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

Documentation Changes

June 2010

Document Number: 252046-028

Contents

Revision History

Revision	Description	Date
-001	• Initial release	November 2002
-002	• Added 1-10 Documentation Changes. • Removed old Documentation Changes items that already have been incorporated in the published Software Developer’s manual	December 2002
-003	• Added 9 -17 Documentation Changes. • Removed Documentation Change #6 - References to bits Gen and Len Deleted. • Removed Documentation Change #4 - VIF Information Added to CLI Discussion	February 2003
-004	• Removed Documentation changes 1-17. • Added Documentation changes 1-24.	June 2003
-005	• Removed Documentation Changes 1-24. • Added Documentation Changes 1-15.	September 2003
-006	• Added Documentation Changes 16- 34.	November 2003
-007	• Updated Documentation changes 14, 16, 17, and 28. • Added Documentation Changes 35-45.	January 2004
-008	• Removed Documentation Changes 1-45. • Added Documentation Changes 1-5.	March 2004
-009	• Added Documentation Changes 7-27.	May 2004
-010	• Removed Documentation Changes 1-27. • Added Documentation Changes 1.	August 2004
-011	• Added Documentation Changes 2-28.	November 2004
-012	• Removed Documentation Changes 1-28. • Added Documentation Changes 1-16.	March 2005
-013	• Updated title. • There are no Documentation Changes for this revision of the document.	July 2005
-014	• Added Documentation Changes 1-21.	September 2005
-015	• Removed Documentation Changes 1-21. • Added Documentation Changes 1-20.	March 9, 2006
-016	• Added Documentation changes 21-23.	March 27, 2006
-017	• Removed Documentation Changes 1-23. • Added Documentation Changes 1-36.	September 2006
-018	• Added Documentation Changes 37-42.	October 2006
-019	• Removed Documentation Changes 1-42. • Added Documentation Changes 1-19.	March 2007
-020	• Added Documentation Changes 20-27.	May 2007
-021	• Removed Documentation Changes 1-27. • Added Documentation Changes 1-6	November 2007
-022	• Removed Documentation Changes 1-6 • Added Documentation Changes 1-6	August 2008
-023	• Removed Documentation Changes 1-6 • Added Documentation Changes 1-21	March 2009

Revision	Description	Date
-024	• Removed Documentation Changes 1-21 • Added Documentation Changes 1-16	June 2009
-025	• Removed Documentation Changes 1-16 • Added Documentation Changes 1-18	September 2009
-026	• Removed Documentation Changes 1-18 • Added Documentation Changes 1-15	December 2009
-027	• Removed Documentation Changes 1-15 • Added Documentation Changes 1-24	March 2010
-028	• Removed Documentation Changes 1-24 • Added Documentation Changes 1-29	June 2010

§

Preface

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

Affected Documents

Document Title	Document Number/Location
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, Part 1	253668
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2	253669

Nomenclature

Documentation Changes include typos, errors, or omissions from the current published specifications. These will be incorporated in any new release of the specification.

Summary Tables 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 Tables

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

Documentation Changes (Sheet 1 of 2)

No.	DOCUMENTATION CHANGES
1	Updates to Chapter 7, Volume 1
2	Updates to Chapter 9, Volume 1
3	Updates to Chapter 11, Volume 1
4	Updates to Chapter 12, Volume 1
5	Updates to Chapter 13, Volume 1
6	Updates to Appendix D, Volume 1
7	Updates to Chapter 2, Volume 2A
8	Updates to Chapter 3, Volume 2A
9	Updates to Chapter 4, Volume 2B
10	Updates to Chapter 5, Volume 2B
11	Updates to Chapter 6, Volume 2B
12	Updates to Appendix A, Volume 2B
13	Updates to Chapter 3, Volume 3A
14	Updates to Chapter 6, Volume 3A
15	Updates to Chapter 8, Volume 3A
16	Updates to Chapter 10, Volume 3A
17	Updates to Chapter 14, Volume 3A
18	Updates to Chapter 16, Volume 3A
19	Updates to Chapter 20, Volume 3B
20	Updates to Chapter 22, Volume 3B
21	Updates to Chapter 23, Volume 3B
22	Updates to Chapter 24, Volume 3B
23	Updates to Chapter 25, Volume 3B
24	Updates to Chapter 26, Volume 3B
25	Updates to Chapter 29, Volume 3B
26	Updates to Chapter 30, Volume 3B

Documentation Changes (Sheet 2 of 2)

No.	DOCUMENTATION CHANGES
27	Updates to Appendix B, Volume 3B
28	Updates to Appendix E, Volume 3B
29	Updates to Appendix H, Volume 3B

Documentation Changes

1. Updates to Chapter 7, Volume 1

Change bars show changes to Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1:* Basic Architecture.

...

7.3.8.2 Conditional Transfer Instructions

The conditional transfer instructions execute jumps or loops that transfer program control to another instruction in the instruction stream if specified conditions are met. The conditions for control transfer are specified with a set of condition codes that define various states of the status flags (CF, ZF, OF, PF, and SF) in the EFLAGS register.

...

Jump if zero instructions — The JECXZ (jump if ECX zero) instruction jumps to the location specified in the destination operand if the ECX register contains the value zero. This instruction can be used in combination with a loop instruction (LOOP, LOOPE, LOOPZ, LOOPNE, or LOOPNZ) to test the ECX register prior to beginning a loop. As described in "Loop instructions on page 7-24, the loop instructions decrement the contents of the ECX register before testing for zero. If the value in the ECX register is zero initially, it will be decremented to FFFFFFFFH on the first loop instruction, causing the loop to be executed 2^{32} times. To prevent this problem, a JECXZ instruction can be inserted at the beginning of the code block for the loop, causing a jump out the loop if the EAX register count is initially zero. When used with repeated string scan and compare instructions, the JECXZ instruction can determine whether the loop terminated because the count reached zero or because the scan or compare conditions were satisfied.

...

7.3.14.2 EFLAGS Transfer Instructions

The EFLAGS transfer instructions allow groups of flags in the EFLAGS register to be copied to a register or memory or be loaded from a register or memory.

The LAHF (load AH from flags) and SAHF (store AH into flags) instructions operate on five of the EFLAGS status flags (SF, ZF, AF, PF, and CF). The LAHF instruction copies the status flags to bits 7, 6, 4, 2, and 0 of the AH register, respectively. The contents of the remaining bits in the register (bits 5, 3, and 1) are unaffected, and the contents of the EFLAGS register remain unchanged. The SAHF instruction copies bits 7, 6, 4, 2, and 0 from the AH register into the SF, ZF, AF, PF, and CF flags, respectively in the EFLAGS register.

The PUSHF (push flags), PUSHFD (push flags double), POPF (pop flags), and POPFD (pop flags double) instructions copy the flags in the EFLAGS register to and from the stack. The PUSHF instruction pushes the lower word of the EFLAGS register onto the stack (see Figure 7-11). The PUSHFD instruction pushes the entire EFLAGS register onto the stack (with the RF and VM flags read as clear).

2. Updates to Chapter 9, Volume 1

Change bars show changes to Chapter 9 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1:* Basic Architecture.

9.4 MMX INSTRUCTIONS

The MMX instruction set consists of 47 instructions, grouped into the following categories:

- Data transfer
- Arithmetic
- Comparison
- Conversion
- Unpacking
- Logical
- Shift
- Empty MMX state instruction (EMMS)

Table 9-2 gives a summary of the instructions in the MMX instruction set. The following sections give a brief overview of the instructions within each group.

NOTES

The MMX instructions described in this chapter are those instructions that are available in an IA-32 processor when CPUID.01H:EDX.MMX[bit 23] = 1.

Section 10.4.4, "SSE 64-Bit SIMD Integer Instructions," and Section 11.4.2, "SSE2 64-Bit and 128-Bit SIMD Integer Instructions," list additional instructions included with SSE/SSE2 extensions that operate on the MMX registers but are not considered part of the MMX instruction set.

...

3. Updates to Chapter 11, Volume 1

Change bars show changes to Chapter 11 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1:* Basic Architecture.

...

11.6.4 Initialization of SSE/SSE2 Extensions

The SSE and SSE2 state is contained in the XMM and MXCSR registers. Upon a hardware reset of the processor, this state is initialized as follows (see Table 11-2):

- All SIMD floating-point exceptions are masked (bits 7 through 12 of the MXCSR register is set to 1).

- All SIMD floating-point exception flags are cleared (bits 0 through 5 of the MXCSR register is set to 0).
- The rounding control is set to round-nearest (bits 13 and 14 of the MXCSR register are set to 00B).
- The flush-to-zero mode is disabled (bit 15 of the MXCSR register is set to 0).
- The denormals-are-zeros mode is disabled (bit 6 of the MXCSR register is set to 0). If the denormals-are-zeros mode is not supported, this bit is reserved and will be set to 0 on initialization.
- Each of the XMM registers is cleared (set to all zeros).

...

4. Updates to Chapter 12, Volume 1

Change bars show changes to Chapter 12 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1:* Basic Architecture.

...

12.12.3 Checking for SSE4.2 Support

Before an application attempts to use the following SSE4.2 instructions: PCMPESTRI/ PCMPESTRM/PCMPISTRI/PCMPISTRM, PCMPGTQ; the application should follow the steps illustrated in Section 11.6.2, "Checking for SSE/SSE2 Support." Next, use the additional step provided below:

Check that the processor supports SSE4.2 (if CPUID.01H:ECX.SSE4_2[bit 20] = 1), SSE4.1 (if CPUID.01H:ECX.SSE4_1[bit 19] = 1), and SSSE3 (if CPUID.01H:ECX.SSSE3[bit 9] = 1).

Before an application attempts to use the CRC32 instruction, it must check that the processor supports SSE4.2 (if CPUID.01H:ECX.SSE4_2[bit 20] = 1).

Before an application attempts to use the POPCNT instruction, it must check that the processor supports SSE4.2 (if CPUID.01H:ECX.SSE4_2[bit 20] = 1) and POPCNT (if CPUID.01H:ECX.POPCNT[bit 23] = 1).

...

5. Updates to Chapter 13, Volume 1

Change bars show changes to Chapter 13 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1:* Basic Architecture.

...

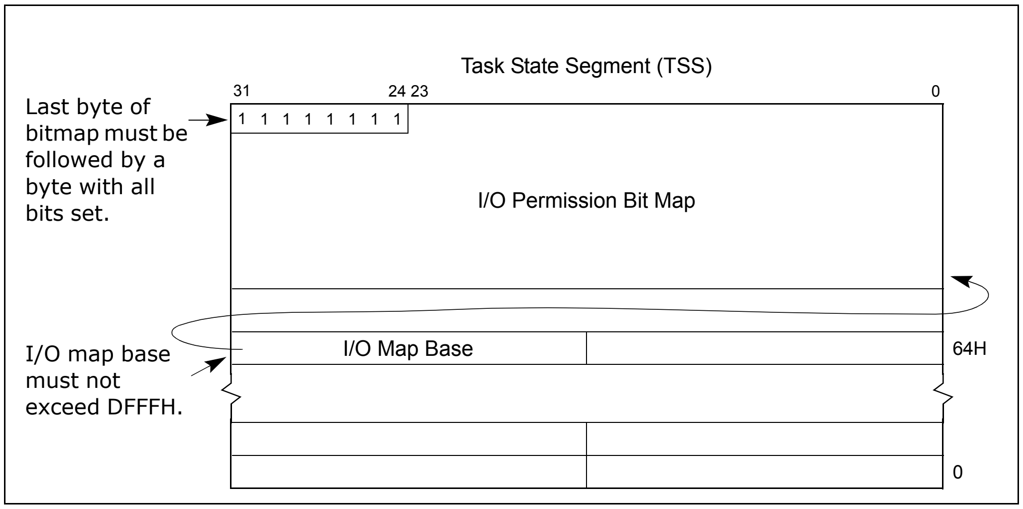
Task State Segment (TSS)
31
24 23
0
1 1 1 1 1 1 1 1
Last byte of bitmap must be followed by a byte with all bits set.
I/O Permission Bit Map
I/O map base must not exceed DFFFH.
I/O Map Base
64H
0

Figure 13-2 I/O Permission Bit Map

...

6. Updates to Appendix D, Volume 1

Change bars show changes to Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1:* Basic Architecture.

...

Example D-4. Reduced-Latency Exception Handler

```
SAVE_ENVIRONMENTPROC
;
;SAVE REGISTERS, ALLOCATE STACK SPACE FOR x87 FPU ENVIRONMENT
    PUSH      EBP
    .
    .
    MOV       EBP, ESP
    SUB       ESP, 28   ;ALLOCATES 28 BYTES (32-bit PROTECTED MODE SIZE)
;SAVE ENVIRONMENT, RESTORE INTERRUPT ENABLE FLAG (IF)
    FNSTENV        [EBP - 28]
    PUSH           [EBP + OFFSET_TO_EFLAGS]  ; COPY OLD EFLAGS TO STACK TOP
    POPFD     ;RESTORE IF TO VALUE BEFORE x87 FPU EXCEPTION
;
;APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE
;
;CLEAR EXCEPTION FLAGS IN STATUS WORD (WHICH IS IN MEMORY)
    MOV       BYTE PTR [EBP-24], 0H
    FLDENV    [EBP-28]
;DE-ALLOCATE STACK SPACE, RESTORE REGISTERS
    MOV       ESP, EBP
    .
    .
    POP     EBP
;
;RETURN TO INTERRUPTED CALCULATION
    IRETD
    SAVE_ENVIRONMENT ENDP
;RESTORE MODIFIED ENVIRONMENT IMAGE
```

...

Case #2: x87 FPU State Swap with Discarded Numeric Exception

Again, assume two threads A and B, both using the floating-point unit. Let A be the thread to have most recently executed a floating-point instruction, but this time let there be a pending numeric exception. Let B be the currently executing thread. When B starts to execute a floating-point instruction the instruction will fault with the DNA exception and enter the DNA handler. (If both numeric and DNA exceptions are pending, the DNA exception takes precedence, in order to support handling the numeric exception in its own context.)

When the FNSAVE starts, it will trigger an interrupt via FERR# because of the pending numeric exception. After some system dependent delay, the numeric exception handler is entered. It may be entered before the FNSAVE starts to execute, or it may be entered shortly after execution of the FNSAVE. Since the x87 FPU Owner is the kernel, the numeric exception handler simply exits, discarding the exception. The DNA handler resumes execution, completing the FNSAVE of the old floating-point context of thread A and the FRSTOR of the floating-point context for thread B.

Thread A eventually gets an opportunity to handle the exception that was discarded during the task switch. After some time, thread B is suspended, and thread A resumes execution. When thread A starts to execute an floating-point instruction, once again the DNA exception handler is entered. B's x87 FPU state is saved with FNSAVE, and A's x87 FPU state is restored with FRSTOR. Note that in restoring the x87 FPU state from A's save area, the pending numeric exception flags are reloaded into the floating-point status word. Now when the DNA exception handler returns, thread A resumes execution of the faulting floating-point instruction just long enough to immediately generate a numeric exception, which now gets handled in the normal way. The net result is that the task switch and resulting x87 FPU state swap via the DNA exception handler causes an extra numeric exception which can be safely discarded.

...

D.4.2 Changes with Intel486, Pentium and Pentium Pro Processors with CR0.NE[bit 5] = 1

With these three generations of the IA-32 architecture, more enhancements and speedup features have been added to the corresponding x87 FPUs. Also, the x87 FPU is now built into the same chip as the processor, which allows further increases in the speed at which the x87 FPU can operate as part of the integrated system. This also means that the native mode of x87 FPU exception handling, selected by setting bit NE of register CR0 to 1, is now entirely internal.

If an unmasked exception occurs during an x87 FPU instruction, the x87 FPU records the exception internally, and triggers the exception handler through interrupt 16 immediately before execution of the next WAIT or x87 FPU instruction (except for no-wait instructions, which will be executed as described in Section D.4.1, "Origin with the Intel 286 and Intel 287, and Intel386 and Intel 387 Processors").

An unmasked numerical exception causes the FERR# output to be activated even with NE = 1, and at exactly the same point in the program flow as it would have been asserted if NE were zero. However, the system would not connect FERR# to a PIC to generate INTR when operating in the native, internal mode. (If the hardware of a system has FERR# connected to trigger IRQ13 in order to support MS-DOS, but an operating system using the native mode is actually running the system, it is the operating system's responsibility to make sure that IRQ13 is not enabled in the slave PIC.) With this configuration a system is immune to the problem discussed in Section D.2.1.3, "No-Wait x87 FPU Instructions Can Get x87 FPU Interrupt in Window," where for Intel486 and Pentium processors a no-wait x87 FPU instruction can get an x87 FPU exception.

...

7. Updates to Chapter 2, Volume 2A

Change bars show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A:* Instruction Set Reference, A-M.

...

Table 2-5 Special Cases of REX Encodings

ModR/M or SIB	Sub-field Encodings	Compatibility Mode Operation	Compatibility Mode Implications	Additional Implications
ModR/M Byte	mod != 11	SIB byte present.	SIB byte required for ESP-based addressing.	REX prefix adds a fourth bit (b) which is not decoded (don't care). SIB byte also required for R12-based addressing.
	r/m = b*100(ESP)			
ModR/M Byte	mod = 0	Base register not used.	EBP without a displacement must be done using mod = 01 with displacement of 0.	REX prefix adds a fourth bit (b) which is not decoded (don't care). Using RBP or R13 without displacement must be done using mod = 01 with a displacement of 0.
	r/m = b*101(EBP)			
SIB Byte	index = 0100(ESP)	Index register not used.	ESP cannot be used as an index register.	REX prefix adds a fourth bit (b) which is decoded. There are no additional implications. The expanded index field allows distinguishing RSP from R12, therefore R12 can be used as an index.
SIB Byte	base = 0101(EBP)	Base register is unused if mod = 0.	Base register depends on mod encoding.	REX prefix adds a fourth bit (b) which is not decoded. This requires explicit displacement to be used with EBP/RBP or R13.

NOTES:

* Don't care about value of REX.B

...

Table 2-7 RIP-Relative Addressing

ModR/M and SIB Sub-field Encodings		Compatibility Mode Operation	64-bit Mode Operation	Additional Implications in 64-bit mode
ModR/M Byte	mod = 00	Disp32	RIP + Disp32	Must use SIB form with normal (zero-based) displacement addressing
	r/m = 101 (none)			
SIB Byte	base = 101 (none)	if mod = 00, Disp32	Same as legacy	None
	index = 100 (none)			
	scale = 0, 1, 2, 4			

...

8. Updates to Chapter 3, Volume 2A

Change bars show changes to Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A:* Instruction Set Reference, A-M.

...

BLENDPD — Blend Packed Double Precision Floating-Point Values

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
66 0F 3A 0D /r ib	BLENDPD *xmm1, xmm2/m128, imm8*	A	Valid	Valid	Select packed DP-FP values from *xmm1* and *xmm2/ m128* from mask specified in imm8 and store the values into *xmm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA

Description

Packed double-precision floating-point values from the source operand (second operand) are conditionally copied to the destination operand depending on the mask bits in the immediate operand. The mask bits are bits [1:0] of the immediate byte (third operand). Each mask bit corresponds to a quadword element in a 128-bit operand.

If a mask bit is "1", then the corresponding quadword in the source operand is copied to the destination, else the quadword element in the destination operand is left unchanged.

Operation

```
IF (imm8[0] = 1)
    THEN DEST[63:0] ← SRC[63:0];
    ELSE DEST[63:0] ← DEST[63:0]; FI;
IF (imm8[1] = 1)
    THEN DEST[127:64] ← SRC[127:64];
    ELSE DEST[127:64] ← DEST[127:64]; FI;
```

...

BLENDPS — Blend Packed Single Precision Floating-Point Values

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
66 0F 3A 0C /r ib	BLENDPS *xmm1, xmm2/m128, imm8*	A	Valid	Valid	Select packed single precision floating-point values from *xmm1* and *xmm2/m128* from mask specified in *imm8* and store the values into *xmm1.*

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA

Description

Packed single-precision floating-point values from the source operand (second operand) are conditionally copied to the destination operand (first operand) depending on the mask bits in the immediate operand. The mask bits are bits [3:0] of the immediate byte (third operand). Each mask bit corresponds to a dword element in a 128-bit operand.

If a mask bit is “1", then the corresponding dword in the source operand is copied to the destination, else the dword element in the destination operand is left unchanged.

Operation

```
IF (imm8[0] = 1)
    THEN DEST[31:0] ← SRC[31:0];
    ELSE DEST[31:0] ← DEST[31:0]; FI;
IF (imm8[1] = 1)
    THEN DEST[63:32] ← SRC[63:32];
    ELSE DEST[63:32] ← DEST[63:32]; FI;
IF (imm8[2] = 1)
    THEN DEST[95:64] ← SRC[95:64];
    ELSE DEST[95:64] ← DEST[95:64]; FI;
IF (imm8[3] = 1)
    THEN DEST[127:96] ← SRC[127:96];
    ELSE DEST[127:96] ← DEST[127:96]; FI;
```

...

BLENDVPD — Variable Blend Packed Double Precision Floating-Point Values

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
66 0F 38 15 /r	BLENDVPD *xmm1, xmm2/m128* , *<XMM0>*	A	Valid	Valid	Select packed DP FP values from *xmm1* and *xmm2* from mask specified in *XMM0* and store the values in *xmm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	implicit XMM0	NA

Description

Packed double-precision floating-point values from the source operand (second argument) are conditionally copied to the destination operand (first argument) depending on the mask bits in the implicit third register argument, XMM0. The mask bits are the most significant bit in each qword element of XMM0. Each mask bit corresponds to a quadword element in a 128-bit operand.

If a mask bit is "1", then the corresponding quadword element in the source operand is copied to the destination, else the quadword element in the destination operand is left unchanged.

The register assignment of the third operand is defined to be the architectural register XMM0.

Operation

```
MASK ← XMM0;
IF (MASK[63] = 1)
    THEN DEST[63:0] ← SRC[63:0];
    ELSE DEST[63:0] ← DEST[63:0]; FI;
IF (MASK[127] = 1)
    THEN DEST[127:64] ← SRC[127:64];
    ELSE DEST[127:64] ← DEST[127:64]; FI;
```

...

BLENDVPS — Variable Blend Packed Single Precision Floating-Point Values

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
66 0F 38 14 /r	BLENDVPS *xmm1, xmm2/m128, <XMM0>*	A	Valid	Valid	Select packed single precision floating-point values from *xmm1* and *xmm2/m128* from mask specified in *XMM0* and store the values into *xmm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	implicit XMM0	NA

Description

Packed single-precision floating-point values from the source operand (second argument) are conditionally written to the destination operand (first argument) depending on the mask bits in the third register argument. The mask bits are the most significant bit in each dword element of XMM0. Each mask bit corresponds to a dword element in a 128-bit operand.

If a mask bit is "1", then the corresponding dword element in the source operand is copied to the destination, else the dword element in the destination operand is left unchanged.

The register assignment of the third operand is defined to be the architectural register XMM0.

Operation

```
MASK ← XMM0;
IF (MASK[31] = 1)
    THEN DEST[31:0] ← SRC[31:0];
    ELSE DEST[31:0] ← DEST[31:0]); FI;
IF (MASK[63] = 1)
    THEN DEST[63:32] ← SRC[63:32]);
    ELSE DEST[63:32] ← DEST[63:32]); FI;
IF (MASK[95] = 1)
    THEN DEST[95:64] ← SRC[95:64]);
    ELSE DEST[95:64] ← DEST[95:64]); FI;
IF (MASK[127] = 1)
    THEN DEST[127:96] ← SRC[127:96]);
    ELSE DEST[127:96] ← DEST[127:96]); FI;
```

...

CMOV*cc*—Conditional Move

...

Operation

```
temp ← SRC
IF condition TRUE
    THEN
        DEST ← temp;
    FI;
ELSE
    IF (OperandSize = 32 and IA-32e mode active)
        THEN
            DEST[63:32] ← 0;
    FI;
```

FI;

...

CPUID—CPU Identification

...

Table 3-12 Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
...		
	Thermal and Power Management Leaf	
06H	EAX	Bit 00: Digital temperature sensor is supported if set Bit 01: Intel Turbo Boost Technology Available (see description of IA32_MISC_ENABLES[38]). Bit 02: ARAT. APIC-Timer-always-running feature is supported if set. Bit 03: Reserved Bit 04: PLN. Power limit notification controls are supported if set. Bit 05: ECMD. Clock modulation duty cycle extension is supported if set. Bit 06: PTM. Package thermal management is supported if set. Bits 31 - 07: Reserved
	EBX	Bits 03 - 00: Number of Interrupt Thresholds in Digital Thermal Sensor Bits 31 - 04: Reserved
	ECX	Bit 00: Hardware Coordination Feedback Capability (Presence of IA32_MPERF and IA32_APERF). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of expected processor performance at frequency specified in CPUID Brand String Bits 02 - 01: Reserved = 0 Bit 03: The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H) Bits 31 - 04: Reserved = 0
	EDX	Reserved = 0
...		

...

INPUT EAX = 0: Returns CPUID's Highest Value for Basic Processor Information and the Vendor Identification String

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register (see Table 3-13) and is processor specific.

