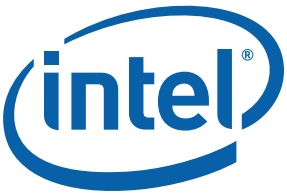

# Intel® 64 and IA-32 Architectures Software Developer's Manual

## Documentation Changes

***May 2007***

**Notice:** The Intel® 64 and IA-32 architectures 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: 252046-020

# Contents

# Revision History

| Version | Description | Date |
|---|---|---|
| -001 | • Initial Release | November 2002 |
| -002 | • Added 1-10 Documentation Changes.<br>• 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.<br>• Removed Documentation Change #6 - References to bits Gen and Len Deleted.<br>• Removed Documentation Change #4 - VIF Information Added to CLI Discussion. | February 2003 |
| -004 | • Removed Documentation changes 1-17.<br>• Added Documentation changes 1-24. | June 2003 |
| -005 | • Removed Documentation Changes 1-24.<br>• Added Documentation Changes 1-15. | September 2003 |
| -006 | • Added Documentation Changes 16- 34. | November 2003 |
| -007 | • Updated Documentation changes 14, 16, 17, and 28.<br>• Added Documentation Changes 35-45. | January 2004 |
| -008 | • Removed Documentation Changes 1-45.<br>• Added Documentation Changes 1-5. | March 2004 |
| -009 | • Added Documentation Changes 7-27. | May 2004 |
| -010 | • Removed Documentation Changes 1-27.<br>• Added Documentation Changes 1. | August 2004 |
| -011 | • Added Documentation Changes 2-28. | November 2004 |
| -012 | • Removed Documentation Changes 1-28.<br>• Added Documentation Changes 1-16. | March 2005 |
| -013 | • Updated title.<br>• There are no Documentation Changes for this revision of the document. | July 2005 |
| -014 | • Added Documentation Changes 1-21. | September 2005 |
| -015 | • Removed Documentation Changes 1-21.<br>• Added Documentation Changes 1-20. | March 9, 2006 |
| -016 | • Added Documentation changes 21-23. | March 27, 2006 |
| -017 | • Removed Documentation Changes 1-23.<br>• Added Documentation Changes 1-36. | September 2006 |
| -018 | • Added Documentation Changes 37-42. | October 2006 |
| -019 | • Removed Documentation Changes 1-42.<br>• Added Documentation Changes 1-19. | March 2007 |
| -020 | • Removed Documentation Change 3 (updated iteration incorporated in new changes).<br>• Added Documentation Changes 19-26. | May 2007 |

# 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 |
|---|---|
| *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture* | 253665 |
| *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M* | 253666 |
| *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z* | 253667 |
| *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide*. | 253668 |
| *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide* | 253669 |

## Nomenclature

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

# Summary Table of Changes

The following table indicates documentation changes which apply to the Intel® 64 and IA-32 architectures. 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 | APIC ID reference corrected |
| 2 | VMPTRST summary table correction |
| 3 | Blocks of pseudocode updated |
| 4 | Material covering handling of VM Exit during Virtual-NMI injection corrected |
| 5 | More information about R8-15 & XMM8-15 transitions |
| 6 | Figure 8-6 corrected |
| 7 | Note added to section on APIC timer |
| 8 | Coverage of PEBS updated |
| 9 | Missing exception added for MFENCE |
| 10 | IA32_MCG_STATUS information added |
| 11 | Introduction section for CPUID updated |
| 12 | Update to CPUID documentation on deterministic cache parameters leaf |
| 13 | Updated pseudocode in VMCALL description |
| 14 | IA32_MCi_STATUS figure corrected |
| 15 | IA32_MCi_STATUS flag description updated |
| 16 | Instruction summaries fixed for MOVD/MOVQ, PMOVMSKB, PINSRW, PEXTRW |
| 17 | Correction to microcode update documentation |
| 18 | PSHUFB compiler intrinsic fixed |
| 19 | RDMSR/RDPMC/RDTSC/WRMSR descriptions updated |
| 20 | Location data corrected |
| 21 | LOOP/LOOPcc description updated |
| 22 | MOV CR and MOV DR sections updated |
| 23 | IRET/IRETD information updated |
| 24 | Table 3-1 updated |
| 25 | MONITOR/MWAIT sections updated |
| 26 | Note on VMX added to microcode update information |

# Documentation Changes

## 1. APIC ID reference corrected

In Section 7.5.5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, an APIC ID reference has been corrected.

---

### 7.5.5 Identifying Logical Processors in an MP System

After the BIOS has completed the MP initialization protocol, each logical processor can be uniquely identified by its local APIC ID. Software can access these APIC IDs in either of the following ways:

- **Read APIC ID for a local APIC** — Code running on a logical processor can execute a MOV instruction to read the processor's local APIC ID register (see Section 8.4.6, "Local APIC ID"). This is the ID to use for directing physical destination mode interrupts to the processor.
- **Read ACPI or MP table** — As part of the MP initialization protocol, the BIOS creates an ACPI table and an MP table. These tables are defined in the Multiprocessor Specification Version 1.4 and provide software with a list of the processors in the system and their local APIC IDs. The format of the ACPI table is derived from the ACPI specification, which is an industry standard power management and platform configuration specification for MP systems.
- **Read Initial APIC ID** — An APIC ID is assigned to a logical processor during power up and is called the initial APIC ID. This is the APIC ID reported by CPUID.1:EBX[31:24] and may be different from the current value read from the local APIC. Use the initial APIC ID to determine the topological relationship between logical processors.

  Bits in the initial APIC ID can be interpreted using several bit masks. Each bit mask can be used to extract an identifier to represent a hierarchical level of the multi-threading resource topology in an MP system (See Section 7.10.1, "Hierarchical Mapping of Shared Resources"). The initial APIC ID may consist of up to four bit-fields. In a non-clustered MP system, the field consists of up to three bit fields.

*... ... ... more text here... ...*

## 2. VMPTRST summary table correction

In Section "VMPTRST—Store Pointer to Virtual-Machine Control Structure" in Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, the summary table has been corrected. See the corrected cells below.

---

### VMPTRST—Store Pointer to Virtual-Machine Control Structure

| Opcode | Instruction | Description |
|---|---|---|
| 0F C7 /7 | VMPTRST m64 | Stores the current VMCS pointer into memory. |

*... ... more text here ... ... ...*

## 3. Blocks of pseudocode updated

In Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, some of the pseudocode has been updated. There are multiple sections, identified by the reproduced code blocks below.

---

*Three blocks of pseudocode were corrected in the "PCMPEQB/PCMPEQW/PCMPEQD—Compare Packed Data for Equal" section. Corrected blocks follow.*

… … …

PCMPEQB instruction with 128-bit operands:

```
IF DEST[7:0] = SRC[7:0]
    THEN DEST[7:0) ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 15th bytes in DEST and SRC *)
IF DEST[127:120] = SRC[127:120]
    THEN DEST[127:120] ← FFH;
    ELSE DEST[127:120] ← 0; FI;
```

… … …

PCMPEQW instruction with 128-bit operands:

```
IF DEST[15:0] = SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd through 7th words in DEST and SRC *)
IF DEST[127:112] = SRC[127:112]
    THEN DEST[127:112] ← FFFFH;
    ELSE DEST[127:112] ← 0; FI;
```

.. … …

PCMPEQD instruction with 128-bit operands:

```
IF DEST[31:0] = SRC[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
(* Continue comparison of 2nd and 3rd doublewords in DEST and SRC *)
IF DEST[127:96] = SRC[127:96]
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 0; FI;
```

… … …

---

*Three blocks of pseudocode were corrected in the "PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than" section. Corrected blocks follow.*

.. … …

PCMPGTB instruction with 128-bit operands:

```
IF DEST[7:0] > SRC[7:0]
    THEN DEST[7:0) ← FFH;
    ELSE DEST[7:0] ← 0; FI;
```

```
    (* Continue comparison of 2nd through 15th bytes in DEST and SRC *)
    IF DEST[127:120] > SRC[127:120]
        THEN DEST[127:120] ← FFH;
        ELSE DEST[127:120] ← 0; FI;

… … …

PCMPGTW instruction with 128-bit operands:
    IF DEST[15:0] > SRC[15:0]
        THEN DEST[15:0] ← FFFFH;
        ELSE DEST[15:0] ← 0; FI;
    (* Continue comparison of 2nd through 7th words in DEST and SRC *)
    IF DEST[63:48] > SRC[127:112]
        THEN DEST[127:112] ← FFFFH;
        ELSE DEST[127:112] ← 0; FI;

… … …

PCMPGTD instruction with 128-bit operands:
    IF DEST[31:0] > SRC[31:0]
        THEN DEST[31:0] ← FFFFFFFFH;
        ELSE DEST[31:0] ← 0; FI;
    (* Continue comparison of 2nd and 3rd doublewords in DEST and SRC *)
    IF DEST[127:96] > SRC[127:96]
        THEN DEST[127:96] ← FFFFFFFFH;
        ELSE DEST[127:96] ← 0; FI;

… … …
```

## 4. Material covering handling of VM Exit during Virtual-NMI injection corrected

In Section 25.7.1.2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, some VM-exit material has been corrected. See the reproduced section below noted by change bars.

---

### 25.7.1.2 Resuming Guest Software after Handling an Exception

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

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

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

If both are true and bit 12 of the VM-exit interruption-information field is 1, NMIs were blocked before guest software executed the IRET instruction that caused the fault that caused the VM exit. The VMM should set bit 3 (blocking by NMI) in the interruptibility-state field (using VMREAD and VMWRITE) before resuming guest software.

- If the “virtual NMIs” VM-execution control is 1, bit 12 of the VM-exit interruption-information field indicates that the VM exit was due to a fault encountered during an execution of the IRET instruction that removed virtual-NMI blocking. In particular, it provides this indication if the following are both true:

  — Bit 31 (valid) in the IDT-vectoring information field is 0.

  — The value of bits 7:0 (vector) of the VM-exit interruption-information field is not 8 (the VM exit is not due to a double-fault exception).

