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

Documentation Changes

May 2004

Note: 64-Bit Extension Technology Software Developer's Guide will be renamed to: *Intel® Extended Memory 64 Technology Software Developer's Guide.*

Notice: The IA-32 Intel® Architecture and Intel® Extended Memory 64 Technology may contain design defects or errors known as errata that may cause the product to deviate from published specifications. Current characterized errata are documented in the specification updates.

Document Number: 252046-009

Contents

Revision History

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

Preface

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

Affected Documents/Related Documents

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

Nomenclature

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

Summary Table of Changes

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

Codes Used in Summary Table

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

Summary Table of Documentation Changes

Number	Documentation Changes
1.	CPUID Sector Size, Cache Line Size Expressions Have Been Updated
2.	LDDQU Description Text Has Been Added
3.	ANDNPD/ANDNPS Exception Lists Corrected
4.	PSADBW Instruction Description Section Corrected
5.	Reference to Wrong Step Corrected
6.	APIC Chapter Updated
7.	IA32_MC1_MISC, IA32_MC2_MISC, and IA32_MC2_ADDR Listings Corrected
8.	Ambiguity Correction
9.	Invalid TSS Conditions Listing Has Been Updated
10.	Description Section Corrected
11.	IA-32e Updates for LLDT, LMSW, LTR, SLDT, SMSW, STR
12.	66H Prefix in 64-bit Mode Information Added
13.	MOV—Move to/from Control Registers Section Has Been Updated
14.	PUSH Description Correction
15.	EFLAG Erroneous Statement Removed
16.	Interrupt Handling Description Corrections
17.	IA32_MTRR_DEF_TYPE MSR Definition Corrected
18.	Correction of Error in EFLAGS Treatment in Virtual-8086 Mode
19.	Table B-3 Correction
20.	Appendix E Edits
21.	MSR_PLATFORM_BRV Information Added
22.	Support Pentium M Processor Section Added
23.	MSR Data for Pentium M Processor Has Been Updated
24.	Cache and TLB Descriptor Table Updated
25.	Brand String Table Updated
26.	Pentium M Processor Sections Updated
27.	Name Change for IA32_DEBUGCTL

Documentation Changes

1. CPUID Sector Size, Cache Line Size Expressions Have Been Updated

IA-32 Intel Architecture Software Developer's Manual, Volume 2A, Chapter 3, CPUID-CPU Identification section, Table 3-13 has been corrected. A number of table cells have been updated to correct inconsistent expressions. An error in 78H has also been corrected. The affected table cells are shown below (not all text in the table has been reproduced).

22H	3rd-level cache: 512 KB, 4-way set associative, 64-byte line size, 2 lines per sector
23H	3rd-level cache: 1 MB, 8-way set associative, 64-byte line size, 2 lines per sector
25H	3rd-level cache: 2 MB, 8-way set associative, 64-byte line size, 2 lines per sector
29H	3rd-level cache: 4 MB, 8-way set associative, 64-byte line size, 2 lines per sector

. . . . . .

78H	2nd-level cache: 1 MB, **4-way set associative**, 64-byte line size
79H	2nd-level cache: 128 KB, 8-way set associative, 64-byte line size, 2 lines per sector
7AH	2nd-level cache: 256 KB, 8-way set associative, 64-byte line size, 2 lines per sector
7BH	2nd-level cache: 512 KB, 8-way set associative, 64-byte line size, 2 lines per sector
7CH	2nd-level cache: 1 MB, 8-way set associative, 64-byte line size, 2 lines per sector

2. LDDQU Description Text Has Been Added

IA-32 Intel Architecture Software Developer's Manual, Volume 2B, Chapter 3, LDDQU: Load Unaligned Integer 128 bits section: text has been added. The affected area is shown below. See the change bar for specific lines.

. . . . . .

Implementation Notes

- If the source is aligned to a 16-byte boundary, based on the implementation, the 16 bytes may be loaded more than once. For that reason, the usage of LDDQU should be avoided when using uncached or write-combining (WC) memory regions. For uncached or WC memory regions, keep using MOVDQU.
- This instruction is a replacement for MOVDQU (load) in situations where cache line splits significantly affect performance. It should not be used in situations where store-load forwarding is performance critical. If performance of store-load forwarding is critical to the application, use MOVDQA store-load pairs when data is 128-bit aligned or MOVDQU store-load pairs when data is 128-bit unaligned.
- **If the memory address is not aligned on 16-byte boundary, some implementations may load up to 32 bytes and return 16 bytes in the destination. Some processor implementations may issue multiple loads to access the appropriate 16 bytes. Developers of multi-threaded or multi-processor software should be aware that on these processors the loads will be performed in a non-atomic way.**

. . . . . .

3. ANDNPD/ANDNPS Exception Lists Corrected

IA-32 Intel Architecture Software Developer's Manual, Volume 2B, Chapter 3 - ANDNPD and ANDNPS sections: exception lists have been updated. The affected area for each section is shown below.

ANDNPD—Bitwise Logical AND NOT 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.

… … …

ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision-Floating-Point Value

... ...

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.

4. PSADBW Instruction Description Section Corrected

IA-32 Intel Architecture Software Developer's Manual, Volume 2B, Chapter 4, the PSADBW-Compute Sum of Absolute Differences section: an error has been corrected in the Description paragraph. The affected area is shown below (not all text in the section has been reproduced). The location of the change is indicated by the change bar.

Description

Computes the absolute value of the difference of 8 unsigned byte integers from the source operand (**second operand**) and from the destination operand (**first operand**). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. Figure 4-5 shows the operation of the PSADBW instruction when using 64-bit operands.

5. Reference to Wrong Step Corrected

IA-32 Intel Architecture Software Developer's Manual, Volume 3, Chapter 7, section 7.5.4.1: contains an incorrect reference in a numbered list. This has been corrected. The affected step is shown below with a change bar to indicate the location of the error (not all steps have been included).

.....

15. Broadcasts an INIT-SIPI-SIPI IPI sequence to the APs to wake them up and initialize them:

```
MOV ESI, ICR_LOW; load address of ICR low dword into ESI
MOV EAX, 000C4500H; load ICR encoding for broadcast INIT IPI
                  ; to all APs into EAX
MOV [ESI], EAX  ; broadcast INIT IPI to all APs
; 10-millisecond delay loop
MOV EAX, 000C46XXH; load ICR encoding for broadcast SIPI IP
                 ; to all APs into EAX, where xx is the
                 ; vector computed in step 10.
MOV [ESI], EAX  ; broadcast SIPI IPI to all APs
; 200-microsecond delay loop
MOV [ESI], EAX  ; broadcast second SIPI IPI to all APs
; 200-microsecond delay loop
```

6. APIC Chapter Updated

IA-32 Intel Architecture Software Developer's Manual, Volume 3, Chapter 8 has been updated. Information has been added to multiple sections; this information indicates the model-specific nature of some features. The new information is shown below (with enough of surrounding text to indicate the new text's location; not all text in the chapter is included). See the change bars to locate the updated lines.

8.4.6. Local APIC ID

At power up, system hardware assigns a unique APIC ID to each local APIC on the system bus (for Pentium 4 and Intel Xeon processors) or on the APIC bus (for P6 family and Pentium processors). The hardware assigned APIC ID is based on system topology and includes encoding for socket position and cluster information (see Figure 7-2).

In MP systems, the local APIC ID is also used as a processor ID by the BIOS and the operating system. However, the ability of software to modify the APIC ID is processor model specific. Because of this, operating system software should avoid writing to the local APIC ID register.

The processor receives the hardware assigned APIC ID by sampling pins A11# and A12# and pins BR0# through BR3# (for the Pentium 4, Intel Xeon, and P6 family processors) and pins BE0# through BE3# (for the Pentium processor). The APIC ID latched from these pins is stored in the APIC ID field of the local APIC ID register (see Figure 8-6), and is used as the initial APIC ID for the processor. It is also the value returned to the EBX register, when the CPUID instruction is executed with a source operand value of 1 in the EAX register.

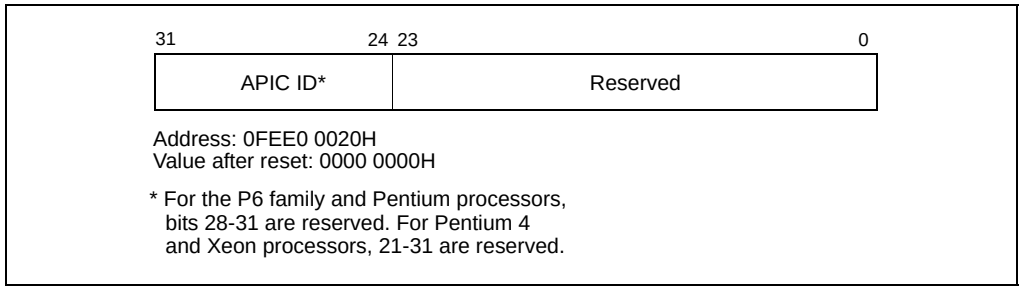

Figure 8-6. Local APIC ID Register

For the P6 family and Pentium processors, the local APIC ID field in the local APIC ID register is 4 bits, and encodings 0H through EH can be used to uniquely identify 15 different processors connected to the APIC bus. For the Pentium 4 and Intel Xeon processors, the xAPIC specification extends the local APIC ID field to 8 bits which can be used to identify up to 255 processors in the system.

Following power up or a hardware reset, software (typically the BIOS software) can modify the APIC ID field in the local APIC ID register for each processor in the system. When changing APIC IDs, software must insure that each APIC ID for each local APIC is unique throughout the system.

... ... omitted text....

8.6. ISSUING INTERPROCESSOR INTERRUPTS

The following sections describe the local APIC facilities that are provided for issuing interprocessor interrupts (IPIs) from software. The primary local APIC facility for issuing IPIs is the interrupt command register (ICR). The ICR can be used for the following functions:

- To send an interrupt to another processor.
- To allow a processor to forward an interrupt that it received but did not service to another processor for servicing.
- To direct the processor to interrupt itself (perform a self interrupt).
- To deliver special IPIs, such as the start-up IPI (SIPI) message, to other processors.

Interrupts generated with this facility are delivered to the other processors in the system through the system bus (for Pentium 4 and Intel Xeon processors) or the APIC bus (for P6 family and Pentium processors). **The ability for a processor to send a lowest priority IPI is model specific and should be avoided by BIOS and operating system software.**

8.6.1. Interrupt Command Register (ICR)

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

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

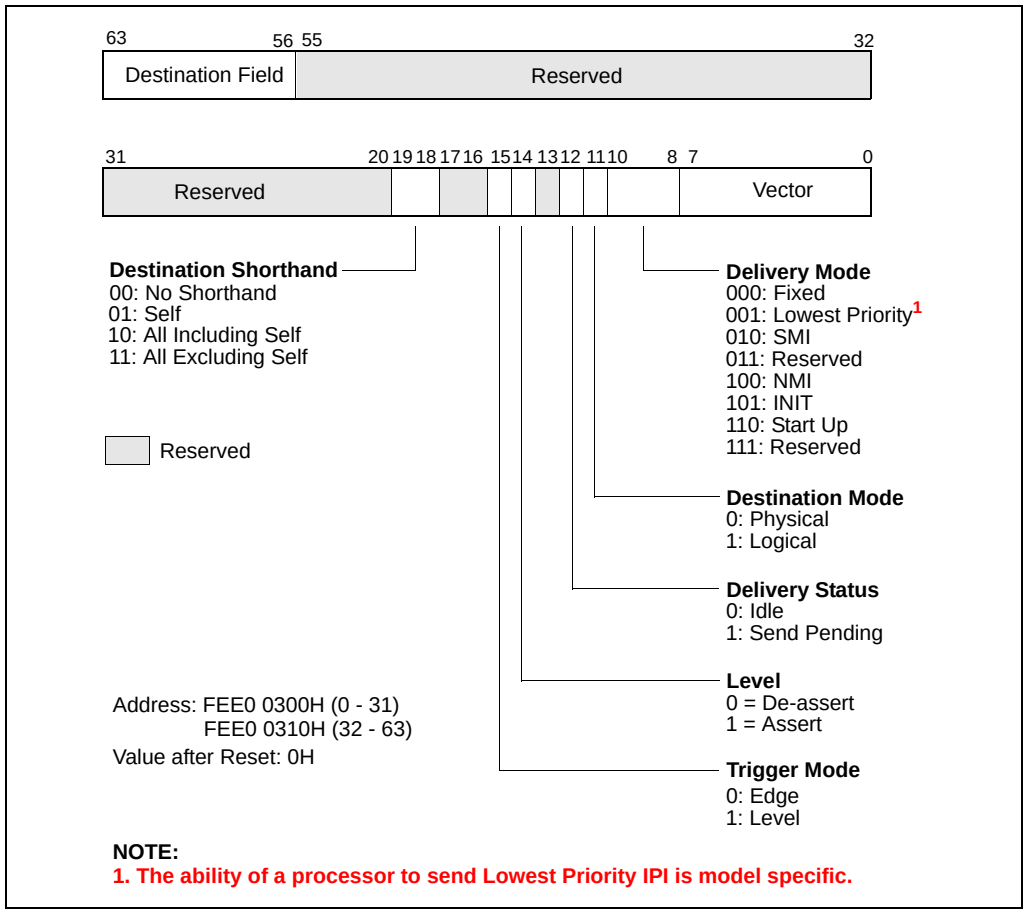

Figure 8-12. Interrupt Command Register (ICR)

The ICR consists of the following fields.