A vendor identification string is also returned in EBX, EDX, and ECX. For Intel processors, the string is "GenuineIntel" and is expressed:

```
EBX ← 756e6547h (* "Genu", with G in the low eight bits of BL *)
EDX ← 49656e69h (* "ineI", with i in the low eight bits of DL *)
ECX ← 6c65746eh (* "ntel", with n in the low eight bits of CL *)
```

INPUT EAX = 80000000H: Returns CPUID's Highest Value for Extended Processor Information

When CPUID executes with EAX set to 80000000H, the processor returns the highest value the processor recognizes for returning extended processor information.

...

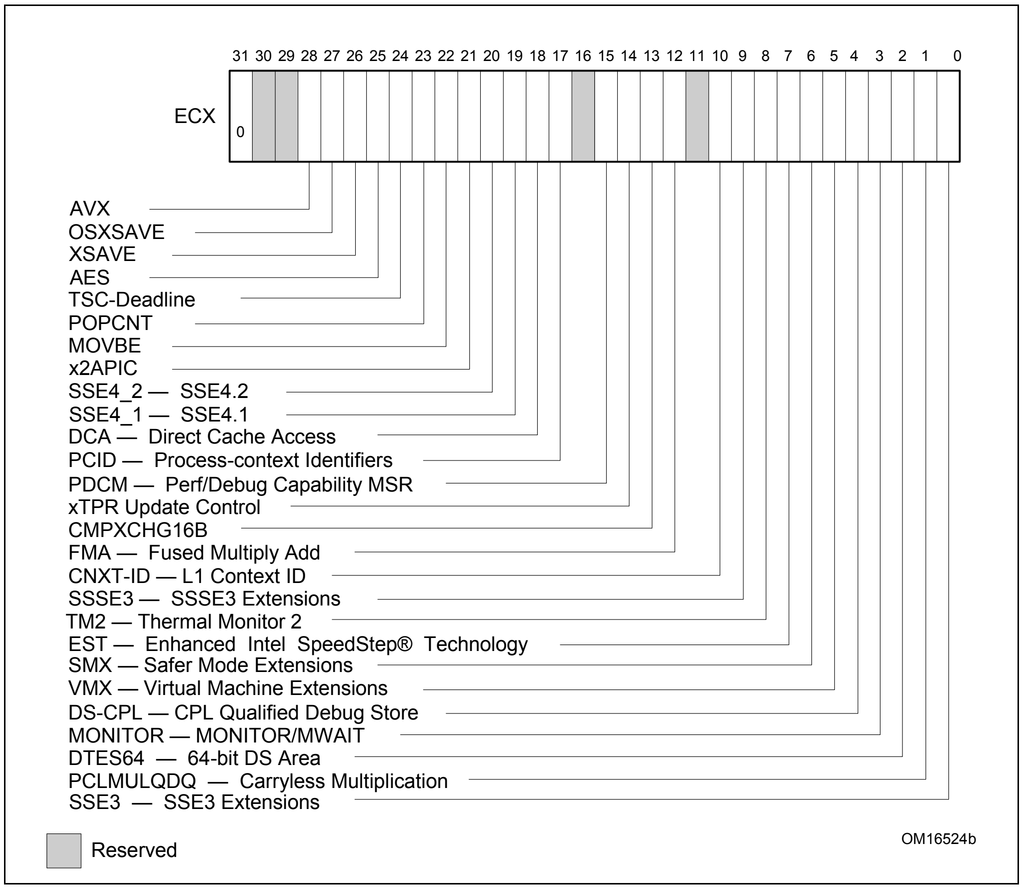

Figure 3-6 Feature Information Returned in the ECX Register

Table 3-15 Feature Information Returned in the ECX Register

Bit #	Mnemonic	Description
0	SSE3	**Streaming SIMD Extensions 3 (SSE3)**. A value of 1 indicates the processor supports this technology.
1	PCLMULQDQ	**PCLMULQDQ**. A value of 1 indicates the processor supports the PCLMULQDQ instruction
2	DTES64	**64-bit DS Area**. A value of 1 indicates the processor supports DS area using 64-bit layout

Table 3-15 Feature Information Returned in the ECX Register (Continued)

Bit #	Mnemonic	Description
3	MONITOR	**MONITOR/MWAIT**. A value of 1 indicates the processor supports this feature.
4	DS-CPL	**CPL Qualified Debug Store**. A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5	VMX	**Virtual Machine Extensions**. A value of 1 indicates that the processor supports this technology
6	SMX	**Safer Mode Extensions**. A value of 1 indicates that the processor supports this technology. See Chapter 6, "Safer Mode Extensions Reference".
7	EST	**Enhanced Intel SpeedStep® technology**. A value of 1 indicates that the processor supports this technology.
8	TM2	**Thermal Monitor 2**. A value of 1 indicates whether the processor supports this technology.
9	SSSE3	A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor
10	CNXT-ID	**L1 Context ID**. A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.
11	Reserved	Reserved
12	FMA	A value of 1 indicates the processor supports FMA extensions using YMM state.
13	CMPXCHG16B	**CMPXCHG16B Available**. A value of 1 indicates that the feature is available. See the "CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes" section in this chapter for a description.
14	xTPR Update Control	**xTPR Update Control**. A value of 1 indicates that the processor supports changing IA32_MISC_ENABLES[bit 23].
15	PDCM	**Perfmon and Debug Capability**: A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.
16	Reserved	Reserved
17	PCID	**Process-context identifiers**. A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.
18	DCA	A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	SSE4.1	A value of 1 indicates that the processor supports SSE4.1.
20	SSE4.2	A value of 1 indicates that the processor supports SSE4.2.
21	x2APIC	A value of 1 indicates that the processor supports x2APIC feature.
22	MOVBE	A value of 1 indicates that the processor supports MOVBE instruction.

Table 3-15 Feature Information Returned in the ECX Register (Continued)

Bit #	Mnemonic	Description
23	POPCNT	A value of 1 indicates that the processor supports the POPCNT instruction.
24	TSC-Deadline	A value of 1 indicates that the processor's local APIC timer supports one-shot operation using a TSC deadline value.
25	AESNI	A value of 1 indicates that the processor supports the AESNI instruction extensions.
26	XSAVE	A value of 1 indicates that the processor supports the XSAVE/ XRSTOR processor extended states feature, the XSETBV/ XGETBV instructions, and the XFEATURE_ENABLED_MASK register (XCR0).
27	OSXSAVE	A value of 1 indicates that the OS has enabled XSETBV/XGETBV instructions to access the XFEATURE_ENABLED_MASK register (XCR0), and support for processor extended state management using XSAVE/XRSTOR.
28	AVX	A value of 1 indicates the processor supports the AVX instruction extensions.
30 - 29	Reserved	Reserved
31	Not Used	Always returns 0

...

Table 3-17 Encoding of CPUID Leaf 2 Descriptors

Value	Type	Description
...		
E4H	Cache	3rd-level cache: 8 MByte, 16-way set associative, 64 byte line size
EAH	Cache	3rd-level cache: 12MByte, 24-way set associative, 64 byte line size
EBH	Cache	3rd-level cache: 18MByte, 24-way set associative, 64 byte line size
ECH	Cache	3rd-level cache: 24MByte, 24-way set associative, 64 byte line size
F0H	Prefetch	64-Byte prefetching
...		

...

DPPS — Dot Product of Packed Single Precision Floating-Point Values

...

Operation

```
IF (imm8[4] = 1)
    THEN Temp1[31:0] ← DEST[31:0] * SRC[31:0];
    ELSE Temp1[31:0] ← +0.0; FI;
IF (imm8[5] = 1)
    THEN Temp1[63:32] ← DEST[63:32] * SRC[63:32];
    ELSE Temp1[63:32] ← +0.0; FI;
IF (imm8[6] = 1)
```

```
    THEN Temp1[95:64] ← DEST[95:64] * SRC[95:64];
    ELSE Temp1[95:64] ← +0.0; FI;
IF (imm8[7] = 1)
    THEN Temp1[127:96] ← DEST[127:96] * SRC[127:96];
    ELSE Temp1[127:96] ← +0.0; FI;

Temp2[31:0] ← Temp1[31:0] + Temp1[63:32];
Temp3[31:0] ← Temp1[95:64] + Temp1[127:96];
Temp4[31:0] ← Temp2[31:0] + Temp3[31:0];

IF (imm8[0] = 1)
    THEN DEST[31:0] ← Temp4[31:0];
    ELSE DEST[31:0] ← +0.0; FI;
IF (imm8[1] = 1)
    THEN DEST[63:32] ← Temp4[31:0];
    ELSE DEST[63:32] ← +0.0; FI;
IF (imm8[2] = 1)
    THEN DEST[95:64] ← Temp4[31:0];
    ELSE DEST[95:64] ← +0.0; FI;
IF (imm8[3] = 1)
    THEN DEST[127:96] ← Temp4[31:0];
    ELSE DEST[127:96] ← +0.0; FI;
```

...

INSERTPS — Insert Packed Single Precision Floating-Point Value

...

Operation

```
IF (SRC = REG) THEN COUNT_S ← imm8[7:6];
    ELSE COUNT_S ← 0; FI;
COUNT_D ← imm8[5:4];
ZMASK ← imm8[3:0];

CASE (COUNT_S) OF
    0:   TMP ← SRC[31:0];
    1:   TMP ← SRC[63:32];
    2:   TMP ← SRC[95:64];
    3:   TMP ← SRC[127:96];

CASE (COUNT_D) OF
    0:   TMP2[31:0] ← TMP;
         TMP2[127:32] ← DEST[127:32];
    1:   TMP2[63:32] ← TMP;
         TMP2[31:0] ← DEST[31:0];
         TMP2[127:64] ← DEST[127:64];
    2:   TMP2[95:64] ← TMP;
         TMP2[63:0] ← DEST[63:0];
         TMP2[127:96] ← DEST[127:96];
    3:    TMP2[127:96] ← TMP;
          TMP2[95:0] ← DEST[95:0];

IF (ZMASK[0] = 1) THEN DEST[31:0] ← 00000000H;
    ELSE DEST[31:0] ← TMP2[31:0];
    IF (ZMASK[1] = 1) THEN DEST[63:32] ← 00000000H;
          ELSE DEST[63:32] ← TMP2[63:32];
          IF (ZMASK[2] = 1) THEN DEST[95:64] ← 00000000H;
                ELSE DEST[95:64] ← TMP2[95:64];
                IF (ZMASK[3] = 1) THEN DEST[127:96] ← 00000000H;
                      ELSE DEST[127:96] ← TMP2[127:96];
                FI;
          FI;
    FI;
FI;
```

...

INT *n*/INTO/INT 3—Call to Interrupt Procedure

...

Operation

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

```
IF PE = 0
    THEN
          GOTO REAL-ADDRESS-MODE;
    ELSE (* PE = 1 *)
          IF (VM = 1 and IOPL < 3 AND INT n)
                THEN
                       #GP(0);
                ELSE (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
                      IF (IA32_EFER.LMA = 0)
                            THEN (* Protected mode, or virtual-8086 mode interrupt *)
                                  GOTO PROTECTED-MODE;
                      ELSE (* IA-32e mode interrupt *)
                            GOTO IA-32e-MODE;
                      FI;
          FI;
FI;
REAL-ADDRESS-MODE:
    IF ((vector_number ∗ 4) + 3) is not within IDT limit
          THEN #GP; FI;
    IF stack not large enough for a 6-byte return information
          THEN #SS; FI;
    Push (EFLAGS[15:0]);
    IF ← 0; (* Clear interrupt flag *)
    TF ← 0; (* Clear trap flag *)
    AC ← 0; (* Clear AC flag *)
    Push(CS);
    Push(IP);
    (* No error codes are pushed *)
    CS ← IDT(Descriptor (vector_number ∗ 4), selector));
    EIP ← IDT(Descriptor (vector_number ∗ 4), offset)); (* 16 bit offset AND 0000FFFFH *)
END;
PROTECTED-MODE:
    IF ((vector_number « 3) + 7) is not within IDT limits
    or selected IDT descriptor is not an interrupt-, trap-, or task-gate type
        THEN #GP((vector_number ∗ 8) + 2 + EXT); FI;
        (* EXT is bit 0 in error code *)
    IF software interrupt (* Generated by INT n, INT 3, or INTO *)
        THEN
            IF gate DPL < CPL
                THEN #GP((vector_number ∗ 8) + 2 ); FI;
                (* PE = 1, DPL<CPL, software interrupt *)
    FI;
    IF gate not present
        THEN #NP((vector_number « 3) + 2 + EXT); FI;
    IF task gate (* Specified in the selected interrupt table descriptor *)
        THEN GOTO TASK-GATE;
        ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
    FI;
END;
IA-32e-MODE:
    IF ((vector_number ∗ 16) + 15) is not in IDT limits
    or selected IDT descriptor is not an interrupt-, or trap-gate type
        THEN #GP((vector_number « 3) + 2 + EXT);
        (* EXT is bit 0 in error code *)
    FI;
    IF software interrupt (* Generated by INT n, INT 3, but not INTO *)
        THEN
            IF gate DPL < CPL
                THEN #GP((vector_number « 3) + 2 );
                (* PE = 1, DPL < CPL, software interrupt *)
            FI;
        ELSE (* Generated by INTO *)
            #UD;
    FI;
    IF gate not present
        THEN #NP((vector_number « 3) + 2 + EXT);
    FI;
    GOTO TRAP-OR-INTERRUPT-GATE; (* Trap/interrupt gate *)
END;
TASK-GATE: (* PE = 1, task gate *)
    Read TSS selector in task gate (IDT descriptor);
        IF local/global bit is set to local or index not within GDT limits
            THEN #GP(TSS selector); FI;
        Access TSS descriptor in GDT;
```

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

```
END;
INTER-PRIVILEGE-LEVEL-INTERRUPT:
   (* PE = 1, interrupt or trap gate, non-conforming code segment, DPL < CPL *)
   IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
       THEN
       (* Identify stack-segment selector for new privilege level in current TSS *)
           IF current TSS is 32-bit
               THEN
                   TSSstackAddress ← (new code-segment DPL ∗ 8) + 4;
                   IF (TSSstackAddress + 5) > current TSS limit
                       THEN #TS(current TSS selector); FI;
                   NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 4);
                   NewESP ← 4 bytes loaded from (TSS base + TSSstackAddress);
               ELSE     (* current TSS is 16-bit *)
                   TSSstackAddress ← (new code-segment DPL ∗ 4) + 2
                   IF (TSSstackAddress + 3) > current TSS limit
                       THEN #TS(current TSS selector); FI;
                   NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 2);
                   NewESP ← 2 bytes loaded from (TSS base + TSSstackAddress);
           FI;
           IF NewSS is NULL
               THEN #TS(EXT); FI;
           IF NewSS index is not within its descriptor-table limits
           or NewSS RPL ≠ new code-segment DPL
               THEN #TS(NewSS + EXT); FI;
           Read new stack-segment descriptor for NewSS in GDT or LDT;
           IF new stack-segment DPL ≠ new code-segment DPL
           or new stack-segment Type does not indicate writable data segment
               THEN #TS(NewSS + EXT); FI;
           IF NewSS is not present
               THEN #SS(NewSS + EXT); FI;
       ELSE (* IA-32e mode *)
           IF IDT-gate IST = 0
               THEN TSSstackAddress ← (new code-segment DPL ∗ 8) + 4;
               ELSE TSSstackAddress ← (IDT gate IST ∗ 8) + 28;
           FI;
           IF (TSSstackAddress + 7) > current TSS limit
               THEN #TS(current TSS selector); FI;
           NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
           NewSS ← new code-segment DPL; (* null selector with RPL = new CPL *)
   FI;
   IF IDT gate is 32-bit
           THEN
               IF new stack does not have room for 24 bytes (error code pushed)
               or 20 bytes (no error code pushed)
                   THEN #SS(NewSS + EXT); FI;
           FI
       ELSE
           IF IDT gate is 16-bit
               THEN
```

```
                    IF new stack does not have room for 12 bytes (error code pushed)
                    or 10 bytes (no error code pushed);
                        THEN #SS(NewSS + EXT); FI;
            ELSE (* 64-bit IDT gate*)
                IF StackAddress is non-canonical
                    THEN #SS(0);FI;
        FI;
FI;
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
    THEN
        IF instruction pointer from IDT gate is not within new code-segment limits
            THEN #GP(0); FI;
        ESP ← NewESP;
        SS ← NewSS; (* Segment descriptor information also loaded *)
    ELSE (* IA-32e mode *)
        IF instruction pointer from IDT gate contains a non-canonical address
            THEN #GP(0); FI:
        RSP ← NewRSP & FFFFFFFFFFFFFFF0H;
        SS ← NewSS;
FI;
IF IDT gate is 32-bit
    THEN
        CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
    ELSE
        IF IDT gate 16-bit
            THEN
                CS:IP ← Gate(CS:IP);
                (* Segment descriptor information also loaded *)
            ELSE (* 64-bit IDT gate *)
                CS:RIP ← Gate(CS:RIP);
                (* Segment descriptor information also loaded *)
        FI;
FI;
IF IDT gate is 32-bit
        THEN
            Push(far pointer to old stack);
            (* Old SS and ESP, 3 words padded to 4 *)
            Push(EFLAGS);
            Push(far pointer to return instruction);
            (* Old CS and EIP, 3 words padded to 4 *)
            Push(ErrorCode); (* If needed, 4 bytes *)
        ELSE
            IF IDT gate 16-bit
                THEN
                    Push(far pointer to old stack);
                    (* Old SS and SP, 2 words *)
                    Push(EFLAGS(15-0]);
                    Push(far pointer to return instruction);
                    (* Old CS and IP, 2 words *)
                    Push(ErrorCode); (* If needed, 2 bytes *)
                    ELSE (* 64-bit IDT gate *)
                        Push(far pointer to old stack);
                        (* Old SS and SP, each an 8-byte push *)
                        Push(RFLAGS); (* 8-byte push *)
                        Push(far pointer to return instruction);
                        (* Old CS and RIP, each an 8-byte push *)
                        Push(ErrorCode); (* If needed, 8-bytes *)
                FI;
        FI;
        CPL ← new code-segment DPL;
        CS(RPL) ← CPL;
        IF IDT gate is interrupt gate
            THEN IF ← 0 (* Interrupt flag set to 0, interrupts disabled *); FI;
        TF ← 0;
        VM ← 0;
        RF ← 0;
        NT ← 0;
    END;
    INTERRUPT-FROM-VIRTUAL-8086-MODE:
        (* Identify stack-segment selector for privilege level 0 in current TSS *)
        IF current TSS is 32-bit
            THEN
                IF TSS limit < 9
                    THEN #TS(current TSS selector); FI;
                NewSS ← 2 bytes loaded from (current TSS base + 8);
                NewESP ← 4 bytes loaded from (current TSS base + 4);
            ELSE (* current TSS is 16-bit *)
                IF TSS limit < 5
                    THEN #TS(current TSS selector); FI;
                NewSS ← 2 bytes loaded from (current TSS base + 4);
                NewESP ← 2 bytes loaded from (current TSS base + 2);
        FI;
        IF NewSS is NULL
            THEN #TS(EXT); FI;
        IF NewSS index is not within its descriptor table limits
        or NewSS RPL ≠ 0
            THEN #TS(NewSS + EXT); FI;
        Read new stack-segment descriptor for NewSS in GDT or LDT;
        IF new stack-segment DPL ≠ 0 or stack segment does not indicate writable data segment
            THEN #TS(NewSS + EXT); FI;
        IF new stack segment not present
            THEN #SS(NewSS + EXT); FI;
        IF IDT gate is 32-bit
            THEN
                IF new stack does not have room for 40 bytes (error code pushed)
                or 36 bytes (no error code pushed)
                    THEN #SS(NewSS + EXT); FI;
            ELSE (* IDT gate is 16-bit)
                IF new stack does not have room for 20 bytes (error code pushed)
                or 18 bytes (no error code pushed)
                    THEN #SS(NewSS + EXT); FI;
        FI;
        IF instruction pointer from IDT gate is not within new code-segment limits
            THEN #GP(0); FI;
        tempEFLAGS ← EFLAGS;
        VM ← 0;
        TF ← 0;
        RF ← 0;
        NT ← 0;
        IF service through interrupt gate
            THEN IF = 0; FI;
        TempSS ← SS;
        TempESP ← ESP;
        SS ← NewSS;
        ESP ← NewESP;
        (* Following pushes are 16 bits for 16-bit IDT gates and 32 bits for 32-bit IDT gates;
        Segment selector pushes in 32-bit mode are padded to two words *)
        Push(GS);
        Push(FS);
        Push(DS);
        Push(ES);
        Push(TempSS);
        Push(TempESP);
        Push(TempEFlags);
        Push(CS);
        Push(EIP);
        GS ← 0; (* Segment registers made NULL, invalid for use in protected mode *)
        FS ← 0;
        DS ← 0;
        ES ← 0;
        CS:IP ← Gate(CS); (* Segment descriptor information also loaded *)
        IF OperandSize = 32
            THEN
                EIP ← Gate(instruction pointer);
            ELSE (* OperandSize is 16 *)
                EIP ← Gate(instruction pointer) AND 0000FFFFH;
        FI;
        (* Start execution of new routine in Protected Mode *)
    END;
    INTRA-PRIVILEGE-LEVEL-INTERRUPT:
        (* PE = 1, DPL = CPL or conforming segment *)
        IF IA32_EFER.LMA = 1 (* IA-32e mode *)
            IF IDT-descriptor IST ≠ 0
                THEN
                    TSSstackAddress ← (IDT-descriptor IST ∗ 8) + 28;
                    IF (TSSstackAddress + 7) > TSS limit
                        THEN #TS(current TSS selector); FI;
                    NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
        FI;
        IF 32-bit gate (* implies IA32_EFER.LMA = 0 *)
```

```
        THEN
            IF current stack does not have room for 16 bytes (error code pushed)
            or 12 bytes (no error code pushed)
                THEN #SS(0); FI;
        ELSE IF 16-bit gate (* implies IA32_EFER.LMA = 0 *)
            IF current stack does not have room for 8 bytes (error code pushed)
            or 6 bytes (no error code pushed)
                THEN #SS(0); FI;
        ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
                IF NewRSP contains a non-canonical address
                    THEN #SS(0);
        FI;
    FI;
    IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
        THEN
            IF instruction pointer from IDT gate is not within new code-segment limit
                THEN #GP(0); FI;
        ELSE
            IF instruction pointer from IDT gate contains a non-canonical address
                THEN #GP(0); FI:
            RSP ← NewRSP & FFFFFFFFFFFFFFF0H;
    FI;
    IF IDT gate is 32-bit (* implies IA32_EFER.LMA = 0 *)
        THEN
            Push (EFLAGS);
            Push (far pointer to return instruction); (* 3 words padded to 4 *)
            CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
            Push (ErrorCode); (* If any *)
        ELSE
            IF IDT gate is 16-bit (* implies IA32_EFER.LMA = 0 *)
                THEN
                    Push (FLAGS);
                    Push (far pointer to return location); (* 2 words *)
                    CS:IP ← Gate(CS:IP);
                    (* Segment descriptor information also loaded *)
                    Push (ErrorCode); (* If any *)
                ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
                    Push(far pointer to old stack);
                    (* Old SS and SP, each an 8-byte push *)
                    Push(RFLAGS); (* 8-byte push *)
                    Push(far pointer to return instruction);
                    (* Old CS and RIP, each an 8-byte push *)
                    Push(ErrorCode); (* If needed, 8 bytes *)
                    CS:RIP ← GATE(CS:RIP);
                    (* Segment descriptor information also loaded *)
            FI;
    FI;
    CS(RPL) ← CPL;
    IF IDT gate is interrupt gate
        THEN IF ← 0; FI; (* Interrupt flag set to 0; interrupts disabled *)
    TF ← 0;
    NT ← 0;
    VM ← 0;
    RF ← 0;
END;
```

...

LDDQU—Load Unaligned Integer 128 Bits

...

Implementation Notes

- If the source is aligned to a 16-byte boundary, based on the implementation, the 16 bytes may be loaded more than once. For that reason, the usage of LDDQU should be avoided when using uncached or write-combining (WC) memory regions. For uncached or WC memory regions, keep using MOVDQU.
- This instruction is a replacement for MOVDQU (load) in situations where cache line splits significantly affect performance. It should not be used in situations where store-load forwarding is performance critical. If performance of store-load forwarding is critical to the application, use MOVDQA store-load pairs when data is 128-bit aligned or MOVDQU store-load pairs when data is 128-bit unaligned.
- If the memory address is not aligned on 16-byte boundary, some implementations may load up to 32 bytes and return 16 bytes in the destination. Some processor implementations may issue multiple loads to access the appropriate 16 bytes. Developers of multi-threaded or multi-processor software should be aware that on these processors the loads will be performed in a non-atomic way.
- If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the memory address is not aligned on an 8-byte boundary.

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

Operation

```
xmm[127:0] = m128;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
LDDQU      __m128i _mm_lddqu_si128(__m128i const *p)
```

Numeric Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.

#NM	If CR0.TS[bit 3] = 1.
#UD	If CR4.OSFXSR[bit 9] = 0.
	If CR0.EM[bit 2] = 1.
	If CPUID.01H:ECX.SSE3[bit 0] = 0.
	If the LOCK prefix is used.
#AC(0)	If alignment checking is enabled and a memory reference is made that is not aligned on an 8-byte boundary. (Generation of this exception depends on processor implementation.)