  If both are true and bit 12 of the VM-exit interruption-information field is 1, there was virtual-NMI blocking before guest software executed the IRET instruction that caused the fault that caused the VM exit. The VMM should set bit 3 (blocking by NMI) in the interruptibility-state field (using VMREAD and VMWRITE) before resuming guest software.

- Bit 31 (valid) of the IDT-vectoring information field indicates, if set, that the exception causing the VM exit occurred while another event was being delivered to guest software. The VMM should ensure that the other event is delivered when guest software is resumed. It can do so using the VM-entry event injection described in Section 22.5 and detailed in the following paragraphs:

  — The VMM can copy (using VMREAD and VMWRITE) the contents of the IDT-vectoring information field (which is presumed valid) to the VM-entry interruption-information field (which, if valid, will cause the exception to be delivered as part of the next VM entry).

    - The VMM should ensure that reserved bits 30:12 in the VM-entry interruption-information field are 0. In particular, the value of bit 12 in the IDT-vectoring information field is undefined after all VM exits. If this bit is copied as 1 into the VM-entry interruption-information field, the next VM entry will fail because the bit should be 0.

    - If the “virtual NMIs” VM-execution control is 1 and the value of bits 10:8 (interruption type) in the IDT-vectoring information field is 2 (indicating NMI), the VM exit occurred during delivery of an NMI that had been injected as part of the previous VM entry. In this case, bit 3 (blocking by NMI) will be 1 in the interruptibility-state field in the VMCS. The VMM should clear this bit; otherwise, the next VM entry will fail (see Section 22.3.1.5).

  — The VMM can also copy the contents of the IDT-vectoring error-code field to the VM-entry exception error-code field. This need not be done if bit 11 (error code valid) is clear in the IDT-vectoring information field.

  — The VMM can also copy the contents of the VM-exit instruction-length field to the VM-entry instruction-length field. This need be done only if bits 10:8 (interruption type) in the IDT-vectoring information field indicate either software interrupt, privileged software exception, or software exception.

## 5. More information about R8-15 & XMM8-15 transitions

In Section 3.4.1.1 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, information covering mode transition behavior for R8-15 and XMM8-15 has been added. This information has been reproduced in context below noted by change bars.

---

### 3.4.1.1 General-Purpose Registers in 64-Bit Mode

In 64-bit mode, there are 16 general purpose registers and the default operand size is 32 bits. However, general-purpose registers are able to work with either 32-bit or 64-bit operands. If a 32-bit operand size is specified: EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D are available. If a 64-bit operand size is specified: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8-R15 are available. R8D-R15D/R8-R15 represent eight new general-purpose registers. All of these registers can be accessed at the byte, word, dword, and qword level. REX prefixes are used to generate 64-bit operand sizes or to reference registers R8-R15.

Registers only available in 64-bit mode (R8-R15 and XMM8-XMM15) are preserved across transitions from 64-bit mode into compatibility mode then back into 64-bit mode. However, values of R8-R15 and XMM8-XMM15 are undefined after transitions from 64-bit mode through compatibility mode to legacy or real mode and then back through compatibility mode to 64-bit mode.

**Table 3-2. Addressable General Purpose Registers**

| Register Type | Without REX | With REX |
|---|---|---|
| Byte Registers | AL, BL, CL, DL, AH, BH, CH, DH | AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8L - R15L |
| Word Registers | AX, BX, CX, DX, DI, SI, BP, SP | AX, BX, CX, DX, DI, SI, BP, SP, R8W - R15W |
| Doubleword Registers | EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP | EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D |
| Quadword Registers | N.A. | RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8 - R15 |

In 64-bit mode, there are limitations on accessing byte registers. An instruction cannot reference legacy high-bytes (for example: AH, BH, CH, DH) and one of the new byte registers at the same time (for example: the low byte of the RAX register). However, instructions may reference legacy low-bytes (for example: AL, BL, CL or DL) and new byte registers at the same time (for example: the low byte of the R8 register, or RBP). The architecture enforces this limitation by changing high-byte references (AH, BH, CH, DH) to low byte references (BPL, SPL, DIL, SIL: the low 8 bits for RBP, RSP, RDI and RSI) for instructions using a REX prefix.

When in 64-bit mode, operand size determines the number of valid bits in the destination general-purpose register:

- 64-bit operands generate a 64-bit result in the destination general-purpose register.
- 32-bit operands generate a 32-bit result, zero-extended to a 64-bit result in the destination general-purpose register.
- 8-bit and 16-bit operands generate an 8-bit or 16-bit result. The upper 56 bits or 48 bits (respectively) of the destination general-purpose register are not be modified by the operation. If the result of an 8-bit or 16-bit operation is intended for 64-bit address calculation, explicitly sign-extend the register to the full 64-bits.

Because the upper 32 bits of 64-bit general-purpose registers are undefined in 32-bit modes, the upper 32 bits of any general-purpose register are not preserved when switching from 64-bit mode to a 32-bit mode (to protected mode or compatibility mode). Software must not depend on these bits to maintain a value after a 64-bit to 32-bit mode switch.

## 6. Figure 8-6 corrected

In Figure 8-6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, bit designations have been corrected. See the corrected figure below.

---

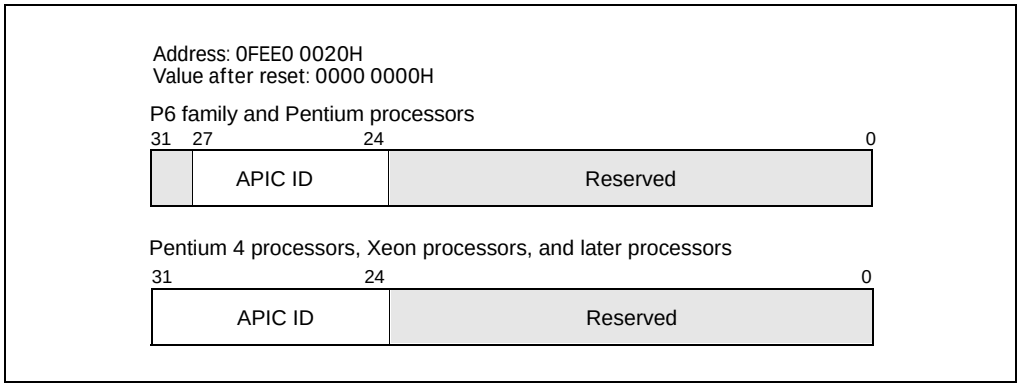


**Figure 8-6. Local APIC ID Register**

## 7. Note added to section on APIC timer

In Section 8.5.4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, a note has been added (discusses deep C-states and GV3 transitions). Part of the section is reproduced below with the change in context noted by change bar.

---

### 8.5.4 APIC Timer

The local APIC unit contains a 32-bit programmable timer that is available to software to time events or operations. This timer is set up by programming four registers: the divide configuration register (see Figure 8-10), the initial-count and current-count registers (see Figure 8-11), and the LVT timer register (see Figure 8-8).

**NOTE**

The APIC timer may temporarily stop while the processor is in deep C-states or during SpeedStep (EST) transitions.

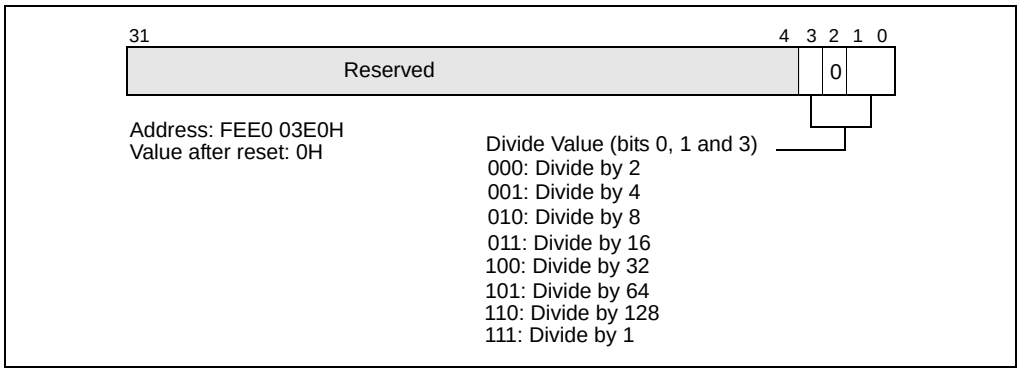


**Figure 8-10. Divide Configuration Register**

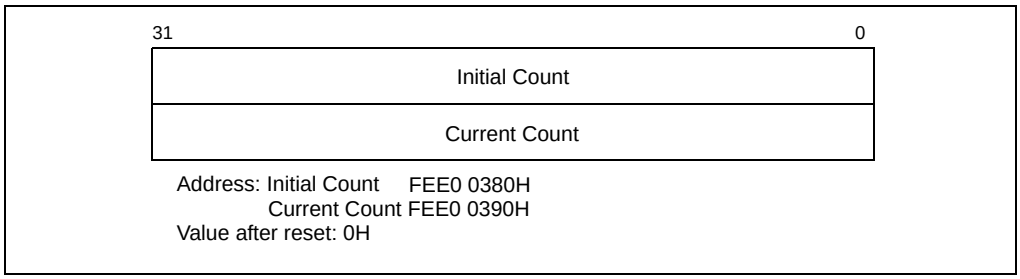


**Figure 8-11. Initial Count and Current Count Registers**

*... ... ... section continues....*

## 8. Coverage of PEBS updated

In Section 18.14.4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, coverage of PEBS has been updated. Coverage has also been updated in Appendix B of the same volume. See the reproductions of the applicable sections below noted by change bars.

---

*Addition to Chapter 18, Vol. 3B.*

### 18.14.4 Precise Even Based Sampling (PEBS)

Processors based on Intel Core microarchitecture also support precise event based sampling (PEBS). This feature was introduced by processors based on Intel NetBurst microarchitecture.

