byte opcode. 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 list one of the following types of opcodes:

- Instruction mnemonic and operand types using the notations listed in Appendix A.2.2.
- An opcode used as an instruction prefix

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

- ModR/M byte is required and is interpreted according to the abbreviations listed in Appendix A.2. and Chapter 2 in *IA-32 Intel Architecture Software Developer's Manual, Volume 2A*. The operand types are listed according to the notations listed in Appendix A.2.2.
- ModR/M byte is required and includes an opcode extension in the reg field within the ModR/M byte. Use Table A-4 when interpreting the ModR/M byte.
- The use of the ModR/M byte is reserved or undefined. This applies to entries that represents an instruction prefix or an entry for instruction without operands related to ModR/M (e.g. 60H, PUSHA; 06H, PUSH ES).

For example to look up the opcode sequence below:

Opcode: 030500000000H

LSB address					MSB address
03	05	00	00	00	00

Opcode 030500000000H for an ADD instruction can be interpreted from the 1-byte opcode map as follows. The first digit (0) of the opcode indicates the row, and the second digit (3) indicates the column in the opcode map tables. 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 that 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, which indicates that a 32-bit displacement follows (00000000H). The reg/opcode portion of the ModR/M byte (bits 3 through 5) is 000, indicating the EAX register. Thus, it can be determined that 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. These group numbers indicate that the instruction uses the reg/opcode bits in the ModR/M byte as an opcode extension (refer to Section A.3.4., “Opcode Extensions For One- And Two-byte Opcodes”).

A.3.2. 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 byte is used as indices to 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), the escape opcode, the upper and lower four bits of the third byte is used as indices to a particular row and column in Table A-3. The two-byte escape sequence consists of a mandatory prefix (either 66H, F2H, or F3H), followed by the escape prefix byte 0FH.

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

- ModR/M byte is required and is interpreted according to the abbreviations listed in Appendix A.2. and Chapter 2 in *IA-32 Intel Architecture Software Developer’s Manual, Volume 2A* for more information on the ModR/M byte, register values, and the various addressing forms. The operand types are listed according to the notations listed in Appendix A.2.2.
- ModR/M byte is required and includes an opcode extension in the reg field within the ModR/M byte. Use Table A-4 when interpreting the ModR/M byte.
- The use of the ModR/M byte is reserved or undefined. This applies to entries that represents an instruction without operands encoded via ModR/M (e.g. 0F77H, EMMS).

For example, the opcode 0FA4050000000003H is located on the two-byte opcode map in row A, column 4. This opcode indicates a SHLD instruction with the operands Ev, Gv, and Ib. These operands are defined 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 indicate that a 32-bit displacement follows, located in the EAX register, and is the source.

The next part of the opcode is the 32-bit displacement for the destination memory operand (00000000H), and finally the immediate byte representing the count of the shift (03H).

By this breakdown, it has been shown that this opcode represents the instruction:

SHLD DS:00000000H, EAX, 3

The next part of the SHLD opcode is the 32-bit displacement for the destination memory operand (00000000H), which is followed by the immediate byte representing the count of the shift (03H). By this breakdown, it has been shown that the opcode 0FA4050000000003H represents the instruction:

SHLD DS:00000000H, EAX, 3.

Lower case is used in the following tables to highlight the mnemonics added by MMX technology, SSE, and SSE2 instructions.

A.3.3. Opcode Map Notes

Table A-1 contains notes on particular encodings in the opcode map tables. These notes are indicated in the following Opcode Maps (Tables A-2 and A-3) by superscripts. For the One-byte Opcode Maps (Table A-2) shading indicates instruction groupings.

Table A-1. Notes on Instruction Encoding in Opcode Map Tables

Symbol	Note
1A	Bits 5, 4, and 3 of ModR/M byte used as an opcode extension (refer to Section A.3.4., “Opcode Extensions For One- 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 completely decode the instruction, see Table A-4. (These instructions include SFENCE, STMXCSR, LDMXCSR, FXRSTOR, and FXSAVE, as well as PREFETCH and its variations.)
1D	The instruction represented by this opcode expression does not have a ModR/M byte following the primary opcode.
1E	Valid encoding for the r/m field of the ModR/M byte is shown in parenthesis.
1F	The instruction represented by this opcode expression does not support both source and destination operands to be registers.
1G	When the source operand is a register, it must be an XMM register.
1H	The instruction represented by this opcode expression does not support any operand to be a memory location.
1J	The instruction represented by this opcode expression does not support register operand.
1K	Valid encoding for the reg/opcode field of the ModR/M byte is shown in parenthesis.

Table A-2. One-byte Opcode Map† ††

	0	1	2	3	4	5	6	7
0	ADD						**PUSH ES^{1D}**	**POP ES^{1D}**
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	**AL, Ib^{1D}**	**eAX, Iv^{1D}**		
1	ADC						**PUSH SS^{1D}**	**POP SS^{1D}**
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	**AL, Ib^{1D}**	**eAX, Iv^{1D}**		
2	AND						**SEG=ES Prefix**	**DAA^{1D}**
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	**AL, Ib^{1D}**	**eAX, Iv^{1D}**		
3	XOR						**SEG=SS Prefix**	**AAA^{1D}**
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	**AL, Ib^{1D}**	**eAX, Iv^{1D}**		
4	**INC general register**							
	eAX^{1D}	**eCX^{1D}**	**eDX^{1D}**	**eBX^{1D}**	**eSP^{1D}**	**eBP^{1D}**	**eSI^{1D}**	**eDI^{1D}**
5	**PUSH general register^{1D}**							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
6	**PUSHA/ PUSHAD^{1D}**	**POPA/ POPAD^{1D}**	BOUND Gv, Ma	ARPL Ew, Gw	**SEG=FS Prefix**	**SEG=GS Prefix**	**Opd Size Prefix**	**Addr Size Prefix**
7	Jcc, Jb - Short-displacement jump on condition							
	O^{1D}	**NO^{1D}**	**B/NAE/C^{1D}**	**NB/AE/NC^{1D}**	**Z/E^{1D}**	**NZ/NE^{1D}**	**BE/NA^{1D}**	**NBE/A^{1D}**
8	Immediate Grp 1^{1A}				TEST		XCHG	
	Eb, Ib	Ev, Iv	Eb, Ib	Ev, Ib	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv
9	**NOP^{1D}**	**XCHG word or double-word register with eAX^{1D}**						
		eCX	eDX	eBX	eSP	eBP	eSI	eDI
A	**MOV^{1D}**				**MOVS/ MOVSB Yb, Xb^{1D}**	**MOVS/ MOVSW/ MOVSD Yv, Xv^{1D}**	**CMPS/ CMPSB Yb, Xb^{1D}**	**CMPS/ CMPSW/ CMPSD Xv, Yv^{1D}**
	AL, Ob	eAX, Ov	Ob, AL	Ov, eAX				
B	**MOV immediate byte into byte register^{1D}**							
	AL	CL	DL	BL	AH	CH	DH	BH
C	Shift Grp 2^{1A}		**RET Iw^{1D}**	**RET^{1D}**	LES Gv, Mp	LDS Gv, Mp	Grp 11^{1A} - MOV	
	Eb, Ib	Ev, Ib					Eb, Ib	Ev, Iv
D	Shift Grp 2^{1A}				**AAM Ib^{1D}**	**AAD Ib^{1D}**		**XLAT/ XLATB^{1D}**
	Eb, 1	Ev, 1	Eb, CL	Ev, CL				
E	**LOOPNE/ LOOPNZ Jb^{1D}**	**LOOPE/ LOOPZ Jb^{1D}**	**LOOP Jb^{1D}**	**JCXZ/ JECXZ Jb^{1D}**	**IN**		**OUT**	
					AL, Ib^{1D}	**eAX, Ib^{1D}**	**Ib, AL^{1D}**	**Ib, eAX^{1D}**
F	**LOCK Prefix**		**REPNE Prefix**	**REP/ REPE Prefix**	**HLT^{1D}**	**CMC^{1D}**	Unary Grp 3^{1A}	
							Eb	Ev

NOTES:

† All blanks in the opcode map shown in Table A-2 are reserved and should not be used. Do not depend on the operation of these undefined or reserved opcodes.

†† To use the table, take the opcode's first Hex character from the row designation and the second character from the column designation. For example: 07H for [POP ES].

Table A-2. One-byte Opcode Map (Continued)

	8	9	A	B	C	D	E	F
0	OR						**PUSH CS^{1D}**	**Escape opcode to 2-byte**
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	**AL, Ib^{1D}**	**eAX, Iv^{1D}**		
1	SBB						**PUSH DS^{1D}**	**POP DS^{1D}**
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	**AL, Ib^{1D}**	**eAX, Iv^{1D}**		
2	SUB						**SEG=CS Prefix**	**DAS^{1D}**
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	**AL, Ib^{1D}**	**eAX, Iv^{1D}**		
3	CMP						**SEG=DS Prefix**	**AAS^{1D}**
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	**AL, Ib^{1D}**	**eAX, Iv^{1D}**		
4	DEC general register							
	eAX^{1D}	eCX^{1D}	eDX^{1D}	eBX^{1D}	eSP^{1D}	eBP^{1D}	eSI^{1D}	eDI^{1D}
5	**POP into general register^{1D}**							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
6	**PUSH Iv^{1D}**	IMUL Gv, Ev, Iv	**PUSH Ib^{1D}**	IMUL Gv, Ev, Ib	**INS/ INSB Yb, DX^{1D}**	**INS/ INSW/ INSD Yv, DX^{1D}**	**OUTS/ OUTSB DX, Xb^{1D}**	**OUTS/ OUTSW/ OUTSD DX, Xv^{1D}**
7	Jcc, Jb- Short displacement jump on condition							
	S^{1D}	**NS^{1D}**	**P/PE^{1D}**	**NP/PO^{1D}**	**L/NGE^{1D}**	**NL/GE^{1D}**	**LE/NG^{1D}**	**NLE/G^{1D}**
8	MOV				MOV Ew, Sw	LEA Gv, M	MOV Sw, Ew	POP Ev
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev				
9	**CBW/ CWDE^{1D}**	**CWD/ CDQ^{1D}**	**CALLF Ap^{1D}**	**FWAIT/ WAIT^{1D}**	**PUSHF/ PUSHFD Fv^{1D}**	**POPF/ POPFD Fv^{1D}**	**SAHF^{1D}**	**LAHF^{1D}**
A	**TEST^{1D}**		**STOS/ STOSB Yb, AL ^{1D}**	**STOS/ STOSW/ STOSD Yv, eAX^{1D}**	**LODS/ LODSB AL, Xb^{1D}**	**LODS/ LODSW/ LODSD eAX, Xv^{1D}**	**SCAS/ SCASB AL, Yb^{1D}**	**SCAS/ SCASW/ SCASD eAX, Yv^{1D}**
	AL, Ib	eAX, Iv						
B	**MOV immediate word or double into word or double register^{1D}**							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
C	**ENTER Iw, Ib^{1D}**	**LEAVE^{1D}**	**RETF Iw^{1D}**	**RETF^{1D}**	**INT 3^{1D}**	**INT Ib^{1D}**	**INTO^{1D}**	**IRET^{1D}**
D	ESC (Escape to coprocessor instruction set)							
E	**CALL**	**JMP**			**IN**		**OUT**	
	Jv^{1D}	**near Jv^{1D}**	**far Ap^{1D}**	**short Jb^{1D}**	**AL, DX^{1D}**	**eAX, DX^{1D}**	**DX, AL^{1D}**	**DX, eAX^{1D}**
F	**CLC^{1D}**	**STC^{1D}**	**CLI^{1D}**	**STI^{1D}**	**CLD^{1D}**	**STD^{1D}**	INC/DEC Grp 4^{1A}	INC/DEC Grp 5^{1A}

Table A-3. Two-byte Opcode Map (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			**CLTS[1D]**	
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) Wsd, Vsd	**MOVLPS Vq, Mq[1F] MOVLPD (66) Vq, Mq[1F] MOVHLPS Vps, Vps MOVDDUP (F2) Vq, Wq[1G] MOVSLDUP (F3) Vps, Wps**	**MOVLPS Mq, Vq[1F] MOVLPD (66) Mq, Vq[1F]**	**UNPCKLPS Vps, Wps UNPCKLPD (66) Vpd, Wpd**	**UNPCKHPS Vps, Wps UNPCKHPD (66) Vpd, Wpd**	**MOVHPS Vq, Mq[1F] MOVHPD (66) Vq, Mq[1F] MOVLHPS Vps, Vps MOVSHDUP (F3) Vps, Wps**	**MOVHPS Mq, Vps[1F] MOVHPD (66) Mq, Vpd[1F]**
2	**MOV Rd, Cd[1H]**	**MOV Rd, Dd[1H]**	**MOV Cd, Rd[1H]**	**MOV Dd, Rd[1H]**	MOV Rd, Td†††		MOV Td, Rd†††	
3	**WRMSR[1D]**	**RDTSC[1D]**	**RDMSR[1D]**	**RDPMC[1D]**	**SYSENTER[1D]**	**SYSEXIT[1D]**		
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, Vps[1H] MOVMSKPD (66) Gd, Vpd[1H]**	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[1D]**

NOTES:

† All blanks in the opcode map shown in Table A-3 are reserved and should not be used. Do not depend on the operation of these undefined or reserved opcodes.

†† To use the table, use 0FH for the first byte of the opcode. For the second byte, take the first Hex character from the row designation and the second character from the column designation. For example: 0F03H for [LSL GV, EW].

††† Not currently supported after Pentium Pro and Pentium II families. Using this opcode on the current generation of processors will generate a #UD. For future processors, this value is reserved.