Real Address Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:ECX.SSE3[bit 0] = 0.
	If the LOCK prefix is used.

Virtual 8086 Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:ECX.SSE3[bit 0] = 0.
	If the LOCK prefix is used.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and a memory reference is made that is not aligned on an 8-byte boundary. (Generation of this exception depends on processor implementation.)

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:ECX.SSE3[bit 0] = 0.
	If the LOCK prefix is used.
#PF(fault-code)	If a page fault occurs.

#AC(0)	If alignment checking is enabled and a memory reference is made that is not aligned on an 8-byte boundary. (Generation of this exception depends on processor implementation.)

...

LFENCE—Load Fence

...

Exceptions (All Modes of Operation)

#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

...

MASKMOVDQU—Store Selected Bytes of Double Quadword

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F F7 */r*	MASKMOVDQU *xmm1*, *xmm2*	A	Valid	Valid	Selectively write bytes from *xmm1* to memory location using the byte mask in *xmm2*. The default memory location is specified by DS:EDI/RDI.

...

MASKMOVQ—Store Selected Bytes of Quadword

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F F7 */r*	MASKMOVQ *mm1*, *mm2*	A	Valid	Valid	Selectively write bytes from *mm1* to memory location using the byte mask in *mm2.* The default memory location is specified by DS:EDI/RDI.

...

MOVDDUP—Move One Double-FP and Duplicate

...

Operation

```
IF (Source = m64)
    THEN
        (* Load instruction *)
```

```
            xmm1[63:0] = m64;
            xmm1[127:64] = m64;
    ELSE
            (* Move instruction *)
            xmm1[63:0] = xmm2[63:0];
            xmm1[127:64] = xmm2[63:0];
FI;
```

...

MOVDQU—Move Unaligned Double Quadword

...

Description

Moves a double quadword from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated.[1]

To move a double quadword to or from memory locations that are known to be aligned on 16-byte boundaries, use the MOVDQA instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

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

Operation

```
DEST ← SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVDQU	void _mm_storeu_si128 (__m128i *p, __m128i a)
MOVDQU	__m128i _mm_loadu_si128 (__m128i *p)

SIMD Floating-Point Exceptions

None.

1. If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the operand is not aligned on an 8-byte boundary.

Protected Mode Exceptions

#AC(0)	If alignment checking is enabled and a memory reference is made that is not aligned on an 8-byte boundary. (Generation of this exception depends on processor implementation.)
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE2[bit 26] = 0.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#AC(0)	If alignment checking is enabled and a memory reference is made that is not aligned on an 8-byte boundary. (Generation of this exception depends on processor implementation.)
#PF(fault-code)	For a page fault.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#AC(0)	If alignment checking is enabled and a memory reference is made that is not aligned on an 8-byte boundary. (Generation of this exception depends on processor implementation.)
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

...

MOVSHDUP—Move Packed Single-FP High and Duplicate

...

Operation

```
IF (Source = m128)
    THEN     (* Load instruction *)
         xmm1[31:0] = m128[63:32];
         xmm1[63:32] = m128[63:32];
         xmm1[95:64] = m128[127:96];
         xmm1[127:96] = m128[127:96];
    ELSE     (* Move instruction *)
         xmm1[31:0] = xmm2[63:32];
         xmm1[63:32] = xmm2[63:32];
         xmm1[95:64] = xmm2[127:96];
         xmm1[127:96] = xmm2[127:96];
FI;
```

...

MOVSLDUP—Move Packed Single-FP Low and Duplicate

...

Operation

```
IF (Source = m128)
    THEN     (* Load instruction *)
         xmm1[31:0] = m128[31:0];
         xmm1[63:32] = m128[31:0];
         xmm1[95:64] = m128[95:64];
         xmm1[127:96] = m128[95::64];
    ELSE     (* Move instruction *)
         xmm1[31:0] = xmm2[31:0];
         xmm1[63:32] = xmm2[31:0];
         xmm1[95:64] = xmm2[95:64];
         xmm1[127:96] = xmm2[95:64];
FI;
```

...

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

...

Description

Moves a double quadword containing two packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand).

This instruction can be used to load an XMM register from a 128-bit memory location, store the contents of an XMM register into a 128-bit memory location, or move data between two XMM registers. When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated.[1]

To move double-precision floating-point values to and from memory locations that are known to be aligned on 16-byte boundaries, use the MOVAPD instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

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

Operation

```
DEST ← SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVUPD	__m128 _mm_loadu_pd(double * p)
MOVUPD	void _mm_storeu_pd(double *p, __m128 a)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#AC(0)	If alignment checking is enabled and a memory reference is made that is not aligned on an 8-byte boundary. (Generation of this exception depends on processor implementation.)
#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.

1. If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the operand is not aligned on an 8-byte boundary.

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

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#AC(0)	If alignment checking is enabled and a memory reference is made that is not aligned on an 8-byte boundary. (Generation of this exception depends on processor implementation.)
#PF(fault-code)	For a page fault.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#AC(0)	If alignment checking is enabled and a memory reference is made that is not aligned on an 8-byte boundary. (Generation of this exception depends on processor implementation.)
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

...

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

...

Description

Moves a double quadword containing four packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, store the contents of an XMM register into a 128-bit memory location, or move data between two XMM registers. When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated.[1]

1. If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the operand is not aligned on an 8-byte boundary.

To move packed single-precision floating-point values to and from memory locations that are known to be aligned on 16-byte boundaries, use the MOVAPS instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

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

Operation

```
DEST ← SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
MOVUPS   __m128 _mm_loadu_ps(double * p)
MOVUPS   void _mm_storeu_ps(double *p, __m128 a)
```

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#AC(0)	If alignment checking is enabled and a memory reference is made that is not aligned on an 8-byte boundary. (Generation of this exception depends on processor implementation.)
#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE[bit 25] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE[bit 25] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#AC(0)	If alignment checking is enabled and a memory reference is made that is not aligned on an 8-byte boundary. (Generation of this exception depends on processor implementation.)
#PF(fault-code)	For a page fault.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#AC(0)	If alignment checking is enabled and a memory reference is made that is not aligned on an 8-byte boundary. (Generation of this exception depends on processor implementation.)
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE[bit 25] = 0.
	If the LOCK prefix is used.

...

9. Updates to Chapter 4, Volume 2B

Change bars show changes to Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B:* Instruction Set Reference, N-Z.

...

PACKUSDW — Pack with Unsigned Saturation

...

Intel C/C++ Compiler Intrinsic Equivalent

```
PACKUSDW        __m128i _mm_packus_epi16(__m128i m1, __m128i m2);
```

...

PBLENDVB — Variable Blend Packed Bytes

...

Operation

```
MASK ← XMM0;
IF (MASK[7] = 1)
    THEN DEST[7:0] ← SRC[7:0];
    ELSE DEST[7:0] ← DEST[7:0]; FI;
IF (MASK[15] = 1)
    THEN DEST[15:8] ← SRC[15:8];
    ELSE DEST[15:8] ← DEST[15:8]; FI;
IF (MASK[23] = 1)
    THEN DEST[23:16] ← SRC[23:16]
    ELSE DEST[23:16] ← DEST[23:16]; FI;
IF (MASK[31] = 1)
    THEN DEST[31:24] ← SRC[31:24]
    ELSE DEST[31:24] ← DEST[31:24]; FI;
IF (MASK[39] = 1)
    THEN DEST[39:32] ← SRC[39:32]
    ELSE DEST[39:32] ← DEST[39:32]; FI;
IF (MASK[47] = 1)
    THEN DEST[47:40] ← SRC[47:40]
    ELSE DEST[47:40] ← DEST[47:40]; FI;
IF (MASK[55] = 1)
    THEN DEST[55:48] ← SRC[55:48]
    ELSE DEST[55:48] ← DEST[55:48]; FI;
IF (MASK[63] = 1)
    THEN DEST[63:56] ← SRC[63:56]
    ELSE DEST[63:56] ← DEST[63:56]; FI;
IF (MASK[71] = 1)
    THEN DEST[71:64] ← SRC[71:64]
    ELSE DEST[71:64] ← DEST[71:64]; FI;
IF (MASK[79] = 1)
    THEN DEST[79:72] ← SRC[79:72]
    ELSE DEST[79:72] ← DEST[79:72]; FI;
IF (MASK[87] = 1)
    THEN DEST[87:80] ← SRC[87:80]
    ELSE DEST[87:80] ← DEST[87:80]; FI;
IF (MASK[95] = 1)
    THEN DEST[95:88] ← SRC[95:88]
    ELSE DEST[95:88] ← DEST[95:88]; FI;
IF (MASK[103] = 1)
    THEN DEST[103:96] ← SRC[103:96]
    ELSE DEST[103:96] ← DEST[103:96]; FI;
IF (MASK[111] = 1)
    THEN DEST[111:104] ← SRC[111:104]
    ELSE DEST[111:104] ← DEST[111:104]; FI;
IF (MASK[119] = 1)
    THEN DEST[119:112] ← SRC[119:112]
    ELSE DEST[119:112] ← DEST[119:112]; FI;
IF (MASK[127] = 1)
    THEN DEST[127:120] ← SRC[127:120]
    ELSE DEST[127:120] ← DEST[127:120]); FI;
```

...

PBLENDW — Blend Packed Words

...

Operation

```
IF (imm8[0] = 1)
    THEN DEST[15:0] ← SRC[15:0];
    ELSE DEST[15:0] ← DEST[15:0]; FI;
IF (imm8[1] = 1)
    THEN DEST[31:16] ← SRC[31:16];
    ELSE DEST[31:16] ← DEST[31:16]); FI;
IF (imm8[2] = 1)
    THEN DEST[47:32] ← SRC[47:32];
    ELSE DEST[47:32] ← DEST[47:32]; FI;
IF (imm8[3] = 1)
    THEN DEST[63:48] ← SRC[63:48];
    ELSE DEST[63:48] ← DEST[63:48]; FI;
IF (imm8[4] = 1)
    THEN DEST[79:64] ← SRC[79:64];
    ELSE DEST[79:64] ← DEST[79:64]; FI;
IF (imm8[5] = 1)
    THEN DEST[95:80] ← SRC[95:80];
    ELSE DEST[95:80] ← DEST[95:80]; FI;
IF (imm8[6] = 1)
    THEN DEST[111:96] ← SRC[111:96];
    ELSE DEST[111:96] ← DEST[111:96]; FI;
IF (imm8[7] = 1)
    THEN DEST[127:112] ← SRC[127:112];
    ELSE DEST[127:112] ← DEST[127:112]; FI;
```

...

PCMPESTRI — Packed Compare Explicit Length Strings, Return Index

...

Description

The instruction compares and processes data from two string fragments based on the encoded value in the Imm8 Control Byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM"), and generates an index stored to ECX.

Each string fragment is represented by two values. The first value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in EAX (for xmm1) or EDX (for xmm2/m128) and represents the number of bytes/words which are valid for the respective xmm/m128 data.

The length of each input is interpreted as being the absolute-value of the value in EAX (EDX). The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit0] when the value in EAX (EDX) is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). The index of the first (or last, according to imm8[6]) set bit of IntRes2 (see Section 4.1.4) is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise
SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise
OFlag – IntRes2[0]
AFlag – Reset
PFlag – Reset

...

PCMPESTRM — Packed Compare Explicit Length Strings, Return Mask

...

Description

The instruction compares data from two string fragments based on the encoded value in the imm8 control byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM"), and generates a mask stored to XMM0.

Each string fragment is represented by two values. The first value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in EAX (for xmm1) or EDX (for xmm2/m128) and represents the number of bytes/words which are valid for the respective xmm/m128 data.

The length of each input is interpreted as being the absolute-value of the value in EAX (EDX). The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit0] when the value in EAX (EDX) is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise
SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise
OFlag –IntRes2[0]
AFlag – Reset
PFlag – Reset

...

PEXTRW—Extract Word

...

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

...

POPCNT — Return the Count of Number of Bits Set to 1

...

Flags Affected

OF, SF, ZF, AF, CF, PF are all cleared. ZF is set if SRC = 0, otherwise ZF is cleared

...

PSHUFB — Packed Shuffle Bytes

...

Operation

PSHUFB with 64 bit operands:

```
for i = 0 to 7 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7...(i*8)+0] ← 0;
    else
        index[2..0] ← SRC[(i*8)+2 .. (i*8)+0];
        DEST[(i*8)+7...(i*8)+0] ← DEST[(index*8+7)..(index*8+0)];
    endif;
}
```

PSHUFB with 128 bit operands:

```
for i = 0 to 15 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7..(i*8)+0] ← 0;
     else
        index[3..0] ← SRC[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← DEST[(index*8+7)..(index*8+0)];
    endif
}
```

...

PSIGNB/PSIGNW/PSIGND — Packed SIGN

...

Operation

PSIGNB with 64 bit operands:

```
    IF (SRC[7:0] < 0 )
        DEST[7:0] ← Neg(DEST[7:0])
    ELSEIF (SRC[7:0] = 0 )
        DEST[7:0] ← 0
    ELSEIF (SRC[7:0] > 0 )
        DEST[7:0] ← DEST[7:0]
    Repeat operation for 2nd through 7th bytes

    IF (SRC[63:56] < 0 )
        DEST[63:56] ← Neg(DEST[63:56])
    ELSEIF (SRC[63:56] = 0 )
        DEST[63:56] ← 0
    ELSEIF (SRC[63:56] > 0 )
        DEST[63:56] ← DEST[63:56]
```

PSIGNB with 128 bit operands:

```
    IF (SRC[7:0] < 0 )
        DEST[7:0] ← Neg(DEST[7:0])
    ELSEIF (SRC[7:0] = 0 )
        DEST[7:0] ← 0
    ELSEIF (SRC[7:0] > 0 )
        DEST[7:0] ← DEST[7:0]
    Repeat operation for 2nd through 15th bytes
    IF (SRC[127:120] < 0 )
        DEST[127:120] ← Neg(DEST[127:120])
    ELSEIF (SRC[127:120] = 0 )
        DEST[127:120] ← 0
    ELSEIF (SRC[127:120] > 0 )
        DEST[127:120] ← DEST[127:120]
```

PSIGNW with 64 bit operands:

```
    IF (SRC[15:0] < 0 )
        DEST[15:0] ← Neg(DEST[15:0])
    ELSEIF (SRC[15:0] = 0 )
        DEST[15:0] ← 0
    ELSEIF (SRC[15:0] > 0 )
        DEST[15:0] ← DEST[15:0]
Repeat operation for 2nd through 3rd words
    IF (SRC[63:48] < 0 )
        DEST[63:48] ← Neg(DEST[63:48])
    ELSEIF (SRC[63:48] = 0 )
        DEST[63:48] ← 0
    ELSEIF (SRC[63:48] > 0 )
        DEST[63:48] ← DEST[63:48]
```

PSIGNW with 128 bit operands:

```
IF (SRC[15:0] < 0 )
    DEST[15:0] ← Neg(DEST[15:0])
ELSEIF (SRC[15:0] = 0 )
    DEST[15:0] ← 0
ELSEIF (SRC[15:0] > 0 )
    DEST[15:0] ← DEST[15:0]
Repeat operation for 2nd through 7th words
IF (SRC[127:112] < 0 )
    DEST[127:112] ← Neg(DEST[127:112])
ELSEIF (SRC[127:112] = 0 )
    DEST[127:112] ← 0
ELSEIF (SRC[127:112] > 0 )
    DEST[127:112] ← DEST[127:112]
```

PSIGND with 64 bit operands:

```
IF (SRC[31:0] < 0 )
    DEST[31:0] ← Neg(DEST[31:0])
ELSEIF (SRC[31:0] = 0 )
    DEST[31:0] ← 0
ELSEIF (SRC[31:0] > 0 )
    DEST[31:0] ← DEST[31:0]
IF (SRC[63:32] < 0 )
    DEST[63:32] ← Neg(DEST[63:32])
ELSEIF (SRC[63:32] = 0 )
    DEST[63:32] ← 0
ELSEIF (SRC[63:32] > 0 )
    DEST[63:32] ← DEST[63:32]
```

PSIGND with 128 bit operands:

```
IF (SRC[31:0] < 0 )
    DEST[31:0] ← Neg(DEST[31:0])
ELSEIF (SRC[31:0] = 0 )
    DEST[31:0] ← 0
ELSEIF (SRC[31:0] > 0 )
    DEST[31:0] ← DEST[31:0]
Repeat operation for 2nd through 3rd double words
IF (SRC[127:96] < 0 )
    DEST[127:96] ← Neg(DEST[127:96])
ELSEIF (SRC[127:96] = 0 )
    DEST[127:96] ← 0
ELSEIF (SRC[127:96] > 0 )
    DEST[127:96] ← DEST[127:96]
```

...

ROUNDPD — Round Packed Double Precision Floating-Point Values

...

Operation

```
IF (imm[2] = ‘1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[63:0] ← ConvertDPFPToInteger_M(SRC[63:0]);
        DEST[127:64] ← ConvertDPFPToInteger_M(SRC[127:64]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[63:0] ← ConvertDPFPToInteger_Imm(SRC[63:0]);
        DEST[127:64] ← ConvertDPFPToInteger_Imm(SRC[127:64]);
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

```
ROUNDPD    __m128 mm_round_pd(__m128d s1, int iRoundMode);
           __m128 mm_floor_pd(__m128d s1);
           __m128 mm_ceil_pd(__m128d s1);
```

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = ‘0; if imm[3] = ‘1, then the Precision Mask in the MXSCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDPD.

...

ROUNDPS — Round Packed Single Precision Floating-Point Values

...

Operation

```
IF (imm[2] = ‘1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[31:0] ← ConvertSPFPToInteger_M(SRC[31:0]);
        DEST[63:32] ← ConvertSPFPToInteger_M(SRC[63:32]);
        DEST[95:64] ← ConvertSPFPToInteger_M(SRC[95:64]);
        DEST[127:96] ← ConvertSPFPToInteger_M(SRC[127:96]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[31:0] ← ConvertSPFPToInteger_Imm(SRC[31:0]);
        DEST[63:32] ← ConvertSPFPToInteger_Imm(SRC[63:32]);
        DEST[95:64] ← ConvertSPFPToInteger_Imm(SRC[95:64]);
        DEST[127:96] ← ConvertSPFPToInteger_Imm(SRC[127:96]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
ROUNDPS    __m128 mm_round_ps(__m128 s1, int iRoundMode);
           __m128 mm_floor_ps(__m128 s1);
           __m128 mm_ceil_ps(__m128 s1);
```

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = ‘0; if imm[3] = ‘1, then the Precision Mask in the MXSCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDPS.

...

ROUNDSD — Round Scalar Double Precision Floating-Point Values

...

Operation

```
IF (imm[2] = ‘1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[63:0] ← ConvertDPFPToInteger_M(SRC[63:0]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[63:0] ← ConvertDPFPToInteger_Imm(SRC[63:0]);
FI;
DEST[127:63] remains unchanged ;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
ROUNDSD   __m128d mm_round_sd(__m128d dst, __m128d s1, int iRoundMode);
          __m128d mm_floor_sd(__m128d dst, __m128d s1);
          __m128d mm_ceil_sd(__m128d dst, __m128d s1);
```

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = ‘0; if imm[3] = ‘1, then the Precision Mask in the MXSCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDSD.

...

ROUNDSS — Round Scalar Single Precision Floating-Point Values

...

Operation

```
IF (imm[2] = ‘1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[31:0] ← ConvertSPFPToInteger_M(SRC[31:0]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[31:0] ← ConvertSPFPToInteger_Imm(SRC[31:0]);
FI;
DEST[127:32] remains unchanged ;
```

Intel C/C++ Compiler Intrinsic Equivalent

ROUNDSS	__m128 mm_round_ss(__m128 dst, __m128 s1, int iRoundMode);
	__m128 mm_floor_ss(__m128 dst, __m128 s1);
	__m128 mm_ceil_ss(__m128 dst, __m128 s1);

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = ‘0; if imm[3] = ‘1, then the Precision Mask in the MXSCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDSS.

...

SFENCE—Store Fence

...

Exceptions (All Operating Modes)

#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

...

SWAPGS—Swap GS Base Register

...

See Table 4-16.

Table 4-16 SWAPGS Operation Parameters

Opcode	ModR/M Byte			Instruction	
	MOD	REG	R/M	Not 64-bit Mode	64-bit Mode
OF 01	MOD ≠ 11	111	xxx	INVLPG	INVLPG
	11	111	000	#UD	SWAPGS
	11	111	≠ 000	#UD	#UD

...

SYSEXIT—Fast Return from Fast System Call

...

Description

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

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

- **IA32_SYSENTER_CS** — Contains a 32-bit value, of which the lower 16 bits are the segment selector for the privilege level 0 code segment in which the processor is currently executing. This value is used to compute the segment selectors for the privilege level 3 code and stack segments.
- **EDX** — Contains the 32-bit offset into the privilege level 3 code segment to the first instruction to be executed in the user code.
- **ECX** — Contains the 32-bit stack pointer for the privilege level 3 stack.

The IA32_SYSENTER_CS MSR can be read from and written to using RDMSR/WRMSR. The register address is listed in Table 4-17. This address is defined to remain fixed for future Intel 64 and IA-32 processors.

When SYSEXIT is executed, the processor:

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

See "SWAPGS—Swap GS Base Register" in this chapter for information about using the SYSENTER and SYSEXIT instructions as companion call and return instructions.

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

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

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

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

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

FI;

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

...

SYSRET—Return From Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 07	SYSRET	A	Valid	Invalid	Return to compatibility mode from fast system call
REX.W + 0F 07	SYSRET	A	Valid	Invalid	Return to 64-bit mode from fast system call

...

UD2—Undefined Instruction

...

Description

Generates an invalid opcode exception. This instruction is provided for software testing to explicitly generate an invalid opcode exception. The opcode for this instruction is reserved for this purpose.

Other than raising the invalid opcode exception, this instruction has no effect on processor state or memory.

Even though it is the execution of the UD2 instruction that causes the invalid opcode exception, the instruction pointer saved by delivery of the exception references the UD2 instruction (and not the following instruction).

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

...

XRSTOR—Restore Processor Extended States

...

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If a bit in XCR0 is 0 and the corresponding bit in HEADER.XSTATE_BV field of the source operand is 1.
	If bytes 23:8 of HEADER is not zero.
	If attempting to write any reserved bits of the MXCSR register with 1.

#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
	If 66H, F3H or F2H prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If a bit in XCR0 is 0 and the corresponding bit in HEADER.XSTATE_BV field of the source operand is 1.
	If bytes 23:8 of HEADER is not zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
	If 66H, F3H or F2H prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in Protected Mode

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

	If a bit in XCR0 is 0 and the corresponding bit in XSAVE.HEADER.XSTATE_BV is 1.
	If bytes 23:8 of HEADER is not zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
	If 66H, F3H or F2H prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

...

XSAVE—Save Processor Extended States

...

Description

Performs a full or partial save of the enabled processor state components to a memory address specified in the destination operand. A full or partial save of the processor states is specified by an implicit mask operand via the register pair, EDX:EAX. The destination operand is a memory location that must be 64-byte aligned.

The implicit 64-bit mask operand in EDX:EAX specifies the subset of enabled processor state components to save into the XSAVE/XRSTOR save area. The XSAVE/XRSTOR save area comprises of individual save area for each processor state components and a header section, see Table 4-18. Each component save area is written if both the corresponding bits in the save mask operand and in the XFEATURE_ENABLED_MASK (XCR0) register are 1. A processor state component save area is not updated if either one of the corresponding bits in the mask operand or the XFEATURE_ENABLED_MASK register is 0. If the mask operand (EDX:EAX) contains all 1's, all enabled processor state components in XFEATURE_ENABLED_MASK is written to the respective component save area.

The bit assignment used for the EDX:EAX register pair matches the XFEATURE_ENABLED_MASK register (see chapter 2 of Vol. 3B). For the XSAVE instruction, software can specify "1" in any bit position of EDX:EAX, irrespective of whether the corresponding bit position in XFEATURE_ENABLED_MASK is valid for the processor. The bit vector in EDX:EAX is "anded" with the XFEATURE_ENABLED_MASK to determine which save area will be written.

The content layout of the XSAVE/XRSTOR save area is architecturally defined to be extendable and enumerated via the sub-leaves of CPUID.0DH leaf. The extendable framework of the XSAVE/XRSTOR layout is depicted by Table 4-18. The layout of the XSAVE/XRSTOR save area is fixed and may contain non-contiguous individual save areas. The XSAVE/XRSTOR save area is not compacted if some features are not saved or are not supported by the processor and/or by system software.

The layout of the register fields of first 512 bytes of the XSAVE/XRSTOR is the same as the FXSAVE/FXRSTOR area (refer to "FXSAVE—Save x87 FPU, MMX Technology, and SSE State" on page 476). But XSAVE/XRSTOR organizes the 512 byte area as x87 FPU states (including FPU operation states, x87/MMX data registers), MXCSR (including MXCSR_MASK), and XMM registers.

Bytes 464:511 are available for software use. The processor does not write to bytes 464:511 when executing XSAVE.

The processor writes 1 or 0 to each HEADER.XSTATE_BV[i] bit field of an enabled processor state component in a manner that is consistent to XRSTOR's interaction with HEADER.XSTATE_BV (see the operation section of XRSTOR instruction). If a processor implementation discern that a processor state component is in its initialized state (according to Table 4-20) it may modify the corresponding bit in the HEADER.XSTATE_BV as '0'.

A destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) will result in a general-protection (#GP) exception being generated. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

...

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
	If 66H, F3H or F2H prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection

exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
	If 66H, F3H or F2H prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
	If 66H, F3H or F2H prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

...

10. Updates to Chapter 5, Volume 2B

Change bars show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B:* Instruction Set Reference, N-Z.

...

INVEPT— Invalidate Translations Derived from EPT

...

Operation