PEBS uses a debug store mechanism and a performance monitoring interrupt to store a set of architectural state information for the processor (See Section 18.15.8). The information provides architectural state of the instruction executed immediately after the instruction that caused the event.

In cases where the same instruction causes BTS and PEBS to be activated, PEBS is processed before BTS are processed. The PMI request is held until the processor completes processing of PEBS and BTS.

For processors based on Intel Core microarchitecture, events that support precise sampling are listed in Table 18-15. The procedure for detecting availability of PEBS is the same as described in Section 18.15.8.1.

**Table 18-15. PEBS Performance Events for Intel Core Microarchitecture**

| Event Name | UMask | Event Select |
|---|---|---|
| INSTR_RETIRED.ANY_P | 00H | C0H |
| X87_OPS_RETIRED.ANY | FEH | C1H |
| BR_INST_RETIRED.MISPRED | 00H | C5H |
| SIMD_INST_RETIRED.ANY | 1FH | C7H |
| MEM_LOAD_RETIRED.L1D_MISS | 01H | CBH |
| MEM_LOAD_RETIRED.L1D_LINE_MISS | 02H | CBH |
| MEM_LOAD_RETIRED.L2_MISS | 04H | CBH |
| MEM_LOAD_RETIRED.L2_LINE_MISS | 08H | CBH |
| MEM_LOAD_RETIRED.DTLB_MISS | 10H | CBH |

### 18.14.4.1 Setting up the PEBS Buffer

For processors based on Intel Core microarchitecture, PEBS is available using IA32_PMC0 only. Use the following procedure to set up the processor and IA32_PMC0 counter for PEBS:

1. Set up the precise event buffering facilities. Place values in the precise event buffer base, precise event index, precise event absolute maximum, precise event interrupt threshold, and precise event counter reset fields of the DS buffer management area. In processors based on Intel Core microarchitecture, PEBS records consist of 64-bit address entries. See Figure 18-24 to set up the precise event records buffer in memory.
2. Enable PEBS. Set the Enable PEBS on PMC0 flag (bit 0) in IA32_PEBS_ENABLE MSR.
3. Set up the IA32_PMC0 performance counter and IA32_PERFEVTSEL0 for an event listed in Table 18-15.

### 18.14.4.2 Writing a PEBS Interrupt Service Routine

PEBS facilities share the same interrupt vector and interrupt service routine (called the DS ISR) with the non-precise event-based sampling and BTS facilities. To handle PEBS interrupts, PEBS handler code must be included in the DS ISR. See Section 18.5.2.2, “Debug Store (DS) Mechanism,” for guidelines when writing the DS ISR.

The service routine can query MSR_PERF_GLOBAL_STATUS to determine which counter(s) caused of overflow condition. The service routine should clear overflow indicator by writing to MSR_PERF_GLOBAL_OVF_CTL.

A comparison of the sequence of requirements to program PEBS for processors based on Intel Core and Intel NetBurst microarchitectures is listed in Table 18-16.

**Table 18-16. Requirements to Program PEBS**

| | For Processors based on Intel Core microarchitecture | For Processors based on Intel NetBurst microarchitecture |
|---|---|---|
| Verify PEBS support of processor/OS | • IA32_MISC_ENABLES.EMON_AVAILABE (bit 7) is set.<br>• IA32_MISC_ENABLES.PEBS_UNAVAILABE (bit 12) is clear. | |
| Ensure counters are in disabled | On initial set up or changing event configurations, write MSR_PERF_GLOBAL_CTRL MSR (0x38F) with 0.<br>On subsequent entries:<br>• Clear all counters if "Counter Freeze on PMI" is not enabled.<br>• If IA32_DebugCTL.Freeze is enabled, counters are automatically disabled.<br>Counters MUST be stopped before writing.[a] | Optional |
| Disable PEBS. | Clear ENABLE PMC0 bit in IA32_PEBS_ENABLE MSR (0x3F1). | Optional |
| Check overflow conditions. | Check MSR_PERF_GLOBAL_STATUS MSR (0x 38E) handle any overflow conditions. | Check OVF flag of each CCCR for overflow condition |
| Clear overflow status. | Clear MSR_PERF_GLOBAL_STATUS MSR (0x 38E) using IA32_CR_PERF_GLOBAL_OVF_CTRL MSR (0x390). | Clear OVF flag of each CCCR. |
| Write "sample-after" values. | Configure the counter(s) with the sample after value. | |
| Configure specific counter configuration MSR. | • Set local enable bit 22 - 1.<br>• Do NOT set local counter PMI/INT bit, bit 20 - 0.<br>• Event programmed must be PEBS capable. | • Set appropriate OVF_PMI bits - 1.<br>• Only CCCR for MSR_IQ_COUNTER4 support PEBS. |
| Allocate buffer for PEBS states. | Allocate a buffer in memory for the precise information. | |
| Program the IA32_DS_AREA MSR. | Program the IA32_DS_AREA MSR. | |
| Configure the PEBS buffer management records. | Configure the PEBS buffer management records in the DS buffer management area. | |
| Configure/Enable PEBS. | Set Enable PMC0 bit in IA32_PEBS_ENABLE MSR (0x3F1). | Configure MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT and MSR_PEBS_MATRIX_HORZ as needed. |
| Enable counters. | Set Enable bits in MSR_PERF_GLOBAL_CTRL MSR (0x38F). | Set each CCCR enable bit 12 - 1. |

a. Counters read while enabled are not guaranteed to be precise with event counts that occur in timing proximity to the RDMSR.

---

*Addition to Appendix B, Vol. 3B.*

**B-1. MSRs in Processors Based on Intel Core Microarchitecture (Contd.)**

| | | | | |
|---|---|---|---|---|
| *... table continues...* | | | | |
| 3F1H | 1009 | IA32_PEBS_ ENABLE | Unique | See Section 18.14.4, "Precise Even Based Sampling (PEBS)." |
| | | 0 | | Enable PEBS on IA32_PMC0. (R/W) |
| *....table continues...* | | | | |

## 9. Missing exception added for MFENCE

In Section "MFENCE—Memory Fence" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, an exception section has been added. See below.

---

### MFENCE—Memory Fence

*... ... .... more material here ... ...*

#### Exceptions (All Modes of Operation)

| | |
|---|---|
| #UD | If CPUID.01H:EDX.SSE2[bit 26] = 0. |

*... ... .... more material here ... ...*

## 10. IA32_MCG_STATUS information added

In Section 7.8.5 of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3A*, information has been updated to better reflect the implementation of IA32_MCG_STATUS. This section is reproduced below noted by change bars.

---

### 7.8.5 Machine Check Architecture

In the HT Technology context, only the IA32_MCG_STATUS MSR is duplicated for each logical processor. This design is compatible with machine check exception handlers that follow guidelines given in Chapter 14. Note that the MCA specification permits duplication of MSRs other than IA32_MCG_STATUS, but current implementations do not take advantage of this. Software that follows the guidelines in Chapter 14 for machine check exception handlers does not need to be aware of whether an implementation duplicates the other machine check MSRs.

The IA32_MCG_STATUS MSR is duplicated for each logical processor so that its machine check in progress bit field (MCIP) can be used to detect recursion on the part of MCA handlers. In addition, the MSR allows each logical processor to determine that a machine-check exception is in progress independent of the actions of another logical processor in the same physical package.

Because the logical processors within a physical package are tightly coupled with respect to shared hardware resources, both logical processors are notified of machine check errors that occur within a given physical processor. If machine-check exceptions are enabled when a fatal error is reported, all the logical processors within a physical

package are dispatched to the machine-check exception handler. If machine-check exceptions are disabled, the logical processors enter the shutdown state and assert the IERR# signal.

When enabling machine-check exceptions, the MCE flag in control register CR4 should be set for each logical processor.

## 11. Introduction section for CPUID updated

In Section "CPUID—CPU Identification" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, a footnote has been added to clarify behavior in 64-bit processors. The impacted area is reproduced below noted by change bars.

---

### CPUID—CPU Identification

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F A2 | CPUID | Valid | Valid | Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well). |

#### Description

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

CPUID returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers.[1] The instruction's output is dependent on the contents of the EAX register upon execution (in some cases, ECX as well). For example, the following pseudocode loads EAX with 00H and causes CPUID to return a Maximum Return Value and the Vendor Identification String in the appropriate registers:

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

1. On Intel 64 processors, CPUID clears the high 32 bits of the RAX/RBX/RCX/RDX registers in all modes.

*.. ... ... section continues ...*

## 12. Update to CPUID documentation on deterministic cache parameters leaf

In Table 3-12 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, information has been added for CPUID.04H:EAX[Bit 10, Bit 11] values. The impacted part of the table has been reproduced below noted by change bars.

---

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

| | |
|---|---|
| *… table continues ….* | |
| | *Deterministic Cache Parameters Leaf* |
| 04H | **NOTES:**<br>04H output depends on the initial value in ECX.<br>See also: "INPUT EAX = 4: Returns Deterministic Cache Parameters for each level on page 3-177. |
| | *Deterministic Cache Parameters Leaf* |
| | EAX Bits 4-0: Cache Type Field<br>0 = Null - No more caches 3 = Unified Cache<br>1 = Data Cache 4-31 = Reserved<br>2 = Instruction Cache<br>Bits 7-5: Cache Level (starts at 1)<br>Bits 8: Self Initializing cache level (does not need SW initialization)<br>Bits 9: Fully Associative cache<br>Bit 10: Write-Back Invalidate/Invalidate<br>0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache<br>1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache.<br>Bit 11: Cache Inclusiveness<br>0 = Cache is not inclusive of lower cache levels.<br>1 = Cache is inclusive of lower cache levels.<br>Bits 13-12: Reserved<br>Bits 25-14: Maximum number of threads sharing this cache in a physical package*<br>Bits 31-26: Maximum number of processor cores in the physical package* **<br>EBX Bits 11-00: L = System Coherency Line Size*<br>Bits 21-12: P = Physical Line partitions*<br>Bits 31-22: W = Ways of associativity* |
| | ECX Bits 31-00: S = Number of Sets*<br>EDX Reserved = 0<br>**NOTES:**<br>* Add one to the return value to get the result.<br>**The returned value is constant for valid initial values in ECX. Valid ECX values start from 0. |
| *… table continues ….* | |

## 13. Updated pseudocode in VMCALL description

In Section "VMCALL—Call to VM Monitor" in Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, the pseudocode has been corrected. See the reproduced segment below noted by change bars.