Table A-3. Two-byte Opcode Map (Proceeding Byte is 0FH)

	8	9	A	B	C	D	E	F
0	**INVD[1D]**	**WBINVD[1D]**		UD2				
1	PREFETCH[1C] (Grp 16[1A])							
2	MOVAPS Vps, Wps MOVAPD (66) Vpd, Wpd	MOVAPS Wps, Vps MOVAPD (66) Wpd, Vpd	CVTPI2PS Vps, Qq CVTSI2SS (F3) Vss, Ed CVTPI2PD (66) Vpd, Qq CVTSI2SD (F2) Vsd, Ed	**MOVNTPS** **Mps, Vps[1F]** **MOVNTPD** **(66)** **Mpd, Vpd[1F]**	CVTTPS2PI Pq, Wq CVTTSS2SI (F3) Gd, Wss CVTTPD2PI (66) Pq, Wpd CVTTSD2SI (F2) Gd, Wsd	CVTPS2PI Pq, Wq CVTSS2SI (F3) Gd, Wss CVTPD2PI (66) Pq, Wpd CVTSD2SI (F2) Gd, Wsd	UCOMISS Vss, Wss UCOMISD (66) Vsd, Wsd	COMISS Vps, Wps COMISD (66) Vsd, Wsd
3								
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, Wq CVTSS2SD (F3) Vsd, Wss CVTPD2PS (66) Vps, Wpd CVTSD2SS (F2) Vss, 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, Qq PUNPCKHBW (66) Vdq, Qdq	PUNPCKHWD Pq, Qq PUNPCKHWD (66) Vdq, Qdq	PUNPCKHDQ Pq, Qq PUNPCKHDQ (66) Vdq, Qdq	PACKSSDW Pq, Qq PACKSSDW (66) Vdq, Qdq	PUNPCKLQDQ (66) Vdq, Wdq	PUNPCKHQD Q (66) Vdq, Wdq	MOVD Pd, Ed MOVD (66) Vd, Ed	MOVQ Pq, Qq MOVDQA (66) Vdq, Wdq MOVDQU (F3) Vdq, Wdq
7	MMX UD (Reserved for future use)				HADDPD (66) Vpd, Wpd HADDPS (F2) Vps, Wps	HSUBPD (66) Vpd, Wpd HSUBPS (F2) Vps, Wps	MOVD Ed, Pd MOVD (66) Ed, Vd MOVQ (F3) Vq, Wq	MOVQ Qq, Pq MOVDQA (66) Wdq, Vdq MOVDQU (F3) Wdq, Vdq

Table A-3. Two-byte Opcode Map (Proceeding Byte is 0FH)

	0	1	2	3	4	5	6	7
8	**Jcc, Jv - Long-displacement jump on condition**							
	O[1D]	**NO[1D]**	**B/C/NAE[1D]**	**AE/NB/NC[1D]**	**E/Z[1D]**	**NE/NZ[1D]**	**BE/NA[1D]**	**A/NBE[1D]**
9	**SETcc, Eb - Byte Set on condition (000)[1K]**							
	O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
A	**PUSH FS[1D]**	**POP FS[1D]**	**CPUID[1D]**	BT Ev, Gv	SHLD Ev, Gv, Ib	SHLD Ev, Gv, CL		
B	CMPXCHG		LSS Mp	BTR Ev, Gv	LFS Mp	LGS 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, Gd[1F]**	**PINSRW Pw, Ew, Ib PINSRW (66) Vw, Ew, Ib**	**PEXTRW Gw, Pw, Ib[1H] PEXTRW (66) Gw, Vw, Ib[1H]**	SHUFPS Vps, Wps, Ib SHUFPD (66) Vpd, Wpd, Ib	Grp 9[1A]
D	**ADDSUBPD (66) Vpd, Wpd ADDSUBPS (F2) Vps, Wps**	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, Qq[1H] MOVDQ2Q (F2) Pq, Vq[1H]**	**PMOVMSKB Gd, Pq[1H] PMOVMSKB (66) Gd, Vdq[1H]**
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, Wq**	**MOVNTQ Mq, Vq[1F] MOVNTDQ (66) Mdq, Vdq[1F]**
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, Pq[1H] MASKMOV- DQU (66) Vdq, Vdq[1H]**

Table A-3. Two-byte Opcode Map (Proceeding Byte is 0FH)

	8	9	A	B	C	D	E	F
8	**Jcc, Jv - Long-displacement jump on condition**							
	S[1D]	**NS[1D]**	**P/PE[1D]**	**NP/PO[1D]**	**L/NGE[1D]**	**NL/GE[1D]**	**LE/NG[1D]**	**NLE/G[1D]**
9	**SETcc, Eb - Byte Set on condition (000)[1K]**							
	S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
A	**PUSH GS[1D]**	**POP GS[1D]**	**RSM[1D]**	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	Gv, Ew
C	**BSWAP[1D]**							
	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
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	

A.3.4. Opcode Extensions For One- And Two-byte Opcodes

Some of the 1-byte and 2-byte opcodes use bits 5, 4, and 3 of the ModR/M byte (the nnn field in Figure A-1) as an extension of the opcode. The value of bits 5, 4, and 3 of the ModR/M byte also corresponds to “/digit“ portion of the opcode notation described in Chapter 3. Those opcodes that have opcode extensions are indicated in Table A-4 with group numbers (Group 1, Group 2, etc.). The group numbers (ranging from 1 to 16) in the second column provide an entry point into Table A-4 where the encoding of the opcode extension field can be found. The valid encoding the r/m field of the ModR/M byte for each instruction can be inferred from the 3rd column.

For example, the ADD instruction with a 1-byte opcode of 80H is a Group 1 instruction. Table A-4 indicates that the opcode extension field that must be encoded in the ModR/M byte for this instruction is 000B. The r/m field for this instruction can be encoded to access a register (11B); or a memory address using addressing modes, i.e.. mem = 00B, 01B, 10B.

mod	nnn	R/M

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

Table A-4. 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							
			000	001	010	011	100	101	110	111
80-83	1	mem, 11B	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
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/Iv		NOT	NEG	MUL AL/eAX	IMUL AL/eAX	DIV AL/eAX	IDIV AL/eAX
FE	4	mem, 11B	INC Eb	DEC Eb						
FF	5	mem, 11B	INC Ev	DEC Ev	CALLN Ev	CALLF Ep ^{1J}	JMPN Ev	JMPF Ep^{1J}	PUSH Ev	
OF OO	6	mem, 11B	SLDT Ew	STR Ev	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
OF 01	7	mem	SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Ew		LMSW Ew	INVLPG Mb
		11B		MONITOR eAX, eCX, eDX (000)1E MWAIT eAX, eCX (001)1E						
OF BA	8	mem, 11B					BT	BTS	BTR	BTC
OF C7	9	mem		CMPXCH8B Mq						
		11B								
OF B9	10	mem								
		11B								

Table A-4. Opcode Extensions for One- and Two-byte Opcodes by Group Number (Contd.)

C6	11	mem, 11B	MOV Eb, Ib							
C7		mem, 11B	MOV Ev, Iv							
OF 71	12	mem								
		11B			PSRLW Pq, Ib PSRLW (66) Pdq, Ib		PSRAW Pq, Ib PSRAW (66) Pdq, Ib		PSLLW Pq, Ib PSLLW (66) Pdq, Ib	
OF 72	13	mem								
		11B			PSRLD Pq, Ib PSRLD (66) Wdq, Ib		PSRAD Pq, Ib PSRAD (66) Wdq, Ib		PSLLD Pq, Ib PSLLD (66) Wdq, Ib	
OF 73	14	mem								
		11B			PSRLQ Pq, Ib PSRLQ (66) Wdq, Ib	PSRLDQ (66) Wdq, Ib			PSLLQ Pq, Ib PSLLQ (66) Wdq, Ib	PSLLDQ (66) Wdq, Ib
OF AE	15	mem	FXSAVE	FXRSTOR	LDMXCSR	STMXCSR				CLFLUSH
		11B						LFENCE (000) ^{1E}	MFENCE (000) ^{1E}	SFENCE (000) ^{1E}
OF 18	16	mem	PREFETCH- NTA	PREFETCH- T0	PREFETCH- T1	PREFETCH- T2				
		11B								

NOTE:

All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined or reserved opcodes.

A.3.5. Escape Opcode Instructions

The opcode maps for the coprocessor escape instruction opcodes (x87 floating-point instruction opcodes) are given in Table A-5 through Table A-20. These opcode maps are grouped by the first byte of the opcode from D8 through DF. Each of these opcodes has a ModR/M byte. If the ModR/M byte is within the range of 00H through BFH, bits 5, 4, and 3 of the ModR/M byte are used as an opcode extension, similar to the technique used for 1-and 2-byte opcodes (refer to Section A.3.4., "Opcode Extensions For One- And Two-byte Opcodes"). 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.3.5.1. OPCODES WITH MODR/M BYTES IN THE 00H THROUGH BFH RANGE

The opcode DD0504000000H can be interpreted as follows. The instruction encoded with this opcode can be located in Section A.3.5.8., "Escape Opcodes with DD as First Byte". Since the ModR/M byte (05H) is within the 00H through BFH range, bits 3 through 5 (000) of this byte indicate the opcode to be for an FLD double-real instruction (refer to Table A-7). The double-real value to be loaded is at 00000004H, which is the 32-bit displacement that follows and belongs to this opcode.

A.3.5.2. OPCODES WITH MODR/M BYTES OUTSIDE THE 00H THROUGH BFH RANGE

The opcode D8C1H illustrates an opcode with a ModR/M byte outside the range of 00H through BFH. The instruction encoded here, can be located in Section A.3.4., "Opcode Extensions For One- And Two-byte Opcodes". In Table A-6, the ModR/M byte C1H indicates row C, column 1, which is an FADD instruction using ST(0), ST(1) as the operands.

A.3.5.3. ESCAPE OPCODES WITH D8 AS FIRST BYTE

Table A-5 and Table A-6 contain the opcodes maps for the escape instruction opcodes that begin with D8H. Table A-5 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (the nnn field in Figure A-1) selects the instruction.

Table A-5. D8 Opcode Map When ModR/M Byte is Within 00H to BFH[1]

nnn Field of ModR/M Byte (refer to Figure A-1)							
000B	001B	010B	011B	100B	101B	110B	111B

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-6 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

Table A-6. D8 Opcode Map When ModR/M Byte is Outside 00H to BFH[1]

	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)

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

A.3.5.4. ESCAPE OPCODES WITH D9 AS FIRST BYTE

Table A-7 and Table A-8 contain opcodes maps for escape instruction opcodes that begin with D9H. Table A-7 shows the opcode map if the accompanying ModR/M byte is within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (the Figure A-1 nnn field) selects the instruction.

Table A-7. D9 Opcode Map When ModR/M Byte is Within 00H to BFH[1].

nnn Field of ModR/M Byte (refer to Figure A-1)							
000B	001B	010B	011B	100B	101B	110B	111B
FLD single-real		FST single-real	FSTP single-real	FLDENV 14/28 bytes	FLDCW 2 bytes	FNSTENV 14/28 bytes	FNSTCW 2 bytes

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-8 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

Table A-8. D9 Opcode Map When ModR/M Byte is Outside 00H to BFH[1]

	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

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

A.3.5.5. ESCAPE OPCODES WITH DA AS FIRST BYTE

Table A-9 and Table A-10 contain the opcodes maps for the escape instruction opcodes that begin with DAH. Table A-9 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (the nnn field in Figure A-1) selects the instruction.

Table A-9. DA Opcode Map When ModR/M Byte is Within 00H to BFH[1]

nnn Field of ModR/M Byte (refer to Figure A-1)							
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

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-10 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

Table A-10. DA Opcode Map When ModR/M Byte is Outside 00H to BFH[1]

	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								

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

A.3.5.6. ESCAPE OPCODES WITH DB AS FIRST BYTE

Table A-11 and Table A-12 contain the opcodes maps for the escape instruction opcodes that begin with DBH. Table A-11 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (the nnn field in Figure A-1) selects the instruction.

Table A-11. DB Opcode Map When ModR/M Byte is Within 00H to BFH[1]

nnn Field of ModR/M Byte (refer to Figure A-1)							
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

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-12 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

Table A-12. DB Opcode Map When ModR/M Byte is Outside 00H to BFH[1]

	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			FNCLEX	FNINIT				
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								

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

A.3.5.7. ESCAPE OPCODES WITH DC AS FIRST BYTE

Table A-13 and Table A-14 contain the opcodes maps for the escape instruction opcodes that begin with DCH. Table A-13 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (the nnn field in Figure A-1) selects the instruction.

Table A-13. DC Opcode Map When ModR/M Byte is Within 00H to BFH[1]

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

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-14 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

Table A-14. DC Opcode Map When ModR/M Byte is Outside 00H to BFH[4]

	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)

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

A.3.5.8. ESCAPE OPCODES WITH DD AS FIRST BYTE

Table A-15 and Table A-16 contain the opcodes maps for the escape instruction opcodes that begin with DDH. Table A-15 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (the nnn field in Figure A-1) selects the instruction.

Table A-15. DD Opcode Map When ModR/M Byte is Within 00H to BFH[1]

nnn Field of ModR/M Byte (refer to Figure A-1)							
000B	001B	010B	011B	100B	101B	110B	111B

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

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

Table A-16. DD Opcode Map When ModR/M Byte is Outside 00H to BFH[1]

	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								

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

A.3.5.9. ESCAPE OPCODES WITH DE AS FIRST BYTE

Table A-17 and Table A-18 contain the opcodes maps for the escape instruction opcodes that begin with DEH. Table A-17 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (the nnn field in Figure A-1) selects the instruction.

Table A-17. DE Opcode Map When ModR/M Byte is Within 00H to BFH[1]

nnn Field of ModR/M Byte (refer to Figure A-1)							
000B	001B	010B	011B	100B	101B	110B	111B

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-18 shows the opcode map if the accompanying ModR/M byte is outside the range of 00H to BFH. In this case the first digit of the ModR/M byte selects the row in the table and the second digit selects the column.

Table A-18. DE Opcode Map When ModR/M Byte is Outside 00H to BFH[1]

	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)

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

A.3.5.10. ESCAPE OPCODES WITH DF AS FIRST BYTE

Table A-19 and Table A-20 contain the opcodes maps for the escape instruction opcodes that begin with DFH. Table A-19 shows the opcode map if the accompanying ModR/M byte within the range of 00H through BFH. Here, the value of bits 5, 4, and 3 (the nnn field in Figure A-1) selects the instruction.