```
IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
   THEN #UD;
ELSIF in VMX non-root operation
   THEN VM exit;
ELSIF CPL > 0
   THEN #GP(0);
   ELSE
       INVEPT_TYPE ← value of register operand;
       IF IA32_VMX_EPT_VPID_CAP MSR indicates that processor does not support INVEPT_TYPE
           THEN VMfail(Invalid operand to INVEPT/INVVPID);
           ELSE       // INVEPT_TYPE must be 1 or 2
               INVEPT_DESC ← value of memory operand;
               EPTP ← INVEPT_DESC[63:0];
               CASE INVEPT_TYPE OF
                   1:                  // single-context invalidation
                       IF VM entry with the "enable EPT" VM execution control set to 1
                       would fail due to the EPTP value
                           THEN VMfail(Invalid operand to INVEPT/INVVPID);
                           ELSE
                               Invalidate mappings associated with EPTP[51:12];
                               VMsucceed;
                       FI;
                       BREAK;
                   2:                  // global invalidation
                       Invalidate mappings associated with all EPTPs;
                       VMsucceed;
                       BREAK;
               ESAC;
       FI;
FI;
```

...

INVVPID— Invalidate Translations Based on VPID

...

Operation

```
IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VM exit;
ELSIF CPL > 0
    THEN #GP(0);
    ELSE
        INVVPID_TYPE ← value of register operand;
        IF IA32_VMX_EPT_VPID_CAP MSR indicates that processor does not support
        INVVPID_TYPE
            THEN VMfail(Invalid operand to INVEPT/INVVPID);
            ELSE            // INVVPID_TYPE must be in the range 0–3
                INVVPID_DESC ← value of memory operand;
                IF INVVPID_DESC[63:16] ≠ 0
                    THEN VMfail(Invalid operand to INVEPT/INVVPID);
                    ELSE
                        CASE INVVPID_TYPE OF
                            0:                  // individual-address invalidation
                                VPID ← INVVPID_DESC[15:0];
                                IF VPID = 0
                                    THEN VMfail(Invalid operand to INVEPT/INVVPID);
                                    ELSE
                                        GL_ADDR ← INVVPID_DESC[127:64];
                                        IF (GL_ADDR is not in a canonical form)
                                            THEN
                                                VMfail(Invalid operand to INVEPT/INVVPID);
                                            ELSE
                                                Invalidate mappings for GL_ADDR tagged with
VPID;
                                                VMsucceed;
                                        FI;
                                FI;
                                BREAK;
                            1:                  // single-context invalidation
                                VPID_CTX ← INVVPID_DESC[15:0];
                                IF VPID = 0
                                    THEN VMfail(Invalid operand to INVEPT/INVVPID);
                                    ELSE
                                        Invalidate all mappings tagged with VPID;
                                        VMsucceed;
                                FI;
                                BREAK;
                            2:                  // all-context invalidation
                                Invalidate all mappings tagged with all non-zero VPIDs;
                                VMsucceed;
                                BREAK;
                            3:                  // single-context invalidation retaining globals
                                VPID ← INVVPID_DESC[15:0];
                        IF VPID = 0
                            THEN VMfail(Invalid operand to INVEPT/INVVPID);
                            ELSE
                                Invalidate all mappings tagged with VPID except global
translations;
                                VMsucceed;
                        FI;
                        BREAK;
                    ESAC;
                FI;
            FI;
FI;
```

...

VMCLEAR—Clear Virtual-Machine Control Structure

...

Operation

```
IF (register operand) or (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA =
1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VM exit;
ELSIF CPL > 0
    THEN #GP(0);
    ELSE
        addr ← contents of 64-bit in-memory operand;
        IF addr is not 4KB-aligned OR
        (processor supports Intel 64 architecture and
        addr sets any bits beyond the physical-address width) OR
        (processor does not support Intel 64 architecture, addr sets any bits in the range 63:32)
            THEN VMfail(VMCLEAR with invalid physical address);
            ELSIF addr = VMXON pointer
                THEN VMfail(VMCLEAR with VMXON pointer);
                ELSE
                    ensure that data for VMCS referenced by the operand is in memory;
                    initialize implementation-specific data in VMCS region;
                    launch state of VMCS referenced by the operand ← "clear"
                    IF operand addr = current-VMCS pointer
                        THEN current-VMCS pointer ← FFFFFFFF_FFFFFFFFH;
                    FI;
                    VMsucceed;
        FI;
FI;
```

...

VMLAUNCH/VMRESUME—Launch/Resume Virtual Machine

...

Operation

```
IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF current-VMCS pointer is not valid
    THEN VMfailInvalid;
ELSIF events are being blocked by MOV SS
    THEN VMfailValid(VM entry with events blocked by MOV SS);
ELSIF (VMLAUNCH and launch state of current VMCS is not "clear")
    THEN VMfailValid(VMLAUNCH with non-clear VMCS);
ELSIF (VMRESUME and launch state of current VMCS is not "launched")
    THEN VMfailValid(VMRESUME with non-launched VMCS);
    ELSE
        Check settings of VMX controls and host-state area;
        IF invalid settings
            THEN VMfailValid(VM entry with invalid VMX-control field(s)) or
                    VMfailValid(VM entry with invalid host-state field(s)) or
                    VMfailValid(VM entry with invalid executive-VMCS pointer)) or
                    VMfailValid(VM entry with non-launched executive VMCS) or
                    VMfailValid(VM entry with executive-VMCS pointer not VMXON pointer) or
                    VMfailValid(VM entry with invalid VM-execution control fields in executive
                    VMCS)
                    as appropriate;
            ELSE
                Attempt to load guest state and PDPTRs as appropriate;
                clear address-range monitoring;
                IF failure in checking guest state or PDPTRs
                    THEN VM entry fails (see Section 22.7, in the
                    Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B);
                    ELSE
                        Attempt to load MSRs from VM-entry MSR-load area;
                        IF failure
                            THEN VM entry fails (see Section 22.7, in the Intel® 64 and IA-32 Archi-
tectures Software Developer's Manual, Volume 3B);
                            ELSE
                                IF VMLAUNCH
                                    THEN launch state of VMCS ← "launched";
                                FI;
                                IF in SMM and "entry to SMM" VM-entry control is 0
                                    THEN
                                        IF "deactivate dual-monitor treatment" VM-entry
                                        control is 0
                                            THEN SMM-transfer VMCS pointer ←
                                            current-VMCS pointer;
                                        FI;
                        IF executive-VMCS pointer is VMX pointer
                            THEN current-VMCS pointer ←
                            VMCS-link pointer;
                            ELSE current-VMCS pointer ←
                            executive-VMCS pointer;
                        FI;
                        leave SMM;
                    FI;
                    VM entry succeeds;
                FI;
            FI;
        FI;
FI;
```

...

VMPTRLD—Load Pointer to Virtual-Machine Control Structure

Operation

```
IF (register operand) or (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA =
1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
    ELSE
        addr ← contents of 64-bit in-memory source operand;
        IF addr is not 4KB-aligned OR
        (processor supports Intel 64 architecture and
        addr sets any bits beyond the processor’s physical-address width) OR
        processor does not support Intel 64 architecture and addr sets any bits in the range 63:32
            THEN VMfail(VMPTRLD with invalid physical address);
        ELSIF addr = VMXON pointer
            THEN VMfail(VMPTRLD with VMXON pointer);
            ELSE
                rev ← 32 bits located at physical address addr;
                IF rev ≠ VMCS revision identifier supported by processor
                    THEN VMfail(VMPTRLD with incorrect VMCS revision identifier);
                    ELSE
                        current-VMCS pointer ← addr;
                        VMsucceed;
                FI;
        FI;
FI;
```

...

VMPTRST—Store Pointer to Virtual-Machine Control Structure

...

Operation

```
IF (register operand) or (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA =
1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
    ELSE
        64-bit in-memory destination operand ← current-VMCS pointer;
        VMsucceed;
FI;
```

...

VMWRITE—Write Field to Virtual-Machine Control Structure

...

Operation

```
IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF current-VMCS pointer is not valid
    THEN VMfailInvalid;
ELSIF register destination operand does not correspond to any VMCS field
    THEN VMfailValid(VMREAD/VMWRITE from/to unsupported VMCS component);
ELSIF VMCS field indexed by register destination operand is read-only)
    THEN VMfailValid(VMWRITE to read-only VMCS component);
    ELSE
        VMCS field indexed by register destination operand ← SRC;
        VMsucceed;
FI;
```

...

VMXOFF—Leave VMX Operation

Opcode	Instruction	Description
0F 01 C4	VMXOFF	Leaves VMX operation.

Description

Takes the logical processor out of VMX operation, unblocks INIT signals, conditionally re-enables A20M, and clears any address-range monitoring.[1]

Operation

```
IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF dual-monitor treatment of SMIs and SMM is active
    THEN VMfail(VMXOFF under dual-monitor treatment of SMIs and SMM);
    ELSE
        leave VMX operation;
        unblock INIT;
        unblock SMI;
        IF outside SMX operation[2]
            THEN unblock and enable A20M;
        FI;
        clear address-range monitoring;
        VMsucceed;
FI;
```

...

VMXON—Enter VMX Operation

...

Operation

```
IF (register operand) or (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA =
1 and CS.L = 0)
    THEN #UD;
ELSIF not in VMX operation
    THEN
        IF (CPL > 0) or (in A20M mode) or
        (the values of CR0 and CR4 are not supported in VMX operation[3]) or
        (bit 0 (lock bit) of IA32_FEATURE_CONTROL MSR is clear) or
        (in SMX operation[4] and bit 1 of IA32_FEATURE_CONTROL MSR is clear) or
        (outside SMX operation and bit 2 of IA32_FEATURE_CONTROL MSR is clear)
            THEN #GP(0);
```

1. See the information on MONITOR/MWAIT in Chapter 8, "Multiple-Processor Management," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.
2. A logical processor is outside SMX operation if GETSEC[SENTER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENTER]. See Chapter 6, "Safer Mode Extensions Reference."
3. See Section 19.8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

```
                ELSE
                    addr ← contents of 64-bit in-memory source operand;
                    IF addr is not 4KB-aligned or
                    (processor supports Intel 64 architecture and
                    addr sets any bits beyond the VMX physical-address width) or
                    (processor does not support Intel 64 architecture and
                    addr sets any bits in the range 63:32)
                        THEN VMfailInvalid;
                        ELSE
                            rev ← 32 bits located at physical address addr;
                            IF rev ≠ VMCS revision identifier supported by processor
                                THEN VMfailInvalid;
                                ELSE
                                    current-VMCS pointer ← FFFFFFFF_FFFFFFFFH;
                                    enter VMX operation;
                                    block INIT signals;
                                    block and disable A20M;
                                    clear address-range monitoring;
                                    VMsucceed;
                            FI;
                    FI;
            FI;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
    ELSE VMfail("VMXON executed in VMX root operation");
FI;
```

...

11. Updates to Chapter 6, Volume 2B

Change bars show changes to Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B:* Instruction Set Reference, N-Z.

...

GETSEC[ENTERACCS] - Execute Authenticated Chipset Code

...

Operation in a Uni-Processor Platform

```
(* The state of the internal flag ACMODEFLAG persists across instruction boundary *)
IF (CR4.SMXE=0)
```

4. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENTER]. A logical processor is outside SMX operation if GETSEC[SENTER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENTER]. See Chapter 6, "Safer Mode Extensions Reference."

```
    THEN #UD;
ELSIF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSIF (GETSEC leaf unsupported)
    THEN #UD;
ELSIF ((in VMX operation) or
    (CR0.PE=0) or (CR0.CD=1) or (CR0.NW=1) or (CR0.NE=0) or
    (CPL>0) or (EFLAGS.VM=1) or
    (IA32_APIC_BASE.BSP=0) or
    (TXT chipset not present) or
    (ACMODEFLAG=1) or (IN_SMM=1))
        THEN #GP(0);
FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
    IF (IA32_MC[I]_STATUS← uncorrectable error)
        THEN #GP(0);
OD;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
    THEN #GP(0);
ACBASE← EBX;
ACSIZE← ECX;
IF (((ACBASE MOD 4096) != 0) or ((ACSIZE MOD 64 )!= 0 ) or (ACSIZE < minimum module size) OR (ACSIZE
> authenticated RAM capacity)) or ((ACBASE+ACSIZE) > (2^32 -1)))
    THEN #GP(0);
IF (secondary thread(s) CR0.CD = 1) or ((secondary thread(s) NOT(wait-for-SIPI)) and
    (secondary thread(s) not in SENTER sleep state)
    THEN #GP(0);
Mask SMI, INIT, A20M, and NMI external pin events;
IA32_MISC_ENABLE← (IA32_MISC_ENABLE & MASK_CONST*)
(* The hexadecimal value of MASK_CONST may vary due to processor implementations *)
A20M← 0;
IA32_DEBUGCTL← 0;
Invalidate processor TLB(s);
Drain Outgoing Transactions;
ACMODEFLAG← 1;
SignalTXTMessage(ProcessorHold);
Load the internal ACRAM based on the AC module size;
(* Ensure that all ACRAM loads hit Write Back memory space *)
IF (ACRAM memory type != WB)
    THEN TXT-SHUTDOWN(#BadACMMType);
IF (AC module header version isnot supported) OR (ACRAM[ModuleType] <> 2)
    THEN TXT-SHUTDOWN(#UnsupportedACM);
 (* Authenticate the AC Module and shutdown with an error if it fails *)
KEY← GETKEY(ACRAM, ACBASE);
KEYHASH← HASH(KEY);
CSKEYHASH← READ(TXT.PUBLIC.KEY);
IF (KEYHASH <> CSKEYHASH)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
SIGNATURE← DECRYPT(ACRAM, ACBASE, KEY);
(* The value of SIGNATURE_LEN_CONST is implementation-specific*)
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
```

```
    ACRAM[SCRATCH.I]← SIGNATURE[I];
COMPUTEDSIGNATURE← HASH(ACRAM, ACBASE, ACSIZE);
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.SIGNATURE_LEN_CONST+I]← COMPUTEDSIGNATURE[I];
IF (SIGNATURE<>COMPUTEDSIGNATURE)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
ACMCONTROL← ACRAM[CodeControl];
IF ((ACMCONTROL.0 = 0) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM
load))
    THEN TXT-SHUTDOWN(#UnexpectedHITM);
IF (ACMCONTROL reserved bits are set)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[GDTBasePtr] < (ACRAM[HeaderLen] * 4 + Scratch_size)) OR
    ((ACRAM[GDTBasePtr] + ACRAM[GDTLimit]) >= ACSIZE))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACMCONTROL.0 = 1) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM
load))
    THEN ACEntryPoint← ACBASE+ACRAM[ErrorEntryPoint];
ELSE
    ACEntryPoint← ACBASE+ACRAM[EntryPoint];
IF ((ACEntryPoint >= ACSIZE) OR (ACEntryPoint < (ACRAM[HeaderLen] * 4 + Scratch_size)))THEN TXT-
SHUTDOWN(#BadACMFormat);
IF (ACRAM[GDTLimit] & FFFF0000h)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel] > (ACRAM[GDTLimit] - 15)) OR (ACRAM[SegSel] < 8))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel].TI=1) OR (ACRAM[SegSel].RPL!=0))
    THEN TXT-SHUTDOWN(#BadACMFormat);
CR0.[PG.AM.WP]← 0;
CR4.MCE← 0;
EFLAGS← 00000002h;
IA32_EFER← 0h;
[E|R]BX← [E|R]IP of the instruction after GETSEC[ENTERACCS];
ECX← Pre-GETSEC[ENTERACCS] GDT.limit:CS.sel;
[E|R]DX← Pre-GETSEC[ENTERACCS] GDT.base;
EBP← ACBASE;
GDTR.BASE← ACBASE+ACRAM[GDTBasePtr];
GDTR.LIMIT← ACRAM[GDTLimit];
CS.SEL← ACRAM[SegSel];
CS.BASE← 0;
CS.LIMIT← FFFFFh;
CS.G← 1;
CS.D← 1;
CS.AR← 9Bh;
DS.SEL← ACRAM[SegSel]+8;
DS.BASE← 0;
DS.LIMIT← FFFFFh;
DS.G← 1;
DS.D← 1;
DS.AR← 93h;
```

```
DR7← 00000400h;
IA32_DEBUGCTL← 0;
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
EIP← ACEntryPoint;
END;
```

...

GETSEC[PARAMETERS]—Report the SMX Parameters

...

Table 6-7 SMX Reporting Parameters Format

Parameter Type EAX[4:0]	Parameter Description	EAX[31:5]	EBX[31:0]	ECX[31:0]
0	NULL	Reserved (0 returned)	Reserved (unmodified)	Reserved (unmodified)
1	Supported AC module versions	Reserved (0 returned)	version comparison mask	version numbers supported
2	Max size of authenticated code execution area	Multiply by 32 for size in bytes	Reserved (unmodified)	Reserved (unmodified)
3	External memory types supported during AC mode	Memory type bit mask	Reserved (unmodified)	Reserved (unmodified)
4	Selective SENTER functionality control	EAX[14:8] correspond to available SENTER function disable controls	Reserved (unmodified)	Reserved (unmodified)
5	TXT extensions support	TXT Feature Extensions Flags (see Table 6-8)	Reserved	Reserved
6-31	Undefined	Reserved (unmodified)	Reserved (unmodified)	Reserved (unmodified)

Table 6-8 TXT Feature Extensions Flags

Bit	Definition	Description
5	Processor based S-CRTM support	Returns 1 if this processor implements a processor-rooted S-CRTM capability and 0 if not (S-CRTM is rooted in BIOS). This flag cannot be used to infer whether the chipset supports TXT or whether the processor support SMX.
6	Machine Check Handling	Returns 1 if it machine check status registers can be preserved through ENTERACCS and SENTER. If this bit is 1, the caller of ENTERACCS and SENTER is not required to clear machine check error status bits before invoking these GETSEC leaves. If this bit returns 0, the caller of ENTERACCS and SENTER must clear all machine check error status bits before invoking these GETSEC leaves.
31:7	Reserved	Reserved for future use. Will return 0.

...

Supported AC module versions (as defined by the AC module HeaderVersion field) can be determined for a particular SMX capable processor by the type 1 parameter. Using EBX to index through the available parameters reported by GETSEC[PARAMETERS] for each unique parameter set returned for type 1, software can determine the complete list of AC module version(s) supported.

For each parameter set, EBX returns the comparison mask and ECX returns the available HeaderVersion field values supported, after AND'ing the target HeaderVersion with the comparison mask. Software can then determine if a particular AC module version is supported by following the pseudo-code search routine given below:

```
parameter_search_index= 0
do {
        EBX= parameter_search_index++
        EAX= 6
        GETSEC
        if (EAX[4:0] = 1) {
            if ((version_query & EBX) = ECX) {
                version_is_supported= 1
                break
        }
    }
} while (EAX[4:0]!= 0)
```

...

GETSEC[SENTER]—Enter a Measured Environment

...

Operation in a Uni-Processor Platform

(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)

```
GETSEC[SENTER] (ILP only):
IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((in VMX root operation) or
    (CR0.PE=0) or (CR0.CD=1) or (CR0.NW=1) or (CR0.NE=0) or
    (CPL>0) or (EFLAGS.VM=1) or
    (IA32_APIC_BASE.BSP=0) or (TXT chipset not present) or
    (SENTERFLAG=1) or (ACMODEFLAG=1) or (IN_SMM=1) or
    (TPM interface is not present) or
    (EDX != (SENTER_EDX_support_mask & EDX)) or
    (IA32_CR_FEATURE_CONTROL[0]=0) or (IA32_CR_FEATURE_CONTROL[15]=0) or
    ((IA32_CR_FEATURE_CONTROL[14:8] & EDX[6:0]) != EDX[6:0]))
        THEN #GP(0);
FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
    IF IA32_MC[I]_STATUS = uncorrectable error
        THEN #GP(0);
    FI;
OD;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
    THEN #GP(0);
ACBASE← EBX;
ACSIZE← ECX;
IF (((ACBASE MOD 4096) != 0) or ((ACSIZE MOD 64) != 0 ) or (ACSIZE < minimum
    module size) or (ACSIZE > AC RAM capacity) or ((ACBASE+ACSIZE) > (2^32 -1)))
        THEN #GP(0);
Mask SMI, INIT, A20M, and NMI external pin events;
SignalTXTMsg(SENTER);
DO
WHILE (no SignalSENTER message);

TXT_SENTER__MSG_EVENT (ILP & RLP):
Mask and clear SignalSENTER event;
Unmask SignalSEXIT event;
IF (in VMX operation)
    THEN TXT-SHUTDOWN(#IllegalEvent);
FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
    IF IA32_MC[I]_STATUS = uncorrectable error
        THEN TXT-SHUTDOWN(#UnrecovMCError);
    FI;
OD;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
    THEN TXT-SHUTDOWN(#UnrecovMCError);
IF (Voltage or bus ratio status are NOT at a known good state)
    THEN IF (Voltage select and bus ratio are internally adjustable)
        THEN
            Make product-specific adjustment on operating parameters;
        ELSE
            TXT-SHUTDOWN(#IllegalVIDBRatio);
FI;

IA32_MISC_ENABLE← (IA32_MISC_ENABLE & MASK_CONST*)
(* The hexadecimal value of MASK_CONST may vary due to processor implementations *)
A20M← 0;
IA32_DEBUGCTL← 0;
Invalidate processor TLB(s);
Drain outgoing transactions;
Clear performance monitor counters and control;
SENTERFLAG← 1;
SignalTXTMsg(SENTERAck);
IF (logical processor is not ILP)
    THEN GOTO RLP_SENTER_ROUTINE;
(* ILP waits for all logical processors to ACK *)
DO
    DONE← TXT.READ(LT.STS);
WHILE (not DONE);
SignalTXTMsg(SENTERContinue);
SignalTXTMsg(ProcessorHold);
FOR I=ACBASE to ACBASE+ACSIZE-1 DO
    ACRAM[I-ACBASE].ADDR← I;
    ACRAM[I-ACBASE].DATA← LOAD(I);
OD;
IF (ACRAM memory type != WB)
    THEN TXT-SHUTDOWN(#BadACMMType);
IF (AC module header version is not supported) OR (ACRAM[ModuleType] <> 2)
    THEN TXT-SHUTDOWN(#UnsupportedACM);
KEY← GETKEY(ACRAM, ACBASE);
KEYHASH← HASH(KEY);
CSKEYHASH← LT.READ(LT.PUBLIC.KEY);
IF (KEYHASH <> CSKEYHASH)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
SIGNATURE← DECRYPT(ACRAM, ACBASE, KEY);
(* The value of SIGNATURE_LEN_CONST is implementation-specific*)
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.I]← SIGNATURE[I];
COMPUTEDSIGNATURE← HASH(ACRAM, ACBASE, ACSIZE);
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.SIGNATURE_LEN_CONST+I]← COMPUTEDSIGNATURE[I];
IF (SIGNATURE != COMPUTEDSIGNATURE)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
ACMCONTROL← ACRAM[CodeControl];
IF ((ACMCONTROL.0 = 0) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM
load))
    THEN TXT-SHUTDOWN(#UnexpectedHITM);
IF (ACMCONTROL reserved bits are set)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[GDTBasePtr] < (ACRAM[HeaderLen] * 4 + Scratch_size)) OR
    ((ACRAM[GDTBasePtr] + ACRAM[GDTLimit]) >= ACSIZE))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACMCONTROL.0 = 1) and (ACMCONTROL.1 = 1) and (snoop hit to modified
    line detected on ACRAM load))
    THEN ACEntryPoint← ACBASE+ACRAM[ErrorEntryPoint];
ELSE
    ACEntryPoint← ACBASE+ACRAM[EntryPoint];
IF ((ACEntryPoint >= ACSIZE) or (ACEntryPoint < (ACRAM[HeaderLen] * 4 + Scratch_size)))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel] > (ACRAM[GDTLimit] - 15)) or (ACRAM[SegSel] < 8))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel].TI=1) or (ACRAM[SegSel].RPL!=0))
    THEN TXT-SHUTDOWN(#BadACMFormat);
ACRAM[SCRATCH.SIGNATURE_LEN_CONST]← EDX;
WRITE(TPM.HASH.START)← 0;
FOR I=0 to SIGNATURE_LEN_CONST + 3 DO
    WRITE(TPM.HASH.DATA)← ACRAM[SCRATCH.I];
WRITE(TPM.HASH.END)← 0;
ACMODEFLAG← 1;
CR0.[PG.AM.WP]← 0;
CR4← 00004000h;
EFLAGS← 00000002h;
IA32_EFER← 0;
EBP← ACBASE;
GDTR.BASE← ACBASE+ACRAM[GDTBasePtr];
GDTR.LIMIT← ACRAM[GDTLimit];
CS.SEL← ACRAM[SegSel];
CS.BASE← 0;
CS.LIMIT← FFFFFh;
CS.G← 1;
CS.D← 1;
CS.AR← 9Bh;
DS.SEL← ACRAM[SegSel]+8;
DS.BASE← 0;
DS.LIMIT← FFFFFh;
DS.G← 1;
DS.D← 1;
DS.AR← 93h;
SS← DS;
ES← DS;
DR7← 00000400h;
IA32_DEBUGCTL← 0;
SignalTXTMsg(UnlockSMRAM);
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
EIP← ACEntryPoint;
END;