---

### Operation

IF not in VMX operation

```
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VM exit;
ELSIF (RFLAGS.VM = 1) OR (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF in SMM or the logical processor does not support the dual-monitor treatment of SMIs and SMM or
the valid bit in the IA32_SMM_MONITOR_CTL MSR is clear
    THEN VMfail (VMCALL executed in VMX root operation);
ELSIF dual-monitor treatment of SMIs and SMM is active
    THEN perform an SMM VM exit (see Section 24.16.2
     of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B);
ELSIF current-VMCS pointer is not valid
    THEN VMfailInvalid;
ELSIF launch state of current VMCS is not clear
    THEN VMfailValid(VMCALL with non-clear VMCS);
ELSIF VM-exit control fields are not valid (see Section 24.16.6.1 of the Intel® 64 and IA-32 Architec-
tures Software Developer's Manual, Volume 3B)
    THEN VMfailValid (VMCALL with invalid VM-exit control fields);
ELSE
    enter SMM;
    read revision identifier in MSEG;
    IF revision identifier does not match that supported by processor
        THEN
            leave SMM;
            VMfailValid(VMCALL with incorrect MSEG revision identifier);
        ELSE
            read SMM-monitor features field in MSEG (see Section 24.16.6.2,
            in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B);
            IF features field is invalid
                THEN
                    leave SMM;
                    VMfailValid(VMCALL with invalid SMM-monitor features);
                ELSE activate dual-monitor treatment of SMIs and SMM (see Section 24.16.6
                in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B);
            FI;
        FI;
FI;
```

## 14. IA32_MC*i*_STATUS figure corrected

In Figure 14.5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, a field definition has been corrected. The figure is reproduced below. See the model-specific error code field.

---

.

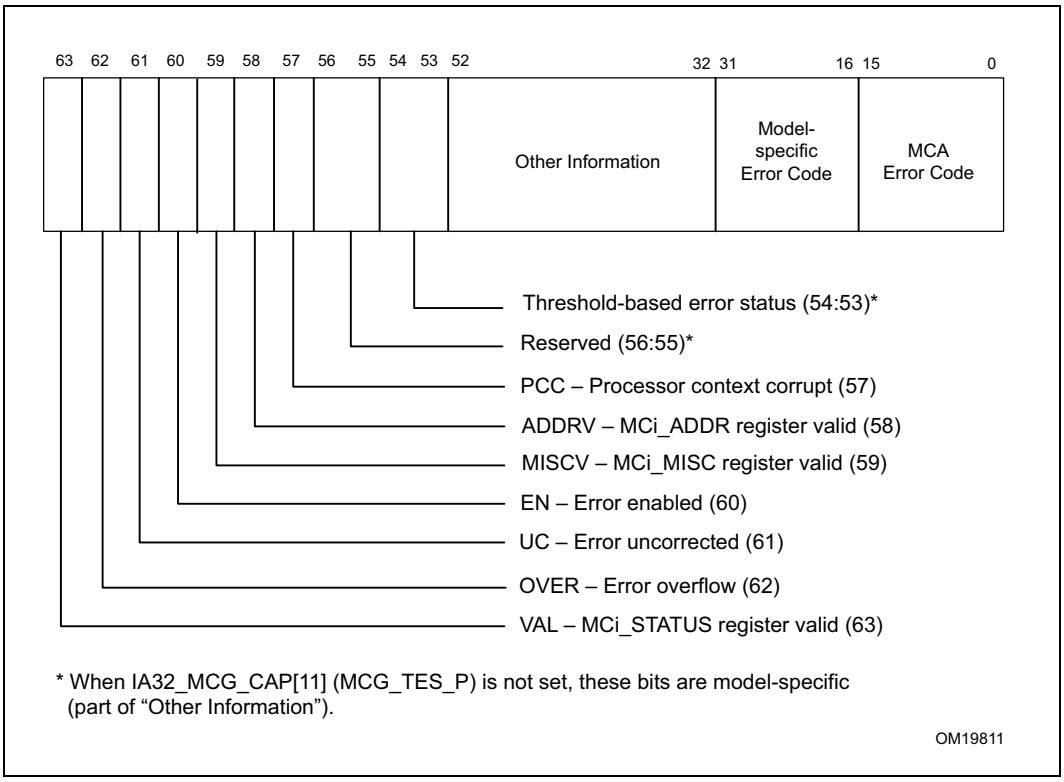


**Figure 14-5. IA32_MC*i*_STATUS Register**

## 15. IA32_MCi_STATUS flag description updated

In Section 14.8.1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, the information describing IA32_MCi_STATUS flags has been corrected. This section is reproduced below noted by change bar.

---

### 14.8.6 Machine-Check Exception Handler

The machine-check exception (#MC) corresponds to vector 18. To service machine-check exceptions, a trap gate must be added to the IDT. The pointer in the trap gate must point to a machine-check exception handler. Two approaches can be taken to designing the exception handler:

1. The handler can merely log all the machine status and error information, then call a debugger or shut down the system.
2. The handler can analyze the reported error information and, in some cases, attempt to correct the error and restart the processor.

For Pentium 4, Intel Xeon, P6 family, and Pentium processors; virtually all machine-check conditions cannot be corrected (they result in abort-type exceptions). The logging of status and error information is therefore a baseline implementation requirement.

When recovery from a machine-check error may be possible, consider the following when writing a machine-check exception handler:

- To determine the nature of the error, the handler must read each of the error-reporting register banks. The count field in the IA32_MCG_CAP register gives number of register banks. The first register of register bank 0 is at address 400H.
- The VAL (valid) flag in each IA32_MC*i*_STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank do not contain valid error information and do not need to be checked.
- To write a portable exception handler, only the MCA error code field in the IA32_MC*i*_STATUS register should be checked. See Section 14.7, “Interpreting the MCA Error Codes,” for information that can be used to write an algorithm to interpret this field.
- The RIPV, PCC, and OVER flags in each IA32_MCi_STATUS register indicate whether recovery from the error is possible. If PCC or OVER are set, recovery is not possible. If RIPV is not set, program execution can not be restarted reliably. When recovery is not possible, the handler typically records the error information and signals an abort to the operating system.
- Correctable errors are corrected automatically by the processor. The UC flag in each IA32_MC*i*_STATUS register indicates whether the processor automatically corrected an error.
- The RIPV flag in the IA32_MCG_STATUS register indicates whether the program can be restarted at the instruction indicated by the instruction pointer (the address of the instruction pushed on the stack when the exception was generated). If this flag is clear, the processor may still be able to be restarted (for debugging purposes) but not without loss of program continuity.
- For unrecoverable errors, the EIPV flag in the IA32_MCG_STATUS register indicates whether the instruction indicated by the instruction pointer pushed on the stack (when the exception was generated) is related to the error. If the flag is clear, the pushed instruction may not be related to the error.
- The MCIP flag in the IA32_MCG_STATUS register indicates whether a machine-check exception was generated. Before returning from the machine-check exception handler, software should clear this flag so that it can be used reliably by an error logging utility. The MCIP flag also detects recursion. The machine-check architecture does not support recursion. When the processor detects machine-check recursion, it enters the shutdown state.

## 16. Instruction summaries fixed for MOVD/MOVQ, PMOVMSKB, PINSRW, PEXTRW

For sections on individual instructions in Chapters 3 and 4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A & 2B*, the positioning of REX prefixes in instruction summary tables has been corrected. The applicable tables are reproduced below noted by change bars.

---

## MOVD/MOVQ—Move Doubleword/Move Quadword

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 6E /*r* | MOVD *mm, r/m32* | Valid | Valid | Move doubleword from *r/m32* to *mm*. |
| REX.W + 0F 6E /*r* | MOVQ *mm, r/m64* | Valid | N.E. | Move quadword from *r/m64* to *mm*. |
| 0F 7E /*r* | MOVD *r/m32, mm* | Valid | Valid | Move doubleword from *mm* to *r/m32*. |
| REX.W + 0F 7E /*r* | MOVQ *r/m64, mm* | Valid | N.E. | Move quadword from *mm* to *r/m64*. |
| 66 0F 6E /*r* | MOVD *xmm, r/m32* | Valid | Valid | Move doubleword from *r/m32* to *xmm*. |
| REX.W 66 0F 6E /*r* | MOVQ *xmm, r/m64* | Valid | N.E. | Move quadword from *r/m64* to *xmm*. |
| 66 0F 7E /*r* | MOVD *r/m32, xmm* | Valid | Valid | Move doubleword from *xmm* register to *r/m32*. |
| REX.W 66 0F 7E /*r* | MOVQ *r/m64, xmm* | Valid | N.E. | Move quadword from *xmm* register to *r/m64*. |

*Text omitted here.....*

## PMOVMSKB—Move Byte Mask

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F D7 /*r* | PMOVMSKB *r32, mm* | Valid | Valid | Move a byte mask of *mm* to *r32*. |
| REX.W + 0F D7 /*r* | PMOVMSKB *r64, mm* | Valid | N.E. | Move a byte mask of mm to the lower 32-bits of r64 and zero-fill the upper 32-bits. |
| 66 0F D7 /*r* | PMOVMSKB *r32, xmm* | Valid | Valid | Move a byte mask of *xmm* to *r32*. |
| 66 REX.W 0F D7 /*r* | PMOVMSKB *r64, xmm* | Valid | N.E. | Move a byte mask of xmm to the lower 32-bits of r64 and zero-fill the upper 32-bits. |