Table A-19. DF Opcode Map When ModR/M Byte is Within 00H to BFH[1]

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

NOTE:

1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

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

Table A-20. DF Opcode Map When ModR/M Byte is Outside 00H to BFH[1]

	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								

NOTE:

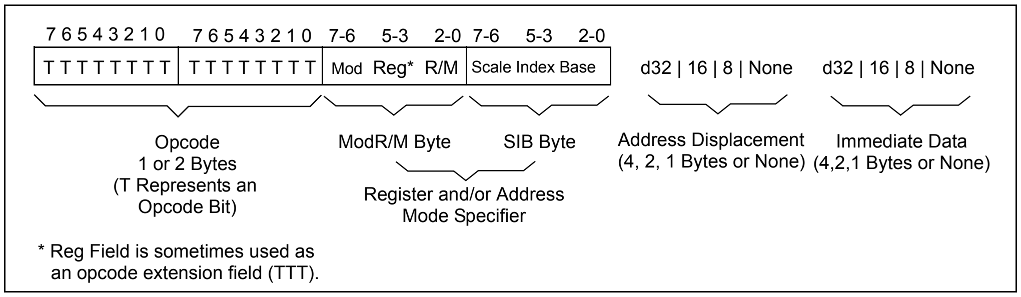
1. All blanks in the opcode map are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

APPENDIX B INSTRUCTION FORMATS AND ENCODINGS

This appendix shows the machine instruction formats and encodings of the IA-32 architecture instructions. The first section describes in detail the IA-32 architecture's machine instruction format. The following sections show the formats and encoding of general-purpose, MMX, P6 family, SSE/SSE2/SSE3, and x87 FPU instructions.

B.1. MACHINE INSTRUCTION FORMAT

All Intel Architecture instructions are encoded using subsets of the general machine instruction format shown in Figure B-1. Each instruction consists of an opcode, a register and/or address mode specifier (if required) consisting of the ModR/M byte and sometimes the scale-index-base (SIB) byte, a displacement (if required), and an immediate data field (if required).



7 6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
7-6 5-3 2-0
7-6 5-3 2-0
T T T T T T T T
T T T T T T T T
Mod Reg* R/M
Scale Index Base
d32 | 16 | 8 | None
d32 | 16 | 8 | None
Opcode 1 or 2 Bytes (T Represents an Opcode Bit)
ModR/M Byte
SIB Byte
Register and/or Address Mode Specifier
Address Displacement (4, 2, 1 Bytes or None)
Immediate Data (4,2,1 Bytes or None)
* Reg Field is sometimes used as an opcode extension field (TTT).

Figure B-1. General Machine Instruction Format

The primary opcode for an instruction is encoded in one or two bytes of the instruction. Some instructions also use an opcode extension field encoded in bits 5, 4, and 3 of the ModR/M byte. Within the primary opcode, smaller encoding fields may be defined. These fields vary according to the class of operation being performed. The fields define such information as register encoding, conditional test performed, or sign extension of immediate byte.

Almost all instructions that refer to a register and/or memory operand have a register and/or address mode byte following the opcode. This byte, the ModR/M byte, consists of the mod field, the reg field, and the R/M field. Certain encodings of the ModR/M byte indicate that a second address mode byte, the SIB byte, must be used.

If the selected addressing mode specifies a displacement, the displacement value is placed immediately following the ModR/M byte or SIB byte. If a displacement is present, the possible sizes are 8, 16, or 32 bits.

If the instruction specifies an immediate operand, the immediate value follows any displacement bytes. An immediate operand, if specified, is always the last field of the instruction.

Table B-1 lists several smaller fields or bits that appear in certain instructions, sometimes within the opcode bytes themselves. The following tables describe these fields and bits and list the allowable values. All of these fields (except the d bit) are shown in the general-purpose instruction formats given in Table B-11.

Table B-1. Special Fields Within Instruction Encodings

Field Name	Description	Number of Bits
reg	General-register specifier (see Table B-2 or B-3)	3
w	Specifies if data is byte or full-sized, where full-sized is either 16 or 32 bits (see Table B-4)	1
s	Specifies sign extension of an immediate data field (see Table B-5)	1
sreg2	Segment register specifier for CS, SS, DS, ES (see Table B-6)	2
sreg3	Segment register specifier for CS, SS, DS, ES, FS, GS (see Table B-6)	3
eee	Specifies a special-purpose (control or debug) register (see Table B-7)	3
tttn	For conditional instructions, specifies a condition asserted or a condition negated (see Table B-8)	4
d	Specifies direction of data operation (see Table B-9)	1

B.1.1. Reg Field (reg)

The reg field in the ModR/M byte specifies a general-purpose register operand. The group of registers specified is modified by the presence of and state of the w bit in an encoding (see Table B-4). Table B-2 shows the encoding of the reg field when the w bit is not present in an encoding, and Table B-3 shows the encoding of the reg field when the w bit is present.

Table B-2. Encoding of reg Field When w Field is Not Present in Instruction

reg Field	Register Selected during 16-Bit Data Operations	Register Selected during 32-Bit Data Operations
000	AX	EAX
001	CX	ECX
010	DX	EDX
011	BX	EBX
100	SP	ESP
101	BP	EBP
110	SI	ESI
111	DI	EDI

Table B-3. Encoding of reg Field When w Field is Present in Instruction

Register Specified by reg Field during 16-Bit Data Operations		
	Function of w Field	
reg	When w = 0	When w = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Register Specified by reg Field during 32-Bit Data Operations		
	Function of w Field	
reg	When w = 0	When w = 1
000	AL	EAX
001	CL	ECX
010	DL	EDX
011	BL	EBX
100	AH	ESP
101	CH	EBP
110	DH	ESI
111	BH	EDI

B.1.2. Encoding of Operand Size Bit (w)

The current operand-size attribute determines whether the processor is performing 16-or 32-bit operations. Within the constraints of the current operand-size attribute, the operand-size bit (w) can be used to indicate operations on 8-bit operands or the full operand size specified with the operand-size attribute (16 bits or 32 bits). Table B-4 shows the encoding of the w bit depending on the current operand-size attribute.

Table B-4. Encoding of Operand Size (w) Bit

w Bit	Operand Size When Operand-Size Attribute is 16 bits	Operand Size When Operand-Size Attribute is 32 bits
0	8 Bits	8 Bits
1	16 Bits	32 Bits

B.1.3. Sign Extend (s) Bit

The sign-extend (s) bit occurs primarily in instructions with immediate data fields that are being extended from 8 bits to 16 or 32 bits. Table B-5 shows the encoding of the s bit.

Table B-5. Encoding of Sign-Extend (s) Bit

s	Effect on 8-Bit Immediate Data	Effect on 16- or 32-Bit Immediate Data
0	None	None
1	Sign-extend to fill 16-bit or 32-bit destination	None

B.1.4. Segment Register Field (sreg)

When an instruction operates on a segment register, the reg field in the ModR/M byte is called the sreg field and is used to specify the segment register. Table B-6 shows the encoding of the sreg field. This field is sometimes a 2-bit field (sreg2) and other times a 3-bit field (sreg3).

Table B-6. Encoding of the Segment Register (sreg) Field

2-Bit sreg2 Field	Segment Register Selected
00	ES
01	CS
10	SS
11	DS

3-Bit sreg3 Field	Segment Register Selected
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS
110	Reserved*
111	Reserved*

* Do not use reserved encodings.

B.1.5. Special-Purpose Register (eee) Field

When the control or debug registers are referenced in an instruction they are encoded in the eee field, which is located in bits 5, 4, and 3 of the ModR/M byte. Table B-7 shows the encoding of the eee field.

Table B-7. Encoding of Special-Purpose Register (eee) Field

eee	Control Register	Debug Register
000	CR0	DR0
001	Reserved*	DR1
010	CR2	DR2
011	CR3	DR3
100	CR4	Reserved*
101	Reserved*	Reserved*
110	Reserved*	DR6
111	Reserved*	DR7

* Do not use reserved encodings.

B.1.6. Condition Test Field (tttn)

For conditional instructions (such as conditional jumps and set on condition), the condition test field (tttn) is encoded for the condition being tested for. The ttt part of the field gives the condition to test and the n part indicates whether to use the condition (n = 0) or its negation (n = 1). For 1-byte primary opcodes, the tttn field is located in bits 3,2,1, and 0 of the opcode byte; for 2-byte primary opcodes, the tttn field is located in bits 3,2,1, and 0 of the second opcode byte. Table B-8 shows the encoding of the tttn field.

Table B-8. Encoding of Conditional Test (tttn) Field

t t t n	Mnemonic	Condition
0000	O	Overflow
0001	NO	No overflow
0010	B, NAE	Below, Not above or equal
0011	NB, AE	Not below, Above or equal
0100	E, Z	Equal, Zero
0101	NE, NZ	Not equal, Not zero
0110	BE, NA	Below or equal, Not above
0111	NBE, A	Not below or equal, Above
1000	S	Sign
1001	NS	Not sign
1010	P, PE	Parity, Parity Even
1011	NP, PO	Not parity, Parity Odd
1100	L, NGE	Less than, Not greater than or equal to
1101	NL, GE	Not less than, Greater than or equal to
1110	LE, NG	Less than or equal to, Not greater than
1111	NLE, G	Not less than or equal to, Greater than

B.1.7. Direction (d) Bit

In many two-operand instructions, a direction bit (d) indicates which operand is considered the source and which is the destination. Table B-9 shows the encoding of the d bit. When used for integer instructions, the d bit is located at bit 1 of a 1-byte primary opcode. This bit does not appear as the symbol "d" in Table B-11; instead, the actual encoding of the bit as 1 or 0 is given. When used for floating-point instructions (in Table B-16), the d bit is shown as bit 2 of the first byte of the primary opcode.

Table B-9. Encoding of Operation Direction (d) Bit

d	Source	Destination
0	reg Field	ModR/M or SIB Byte
1	ModR/M or SIB Byte	reg Field

B.1.8. Other Notes

Table B-10 contains notes on particular encodings. These notes are indicated in the tables shown in the following sections by superscripts.

Table B-10. Notes on Instruction Encoding

Symbol	Note
A	A value of 11B in bits 7, and 6 of the ModR/M byte is reserved.

B.2. GENERAL-PURPOSE INSTRUCTION FORMATS AND ENCODINGS

Table B-11 shows the machine instruction formats and encodings of the general purpose instructions.

Table B-11. General Purpose Instruction Formats and Encodings

Instruction and Format	Encoding
AAA – ASCII Adjust after Addition	0011 0111
AAD – ASCII Adjust AX before Division	1101 0101 : 0000 1010
AAM – ASCII Adjust AX after Multiply	1101 0100 : 0000 1010
AAS – ASCII Adjust AL after Subtraction	0011 1111
ADC – ADD with Carry	
register1 to register2	0001 000w : 11 reg1 reg2
register2 to register1	0001 001w : 11 reg1 reg2
memory to register	0001 001w : mod reg r/m
register to memory	0001 000w : mod reg r/m
immediate to register	1000 00sw : 11 010 reg : immediate data
immediate to AL, AX, or EAX	0001 010w : immediate data
immediate to memory	1000 00sw : mod 010 r/m : immediate data

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
ADD – Add	
register1 to register2	0000 000w : 11 reg1 reg2
register2 to register1	0000 001w : 11 reg1 reg2
memory to register	0000 001w : mod reg r/m
register to memory	0000 000w : mod reg r/m
immediate to register	1000 00sw : 11 000 reg : immediate data
immediate to AL, AX, or EAX	0000 010w : immediate data
immediate to memory	1000 00sw : mod 000 r/m : immediate data
AND – Logical AND	
register1 to register2	0010 000w : 11 reg1 reg2
register2 to register1	0010 001w : 11 reg1 reg2
memory to register	0010 001w : mod reg r/m
register to memory	0010 000w : mod reg r/m
immediate to register	1000 00sw : 11 100 reg : immediate data
immediate to AL, AX, or EAX	0010 010w : immediate data
immediate to memory	1000 00sw : mod 100 r/m : immediate data
ARPL – Adjust RPL Field of Selector	
from register	0110 0011 : 11 reg1 reg2
from memory	0110 0011 : mod reg r/m
BOUND – Check Array Against Bounds	0110 0010 : mod^{A} reg r/m
BSF – Bit Scan Forward	
register1, register2	0000 1111 : 1011 1100 : 11 reg1 reg2
memory, register	0000 1111 : 1011 1100 : mod reg r/m
BSR – Bit Scan Reverse	
register1, register2	0000 1111 : 1011 1101 : 11 reg1 reg2
memory, register	0000 1111 : 1011 1101 : mod reg r/m
BSWAP – Byte Swap	0000 1111 : 1100 1 reg
BT – Bit Test	
register, immediate	0000 1111 : 1011 1010 : 11 100 reg: imm8 data
memory, immediate	0000 1111 : 1011 1010 : mod 100 r/m : imm8 data
register1, register2	0000 1111 : 1010 0011 : 11 reg2 reg1
memory, reg	0000 1111 : 1010 0011 : mod reg r/m

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
BTC – Bit Test and Complement	
register, immediate	0000 1111 : 1011 1010 : 11 111 reg: imm8 data
memory, immediate	0000 1111 : 1011 1010 : mod 111 r/m : imm8 data
register1, register2	0000 1111 : 1011 1011 : 11 reg2 reg1
memory, reg	0000 1111 : 1011 1011 : mod reg r/m
BTR – Bit Test and Reset	
register, immediate	0000 1111 : 1011 1010 : 11 110 reg: imm8 data
memory, immediate	0000 1111 : 1011 1010 : mod 110 r/m : imm8 data
register1, register2	0000 1111 : 1011 0011 : 11 reg2 reg1
memory, reg	0000 1111 : 1011 0011 : mod reg r/m
BTS – Bit Test and Set	
register, immediate	0000 1111 : 1011 1010 : 11 101 reg: imm8 data
memory, immediate	0000 1111 : 1011 1010 : mod 101 r/m : imm8 data
register1, register2	0000 1111 : 1010 1011 : 11 reg2 reg1
memory, reg	0000 1111 : 1010 1011 : mod reg r/m
CALL – Call Procedure (in same segment)	
direct	1110 1000 : full displacement
register indirect	1111 1111 : 11 010 reg
memory indirect	1111 1111 : mod 010 r/m
CALL – Call Procedure (in other segment)	
direct	1001 1010 : unsigned full offset, selector
indirect	1111 1111 : mod 011 r/m
CBW – Convert Byte to Word	1001 1000
CDQ – Convert Doubleword to Qword	1001 1001
CLC – Clear Carry Flag	1111 1000
CLD – Clear Direction Flag	1111 1100
CLI – Clear Interrupt Flag	1111 1010
CLTS – Clear Task-Switched Flag in CR0	0000 1111 : 0000 0110
CMC – Complement Carry Flag	1111 0101