Vector The vector number of the interrupt being sent.

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

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

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

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

011 (Reserved)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Note that not all the combinations of options for the ICR are valid. Table 8-2 shows the valid combinations for the fields in the ICR for the Pentium 4 and Intel Xeon processors; Table 8-3 shows the valid combinations for the fields in the ICR for the P6 family processors.

Table 8-2. Valid Combinations for the Pentium 4 and Intel Xeon Processors' Local xAPIC Interrupt Command Register

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

NOTES:
1. **The ability of a processor to send a lowest priority IPI is model specific**.
2. For these interrupts, if the trigger mode bit is 1 (Level), the local xAPIC will override the bit setting and issue the interrupt as an edge triggered interrupt.
3. X—don't care.
4. When using the "lowest priority" delivery mode and the "all excluding self" destination, the IPI can be redirected back to the issuing APIC, which is essentially the same as the "all including self" destination mode.

Table 8-3. Valid Combinations for the P6 Family Processors Local APIC Interrupt Command Register

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

NOTES:
1. **The ability of a processor to send a lowest priority IPI is model specific.**
2. Treated as edge triggered if level bit is set to 1, otherwise ignored.

3. Treated as edge triggered when Level bit is set to 1; treated as “INIT Level Deassert” message when level bit is set to 0 (deassert). Only INIT level deassert messages are allowed to have the level bit set to 0. For all other messages the level bit must be set to 1.
4. X—Don’t care.
5. The behavior of the APIC is undefined.

8.6.2. Determining IPI Destination

The destination of an IPI can be one, all, or a subset (group) of the processors on the system bus. The sender of the IPI specifies the destination of an IPI with the following APIC registers and fields within the registers:

- The ICR register—The following fields in the ICR register are used to specify the destination of an IPI:
  - — Destination Mode—selects one of two destination modes (physical or logical).
  - — Destination field—In physical destination mode, used to specify the APIC ID of the destination processor; in logical destination mode, used to specify a message destination address (MDA) that can be used to select specific processors in clusters.
  - — Destination Shorthand—A quick method of specifying all processors, all excluding self, or self as the destination.
  - **— Delivery mode, Lowest Priority—Architecturally specifies that a lowest-priority arbitration mechanism be used to select a destination processor from a specified group of processors. The ability of a processor to send a lowest priority IPI is model specific and should be avoided by BIOS and operating system software.**
- Local destination register (LDR)—Used in conjunction with the logical destination mode and MDAs to select the destination processors.
- Destination format register (DFR)—Used in conjunction with the logical destination mode and MDAs to select the destination processors.

How the ICR, LDR, and DFR are used to select an IPI destination depends on the destination mode used: physical, logical, broadcast/self, or lowest-priority delivery mode. These destination modes are described in the following sections.

8.6.2.1. PHYSICAL DESTINATION MODE

In physical destination mode, the destination processor is specified by its local APIC ID (see Section 8.4.6., *Local APIC ID*). For Pentium 4 and Intel Xeon processors, either a single destination (local APIC IDs 00H through FEH) or a broadcast to all APICs (the APIC ID is FFH) may be specified in physical destination mode.

A broadcast IPI (bits 28-31 of the MDA are 1's) or I/O subsystem initiated interrupt with lowest priority delivery mode is not supported in physical destination mode and must not be configured by software. Also, for any non-broadcast IPI or I/O subsystem initiated interrupt with lowest priority delivery mode, software must ensure that APICs defined in the interrupt address are present and enabled to receive interrupts.

For the P6 family and Pentium processors, a single destination is specified in physical destination mode with a local APIC ID of 0H through 0EH, allowing up to 15 local APICs to be addressed on the APIC bus. A broadcast to all local APICs is specified with 0FH.

NOTE

The actual number of local APICs that can be addressed on the system bus may be restricted by hardware.

8.6.2.2. LOGICAL DESTINATION MODE

In logical destination mode, IPI destination is specified using an 8-bit message destination address (MDA), which is entered in the destination field of the ICR. Upon receiving an IPI message that was sent using logical destination mode, a local APIC compares the MDA in the message with the values in its LDR and DFR to determine if it should accept and handle the IPI. **For both configurations of logical destination mode, when combined with lowest priority delivery mode, software is responsible for ensuring that all of the local APICs included in or addressed by the IPI or I/O subsystem interrupt are present and enabled to receive the interrupt.**

Figure 13 shows the layout of the logical destination register (LDR). The 8-bit logical APIC ID field in this register is used to create an identifier that can be compared with the MDA.

NOTE

The logical APIC ID should not be confused with the local APIC ID that is contained in the local APIC ID register.

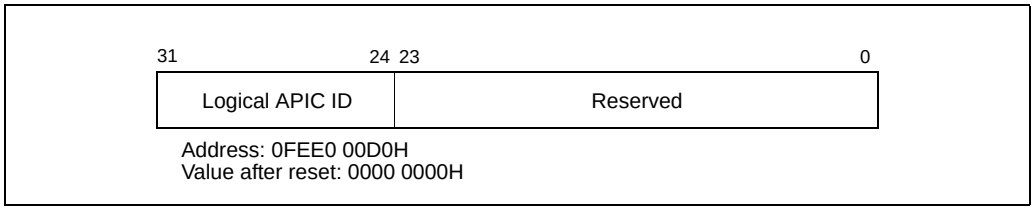

Figure 8-13. Logical Destination Register (LDR)

Figure 14 shows the layout of the destination format register (DFR). The 4-bit model field in this register selects one of two models (flat or cluster) that can be used to interpret the MDA when using logical destination mode.

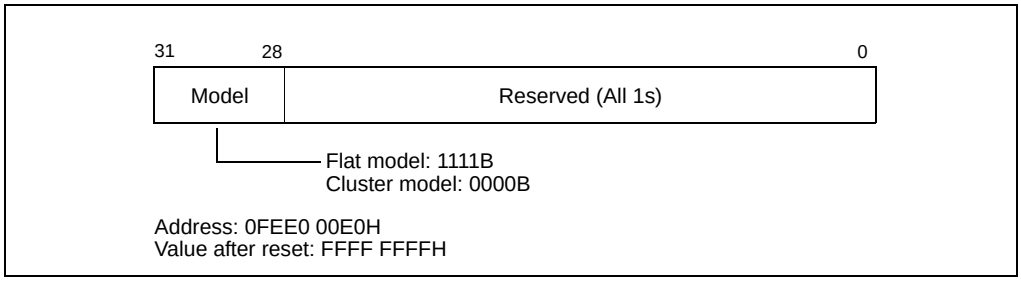

Figure 8-14. Destination Format Register (DFR)

The interpretation of MDA for the two models is described in the following paragraphs.

Flat Model. This model is selected by programming DFR bits 28 through 31 to 1111. Here, a unique logical APIC ID can be established for up to 8 local APICs by setting a different bit in the logical APIC ID field of the LDR for each local APIC. An group of local APICs can then be selected by setting one or more bits in the MDA.

Each local APIC performs a bit-wise AND of the MDA and its logical APIC ID. If a true condition is detected, the local APIC accepts the IPI message. A broadcast to all APICs is achieved by setting the MDA to all 1s.

Cluster Model. This model is selected by programming DFR bits 28 through 31 to 0000. This model supports two basic destination schemes: flat cluster and hierarchical cluster.

The flat cluster destination model is only supported for P6 family and Pentium processors. Using this model, all APICs are assumed to be connected through the APIC bus. Bits 28 through 31 of the MDA contains the encoded address of the destination cluster, and bits 24 through 27 identify up to four local APICs within the cluster (each bit is assigned to one local APIC in the cluster, as in the flat connection model). To identify one or more local APICs, bits 28 through 31 of the MDA are compared with bits 28 through 31 of the LDR to determine if a local APIC is part of the cluster. Bits 24 through 27 of the MDA are compared with Bits 24 through 27 of the LDR to identify a local APICs within the cluster.

Sets of processors within a cluster can be specified by writing the target cluster address in bits 28 through 31 of the MDA and setting selected bits in bits 24 through 27 of the MDA, corresponding to the chosen members of the cluster. In this mode, 15 clusters (with cluster addresses of 0 through 14) each having 4 local APICs can be specified in the message. For the P6 and Pentium processor's local APICs, however, the APIC arbitration ID supports only 15 APIC agents, and hence the total number of processors and their local APICs supported in this mode is limited to 15. Broadcast to all local APICs is achieved by setting all destination bits to one. This guarantees a match on all clusters, and selects all APICs in each cluster. **A broadcast IPI or I/O subsystem broadcast interrupt with lowest priority delivery mode is not supported in cluster mode and must not be configured by software.**

The hierarchical cluster destination model can be used with Pentium 4, Intel Xeon, P6 family, or Pentium processors. With this model, a hierarchical network can be created by connecting different flat clusters via independent system or APIC buses. This scheme requires a cluster manager within each cluster, which is responsible for handling message passing between system or APIC buses. One cluster contains up to 4 agents. Thus 15 cluster managers, each with 4 agents, can form a network of up to 60 APIC agents. Note that hierarchical APIC networks requires a special cluster manager device, which is not part of the local or the I/O APIC units.

... ... omitted text....

8.11.1. Message Address Register Format

The format of the Message Address Register (lower 32-bits) is shown in Figure .

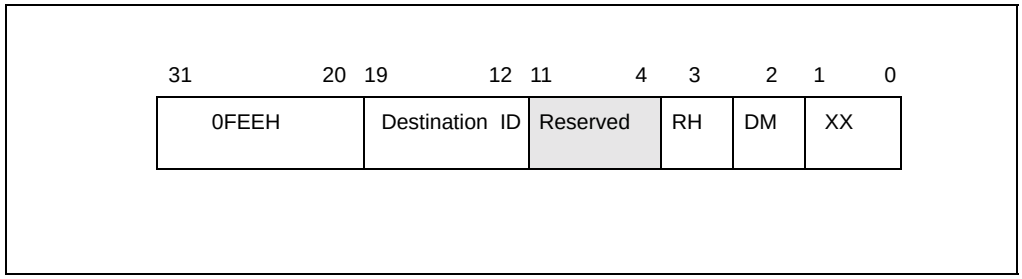

Figure 8-23. Layout of the MSI Message Address Register

Fields in the Message Address Register are as follows:

1. Bits 31-20: These bits contain a fixed value for interrupt messages (0FEEH). This value locates interrupts at the 1MB area with a base address of 4G – 18M. All accesses to this region are directed as interrupt messages. Care must to be taken to ensure that no other device claims the region as I/O space.
2. Destination ID: This field contains an 8-bit destination ID. It identifies the message's target processor(s). The destination ID corresponds to bits 63:56 of the I/O APIC Redirection Table Entry if the IOAPIC is used to dispatch the interrupt to the processor(s).
3. Redirection Hint Indication (RH): This bit indicates whether the message should be directed to the processor with the lowest interrupt priority among processors that can receive the interrupt.