RLP_SENTER_ROUTINE: (RLP only)
Mask SMI, INIT, A20M, and NMI external pin events
```

```
Unmask SignalWAKEUP event;
Wait for SignalSENTERContinue message;
IA32_APIC_BASE.BSP← 0;
GOTO SENTER sleep state;
END;
```

...

GETSEC[WAKEUP]—Wake up sleeping processors in measured environment

...

Description

The GETSEC[WAKEUP] leaf function broadcasts a wake-up message to all logical processors currently in the SENTER sleep state. This GETSEC leaf must be executed only by the ILP, in order to wake-up the RLPs. Responding logical processors (RLPs) enter the SENTER sleep state after completion of the SENTER rendezvous sequence.

...

Operation

```
(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)
IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason=”GETSEC instruction”);
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or (SENTERFLAG=0) or (ACMODEFLAG=1) or
(IN_SMM=0) or (in VMX operation) or (IA32_APIC_BASE.BSP=0) or (TXT chipset not present))
    THEN #GP(0);
ELSE
    SignalTXTMsg(WAKEUP);
END;

RLP_SIPI_WAKEUP_FROM_SENTER_ROUTINE: (RLP only)
WHILE (no SignalWAKEUP event);
Mask SMI, A20M, and NMI external pin events (unmask INIT);
Mask SignalWAKEUP event;
Invalidate processor TLB(s);
Drain outgoing transactions;
TempGDTRLIMIT← LOAD(LT.MLE.JOIN);
TempGDTRBASE← LOAD(LT.MLE.JOIN+4);
TempSegSel← LOAD(LT.MLE.JOIN+8);
TempEIP← LOAD(LT.MLE.JOIN+12);
IF (TempGDTLimit & FFFF0000h)
    THEN TXT-SHUTDOWN(#BadJOINFormat);
IF ((TempSegSel > TempGDTRLIMIT-15) or (TempSegSel < 8))
    THEN TXT-SHUTDOWN(#BadJOINFormat);
IF ((TempSegSel.TI=1) or (TempSegSel.RPL!=0))
```