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
CMP – Compare Two Operands	
register1 with register2	0011 100w : 11 reg1 reg2
register2 with register1	0011 101w : 11 reg1 reg2
memory with register	0011 100w : mod reg r/m
register with memory	0011 101w : mod reg r/m
immediate with register	1000 00sw : 11 111 reg : immediate data
immediate with AL, AX, or EAX	0011 110w : immediate data
immediate with memory	1000 00sw : mod 111 r/m : immediate data
CMPS/CMPSB/CMPSW/CMPSD – Compare String Operands	1010 011w
CMPXCHG – Compare and Exchange	
register1, register2	0000 1111 : 1011 000w : 11 reg2 reg1
memory, register	0000 1111 : 1011 000w : mod reg r/m
CPUID – CPU Identification	0000 1111 : 1010 0010
CWD – Convert Word to Doubleword	1001 1001
CWDE – Convert Word to Doubleword	1001 1000
DAA – Decimal Adjust AL after Addition	0010 0111
DAS – Decimal Adjust AL after Subtraction	0010 1111
DEC – Decrement by 1	
register	1111 111w : 11 001 reg
register (alternate encoding)	0100 1 reg
memory	1111 111w : mod 001 r/m
DIV – Unsigned Divide	
AL, AX, or EAX by register	1111 011w : 11 110 reg
AL, AX, or EAX by memory	1111 011w : mod 110 r/m
ENTER – Make Stack Frame for High Level Procedure	1100 1000 : 16-bit displacement : 8-bit level (L)
HLT – Halt	1111 0100
IDIV – Signed Divide	
AL, AX, or EAX by register	1111 011w : 11 111 reg
AL, AX, or EAX by memory	1111 011w : mod 111 r/m

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
IMUL – Signed Multiply	
AL, AX, or EAX with register	1111 011w : 11 101 reg
AL, AX, or EAX with memory	1111 011w : mod 101 reg
register1 with register2	0000 1111 : 1010 1111 : 11 : reg1 reg2
register with memory	0000 1111 : 1010 1111 : mod reg r/m
register1 with immediate to register2	0110 10s1 : 11 reg1 reg2 : immediate data
memory with immediate to register	0110 10s1 : mod reg r/m : immediate data
IN – Input From Port	
fixed port	1110 010w : port number
variable port	1110 110w
INC – Increment by 1	
reg	1111 111w : 11 000 reg
reg (alternate encoding)	0100 0 reg
memory	1111 111w : mod 000 r/m
INS – Input from DX Port	0110 110w
INT n – Interrupt Type n	1100 1101 : type
INT – Single-Step Interrupt 3	1100 1100
INTO – Interrupt 4 on Overflow	1100 1110
INVD – Invalidate Cache	0000 1111 : 0000 1000
INVLPG – Invalidate TLB Entry	0000 1111 : 0000 0001 : mod 111 r/m
IRET/IRETD – Interrupt Return	1100 1111
J*cc* – Jump if Condition is Met	
8-bit displacement	0111 tttn : 8-bit displacement
full displacement	0000 1111 : 1000 tttn : full displacement
JCXZ/JECXZ – Jump on CX/ECX Zero Address-size prefix differentiates JCXZ and JECXZ	1110 0011 : 8-bit displacement
JMP – Unconditional Jump (to same segment)	
short	1110 1011 : 8-bit displacement
direct	1110 1001 : full displacement
register indirect	1111 1111 : 11 100 reg
memory indirect	1111 1111 : mod 100 r/m

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
JMP – Unconditional Jump (to other segment)	
direct intersegment	1110 1010 : unsigned full offset, selector
indirect intersegment	1111 1111 : mod 101 r/m
LAHF – Load Flags into AHRegister	1001 1111
LAR – Load Access Rights Byte	
from register	0000 1111 : 0000 0010 : 11 reg1 reg2
from memory	0000 1111 : 0000 0010 : mod reg r/m
LDS – Load Pointer to DS	1100 0101 : mod^A reg r/m
LEA – Load Effective Address	1000 1101 : mod^A reg r/m
LEAVE – High Level Procedure Exit	1100 1001
LES – Load Pointer to ES	1100 0100 : mod^A reg r/m
LFS – Load Pointer to FS	0000 1111 : 1011 0100 : mod^A reg r/m
LGDT – Load Global Descriptor Table Register	0000 1111 : 0000 0001 : mod^A 010 r/m
LGS – Load Pointer to GS	0000 1111 : 1011 0101 : mod^A reg r/m
LIDT – Load Interrupt Descriptor Table Register	0000 1111 : 0000 0001 : mod^A 011 r/m
LLDT – Load Local Descriptor Table Register	
LDTR from register	0000 1111 : 0000 0000 : 11 010 reg
LDTR from memory	0000 1111 : 0000 0000 : mod 010 r/m
LMSW – Load Machine Status Word	
from register	0000 1111 : 0000 0001 : 11 110 reg
from memory	0000 1111 : 0000 0001 : mod 110 r/m
LOCK – Assert LOCK# Signal Prefix	1111 0000
LODS/LODSB/LODSW/LODSD – Load String Operand	1010 110w
LOOP – Loop Count	1110 0010 : 8-bit displacement
LOOPZ/LOOPE – Loop Count while Zero/Equal	1110 0001 : 8-bit displacement
LOOPNZ/LOOPNE – Loop Count while not Zero/Equal	1110 0000 : 8-bit displacement
LSL – Load Segment Limit	
from register	0000 1111 : 0000 0011 : 11 reg1 reg2
from memory	0000 1111 : 0000 0011 : mod reg r/m
LSS – Load Pointer to SS	0000 1111 : 1011 0010 : mod^A reg r/m

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
LTR – Load Task Register	
from register	0000 1111 : 0000 0000 : 11 011 reg
from memory	0000 1111 : 0000 0000 : mod 011 r/m
MOV – Move Data	
register1 to register2	1000 100w : 11 reg1 reg2
register2 to register1	1000 101w : 11 reg1 reg2
memory to reg	1000 101w : mod reg r/m
reg to memory	1000 100w : mod reg r/m
immediate to register	1100 011w : 11 000 reg : immediate data
immediate to register (alternate encoding)	1011 w reg : immediate data
immediate to memory	1100 011w : mod 000 r/m : immediate data
memory to AL, AX, or EAX	1010 000w : full displacement
AL, AX, or EAX to memory	1010 001w : full displacement
MOV – Move to/from Control Registers	
CR0 from register	0000 1111 : 0010 0010 : 11 000 reg
CR2 from register	0000 1111 : 0010 0010 : 11 010reg
CR3 from register	0000 1111 : 0010 0010 : 11 011 reg
CR4 from register	0000 1111 : 0010 0010 : 11 100 reg
register from CR0-CR4	0000 1111 : 0010 0000 : 11 eee reg
MOV – Move to/from Debug Registers	
DR0-DR3 from register	0000 1111 : 0010 0011 : 11 eee reg
DR4-DR5 from register	0000 1111 : 0010 0011 : 11 eee reg
DR6-DR7 from register	0000 1111 : 0010 0011 : 11 eee reg
register from DR6-DR7	0000 1111 : 0010 0001 : 11 eee reg
register from DR4-DR5	0000 1111 : 0010 0001 : 11 eee reg
register from DR0-DR3	0000 1111 : 0010 0001 : 11 eee reg
MOV – Move to/from Segment Registers	
register to segment register	1000 1110 : 11 sreg3 reg
register to SS	1000 1110 : 11 sreg3 reg
memory to segment reg	1000 1110 : mod sreg3 r/m
memory to SS	1000 1110 : mod sreg3 r/m
segment register to register	1000 1100 : 11 sreg3 reg
segment register to memory	1000 1100 : mod sreg3 r/m

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
MOVS/MOVSB/MOVSW/MOVSD – Move Data from String to String	1010 010w
MOVSX – Move with Sign-Extend	
register2 to register1	0000 1111 : 1011 111w : 11 reg1 reg2
memory to reg	0000 1111 : 1011 111w : mod reg r/m
MOVZX – Move with Zero-Extend	
register2 to register1	0000 1111 : 1011 011w : 11 reg1 reg2
memory to register	0000 1111 : 1011 011w : mod reg r/m
MUL – Unsigned Multiply	
AL, AX, or EAX with register	1111 011w : 11 100 reg
AL, AX, or EAX with memory	1111 011w : mod 100 reg
NEG – Two's Complement Negation	
register	1111 011w : 11 011 reg
memory	1111 011w : mod 011 r/m
NOP – No Operation	1001 0000
NOT – One's Complement Negation	
register	1111 011w : 11 010 reg
memory	1111 011w : mod 010 r/m
OR – Logical Inclusive OR	
register1 to register2	0000 100w : 11 reg1 reg2
register2 to register1	0000 101w : 11 reg1 reg2
memory to register	0000 101w : mod reg r/m
register to memory	0000 100w : mod reg r/m
immediate to register	1000 00sw : 11 001 reg : immediate data
immediate to AL, AX, or EAX	0000 110w : immediate data
immediate to memory	1000 00sw : mod 001 r/m : immediate data
OUT – Output to Port	
fixed port	1110 011w : port number
variable port	1110 111w
OUTS – Output to DX Port	0110 111w

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
POP – Pop a Word from the Stack	
register	1000 1111 : 11 000 reg
register (alternate encoding)	0101 1 reg
memory	1000 1111 : mod 000 r/m
POP – Pop a Segment Register from the Stack (Note: CS can not be sreg2 in this usage.)	
segment register DS, ES	000 sreg2 111
segment register SS	000 sreg2 111
segment register FS, GS	0000 1111: 10 sreg3 001
POPA/POPAD – Pop All General Registers	0110 0001
POPF/POPFD – Pop Stack into FLAGS or EFLAGS Register	1001 1101
PUSH – Push Operand onto the Stack	
register	1111 1111 : 11 110 reg
register (alternate encoding)	0101 0 reg
memory	1111 1111 : mod 110 r/m
immediate	0110 10s0 : immediate data
PUSH – Push Segment Register onto the Stack	
segment register CS,DS,ES,SS	000 sreg2 110
segment register FS,GS	0000 1111: 10 sreg3 000
PUSHA/PUSHAD – Push All General Registers	0110 0000
PUSHF/PUSHFD – Push Flags Register onto the Stack	1001 1100
RCL – Rotate thru Carry Left	
register by 1	1101 000w : 11 010 reg
memory by 1	1101 000w : mod 010 r/m
register by CL	1101 001w : 11 010 reg
memory by CL	1101 001w : mod 010 r/m
register by immediate count	1100 000w : 11 010 reg : imm8 data
memory by immediate count	1100 000w : mod 010 r/m : imm8 data
RCR – Rotate thru Carry Right	
register by 1	1101 000w : 11 011 reg
memory by 1	1101 000w : mod 011 r/m
register by CL	1101 001w : 11 011 reg

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
memory by CL	1101 001w : mod 011 r/m
register by immediate count	1100 000w : 11 011 reg : imm8 data
memory by immediate count	1100 000w : mod 011 r/m : imm8 data
RDMSR – Read from Model-Specific Register	0000 1111 : 0011 0010
RDPMC – Read Performance Monitoring Counters	0000 1111 : 0011 0011
RDTSC – Read Time-Stamp Counter	0000 1111 : 0011 0001
REP INS – Input String	1111 0011 : 0110 110w
REP LODS – Load String	1111 0011 : 1010 110w
REP MOVS – Move String	1111 0011 : 1010 010w
REP OUTS – Output String	1111 0011 : 0110 111w
REP STOS – Store String	1111 0011 : 1010 101w
REPE CMPS – Compare String	1111 0011 : 1010 011w
REPE SCAS – Scan String	1111 0011 : 1010 111w
REPNE CMPS – Compare String	1111 0010 : 1010 011w
REPNE SCAS – Scan String	1111 0010 : 1010 111w
RET – Return from Procedure (to same segment)	
no argument	1100 0011
adding immediate to SP	1100 0010 : 16-bit displacement
RET – Return from Procedure (to other segment)	
intersegment	1100 1011
adding immediate to SP	1100 1010 : 16-bit displacement
ROL – Rotate Left	
register by 1	1101 000w : 11 000 reg
memory by 1	1101 000w : mod 000 r/m
register by CL	1101 001w : 11 000 reg
memory by CL	1101 001w : mod 000 r/m
register by immediate count	1100 000w : 11 000 reg : imm8 data
memory by immediate count	1100 000w : mod 000 r/m : imm8 data
ROR – Rotate Right	
register by 1	1101 000w : 11 001 reg
memory by 1	1101 000w : mod 001 r/m
register by CL	1101 001w : 11 001 reg

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
memory by CL	1101 001w : mod 001 r/m
register by immediate count	1100 000w : 11 001 reg : imm8 data
memory by immediate count	1100 000w : mod 001 r/m : imm8 data
RSM – Resume from System Management Mode	0000 1111 : 1010 1010
SAHF – Store AH into Flags	1001 1110
SAL – Shift Arithmetic Left	same instruction as SHL
SAR – Shift Arithmetic Right	
register by 1	1101 000w : 11 111 reg
memory by 1	1101 000w : mod 111 r/m
register by CL	1101 001w : 11 111 reg
memory by CL	1101 001w : mod 111 r/m
register by immediate count	1100 000w : 11 111 reg : imm8 data
memory by immediate count	1100 000w : mod 111 r/m : imm8 data
SBB – Integer Subtraction with Borrow	
register1 to register2	0001 100w : 11 reg1 reg2
register2 to register1	0001 101w : 11 reg1 reg2
memory to register	0001 101w : mod reg r/m
register to memory	0001 100w : mod reg r/m
immediate to register	1000 00sw : 11 011 reg : immediate data
immediate to AL, AX, or EAX	0001 110w : immediate data
immediate to memory	1000 00sw : mod 011 r/m : immediate data
SCAS/SCASB/SCASW/SCASD – Scan String	1010 111w
SETcc – Byte Set on Condition	
register	0000 1111 : 1001 tttn : 11 000 reg
memory	0000 1111 : 1001 tttn : mod 000 r/m
SGDT – Store Global Descriptor Table Register	0000 1111 : 0000 0001 : mod^{A} 000 r/m
SHL – Shift Left	
register by 1	1101 000w : 11 100 reg
memory by 1	1101 000w : mod 100 r/m
register by CL	1101 001w : 11 100 reg
memory by CL	1101 001w : mod 100 r/m
register by immediate count	1100 000w : 11 100 reg : imm8 data
memory by immediate count	1100 000w : mod 100 r/m : imm8 data