    - **When RH is 0, the interrupt is directed to the processor listed in the Destination ID field.**
    - **When RH is 1 and the physical destination mode is used, the Destination ID field must not be set to 0xFF; it must point to a processor that is present and enabled to receive the interrupt.**
    - **When RH is 1 and the logical destination mode is active in a system using a flat addressing model, the Destination ID field must be set so that bits set to 1 identify processors that are present and enabled to receive the interrupt.**
    - **If RH is set to 1 and the logical destination mode is active in a system using cluster addressing model, then Destination ID field must not be set to 0xFF; the processors identified with this field must be present and enabled to receive the interrupt.**

Destination Mode (DM): This bit indicates whether the Destination ID field should be interpreted as logical or physical APIC ID for delivery of the lowest priority interrupt. If RH is 1 and DM is 0, the Destination ID field is in physical destination mode and only the processor in the system that has the matching APIC ID is considered for delivery of that interrupt (this means no re-direction). If RH is 1 and DM is 1, the Destination ID Field is interpreted as in logical destination mode and the redirection is limited to only those processors that are part of the logical group of processors based on the processor's logical APIC ID and the Destination ID field in the message. The logical group of processors consists of those identified by matching the 8-bit Destination ID with the logical destination identified by the Destination Format Register and the Logical Destination Register in each local APIC. The details are similar to those described in Section 8.6.2., *Determining IPI Destination*. If RH is 0, then the DM bit is ignored and the message is sent ahead independent of whether the physical or logical destination mode is used.

7. IA32_MC1_MISC, IA32_MC2_MISC and IA32_MC2_ADDR Listings Corrected

For Table B-5, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*; a footnote has been added to clarify when and why specific MSRs may or may not be present. The affected cells in the table are shown below (not all cells in the table have been included).

Table B-5. IA-32 Architectural MSRs

Register Address		Architectural Name	Former Name	IA-32 Processor Family Introduced In
... .				
407H	1031	IA32_MC1_MISC[1]	MC1_MISC	P6 Family Processors
408H	1032	IA32_MC2_CTL	MC2_CTL	P6 Family Processors
409H	1033	IA32_MC2_STATUS	MC2_STATUS	P6 Family Processors
40AH	1034	IA32_MC2_ADDR[a]	MC2_ADDR	P6 Family Processors
40BH	1035	IA32_MC2_MISC[1]	MC2_MISC	P6 Family Processors
40CH	1036	IA32_MC3_CTL	MC3_CTL	P6 Family Processors
40DH	1037	IA32_MC3_STATUS	MC3_STATUS	P6 Family Processors
40EH	1038	IA32_MC3_ADDR	MC3_ADDR	P6 Family Processors
40FH	1039	IA32_MC3_MISC	MC3_MISC	P6 Family Processors
600H	1536	IA32_DS_AREA		Pentium 4 Processor

a. These MSRs may or may not be present; this depends on flag settings in IA32_MC*i*_STATUS. See Section 14.3.2.3. and Section 14.3.2.4. for more information.

*** *** ***

8. Ambiguity Correction

For *Sections 4.11.3 and 4.12, IA-32 Intel Architecture Software Developer's Manual, Volume 3;* the wording has been corrected to be more precise. Updated text and table cells are marked by change bars.

4.11.3. Page Type

The page-level protection mechanism recognizes two page types:

- Read-only access (R/W flag is 0).
- Read/write access (R/W flag is 1).

When the processor is in supervisor mode and the WP flag in register CR0 is clear (its state following reset initialization), all pages are both readable and writable (write-protection is ignored). When the processor is in user mode, it can write only to user-mode pages that are read/write accessible. User-mode pages which are read/write or read-only are readable; supervisor-mode pages are neither readable nor writable from user mode. A page-fault exception is generated on any attempt to violate the protection rules.

The P6 family, Pentium, and Intel486 processors allow user-mode pages to be write-protected against supervisor-mode access. **Setting the WP flag in register CR0 to 1 enables supervisor-mode sensitivity to user-mode, write protected pages. Supervisor pages which are read-only are not writeable from any privilege level, regardless of WP setting.** This supervisor write-protect feature is useful for implementing a "copy-on-write" strategy used by some operating systems, such as UNIX*, for task creation (also called forking or spawning). When a new task is created, it is possible to copy the entire address space of the parent task. This gives the child task a complete, duplicate set of the parent's segments and pages. An alternative copy-on-write strategy saves memory space and time by mapping the child's segments and pages to the same segments and pages used by the parent task. A private copy of a page gets created only when one of the tasks writes to the page. By using the WP flag and marking the shared pages as read-only, the supervisor can detect an attempt to write to a user-level page, and can copy the page at that time.

4.11.4. Combining Protection of Both Levels of Page Tables

For any one page, the protection attributes of its page-directory entry (first-level page table) may differ from those of its page-table entry (second-level page table). The processor checks the protection for a page in both its page-directory and the page-table entries. Table 4-4 shows the protection provided by the possible combinations of protection attributes when the WP flag is clear.

4.11.5. Overrides to Page Protection

The following types of memory accesses are checked as if they are privilege-level 0 accesses, regardless of the CPL at which the processor is currently operating:

- Access to segment descriptors in the GDT, LDT, or IDT.
- Access to an inner-privilege-level stack during an inter-privilege-level call or a call to in exception or interrupt handler, when a change of privilege level occurs.

4.12. COMBINING PAGE AND SEGMENT PROTECTION

When paging is enabled, the processor evaluates segment protection first, then evaluates page protection. If the processor detects a protection violation at either the segment level or the page level, the memory access is not carried out and an exception is generated. If an exception is generated by segmentation, no paging exception is generated.

Page-level protections cannot be used to override segment-level protection. For example, a code segment is by definition not writable. If a code segment is paged, setting the R/W flag for the pages to read-write does not make the pages writable. Attempts to write into the pages will be blocked by segment-level protection checks.

Page-level protection can be used to enhance segment-level protection. For example, if a large read-write data segment is paged, the page-protection mechanism can be used to write-protect individual pages.

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

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

NOTE:

* If the WP flag of CR0 is set, the access type is determined by the R/W flags of the page-directory and page-table entries.

*** *** ***

9. Listing of Invalid TSS Conditions Has Been Updated

For Table 5-6, IA-32 Intel Architecture Software Developer's Manual, Volume 3; Additional invalid conditions have been listed. The updated table is as follows:

Interrupt 10—Invalid TSS Exception (#TS)

Exception Class Fault.

Description

Indicates that there was an error related to a TSS. Such an error might be detected during a task switch or during the execution of instructions that use information from a TSS. Table 5-6 shows the conditions that cause an invalid TSS exception to be generated.

Table 5-6. Invalid TSS Conditions

Error Code Index	Invalid Condition
TSS segment selector index	The TSS segment limit is less than 67H for 32-bit TSS or less than 2CH for 16-bit TSS.
TSS segment selector index	During an IRET task switch, the TI flag in the TSS segment selector indicates the LDT.
TSS segment selector index	During an IRET task switch, the TSS segment selector exceeds descriptor table limit.
TSS segment selector index	During an IRET task switch, the busy flag in the TSS descriptor indicates an inactive task.
TSS segment selector index	During an IRET task switch, an attempt to load the backlink limit faults.
TSS segment selector index	During an IRET task switch, the backlink is a NULL selector.
TSS segment selector index	During an IRET task switch, the backlink points to a descriptor which is not a busy TSS.
TSS segment selector index	The new TSS descriptor is beyond the GDT limit.
TSS segment selector index	The new TSS descriptor is not writeable.
TSS segment selector index	Stores to the old TSS encounter a fault condition.
TSS segment selector index	The old TSS descriptor is not writeable for a jump or IRET task switch.
TSS segment selector index	The new TSS backlink is not writeable for a call or exception task switch.
TSS segment selector index	The new TSS selector is null on an attempt to lock the new TSS.
TSS segment selector index	The new TSS selector has the TI bit set on an attempt to lock the new TSS.
TSS segment selector index	The new TSS descriptor is not an available TSS descriptor on an attempt to lock the new TSS.
LDT segment selector index	LDT or LDT not present.
Stack segment selector index	The stack segment selector exceeds descriptor table limit.
Stack segment selector index	The stack segment selector is NULL.
Stack segment selector index	The stack segment descriptor is a non-data segment.
Stack segment selector index	The stack segment is not writable.
Stack segment selector index	The stack segment DPL != CPL.
Stack segment selector index	The stack segment selector RPL != CPL.

Table 5-6. Invalid TSS Conditions (Continued)

Error Code Index	Invalid Condition
Code segment selector index	The code segment selector exceeds descriptor table limit.
Code segment selector index	The code segment selector is NULL.
Code segment selector index	The code segment descriptor is not a code segment type.
Code segment selector index	The nonconforming code segment DPL != CPL.
Code segment selector index	The conforming code segment DPL is greater than CPL.
Data segment selector index	The data segment selector exceeds the descriptor table limit.
Data segment selector index	The data segment descriptor is not a readable code or data type.
Data segment selector index	The data segment descriptor is a nonconforming code type and RPL > DPL.
Data segment selector index	The data segment descriptor is a nonconforming code type and CPL > DPL.
TSS segment selector index	The TSS segment selector is NULL for LTR.
TSS segment selector index	The TSS segment selector has the TI bit set for LTR.
TSS segment selector index	The TSS segment descriptor/upper descriptor is beyond the GDT segment limit.
TSS segment selector index	The TSS segment descriptor is not an available TSS type.

*** *** ***

10. Description Section Corrected

For the OUTS/OUTSB/OUTSW/OUTSD—Output String to Port section, *Chapter 4, IA-32 Intel Architecture Software Developer's Manual, Volume 2B;* text in the Description sub-section has been updated to correct an erroneous reference. The affected area is shown below in context; affected lines are marked by a change bar.

OUTS/OUTSB/OUTSW/OUTSD—Output String to Port

Opcode	Instruction	Description
6E	OUTS DX, m8	Output byte from memory location specified in DS:(E)SI to I/O port specified in DX
6F	OUTS DX, m16	Output word from memory location specified in DS:(E)SI to I/O port specified in DX
6F	OUTS DX, m32	Output doubleword from memory location specified in DS:(E)SI to I/O port specified in DX
6E	OUTSB	Output byte from memory location specified in DS:(E)SI to I/O port specified in DX
6F	OUTSW	Output word from memory location specified in DS:(E)SI to I/O port specified in DX
6F	OUTSD	Output doubleword from memory location specified in DS:(E)SI to I/O port specified in DX

Description

Copies data from the source operand (second operand) to the I/O port specified with the destination operand (first operand). The source operand is a memory location, the address of which is read from either the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). (The DS segment may be overridden with a segment override prefix.) The destination operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

*** *** ***

11. IA-32e Updates for LLDT, LMSW, LTR, SLDT, SMSW, STR

For LLDT, LMSW, LTR, SLDT, SMSW, STR; Chapters 2 & 3 in the *Intel® Extended Memory 64 Technology Software Developer's Guide, Volumes 1 & 2;* IA-32e Mode Operation sections have been updated. Change bars mark the corrections.

LLDT—Load Local Descriptor Table Register

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
0F 00 /2	LLDT r/m16	Valid	Valid	Load segment selector r/m16 into LDTR

Flags Affected

None.

IA-32e Mode Operation

Operand size fixed at 16 bits.

References 64-bit mode descriptor to load 64-bit base.

*** *** ***

LMSW—Load Machine Status Word

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
0F 01 /6	LMSW r/m16	Valid	Valid	Loads r/m16 in machine status word of CR0

Flags Affected

None.

IA-32e Mode Operation

Same as legacy mode.

Operand size fixed at 16 bits.

*** *** ***

LTR—Load Task Register

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
0F 00 /3	LTR r/m16	Valid	Valid	Load r/m16 into task register

Flags Affected

None.

IA-32e Mode Operation

Operand size fixed at 16 bits.

References 64-bit mode descriptor to load 64-bit base.

*** *** ***

SLDT—Store Local Descriptor Table Register

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
0F 00 /0	SLDT *r/m16*	Valid	Valid	Stores segment selector from LDTR in r/m16

Flags Affected

None.