```
    THEN TXT-SHUTDOWN(#BadJOINFormat);
CR0.[PG,CD,W,AM,WP]← 0;
CR0.[NE,PE]← 1;
CR4← 00004000h;
EFLAGS← 00000002h;
IA32_EFER← 0;
GDTR.BASE← TempGDTRBASE;
GDTR.LIMIT← TempGDTRLIMIT;
CS.SEL← TempSegSel;
CS.BASE← 0;
CS.LIMIT← FFFFFh;
CS.G← 1;
CS.D← 1;
CS.AR← 9Bh;
DS.SEL← TempSegSel+8;
DS.BASE← 0;
DS.LIMIT← FFFFFh;
DS.G← 1;
DS.D← 1;
DS.AR← 93h;
SS← DS;
ES← DS;
DR7← 00000400h;
IA32_DEBUGCTL← 0;
EIP← TempEIP;
END;
```

12. Updates to Appendix A, Volume 2B

Change bars show changes to Appendix A of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B:* Instruction Set Reference, N-Z.

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

	8	9	A	B	C	D	E	F
0	OR						PUSH CS^{i64}	2-byte escape (Table A-3)
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
1	SBB						PUSH DS^{i64}	POP DS^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
2	SUB						SEG=CS (Prefix)	DAS^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
3	CMP						SEG=DS (Prefix)	AAS^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
4	DEC^{i64} general register / REX^{o64} Prefixes							
	eAX REX.W	eCX REX.WB	eDX REX.WX	eBX REX.WXB	eSP REX.WR	eBP REX.WRB	eSI REX.WRX	eDI REX.WRXB
5	POP^{d64} into general register							
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15

	8	9	A	B	C	D	E	F
6	PUSH^{d64} Iz	IMUL Gv, Ev, Iz	PUSH^{d64} Ib	IMUL Gv, Ev, Ib	INS/ INSB Yb, DX	INS/ INSW/ INSD Yz, DX	OUTS/ OUTSB DX, Xb	OUTS/ OUTSW/ OUTSD DX, Xz
7	Jcc^{f64}, Jb- Short displacement jump on condition							
	S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G
8	MOV				MOV Ev, Sw	LEA Gv, M	MOV Sw, Ew	Grp 1A^{1A} POP^{d64} Ev
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev				
9	CBW/ CWDE/ CDQE	CWD/ CDQ/ CQO	CALLF^{i64} Ap	FWAIT/ WAIT	PUSHF/D/Q ^{d64}/ Fv	POPF/D/Q ^{d64}/ Fv	SAHF	LAHF
A	TEST		STOS/B Yb, AL	STOS/W/D/Q Yv, rAX	LODS/B AL, Xb	LODS/W/D/Q rAX, Xv	SCAS/B AL, Yb	SCAS/W/D/Q rAX, Xv
	AL, Ib	rAX, Iz						
B	MOV immediate word or double into word, double, or quad register							
	rAX/r8, Iv	rCX/r9, Iv	rDX/r10, Iv	rBX/r11, Iv	rSP/r12, Iv	rBP/r13, Iv	rSI/r14, Iv	rDI/r15 , Iv
C	ENTER Iw, Ib	LEAVE^{d64}	RETF Iw	RETF	INT 3	INT Ib	INTO^{i64}	IRET/D/Q
D	ESC (Escape to coprocessor instruction set)							
E	CALL^{f64}	JMP			IN		OUT	
	Jz	near^{f64} Jz	far^{i64} Ap	short^{f64} Jb	AL, DX	eAX, DX	DX, AL	DX, eAX
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp 4^{1A}	INC/DEC Grp 5^{1A}

NOTES:

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

13. Updates to Chapter 3, Volume 3A

Change bars show changes to Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

3.4.5 Segment Descriptors

...

The flags and fields in a segment descriptor are as follows:

Segment limit field

Specifies the size of the segment. The processor puts together the two segment limit fields to form a 20-bit value. The processor interprets the segment limit in one of two ways, depending on the setting of the G (granularity) flag:

- If the granularity flag is clear, the segment size can range from 1 byte to 1 MByte, in byte increments.
- If the granularity flag is set, the segment size can range from 4 KBytes to 4 GBytes, in 4-KByte increments.

The processor uses the segment limit in two different ways, depending on whether the segment is an expand-up or an expand-down segment. See Section 3.4.5.1, "Code- and Data-Segment Descriptor Types", for more information about segment types. For expand-up segments, the offset in a logical address can range from 0 to the segment limit. Offsets greater than the segment limit generate general-protection exceptions (#GP, for all segment other than SS) or stack-fault exceptions (#SS for the SS segment). For expand-down segments, the segment limit has the reverse function; the offset can range from the segment limit plus 1 to FFFFFFFFH or FFFFH, depending on the setting of the B flag. Offsets less than or equal to the segment limit generate general-protection exceptions or stack-fault exceptions. Decreasing the value in the segment limit field for an expand-down segment allocates new memory at the bottom of the segment's address space, rather than at the top. IA-32 architecture stacks always grow downwards, making this mechanism convenient for expandable stacks.

...

14. Updates to Chapter 6, Volume 3A

Change bars show changes to Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

6.15 EXCEPTION AND INTERRUPT REFERENCE

...

Interrupt 17—Alignment Check Exception (#AC)

Exception Class Fault.

Description

Indicates that the processor detected an unaligned memory operand when alignment checking was enabled. Alignment checks are only carried out in data (or stack) accesses (not in code fetches or system segment accesses). An example of an alignment-check violation is a word stored at an odd byte address, or a doubleword stored at an address that is not an integer multiple of 4. Table 6-7 lists the alignment requirements various data types recognized by the processor.

Table 6-7 Alignment Requirements by Data Type

Data Type	Address Must Be Divisible By
Word	2
Doubleword	4
Single-precision floating-point (32-bits)	4

Table 6-7 Alignment Requirements by Data Type

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

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

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

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

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

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

The FXSAVE/XSAVE and FXRSTOR/XRSTOR instructions save and restore a 512-byte data structure, the first byte of which must be aligned on a 16-byte boundary. If the alignment-check exception (#AC) is enabled when executing these instructions (and CPL is 3), a misaligned memory operand can cause either an alignment-check exception or a general-protection exception (#GP) depending on the processor implementation (see "FXSAVE-Save x87 FPU, MMX, SSE, and SSE2 State" and "FXRSTOR-Restore x87 FPU, MMX, SSE, and SSE2 State" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*; see "*XSAVE—Save Processor Extended States*" and "*XRSTOR—Restore Processor Extended States*" in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*).

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

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

Exception Error Code

Yes (always zero).

Saved Instruction Pointer

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

Program State Change

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

15. Updates to Chapter 8, Volume 3A

Change bars show changes to Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A:* System Programming Guide, Part 1.

...

8.1.1 Guaranteed Atomic Operations

The Intel486 processor (and newer processors since) guarantees that the following basic memory operations will always be carried out atomically:

- Reading or writing a byte
- Reading or writing a word aligned on a 16-bit boundary
- Reading or writing a doubleword aligned on a 32-bit boundary

The Pentium processor (and newer processors since) guarantees that the following additional memory operations will always be carried out atomically:

- Reading or writing a quadword aligned on a 64-bit boundary
- 16-bit accesses to uncached memory locations that fit within a 32-bit data bus

The P6 family processors (and newer processors since) guarantee that the following additional memory operation will always be carried out atomically:

- Unaligned 16-, 32-, and 64-bit accesses to cached memory that fit within a cache line

Accesses to cacheable memory that are split across cache lines and page boundaries are not guaranteed to be atomic by the Intel Core 2 Duo, Intel® Atom™, Intel Core Duo, Pentium M, Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors. The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium M, Pentium 4, Intel Xeon, and P6 family processors provide bus control signals that permit external memory subsystems to make split accesses atomic; however, nonaligned data accesses will seriously impact the performance of the processor and should be avoided.

...

16. Updates to Chapter 10, Volume 3A

Change bars show changes to Chapter 10 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

10.5.1 Local Vector Table

The local vector table (LVT) allows software to specify the manner in which the local interrupts are delivered to the processor core. It consists of the following 32-bit APIC registers (see Figure 10-8), one for each local interrupt:

...

Timer Mode	Bits 18:17 selects the timer mode (see Section 10.5.4): (00b) one-shot mode using a count-down value, (01b) periodic mode reloading a count-down value, (10b) TSC-Deadline mode using absolute target value in IA32_TSC_DEADLINE MSR (see Section 10.5.4.1), (11b) is reserved.

...

10.5.4.1 TSC-Deadline Mode

If CPUID.01H:ECX.TSC_Deadline[bit 24] = 1, the local APIC timer mode is determined by bits 18:17 of the LVT Timer Register (see Figure 10-8). If CPUID.01H:ECX.TSC_Deadline[bit 24] = 0, the local APIC timer mode is determined by bit 17 of the LVT Timer Register; bit 18 of the register is reserved. A write to the LVT timer register that changes the timer mode disarms the local APIC timer. The supported timer mode is given in Table 10-3.

Table 10-3 Local APIC Timer Modes

LVT Bits [18:17]	Timer Mode
00b	One-shot mode, program count-down value in an initial-count register. See Section 10.5.4
01b	Periodic mode, program interval value in an initial-count register. See Section 10.5.4
10b	TSC-Deadline mode, program target value in IA32_TSC_DEADLINE MSR.
11b	Reserved

The TSC-Deadline mode allows software to use local APIC timer to generate one-shot event with an absolute target value. The three modes of the local APIC timer are mutually exclusive.

In TSC-Deadline mode, writes to the initial-count register are ignored; and current-count register always reads 0.

- IA32_TSC_DEADLINE MSR

  The TSC-Deadline mode local APIC timer is disarmed by writing 0 into the IA32_TSC_DEADLINE MSR. Writing a non-zero value into IA32_TSC_DEADLINE arms the timer. When the timer generates the timer event, it disarms by clearing IA32_TSC_DEADLINE. Transitioning between TSC-Deadline mode and other Local APIC timer modes always disarms the timer. A timer event is generated when the logical processor's timestamp counter equals or exceeds the target value in the IA32_TSC_DEADLINE MSR.

  IA32_TSC_DEADLINE is a per-logical processor MSR with non-serializing behavior, i.e., it has been optimized for low cost of access.

  Software specifies an unsigned 64-bit target value in the IA32_TSC_DEADLINE MSR for each logical processor that needs a one-shot timer event. Each logical processor may have a unique timer event. The target value represents an absolute time tick on which the timer event can be delivered to that logical processor.

  The hardware reset value of IA32_TSC_DEADLINE is 0. In Local APIC Timer mode (LVT bit 18 = 0), IA32_TSC_DEADLINE reads zero and writes are ignored.

- TSC-Deadline Mode Programming Model

  Software can configure the TSC-Deadline Timer to deliver a one shot timer event using the following algorithm:

  a. Detect TSC-Deadline Timer feature support by verifying CPUID.1:ECX.24 = 1.

  b. Select the TSC-Deadline Timer mode by programming bits 18:17 of the LVT Timer register with 10b.

  c. Program the IA32_TSC_DEADLINE MSR with the target TSC value when the timer tick is needed. This causes the processor to "arm" the TSC-Deadline Timer.

  d. The processor generates a timer event when the value of IA32_TSC MSR is greater than or equal to that of IA32_TSC_DEADLINE, then "disarms" the local APIC Timer and sets IA32_TSC_DEADLINE to 0. Both IA32_TSC MSR and IA32_TSC_DEADLINE are 64-bit unsigned integers.

  e. Software can re-arm the TSC-Deadline Timer by repeating step c.

- TSC-Deadline Mode Usage Guidelines

  a. Writes to the IA32_TSC_DEADLINE MSR are not serialized. Therefore, system software should not use "WRMSR to IA32_TSC_DEADLINE" as a serializing instruction. Read and write accesses to the IA32_TSC_DEADLINE and other MSR registers will occur in program order.

  b. Software can cancel or disarm the TSC-Deadline Timer at any time by writing a value of 0 to IA32_TSC_DEADLINE.

  c. If TSC-Deadline Timer is already in armed state, software can move TSC deadline forward or backward by writing the new TSC value to the IA32_TSC_DEADLINE MSR.

  d. If software cancels or moves the TSC deadline forward or backward, race conditions caused by "in-flight" interrupts can still result in the delivery of a spurious timer interrupt. Software is expected to detect the spurious interrupt by checking the current value in IA32_TSC MSR to see if the interrupt was expected.

  e. If the processor’s xAPIC is in the legacy mode (local APIC registers are programmed via MMIO interfaces), software must serialize between the MMIO write to the LVT entry and the MSR write to IA32_TSC_DEADLINE. Note that if the APIC is in the extended (X2APIC) mode, no serialization is required between the two MSR writes to the LVT and IA32_TSC_DEADLINE MSR.

An example for serializing writes in legacy xAPIC mode is shown below:

1. MMIO write to LVT Timer Register bit 18:17 = 10b.
2. WRMSR(IA32_TSC_DEADLINE) with a value much larger than current TSC.
3. If RDMSR of IA32_TSC_DEADLINE returns zero, go to step 2.
4. WRMSR(IA32_TSC_DEADLINE) with a desired TSC-Deadline Timer value.

...

10.12.6 System Software Transitions

This section describes implications for the x2APIC across system state transitions - specifically initialization and booting.

Support for the x2APIC architecture can be implemented in the local APIC unit. All existing PCI/MSI capable devices and IOxAPIC unit should work with the x2APIC extensions defined in this document. The x2APIC architecture also provides flexibility to cope with the underlying fabrics that connect the PCI devices, IOxAPICs and Local APIC units.

The extensions provided in this specification translate into modifications to:

- the local APIC unit,
- the underlying fabrics connecting Message Signaled Interrupts (MSI) capable PCI devices to local xAPICs,
- the underlying fabrics connecting the IOxAPICs to the local APIC units.

However no modifications are required to PCI or PCIe devices that support direct interrupt delivery to the processors via Message Signaled Interrupts. Similarly no modifications are required to the IOxAPIC. The routing of interrupts from these devices in x2APIC mode leverages the interrupt remapping architecture specified in the Intel® Virtualization Technology for Directed I/O, Rev 1.2 specification. As a result, BIOS must enumerate support for and software must enable interrupt remapping with Extended Interrupt Mode Enabled (EIME) before it enables x2APIC.

Modifications to ACPI interfaces to support x2APIC are described in Appendix A, “ACPI Extensions for x2APIC Support”, of the Intel® 64 Architecture x2APIC Specification.

The default will be for the BIOS to pass the control to the OS with the local x2APICs in xAPIC mode if all x2APIC IDs reported by CPUID.0BH:EDX are less than 255, and in x2APIC mode if there are any logical processor reporting its x2APIC ID at 255 or greater.

...

17. Updates to Chapter 14, Volume 3A

Change bars show changes to Chapter 14 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A:* System Programming Guide, Part 1.

...

14.5.3.1 Extension of Software Controlled Clock Modulation

Extension of the software controlled clock modulation facility supports on-demand clock modulation duty cycle with 4-bit dynamic range (increased from 3-bit range). Granularity of clock modulation duty cycle is increased to 6.25% (compared to 12.5%).

Four bit dynamic range control is provided by using bit 0 in conjunction with bits 3:1 of the IA32_CLOCK_MODULATION MSR (see Figure 14-11).

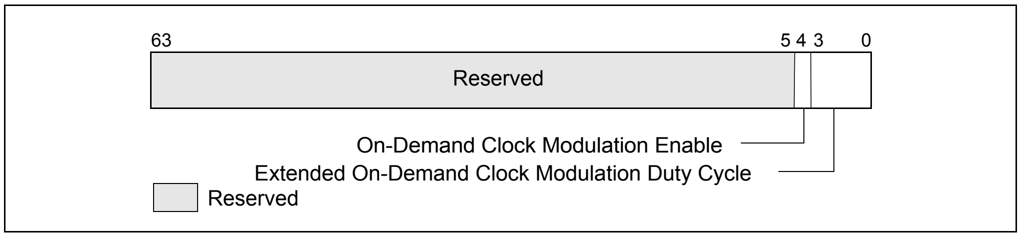
63
5 4 3
0
Reserved
On-Demand Clock Modulation Enable
Extended On-Demand Clock Modulation Duty Cycle
Reserved

Figure 14-11 IA32_CLOCK_MODULATION MSR with Clock Modulation Extension

Extension to software controlled clock modulation is supported only if CPUID.06H:EAX[Bit 5] = 1. If CPUID.06H:EAX[Bit 5] = 0, then bit 0 of IA32_CLOCK_MODULATION is reserved.

...

14.5.4.1 Detection of Software Controlled Clock Modulation Extension

Processor's support of software controlled clock modulation extension is indicated by CPUID.06H:EAX[Bit 5] = 1.

...

14.5.5.2 Reading the Digital Sensor

...

- **Power Limitation Status (bit 10, RO)** — Indicates whether the processor is currently operating below OS-requested P-state (specified in IA32_PERF_CTL) or OS-requested clock modulation duty cycle (specified in IA32_CLOCK_MODULATION). This field is supported only if CPUID.06H:EAX[bit 4] = 1. Package level power limit notification can be delivered independently to IA32_PACKAGE_THERM_STATUS MSR.
- **Power Notification Log (bit 11, R/WCO)** — Sticky bit that indicates the processor went below OS-requested P-state or OS-requested clock modulation duty cycle since the last clearing of this or RESET. This field is supported only if CPUID.06H:EAX[bit 4] = 1. Package level power limit notification is indicated independently in IA32_PACKAGE_THERM_STATUS MSR.

...

- **Power Limit Notification Enable (bit 24, R/W)** — Enables the generation of power notification events when the processor went below OS-requested P-state or OS-requested clock modulation duty cycle. This field is supported only if CPUID.06H:EAX[bit 4] = 1. Package level power limit notification can be enabled independently by IA32_PACKAGE_THERM_INTERRUPT MSR.

14.5.6 Power Limit Notification

Platform firmware may be capable of specifying a power limit to restrict power delivered to a platform component, such as a physical processor package. This constraint imposed by platform firmware may occasionally cause the processor to operate below OS-requested P or T-state. A power limit notification event can be delivered using the existing thermal LVT entry in the local APIC.

Software can enumerate the presence of the processor's support for power limit notification by verifying CPUID.06H:EAX[bit 4] = 1.

If CPUID.06H:EAX[bit 4] = 1, then IA32_THERM_INTERRUPT and IA32_THERM_STATUS provides the following facility to manage power limit notification:

- Bits 10 and 11 in IA32_THERM_STATUS informs software of the occurrence of processor operating below OS-requested P-state or clock modulation duty cycle setting (see Figure 14-12).
- Bit 24 in IA32_THERM_INTERRUPT enables the local APIC to deliver a thermal event when the processor went below OS-requested P-state or clock modulation duty cycle setting (see Figure 14-13).

14.6 PACKAGE LEVEL THERMAL MANAGEMENT

The thermal management facilities like IA32_THERM_INTERRUPT and IA32_THERM_STATUS are often implemented with a processor core granularity. To facilitate software manage thermal events from a package level granularity, two architectural MSR is provided for package level thermal management. The IA32_PACKAGE_THERM_STATUS and IA32_PACKAGE_THERM_INTERRUPT MSRs use similar interfaces as IA32_THERM_STATUS and IA32_THERM_INTERRUPT, but are shared in each physical processor package.

Software can enumerate the presence of the processor's support for package level thermal management facility (IA32_PACKAGE_THERM_STATUS and IA32_PACKAGE_THERM_INTERRUPT) by verifying CPUID.06H:EAX[bit 6] = 1.

The layout of IA32_PACKAGE_THERM_STATUS MSR is shown in Figure 14-14.

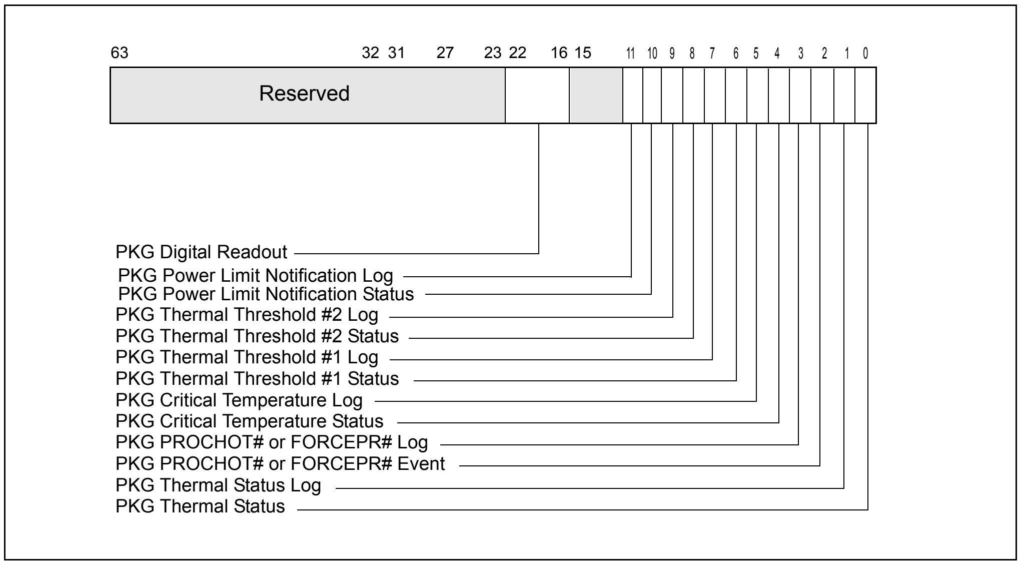
63
32 31
27
23 22
16 15
11 10 9 8 7 6 5 4 3 2 1 0
Reserved
PKG Digital Readout
PKG Power Limit Notification Log
PKG Power Limit Notification Status
PKG Thermal Threshold #2 Log
PKG Thermal Threshold #2 Status
PKG Thermal Threshold #1 Log
PKG Thermal Threshold #1 Status
PKG Critical Temperature Log
PKG Critical Temperature Status
PKG PROCHOT# or FORCEPR# Log
PKG PROCHOT# or FORCEPR# Event
PKG Thermal Status Log
PKG Thermal Status

Figure 14-14 IA32_PACKAGE_THERM_STATUS Register

- **Package Thermal Status (bit 0, RO)** — This bit indicates whether the digital thermal sensor high-temperature output signal (PROCHOT#) for the package is currently active. Bit 0 = 1 indicates the feature is active. This bit may not be written by software; it reflects the state of the digital thermal sensor.
- **Package Thermal Status Log (bit 1, R/WC0)** — This is a sticky bit that indicates the history of the thermal sensor high temperature output signal (PROCHOT#) of the package. Bit 1 = 1 if package PROCHOT# has been asserted since a previous RESET or the last time software cleared the bit. Software may clear this bit by writing a zero.
- **Package PROCHOT# Event (bit 2, RO)** — Indicates whether package PROCHOT# is being asserted by another agent on the platform.
- **Package PROCHOT# Log (bit 3, R/WC0)** — Sticky bit that indicates whether package PROCHOT# has been asserted by another agent on the platform since the last clearing of this bit or a reset. If bit 3 = 1, package PROCHOT# has been externally asserted. Software may clear this bit by writing a zero.
- **Package Critical Temperature Status (bit 4, RO)** — Indicates whether the package critical temperature detector output signal is currently active. If bit 4 = 1, the package critical temperature detector output signal is currently active.
- **Package Critical Temperature Log (bit 5, R/WC0)** — Sticky bit that indicates whether the package critical temperature detector output signal has been asserted since the last clearing of this bit or reset. If bit 5 = 1, the output signal has been asserted. Software may clear this bit by writing a zero.
- **Package Thermal Threshold #1 Status (bit 6, RO)** — Indicates whether the actual package temperature is currently higher than or equal to the value set in Package Thermal Threshold #1. If bit 6 = 0, the actual temperature is lower. If bit 6 = 1, the actual temperature is greater than or equal to PTT#1. Quantitative information of actual package temperature can be inferred from Package Digital Readout, bits 22:16.

- **Package Thermal Threshold #1 Log (bit 7, R/WC0)** — Sticky bit that indicates whether the Package Thermal Threshold #1 has been reached since the last clearing of this bit or a reset. If bit 7 = 1, the Package Threshold #1 has been reached. Software may clear this bit by writing a zero.
- **Package Thermal Threshold #2 Status (bit 8, RO)** — Indicates whether actual package temperature is currently higher than or equal to the value set in Package Thermal Threshold #2. If bit 8 = 0, the actual temperature is lower. If bit 8 = 1, the actual temperature is greater than or equal to PTT#2. Quantitative information of actual temperature can be inferred from Package Digital Readout, bits 22:16.
- **Package Thermal Threshold #2 Log (bit 9, R/WC0)** — Sticky bit that indicates whether the Package Thermal Threshold #2 has been reached since the last clearing of this bit or a reset. If bit 9 = 1, the Package Thermal Threshold #2 has been reached. Software may clear this bit by writing a zero.
- **Package Power Limitation Status (bit 10, RO)** — Indicates package power limit is forcing one ore more processors to operate below OS-requested P-state. Note that package power limit violation may be caused by processor cores or by devices residing in the uncore. Software can examine IA32_THERM_STATUS to determine if the cause originates from a processor core (see Figure 14-12).
- **Package Power Notification Log (bit 11, R/WCO)** — Sticky bit that indicates any processor in the package went below OS-requested P-state or OS-requested clock modulation duty cycle since the last clearing of this or RESET.
- **Package Digital Readout (bits 22:16, RO)** — Package digital temperature reading in 1 degree Celsius relative to the package TCC activation temperature.

  0: Package TCC Activation temperature,

  1: (PTCC Activation - 1) , etc. See the processor’s data sheet for details regarding PTCC activation.

  A lower reading in the Package Digital Readout field (bits 22:16) indicates a higher actual temperature.

The layout of IA32_PACKAGE_THERM_INTERRUPT MSR is shown in Figure 14-15.

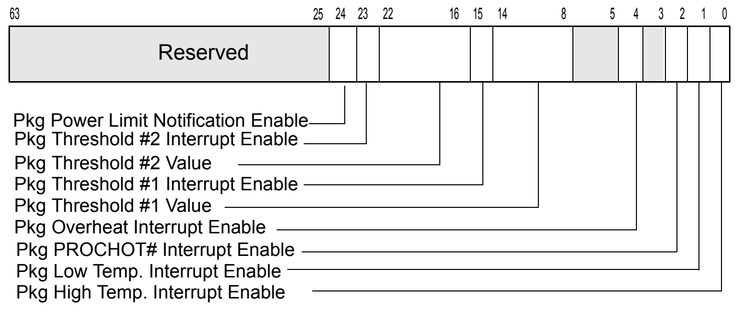
63
25
24
23
22
16
15
14
8
5
4
3
2
1
0
Reserved
Pkg Power Limit Notification Enable
Pkg Threshold #2 Interrupt Enable
Pkg Threshold #2 Value
Pkg Threshold #1 Interrupt Enable
Pkg Threshold #1 Value
Pkg Overheat Interrupt Enable
Pkg PROCHOT# Interrupt Enable
Pkg Low Temp. Interrupt Enable
Pkg High Temp. Interrupt Enable

Figure 14-15 IA32_PACKAGE_THERM_INTERRUPT Register

- **Package High-Temperature Interrupt Enable (bit 0, R/W)** — This bit allows the BIOS to enable the generation of an interrupt on the transition from low-temperature to a package high-temperature threshold. Bit 0 = 0 (default) disables interrupts; bit 0 = 1 enables interrupts.

- **Package Low-Temperature Interrupt Enable (bit 1, R/W)** — This bit allows the BIOS to enable the generation of an interrupt on the transition from high-temperature to a low-temperature (TCC de-activation). Bit 1 = 0 (default) disables interrupts; bit 1 = 1 enables interrupts.
- **Package PROCHOT# Interrupt Enable (bit 2, R/W)** — This bit allows the BIOS or OS to enable the generation of an interrupt when Package PROCHOT# has been asserted by another agent on the platform and the Bidirectional Prochot feature is enabled. Bit 2 = 0 disables the interrupt; bit 2 = 1 enables the interrupt.
- **Package Critical Temperature Interrupt Enable (bit 4, R/W)** — Enables the generation of an interrupt when the Package Critical Temperature Detector has detected a critical thermal condition. The recommended response to this condition is a system shutdown. Bit 4 = 0 disables the interrupt; bit 4 = 1 enables the interrupt.
- **Package Threshold #1 Value (bits 14:8, R/W)** — A temperature threshold, encoded relative to the Package TCC Activation temperature (using the same format as the Digital Readout). This threshold is compared against the Package Digital Readout and is used to generate the Package Thermal Threshold #1 Status and Log bits as well as the Package Threshold #1 thermal interrupt delivery.
- **Package Threshold #1 Interrupt Enable (bit 15, R/W)** — Enables the generation of an interrupt when the actual temperature crosses the Package Threshold #1 setting in any direction. Bit 15 = 0 enables the interrupt; bit 15 = 1 disables the interrupt.
- **Package Threshold #2 Value (bits 22:16, R/W)** —A temperature threshold, encoded relative to the PTCC Activation temperature (using the same format as the Package Digital Readout). This threshold is compared against the Package Digital Readout and is used to generate the Package Thermal Threshold #2 Status and Log bits as well as the Package Threshold #2 thermal interrupt delivery.
- **Package Threshold #2 Interrupt Enable (bit 23, R/W)** — Enables the generation of an interrupt when the actual temperature crosses the Package Threshold #2 setting in any direction. Bit 23 = 0 enables the interrupt; bit 23 = 1 disables the interrupt.
- **Package Power Limit Notification Enable (bit 24, R/W)** — Enables the generation of package power notification events.

14.6.1 Support for Passive and Active cooling

Passive and active cooling may be controlled by the OS power management agent through ACPI control methods. On platforms providing package level thermal management facility described in the previous section, it is recommended that active cooling (FAN control) should be driven by measuring the package temperature using the IA32_PACKAGE_THERM_INTERRUPT MSR.

Passive cooling (frequency throttling) should be driven by measuring (a) the core and package temperatures, or (b) only the package temperature. If measured package temperature led the power management agent to choose which core to execute passive cooling, then all cores need to execute passive cooling. Core temperature is measured using the IA32_THERMAL_STATUS and IA32_THERMAL_INTERRUPT MSRs. The exact implementation details depend on the platform firmware and possible solutions include defining two different thermal zones (one for core temperature and passive cooling and the other for package temperature and active cooling).

14.7 PLATFORM SPECIFIC POWER MANAGEMENT SUPPORT

This section covers power management interfaces that are not architectural but addresses the power management needs of several platform specific components. Specifically, RAPL (Running Average Power Limit) interfaces provide mechanisms to enforce power consumption limit. Power limiting usages have specific usages in client and server platforms.

For client platform power limit control and for server platforms used in a data center, the following power and thermal related usages are desirable:

- Platform Thermal Management: Robust mechanisms to manage component, platform, and group-level thermals, either proactively or reactively (e.g., in response to a platform-level thermal trip point).
- Platform Power Limiting: More deterministic control over the system's power consumption, for example to meet battery life targets on rack- or container-level power consumption goals within a datacenter.
- Power/Performance Budgeting: Efficient means to control the power consumed (and therefore the sustained performance delivered) within and across platforms.

The server and client usage models are addressed by RAPL interfaces, which exposes multiple domains of power rationing within each processor socket. Generally, these RAPL domains may be viewed to include hierarchically:

- Package domain is the processor die.
- Memory domain include the directly-attached DRAM; additional power plane may constitutes a separate domain.

In order to manage the power consumed across multiple sockets via RAPL, individual limits must be programmed for each processor complex. Programming specific RAPL domain across multiple sockets is not supported.

14.7.1 RAPL Interfaces

RAPL interfaces consist of non-architectural MSRs. Each RAPL domain supports the following set of capabilities, some of which are optional as stated below.

- Power limit - MSR interfaces to specify power limit, time window; lock bit, clamp bit etc.
- Energy Status - Power metering interface providing energy consumption information.
- Perf Status (Optional) - Interface providing information on the performance effects (regression) due to power limits. It is defined as a duration metric that measures the power limit effect in the respective domain. The meaning of duration is domain specific.
- Power Info (Optional) - Interface providing information on the range of parameters for a given domain, minimum power, maximum power etc.
- Policy (Optional) - 4-bit priority information which is a hint to hardware for dividing budget between sub-domains in a parent domain.

Each of the above capabilities requires specific units in order to describe them. Power is expressed in Watts, Time is expressed in Seconds and Energy is expressed in Joules. Scaling factors are supplied to each unit to make the information presented meaningful in a finite number of bits. Units for power, energy and time are exposed in the read-only MSR_RAPL_POWER_UNIT MSR.

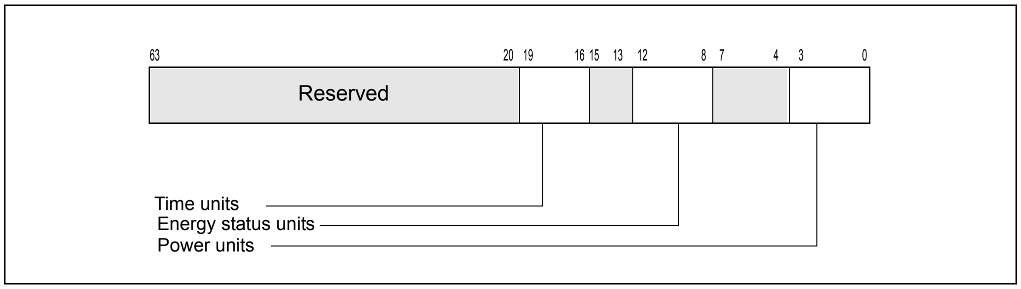

Figure 14-16 MSR_RAPL_POWER_UNIT Register

MSR_RAPL_POWER_UNIT (Figure 14-16) provides the following information across all RAPL domains:

- **Power Units** (bits 3:0): Power related information (in Watts) is based on the multiplier, 1/ 2^PU; where PU is an unsigned integer represented by bits 3:0. Default value is 0011b, indicating power unit is in 1/8 Watts increment.
- **Energy Status Units** (bit 12:8): Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 10000b, indicating energy status unit is in 15.3 micro-Joules increment.
- **Time Units** (bits 19:16): Time related information (in Seconds) is based on the multiplier, 1/ 2^TU; where TU is an unsigned integer represented by bits 19:16. Default value is 1010b, indicating time unit is in 976 micro-seconds increment.

14.7.2 RAPL Domains and Platform Specificity

The specific RAPL domains available in a platform varies across product segments. Platforms targeting client segment support the following RAPL domain hierarchy:

- Package
- Two power planes: PP0 and PP1 (PP1 may reflect to uncore devices)

Platforms targeting server segment support the following RAPL domain hierarchy:

- Package
- Power plane: PP0
- DRAM

Each level of the RAPL hierarchy provides respective set of RAPL interface MSRs. Table 14-2 lists the RAPL MSR interfaces available for each RAPL domain. The power limit MSR of each RAPL domain is located at offset 0 relative to an MSR base address which is non-architectural (see Appendix B). The energy status MSR of each domain is located at offset 1 relative to the MSR base address of respective domain.

Table 14-2 RAPL MSR Interfaces and RAPL Domains

Domain	Power Limit (Offset 0)	Energy Status (Offset 1)	Policy (Offset 2)	Perf Status (Offset 3)	Power Info (Offset 4)

Table 14-2 RAPL MSR Interfaces and RAPL Domains

PKG	MSR_PKG_POWER_LIMIT	MSR_PKG_ENERGY_STATUS	RESERVED	MSR_PKG_RAPL_PERF_STATUS	MSR_PKG_POWER_INFO
DRAM	MSR_DRAM_POWER_LIMIT	MSR_DRAM_ENERGY_STATUS	RESERVED	MSR_DRAM_RAPL_PERF_STATUS	MSR_DRAM_POWER_INFO
PP0	MSR_PP0_POWER_LIMIT	MSR_PP0_ENERGY_STATUS	MSR_PP0_POLICY	RESERVED	RESERVED
PP1	MSR_PP1_POWER_LIMIT	MSR_PP1_ENERGY_STATUS	MSR_PP1_POLICY	RESERVED	RESERVED

The presence of the optional MSR interfaces (the three right-most columns of Table 14-2) may be model-specific. See Appendix B for detail.

14.7.3 Package RAPL Domain

The MSR interfaces defined for the package RAPL domain are:

- MSR_PKG_POWER_LIMIT allows software to set power limits for the package and measurement attributes associated with each limit,
- MSR_PKG_ENERGY_STATUS reports measured actual energy usage,
- MSR_PKG_POWER_INFO reports the package power range information for RAPL usage.

MSR_PKG_RAPL_PERF_STATUS can report the performance impact of power limiting, but its availability may be model-specific

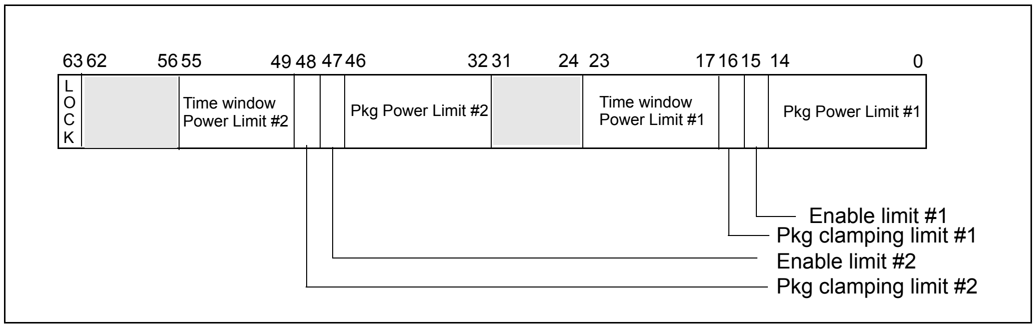

Figure 14-17 MSR_PKG_POWER_LIMIT Register

MSR_PKG_POWER_LIMIT allows a software agent to define power limitation for the package domain. Power limitation is defined in terms of average power usage (Watts) over a time window specified in MSR_PKG_POWER_LIMIT. Two power limits can be specified, corresponding to time windows of different sizes. Each power limit provides independent clamping control that would permit the processor cores to go below OS-requested state to meet the power limits. A lock mechanism allow the software agent to enforce power limit settings. Once the lock bit is set, the power limit settings are static and un-modifiable until next RESET.

The bit fields of MSR_PKG_POWER_LIMIT (Figure 14-17) are:

- **Package Power Limit #1**(bits 14:0): Sets the average power usage limit of the package domain corresponding to time window # 1. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Enable Power Limit #1**(bit 15): 0 = disabled; 1 = enabled.
- **Package Clamping Limitation #1** (bits 16): Allow going below OS-requested P/T state setting during time window specified by bits 23:17.
- **Time Window for Power Limit #1** (bits 23:17): Indicates the length of time window over which the power limit #1 The numeric value encoded by bits 23:17 is represented by the product of 2^Y *F; where F is a single-digit decimal floating-point value between 1.0 and 1.3 with the fraction digit represented by bits 23:22, Y is an unsigned integer represented by bits 21:17. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.
- **Package Power Limit #2**(bits 46:32): Sets the average power usage limit of the package domain corresponding to time window # 2. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Enable Power Limit #2**(bit 47): 0 = disabled; 1 = enabled.
- **Package Clamping Limitation #2** (bits 48): Allow going below OS-requested P/T state setting during time window specified by bits 23:17.
- **Time Window for Power Limit #2** (bits 55:49): Indicates the length of time window over which the power limit #2 The numeric value encoded by bits 23:17 is represented by the product of 2^Y *F; where F is a single-digit decimal floating-point value between 1.0 and 1.3 with the fraction digit represented by bits 23:22, Y is an unsigned integer represented by bits 21:17. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT. This field may have a hard-coded value in hardware and ignores values written by software.
- **Lock** (bits 63): If set, all write attempts to this MSR are ignored until next RESET.

MSR_PKG_ENERGY_STATUS is a read-only MSR. It reports the actual energy use for the package domain. This MSR is updated every ~1msec. It has a wraparound time of around 60 secs when power consumption is high, and may be longer otherwise.

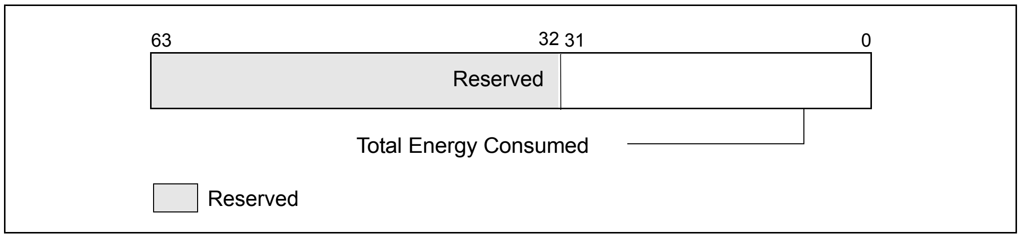

Figure 14-18 MSR_PKG_ENERGY_STATUS MSR

- **Total Energy Consumed** (bits 31:0): The unsigned integer value represents the total amount of energy consumed since that last time this register is cleared. The unit of this field is specified by the "Energy Status Units" field of MSR_RAPL_POWER_UNIT.

MSR_PKG_POWER_INFO is a read-only MSR. It reports the package power range information for RAPL usage. This MSR provides maximum/minimum values (derived from electrical specification), thermal specification power of the package domain. It also provides the largest possible time window for software to program the RAPL interface.

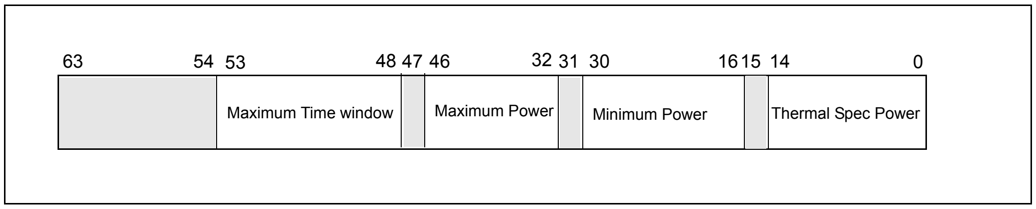

Figure 14-19 MSR_PKG_POWER_INFO Register

- **Thermal Spec Power** (bits 14:0): The unsigned integer value is the equivalent of thermal specification power of the package domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Minimum Power** (bits 30:16): The unsigned integer value is the equivalent of minimum power derived from electrical spec of the package domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Maximum Power** (bits 46:32): The unsigned integer value is the equivalent of maximum power derived from the electrical spec of the package domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Maximum Time Window** (bits 46:32): The unsigned integer value is the equivalent of largest acceptable value to program the time window of MSR_PKG_POWER_LIMIT. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.

MSR_PKG_PERF_STATUS is a read-only MSR. It reports the total time for which the package was throttled due to the RAPL power limits. Throttling in this context is defined as going below the OS-requested P-state or T-state. It has a wrap-around time of many hours. The availability of this MSR is platform specific (see Appendix B).

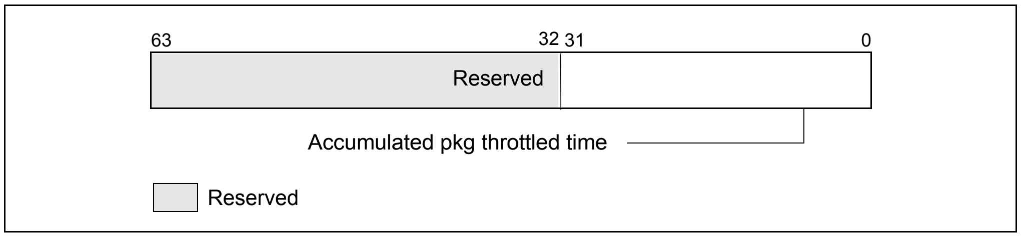

Figure 14-20 MSR_PKG_PERF_STATUS MSR

- **Accumulated Package Throttled Time** (bits 31:0): The unsigned integer value represents the cumulative time (since the last time this register is cleared) that the package has throttled. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.

14.7.4 PP0/PP1 RAPL Domains

The MSR interfaces defined for the PP0 and PP1 domains are identical in layout. Generally, PP0 refers to the processor cores. The availability of PP1 RAPL domain interface is platform-specific. For a client platform, PP1 domain refers to the power plane of a specific device in the uncore. For server platforms, PP1 domain is not supported, but its PP0 domain supports the MSR_PP0_PERF_STATUS interface.

- MSR_PP0_POWER_LIMIT/MSR_PP1_POWER_LIMIT allow software to set power limits for the respective power plane domain.
- MSR_PP0_ENERGY_STATUS/MSR_PP1_ENERGY_STATUS report actual energy usage on a power plane.
- MSR_PP0_POLICY/MSR_PP1_POLICY allow software to adjust balance for respective power plane.

MSR_PP0_PERF_STATUS can report the performance impact of power limiting, but it is not available in client platform.

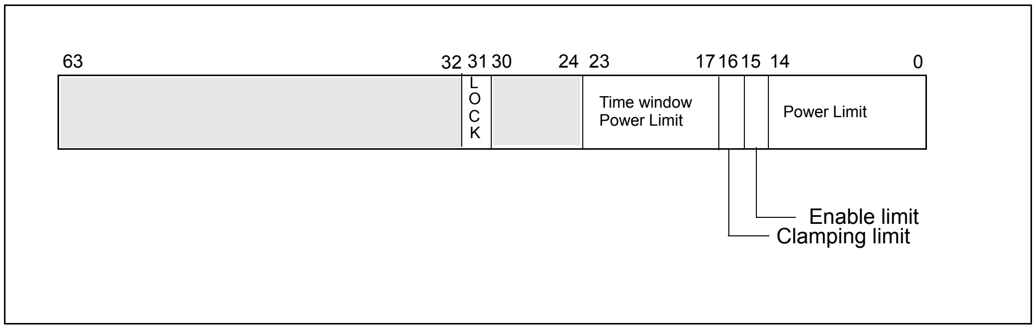

Figure 14-21 MSR_PP0_POWER_LIMIT/MSR_PP1_POWER_LIMIT Register

MSR_PP0_POWER_LIMIT/MSR_PP1_POWER_LIMIT allows a software agent to define power limitation for the respective power plane domain. A lock mechanism in each power plane domain allow the software agent to enforce power limit settings independently. Once a lock bit is set, the power limit settings in that power plane are static and un-modifiable until next RESET.

The bit fields of MSR_PP0_POWER_LIMIT/MSR_PP1_POWER_LIMIT (Figure 14-21) are:

- **Power Limit** (bits 14:0): Sets the average power usage limit of the respective power plane domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Enable Power Limit** (bit 15): 0 = disabled; 1 = enabled.
- **Clamping Limitation** (bits 16): Allow going below OS-requested P/T state setting during time window specified by bits 23:17.
- **Time Window for Power Limit** (bits 23:17): Indicates the length of time window over which the power limit #1 The numeric value encoded by bits 23:17 is represented by the product of 2^Y *F; where F is a single-digit decimal floating-point value between 1.0 and 1.3 with the fraction digit represented by bits 23:22, Y is an unsigned integer represented by bits 21:17. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.
- **Lock** (bits 63): If set, all write attempts to the MSR and corresponding policy MSR_PP0_POLICY/MSR_PP1_POLICY are ignored until next RESET.

MSR_PP0_ENERGY_STATUS/MSR_PP1_ENERGY_STATUS is a read-only MSR. It reports the actual energy use for the respective power plane domain. This MSR is updated every ~1msec.

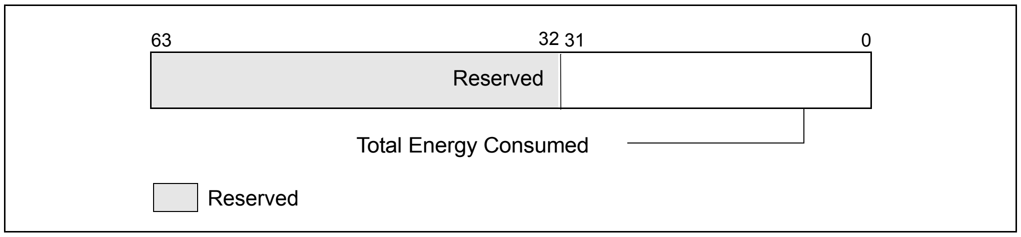

Figure 14-22 MSR_PP0_ENERGY_STATUS/MSR_PP1_ENERGY_STATUS MSR

- **Total Energy Consumed** (bits 31:0): The unsigned integer value represents the total amount of energy consumed since that last time this register is cleared. The unit of this field is specified by the "Energy Status Units" field of MSR_RAPL_POWER_UNIT.

MSR_PP0_POLICY/MSR_PP1_POLICY provide balance power policy control for each power plane by providing inputs to the power budgeting management algorithm. On the platform that supports PP0 (IA cores) and PP1 (uncore graphic device), the default value give priority to the non-IA power plane. These MSRs enable the PCU to balance power consumption between the IA cores and uncore graphic device.

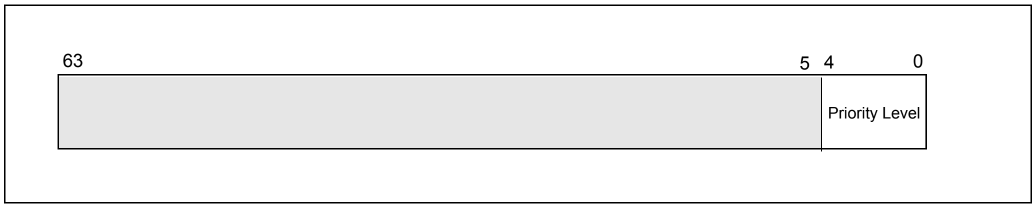

Figure 14-23 MSR_PP0_POLICY/MSR_PP1_POLICY Register

- **Priority Level** (bits 4:0): Priority level input to the PCU for respective power plane. PP0 covers the IA processor cores, PP1 covers the uncore graphic device. The value 31 is considered highest priority.

MSR_PP0_PERF_STATUS is a read-only MSR. It reports the total time for which the PP0 domain was throttled due to the power limits. This MSR is supported only in server platform. Throttling in this context is defined as going below the OS-requested P-state or T-state.

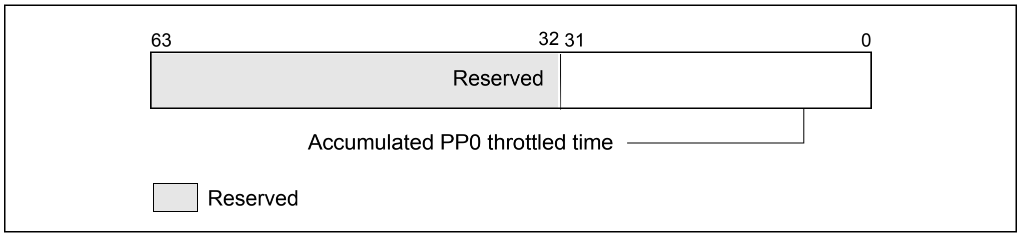

Figure 14-24 MSR_PP0_PERF_STATUS MSR

- **Accumulated PP0 Throttled Time** (bits 31:0): The unsigned integer value represents the cumulative time (since the last time this register is cleared) that the PP0 domain has throttled. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.

14.7.5 DRAM RAPL Domain

The MSR interfaces defined for the DRAM domain is supported only in the server platform. The MSR interfaces are:

- MSR_DRAM_POWER_LIMIT allows software to set power limits for the DRAM domain and measurement attributes associated with each limit,
- MSR_DRAM_ENERGY_STATUS reports measured actual energy usage,
- MSR_DRAM_POWER_INFO reports the DRAM domain power range information for RAPL usage.
- MSR_DRAM_RAPL_PERF_STATUS can report the performance impact of power limiting.

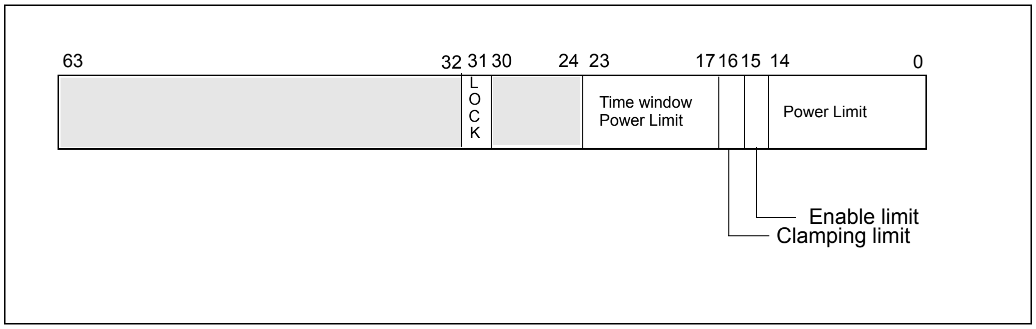

Figure 14-25 MSR_DRAM_POWER_LIMIT Register

MSR_DRAM_POWER_LIMIT allows a software agent to define power limitation for the DRAM domain. Power limitation is defined in terms of average power usage (Watts) over a time window specified in MSR_DRAM_POWER_LIMIT. A power limit can be specified along with a time window. A lock mechanism allow the software agent to enforce power limit settings. Once the lock bit is set, the power limit settings are static and un-modifiable until next RESET.

The bit fields of MSR_DRAM_POWER_LIMIT (Figure 14-17) are:

- **DRAM Power Limit #1**(bits 14:0): Sets the average power usage limit of the DRAM domain corresponding to time window # 1. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Enable Power Limit #1**(bit 15): 0 = disabled; 1 = enabled.
- **Time Window for Power Limit** (bits 23:17): Indicates the length of time window over which the power limit The numeric value encoded by bits 23:17 is represented by the product of 2^Y *F; where F is a single-digit decimal floating-point value between 1.0 and 1.3 with the fraction digit represented by bits 23:22, Y is an unsigned integer represented by bits 21:17. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.
- **Lock** (bits 63): If set, all write attempts to this MSR are ignored until next RESET.