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
SHLD – Double Precision Shift Left	
register by immediate count	0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8
memory by immediate count	0000 1111 : 1010 0100 : mod reg r/m : imm8
register by CL	0000 1111 : 1010 0101 : 11 reg2 reg1
memory by CL	0000 1111 : 1010 0101 : mod reg r/m
SHR – Shift Right	
register by 1	1101 000w : 11 101 reg
memory by 1	1101 000w : mod 101 r/m
register by CL	1101 001w : 11 101 reg
memory by CL	1101 001w : mod 101 r/m
register by immediate count	1100 000w : 11 101 reg : imm8 data
memory by immediate count	1100 000w : mod 101 r/m : imm8 data
SHRD – Double Precision Shift Right	
register by immediate count	0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8
memory by immediate count	0000 1111 : 1010 1100 : mod reg r/m : imm8
register by CL	0000 1111 : 1010 1101 : 11 reg2 reg1
memory by CL	0000 1111 : 1010 1101 : mod reg r/m
SIDT – Store Interrupt Descriptor Table Register	0000 1111 : 0000 0001 : mod^A 001 r/m
SLDT – Store Local Descriptor Table Register	
to register	0000 1111 : 0000 0000 : 11 000 reg
to memory	0000 1111 : 0000 0000 : mod 000 r/m
SMSW – Store Machine Status Word	
to register	0000 1111 : 0000 0001 : 11 100 reg
to memory	0000 1111 : 0000 0001 : mod 100 r/m
STC – Set Carry Flag	1111 1001
STD – Set Direction Flag	1111 1101
STI – Set Interrupt Flag	1111 1011
STOS/STOSB/STOSW/STOSD – Store String Data	1010 101w
STR – Store Task Register	
to register	0000 1111 : 0000 0000 : 11 001 reg
to memory	0000 1111 : 0000 0000 : mod 001 r/m

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
SUB – Integer Subtraction	
register1 to register2	0010 100w : 11 reg1 reg2
register2 to register1	0010 101w : 11 reg1 reg2
memory to register	0010 101w : mod reg r/m
register to memory	0010 100w : mod reg r/m
immediate to register	1000 00sw : 11 101 reg : immediate data
immediate to AL, AX, or EAX	0010 110w : immediate data
immediate to memory	1000 00sw : mod 101 r/m : immediate data
TEST – Logical Compare	
register1 and register2	1000 010w : 11 reg1 reg2
memory and register	1000 010w : mod reg r/m
immediate and register	1111 011w : 11 000 reg : immediate data
immediate and AL, AX, or EAX	1010 100w : immediate data
immediate and memory	1111 011w : mod 000 r/m : immediate data
UD2 – Undefined instruction	0000 FFFF : 0000 1011
VERR – Verify a Segment for Reading	
register	0000 1111 : 0000 0000 : 11 100 reg
memory	0000 1111 : 0000 0000 : mod 100 r/m
VERW – Verify a Segment for Writing	
register	0000 1111 : 0000 0000 : 11 101 reg
memory	0000 1111 : 0000 0000 : mod 101 r/m
WAIT – Wait	1001 1011
WBINVD – Writeback and Invalidate Data Cache	0000 1111 : 0000 1001
WRMSR – Write to Model-Specific Register	0000 1111 : 0011 0000
XADD – Exchange and Add	
register1, register2	0000 1111 : 1100 000w : 11 reg2 reg1
memory, reg	0000 1111 : 1100 000w : mod reg r/m
XCHG – Exchange Register/Memory with Register	
register1 with register2	1000 011w : 11 reg1 reg2
AX or EAX with reg	1001 0 reg
memory with reg	1000 011w : mod reg r/m
XLAT/XLATB – Table Look-up Translation	1101 0111

Table B-11. General Purpose Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
XOR – Logical Exclusive OR	
register1 to register2	0011 000w : 11 reg1 reg2
register2 to register1	0011 001w : 11 reg1 reg2
memory to register	0011 001w : mod reg r/m
register to memory	0011 000w : mod reg r/m
immediate to register	1000 00sw : 11 110 reg : immediate data
immediate to AL, AX, or EAX	0011 010w : immediate data
immediate to memory	1000 00sw : mod 110 r/m : immediate data
Prefix Bytes	
address size	0110 0111
LOCK	1111 0000
operand size	0110 0110
CS segment override	0010 1110
DS segment override	0011 1110
ES segment override	0010 0110
FS segment override	0110 0100
GS segment override	0110 0101
SS segment override	0011 0110

B.3. PENTIUM FAMILY INSTRUCTION FORMATS AND ENCODINGS

The following table shows formats and encodings introduced by the Pentium Family.

Table B-12. Pentium Family Instruction Formats and Encodings

Instruction and Format	Encoding
CMPXCHG8B – Compare and Exchange 8 Bytes	
memory, register	0000 1111 : 1100 0111 : mod 001 r/m

B.4. MMX INSTRUCTION FORMATS AND ENCODINGS

All MMX instructions, except the EMMS instruction, use the a format similar to the 2-byte Intel Architecture integer format. Details of subfield encodings within these formats are presented below.

B.4.1. Granularity Field (gg)

The granularity field (gg) indicates the size of the packed operands that the instruction is operating on. When this field is used, it is located in bits 1 and 0 of the second opcode byte. Table B-13 shows the encoding of this gg field.

Table B-13. Encoding of Granularity of Data Field (gg)

gg	Granularity of Data
00	Packed Bytes
01	Packed Words
10	Packed Doublewords
11	Quadword

B.4.2. MMX Technology and General-Purpose Register Fields (mmxreg and reg)

When MMX technology registers (mmxreg) are used as operands, they are encoded in the ModR/M byte in the reg field (bits 5, 4, and 3) and/or the R/M field (bits 2, 1, and 0).

If an MMX instruction operates on a general-purpose register (reg), the register is encoded in the R/M field of the ModR/M byte.

B.4.3. MMX Instruction Formats and Encodings Table

Table B-14 shows the formats and encodings of the integer instructions.

Table B-14. MMX Instruction Formats and Encodings

Instruction and Format	Encoding
EMMS - Empty MMX technology state	0000 1111:01110111
MOVD - Move doubleword	
reg to mmreg	0000 1111:01101110: 11 mmxreg reg
reg from mmxreg	0000 1111:01111110: 11 mmxreg reg
mem to mmxreg	0000 1111:01101110: mod mmxreg r/m
mem from mmxreg	0000 1111:01111110: mod mmxreg r/m

Table B-14. MMX Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
MOVQ - Move quadword	
mmxreg2 to mmxreg1	0000 1111:01101111: 11 mmxreg1 mmxreg2
mmxreg2 from mmxreg1	0000 1111:01111111: 11 mmxreg1 mmxreg2
mem to mmxreg	0000 1111:01101111: mod mmxreg r/m
mem from mmxreg	0000 1111:01111111: mod mmxreg r/m
PACKSSDW[1] - Pack dword to word data (signed with saturation)	
mmxreg2 to mmxreg1	0000 1111:01101011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:01101011: mod mmxreg r/m
PACKSSWB[1] - Pack word to byte data (signed with saturation)	
mmxreg2 to mmxreg1	0000 1111:01100011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:01100011: mod mmxreg r/m
PACKUSWB[1] - Pack word to byte data (unsigned with saturation)	
mmxreg2 to mmxreg1	0000 1111:01100111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:01100111: mod mmxreg r/m
PADD - Add with wrap-around	
mmxreg2 to mmxreg1	0000 1111: 111111gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 111111gg: mod mmxreg r/m
PADDS - Add signed with saturation	
mmxreg2 to mmxreg1	0000 1111: 111011gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 111011gg: mod mmxreg r/m
PADDUS - Add unsigned with saturation	
mmxreg2 to mmxreg1	0000 1111: 110111gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 110111gg: mod mmxreg r/m
PAND - Bitwise And	
mmxreg2 to mmxreg1	0000 1111:11011011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11011011: mod mmxreg r/m
PANDN - Bitwise AndNot	
mmxreg2 to mmxreg1	0000 1111:11011111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11011111: mod mmxreg r/m

Table B-14. MMX Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
PCMPEQ - Packed compare for equality	
mmxreg1 with mmxreg2	0000 1111:011101gg: 11 mmxreg1 mmxreg2
mmxreg with memory	0000 1111:011101gg: mod mmxreg r/m
PCMPGT - Packed compare greater (signed)	
mmxreg1 with mmxreg2	0000 1111:011001gg: 11 mmxreg1 mmxreg2
mmxreg with memory	0000 1111:011001gg: mod mmxreg r/m
PMADDWD - Packed multiply add	
mmxreg2 to mmxreg1	0000 1111:11110101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11110101: mod mmxreg r/m
PMULHUW - Packed multiplication, store high word (unsigned)	
mmxreg2 to mmxreg1	0000 1111: 1110 0100: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1110 0100: mod mmxreg r/m
PMULHW - Packed multiplication, store high word	
mmxreg2 to mmxreg1	0000 1111:11100101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11100101: mod mmxreg r/m
PMULLW - Packed multiplication, store low word	
mmxreg2 to mmxreg1	0000 1111:11010101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11010101: mod mmxreg r/m
POR - Bitwise Or	
mmxreg2 to mmxreg1	0000 1111:11101011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11101011: mod mmxreg r/m
PSLL^2 - Packed shift left logical	
mmxreg1 by mmxreg2	0000 1111:111100gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:111100gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:011100gg: 11 110 mmxreg: imm8 data
PSRA^2 - Packed shift right arithmetic	
mmxreg1 by mmxreg2	0000 1111:111000gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:111000gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:011100gg: 11 100 mmxreg: imm8 data

Table B-14. MMX Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
PSRL[2] - Packed shift right logical	
mmxreg1 by mmxreg2	0000 1111:110100gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:110100gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:011100gg: 11 010 mmxreg: imm8 data
PSUB - Subtract with wrap-around	
mmxreg2 from mmxreg1	0000 1111:111110gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:111110gg: mod mmxreg r/m
PSUBS - Subtract signed with saturation	
mmxreg2 from mmxreg1	0000 1111:111010gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:111010gg: mod mmxreg r/m
PSUBUS - Subtract unsigned with saturation	
mmxreg2 from mmxreg1	0000 1111:110110gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:110110gg: mod mmxreg r/m
PUNPCKH - Unpack high data to next larger type	
mmxreg2 to mmxreg1	0000 1111:011010gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:011010gg: mod mmxreg r/m
PUNPCKL - Unpack low data to next larger type	
mmxreg2 to mmxreg1	0000 1111:011000gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:011000gg: mod mmxreg r/m
PXOR - Bitwise Xor	
mmxreg2 to mmxreg1	0000 1111:11101111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11101111: mod mmxreg r/m

NOTES:

1 The pack instructions perform saturation from signed packed data of one type to signed or unsigned data of the next smaller type.

2 The format of the shift instructions has one additional format to support shifting by immediate shift-counts. The shift operations are not supported equally for all data types.

B.5. P6 FAMILY INSTRUCTION FORMATS AND ENCODINGS

Table B-15 shows the formats and encodings for several instructions that were introduced into the IA-32 architecture in the P6 family processors.

Table B-15. Formats and Encodings of P6 Family Instructions

Instruction and Format	Encoding
CMOVcc – Conditional Move	
register2 to register1	0000 1111: 0100 tttn : 11 reg1 reg2
memory to register	0000 1111 : 0100 tttn : mod reg r/m
FCMOVcc – Conditional Move on EFLAG Register Condition Codes	
move if below (B)	11011 010 : 11 000 ST(i)
move if equal (E)	11011 010 : 11 001 ST(i)
move if below or equal (BE)	11011 010 : 11 010 ST(i)
move if unordered (U)	11011 010 : 11 011 ST(i)
move if not below (NB)	11011 011 : 11 000 ST(i)
move if not equal (NE)	11011 011 : 11 001 ST(i)
move if not below or equal (NBE)	11011 011 : 11 010 ST(i)
move if not unordered (NU)	11011 011 : 11 011 ST(i)
FCOMI – Compare Real and Set EFLAGS	11011 011 : 11 110 ST(i)
FXRSTOR—Restore x87 FPU, MMX, SSE, and SSE2 State	00001111:10101110: mod^A 001 r/m
FXSAVE—Save x87 FPU, MMX, SSE, and SSE2 State	00001111:10101110: mod^A 000 r/m
SYSENTER—Fast System Call	00001111:00110100
SYSEXIT—Fast Return from Fast System Call	00001111:00110101

NOTE:

1 In FXSAVE and FXRSTOR, “mod=11” is reserved.

B.6. SSE INSTRUCTION FORMATS AND ENCODINGS

The SSE instructions use the ModR/M format and are preceded by the 0FH prefix byte. In general, operations are not duplicated to provide two directions (that is, separate load and store variants).

The following three tables (Tables B-16, B-17, and B-18) show the formats and encodings for the SSE SIMD floating-point, SIMD integer, and cacheability and memory ordering instructions, respectively. Some SSE instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. These mandatory prefixes are included in the tables.