IA-32e Mode Operation

The behavior of the SLDT instruction is defined by the following examples.

- SLDT r16 operands size 16, store 16-bit selector in r16
- SLDT r32 operands size 32, zero-extend 16-bit selector and store in r32
- SLDT r64 operands size 64, zero-extend 16-bit selector and store in r64
- SLDT m16 operands size 16, store 16-bit selector in m16
- SLDT m16 operands size 32, store 16-bit selector in m16 (not m32)
- SLDT m16 operands size 64, store 16-bit selector in m16 (not m64)

*** *** ***

SMSW—Store Machine Status Word

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
0F 01 /4	SMSW *r/m16*	Valid	Valid	Store machine status word to *r/m16*

Flags Affected

None.

IA-32e Mode Operation

The behavior of the SMSW instruction is defined by the following examples.

- SMSW r16 operands size 16, store CR0[15:0] in r16
- SMSW r32 operands size 32, zero-extend CR0[31:0], and store in r32
- SMSW r64 operands size 64, zero-extend CR0[63:0], and store in r64
- SMSW m16 operands size 16, store CR0[15:0] in m16
- SMSW m16 operands size 32, store CR0[15:0] in m16 (not m32)
- SMSW m16 operands size 64, store CR0[15:0] in m16 (not m64)

*** *** ***

STR—Store Task Register

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
0F 00 /1	STR *r/m16*	Valid	Valid	Stores segment selector from TR in r/m16

Flags Affected

None.

IA-32e Mode Operation

Same as legacy mode.

Memory operand fixed at 16 bits.

Zero extend 2 byte TR selector to 64 bits and store to register operand.

*** *** ***

12. 66H Prefix in 64-bit Mode Information Added

In Chapter 1, *Intel® Extended Memory 64 Technology Software Developer's Guide, Volume 1;* a section has been added to address the 66H prefix. This section is shown below.

1.7.1. Other guidelines

- In the initial implementation of Intel® EM64T, an operand-size prefix (66H) is ignored when used in 64-bit mode with a near branch. In 64-bit mode, a near branch uses 32-bit displacement (the instruction pointer is advanced to a linear address that is the next sequential instruction offset by a 32 bit displacement, sign extended to 64-bit). Software must not rely on this behavior as future implementations may be different.

.*** *** ***

13. MOV—Move to/from Control Registers Section Has Been Updated

In the MOV—Move to/from Control Registers section, *Chapter 4, Intel® Extended Memory 64 Technology Software Developer's Guide, Volume 2;* exception sub-sections have been updated. The entire section is shown below with affected areas marked with change bars.

MOV—Move to/from Control Registers

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 22 /*r*	MOV CR0,*r32*	Valid	Valid	Move *r32* to CR0
REX.W + 0F 22 /*r*	MOV CR0,*r64*	Valid	N.E.	Move *r64* to extended CR0.
0F 22 /*r*	MOV CR2,*r32*	Valid	Valid	Move *r32* to CR2
REX.W + 0F 22 /*r*	MOV CR2,*r64*	Valid	N.E.	Move *r64* to extended CR2.
0F 22 /*r*	MOV CR3,*r32*	Valid	Valid	Move *r32* to CR3
REX.W + 0F 22 /*r*	MOV CR3,*r64*	Valid	N.E.	Move *r64* to extended CR3.
0F 22 /*r*	MOV CR4,*r32*	Valid	Valid	Move *r32* to CR4
REX.W + 0F 22 /*r*	MOV CR4,*r64*	Valid	N.E.	Move *r64* to extended CR4.
0F 20 /*r*	MOV *r32*,CR0	Valid	Valid	Move CR0 to *r32*
REX.W + 0F 20 /*r*	MOV *r64*,CR0	Valid	N.E.	Move extended CR0 to *r64*.
0F 20 /*r*	MOV *r32*,CR2	Valid	Valid	Move CR2 to *r32*
REX.W + 0F 20 /*r*	MOV *r64*,CR2	Valid	N.E.	Move extended CR2 to *r64*.
0F 20 /*r*	MOV *r32*,CR3	Valid	Valid	Move CR3 to *r32*
REX.W + 0F 20 /*r*	MOV *r64*,CR3	Valid	N.E.	Move extended CR3 to *r64*.
0F 20 /*r*	MOV *r32*,CR4	Valid	Valid	Move CR4 to *r32*
REX.W + 0F 20 /*r*	MOV *r64*,CR4	Valid	N.E.	Move extended CR4 to *r64*.
0F 20 /*r*	MOV *r32*,CR8	Valid	N.E.	Move CR8 to *r32*
REX.W + 0F 20 /*r*	MOV *r64*,CR8	Valid	N.E.	Move extended CR8 to *r64*.

Flags Affected

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

IA-32e Mode Operation

Promoted to 64-bits.

Operand size fixed at 64-bits (see Control registers section).

Enables access to new registers R8-R15.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
	If an attempt is made to write a 1 to any reserved bit in CR4.
	Attempting to activate IA-32e mode (MOV CR0) with a 286 TSS in TR.
	Attempting to activate IA-32e mode (MOV CR0) with CR4.PAE not set.
	Attempting to activate IA-32e mode (MOV CR0) with a CS that has the L-bit set.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

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

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).

Attempting to clear CR0.PG.

If an attempt is made to write a 1 to any reserved bit in CR4.

If an attempt is made to write a 1 to any reserved bit in CR8.

If an attempt is made to write a 1 to any reserved bit in CR3.

If an attempt is made to leave IA-32e mode by clearing CR4.PAE.

*** *** ***

14. PUSH Description Correction

In the PUSH—Push Word or Doubleword Onto the Stack section, *Chapter 3, Intel® Extended Memory 64 Technology Software Developer's Guide, Volume 2;* the summary table has been updated. The table is shown below with change bars marking affected cells.

PUSH—Push Word or Doubleword Onto the Stack

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
FF /6	PUSH *r/m16*	Valid	Valid	Push *r/m16*
FF /6	PUSH *r/m32*	N.E.	Valid	Push *r/m32*
FF /6	PUSH *r/m64*	Valid	N.E.	Push *r/m64.* Default operand size 64-bits.
50+*rw*	PUSH *r16*	Valid	Valid	Push *r16*
50+*rd*	PUSH *r32*	N.E.	Valid	Push *r32*
50+*rd*	PUSH *r64*	Valid	N.E.	Push *r64.* Default operand size 64-bits.
6A	PUSH *imm8*	Valid	Valid	Push *imm8*
68	PUSH *imm16*	Valid	Valid	Push *imm16*
68	PUSH *imm32*	N.E.	Valid	Push *imm32*
68	PUSH *imm64*	Valid	N.E.	Push zero-extended *imm32.* Default operand size 64-bits.
0E	PUSH CS	Inv.	Valid	Push CS
16	PUSH SS	Inv.	Valid	Push SS
1E	PUSH DS	Inv.	Valid	Push DS
06	PUSH ES	Inv.	Valid	Push ES
0F A0	PUSH FS	Valid	Valid	Push FS and decrement stack pointer by 16 bits.
0F A0	PUSH FS	N.E.	Valid	Push FS and decrement stack pointer by 32 bits.
0F A0	PUSH FS	Valid	N.E.	Push FS. Default operand size 64-bits. (66h override causes 16-bit operation)
0F A8	PUSH GS	Valid	Valid	Push GS and decrement stack pointer by 16 bits.
0F A8	PUSH GS	N.E.	Valid	Push GS and decrement stack pointer by 32 bits.
0F A8	PUSH GS	Valid	N.E.	Push GS, Default operand size 64-bits. (66h override causes 16-bit operation

*** *** ***

15. EFLAG Erroneous Statement Removed

In Section 16.3.2, *IA-32 Intel Architecture Software Developer's Manual, Volume 3;* text in a numbered list has been edited to remove an error. The updated text is provided below (in context). Not all of the section is included.

If the IF flag is set and the VIF and VIP flags are enabled, and the processor receives a maskable hardware interrupt (interrupt vector 0 through 255), the processor performs and the interrupt handler software should perform the following operations:

1. The processor invokes the protected-mode interrupt handler for the interrupt received, as described in the following steps. These steps are almost identical to those described for method 1 interrupt and exception handling in Section 16.3.1.1., *Handling an Interrupt or Exception Through a Protected-Mode Trap or Interrupt Gate*:
   a. Switches to 32-bit protected mode and privilege level 0.
   b. Saves the state of the processor on the privilege-level 0 stack. The states of the EIP, CS, EFLAGS, ESP, SS, ES, DS, FS, and GS registers are saved (see Figure 16-4).
   c. Clears the segment registers.
   d. Clears the VM flag in the EFLAGS register.
   e. Begins executing the selected protected-mode interrupt handler.
2. The recommended action of the protected-mode interrupt handler is to read the VM flag from the EFLAGS image on the stack. If this flag is set, the handler makes a call to the virtual-8086 monitor.

*** *** ***

16. Interrupt Handling Description Corrections

In Sections 16.3.1.1 through 16.3.3, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*; a number of corrections have been made. The area involved is shown below with affected areas marked with change bars.

16.3.1.1. HANDLING AN INTERRUPT OR EXCEPTION THROUGH A PROTECTED-MODE TRAP OR INTERRUPT GATE

When an interrupt or exception vector points to a 32-bit trap or interrupt gate in the IDT, the gate must in turn point to a nonconforming, privilege-level 0, code segment. When accessing this code segment, processor performs the following steps.

1. Switches to 32-bit protected mode and privilege level 0.
2. Saves the state of the processor on the privilege-level 0 stack. The states of the EIP, CS, EFLAGS, ESP, SS, ES, DS, FS, and GS registers are saved (see Figure 16-4).
3. Clears the segment registers. Saving the DS, ES, FS, and GS registers on the stack and then clearing the registers lets the interrupt or exception handler safely save and restore these registers regardless of the type segment selectors they contain (protected-mode or 8086-style). The interrupt and exception handlers, which may be called in the context of either a protected-mode task or a virtual-8086-mode task, can use the same code sequences for saving and restoring the registers for any task. Clearing these registers before execution of the IRET instruction does not cause a trap in the interrupt handler. Interrupt procedures that expect values in the segment registers or that return values in the segment registers must use the register images saved on the stack for privilege level 0.
4. Clears VM, NT, RF and TF flags (in the EFLAGS register). If the gate is an interrupt gate, clears the IF flag.
5. Begins executing the selected interrupt or exception handler.

If the trap or interrupt gate references a procedure in a conforming segment or in a segment at a privilege level other than 0, the processor generates a general-protection exception (#GP). Here, the error code is the segment selector of the code segment to which a call was attempted.

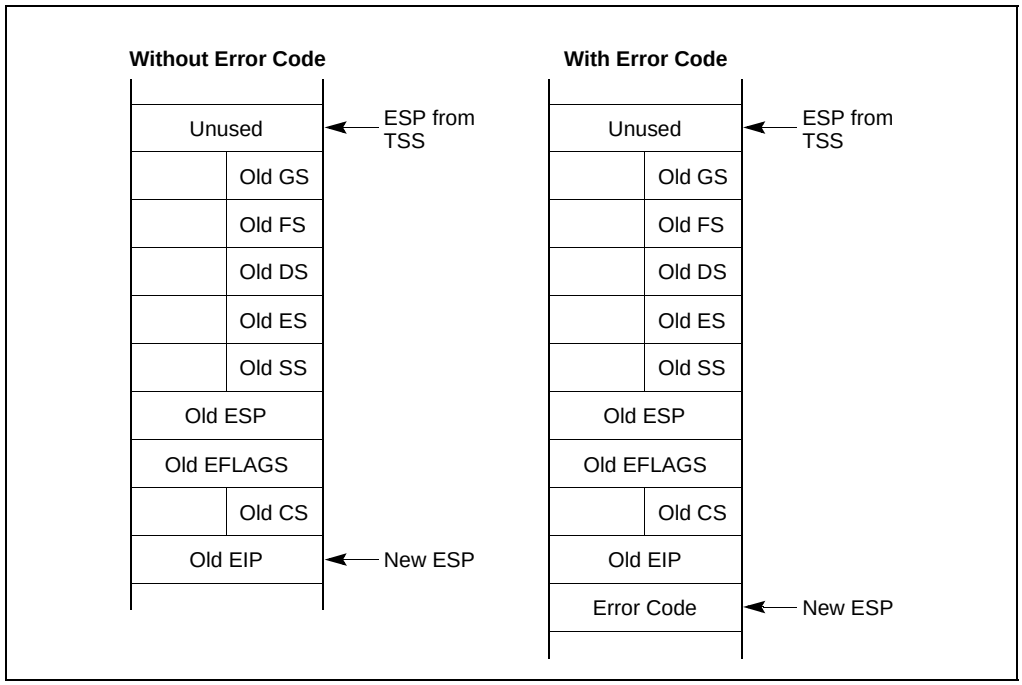

Figure 16-4. Privilege Level 0 Stack After Interrupt or Exception in Virtual-8086 Mode

Interrupt and exception handlers can examine the VM flag on the stack to determine if the interrupted procedure was running in virtual-8086 mode. If so, the interrupt or exception can be handled in one of three ways:

- The protected-mode interrupt or exception handler that was called can handle the interrupt or exception.
- The protected-mode interrupt or exception handler can call the virtual-8086 monitor to handle the interrupt or exception.
- The virtual-8086 monitor (if called) can in turn pass control back to the 8086 program's interrupt and exception handler.