*Text omitted here.....*

## PINSRW—Insert Word

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F C4 /*r* ib | PINSRW *mm*, *r32/m16*, *imm8* | Valid | Valid | Insert the low word from *r32* or from *m16* into *mm* at the word position specified by *imm8* |
| REX.W + 0F C4 /*r* ib | PINSRW *mm*, *r64/m16*, *imm8* | Valid | N.E. | Insert the low word from *r64* or from *m16* into *mm* at the word position specified by *imm8* |
| 66 0F C4 /*r* ib | PINSRW xmm, *r32/m16*, imm8 | Valid | Valid | Move the low word of *r32* or from *m16* into xmm at the word position specified by *imm8*. |
| 66 REX.W 0F C4 /*r* ib | PINSRW xmm, *r64/m16*, imm8 | Valid | N.E. | Move the low word of *r64* or from *m16* into xmm at the word position specified by *imm8*. |

*Text omitted here.....*

## PEXTRW—Extract Word

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F C5 /*r* ib | PEXTRW *r32*, *mm*, *imm8* | Valid | Valid | Extract the word specified by *imm8* from *mm* and move it to *r32*, bits 15-0. Zero-extend the result. |
| REX.W + 0F C5 /*r* ib | PEXTRW *r64*, *mm*, *imm8* | Valid | N.E. | Extract the word specified by *imm8* from *mm* and move it to *r64*, bits 15-0. Zero-extend the result. |
| 66 0F C5 /*r* ib | PEXTRW *r32*, *xmm*, *imm8* | Valid | Valid | Extract the word specified by *imm8* from *xmm* and move it to *r32*, bits 15-0. Zero-extend the result. |
| 66 REX.W 0F C5 /*r* ib | PEXTRW *r64*, *xmm*, *imm8* | Valid | N.E. | Extract the word specified by *imm8* from *xmm* and move it to *r64*, bits 15-0. Zero-extend the result. |

*Text omitted here.....*

## 17. Correction to microcode update documentation

In Section 9.11.6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, additions made pertaining to 64-bit support. The section is reproduced below noted by change bars.

---

### 9.11.6 Microcode Update Loader

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

Example 9-8 below represents the update loader with a loader revision of 00000001H. Note that the microcode update must be aligned on a 16-byte boundary and the size of the microcode update must be 1-KByte granular.

**Example 9-8. Assembly Code Example of Simple Microcode Update Loader**

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

The loader shown in Example 9-8 assumes that *update* is the address of a microcode update (header and data) embedded within the code segment of the BIOS. It also assumes that the processor is operating in real mode. The data may reside anywhere in memory, aligned on a 16-byte boundary, that is accessible by the processor within its current operating mode.

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

- In 64-bit mode, EAX contains the lower 32-bits of the microcode update linear address. In protected mode, EAX contains the full 32-bit linear address of the microcode update.
- In 64-bit mode, EDX contains the upper 32-bits of the microcode update linear address. In protected mode, EDX equals zero.
- ECX contains 79H (address of IA32_BIOS_UPDT_TRIG).

Other requirements are:

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

*Section continues, omitted material starts here.....*

## 18. PSHUFB compiler intrinsic fixed

In Section "PSHUFB — Packed Shuffle Bytes" in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, a compiler intrinsic has been corrected. The new subsection is reproduced below.

---

### Intel C/C++ Compiler Intrinsic Equivalent

PSHUFB __m64 _mm_shuffle_pi8 (__m64 a, __m64 b)

PSHUFB __m128i _mm_shuffle_epi8 (__m128i a, __m128i b)

## 19. RDMSR/RDPMC/RDTSC/WRMSR descriptions updated

In the subsections covering RDMSR, RDPMC, RDTSC and WRMSR in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, descriptions have been updated to correct errors and enforce consistency. The new language is provided below. See the change bars.

---

## RDMSR—Read from Model Specific Register

| Opcode* | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 32 | RDMSR | Valid | Valid | Load MSR specified by ECX into EDX:EAX. |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

### Description

Loads the contents of a 64-bit model specific register (MSR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the MSR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) will be generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception.

The MSRs control functions for testability, execution tracing, performance-monitoring, and machine check errors. Appendix B, "Model-Specific Registers (MSRs)," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, lists all the MSRs that can be read with this instruction and their addresses. Note that each processor family has its own set of MSRs.

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



### IA-32 Architecture Compatibility

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

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 21 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*, for more information about the behavior of this instruction in VMX non-root operation.

### Operation

EDX:EAX ← MSR[ECX];

### Flags Affected

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| | If the value in ECX specifies a reserved or unimplemented MSR address. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If the value in ECX specifies a reserved or unimplemented MSR address. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

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

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| | If the value in ECX or RCX specifies a reserved or unimplemented MSR address. |
| #UD | If the LOCK prefix is used. |

## RDPMC—Read Performance-Monitoring Counters

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

### Description

Loads the 40-bit performance-monitoring counter specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 8 bits of the counter and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) See below for the treatment of the EDX register for “fast” reads.

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

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

| Processor Family | CPUID Family/Model/ Other Signatures | Valid PMC Index Range | 40-bit Counters |
|---|---|---|---|
| P6 | Family 06H | 0, 1 | 0, 1 |
| Pentium® 4, Intel® Xeon processors | Family 0FH; Model 00H, 01H, 02H | ≥ 0 and ≤ 17 | ≥ 0 and ≤ 17 |
| Pentium 4, Intel Xeon processors | (Family 0FH; Model 03H, 04H, 06H) and (L3 is absent) | ≥ 0 and ≤ 17 | ≥ 0 and ≤ 17 |
| Pentium M processors | Family 06H, Model 09H, 0DH | 0, 1 | 0, 1 |
| 64-bit Intel Xeon processors with L3 | (Family 0FH; Model 03H, 04H) and (L3 is present) | ≥ 0 and ≤ 25 | ≥ 0 and ≤ 17 |
| Intel® Core™ Solo and Intel Core Duo processors, Dual-core Intel Xeon processor LV | Family 06H, Model 0EH | 0, 1 | 0, 1 |
| Intel® Core™2 Duo processor, Intel Xeon processor 3000, 5100, 5300 Series - general-purpose PMC | Family 06H, Model 0FH | 0, 1 | 0, 1 |
| Intel Xeon processors 7100 series with L3 | (Family 0FH; Model 06H) and (L3 is present) | ≥ 0 and ≤ 25 | ≥ 0 and ≤ 17 |

The Pentium 4 and Intel Xeon processors also support “fast” (32-bit) and “slow” (40-bit) reads on the first 18 performance counters. Selected this option using ECX[bit 31]. If bit 31 is set, RDPMC reads only the low 32 bits of the selected

performance counter. If bit 31 is clear, all 40 bits are read. A 32-bit result is returned in EAX and EDX is set to 0. A 32-bit read executes faster on Pentium 4 processors and Intel Xeon processors than a full 40-bit read.

On 64-bit Intel Xeon processors with L3, performance counters with indices 18-25 are 32-bit counters. EDX is cleared after executing RDPMC for these counters. On Intel Xeon processor 7100 series with L3, performance counters with indices 18-25 are also 32-bit counters.

In Intel Core 2 processor family, Intel Xeon processor 3000, 5100, and 5300 series, the fixed-function performance counters are 48-bit wide and can be accessed by RDMPC with ECX between from 8000_0000H and 8000_0002H.

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

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

The performance-monitoring counters are event counters that can be programmed to count events such as the number of instructions decoded, number of interrupts received, or number of cache loads. Appendix A, “Performance Monitoring Events,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*, lists the events that can be counted for various processors in the Intel 64 and IA-32 architecture families.

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

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

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

## Operation

```
(* Intel Core 2 Duo processor family and Intel Xeon processor 3000, 5100, 5300 series*)

IF (ECX = 0 or 1) and ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN IF (ECX[31] = 1)
        EAX ← IA32_FIXED_CTR(ECX)[30:0];
        EDX ← IA32_FIXED_CTR(ECX)[39:32];
    ELSE IF (ECX[30:0] in valid range)
        EAX ← PMC(ECX[30:0])[31:0];
```

```
        EDX ← PMC(ECX[30:0])[39:32];
    ELSE IF (ECX[31] and ECX[30:0] in valid fixed-counter range)
        EAX ← FIXED_PMC(ECX[30:0])[31:0];
        EDX ← FIXED_PMC(ECX[30:0])[47:32];
    ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0);
FI;

(* P6 family processors and Pentium processor with MMX technology *)

IF (ECX = 0 or 1) and ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN
        EAX ← PMC(ECX)[31:0];
        EDX ← PMC(ECX)[39:32];
    ELSE (* ECX is not 0 or 1 or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0);
FI;
(* Processors with CPUID family 15 *)
IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN IF (ECX[30:0] = 0:17)
        THEN IF ECX[31] = 0
            THEN
                EAX ← PMC(ECX[30:0])[31:0]; (* 40-bit read *)
                EDX ← PMC(ECX[30:0])[39:32];
        ELSE (* ECX[31] = 1*)
            THEN
                EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
                EDX ← 0;
        FI;
    ELSE IF (*64-bit Intel Xeon processor with L3 *)
        THEN IF (ECX[30:0] = 18:25 )
            EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
            EDX ← 0;
        FI;
    ELSE IF (*Intel Xeon processor 7100 series with L3 *)
        THEN IF (ECX[30:0] = 18:25 )
            EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
            EDX ← 0;
        FI;
    ELSE (* Invalid PMC index in ECX[30:0], see Table 4-4. *)
        GP(0);
    FI;
ELSE (* CR4.PCE = 0 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
    #GP(0);
FI;
```

## Flags Affected

None.



### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.<br>If an invalid performance counter index is specified (see Table 4-2).<br>(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If an invalid performance counter index is specified (see Table 4-2).<br>(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If the PCE flag in the CR4 register is clear.<br>If an invalid performance counter index is specified (see Table 4-2).<br>(Pentium 4 and Intel Xeon processors) If the value in ECX[30:0] is not within the valid range. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.<br>If an invalid performance counter index is specified in ECX[30:0] (see Table 4-2). |
| #UD | If the LOCK prefix is used. |

## RDTSC—Read Time-Stamp Counter

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 31 | RDTSC | Valid | Valid | Read time-stamp counter into EDX:EAX. |

### Description

Loads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.)