Table B-16. Formats and Encodings of SSE Floating-Point Instructions

Instruction and Format	Encoding
ADDPS—Add Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011000: mod xmmreg r/m
ADDSS—Add Scalar Single-Precision Floating-Point Values	
xmmreg to xmmreg	11110011:00001111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011000: mod xmmreg r/m
ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01010101:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01010101: mod xmmreg r/m
ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01010100:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01010100: mod xmmreg r/m
CMPPS—Compare Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg, imm8	00001111:11000010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	00001111:11000010: mod xmmreg r/m: imm8
CMPSS—Compare Scalar Single-Precision Floating-Point Values	
xmmreg to xmmreg, imm8	11110011:00001111:11000010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	11110011:00001111:11000010: mod xmmreg r/m: imm8

Table B-16. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS	
xmmreg to xmmreg	00001111:00101111:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:00101111: mod xmmreg r/m
CVTPI2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values	
mmreg to xmmreg	00001111:00101010:11 xmmreg1 mmreg1
mem to xmmreg	00001111:00101010: mod xmmreg r/m
CVTPS2PI—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	00001111:00101101:11 mmreg1 xmmreg1
mem to mmreg	00001111:00101101: mod mmreg r/m
CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value	
r32 to xmmreg1	11110011:00001111:00101010:11 xmmreg r32
mem to xmmreg	11110011:00001111:00101010: mod xmmreg r/m
CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	11110011:00001111:00101101:11 r32 xmmreg
mem to r32	11110011:00001111:00101101: mod r32 r/m
CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	00001111:00101100:11 mmreg1 xmmreg1
mem to mmreg	00001111:00101100: mod mmreg r/m
CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	11110011:00001111:00101100:11 r32 xmmreg1
mem to r32	11110011:00001111:00101100: mod r32 r/m
DIVPS—Divide Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01011110:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011110: mod xmmreg r/m

Table B-16. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
DIVSS—Divide Scalar Single-Precision Floating-Point Values	
xmmreg to xmmreg	11110011:00001111:01011110:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011110: mod xmmreg r/m
LDMXCSR—Load MXCSR Register State	
m32 to MXCSR	00001111:10101110:mod^{A} 010 mem
MAXPS—Return Maximum Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01011111:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011111: mod xmmreg r/m
MAXSS—Return Maximum Scalar Double-Precision Floating-Point Value	
xmmreg to xmmreg	11110011:00001111:01011111:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011111: mod xmmreg r/m
MINPS—Return Minimum Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01011101:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011101: mod xmmreg r/m
MINSS—Return Minimum Scalar Double-Precision Floating-Point Value	
xmmreg to xmmreg	11110011:00001111:01011101:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011101: mod xmmreg r/m
MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	00001111:00101000:11 xmmreg2 xmmreg1
mem to xmmreg1	00001111:00101000: mod xmmreg r/m
xmmreg1 to xmmreg2	00001111:00101001:11 xmmreg1 xmmreg2
xmmreg1 to mem	00001111:00101001: mod xmmreg r/m
MOVHLPS—Move Packed Single-Precision Floating-Point Values High to Low	
xmmreg to xmmreg	00001111:00010010:11 xmmreg1 xmmreg2
MOVHPS—Move High Packed Single-Precision Floating-Point Values	
mem to xmmreg	00001111:00010110: mod xmmreg r/m
xmmreg to mem	00001111:00010111: mod xmmreg r/m

Table B-16. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High	
xmmreg to xmmreg	00001111:00010110:11 xmmreg1 xmmreg2
MOVLPS—Move Low Packed Single-Precision Floating-Point Values	
mem to xmmreg	00001111:00010010: mod xmmreg r/m
xmmreg to mem	00001111:00010011: mod xmmreg r/m
MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask	
xmmreg to r32	00001111:01010000:11 r32 xmmreg
MOVSS—Move Scalar Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	11110011:00001111:00010000:11 xmmreg2 xmmreg1
mem to xmmreg1	11110011:00001111:00010000: mod xmmreg r/m
xmmreg1 to xmmreg2	11110011:00001111:00010001:11 xmmreg1 xmmreg2
xmmreg1 to mem	11110011:00001111:00010001: mod xmmreg r/m
MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	00001111:00010000:11 xmmreg2 xmmreg1
mem to xmmreg1	00001111:00010000: mod xmmreg r/m
xmmreg1 to xmmreg2	00001111:00010001:11 xmmreg1 xmmreg2
xmmreg1 to mem	00001111:00010001: mod xmmreg r/m
MULPS—Multiply Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011001: mod xmmreg rm
MULSS—Multiply Scalar Single-Precision Floating-Point Values	
xmmreg to xmmreg	11110011:00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011001: mod xmmreg r/m
ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01010110:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01010110 mod xmmreg r/m

Table B-16. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01010011:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01010011: mod xmmreg r/m
RCPSS—Compute Reciprocals of Scalar Single-Precision Floating-Point Value	
xmmreg to xmmreg	11110011:00001111:01010011:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01010011: mod xmmreg r/m
RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01010010:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01010010 mode xmmreg r/m
RSQRTSS—Compute Reciprocals of Square Roots of Scalar Single-Precision Floating-Point Value	
xmmreg to xmmreg	11110011:00001111:01010010:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01010010 mod xmmreg r/m
SHUFPS—Shuffle Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg, imm8	00001111:11000110:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	00001111:11000110: mod xmmreg r/m: imm8
SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01010001:11 xmmreg1 xmmreg 2
mem to xmmreg	00001111:01010001 mod xmmreg r/m
SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value	
xmmreg to xmmreg	11110011:00001111:01010001:11 xmmreg1 xmmreg 2
mem to xmmreg	11110011:00001111:01010001:mod xmmreg r/m
STMXCSR—Store MXCSR Register State	
MXCSR to mem	00001111:10101110:mod^{A} 011 mem
SUBPS—Subtract Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01011100:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011100:mod xmmreg r/m

Table B-16. Formats and Encodings of SSE Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
SUBSS—Subtract Scalar Single-Precision Floating-Point Values	
xmmreg to xmmreg	11110011:00001111:01011100:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011100:mod xmmreg r/m
UCOMISS—Unordered Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS	
xmmreg to xmmreg	00001111:00101110:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:00101110 mod xmmreg r/m
UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:00010101:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:00010101 mod xmmreg r/m
UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:00010100:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:00010100 mod xmmreg r/m
XORPS—Bitwise Logical XOR of Single-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01010111:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01010111 mod xmmreg r/m

Table B-17. Formats and Encodings of SSE Integer Instructions

Instruction and Format	Encoding
PAVGB/PAVGW—Average Packed Integers	
mmreg to mmreg	00001111:11100000:11 mmreg1 mmreg2
	00001111:11100011:11 mmreg1 mmreg2
mem to mmreg	00001111:11100000 mod mmreg r/m
	00001111:11100011 mod mmreg r/m
PEXTRW—Extract Word	
mmreg to reg32, imm8	00001111:11000101:11 r32 mmreg: imm8
PINSRW - Insert Word	
reg32 to mmreg, imm8	00001111:11000100:11 mmreg r32: imm8
m16 to mmreg, imm8	00001111:11000100 mod mmreg r/m: imm8
PMAXSW—Maximum of Packed Signed Word Integers	
mmreg to mmreg	00001111:11101110:11 mmreg1 mmreg2
mem to mmreg	00001111:11101110 mod mmreg r/m
PMAXUB—Maximum of Packed Unsigned Byte Integers	
mmreg to mmreg	00001111:11011110:11 mmreg1 mmreg2
mem to mmreg	00001111:11011110 mod mmreg r/m
PMINSW—Minimum of Packed Signed Word Integers	
mmreg to mmreg	00001111:11101010:11 mmreg1 mmreg2
mem to mmreg	00001111:11101010 mod mmreg r/m
PMINUB—Minimum of Packed Unsigned Byte Integers	
mmreg to mmreg	00001111:11011010:11 mmreg1 mmreg2
mem to mmreg	00001111:11011010 mod mmreg r/m
PMOVMSKB - Move Byte Mask To Integer	
mmreg to reg32	00001111:11010111:11 r32 mmreg
PMULHUW—Multiply Packed Unsigned Integers and Store High Result	
mmreg to mmreg	00001111:11100100:11 mmreg1 mmreg2
mem to mmreg	00001111:11100100 mod mmreg r/m
PSADBW—Compute Sum of Absolute Differences	
mmreg to mmreg	00001111:11110110:11 mmreg1 mmreg2
mem to mmreg	00001111:11110110 mod mmreg r/m

Table B-17. Formats and Encodings of SSE Integer Instructions (Contd.)

Instruction and Format	Encoding
PSHUFW—Shuffle Packed Words	
mmreg to mmreg, imm8	00001111:01110000:11 mmreg1 mmreg2: imm8
mem to mmreg, imm8	00001111:01110000:11 mod mmreg r/m: imm8

Table B-18. Format and Encoding of SSE Cacheability and Memory Ordering Instructions

Instruction and Format	Encoding
MASKMOVQ—Store Selected Bytes of Quadword	
mmreg to mmreg	00001111:11110111:11 mmreg1 mmreg2
MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint	
xmmreg to mem	00001111:00101011: mod xmmreg r/m
MOVNTQ—Store Quadword Using Non-Temporal Hint	
mmreg to mem	00001111:11100111: mod mmreg r/m
PREFETCHT0—Prefetch Temporal to All Cache Levels	00001111:00011000:mod^A 001 mem
PREFETCHT1—Prefetch Temporal to First Level Cache	00001111:00011000:mod^A 010 mem
PREFETCHT2—Prefetch Temporal to Second Level Cache	00001111:00011000:mod^A 011 mem
PREFETCHNTA—Prefetch Non-Temporal to All Cache Levels	00001111:00011000:mod^A 000 mem
SFENCE—Store Fence	00001111:10101110:11 111 000

B.7. SSE2 INSTRUCTION FORMATS AND ENCODINGS

The SSE2 instructions use the ModR/M format and are preceded by the 0FH prefix byte. In general, operations are not duplicated to provide two directions (that is, separate load and store variants).

The following three tables show the formats and encodings for the SSE2 SIMD floating-point, SIMD integer, and cacheability instructions, respectively. Some SSE2 instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. These prefixes are included in the tables.

B.7.1. Granularity Field (gg)

The granularity field (gg) indicates the size of the packed operands that the instruction is operating on. When this field is used, it is located in bits 1 and 0 of the second opcode byte. Table B-19 shows the encoding of this gg field.

Table B-19. Encoding of Granularity of Data Field (gg)

gg	Granularity of Data
00	Packed Bytes
01	Packed Words
10	Packed Doublewords
11	Quadword

Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions

Instruction and Format	Encoding
ADDPD - Add Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011000: mod xmmreg r/m
ADDSD - Add Scalar Double-Precision Floating-Point Values	
xmmreg to xmmreg	11110010:00001111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011000: mod xmmreg r/m
ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01010101:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01010101: mod xmmreg r/m

Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01010100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01010100: mod xmmreg r/m
CMPPD—Compare Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg, imm8	01100110:00001111:11000010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	01100110:00001111:11000010: mod xmmreg r/m: imm8
CMPSD—Compare Scalar Double-Precision Floating-Point Values	
xmmreg to xmmreg, imm8	11110010:00001111:11000010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	11110010:00001111:11000010: mod xmmreg r/m: imm8
COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS	
xmmreg to xmmreg	01100110:00001111:00101111:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:00101111: mod xmmreg r/m
CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values	
mmreg to xmmreg	01100110:00001111:00101010:11 xmmreg1 mmreg1
mem to xmmreg	01100110:00001111:00101010: mod xmmreg r/m
CVTPD2PI—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	01100110:00001111:00101101:11 mmreg1 xmmreg1
mem to mmreg	01100110:00001111:00101101: mod mmreg r/m
CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value	
r32 to xmmreg1	11110010:00001111:00101010:11 xmmreg r32
mem to xmmreg	11110010:00001111:00101010: mod xmmreg r/m

Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	11110010:00001111:00101101:11 r32 xmmreg
mem to r32	11110010:00001111:00101101: mod r32 r/m
CVTTPD2PI—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	01100110:00001111:00101100:11 mmreg xmmreg
mem to mmreg	01100110:00001111:00101100: mod mmreg r/m
CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	11110010:00001111:00101100:11 r32 xmmreg
mem to r32	11110010:00001111:00101100: mod r32 r/m
CVTPD2PS—Covert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01011010:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011010: mod xmmreg r/m
CVTPS2PD—Covert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01011010:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011010: mod xmmreg r/m
CVTSD2SS—Covert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value	
xmmreg to xmmreg	11110010:00001111:01011010:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011010: mod xmmreg r/m
CVTSS2SD—Covert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value	
xmmreg to xmmreg	11110011:00001111:01011010:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011010: mod xmmreg r/m

Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to xmmreg	11110010:00001111:11100110:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:11100110: mod xmmreg r/m
CVTTPD2DQ—Convert With Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to xmmreg	01100110:00001111:11100110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11100110: mod xmmreg r/m
CVTDQ2PD—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values	
xmmreg to xmmreg	11110011:00001111:11100110:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:11100110: mod xmmreg r/m
CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to xmmreg	01100110:00001111:01011011:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011011: mod xmmreg r/m
CVTTPS2DQ—Convert With Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to xmmreg	11110011:00001111:01011011:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01011011: mod xmmreg r/m
CVTDQ2PS—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	00001111:01011011:11 xmmreg1 xmmreg2
mem to xmmreg	00001111:01011011: mod xmmreg r/m
DIVPD—Divide Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01011110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011110: mod xmmreg r/m
DIVSD—Divide Scalar Double-Precision Floating-Point Values	
xmmreg to xmmreg	11110010:00001111:01011110:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011110: mod xmmreg r/m

Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
MAXPD—Return Maximum Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01011111:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011111: mod xmmreg r/m
MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value	
xmmreg to xmmreg	11110010:00001111:01011111:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011111: mod xmmreg r/m
MINPD—Return Minimum Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01011101:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011101: mod xmmreg r/m
MINSD—Return Minimum Scalar Double-Precision Floating-Point Value	
xmmreg to xmmreg	11110010:00001111:01011101:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011101: mod xmmreg r/m
MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	01100110:00001111:00101001:11 xmmreg2 xmmreg1
mem to xmmreg1	01100110:00001111:00101001: mod xmmreg r/m
xmmreg1 to xmmreg2	01100110:00001111:00101000:11 xmmreg1 xmmreg2
xmmreg1 to mem	01100110:00001111:00101000: mod xmmreg r/m
MOVHPD—Move High Packed Double-Precision Floating-Point Values	
mem to xmmreg	01100110:00001111:00010111: mod xmmreg r/m
xmmreg to mem	01100110:00001111:00010110: mod xmmreg r/m
MOVLPD—Move Low Packed Double-Precision Floating-Point Values	
mem to xmmreg	01100110:00001111:00010011: mod xmmreg r/m
xmmreg to mem	01100110:00001111:00010010: mod xmmreg r/m
MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask	
xmmreg to r32	01100110:00001111:01010000:11 r32 xmmreg

Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
MOVSD—Move Scalar Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	11110010:00001111:00010001:11 xmmreg2 xmmreg1
mem to xmmreg1	11110010:00001111:00010001: mod xmmreg r/m
xmmreg1 to xmmreg2	11110010:00001111:00010000:11 xmmreg1 xmmreg2
xmmreg1 to mem	11110010:00001111:00010000: mod xmmreg r/m
MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	01100110:00001111:00010001:11 xmmreg2 xmmreg1
mem to xmmreg1	01100110:00001111:00010001: mod xmmreg r/m
xmmreg1 to xmmreg2	01100110:00001111:00010000:11 xmmreg1 xmmreg2
xmmreg1 to mem	01100110:00001111:00010000: mod xmmreg r/m
MULPD—Multiply Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011001: mod xmmreg rm
MULSD—Multiply Scalar Double-Precision Floating-Point Values	
xmmreg to xmmreg	11110010:00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011001: mod xmmreg r/m
ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01010110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01010110: mod xmmreg r/m
SHUFPD—Shuffle Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg, imm8	01100110:00001111:11000110:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	01100110:00001111:11000110: mod xmmreg r/m: imm8
SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01010001:11 xmmreg1 xmmreg 2
mem to xmmreg	01100110:00001111:01010001: mod xmmreg r/m

Table B-20. Formats and Encodings of SSE2 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value	
xmmreg to xmmreg	11110010:00001111:01010001:11 xmmreg1 xmmreg 2
mem to xmmreg	11110010:00001111:01010001: mod xmmreg r/m
SUBPD—Subtract Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01011100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01011100: mod xmmreg r/m
SUBSD—Subtract Scalar Double-Precision Floating-Point Values	
xmmreg to xmmreg	11110010:00001111:01011100:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01011100: mod xmmreg r/m
UCOMISD—Unordered Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS	
xmmreg to xmmreg	01100110:00001111:00101110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:00101110: mod xmmreg r/m
UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:00010101:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:00010101: mod xmmreg r/m
UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:00010100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:00010100: mod xmmreg r/m
XORPD—Bitwise Logical OR of Double-Precision Floating-Point Values	
xmmreg to xmmreg	01100110:00001111:01010111:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01010111: mod xmmreg r/m

Table B-21. Formats and Encodings of SSE2 Integer Instructions

Instruction and Format	Encoding
MOVD - Move Doubleword	
reg to xmmeg	01100110:0000 1111:01101110: 11 xmmreg reg
reg from xmmreg	01100110:0000 1111:01111110: 11 xmmreg reg
mem to xmmreg	01100110:0000 1111:01101110: mod xmmreg r/m
mem from xmmreg	01100110:0000 1111:01111110: mod xmmreg r/m
MOVDQA—Move Aligned Double Quadword	
xmmreg to xmmreg	01100110:00001111:01101111:11 xmmreg1 xmmreg2
	01100110:00001111:01111111:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01101111: mod xmmreg r/m
mem from xmmreg	01100110:00001111:01111111: mod xmmreg r/m
MOVDQU—Move Unaligned Double Quadword	
xmmreg to xmmreg	11110011:00001111:01101111:11 xmmreg1 xmmreg2
	11110011:00001111:01111111:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01101111: mod xmmreg r/m
mem from xmmreg	11110011:00001111:01111111: mod xmmreg r/m
MOVQ2DQ—Move Quadword from MMX to XMM Register	
mmreg to xmmreg	11110011:00001111:11010110:11 mmreg1 mmreg2
MOVDQ2Q—Move Quadword from XMM to MMX Register	
xmmreg to mmreg	11110010:00001111:11010110:11 mmreg1 mmreg2
MOVQ - Move Quadword	
xmmreg2 to xmmreg1	11110011:00001111:01111110: 11 xmmreg1 xmmreg2
xmmreg2 from xmmreg1	01100110:00001111:11010110: 11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:01111110: mod xmmreg r/m
mem from xmmreg	01100110:00001111:11010110: mod xmmreg r/m
PACKSSDW[1] - Pack Dword To Word Data (signed with saturation)	
xmmreg2 to xmmreg1	01100110:0000 1111:01101011: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:01101011: mod xmmreg r/m

Table B-21. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PACKSSWB - Pack Word To Byte Data (signed with saturation)	
xmmreg2 to xmmreg1	01100110:0000 1111:01100011: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:01100011: mod xmmreg r/m
PACKUSWB - Pack Word To Byte Data (unsigned with saturation)	
xmmreg2 to xmmreg1	01100110:0000 1111:01100111: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:01100111: mod xmmreg r/m
PADDQ—Add Packed Quadword Integers	
mmreg to mmreg	00001111:11010100:11 mmreg1 mmreg2
mem to mmreg	00001111:11010100: mod mmreg r/m
xmmreg to xmmreg	01100110:00001111:11010100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11010100: mod xmmreg r/m
PADD - Add With Wrap-around	
xmmreg2 to xmmreg1	01100110:0000 1111: 111111gg: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111: 111111gg: mod xmmreg r/m
PADDS - Add Signed With Saturation	
xmmreg2 to xmmreg1	01100110:0000 1111: 111011gg: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111: 111011gg: mod xmmreg r/m
PADDUS - Add Unsigned With Saturation	
xmmreg2 to xmmreg1	01100110:0000 1111: 110111gg: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111: 110111gg: mod xmmreg r/m
PAND - Bitwise And	
xmmreg2 to xmmreg1	01100110:0000 1111:11011011: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:11011011: mod xmmreg r/m
PANDN - Bitwise AndNot	
xmmreg2 to xmmreg1	01100110:0000 1111:11011111: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:11011111: mod xmmreg r/m
PAVGB—Average Packed Integers	
xmmreg to xmmreg	01100110:00001111:11100000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11100000 mod xmmreg r/m

Table B-21. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PAVGW—Average Packed Integers	
xmmreg to xmmreg	01100110:00001111:11100011:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11100011 mod xmmreg r/m
PCMPEQ - Packed Compare For Equality	
xmmreg1 with xmmreg2	01100110:0000 1111:011101gg: 11 xmmreg1 xmmreg2
xmmreg with memory	01100110:0000 1111:011101gg: mod xmmreg r/m
PCMPGT - Packed Compare Greater (signed)	
xmmreg1 with xmmreg2	01100110:0000 1111:011001gg: 11 xmmreg1 xmmreg2
xmmreg with memory	01100110:0000 1111:011001gg: mod xmmreg r/m
PEXTRW—Extract Word	
xmmreg to reg32, imm8	01100110:00001111:11000101:11 r32 xmmreg: imm8
PINSRW - Insert Word	
reg32 to xmmreg, imm8	01100110:00001111:11000100:11 xmmreg r32: imm8
m16 to xmmreg, imm8	01100110:00001111:11000100 mod xmmreg r/m: imm8
PMADDWD - Packed Multiply Add	
xmmreg2 to xmmreg1	01100110:0000 1111:11110101: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:11110101: mod xmmreg r/m
PMAXSW—Maximum of Packed Signed Word Integers	
xmmreg to xmmreg	01100110:00001111:11101110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11101110 mod xmmreg r/m
PMAXUB—Maximum of Packed Unsigned Byte Integers	
xmmreg to xmmreg	01100110:00001111:11011110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11011110 mod xmmreg r/m
PMINSW—Minimum of Packed Signed Word Integers	
xmmreg to xmmreg	01100110:00001111:11101010:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11101010 mod xmmreg r/m
PMINUB—Minimum of Packed Unsigned Byte Integers	
xmmreg to xmmreg	01100110:00001111:11011010:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11011010 mod xmmreg r/m

Table B-21. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PMOVMSKB - Move Byte Mask To Integer	
xmmreg to reg32	01100110:00001111:11010111:11 r32 xmmreg
PMULHUW - Packed multiplication, store high word (unsigned)	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 0100: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1110 0100: mod xmmreg r/m
PMULHW - Packed Multiplication, store high word	
xmmreg2 to xmmreg1	01100110:0000 1111:11100101: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:11100101: mod xmmreg r/m
PMULLW - Packed Multiplication, store low word	
xmmreg2 to xmmreg1	01100110:0000 1111:11010101: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:11010101: mod xmmreg r/m
PMULUDQ—Multiply Packed Unsigned Doubleword Integers	
mmreg to mmreg	00001111:11110100:11 mmreg1 mmreg2
mem to mmreg	00001111:11110100: mod mmreg r/m
xmmreg to xmmreg	01100110:00001111:11110100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11110100: mod xmmreg r/m
POR - Bitwise Or	
xmmreg2 to xmmreg1	01100110:0000 1111:11101011: 11 xmmreg1 xmmreg2
xmemory to xmmreg	01100110:0000 1111:11101011: mod xmmreg r/m
PSADBW—Compute Sum of Absolute Differences	
xmmreg to xmmreg	01100110:00001111:11110110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11110110: mod xmmreg r/m
PSHUFLW—Shuffle Packed Low Words	
xmmreg to xmmreg, imm8	11110010:00001111:01110000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	11110010:00001111:01110000:11 mod xmmreg r/m: imm8
PSHUFHW—Shuffle Packed High Words	
xmmreg to xmmreg, imm8	11110011:00001111:01110000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	11110011:00001111:01110000:11 mod xmmreg r/m: imm8

Table B-21. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PSHUFD—Shuffle Packed Doublewords	
xmmreg to xmmreg, imm8	01100110:00001111:01110000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	01100110:00001111:01110000:11 mod xmmreg r/m: imm8
PSLLDQ—Shift Double Quadword Left Logical	
xmmreg, imm8	01100110:00001111:01110011:11 111 xmmreg: imm8
PSLL - Packed Shift Left Logical	
xmmreg1 by xmmreg2	01100110:0000 1111:111100gg: 11 xmmreg1 xmmreg2
xmmreg by memory	01100110:0000 1111:111100gg: mod xmmreg r/m
xmmreg by immediate	01100110:0000 1111:011100gg: 11 110 xmmreg: imm8 data
PSRA - Packed Shift Right Arithmetic	
xmmreg1 by xmmreg2	01100110:0000 1111:111000gg: 11 xmmreg1 xmmreg2
xmmreg by memory	01100110:0000 1111:111000gg: mod xmmreg r/m
xmmreg by immediate	01100110:0000 1111:011100gg: 11 100 xmmreg: imm8 data
PSRLDQ—Shift Double Quadword Right Logical	
xmmreg, imm8	01100110:00001111:01110011:11 011 xmmreg: imm8
PSRL - Packed Shift Right Logical	
xmmxreg1 by xmmxreg2	01100110:0000 1111:110100gg: 11 xmmreg1 xmmreg2
xmmxreg by memory	01100110:0000 1111:110100gg: mod xmmreg r/m
xmmxreg by immediate	01100110:0000 1111:011100gg: 11 010 xmmreg: imm8 data
PSUBQ—Subtract Packed Quadword Integers	
mmreg to mmreg	00001111:11111011:11 mmreg1 mmreg2
mem to mmreg	00001111:11111011: mod mmreg r/m
xmmreg to xmmreg	01100110:00001111:11111011:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11111011: mod xmmreg r/m
PSUB - Subtract With Wrap-around	
xmmreg2 from xmmreg1	01100110:0000 1111:111110gg: 11 xmmreg1 xmmreg2
memory from xmmreg	01100110:0000 1111:111110gg: mod xmmreg r/m
PSUBS - Subtract Signed With Saturation	
xmmreg2 from xmmreg1	01100110:0000 1111:111010gg: 11 xmmreg1 xmmreg2
memory from xmmreg	01100110:0000 1111:111010gg: mod xmmreg r/m

Table B-21. Formats and Encodings of SSE2 Integer Instructions (Contd.)

Instruction and Format	Encoding
PSUBUS - Subtract Unsigned With Saturation	
xmmreg2 from xmmreg1	0000 1111:110110gg: 11 xmmreg1 xmmreg2
memory from xmmreg	0000 1111:110110gg: mod xmmreg r/m
PUNPCKH—Unpack High Data To Next Larger Type	
xmmreg to xmmreg	01100110:00001111:011010gg:11 xmmreg1 Xmmreg2
mem to xmmreg	01100110:00001111:011010gg: mod xmmreg r/m
PUNPCKHQDQ—Unpack High Data	
xmmreg to xmmreg	01100110:00001111:01101101:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01101101: mod xmmreg r/m
PUNPCKL—Unpack Low Data To Next Larger Type	
xmmreg to xmmreg	01100110:00001111:011000gg:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:011000gg: mod xmmreg r/m
PUNPCKLQDQ—Unpack Low Data	
xmmreg to xmmreg	01100110:00001111:01101100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01101100: mod xmmreg r/m
PXOR - Bitwise Xor	
xmmreg2 to xmmreg1	01100110:0000 1111:11101111: 11 xmmreg1 xmmreg2
memory to xmmreg	01100110:0000 1111:11101111: mod xmmreg r/m

Table B-22. Format and Encoding of SSE2 Cacheability Instructions

Instruction and Format	Encoding
MASKMOVDQU—Store Selected Bytes of Double Quadword	
xmmreg to xmmreg	01100110:00001111:11110111:11 xmmreg1 xmmreg2
CLFLUSH—Flush Cache Line	
mem	00001111:10101110:mod r/m
MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint	
xmmreg to mem	01100110:00001111:00101011: mod xmmreg r/m
MOVNTDQ—Store Double Quadword Using Non-Temporal Hint	
xmmreg to mem	01100110:00001111:11100111: mod xmmreg r/m

Table B-22. Format and Encoding of SSE2 Cacheability Instructions (Contd.)

Instruction and Format	Encoding
MOVNTI—Store Doubleword Using Non-Temporal Hint	
reg to mem	00001111:11000011: mod reg r/m
PAUSE—Spin Loop Hint	11110011:10010000
LFENCE—Load Fence	00001111:10101110: 11 101 000
MFENCE—Memory Fence	00001111:10101110: 11 110 000

B.7.2. SSE3 Formats and Encodings Table

The tables in this section provide Prescott formats and encodings. Some SSE3 instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. These prefixes are included in the tables.