If the interrupt or exception is handled with a protected-mode handler, the handler can return to the interrupted program in virtual-8086 mode by executing an IRET instruction. This instruction loads the EFLAGS and segment registers from the images saved in the privilege level 0 stack (see Figure 16-4). A set VM flag in the EFLAGS image causes the processor to switch back to virtual-8086 mode. The CPL at the time the IRET instruction is executed must be 0, otherwise the processor does not change the state of the VM flag.

The virtual-8086 monitor runs at privilege level 0, like the protected-mode interrupt and exception handlers. It is commonly closely tied to the protected-mode general-protection exception (#GP, vector 13) handler. If the protected-mode interrupt or exception handler calls the virtual-8086 monitor to handle the interrupt or exception, the return from the virtual-8086 monitor to the interrupted virtual-8086 mode program requires two return instructions: a RET instruction to return to the protected-mode handler and an IRET instruction to return to the interrupted program.

The virtual-8086 monitor has the option of directing the interrupt and exception back to an interrupt or exception handler that is part of the interrupted 8086 program, as described in Section 16.3.1.2., *Handling an Interrupt or Exception With an 8086 Program Interrupt or Exception Handler*.

16.3.1.2. HANDLING AN INTERRUPT OR EXCEPTION WITH AN 8086 PROGRAM INTERRUPT OR EXCEPTION HANDLER

Because it was designed to run on an 8086 processor, an 8086 program running in a virtual-8086-mode task contains an 8086-style interrupt vector table, which starts at linear address 0. If the virtual-8086 monitor correctly directs an interrupt or exception vector back to the virtual-8086-mode task it came from, the handlers in the 8086 program can handle the interrupt or exception. The virtual-8086 monitor must carry out the following steps to send an interrupt or exception back to the 8086 program:

1. Use the 8086 interrupt vector to locate the appropriate handler procedure in the 8086 program interrupt table.
2. Store the EFLAGS (low-order 16 bits only), CS and EIP values of the 8086 program on the privilege-level 3 stack. This is the stack that the virtual-8086-mode task is using. (The 8086 handler may use or modify this information.)
3. Change the return link on the privilege-level 0 stack to point to the privilege-level 3 handler procedure.
4. Execute an IRET instruction to pass control to the 8086 program handler.
5. When the IRET instruction from the privilege-level 3 handler triggers a general-protection exception (#GP) and thus effectively again calls the virtual-8086 monitor, restore the return link on the privilege-level 0 stack to point to the original, interrupted, privilege-level 3 procedure.
6. Copy the low order 16 bits of the EFLAGS image from the privilege-level 3 stack to the privilege-level 0 stack (because some 8086 handlers modify these flags to return information to the code that caused the interrupt).
7. Execute an IRET instruction to pass control back to the interrupted 8086 program.

Note that if an operating system intends to support all 8086 MS-DOS-based programs, it is necessary to use the actual 8086 interrupt and exception handlers supplied with the program. The reason for this is that some programs modify their own interrupt vector table to substitute (or hook in series) their own specialized interrupt and exception handlers.

16.3.1.3. HANDLING AN INTERRUPT OR EXCEPTION THROUGH A TASK GATE

When an interrupt or exception vector points to a task gate in the IDT, the processor performs a task switch to the selected interrupt- or exception-handling task. The following actions are carried out as part of this task switch:

1. The EFLAGS register with the VM flag set is saved in the current TSS.
2. The link field in the TSS of the called task is loaded with the segment selector of the TSS for the interrupted virtual-8086-mode task.
3. The EFLAGS register is loaded from the image in the new TSS, which clears the VM flag and causes the processor to switch to protected mode.
4. The NT flag in the EFLAGS register is set.
5. The processor begins executing the selected interrupt- or exception-handler task.

When an IRET instruction is executed in the handler task and the NT flag in the EFLAGS register is set, the processors switches from a protected-mode interrupt- or exception-handler task back to a virtual-8086-mode task. Here, the EFLAGS and segment registers are loaded from images saved in the TSS for the virtual-8086-mode task. If the VM flag is set in the EFLAGS image, the processor switches back to virtual-8086 mode on the task switch. The CPL at the time the IRET instruction is executed must be 0, otherwise the processor does not change the state of the VM flag.

16.3.2. Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism

Maskable hardware interrupts are those interrupts that are delivered through the INTR# pin or through an interrupt request to the local APIC (see Section 5.3.2., “Maskable Hardware Interrupts”). These interrupts can be inhibited (masked) from interrupting an executing program or task by clearing the IF flag in the EFLAGS register.

When the VME flag in control register CR4 is set and the IOPL field in the EFLAGS register is less than 3, two additional flags are activated in the EFLAGS register:

- VIF (virtual interrupt) flag, bit 19 of the EFLAGS register.
- VIP (virtual interrupt pending) flag, bit 20 of the EFLAGS register.

These flags provide the virtual-8086 monitor with more efficient control over handling maskable hardware interrupts that occur during virtual-8086 mode tasks. They also reduce interrupt-handling overhead, by eliminating the need for all IF related operations (such as PUSHF, POPF, CLI, and STI instructions) to trap to the virtual-8086 monitor. The purpose and use of these flags are as follows.

NOTE

The VIF and VIP flags are only available in IA-32 processors that support the virtual mode extensions. These extensions were introduced in the IA-32 architecture with the Pentium processor. When this mechanism is either not available or not enabled, maskable hardware interrupts are handled as class 1 interrupts. Here, if VIF and VIP flags are needed, the virtual-8086 monitor can implement them in software.

Existing 8086 programs commonly set and clear the IF flag in the EFLAGS register to enable and disable maskable hardware interrupts, respectively; for example, to disable interrupts while handling another interrupt or an exception. This practice works well in single task environments, but

can cause problems in multitasking and multiple-processor environments, where it is often desirable to prevent an application program from having direct control over the handling of hardware interrupts. When using earlier IA-32 processors, this problem was often solved by creating a virtual IF flag in software. The IA-32 processors (beginning with the Pentium processor) provide hardware support for this virtual IF flag through the VIF and VIP flags.

The VIF flag is a virtualized version of the IF flag, which an application program running from within a virtual-8086 task can used to control the handling of maskable hardware interrupts. When the VIF flag is enabled, the CLI and STI instructions operate on the VIF flag instead of the IF flag. When an 8086 program executes the CLI instruction, the processor clears the VIF flag to request that the virtual-8086 monitor inhibit maskable hardware interrupts from interrupting program execution; when it executes the STI instruction, the processor sets the VIF flag requesting that the virtual-8086 monitor enable maskable hardware interrupts for the 8086 program. But actually the IF flag, managed by the operating system, always controls whether maskable hardware interrupts are enabled. Also, if under these circumstances an 8086 program tries to read or change the IF flag using the PUSHF or POPF instructions, the processor will change the VIF flag instead, leaving IF unchanged.

The VIP flag provides software a means of recording the existence of a deferred (or pending) maskable hardware interrupt. This flag is read by the processor but never explicitly written by the processor; it can only be written by software.

If the IF flag is set and the VIF and VIP flags are enabled, and the processor receives a maskable hardware interrupt (interrupt vector 0 through 255), the processor performs and the interrupt handler software should perform the following operations:

1. The processor invokes the protected-mode interrupt handler for the interrupt received, as described in the following steps. These steps are almost identical to those described for method 1 interrupt and exception handling in Section 16.3.1.1., *Handling an Interrupt or Exception Through a Protected-Mode Trap or Interrupt Gate*:
   a. Switches to 32-bit protected mode and privilege level 0.
   b. Saves the state of the processor on the privilege-level 0 stack. The states of the EIP, CS, EFLAGS, ESP, SS, ES, DS, FS, and GS registers are saved (see Figure 16-4).
   c. Clears the segment registers.
   d. Clears the VM flag in the EFLAGS register.
   e. Begins executing the selected protected-mode interrupt handler.
2. The recommended action of the protected-mode interrupt handler is to read the VM flag from the EFLAGS image on the stack. If this flag is set, the handler makes a call to the virtual-8086 monitor.
3. The virtual-8086 monitor should read the VIF flag in the EFLAGS register.
   — If the VIF flag is clear, the virtual-8086 monitor sets the VIP flag in the EFLAGS image on the stack to indicate that there is a deferred interrupt pending and returns to the protected-mode handler.
   — If the VIF flag is set, the virtual-8086 monitor can handle the interrupt if it “belongs” to the 8086 program running in the interrupted virtual-8086 task; otherwise, it can call the protected-mode interrupt handler to handle the interrupt.
4. The protected-mode handler executes a return to the program executing in virtual-8086 mode.
5. Upon returning to virtual-8086 mode, the processor continues execution of the 8086 program.

When the 8086 program is ready to receive maskable hardware interrupts, it executes the STI instruction to set the VIF flag (enabling maskable hardware interrupts). Prior to setting the VIF flag,

the processor automatically checks the VIP flag and does one of the following, depending on the state of the flag:

- If the VIP flag is clear (indicating no pending interrupts), the processor sets the VIF flag.
- If the VIP flag is set (indicating a pending interrupt), the processor generates a general-protection exception (#GP).

The recommended action of the protected-mode general-protection exception handler is to then call the virtual-8086 monitor and let it handle the pending interrupt. After handling the pending interrupt, the typical action of the virtual-8086 monitor is to clear the VIP flag and set the VIF flag in the EFLAGS image on the stack, and then execute a return to the virtual-8086 mode. The next time the processor receives a maskable hardware interrupt, it will then handle it as described in steps 1 through 5 earlier in this section.

If the processor finds that both the VIF and VIP flags are set at the beginning of an instruction, it generates a general-protection exception. This action allows the virtual-8086 monitor to handle the pending interrupt for the virtual-8086 mode task for which the VIF flag is enabled. Note that this situation can only occur immediately following execution of a POPF or IRET instruction or upon entering a virtual-8086 mode task through a task switch.

Note that the states of the VIF and VIP flags are not modified in real-address mode or during transitions between real-address and protected modes.

NOTE

The virtual interrupt mechanism described in this section is also available for use in protected mode, see Section 16.4., “Protected-Mode Virtual Interrupts”.

16.3.3. Class 3—Software Interrupt Handling in Virtual-8086 Mode

When the processor receives a software interrupt (an interrupt generated with the INT *n* instruction) while in virtual-8086 mode, it can use any of six different methods to handle the interrupt. The method selected depends on the settings of the VME flag in control register CR4, the IOPL field in the EFLAGS register, and the software interrupt redirection bit map in the TSS. Table 16-2 lists the six methods of handling software interrupts in virtual-8086 mode and the respective settings of the VME flag, IOPL field, and the bits in the interrupt redirection bit map for each method. The table also summarizes the various actions the processor takes for each method.