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See “Time Stamp Counter” in Chapter 18 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*, for specific details of the time stamp counter behavior.

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

The time-stamp counter can also be read with the RDMSR instruction, when executing at privilege level 0.

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

This instruction was introduced by the Pentium processor.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 21 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*, for more information about the behavior of this instruction in VMX non-root operation.

### Operation

```
IF (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)
    THEN EDX:EAX ← TimeStampCounter;
    ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
        #GP(0);
FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the TSD flag in register CR4 is set and the CPL is greater than 0. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If the TSD flag in register CR4 is set. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## WRMSR—Write to Model Specific Register

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 30 | WRMSR | Valid | Valid | Write the value in EDX:EAX to MSR specified by ECX. |

### Description

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected MSR and the contents of the EAX register are copied to low-order 32 bits of the MSR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an MSR should be set to values previously read.

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

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated. This includes global entries (see “Translation Lookaside Buffers (TLBs)” in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Appendix B, “Model-Specific Registers (MSRs)”, in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B,* lists all MSRs that can be read with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The WRMSR instruction is a serializing instruction (see “Serializing Instructions” in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A)*.

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

### IA-32 Architecture Compatibility

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

### Operation

MSR[ECX] ← EDX:EAX;

### Flags Affected

None.

### Protected Mode Exceptions

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

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If the value in ECX specifies a reserved or unimplemented MSR address. |
| | If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

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

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## 20. Location data corrected

In Table B-1 in Appendix B of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, the status of MSR location1E0H has been changed. It is now reserved. See IA32_MISC_ENABLE for the fast string enable bit in the Intel Core Microarchitecture.

## 21. LOOP/LOOPcc description updated

In the subsection LOOP/LOOPcc in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, the description has been updated to an correct errors. The old version incorrectly represented LOOP as REX.W dependent.

---

## LOOP/LOOP*cc*—Loop According to ECX Counter

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| E2 *cb* | LOOP *rel8* | Valid | Valid | Decrement count; jump short if count ≠ 0. |
| E1 *cb* | LOOPE *rel8* | Valid | Valid | Decrement count; jump short if count ≠ 0 and ZF = 1. |
| E0 *cb* | LOOPNE *rel8* | Valid | Valid | Decrement count; jump short if count ≠ 0 and ZF = 0. |

### Description

Performs a loop operation using the RCX, ECX or CX register as a counter (depending on whether address size is 64 bits, 32 bits, or 16 bits). Note that the LOOP instruction ignores REX.W; but 64-bit address size can be over-ridden using a 67H prefix.

Each time the LOOP instruction is executed, the count register is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the IP/EIP/RIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of –128 to +127 are allowed with this instruction.

Some forms of the loop instruction (LOOP*cc*) also accept the ZF flag as a condition for terminating the loop before the count reaches zero. With these forms of the instruction, a condition code (*cc*) is associated with each instruction to indicate the condition being tested for. Here, the LOOP*cc* instruction itself does not affect the state of the ZF flag; the ZF flag is changed by other instructions in the loop.

### Operation

```
IF (AddressSize = 32)
    THEN Count is ECX;
ELSE IF (AddressSize = 64)
    Count is RCX;
ELSE Count is CX;
FI;

Count ← Count – 1;

IF Instruction is not LOOP
    THEN
        IF (Instruction ← LOOPE) or (Instruction ← LOOPZ)
            THEN IF (ZF = 1) and (Count ≠ 0)
                    THEN BranchCond ← 1;
                    ELSE BranchCond ← 0;
                FI;
            ELSE (Instruction = LOOPNE) or (Instruction = LOOPNZ)
```

```
                IF (ZF = 0 ) and (Count ≠ 0)
                    THEN BranchCond ← 1;
                    ELSE BranchCond ← 0;
                FI;
        FI;
    ELSE (* Instruction = LOOP *)
        IF (Count ≠ 0)
            THEN BranchCond ← 1;
            ELSE BranchCond ← 0;
        FI;
FI;

IF BranchCond = 1
    THEN
        IF OperandSize = 32
            THEN EIP ← EIP + SignExtend(DEST);
            ELSE IF OperandSize = 64
                THEN RIP ← RIP + SignExtend(DEST);
                FI;
            ELSE IF OperandSize = 16
                THEN EIP ← EIP AND 0000FFFFH;
                FI;
            ELSE IF OperandSize = (32 or 64)
                THEN IF (R/E)IP < CS.Base or (R/E)IP > CS.Limit
                    #GP; FI;
                FI;
        FI;
    ELSE
        Terminate loop and continue program execution at (R/E)IP;
FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the offset being jumped to is beyond the limits of the CS segment. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the offset being jumped to is in a non-canonical form. |
| #UD | If the LOCK prefix is used. |

## 22. MOV CR and MOV DR sections updated

In the subsections covering MOV CR and MOV DR in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A,* descriptions have been updated to correct errors and enforce consistency. Both opcode tables have been updated, information on the use of REX prefixes has been updated, and changes have been made to the exception listings.

---

## MOV—Move to/from Control Registers

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 20 /0 | MOV *r32*,CR0 | N.E. | Valid | Move CR0 to *r32*. |
| 0F 20 /0 | MOV *r64*,CR0 | Valid | N.E. | Move extended CR0 to *r64*. |
| 0F 20 /2 | MOV *r32*,CR2 | N.E. | Valid | Move CR2 to *r32*. |
| 0F 20 /2 | MOV *r64*,CR2 | Valid | N.E. | Move extended CR2 to *r64*. |
| 0F 20 /3 | MOV *r32*,CR3 | N.E. | Valid | Move CR3 to *r32*. |
| 0F 20 /3 | MOV *r64*,CR3 | Valid | N.E. | Move extended CR3 to *r64*. |
| 0F 20 /4 | MOV *r32*,CR4 | N.E. | Valid | Move CR4 to *r32*. |
| 0F 20 /4 | MOV *r64*,CR4 | Valid | N.E. | Move extended CR4 to *r64*. |
| REX.R + 0F 20 /0 | MOV *r64*,CR8 | Valid | N.E. | Move extended CR8 to *r64*.[1] |
| 0F 22 /0 | MOV CR0,*r32* | N.E. | Valid | Move *r32* to CR0. |
| 0F 22 /0 | MOV CR0,*r64* | Valid | N.E. | Move *r64* to extended CR0. |
| 0F 22 /2 | MOV CR2,*r32* | N.E. | Valid | Move *r32* to CR2. |
| 0F 22 /2 | MOV CR2,*r64* | Valid | N.E. | Move *r64* to extended CR2. |
| 0F 22 /3 | MOV CR3,*r32* | N.E. | Valid | Move *r32* to CR3. |
| 0F 22 /3 | MOV CR3,*r64* | Valid | N.E. | Move *r64* to extended CR3. |
| 0F 22 /4 | MOV CR4,*r32* | N.E. | Valid | Move *r32* to CR4. |
| 0F 22 /4 | MOV CR4,*r64* | Valid | N.E. | Move *r64* to extended CR4. |
| REX.R + 0F 22 /0 | MOV CR8,*r64* | Valid | N.E. | Move *r64* to extended CR8. |

**NOTE:**

1. MOV CR* instructions, except for MOV CR8, are serializing instructions. MOV CR8 is not architecturally defined as a serializing instruction. For more information, see Chapter 7 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.*

### Description

Moves the contents of a control register (CR0, CR2, CR3, CR4, or CR8) to a general-purpose register or the contents of a general purpose register to a control register. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of

the operand-size attribute. (See “Control Registers” in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A,* for a detailed description of the flags and fields in the control registers.) This instruction can be executed only when the current privilege level is 0.

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

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

These instructions have the following side effect:

- When writing to control register CR3, all non-global TLB entries are flushed (see “Translation Lookaside Buffers (TLBs)” in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A)*.

The following side effects are implementation specific for the Pentium 4, Intel Xeon, and P6 processor family. Software should not depend on this functionality in all Intel 64 or IA-32 processors:

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

In 64-bit mode, the instruction’s default operation size is 64 bits. The REX.R prefix must be used to access CR8. Use of REX.B permits access to additional registers (R8-R15). Use of the REX.W prefix or 66H prefix is ignored. See the summary chart at the beginning of this section for encoding data and limits.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 21 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B,* for more information about the behavior of this instruction in VMX non-root operation.

## Operation

```
DEST ← SRC;
```

## Flags Affected

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

### Protected Mode Exceptions

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

### 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). |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

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

### Compatibility Mode Exceptions

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

### 64-Bit Mode Exceptions

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

## MOV—Move to/from Debug Registers

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 21/*r* | MOV *r32*, DR0-DR7 | N.E. | Valid | Move debug register to *r32* |
| 0F 21/*r* | MOV *r64*, DR0-DR7 | Valid | N.E. | Move extended debug register to *r64*. |
| 0F 23 /*r* | MOV DR0-DR7, *r32* | N.E. | Valid | Move *r32* to debug register |
| 0F 23 /*r* | MOV DR0-DR7, *r64* | Valid | N.E. | Move *r64* to extended debug register. |

### Description

Moves the contents of a debug register (DR0, DR1, DR2, DR3, DR4, DR5, DR6, or DR7) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See Chapter 18, “Debugging and Performance Monitoring”, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A,* for a detailed description of the flags and fields in the debug registers.)

The instructions must be executed at privilege level 0 or in real-address mode.

When the debug extension (DE) flag in register CR4 is clear, these instructions operate on debug registers in a manner that is compatible with Intel386 and Intel486 processors. In this mode, references to DR4 and DR5 refer to DR6 and DR7, respectively. When the DE flag in CR4 is set, attempts to reference DR4 and DR5 result in an undefined opcode (#UD) exception. (The CR4 register was added to the IA-32 Architecture beginning with the Pentium processor.)