Table B-23. Formats and Encodings of SSE3 Floating-Point Instructions

Instruction and Format	Encoding
ADDSUBPD—Add /Sub packed DP FP numbers from XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	01100110:00001111:11010000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11010000: mod xmmreg r/m
ADDSUBPS — Add /Sub packed SP FP numbers from XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	11110010:00001111:11010000:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:11010000: mod xmmreg r/m
HADDPD — Add horizontally packed DP FP numbers XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	01100110:00001111:01111100:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01111100: mod xmmreg r/m
HADDPS — Add horizontally packed SP FP numbers XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	11110010:00001111:01111100:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01111100: mod xmmreg r/m
HSUBPD — Sub horizontally packed DP FP numbers XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	01100110:00001111:01111101:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:01111101: mod xmmreg r/m

Table B-23. Formats and Encodings of SSE3 Floating-Point Instructions (Contd.)

Instruction and Format	Encoding
HSUBPS — Sub horizontally packed SP FP numbers XMM2/Mem to XMM1	
xmmreg2 to xmmreg1	11110010:00001111:01111101:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:01111101: mod xmmreg r/m

Table B-24. Formats and Encodings for SSE3 Event Management Instructions

Instruction and Format	Encoding
MONITOR — Set up a linear address range to be monitored by hardware	
eax, ecx, edx	0000 1111 : 0000 0001:11 001 000
MWAIT — Wait until write-back store performed within the range specified by the instruction MONITOR	
eax, ecx	0000 1111 : 0000 0001:11 001 001

Table B-25. Formats and Encodings for SSE3 Integer and Move Instructions

Instruction and Format	Encoding
FISTTP — Store ST in int16 (chop) and pop	
m16int	11011 111 : mod^A 001 r/m
FISTTP — Store ST in int32 (chop) and pop	
m32int	11011 011 : mod^A 001 r/m
FISTTP — Store ST in int64 (chop) and pop	
m64int	11011 101 : mod^A 001 r/m
LDDQU — Load unaligned integer 128-bit	
xmm, m128	11110010:00001111:11110000: mod^A xmmreg r/m
MOVDDUP — Move 64 bits representing one DP data from XMM2/Mem to XMM1 and duplicate	
xmmreg2 to xmmreg1	11110010:00001111:00010010:11 xmmreg1 xmmreg2
mem to xmmreg	11110010:00001111:00010010: mod xmmreg r/m
MOVSHDUP — Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate high	
xmmreg2 to xmmreg1	11110011:00001111:00010110:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:00010110: mod xmmreg r/m

Table B-25. Formats and Encodings for SSE3 Integer and Move Instructions (Contd.)

Instruction and Format	Encoding
MOVSLDUP — Move 128 bits representing 4 SP data from XMM2/Mem to XMM1 and duplicate low	
xmmreg2 to xmmreg1	11110011:00001111:00010010:11 xmmreg1 xmmreg2
mem to xmmreg	11110011:00001111:00010010: mod xmmreg r/m

B.8. FLOATING-POINT INSTRUCTION FORMATS AND ENCODINGS

Table B-26 shows the five different formats used for floating-point instructions In all cases, instructions are at least two bytes long and begin with the bit pattern 11011.

Table B-26. General Floating-Point Instruction Formats

	Instruction										Optional Fields	
	First Byte				Second Byte							
1	11011	OPA		1	mod		1	OPB		r/m	s-i-b	disp
2	11011	MF		OPA	mod		OPB			r/m	s-i-b	disp
3	11011	d	P	OPA	1	1	OPB	R		ST(i)		
4	11011	0	0	1	1	1	1	OP				
5	11011	0	1	1	1	1	1	OP				
	15–11	10	9	8	7	6	5	4	3	2 1 0		

MF = Memory Format
- 00 — 32-bit real
- 01 — 32-bit integer
- 10 — 64-bit real
- 11 — 16-bit integer

P = Pop
- 0 — Do not pop stack
- 1 — Pop stack after operation

d = Destination
- 0 — Destination is ST(0)
- 1 — Destination is ST(i)

R XOR d = 0 — Destination OP Source
R XOR d = 1 — Source OP Destination

ST(i) = Register stack element *i*
000 = Stack Top
001 = Second stack element
.
.
.
111 = Eighth stack element

The Mod and R/M fields of the ModR/M byte have the same interpretation as the corresponding fields of the integer instructions. The SIB byte and disp (displacement) are optionally present in instructions that have Mod and R/M fields. Their presence depends on the values of Mod and R/M, as for integer instructions.

Table B-27 shows the formats and encodings of the floating-point instructions.

Table B-27. Floating-Point Instruction Formats and Encodings

Instruction and Format	Encoding
F2XM1 – Compute $2^{ST(0)}$ – 1	11011 001 : 1111 0000
FABS – Absolute Value	11011 001 : 1110 0001
FADD – Add	
ST(0) ← ST(0) + 32-bit memory	11011 000 : mod 000 r/m
ST(0) ← ST(0) + 64-bit memory	11011 100 : mod 000 r/m
ST(d) ← ST(0) + ST(i)	11011 d00 : 11 000 ST(i)
FADDP – Add and Pop	
ST(0) ← ST(0) + ST(i)	11011 110 : 11 000 ST(i)
FBLD – Load Binary Coded Decimal	11011 111 : mod 100 r/m
FBSTP – Store Binary Coded Decimal and Pop	11011 111 : mod 110 r/m
FCHS – Change Sign	11011 001 : 1110 0000
FCLEX – Clear Exceptions	11011 011 : 1110 0010
FCOM – Compare Real	
32-bit memory	11011 000 : mod 010 r/m
64-bit memory	11011 100 : mod 010 r/m
ST(i)	11011 000 : 11 010 ST(i)
FCOMP – Compare Real and Pop	
32-bit memory	11011 000 : mod 011 r/m
64-bit memory	11011 100 : mod 011 r/m
ST(i)	11011 000 : 11 011 ST(i)
FCOMPP – Compare Real and Pop Twice	11011 110 : 11 011 001
FCOMIP – Compare Real, Set EFLAGS, and Pop	11011 111 : 11 110 ST(i)
FCOS – Cosine of ST(0)	11011 001 : 1111 1111
FDECSTP – Decrement Stack-Top Pointer	11011 001 : 1111 0110
FDIV – Divide	
ST(0) ← ST(0) ÷ 32-bit memory	11011 000 : mod 110 r/m
ST(0) ← ST(0) ÷ 64-bit memory	11011 100 : mod 110 r/m
ST(d) ← ST(0) ÷ ST(i)	11011 d00 : 1111 R ST(i)

Table B-27. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
FDIVP – Divide and Pop	
ST(0) ← ST(0) ÷ ST(i)	11011 110 : 1111 1 ST(i)
FDIVR – Reverse Divide	
ST(0) ← 32-bit memory ÷ ST(0)	11011 000 : mod 111 r/m
ST(0) ← 64-bit memory ÷ ST(0)	11011 100 : mod 111 r/m
ST(d) ← ST(i) ÷ ST(0)	11011 d00 : 1111 R ST(i)
FDIVRP – Reverse Divide and Pop	
ST(0) ¨ ST(i) ÷ ST(0)	11011 110 : 1111 0 ST(i)
FFREE – Free ST(i) Register	11011 101 : 1100 0 ST(i)
FIADD – Add Integer	
ST(0) ← ST(0) + 16-bit memory	11011 110 : mod 000 r/m
ST(0) ← ST(0) + 32-bit memory	11011 010 : mod 000 r/m
FICOM – Compare Integer	
16-bit memory	11011 110 : mod 010 r/m
32-bit memory	11011 010 : mod 010 r/m
FICOMP – Compare Integer and Pop	
16-bit memory	11011 110 : mod 011 r/m
32-bit memory	11011 010 : mod 011 r/m
FIDIV	
ST(0) ← ST(0) ÷ 16-bit memory	11011 110 : mod 110 r/m
ST(0) ← ST(0) ÷ 32-bit memory	11011 010 : mod 110 r/m
FIDIVR	
ST(0) ← 16-bit memory ÷ ST(0)	11011 110 : mod 111 r/m
ST(0) ← 32-bit memory ÷ ST(0)	11011 010 : mod 111 r/m
FILD – Load Integer	
16-bit memory	11011 111 : mod 000 r/m
32-bit memory	11011 011 : mod 000 r/m
64-bit memory	11011 111 : mod 101 r/m
FIMUL	
ST(0) ← ST(0) × 16-bit memory	11011 110 : mod 001 r/m
ST(0) ← ST(0) × 32-bit memory	11011 010 : mod 001 r/m
FINCSTP – Increment Stack Pointer	11011 001 : 1111 0111
FINIT – Initialize Floating-Point Unit	

Table B-27. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
FIST – Store Integer	
16-bit memory	11011 111 : mod 010 r/m
32-bit memory	11011 011 : mod 010 r/m
FISTP – Store Integer and Pop	
16-bit memory	11011 111 : mod 011 r/m
32-bit memory	11011 011 : mod 011 r/m
64-bit memory	11011 111 : mod 111 r/m
FISUB	
ST(0) ← ST(0) - 16-bit memory	11011 110 : mod 100 r/m
ST(0) ← ST(0) - 32-bit memory	11011 010 : mod 100 r/m
FISUBR	
ST(0) ← 16-bit memory – ST(0)	11011 110 : mod 101 r/m
ST(0) ← 32-bit memory – ST(0)	11011 010 : mod 101 r/m
FLD – Load Real	
32-bit memory	11011 001 : mod 000 r/m
64-bit memory	11011 101 : mod 000 r/m
80-bit memory	11011 011 : mod 101 r/m
ST(i)	11011 001 : 11 000 ST(i)
FLD1 – Load +1.0 into ST(0)	11011 001 : 1110 1000
FLDCW – Load Control Word	11011 001 : mod 101 r/m
FLDENV – Load FPU Environment	11011 001 : mod 100 r/m
FLDL2E – Load $\log_2(\varepsilon)$ into ST(0)	11011 001 : 1110 1010
FLDL2T – Load $\log_2(10)$ into ST(0)	11011 001 : 1110 1001
FLDLG2 – Load $\log_{10}(2)$ into ST(0)	11011 001 : 1110 1100
FLDLN2 – Load $\log_\varepsilon(2)$ into ST(0)	11011 001 : 1110 1101
FLDPI – Load π into ST(0)	11011 001 : 1110 1011

Table B-27. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
FLDZ – Load +0.0 into ST(0)	11011 001 : 1110 1110
FMUL – Multiply	
ST(0) ← ST(0) × 32-bit memory	11011 000 : mod 001 r/m
ST(0) ← ST(0) × 64-bit memory	11011 100 : mod 001 r/m
ST(d) ← ST(0) × ST(i)	11011 d00 : 1100 1 ST(i)
FMULP – Multiply	
ST(i) ← ST(0) × ST(i)	11011 110 : 1100 1 ST(i)
FNOP – No Operation	11011 001 : 1101 0000
FPATAN – Partial Arctangent	11011 001 : 1111 0011
FPREM – Partial Remainder	11011 001 : 1111 1000
FPREM1 – Partial Remainder (IEEE)	11011 001 : 1111 0101
FPTAN – Partial Tangent	11011 001 : 1111 0010
FRNDINT – Round to Integer	11011 001 : 1111 1100
FRSTOR – Restore FPU State	11011 101 : mod 100 r/m
FSAVE – Store FPU State	11011 101 : mod 110 r/m
FSCALE – Scale	11011 001 : 1111 1101
FSIN – Sine	11011 001 : 1111 1110
FSINCOS – Sine and Cosine	11011 001 : 1111 1011
FSQRT – Square Root	11011 001 : 1111 1010
FST – Store Real	
32-bit memory	11011 001 : mod 010 r/m
64-bit memory	11011 101 : mod 010 r/m
ST(i)	11011 101 : 11 010 ST(i)
FSTCW – Store Control Word	11011 001 : mod 111 r/m
FSTENV – Store FPU Environment	11011 001 : mod 110 r/m
FSTP – Store Real and Pop	
32-bit memory	11011 001 : mod 011 r/m
64-bit memory	11011 101 : mod 011 r/m
80-bit memory	11011 011 : mod 111 r/m
ST(i)	11011 101 : 11 011 ST(i)
FSTSW – Store Status Word into AX	11011 111 : 1110 0000
FSTSW – Store Status Word into Memory	11011 101 : mod 111 r/m

Table B-27. Floating-Point Instruction Formats and Encodings (Contd.)

Instruction and Format	Encoding
FSUB – Subtract	
$ST(0) \leftarrow ST(0)$ – 32-bit memory	11011 000 : mod 100 r/m
$ST(0) \leftarrow ST(0)$ – 64-bit memory	11011 100 : mod 100 r/m
$ST(d) \leftarrow ST(0) – ST(i)$	11011 d00 : 1110 R ST(i)
FSUBP – Subtract and Pop	
$ST(0) \leftarrow ST(0) – ST(i)$	11011 110 : 1110 1 ST(i)
FSUBR – Reverse Subtract	
$ST(0) \leftarrow$ 32-bit memory – ST(0)	11011 000 : mod 101 r/m
$ST(0) \leftarrow$ 64-bit memory – ST(0)	11011 100 : mod 101 r/m
$ST(d) \leftarrow ST(i) – ST(0)$	11011 d00 : 1110 R ST(i)
FSUBRP – Reverse Subtract and Pop	
$ST(i) \leftarrow ST(i) – ST(0)$	11011 110 : 1110 0 ST(i)
FTST – Test	11011 001 : 1110 0100
FUCOM – Unordered Compare Real	11011 101 : 1110 0 ST(i)
FUCOMP – Unordered Compare Real and Pop	11011 101 : 1110 1 ST(i)
FUCOMPP – Unordered Compare Real and Pop Twice	11011 010 : 1110 1001
FUCOMI – Unorderd Compare Real and Set EFLAGS	11011 011 : 11 101 ST(i)
FUCOMIP – Unorderd Compare Real, Set EFLAGS, and Pop	11011 111 : 11 101 ST(i)
FXAM – Examine	11011 001 : 1110 0101
FXCH – Exchange ST(0) and ST(i)	11011 001 : 1100 1 ST(i)
FXTRACT – Extract Exponent and Significand	11011 001 : 1111 0100
FYL2X – $ST(1) \times \log_2(ST(0))$	11011 001 : 1111 0001
FYL2XP1 – $ST(1) \times \log_2(ST(0) + 1.0)$	11011 001 : 1111 1001
FWAIT – Wait until FPU Ready	1001 1011