The VME flag enables the virtual mode extensions for the Pentium and later IA-32 processors. When this flag is clear, the processor responds to interrupts and exceptions in virtual-8086 mode in the same manner as an Intel386 or Intel486 processor does. When this flag is set, the virtual mode extension provides the following enhancements to virtual-8086 mode:

- Speeds up the handling of software-generated interrupts in virtual-8086 mode by allowing the processor to bypass the virtual-8086 monitor and redirect software interrupts back to the interrupt handlers that are part of the currently running 8086 program.
- Supports virtual interrupts for software written to run on the 8086 processor.

The IOPL value interacts with the VME flag and the bits in the interrupt redirection bit map to determine how specific software interrupts should be handled.

The software interrupt redirection bit map (see Figure 16-5) is a 32-byte field in the TSS. This map is located directly below the I/O permission bit map in the TSS. Each bit in the interrupt redirection bit map is mapped to an interrupt vector. Bit 0 in the interrupt redirection bit map (which maps to vector zero in the interrupt table) is located at the I/O base map address in the TSS minus 32 bytes. When a bit in this bit map is set, it indicates that the associated software interrupt (interrupt gener-

ated with an INT *n* instruction) should be handled through the protected-mode IDT and interrupt and exception handlers. When a bit in this bit map is clear, the processor redirects the associated software interrupt back to the interrupt table in the 8086 program (located at linear address 0 in the program's address space).

NOTE

The software interrupt redirection bit map does not affect hardware generated interrupts and exceptions. Hardware generated interrupts and exceptions are always handled by the protected-mode interrupt and exception handlers.

Table 16-2. Software Interrupt Handling Methods While in Virtual-8086 Mode

Method	VME	IOPL	Bit in Redir. Bitmap*	Processor Action
1	0	3	X	**Interrupt directed to a protected-mode interrupt handler:** - Switches to privilege-level 0 stack - Pushes GS, FS, DS and ES onto privilege-level 0 stack - Pushes SS, ESP, EFLAGS, CS and EIP of interrupted task onto privilege-level 0 stack - Clears VM, RF, NT, and TF flags - If serviced through interrupt gate, clears IF flag - Clears GS, FS, DS and ES to 0 - Sets CS and EIP from interrupt gate
2	0	< 3	X	**Interrupt directed to protected-mode general-protection exception (#GP) handler.**
3	1	< 3	1	**Interrupt directed to a protected-mode general-protection exception (#GP) handler; VIF and VIP flag support for handling class 2 maskable hardware interrupts.**
4	1	3	1	**Interrupt directed to protected-mode interrupt handler:** (see method 1 processor action).
5	1	3	0	**Interrupt redirected to 8086 program interrupt handler:** - Pushes EFLAGS - Pushes CS and EIP (lower 16 bits only) - Clears IF flag - Clears TF flag - Loads CS and EIP (lower 16 bits only) from selected entry in the interrupt vector table of the current virtual-8086 task
6	1	< 3	0	**Interrupt redirected to 8086 program interrupt handler; VIF and VIP flag support for handling class 2 maskable hardware interrupts:** - Pushes EFLAGS with IOPL set to 3 and VIF copied to IF - Pushes CS and EIP (lower 16 bits only) - Clears the VIF flag - Clears TF flag - Loads CS and EIP (lower 16 bits only) from selected entry in the interrupt vector table of the current virtual-8086 task

NOTE:

* When set to 0, software interrupt is redirected back to the 8086 program interrupt handler; when set to 1, interrupt is directed to protected-mode handler.

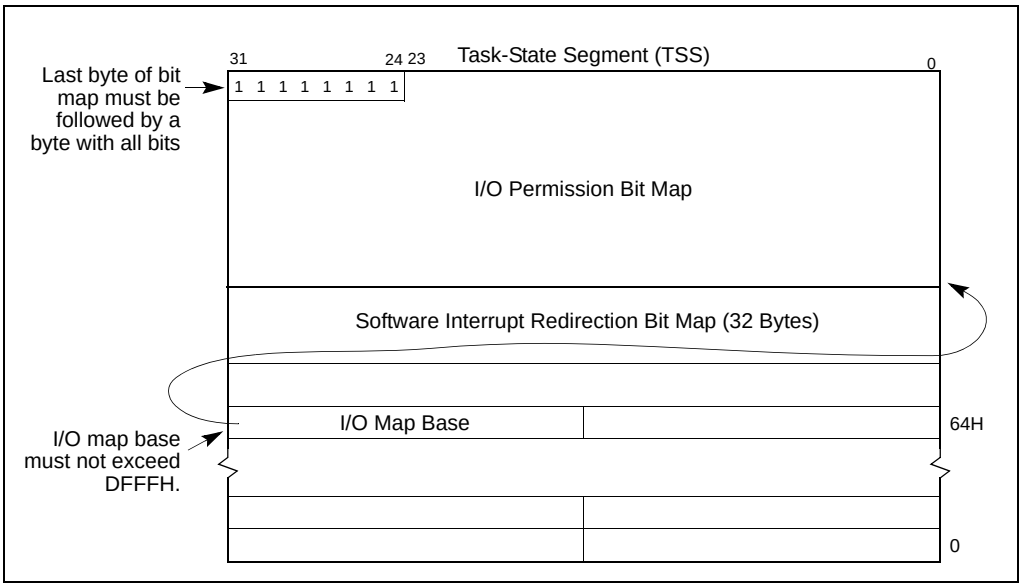

Figure 16-5. Software Interrupt Redirection Bit Map in TSS

Redirecting software interrupts back to the 8086 program potentially speeds up interrupt handling because a switch back and forth between virtual-8086 mode and protected mode is not required. This latter interrupt-handling technique is particularly useful for 8086 operating systems (such as MS-DOS) that use the INT *n* instruction to call operating system procedures.

The CPUID instruction can be used to verify that the virtual mode extension is implemented on the processor. Bit 1 of the feature flags register (EDX) indicates the availability of the virtual mode extension (see "CPUID—CPU Identification" in Chapter 3 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 2*).

The following sections describe the six methods (or mechanisms) for handling software interrupts in virtual-8086 mode. See Section , *In Section 16.3.2, IA-32 Intel Architecture Software Developer's Manual, Volume 3; text in a numbered list has been edited to remove an error. The updated text is provided below (in context). Not all of the section is included.*, for a description of the use of the VIF and VIP flags in the EFLAGS register for handling maskable hard

*** *** ***

17. IA32_MTRR_DEF_TYPE MSR Definition Corrected

In Section 10.11.2.1, item “Type field, bits 0 through 7”, *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*; the first paragraph has been corrected. The paragraph is shown below in context with a change bar marking the affected area.

10.11.2.1. IA32_MTRR_DEF_TYPE MSR

The IA32_MTRR_DEF_TYPE MSR (named MTRRdefType MSR for the P6 family processors) sets the default properties of the regions of physical memory that are not encompassed by MTRRs. The functions of the flags and field in this register are as follows:

Type field, bits 0 through 7

Indicates the default memory type used for those physical memory address ranges that do not have a memory type specified for them by an MTRR (see Table 10-8 for the encoding of this field). The legal values for this field are 0, 1, 4, 5, and 6. All other values result in a general-protection exception (#GP) being generated.

Intel recommends the use of the UC (uncached) memory type for all physical memory addresses where memory does not exist. To assign the UC type to nonexistent memory locations, it can either be specified as the default type in the Type field or be explicitly assigned with the fixed and variable MTRRs.

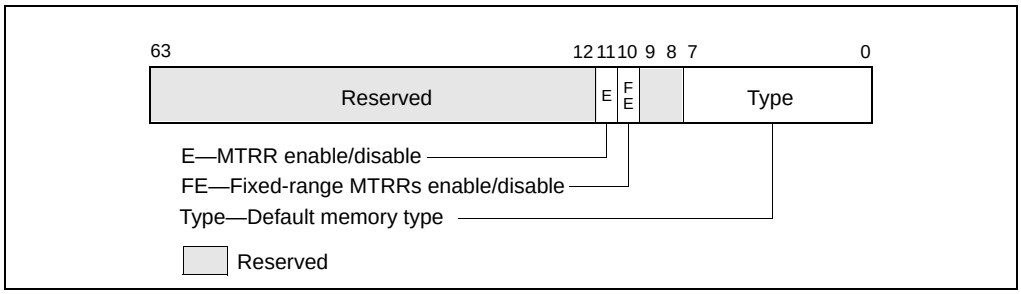

Figure 10-5. IA32_MTRR_DEF_TYPE MSR

*** *** ***

18. Correction of Error in EFLAGS Treatment in Virtual-8086 Mode

For the INT n/INTO/INT 3—Call to Interrupt Procedure section, Chapter 3, *IA-32 Intel Architecture Software Developer's Manual, Volume 2A*; the Operation sub-section has been corrected. This section is shown below with a change bar marking the affected line.