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

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

### Operation

```
IF ((DE = 1)  and (SRC or DEST = DR4 or DR5))
    THEN
        #UD;
    ELSE
        DEST ← SRC;

FI;
```

### Flags Affected

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

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| #UD | If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5. |
| | If the LOCK prefix is used. |
| #DB | If any debug register is accessed while the DR7.GD[bit 13] = 1. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #UD | If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5. |
| | If the LOCK prefix is used. |
| #DB | If any debug register is accessed while the DR7.GD[bit 13] = 1. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | The debug registers cannot be loaded or read when in virtual-8086 mode. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| #UD | If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5. |
| | If the LOCK prefix is used. |
| #DB | If any debug register is accessed while the DR7.GD[bit 13] = 1. |

## 23. IRET/IRETD information updated

In the subsection covering IRET/IRETD in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, the description has been updated to correct the treatment of VM. The updated text is below.

---

## IRET/IRETD—Interrupt Return

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

## Description

Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions are also used to perform a return from a nested task. (A nested task is created when a CALL instruction is used to initiate a task switch or when an interrupt or exception causes a task switch to an interrupt or exception handler.) See the section titled “Task Linking” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

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

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

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

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

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack).

As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

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

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



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

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 21 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for more information about the behavior of this instruction in VMX non-root operation.

### Operation

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

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

PROTECTED-MODE:
    IF VM = 1 (* Virtual-8086 mode: PE = 1, VM = 1 *)
        THEN
            GOTO RETURN-FROM-VIRTUAL-8086-MODE; (* PE = 1, VM = 1 *)
    FI;
    IF NT = 1
        THEN
            GOTO TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
    FI;
    IF OperandSize = 32
        THEN
```

```
                IF top 12 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                tempEIP ← Pop();
                tempCS ← Pop();
                tempEFLAGS ← Pop();
            ELSE (* OperandSize = 16 *)
                IF top 6 bytes of stack are not within stack limits
                            THEN #SS(0); FI;
                tempEIP ← Pop();
                tempCS ← Pop();
                tempEFLAGS ← Pop();
                tempEIP ← tempEIP AND FFFFH;
                tempEFLAGS ← tempEFLAGS AND FFFFH;
        FI;
        IF tempEFLAGS(VM) = 1 and CPL = 0
            THEN
                GOTO RETURN-TO-VIRTUAL-8086-MODE;
            ELSE
                GOTO PROTECTED-MODE-RETURN;
        FI;

IA-32e-MODE:
        IF NT = 1
            THEN #GP(0);
        ELSE IF OperandSize = 32
            THEN
                IF top 12 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                tempEIP ← Pop();
                tempCS ← Pop();
                tempEFLAGS ← Pop();
            ELSE IF OperandSize = 16
                THEN
                    IF top 6 bytes of stack are not within stack limits
                            THEN #SS(0); FI;
                    tempEIP ← Pop();
                    tempCS ← Pop();
                    tempEFLAGS ← Pop();
                    tempEIP ← tempEIP AND FFFFH;
                    tempEFLAGS ← tempEFLAGS AND FFFFH;
                FI;
            ELSE (* OperandSize = 64 *)
                THEN
                        tempRIP ← Pop();
                        tempCS ← Pop();
                        tempEFLAGS ← Pop();
                        tempRSP ← Pop();
                        tempSS ← Pop();
        FI;
        GOTO IA-32e-MODE-RETURN;
```

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

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

TASK-RETURN: (* PE = 1, VM = 0, NT = 1 *)
    Read segment selector in link field of current TSS;
    IF local/global bit is set to local
    or index not within GDT limits
```

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

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

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



```
            IF OperandSize = 32 or OperandSize = 64
                THEN EFLAGS(VIF, VIP) ← tempEFLAGS; FI;
    FI;
END;

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

```
        DO
            IF segment register points to data or non-conforming code segment
            and CPL > segment descriptor DPL (* Stored in hidden part of segment register *)
                THEN (* Segment register invalid *)
                    SegmentSelector ← 0; (* NULL segment selector *)
            FI;
        OD;
END;

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

### Flags Affected

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

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the return code or stack segment selector is NULL. |
| | If the return instruction pointer is not within the return code segment limit. |
| #GP(selector) | If a segment selector index is outside its descriptor table limits. |
| | If the return code segment selector RPL is greater than the CPL. |
| | If the DPL of a conforming-code segment is greater than the return code segment selector RPL. |
| | If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector. |
| | If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector. |

| | |
|---|---|
| | If the stack segment is not a writable data segment. |
| | If the stack segment selector RPL is not equal to the RPL of the return code segment selector. |
| | If the segment descriptor for a code segment does not indicate it is a code segment. |
| | If the segment selector for a TSS has its local/global bit set for local. |
| | If a TSS segment descriptor specifies that the TSS is not busy. |
| | If a TSS segment descriptor specifies that the TSS is not available. |
| #SS(0) | If the top bytes of stack are not within stack limits. |
| #NP(selector) | If the return code or stack segment is not present. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

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

### Virtual-8086 Mode Exceptions

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

### Compatibility Mode Exceptions

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

Other exceptions same as in Protected Mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If EFLAGS.NT[bit 14] = 1. |
| | If the return code segment selector is NULL. |
| | If the stack segment selector is NULL going back to compatibility mode. |
| | If the stack segment selector is NULL going back to CPL3 64-bit mode. |
| | If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode. |
| | If the return instruction pointer is not within the return code segment limit. |
| | If the return instruction pointer is non-canonical. |

| | |
|---|---|
| #GP(Selector) | If a segment selector index is outside its descriptor table limits. |
| | If a segment descriptor memory address is non-canonical. |
| | If the segment descriptor for a code segment does not indicate it is a code segment. |
| | If the proposed new code segment descriptor has both the D-bit and L-bit set. |
| | If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector. |
| | If CPL is greater than the RPL of the code segment selector. |
| | If the DPL of a conforming-code segment is greater than the return code segment selector RPL. |
| | If the stack segment is not a writable data segment. |
| | If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector. |
| | If the stack segment selector RPL is not equal to the RPL of the return code segment selector. |
| #SS(0) | If an attempt to pop a value off the stack violates the SS limit. |
| | If an attempt to pop a value off the stack causes a non-canonical address to be referenced. |
| #NP(selector) | If the return code or stack segment is not present. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled. |
| #UD | If the LOCK prefix is used. |

## 24. Table 3-1 updated

In Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, Table 3-1 has been updated. The updated table is reprinted below. See the change bars.

---

**Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro**

| byte register | | | word register | | | dword register | | | quadword register (64-Bit Mode only) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Register | REX.B | Reg Field | Register | REX.B | Reg Field | Register | REX.B | Reg Field | Register | REX.B | Reg Field |
| AL | None | 0 | AX | None | 0 | EAX | None | 0 | RAX | None | 0 |
| CL | None | 1 | CX | None | 1 | ECX | None | 1 | RCX | None | 1 |
| DL | None | 2 | DX | None | 2 | EDX | None | 2 | RDX | None | 2 |
| BL | None | 3 | BX | None | 3 | EBX | None | 3 | RBX | None | 3 |
| AH | Not encodable (N.E.) | 4 | SP | None | 4 | ESP | None | 4 | N/A | N/A | N/A |
| CH | N.E. | 5 | BP | None | 5 | EBP | None | 5 | N/A | N/A | N/A |
| DH | N.E. | 6 | SI | None | 6 | ESI | None | 6 | N/A | N/A | N/A |
| BH | N.E. | 7 | DI | None | 7 | EDI | None | 7 | N/A | N/A | N/A |
| SPL | Yes | 4 | SP | None | 4 | ESP | None | 4 | RSP | None | 4 |

**Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro (Contd.)**

| byte register | | | word register | | | dword register | | | quadword register (64-Bit Mode only) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Register | REX.B | Reg Field | Register | REX.B | Reg Field | Register | REX.B | Reg Field | Register | REX.B | Reg Field |
| BPL | Yes | 5 | BP | None | 5 | EBP | None | 5 | RBP | None | 5 |
| SIL | Yes | 6 | SI | None | 6 | ESI | None | 6 | RSI | None | 6 |
| DIL | Yes | 7 | DI | None | 7 | EDI | None | 7 | RDI | None | 7 |
| Registers R8 - R15 (see below): Available in 64-Bit Mode Only | | | | | | | | | | | |
| R8L | Yes | 0 | R8W | Yes | 0 | R8D | Yes | 0 | R8 | Yes | 0 |
| R9L | Yes | 1 | R9W | Yes | 1 | R9D | Yes | 1 | R9 | Yes | 1 |
| R10L | Yes | 2 | R10W | Yes | 2 | R10D | Yes | 2 | R10 | Yes | 2 |
| R11L | Yes | 3 | R11W | Yes | 3 | R11D | Yes | 3 | R11 | Yes | 3 |
| R12L | Yes | 4 | R12W | Yes | 4 | R12D | Yes | 4 | R12 | Yes | 4 |
| R13L | Yes | 5 | R13W | Yes | 5 | R13D | Yes | 5 | R13 | Yes | 5 |
| R14L | Yes | 6 | R14W | Yes | 6 | R14D | Yes | 6 | R14 | Yes | 6 |
| R15L | Yes | 7 | R15W | Yes | 7 | R15D | Yes | 7 | R15 | Yes | 7 |

## 25. MONITOR/MWAIT sections updated

In Chapter 3, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, for the subsections covering MONITOR and MWAIT; descriptions have been updated to correct errors and enforce consistency. The focus is on the exception sections. Both subsections are reprinted below. See the change bars.

---

### MONITOR—Set Up Monitor Address

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| OF 01 *C8* | MONITOR | Valid | Valid | Sets up a linear address range to be monitored by hardware and activates the monitor. The address range should be a write-back memory caching type. The default address is DS:EAX. |

#### Description

The MONITOR instruction arms address monitoring hardware using an address specified in EAX (the address range that the monitoring hardware checks for store operations can be determined by using CPUID). A store to an address within the specified address range triggers the monitoring hardware. The state of monitor hardware is used by MWAIT.