Operation

The following operational description applies not only to the INT *n* and INTO instructions, but also to external interrupts and exceptions.

```
IF PE = 0
    THEN
        GOTO REAL-ADDRESS-MODE;
    ELSE (* PE = 1 *)
        IF (VM = 1 AND IOPL < 3 AND INT n)
            THEN
                #GP(0);
            ELSE (* protected mode or virtual-8086 mode interrupt *)
                GOTO PROTECTED-MODE;
        FI;
FI;

REAL-ADDRESS-MODE:
    IF ((DEST * 4) + 3) is not within IDT limit THEN #GP; FI;
    IF stack not large enough for a 6-byte return information THEN #SS; FI;
    Push (EFLAGS[15:0]);
    IF ← 0; (* Clear interrupt flag *)
    TF ← 0; (* Clear trap flag *)
    AC ← 0; (*Clear AC flag*)
    Push(CS);
    Push(IP);
    (* No error codes are pushed *)
    CS ← IDT(Descriptor (vector_number * 4), selector));
    EIP ← IDT(Descriptor (vector_number * 4), offset)); (* 16 bit offset AND 0000FFFFH *)
END;

PROTECTED-MODE:
    IF ((DEST * 8) + 7) is not within IDT limits
        OR selected IDT descriptor is not an interrupt-, trap-, or task-gate type
            THEN #GP((DEST * 8) + 2 + EXT);
            (* EXT is bit 0 in error code *)
    FI;
    IF software interrupt (* generated by INT n, INT 3, or INTO *)
        THEN
            IF gate descriptor DPL < CPL
                THEN #GP((vector_number * 8) + 2 );
                (* PE = 1, DPL<CPL, software interrupt *)
            FI;
    FI;
    IF gate not present THEN #NP((vector_number * 8) + 2 + EXT); FI;
    IF task gate (* specified in the selected interrupt table descriptor *)
        THEN GOTO TASK-GATE;
        ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
```

```
    FI;
END;

TASK-GATE: (* PE = 1, task gate *)
    Read segment selector in task gate (IDT descriptor);
        IF local/global bit is set to local
            OR index not within GDT limits
                THEN #GP(TSS selector);
        FI;
        Access TSS descriptor in GDT;
        IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
            THEN #GP(TSS selector);
        FI;
        IF TSS not present
            THEN #NP(TSS selector);
        FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF interrupt caused by fault with error code
        THEN
            IF stack limit does not allow push of error code
                THEN #SS(0);
            FI;
            Push(error code);
    FI;
    IF EIP not within code segment limit
        THEN #GP(0);
    FI;
END;
TRAP-OR-INTERRUPT-GATE
    Read segment selector for trap or interrupt gate (IDT descriptor);
    IF segment selector for code segment is null
        THEN #GP(0H + EXT); (* null selector with EXT flag set *)
    FI;
    IF segment selector is not within its descriptor table limits
        THEN #GP(selector + EXT);
    FI;
    Read trap or interrupt handler descriptor;
    IF descriptor does not indicate a code segment
        OR code segment descriptor DPL > CPL
            THEN #GP(selector + EXT);
    FI;
    IF trap or interrupt gate segment is not present,
        THEN #NP(selector + EXT);
    FI;
    IF code segment is non-conforming AND DPL < CPL
        THEN IF VM=0
            THEN
                GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
                (* PE = 1, interrupt or trap gate, nonconforming *)
                (* code segment, DPL<CPL, VM = 0 *)
            ELSE (* VM = 1 *)
                IF code segment DPL ≠ 0 THEN #GP(new code segment selector); FI;
                GOTO INTERRUPT-FROM-VIRTUAL-8086-MODE;
                (* PE = 1, interrupt or trap gate, DPL<CPL, VM = 1 *)
        FI;
```

```
        ELSE (* PE = 1, interrupt or trap gate, DPL ≥ CPL *)
            IF VM = 1 THEN #GP(new code segment selector); FI;
            IF code segment is conforming OR code segment DPL = CPL
                THEN
                    GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
                ELSE
                    #GP(CodeSegmentSelector + EXT);
                    (* PE = 1, interrupt or trap gate, nonconforming *)
                    (* code segment, DPL>CPL *)
            FI;
    FI;
END;

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
            IF (TSSstackAddress + 7) > TSS limit
                THEN #TS(current TSS selector); FI;
            NewSS ← TSSstackAddress + 4;
            NewESP ← stack address;
        ELSE (* TSS is 16-bit *)
            TSSstackAddress ← (new code segment DPL * 4) + 2
            IF (TSSstackAddress + 4) > TSS limit
                THEN #TS(current TSS selector); FI;
            NewESP ← TSSstackAddress;
            NewSS ← TSSstackAddress + 2;
    FI;
    IF segment selector is null THEN #TS(EXT); FI;
    IF segment selector index is not within its descriptor table limits
        OR segment selector's RPL ≠ DPL of code segment,
            THEN #TS(SS selector + EXT);
    FI;
Read segment descriptor for stack segment in GDT or LDT;
    IF stack segment DPL ≠ DPL of code segment,
        OR stack segment does not indicate writable data segment,
            THEN #TS(SS selector + EXT);
    FI;
    IF stack segment not present THEN #SS(SS selector+EXT); FI;
    IF 32-bit gate
        THEN
            IF new stack does not have room for 24 bytes (error code pushed)
                OR 20 bytes (no error code pushed)
                    THEN #SS(segment selector + EXT);
            FI;
        ELSE (* 16-bit gate *)
            IF new stack does not have room for 12 bytes (error code pushed)
                OR 10 bytes (no error code pushed);
                    THEN #SS(segment selector + EXT);
            FI;
    FI;
    IF instruction pointer is not within code segment limits THEN #GP(0); FI;
    SS:ESP ← TSS(NewSS:NewESP) (* segment descriptor information also loaded *)
```

```
    IF 32-bit gate
        THEN
            CS:EIP ← Gate(CS:EIP); (* segment descriptor information also loaded *)
        ELSE (* 16-bit gate *)
            CS:IP ← Gate(CS:IP); (* segment descriptor information also loaded *)
    FI;
    IF 32-bit gate
        THEN
            Push(far pointer to old stack); (* old SS and ESP, 3 words padded to 4 *);
            Push(EFLAGS);
            Push(far pointer to return instruction); (* old CS and EIP, 3 words padded to 4*);
            Push(ErrorCode); (* if needed, 4 bytes *)
        ELSE(* 16-bit gate *)
            Push(far pointer to old stack); (* old SS and SP, 2 words *);
            Push(EFLAGS(15..0]);
            Push(far pointer to return instruction); (* old CS and IP, 2 words *);
            Push(ErrorCode); (* if needed, 2 bytes *)
    FI;
    CPL ← CodeSegmentDescriptor(DPL);
    CS(RPL) ← CPL;
    IF interrupt gate
        THEN IF ← 0 (*interrupt flag set to 0: disabled*);
    FI;
    TF ← 0;
    VM ← 0;
    RF ← 0;
    NT ← 0;
END;
INTERRUPT-FROM-VIRTUAL-8086-MODE:
    (* Check segment selector and descriptor for privilege level 0 stack in current TSS *)
    IF current TSS is 32-bit TSS
        THEN
            TSSstackAddress ← (new code segment DPL * 8) + 4
            IF (TSSstackAddress + 7) > TSS limit
                THEN #TS(current TSS selector); FI;
            NewSS ← TSSstackAddress + 4;
            NewESP ← stack address;
        ELSE (* TSS is 16-bit *)
            TSSstackAddress ← (new code segment DPL * 4) + 2
            IF (TSSstackAddress + 4) > TSS limit
                THEN #TS(current TSS selector); FI;
            NewESP ← TSSstackAddress;
            NewSS ← TSSstackAddress + 2;
    FI;
        IF segment selector is null THEN #TS(EXT); FI;
        IF segment selector index is not within its descriptor table limits
            OR segment selector's RPL ≠ DPL of code segment,
                THEN #TS(SS selector + EXT);
        FI;
    Access segment descriptor for stack segment in GDT or LDT;
    IF stack segment DPL ≠ DPL of code segment,
        OR stack segment does not indicate writable data segment,
            THEN #TS(SS selector + EXT);
    FI;
    IF stack segment not present
```

```
            THEN #SS(SS selector+EXT);
        FI;
        IF 32-bit gate
            THEN
                IF new stack does not have room for 40 bytes (error code pushed)
                    OR 36 bytes (no error code pushed);
                        THEN #SS(segment selector + EXT);
                FI;
            ELSE (* 16-bit gate *)
                IF new stack does not have room for 20 bytes (error code pushed)
                    OR 18 bytes (no error code pushed);
                        THEN #SS(segment selector + EXT);
                FI;
        FI;
        IF instruction pointer is not within code segment limits
            THEN #GP(0);
        FI;
        tempEFLAGS ← EFLAGS;
        VM ← 0;
        TF ← 0;
        RF ← 0;
        NT ← 0;
        IF service through interrupt gate
            THEN IF = 0;
        FI;
        TempSS ← SS;
        TempESP ← ESP;
        SS:ESP ← TSS(SS0:ESP0); (* Change to level 0 stack segment *)
        (* Following pushes are 16 bits for 16-bit gate and 32 bits for 32-bit gates *)
        (* Segment selector pushes in 32-bit mode are padded to two words *)
        Push(GS);
        Push(FS);
        Push(DS);
        Push(ES);
        Push(TempSS);
        Push(TempESP);
        Push(TempEFlags);
        Push(CS);
        Push(EIP);
        GS ← 0; (*segment registers nullified, invalid in protected mode *)
        FS ← 0;
        DS ← 0;
        ES ← 0;
        CS ← Gate(CS);
        IF OperandSize = 32
            THEN
                EIP ← Gate(instruction pointer);
            ELSE (* OperandSize is 16 *)
                EIP ← Gate(instruction pointer) AND 0000FFFFH;
        FI;
        (* Starts execution of new routine in Protected Mode *)
    END;

    INTRA-PRIVILEGE-LEVEL-INTERRUPT:
        (* PE=1, DPL = CPL or conforming segment *)
```

```
    IF 32-bit gate
        THEN
            IF current stack does not have room for 16 bytes (error code pushed)
                OR 12 bytes (no error code pushed); THEN #SS(0);
            FI;
        ELSE (* 16-bit gate *)
            IF current stack does not have room for 8 bytes (error code pushed)
                OR 6 bytes (no error code pushed); THEN #SS(0);
            FI;
    FI;
    IF instruction pointer not within code segment limit
        THEN #GP(0);
    FI;
    IF 32-bit gate
        THEN
            Push (EFLAGS);
            Push (far pointer to return instruction); (* 3 words padded to 4 *)
            CS:EIP ← Gate(CS:EIP); (* segment descriptor information also loaded *)
            Push (ErrorCode); (* if any *)
        ELSE (* 16-bit gate *)
            Push (FLAGS);
            Push (far pointer to return location); (* 2 words *)
            CS:IP ← Gate(CS:IP); (* segment descriptor information also loaded *)
            Push (ErrorCode); (* if any *)
    FI;
    CS(RPL) ← CPL;
    IF interrupt gate
        THEN IF ← 0; (*interrupt flag set to 0: disabled*)
    FI;
    TF ← 0;
    NT ← 0;
    VM ← 0;
    RF ← 0;
END;
```

*** *** ***

19. Table B-3 Correction

For Table B-3, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*; an error in the DEBUGCTLMSR entry has been corrected. The corrected cells are shown below.

Table B-3. MSRs in the P6 Family Processors

Register Address		Register Name	Bit Description
Hex	Dec		
... ...			
1D9H	473	DEBUGCTLMSR	
		0	Enable/Disable Last Branch Records
		1	Branch Trap Flag
		2	Performance Monitoring/Break Point Pins
		3	Performance Monitoring/Break Point Pins
		4	Performance Monitoring/Break Point Pins
		5	Performance Monitoring/Break Point Pins
		6	Enable/Disable Execution Trace Messages
		31:7	Reserved

*** *** ***

20. Updates to Appendix E

For Appendix E, IA-32 Intel Architecture Software Developer's Manual, Volume 3; some architectural information has been removed due to redundancy. The updated appendix is below.

APPENDIX E
INTERPRETING MACHINE-CHECK ERROR CODES

Encoding of the model-specific and other information fields is different for 06H and 0FH processor families. The differences are documented in the following sections.

E.1. INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY 06H MACHINE ERROR CODES FOR MACHINE CHECK

Table E.1. provides information for interpreting additional family 06H model-specific fields for external bus errors. These errors are reported in the IA32_MCi_STATUS MSRs. They are reported (architecturally) as compound errors with a general form of *0000 1PPT RRRR IILL* in the MCA error code field. See Chapter 14 for information on the interpretation of compound error codes.

Table E-1. Incremental Decoding Information: Processor Family 06H Machine Error Codes For Machine Check

Type	Bit No.	Bit Function	Bit Description
MCA error codes[a]	0-15		
Model specific errors	16-18	Reserved	Reserved
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

Table E-1. Incremental Decoding Information: Processor Family 06H Machine Error Codes For Machine Check (Continued)

Type	Bit No.	Bit Function	Bit Description
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
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-1. Incremental Decoding Information: Processor Family 06H Machine Error Codes For Machine Check (Continued)

Type	Bit No.	Bit Function	Bit Description
	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.
Status register validity indicators[1]	57-63		

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

E.2. INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY 0FH MACHINE ERROR CODES FOR MACHINE CHECK

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

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

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

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

Table E-3 provides information on interpreting additional family 07H, model specific fields for memory hierarchy errors. These errors are reported in one of the IA32_MCi_STATUS MSRs. These errors are reported, architecturally, as compound errors with a general form of *0000 0001 RRRR TTLL* in the MCA error code field. See Chapter 14 for how to interpret the compound error code.

Table E-3. Decoding Family 07H Machine Check Codes for Memory Hierarchy Errors

Type	Bit No.	Bit Function	Bit Description
MCA error codes[a]	0-15		
Model specific error codes	16-17	Tag Error Code	Contains the tag error code for this machine check error: 00 = No error detected 01 = Parity error on tag miss with a clean line 10 = Parity error/multiple tag match on tag hit 11 = Parity error/multiple tag match on tag miss
	18-19	Data Error Code	Contains the data error code for this machine check error: 00 = No error detected 01 = Single bit error 10 = Double bit error on a clean line 11 = Double bit error on a modified line
	20	L3 Error	This bit is set if the machine check error originated in the L3 (it can be ignored for invalid PIC request errors): 1 = L3 error 0 = L2 error
	21	Invalid PIC Request	Indicates error due to invalid PIC request (access was made to PIC space with WB memory): 1 = Invalid PIC request error 0 = No invalid PIC request error
	22-31	Reserved	Reserved
Other Information	32-39	8-bit Error Count	Holds a count of the number of errors since reset. The counter begins at 0 for the first error and saturates at a count of 254.
	40-56	Reserved	Reserved
Status register validity indicators[1]	57-63		

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

21. MSR_PLATFORM_BRV Information Added

For Table B-1, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*.; documentation for MSR_PLATFORM_BRV has been added. The added cells are shown below.

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

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
... ...					
1A1H	417	MSR_PLATFORM_BRV	3	Shared	**Platform Feature Requirements.** (R)
		17:0			**Reserved.**
		18			**PLATFORM Requirements**: When set to 1, indicates the processor has specific platform requirements. The details of the platform requirements are listed in the respective data sheets of the processor.
		63:19			**Reserved.**

*** *** ***

22. Section On Support Processor Added

In Chapter 15, IA-32 Intel Architecture Software Developer's Manual, Volume 3; a section has been added to cover last branch, interrupt, and exception recording for the Pentium M processor.

15.6. LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (PENTIUM M PROCESSORS)

Like the Pentium 4 and Intel Xeon processor family, Pentium M processors provide last branch interrupt and exception recording. The capability operates almost identically to that found in Pentium 4 and Intel Xeon processors. There are differences in the shape of the stack and in some MSR names and locations. Note the following:

- MSR_DEBUGCTLB MSR — Enables debug trace interrupt, debug trace store, trace messages enable, performance monitoring breakpoint flags, single stepping on branches, and last branch. For Pentium M processors, this MSR is located at register address 01D9H. See Figure 15-6 and the entries below for a description of the flags.
  - — **LBR (last branch/interrupt/exception) flag (bit 0) —** When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the "Last Branch Record (LBR) Stack" bullet below.
  - — **BTF (single-step on branches) flag (bit 1) —** When set, the processor treats the TF flag in the EFLAGS register as a "single-step on branches" flag rather than a "single-step on instructions" flag. This mechanism allows single-stepping the processor on taken branches, interrupts, and exceptions. See Section 15.5.4., "Single-Stepping on Branches, Exceptions, and Interrupts" for more information about the BTF flag.
  - — **PB*i* (performance monitoring/breakpoint pins) flags (bits 5-2) —** When these flags are set, the performance monitoring/breakpoint pins on the processor (BP0#, BP1#, BP2#, and BP3#) report breakpoint matches in the corresponding breakpoint-address registers (DR0 through DR3). The processor asserts then deasserts the corresponding BP*i*# pin when a breakpoint match occurs. When a PB*i* flag is clear, the performance monitoring/breakpoint pins report performance events. Processor execution is not affected by reporting performance events.
  - — **TR (trace message enable) flag (bit 6) —** When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception, it sends the branch record out on the system bus as a branch trace message (BTM). See Section 15.5.5., "Branch Trace Messages" for more information about the TR flag.
  - — **BTS (branch trace store) flag (bit 7) —** When set, enables the BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 15.10.5., "DS Save Area".
  - — **BTINT (branch trace interrupt) flag (bits 8) —** When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 15.5.7., "Branch Trace Store (BTS)" for a description of this mechanism.

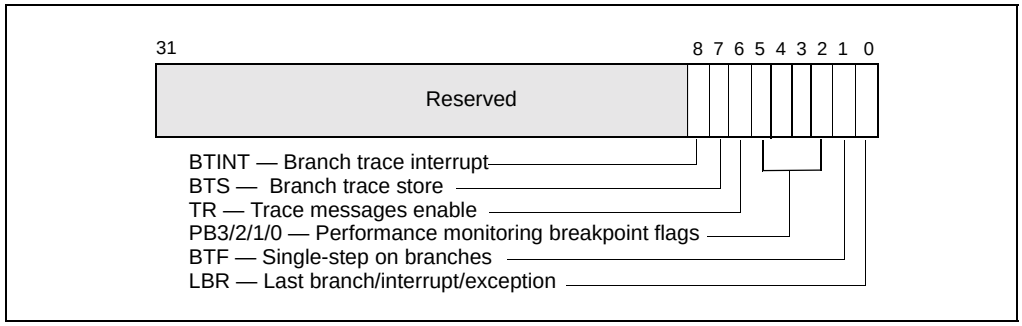

Figure 15-6. MSR_DEBUGCTLB MSR for Pentium M Processors

- Debug store (DS) feature flag (bit 21), returned by the CPUID instruction — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer. See also: Section 15.5.7.
- Last Branch Record (LBR) Stack — The LBR stack consists of 8 MSRs (MSR_LASTBRANCH_0 through MSR_LASTBRANCH_7); bits 31-0 hold the ‘from’ address, bits 63-32 hold the ‘to’ address. For Pentium M Processors, these pairs are located at register addresses 040H-047H. See Figure 15-7
- Last Branch Record Top-of-Stack (TOS) Pointer — The TOS Pointer MSR contains a 3-bit pointer (bits 2-0) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. For Pentium M Processors, this MSR is located at register address 01C9H.
- For compatibility, the Pentium M processor provides two 32 bit MSRs (the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs) that duplicate the functions of the LastExceptionToIP and LastExceptionFromIP MSRs found in P6 family processors.
-

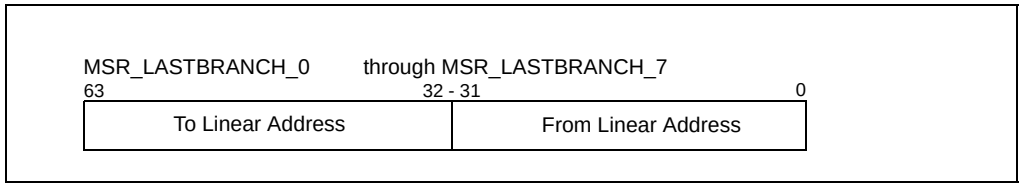

Figure 15-7. LBR Branch Record Layout for the Pentium M Processor

For more detail on these capabilities, see Section 15.5., “Last Branch, Interrupt, and Exception Recording (Pentium 4 and Intel Xeon Processors)” and Section B.2., “MSRs In the Pentium M Processor”.

23. MSR Data for Pentium M Processor Has Been Updated

In Section B2, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*; new MSR coverage has been added for the Pentium M processor. Updated table cells are shown below.

Table E-2. MSRs in Pentium M Processors

Register Address		Register Name	Bit Description
Hex	**Dec**		
... ...			
40H	64	MSR_LASTBRANCH_0	**Last Branch Record 0.** (R/W) One of 8 last branch record registers on the last branch record stack: bits 31-0 hold the 'from' address and bits 63-32 hold the 'to' address. See also: • Last Branch Record Stack TOS at 1C9H • Section 15.6., "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)"
41H	65	MSR_LASTBRANCH_1	**Last Branch Record 1.** (R/W) See description of MSR_LASTBRANCH_0.
42H	66	MSR_LASTBRANCH_2	**Last Branch Record 2.** (R/W) See description of MSR_LASTBRANCH_0.
43H	67	MSR_LASTBRANCH_3	**Last Branch Record 3.** (R/W) See description of MSR_LASTBRANCH_0.
44H	68	MSR_LASTBRANCH_4	**Last Branch Record 4.** (R/W) See description of MSR_LASTBRANCH_0.
45H	69	MSR_LASTBRANCH_5	**Last Branch Record 5.** (R/W) See description of MSR_LASTBRANCH_0.
46H	70	MSR_LASTBRANCH_6	**Last Branch Record 6.** (R/W) See description of MSR_LASTBRANCH_0.
47H	71	MSR_LASTBRANCH_7	**Last Branch Record 7.** (R/W) See description of MSR_LASTBRANCH_0.
... ...			
1C9H	457	MSR_LASTBRANCH_TOS	**Last Branch Record Stack TOS. (R)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See also: • MSR_LASTBRANCH_0 (at 40H) • Section 15.6., "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)"
1D9H	473	IA32_DEBUGCTL	**Debug Control.** (R/W) Controls how several debug features are used. Bit definitions are discussed in the referenced section. See Section 15.6., "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)".
1DDH	477	MSR_LER_TO_LIP	**Last Exception Record To Linear IP.** (R) This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 15.6., "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)" and Section 15.7.2., "Last Branch and Last Exception MSRs (P6 Family Processors)".

Table E-2. MSRs in Pentium M Processors (Continued)

Register Address		Register Name	Bit Description
Hex	Dec		
1DEH	478	MSR_LER_FROM_LIP	**Last Exception Record From Linear IP.** (R) Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 15.6., “Last Branch, Interrupt, and Exception Recording (Pentium M Processors)” and Section 15.7.2., “Last Branch and Last Exception MSRs (P6 Family Processors)”.
... ...			
600H	1536	IA32_DS_AREA	**DS Save Area.** (R/W) Points to the DS buffer management area, which is used to manage the BTS and PEBS buffers. See Section 15.10.4., “Debug Store (DS) Mechanism”.
		31:0	**DS Buffer Management Area.** Linear address of the first byte of the DS buffer management area.
		63:32	**Reserved.**

*** *** ***

24. Cache and TLB Descriptor Table Updated

In the CPUID section, Chapter 3, *IA-32 Intel Architecture Software Developer's Manual, Volume 2A*; Table 3-13 has been updated. Updated cells are shown below.

Table 3-13. Encoding of Cache and TLB Descriptors

Descriptor Value	Cache or TLB Description
... ...	
22H	3rd-level cache: 512K Bytes, 4-way set associative, 64 byte line size, 2 lines per sector
23H	3rd-level cache: 1M Bytes, 8-way set associative, 64 byte line size, 2 lines per sector
25H	3rd-level cache: 2M Bytes, 8-way set associative, 64 byte line size, 2 lines per sector
29H	3rd-level cache: 4M Bytes, 8-way set associative, 64 byte line size, 2 lines per sector
... ...	
78H	2nd-level cache: 1M Byte, 4-way set associative, 64byte line size
79H	2nd-level cache: 128KB, 8-way set associative, 64 byte line size, 2 lines per sector
7AH	2nd-level cache: 256KB, 8-way set associative, 64 byte line size, 2 lines per sector
7BH	2nd-level cache: 512KB, 8-way set associative, 64 byte line size, 2 lines per sector
7CH	2nd-level cache: 1MB, 8-way set associative, 64 byte line size, 2 lines per sector
... ...	
7FH	2nd-level cache: 512KB, 2-way set associative, 64-byte line size
... ...	
F0H	64-Byte Prefetching
F1H	128-Byte Prefetching

*** *** ***

25. Brand String Table Updated

In the CPUID section, Chapter 3, *IA-32 Intel Architecture Software Developer's Manual, Volume 2A*; Table 3-15 has been updated. Updated cells are shown below.

Table 3-15. Mapping of Brand Indices and IA-32 Processor Brand Strings

Brand Index	Brand String
... ...	
11H	Mobile Genuine Intel(R) processor
12H	Intel(R) Celeron(R) M processor
... ...	
15H	Mobile Genuine Intel(R) processor
... ...	
17H	Mobile Intel(R) Celeron(R) processor†
18H – 0FFH	RESERVED

*** *** ***

26. Pentium M Processor Sections Updated

Chapter 2, *IA-32 Intel Architecture Software Developer's Manual, Volume 1*; Sections 2.1.9 and 2.2.3 have been updated. Both updated sections are shown below.

.....

2.1.9. The Intel® Pentium® M Processor (2003-2004)

The Intel Pentium M processor family is a high performance, low power mobile processor family with microarchitectural enhancements over previous generations of Intel mobile processors. This family is designed for extending battery life and seamless integration with platform innovations that enable new usage models (such as extended mobility, ultra thin form-factors, and integrated wireless networking).

... ... omitted text ....

2.2.3. The Intel Pentium M Processor Family

The Intel Pentium M processor family is designed for low power consumption. It's enhanced microarchitecture includes the following features:

- Support for Intel Architecture with Dynamic Execution
- A high performance, low-power core manufactured using Intel's advanced process technology with copper interconnect
- On-die, primary 32-kbyte instruction cache and 32-kbyte write-back data cache
- On-die, second-level cache (up to 2-MByte) with Advanced Transfer Cache Architecture
- Advanced Branch Prediction and Data Prefetch Logic
- Support for MMX™ Technology, Streaming SIMD instructions, and the SSE2 instruction set
- A 400 MHz, Source-Synchronous Processor System Bus
- Advanced power management using Enhanced Intel® SpeedStep® Technology

These features are designed to facilitate seamless integration with additional platform sub-systems (battery enhancements, efficient wireless networking, and more advanced graphics technologies).

*** *** ***

27. Name Change for IA32_DEBUGCTL

IA-32 Intel Architecture Software Developer's Manual, Volume 3, the IA32_DEBUGCTL was previously classified as architectural MSR. Variants of the MSR have been renamed as follows:.

- The Pentium 4 processor DEBUGCTL MSR is now MSR_DEBUGCTLA.
- The Pentium M processor DEBUGCTL MSR is now MSR_DEBUGCTLB.