The content of EAX is an effective address. By default, the DS segment is used to create a linear address that is monitored. Segment overrides can be used.

ECX and EDX are also used. They communicate other information to MONITOR. ECX specifies optional extensions. EDX specifies optional hints; it does not change the architectural behavior of the instruction. For the Pentium 4 processor (family 15, model 3), no

extensions or hints are defined. Undefined hints in EDX are ignored by the processor; undefined extensions in ECX raises a general protection fault.

The address range must use memory of the write-back type. Only write-back memory will correctly trigger the monitoring hardware. Additional information on determining what address range to use in order to prevent false wake-ups is described in Chapter 7, *Multiple-Processor Management* of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.*

The MONITOR instruction is ordered as a load operation with respect to other memory transactions. The instruction can be used at all privilege levels and is subject to the permission checking and faults associated with a byte load. Like a load, MONITOR sets the A-bit but not the D-bit in page tables.

The MONITOR CPUID feature flag (ECX bit 3; CPUID executed EAX = 1) indicates the availability of MONITOR and MWAIT in the processor. When set, the unconditional execution of MONITOR is supported at privilege levels 0; conditional execution is supported at privilege levels 1 through 3 (test for the appropriate support before unconditional use). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLES MSR; disabling MONITOR clears the CPUID feature flag and causes execution to generate an illegal opcode exception.

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

## Operation

MONITOR sets up an address range for the monitor hardware using the content of EAX as an effective address and puts the monitor hardware in armed state. Always use memory of the write-back caching type. A store to the specified address range will trigger the monitor hardware. The content of ECX and EDX are used to communicate other information to the monitor hardware.

## Intel C/C++ Compiler Intrinsic Equivalent

MONITOR void _mm_monitor(void const *p, unsigned extensions,unsigned hints)

### Numeric Exceptions

None

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| | If ECX ¼ 0. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | For a page fault. |
| #UD | If CPUID.01H:ECX.MONITOR[bit 3] = 0. |
| | If current privilege level is not 0. |

### Real Address Mode Exceptions

| | |
|---|---|
| #GP | If any part of the operand in the CS, DS, ES, FS, or GS segment lies outside of the effective address space from 0 to FFFFH. |
| | If ECX ¼ 0. |
| #SS | If any part of the operand in the SS segment lies outside of the effective address space from 0 to FFFFH. |
| #UD | If CPUID.01H:ECX.MONITOR[bit 3] = 0. |

### Virtual 8086 Mode Exceptions

| | |
|---|---|
| #UD | The MONITOR instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1). |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the linear address of the operand in the CS, DS, ES, FS, or GS segment is in a non-canonical form.<br>If RCX ¼ 0. |
| #SS(0) | If the linear address of the operand in the SS segment is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #UD | If the current privilege level is not 0.<br>If CPUID.01H:ECX.MONITOR[bit 3] = 0. |

## MWAIT—Monitor Wait

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| OF 01 *C9* | MWAIT | Valid | Valid | A hint that allow the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events. |

### Description

MWAIT instruction provides hints to allow the processor to enter an implementation-dependent optimized state. There are two principal targeted usages: address-range monitor and advanced power management. Both usages of MWAIT require the use of the MONITOR instruction.

A CPUID feature flag (ECX bit 3; CPUID executed EAX = 1) indicates the availability of MONITOR and MWAIT in the processor. When set, the unconditional execution of MWAIT is supported at privilege levels 0; conditional execution is supported at privilege levels 1 through 3 (test for the appropriate support before unconditional use). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLES MSR; disabling MWAIT clears the CPUID feature flag and causes execution to generate an illegal opcode exception.

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

### MWAIT for Address Range Monitoring

For address-range monitoring, the MWAIT instruction operates with the MONITOR instruction. The two instructions allow the definition of an address at which to wait (MONITOR) and a implementation-dependent-optimized operation to commence at the wait address (MWAIT). The execution of MWAIT is a hint to the processor that it can enter an implementation-dependent-optimized state while waiting for an event or a store operation to the address range armed by MONITOR.

ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. For Pentium 4 processors (CPUID signature family 15 and model 3), non-zero values for EAX and ECX are reserved.

A store to the address range armed by the MONITOR instruction, an interrupt, an NMI or SMI, a debug exception, a machine check exception, the BINIT# signal, the INIT# signal, or the RESET# signal will exit the implementation-dependent-optimized state. Note that an interrupt will cause the processor to exit only if the state was entered with interrupts enabled.

If a store to the address range causes the processor to exit, execution will resume at the instruction following the MWAIT instruction. If an interrupt (including NMI) caused the processor to exit the implementation-dependent-optimized state, the processor will exit the state and handle the interrupt. If an SMI caused the processor to exit the implementation-dependent-optimized state, execution will resume at the instruction following MWAIT after handling of the SMI. Unlike the HLT instruction, the MWAIT instruction does not support a restart at the MWAIT instruction. There may also be other implementation-dependent events or time-outs that may take the processor out of the implementation-dependent-optimized state and resume execution at the instruction following the MWAIT.

If the preceding MONITOR instruction did not successfully arm an address range or if the MONITOR instruction has not been executed prior to executing MWAIT, then the processor will not enter the implementation-dependent-optimized state. Execution will resume at the instruction following the MWAIT.

### MWAIT for Power Management

MWAIT accepts a hint and optional extension to the processor that it can enter a specified target C state while waiting for an event or a store operation to the address range armed by MONITOR. Support for MWAIT extensions for power management is indicated by CPUID.05H.ECX[0] reporting 1.

EAX and ECX will be used to communicate the additional information to the MWAIT instruction, such as the kind of optimized state the processor should enter. ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. A given processor implementation may choose to ignore the hint and continue executing the next instruction. Future processor implementations may implement several optimized “waiting” states and will select among those states based on the hint argument.

Table 3-62 describes the meaning of ECX and EAX registers for MWAIT extensions.

**Table 3-62. MWAIT Extension Register (ECX)**

| Bits | Description |
|---|---|
| 0 | Treat Interrupt as break-event, even when interrupts are disabled (EFLAGS.IF=0) |
| 31: 1 | Reserved |

**Table 3-63. MWAIT Hints Register (EAX)**

| Bits | Description |
|---|---|
| 3 : 0 | Sub C-state within a C-state, indicated by bits [7:4] |
| 7 : 4 | Target C-state*<br>Value of 0 means C1; 1 means C2 and so on<br>Value of 01111B means C0<br><br>Note: Target C states for MWAIT entensions are processor-specific C-states, not ACPI C-states |
| 31: 8 | Reserved |

Note that if MWAIT is used to enter any of the C-states that are numerically higher than C1, a store to the address range armed by the MONITOR instruction will cause the processor to exit MWAIT only if the store was originated by other processor agents. A store from non-processor agent may not cause the processor to exit MWAIT in such cases

For additional details of MWAIT extensions, see Chapter 13, “Power and Thermal Management,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.*

## Operation

```
(* MWAIT takes the argument in EAX as a hint extension and is architected to take the argument in ECX as an instruction extension MWAIT EAX, ECX *)
{
WHILE (! (“Monitor Hardware is in armed state”)) {
    implementation_dependent_optimized_state(EAX, ECX); }
Set the state of Monitor Hardware as triggered;
}
```

## Intel C/C++ Compiler Intrinsic Equivalent

MWAIT  void _mm_mwait(unsigned extensions, unsigned hints)

## Example

MONITOR/MWAIT instruction pair must be coded in the same loop because execution of the MWAIT instruction will trigger the monitor hardware. It is not a proper usage to execute MONITOR once and then execute MWAIT in a loop. Setting up MONITOR without executing MWAIT has no adverse effects.

Typically the MONITOR/MWAIT pair is used in a sequence, such as:

```
EAX = Logical Address(Trigger)
ECX = 0 (*Hints *)
EDX = 0 (* Hints *)

IF ( !trigger_store_happened) {
    MONITOR EAX, ECX, EDX
    IF ( !trigger_store_happened ) {
        MWAIT EAX, ECX
    }
}
```

The above code sequence makes sure that a triggering store does not happen between the first check of the trigger and the execution of the monitor instruction. Without the second check that triggering store would go un-noticed. Typical usage of MONITOR and MWAIT would have the above code sequence within a loop.

### Numeric Exceptions

None

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| | If ECX = 0. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | For a page fault. |
| #UD | If CPUID.01H:ECX.MONITOR[bit 3] = 0. |
| | If current privilege level is not 0. |

### Real Address Mode Exceptions

| | |
|---|---|
| #GP | If any part of the operand in the CS, DS, ES, FS, or GS segment lies outside of the effective address space from 0 to FFFFH. |
| | If ECX ≠ 0. |
| #SS | If any part of the operand in the SS segment lies outside of the effective address space from 0 to FFFFH. |
| #UD | If CPUID.01H:ECX.MONITOR[bit 3] = 0. |

### Virtual 8086 Mode Exceptions

| | |
|---|---|
| #UD | The MONITOR instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1). |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the linear address of the operand in the CS, DS, ES, FS, or GS segment is in a non-canonical form. |
| | If RCX ≠ 0. |
| #SS(0) | If the linear address of the operand in the SS segment is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #UD | If the current privilege level is not 0. |
| | If CPUID.01H:ECX.MONITOR[bit 3] = 0. |



## 26. Note on VMX added to microcode update information

In Section 26.4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, a note has been added.

---

## 26.4 MICROCODE UPDATE FACILITY

The microcode code update facility may be invoked at various points during the operation of a platform. Typically, the BIOS invokes the facility on all processors during the BIOS boot process. This is sufficient to boot the BIOS and operating system. As a microcode update more current than the system BIOS may be available, system software should provide another mechanism for invoking the microcode update facility. The implications of the microcode update mechanism on the design of the VMM are described in this section.

**NOTE**

Microcode updates must not be performed during VMX non-root operation. Updates performed in VMX non-root operation may result in unpredictable system behavior.