MSR_DRAM_ENERGY_STATUS is a read-only MSR. It reports the actual energy use for the DRAM domain. This MSR is updated every ~1msec.

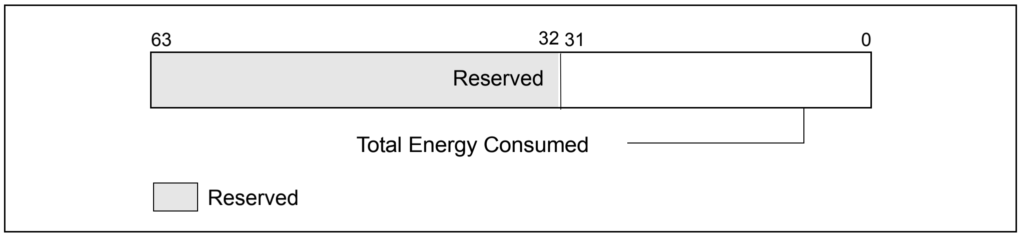

Figure 14-26 MSR_DRAM_ENERGY_STATUS MSR

- **Total Energy Consumed** (bits 31:0): The unsigned integer value represents the total amount of energy consumed since that last time this register is cleared. The unit of this field is specified by the "Energy Status Units" field of MSR_RAPL_POWER_UNIT.

MSR_DRAM_POWER_INFO is a read-only MSR. It reports the DRAM power range information for RAPL usage. This MSR provides maximum/minimum values (derived from electrical specification), thermal specification power of the DRAM domain. It also provides the largest possible time window for software to program the RAPL interface.

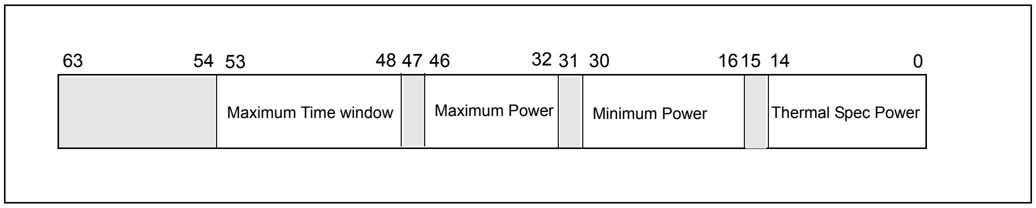

Figure 14-27 MSR_DRAM_POWER_INFO Register

- **Thermal Spec Power** (bits 14:0): The unsigned integer value is the equivalent of thermal specification power of the DRAM domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Minimum Power** (bits 30:16): The unsigned integer value is the equivalent of minimum power derived from electrical spec of the DRAM domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.

- **Maximum Power** (bits 46:32): The unsigned integer value is the equivalent of maximum power derived from the electrical spec of the DRAM domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT.
- **Maximum Time Window** (bits 46:32): The unsigned integer value is the equivalent of largest acceptable value to program the time window of MSR_DRAM_POWER_LIMIT. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.

MSR_DRAM_PERF_STATUS is a read-only MSR. It reports the total time for which the package was throttled due to the RAPL power limits. Throttling in this context is defined as going below the OS-requested P-state or T-state. It has a wrap-around time of many hours. The availability of this MSR is platform specific (see Appendix B).

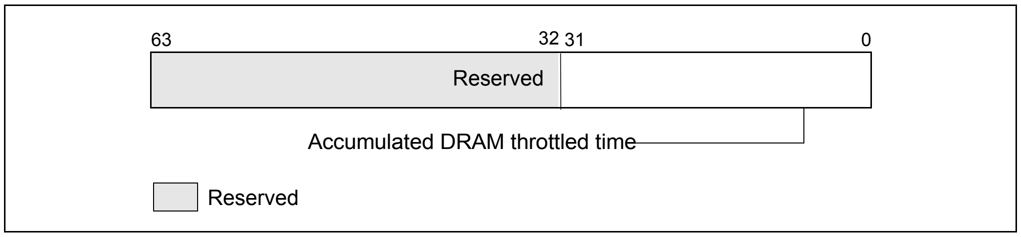

Figure 14-28 MSR_DRAM_PERF_STATUS MSR

- **Accumulated Package Throttled Time** (bits 31:0): The unsigned integer value represents the cumulative time (since the last time this register is cleared) that the DRAM domain has throttled. The unit of this field is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT.

...

18. Updates to Chapter 16, Volume 3A

Change bars show changes to Chapter 16 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

16.4.4 Branch Trace Messages

Setting the TR flag (bit 6) in the IA32_DEBUGCTL MSR enables branch trace messages (BTMs). Thereafter, when the processor detects a branch, exception, or interrupt, it sends a branch record out on the system bus as a BTM. A debugging device that is monitoring the system bus can read these messages and synchronize operations with taken branch, interrupt, and exception events.

When interrupts or exceptions occur in conjunction with a taken branch, additional BTMs are sent out on the bus, as described in Section 16.4.2, "Monitoring Branches, Exceptions, and Interrupts."

For IA processor families starting with Pentium 4, Pentium M and through Intel Core 2 processors and most of the initial Atom processor family, the processor can collect branch records in the LBR stack and at the same time sending/storing BTMs when both the TR and LBR flags are set in the IA32_DEBUGCTL MSR (or the equivalent MSR_DEBUGCTLA, MSR_DEBUGCTLB). The following exceptions apply:

- The content of LBR stack is undefined when TR is set for the P6 processor family;
- BTM may not be observable on Intel Atom processor family processors that do not provide an externally visible system bus.

...

16.4.9 BTS and DS Save Area

The **Debug store (DS)** feature flag (bit 21), returned by CPUID.1:EDX[21] Indicates that the processor provides the debug store (DS) mechanism. This mechanism allows BTMs to be stored in a memory-resident BTS buffer. See Section 16.4.5, "Branch Trace Store (BTS)." Precise event-based sampling (PEBS, see Section 30.4.4, "Precise Event Based Sampling (PEBS),") also uses the DS save area provided by debug store mechanism. When CPUID.1:EDX[21] is set, the following BTS facilities are available:

- The BTS_UNAVAILABLE flag in the IA32_MISC_ENABLE MSR indicates (when clear) the availability of the BTS facilities, including the ability to set the BTS and BTINT bits in the MSR_DEBUGCTLA MSR.
- The IA32_DS_AREA MSR can be programmed to point to the DS save area.

The debug store (DS) save area is a software-designated area of memory that is used to collect the following two types of information:

- **Branch records —** When the BTS flag in the IA32_DEBUGCTL MSR is set, a branch record is stored in the BTS buffer in the DS save area whenever a taken branch, interrupt, or exception is detected.
- **PEBS records —** When a performance counter is configured for PEBS, a PEBS record is stored in the PEBS buffer in the DS save area after the counter overflow occurs. This record contains the architectural state of the processor (state of the 8 general purpose registers, EIP register, and EFLAGS register) at the next occurrence of the PEBS event that caused the counter to overflow. When the state information has been logged, the counter is automatically reset to a preselected value, and event counting begins again.

NOTE

On processors based on Intel Core microarchitecture, PEBS is supported only for a subset of the performance events. In Intel Atom processor family, all performance monitoring events can be programmed to use PEBS.

...

16.5.1 LBR Stack

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported across Intel Core 2, Intel Xeon and Intel Atom processor families.

Four pairs of MSRs are supported in the LBR stack for Intel Core 2 and Intel Xeon processor families:

- **Last Branch Record (LBR) Stack**

- — MSR_LASTBRANCH_0_FROM_IP (address 40H) through MSR_LASTBRANCH_3_FROM_IP (address 43H) store source addresses
- — MSR_LASTBRANCH_0_TO_IP (address 60H) through MSR_LASTBRANCH_3_TO_IP (address 63H) store destination addresses

- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 2 bits of the TOS Pointer MSR (MSR_LASTBRANCH_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

Eight pairs of MSRs are supported in the LBR stack for Intel Atom processors:

- **Last Branch Record (LBR) Stack**
  - — MSR_LASTBRANCH_0_FROM_IP (address 40H) through MSR_LASTBRANCH_7_FROM_IP (address 47H) store source addresses
  - — MSR_LASTBRANCH_0_TO_IP (address 60H) through MSR_LASTBRANCH_7_TO_IP (address 67H) store destination addresses
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 3 bits of the TOS Pointer MSR (MSR_LASTBRANCH_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

For compatibility, the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs) duplicate functions of the LastExceptionToIP and LastExceptionFromIP MSRs found in P6 family processors.

...

19. Updates to Chapter 20, Volume 3B

Change bars show changes to Chapter 20 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

20.8 RESTRICTIONS ON VMX OPERATION

...

NOTES

The first processors to support VMX operation require that the following bits be 1 in VMX operation: CR0.PE, CR0.NE, CR0.PG, and CR4.VMXE. The restrictions on CR0.PE and CR0.PG imply that VMX operation is supported only in paged protected mode (including IA-32e mode). Therefore, guest software cannot be run in unpaged protected mode or in real-address mode. See Section 27.2, "Supporting Processor Operating Modes in Guest Environments," for a discussion of how a VMM might support guest software that expects to run in unpaged protected mode or in real-address mode.

Later processors support a VM-execution control called "unrestricted guest" (see Section 21.6.2). If this control is 1, CR0.PE and CR0.PG may be 0 in VMX non-root operation (even if the capability MSR IA32_VMX_CR0_FIXED0 reports otherwise).[1] Such processors allow guest software to run in unpaged protected mode or in real-address mode.

...

20. Updates to Chapter 22, Volume 3B

Change bars show changes to Chapter 22 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

22.1.3 Instructions That Cause VM Exits Conditionally

...

- **MWAIT.** The MWAIT instruction causes a VM exit if the "MWAIT exiting" VM-execution control is 1. If this control is 0, the behavior of the MWAIT instruction may be modified (see Section 22.4).

...

22.2.1.1 Linear Accesses That Cause APIC-Access VM Exits

Whether a linear access to the APIC-access page causes an APIC-access VM exit depends in part of the nature of the translation used by the linear address:

- If the linear access uses a translation with a 4-KByte page, it causes an APIC-access VM exit.
- If the linear access uses a translation with a large page (2-MByte, 4-MByte, or 1-GByte), the access may or may not cause an APIC-access VM exit. Section 22.5.1 describes the treatment of such accesses that do not cause an APIC-access VM exits.

  If CR0.PG = 1 and EPT is in use (the "enable EPT" VM-execution control is 1), a linear access uses a translation with a large page only if a large page is specified by both the guest paging structures and the EPT paging structures.[2]

...

1. "Unrestricted guest" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the "unrestricted guest" VM-execution control were 0. See Section 21.6.2.
2. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PG must be 1 in VMX operation, CR0.PG must be 1 unless the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1. "Enable EPT" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the "enable EPT" VM-execution control were 0. See Section 21.6.2.

22.2.2 Guest-Physical Accesses to the APIC-Access Page

An access to the APIC-access page is called a **guest-physical access** if (1) CR0.PG = 1;[1] (2) the "enable EPT" VM-execution control is 1;[2] (3) the access's physical address is the result of an EPT translation; and (4) either (a) the access was not generated by a linear address; or (b) the access's guest-physical address is not the translation of the access's linear address. Guest-physical accesses include the following when guest-physical addresses are being translated using EPT:

- Reads from the guest paging structures when translating a linear address (such an access uses a guest-physical address that is not the translation of that linear address).
- Loads of the page-directory-pointer-table entries by MOV to CR when the logical processor is using (or that causes the logical processor to use) PAE paging.[3]
- Updates to the accessed and dirty bits in the guest paging structures when using a linear address (such an access uses a guest-physical address that is not the translation of that linear address).

Section 22.2.2.1 specifies when guest-physical accesses to the APIC-access page might not cause APIC-access VM exits. In general, the treatment of APIC-access VM exits caused by guest-physical accesses is similar to that of EPT violations. Based upon this treatment, Section 22.2.2.2 specifies the priority of such VM exits with respect to other events.

22.2.2.1 Guest-Physical Accesses That Might Not Cause APIC-Access VM Exits

Whether a guest-physical access to the APIC-access page causes an APIC-access VM exit depends on the nature of the EPT translation used by the guest-physical address and on how software is managing information cached from the EPT paging structures. The following items detail cases in which a guest-physical access to the APIC-access page might not cause an APIC-access VM exit:

- If the access uses a guest-physical address whose translation to the APIC-access page uses an EPT PDPTE that maps a 1-GByte page (because bit 7 of the EPT PDPTE is 1).
- If the access uses a guest-physical address whose translation to the APIC-access page uses an EPT PDE that maps a 2-MByte page (because bit 7 of the EPT PDE is 1).

...

1. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PG must be 1 in VMX operation, CR0.PG must be 1 unless the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
2. "Enable EPT" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the "enable EPT" VM-execution control were 0. See Section 21.6.2.
3. A logical processor uses PAE paging if CR0.PG = 1, CR4.PAE = 1 and IA32_EFER.LMA = 0. See Section 4.4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

22.4 CHANGES TO INSTRUCTION BEHAVIOR IN VMX NON-ROOT OPERATION

...

- **MWAIT.** Behavior of the MWAIT instruction (which always causes an invalid-opcode exception—#UD—if CPL > 0) is determined by the setting of the "MWAIT exiting" VM-execution control:
  - — If the "MWAIT exiting" VM-execution control is 1, MWAIT causes a VM exit (see Section 22.1.3).
  - — If the "MWAIT exiting" VM-execution control is 0, MWAIT operates normally if any of the following is true: (1) the "interrupt-window exiting" VM-execution control is 0; (2) ECX[0] is 0; or (3) RFLAGS.IF = 1.
  - — If the "MWAIT exiting" VM-execution control is 0, the "interrupt-window exiting" VM-execution control is 1, ECX[0] = 1, and RFLAGS.IF = 0, MWAIT does not cause the processor to enter an implementation-dependent optimized state; instead, control passes to the instruction following the MWAIT instruction.

...

21. Updates to Chapter 23, Volume 3B

Change bars show changes to Chapter 23 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

23.3.1.1 Checks on Guest Control Registers, Debug Registers, and MSRs

The following checks are performed on fields in the guest-state area corresponding to control registers, debug registers, and MSRs:

- The CR0 field must not set any bit to a value not supported in VMX operation (see Section 20.8). The following are exceptions:
  - — Bit 0 (corresponding to CR0.PE) and bit 31 (PG) are not checked if the "unrestricted guest" VM-execution control is 1.[1]
  - — Bit 29 (corresponding to CR0.NW) and bit 30 (CD) are never checked because the values of these bits are not changed by VM entry; see Section 23.3.2.1.
- If bit 31 in the CR0 field (corresponding to PG) is 1, bit 0 in that field (PE) must also be 1.[2]
- The CR4 field must not set any bit to a value not supported in VMX operation (see Section 20.8).

1. "Unrestricted guest" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the "unrestricted guest" VM-execution control were 0. See Section 21.6.2.
2. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PE must be 1 in VMX operation, bit 0 in the CR0 field must be 1 unless the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

- If the "load debug controls" VM-entry control is 1, bits reserved in the IA32_DEBUGCTL MSR must be 0 in the field for that register. The first processors to support the virtual-machine extensions supported only the 1-setting of this control and thus performed this check unconditionally.
- The following checks are performed on processors that support Intel 64 architecture:
  - If the "IA-32e mode guest" VM-entry control is 1, bit 31 in the CR0 field (corresponding to CR0.PG) and bit 5 in the CR4 field (corresponding to CR4.PAE) must each be 1.[1]
  - If the "IA-32e mode guest" VM-entry control is 0, bit 17 in the CR4 field (corresponding to CR4.PCIDE) must each be 0.
  - The CR3 field must be such that bits 63:52 and bits in the range 51:32 beyond the processor's physical-address width are 0.[2,3]
  - If the "load debug controls" VM-entry control is 1, bits 63:32 in the DR7 field must be 0. The first processors to support the virtual-machine extensions supported only the 1-setting of this control and thus performed this check unconditionally (if they supported Intel 64 architecture).
  - The IA32_SYSENTER_ESP field and the IA32_SYSENTER_EIP field must each contain a canonical address.
- If the "load IA32_PERF_GLOBAL_CTRL" VM-entry control is 1, bits reserved in the IA32_PERF_GLOBAL_CTRL MSR must be 0 in the field for that register (see Figure 30-3).
- If the "load IA32_PAT" VM-entry control is 1, the value of the field for the IA32_PAT MSR must be one that could be written by WRMSR without fault at CPL 0. Specifically, each of the 8 bytes in the field must have one of the values 0 (UC), 1 (WC), 4 (WT), 5 (WP), 6 (WB), or 7 (UC-).
- If the "load IA32_EFER" VM-entry control is 1, the following checks are performed on the field for the IA32_EFER MSR :
  - Bits reserved in the IA32_EFER MSR must be 0.
  - Bit 10 (corresponding to IA32_EFER.LMA) must equal the value of the "IA-32e mode guest" VM-exit control. It must also be identical to bit 8 (LME) if bit 31 in the CR0 field (corresponding to CR0.PG) is 1.[4]

...

1. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PG must be 1 in VMX operation, bit 31 in the CR0 field must be 1 unless the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
2. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
3. Bit 63 of the CR3 field in the guest-state area must be 0. This is true even though, If CR4.PCIDE = 1, bit 63 of the source operand to MOV to CR3 is used to determine whether cached translation information is invalidated.
4. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PG must be 1 in VMX operation, bit 31 in the CR0 field must be 1 unless the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

23.3.1.2 Checks on Guest Segment Registers

...

- If the guest will not be virtual-8086, the different sub-fields are considered separately:
  - — Bits 3:0 (Type).
    - CS. The values allowed depend on the setting of the "unrestricted guest" VM-execution control:
      - — If the control is 0, the Type must be 9, 11, 13, or 15 (accessed code segment).
      - — If the control is 1, the Type must be either 3 (read/write accessed expand-up data segment) or one of 9, 11, 13, and 15 (accessed code segment).
    - SS. If SS is usable, the Type must be 3 or 7 (read/write, accessed data segment).
    - DS, ES, FS, GS. The following checks apply if the register is usable:
      - — Bit 0 of the Type must be 1 (accessed).
      - — If bit 3 of the Type is 1 (code segment), then bit 1 of the Type must be 1 (readable).
  - — Bit 4 (S). If the register is CS or if the register is usable, S must be 1.
  - — Bits 6:5 (DPL).
    - CS.
      - — If the Type is 3 (read/write accessed expand-up data segment), the DPL must be 0. The Type can be 3 only if the "unrestricted guest" VM-execution control is 1.
      - — If the Type is 9 or 11 (non-conforming code segment), the DPL must equal the DPL in the access-rights field for SS.
      - — If the Type is 13 or 15 (conforming code segment), the DPL cannot be greater than the DPL in the access-rights field for SS.
    - SS.
      - — If the "unrestricted guest" VM-execution control is 0, the DPL must equal the RPL from the selector field.
      - — The DPL must be 0 either if the Type in the access-rights field for CS is 3 (read/write accessed expand-up data segment) or if bit 0 in the CR0 field (corresponding to CR0.PE) is 0.[1]

...

1. The following apply if either the "unrestricted guest" VM-execution control or bit 31 of the primary processor-based VM-execution controls is 0: (1) bit 0 in the CR0 field must be 1 if the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PE must be 1 in VMX operation; and (2) the Type in the access-rights field for CS cannot be 3.

23.3.1.4 Checks on Guest RIP and RFLAGS

The following checks are performed on fields in the guest-state area corresponding to RIP and RFLAGS:

- RIP. The following checks are performed on processors that support Intel 64 architecture:
  - Bits 63:32 must be 0 if the "IA-32e mode guest" VM-entry control is 0 or if the L bit (bit 13) in the access-rights field for CS is 0.
  - If the processor supports N < 64 linear-address bits, bits 63:N must be identical if the "IA-32e mode guest" VM-entry control is 1 and the L bit in the access-rights field for CS is 1.[1] (No check applies if the processor supports 64 linear-address bits.)
- RFLAGS.
  - Reserved bits 63:22 (bits 31:22 on processors that do not support Intel 64 architecture), bit 15, bit 5 and bit 3 must be 0 in the field, and reserved bit 1 must be 1.
  - The VM flag (bit 17) must be 0 either if the "IA-32e mode guest" VM-entry control is 1 or if bit 0 in the CR0 field (corresponding to CR0.PE) is 0.[2]
  - The IF flag (RFLAGS[bit 9]) must be 1 if the valid bit (bit 31) in the VM-entry interruption-information field is 1 and the interruption type (bits 10:8) is external interrupt.

...

23.3.2.1 Loading Guest Control Registers, Debug Registers, and MSRs

The following items describe how guest control registers, debug registers, and MSRs are loaded on VM entry:

- CR0 is loaded from the CR0 field with the exception of the following bits, which are never modified on VM entry: ET (bit 4); reserved bits 15:6, 17, and 28:19; NW (bit 29) and CD (bit 30).[3] The values of these bits in the CR0 field are ignored.
- CR3 and CR4 are loaded from the CR3 field and the CR4 field, respectively.
- If the "load debug controls" VM-execution control is 1, DR7 is loaded from the DR7 field with the exception that bit 12 and bits 15:14 are always 0 and bit 10 is always 1. The values of these bits in the DR7 field are ignored.

  The first processors to support the virtual-machine extensions supported only the 1-setting of the "load debug controls" VM-execution control and thus always loaded DR7 from the DR7 field.

1. Software can determine the number N by executing CPUID with 80000008H in EAX. The number of linear-address bits supported is returned in bits 15:8 of EAX.
2. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PE must be 1 in VMX operation, bit 0 in the CR0 field must be 1 unless the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
3. Bits 15:6, bit 17, and bit 28:19 of CR0 and CR0.ET are unchanged by executions of MOV to CR0. Bits 15:6, bit 17, and bit 28:19 of CR0 are always 0 and CR0.ET is always 1.

- The following describes how some MSRs are loaded using fields in the guest-state area:
  - — If the "load debug controls" VM-execution control is 1, the IA32_DEBUGCTL MSR is loaded from the IA32_DEBUGCTL field. The first processors to support the virtual-machine extensions supported only the 1-setting of this control and thus always loaded the IA32_DEBUGCTL MSR from the IA32_DEBUGCTL field.
  - — The IA32_SYSENTER_CS MSR is loaded from the IA32_SYSENTER_CS field. Since this field has only 32 bits, bits 63:32 of the MSR are cleared to 0.
  - — The IA32_SYSENTER_ESP and IA32_SYSENTER_EIP MSRs are loaded from the IA32_SYSENTER_ESP field and the IA32_SYSENTER_EIP field, respectively. On processors that do not support Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are cleared to 0.
  - — The following are performed on processors that support Intel 64 architecture:
    - The MSRs FS.base and GS.base are loaded from the base-address fields for FS and GS, respectively (see Section 23.3.2.2).
    - If the "load IA32_EFER" VM-entry control is 0, bits in the IA32_EFER MSR are modified as follows:
      - — IA32_EFER.LMA is loaded with the setting of the "IA-32e mode guest" VM-entry control.
      - — If CR0 is being loaded so that CR0.PG = 1, IA32_EFER.LME is also loaded with the setting of the "IA-32e mode guest" VM-entry control.[1] Otherwise, IA32_EFER.LME is unmodified.

...

23.3.2.2 Loading Guest Segment Registers and Descriptor-Table Registers

For each of CS, SS, DS, ES, FS, GS, TR, and LDTR, fields are loaded from the guest-state area as follows:

- The unusable bit is loaded from the access-rights field. This bit can never be set for TR (see Section 23.3.1.2). If it is set for one of the other registers, the following apply:
  - — For each of CS, SS, DS, ES, FS, and GS, uses of the segment cause faults (general-protection exception or stack-fault exception) outside 64-bit mode, just as they would had the segment been loaded using a null selector. This bit does not cause accesses to fault in 64-bit mode.
  - — If this bit is set for LDTR, uses of LDTR cause general-protection exceptions in all modes, just as they would had LDTR been loaded using a null selector.

  If this bit is clear for any of CS, SS, DS, ES, FS, GS, TR, and LDTR, a null selector value does not cause a fault (general-protection exception or stack-fault exception).
- TR. The selector, base, limit, and access-rights fields are loaded.
- CS.

1. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PG must be 1 in VMX operation, VM entry must be loading CR0 so that CR0.PG = 1 unless the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

- — The following fields are always loaded: selector, base address, limit, and (from the access-rights field) the L, D, and G bits.
- — For the other fields, the unusable bit of the access-rights field is consulted:
    - If the unusable bit is 0, all of the access-rights field is loaded.
    - If the unusable bit is 1, the remainder of CS access rights are undefined after VM entry.

- SS, DS, ES, FS, GS, and LDTR.
    - — The selector fields are loaded.
    - — For the other fields, the unusable bit of the corresponding access-rights field is consulted:
        - If the unusable bit is 0, the base-address, limit, and access-rights fields are loaded.
        - If the unusable bit is 1, the base address, the segment limit, and the remainder of the access rights are undefined after VM entry with the following exceptions:
            - — Bits 3:0 of the base address for SS are cleared to 0.
            - — SS.DPL is always loaded from the SS access-rights field. This will be the current privilege level (CPL) after the VM entry completes.
            - — SS.B is always set to 1.
            - — The base addresses for FS and GS are loaded from the corresponding fields in the VMCS. On processors that support Intel 64 architecture, the values loaded for base addresses for FS and GS are also manifest in the FS.base and GS.base MSRs.
            - — On processors that support Intel 64 architecture, the base address for LDTR is set to an undefined but canonical value.
            - — On processors that support Intel 64 architecture, bits 63:32 of the base addresses for SS, DS, and ES are cleared to 0.

GDTR and IDTR are loaded using the base and limit fields.

...

23.5.1.3 Event Injection for VM Entries to Real-Address Mode

If VM entry is loading CR0.PE with 0, any injected vectored event is delivered as would normally be done in real-address mode.[1] Specifically, VM entry uses the vector provided in the VM-entry interruption-information field to select a 4-byte entry from an interrupt-vector table at the linear address in IDTR.base. Further details are provided in Section 15.1.4 in Volume 3A of the *IA-32 Intel® Architecture Software Developer's Manual*.

Because bit 11 (deliver error code) in the VM-entry interruption-information field must be 0 if CR0.PE will be 0 after VM entry (see Section 23.2.1.3), vectored events injected with CR0.PE = 0 do not push an error code on the stack. This is consistent with event delivery in real-address mode.

If event delivery encounters a fault (due to a violation of IDTR.limit or of SS.limit), the fault is treated as if it had occurred during event delivery in VMX non-root operation. Such a fault may lead to a VM exit as discussed in Section 23.5.1.2.

1. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PE must be 1 in VMX operation, VM entry must be loading CR0.PE with 1 unless the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

...

22. Updates to Chapter 24, Volume 3B

Change bars show changes to Chapter 24 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

24.2.2 Information for VM Exits Due to Vectored Events

...

The following items detail the use of these fields:

- **VM-exit interruption information** (format given in Table 21-14). The following items detail how this field is established for VM exits due to these events:
  - — For an exception, bits 7:0 receive the exception vector (at most 31). For an NMI, bits 7:0 are set to 2. For an external interrupt, bits 7:0 receive the interrupt number.
  - — Bits 10:8 are set to 0 (external interrupt), 2 (non-maskable interrupt), 3 (hardware exception), or 6 (software exception). Hardware exceptions comprise all exceptions except breakpoint exceptions (#BP; generated by INT3) and overflow exceptions (#OF; generated by INTO); these are software exceptions. BOUND-range exceeded exceptions (#BR; generated by BOUND) and invalid opcode exceptions (#UD) generated by UD2 are hardware exceptions.
  - — Bit 11 is set to 1 if the VM exit is caused by a hardware exception that would have delivered an error code on the stack. This bit is always 0 if the VM exit occurred while the logical processor was in real-address mode (CR0.PE=0).[1] If bit 11 is set to 1, the error code is placed in the VM-exit interruption error code (see below).

...

24.2.3 Information for VM Exits During Event Delivery

...

The following items detail the use of these fields:

- IDT-vectoring information (format given in Table 21-15). The following items detail how this field is established for VM exits that occur during event delivery:
  - — If the VM exit occurred during delivery of an exception, bits 7:0 receive the exception vector (at most 31). If the VM exit occurred during delivery of an NMI, bits 7:0 are set to 2. If the VM exit occurred during delivery of an external interrupt, bits 7:0 receive the interrupt number.
  - — Bits 10:8 are set to indicate the type of event that was being delivered when the VM exit occurred: 0 (external interrupt), 2 (non-maskable interrupt), 3

1. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PE must be 1 in VMX operation, a logical processor cannot be in real-address mode unless the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

(hardware exception), 4 (software interrupt), 5 (privileged software interrupt), or 6 (software exception).

Hardware exceptions comprise all exceptions except breakpoint exceptions (#BP; generated by INT3) and overflow exceptions (#OF; generated by INTO); these are software exceptions. BOUND-range exceeded exceptions (#BR; generated by BOUND) and invalid opcode exceptions (#UD) generated by UD2 are hardware exceptions.

Bits 10:8 may indicate privileged software interrupt if such an event was injected as part of VM entry.

- Bit 11 is set to 1 if the VM exit occurred during delivery of a hardware exception that would have delivered an error code on the stack. This bit is always 0 if the VM exit occurred while the logical processor was in real-address mode (CR0.PE=0).[1] If bit 11 is set to 1, the error code is placed in the IDT-vectoring error code (see below).
- Bit 12 is undefined.
- Bits 30:13 are always set to 0.
- Bit 31 is always set to 1.

...

24.5.1 Loading Host Control Registers, Debug Registers, MSRs

VM exits load new values for controls registers, debug registers, and some MSRs:

- CR0, CR3, and CR4 are loaded from the CR0 field, the CR3 field, and the CR4 field, respectively, with the following exceptions:
  - The following bits are not modified:
    - For CR0, ET, CD, NW; bits 63:32 (on processors that support Intel 64 architecture), 28:19, 17, and 15:6; and any bits that are fixed in VMX operation (see Section 20.8).[2]
    - For CR3, bits 63:52 and bits in the range 51:32 beyond the processor's physical-address width (they are cleared to 0).[3] (This item applies only to processors that support Intel 64 architecture.)
    - For CR4, any bits that are fixed in VMX operation (see Section 20.8).
  - CR4.PAE is set to 1 if the "host address-space size" VM-exit control is 1.
  - CR4.PCIDE is set to 0 if the "host address-space size" VM-exit control is 0.
- DR7 is set to 400H.
- The following MSRs are established as follows:
  - The IA32_DEBUGCTL MSR is cleared to 00000000_00000000H.

1. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PE must be 1 in VMX operation, a logical processor cannot be in real-address mode unless the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
2. Bits 28:19, 17, and 15:6 of CR0 and CR0.ET are unchanged by executions of MOV to CR0. CR0.ET is always 1 and the other bits are always 0.
3. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

— The IA32_SYSENTER_CS MSR is loaded from the IA32_SYSENTER_CS field. Since that field has only 32 bits, bits 63:32 of the MSR are cleared to 0.

— IA32_SYSENTER_ESP MSR and IA32_SYSENTER_EIP MSR are loaded from the IA32_SYSENTER_ESP field and the IA32_SYSENTER_EIP field, respectively.

If the processor does not support the Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are cleared to 0.

If the processor does support the Intel 64 architecture and the processor supports N < 64 linear-address bits, each of bits 63:N is set to the value of bit N–1.[1]

— The following steps are performed on processors that support Intel 64 architecture:

- The MSRs FS.base and GS.base are loaded from the base-address fields for FS and GS, respectively (see Section 24.5.2).
- The LMA and LME bits in the IA32_EFER MSR are each loaded with the setting of the "host address-space size" VM-exit control.

— If the "load IA32_PERF_GLOBAL_CTRL" VM-exit control is 1, the IA32_PERF_GLOBAL_CTRL MSR is loaded from the IA32_PERF_GLOBAL_CTRL field. Bits that are reserved in that MSR are maintained with their reserved values.

— If the "load IA32_PAT" VM-exit control is 1, the IA32_PAT MSR is loaded from the IA32_PAT field. Bits that are reserved in that MSR are maintained with their reserved values.

— If the "load IA32_EFER" VM-exit control is 1, the IA32_EFER MSR is loaded from the IA32_EFER field. Bits that are reserved in that MSR are maintained with their reserved values.

With the exception of FS.base and GS.base, any of these MSRs is subsequently overwritten if it appears in the VM-exit MSR-load area. See Section 24.6.

24.5.2 Loading Host Segment and Descriptor-Table Registers

Each of the registers CS, SS, DS, ES, FS, GS, and TR is loaded as follows (see below for the treatment of LDTR):

- The selector is loaded from the selector field. The segment is unusable if its selector is loaded with zero. The checks specified Section 23.3.1.2 limit the selector values that may be loaded. In particular, CS and TR are never loaded with zero and are thus never unusable. SS can be loaded with zero only on processors that support Intel 64 architecture and only if the VM exit is to 64-bit mode (64-bit mode allows use of segments marked unusable).
- The base address is set as follows:

  — CS. Cleared to zero.

  — SS, DS, and ES. Undefined if the segment is unusable; otherwise, cleared to zero.

1. Software can determine the number N by executing CPUID with 80000008H in EAX. The number of linear-address bits supported is returned in bits 15:8 of EAX.

   - FS and GS. Undefined (but, on processors that support Intel 64 architecture, canonical) if the segment is unusable and the VM exit is not to 64-bit mode; otherwise, loaded from the base-address field.

     If the processor supports the Intel 64 architecture and the processor supports N < 64 linear-address bits, each of bits 63:N is set to the value of bit N–1.[1] The values loaded for base addresses for FS and GS are also manifest in the FS.base and GS.base MSRs.
   - TR. Loaded from the host-state area. If the processor supports the Intel 64 architecture and the processor supports N < 64 linear-address bits, each of bits 63:N is set to the value of bit N–1.
- The segment limit is set as follows:
   - CS. Set to FFFFFFFFH (corresponding to a descriptor limit of FFFFFH and a G-bit setting of 1).
   - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to FFFFFFFFH.
   - TR. Set to 00000067H.
- The type field and S bit are set as follows:
   - CS. Type set to 11 and S set to 1 (execute/read, accessed, non-conforming code segment).
   - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, type set to 3 and S set to 1 (read/write, accessed, expand-up data segment).
   - TR. Type set to 11 and S set to 0 (busy 32-bit task-state segment).
- The DPL is set as follows:
   - CS, SS, and TR. Set to 0. The current privilege level (CPL) will be 0 after the VM exit completes.
   - DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 0.
- The P bit is set as follows:
   - CS, TR. Set to 1.
   - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.
- On processors that support Intel 64 architecture, CS.L is loaded with the setting of the "host address-space size" VM-exit control. Because the value of this control is also loaded into IA32_EFER.LMA (see Section 24.5.1), no VM exit is ever to compatibility mode (which requires IA32_EFER.LMA = 1 and CS.L = 0).
- D/B.
   - CS. Loaded with the inverse of the setting of the "host address-space size" VM-exit control. For example, if that control is 0, indicating a 32-bit guest, CS.D/B is set to 1.
   - SS. Set to 1.
   - DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.
   - TR. Set to 0.

1. Software can determine the number N by executing CPUID with 80000008H in EAX. The number of linear-address bits supported is returned in bits 15:8 of EAX.

- G.
  - — CS. Set to 1.
  - — SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.
  - — TR. Set to 0.

The host-state area does not contain a selector field for LDTR. LDTR is established as follows on all VM exits: the selector is cleared to 0000H, the segment is marked unusable and is otherwise undefined (although the base address is always canonical).

The base addresses for GDTR and IDTR are loaded from the GDTR base-address field and the IDTR base-address field, respectively. If the processor supports the Intel 64 architecture and the processor supports N < 64 linear-address bits, each of bits 63:N of each base address is set to the value of bit N–1 of that base address. The GDTR and IDTR limits are each set to FFFFH.

...

24.6 LOADING MSRS

VM exits may load MSRs from the VM-exit MSR-load area (see Section 21.7.2). Specifically each entry in that area (up to the number specified in the VM-exit MSR-load count) is processed in order by loading the MSR indexed by bits 31:0 with the contents of bits 127:64 as they would be written by WRMSR.

Processing of an entry fails in any of the following cases:

- The value of bits 31:0 is either C0000100H (the IA32_FS_BASE MSR) or C0000101H (the IA32_GS_BASE MSR).
- The value of bits 31:8 is 000008H, meaning that the indexed MSR is one that allows access to an APIC register when the local APIC is in x2APIC mode.
- The value of bits 31:0 indicates an MSR that can be written only in system-management mode (SMM) and the VM exit will not end in SMM. (IA32_SMM_MONITOR_CTL is an MSR that can be written only in SMM.)
- The value of bits 31:0 indicates an MSR that cannot be loaded on VM exits for model-specific reasons. A processor may prevent loading of certain MSRs even if they can normally be written by WRMSR. Such model-specific behavior is documented in Appendix B.
- Bits 63:32 are not all 0.
- An attempt to write bits 127:64 to the MSR indexed by bits 31:0 of the entry would cause a general-protection exception if executed via WRMSR with CPL = 0.[1]

If processing fails for any entry, a VMX abort occurs. See Section 24.7.

If any MSR is being loaded in such a way that would architecturally require a TLB flush, the TLBs are updated so that, after VM exit, the logical processor does not use any translations that were cached before the transition.

1. Note the following about processors that support Intel 64 architecture. If CR0.PG = 1, WRMSR to the IA32_EFER MSR causes a general-protection exception if it would modify the LME bit. Since CR0.PG is always 1 in VMX operation, the IA32_EFER MSR should not be included in the VM-exit MSR-load area for the purpose of modifying the LME bit.

23. Updates to Chapter 25, Volume 3B

Change bars show changes to Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

25.2.1 EPT Overview

EPT is used when the "enable EPT" VM-execution control is 1.[1] It translates the guest-physical addresses used in VMX non-root operation and those used by VM entry for event injection.

The translation from guest-physical addresses to physical addresses is determined by a set of **EPT paging structures**. The EPT paging structures are similar to those used to translate linear addresses while the processor is in IA-32e mode. Section 25.2.2 gives the details of the EPT paging structures.

If CR0.PG = 1, linear addresses are translated through paging structures referenced through control register CR3 . While the "enable EPT" VM-execution control is 1, these are called **guest paging structures**. There are no guest paging structures if CR0.PG = 0.[2]

...

25.2.4.2 Memory Type Used for Translated Guest-Physical Addresses

The **effective memory type** of a memory access using a guest-physical address (an access that is translated using EPT) is the memory type that is used to access memory. The effective memory type is based on the value of bit 30 (cache disable—CD) in control register CR0; the **last** EPT paging-structure entry used to translate the guest-physical address (either an EPT PDE with bit 7 set to 1 or an EPT PTE); and the PAT memory type (see below):

- The **PAT memory type** depends on the value of CR0.PG:
  - If CR0.PG = 0, the PAT memory type is WB (writeback).[3]
  - If CR0.PG = 1, the PAT memory type is the memory type selected from the IA32_PAT MSR as specified in Section 11.12.3, "Selecting a Memory Type from the PAT".[4]

...

1. "Enable EPT" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, the logical processor operates as if the "enable EPT" VM-execution control were 0. See Section 21.6.2.
2. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PG must be 1 in VMX operation, CR0.PG can be 0 in VMX non-root operation only if the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
3. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PG must be 1 in VMX operation, CR0.PG can be 0 in VMX non-root operation only if the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

25.3.2 Creating and Using Cached Translation Information

...

- The following items describe the creation of mappings while EPT is in use:
  - Guest-physical mappings may be created. They are derived from the EPT paging structures referenced (directly or indirectly) by bits 51:12 of the current EPTP. These 40 bits contain the address of the EPT-PML4-table. (the notation **EP4TA** refers to those 40 bits). Newly created guest-physical mappings are associated with the current EP4TA.
  - Combined mappings may be created. They are derived from the EPT paging structures referenced (directly or indirectly) by the current EP4TA. If CR0.PG = 1, they are also derived from the paging structures referenced (directly or indirectly) by the current value of CR3. They are associated with the current VPID, the current PCID, and the current EP4TA.[1] No combined paging-structure-cache entries are created if CR0.PG = 0.[2]
  - No guest-physical mappings or combined mappings are created with information derived from EPT paging-structure entries that are not present (bits 2:0 are all 0) or that are misconfigured (see Section 25.2.3.1).
  - No combined mappings are created with information derived from guest paging-structure entries that are not present or that set reserved bits.
  - No linear mappings are created while EPT is in use.

...

24. Updates to Chapter 26, Volume 3B

Change bars show changes to Chapter 26 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

4. Table 11-11 in Section 11.12.3, "Selecting a Memory Type from the PAT" illustrates how the PAT memory type is selected based on the values of the PAT, PCD, and PWT bits in a page-table entry (or page-directory entry with PS = 1). For accesses to a guest paging-structure entry X, the PAT memory type is selected from the table by using a value of 0 for the PAT bit with the values of PCD and PWT from the paging-structure entry Y that references X (or from CR3 if X is in the root paging structure). With PAE paging, the PAT memory type for accesses to the PDPTEs is WB.

1. At any given time, a logical processor may be caching combined mappings for a VPID and a PCID that are associated with different EP4TAs. Similarly, it may be caching combined mappings for an EP4TA that are associated with different VPIDs and PCIDs.

2. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PG must be 1 in VMX operation, CR0.PG can be 0 in VMX non-root operation only if the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.

Table 26-3 SMRAM State Save Map for Intel 64 Architecture

Offset (Added to SMBASE + 8000H)	Register	Writable?
...		
7ED8H	Value of EPTP VM-execution control field	No
7ED7H - 7EA0H	Reserved	No
7E9CH	LDT Base (lower 32 bits)	No
7E98H	Reserved	No
7E94H	IDT Base (lower 32 bits)	No
7E90H	Reserved	No
7E8CH	GDT Base (lower 32 bits)	No
...		

NOTE:

1. The two most significant bytes are reserved.

...

26.14.2 Default Treatment of RSM

...

Ordinary execution of RSM restores processor state from SMRAM. Under the default treatment, processors that support VMX operation perform RSM as follows:

```
IF VMXE = 1 in CR4 image in SMRAM
   THEN fail and enter shutdown state;
   ELSE
       restore state normally from SMRAM;
       invalidate linear mappings and combined mappings associated with all VPIDs and all PCIDs;
combined mappings are invalidated for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP; see
Section 25.3);
       IF the logical processor supports SMX operation andthe Intel® TXT private space was unlocked
at the time of the last SMI (as saved)
           THEN unlock the TXT private space;
       FI;
       CR4.VMXE ← value stored internally;
       IF internal storage indicates that the logical processor
       had been in VMX operation (root or non-root)
           THEN
               enter VMX operation (root or non-root);
               restore VMX-critical state as defined in Section 26.14.1;
               set to their fixed values any bits in CR0 and CR4 whose values must be fixed in VMX
operation (see Section 20.8);[1]
               IF RFLAGS.VM = 0 AND (in VMX root operation OR the “unrestricted guest” VM-
execution control is 0)[2]
```

```
                THEN
                    CS.RPL ← SS.DPL;
                    SS.RPL ← SS.DPL;
            FI;
            restore current VMCS pointer;
    FI;
    leave SMM;
    IF logical processor will be in VMX operation or in SMX operation after RSM
        THEN block A20M and leave A20M mode;
    FI;
FI;
```

RSM unblocks SMIs. It restores the state of blocking by NMI (see Table 21-3 in Section 21.4.2) as follows:

- If the RSM is not to VMX non-root operation or if the "virtual NMIs" VM-execution control will be 0, the state of NMI blocking is restored normally.
- If the RSM is to VMX non-root operation and the "virtual NMIs" VM-execution control will be 1, NMIs are not blocked after RSM. The state of virtual-NMI blocking is restored as part of VMX-critical state.

INIT signals are blocked after RSM if and only if the logical processor will be in VMX root operation.

If RSM returns a logical processor to VMX non-root operation, it re-establishes the controls associated with the current VMCS. If the "interrupt-window exiting" VM-execution control is 1, a VM exit occurs immediately after RSM if the enabling conditions apply. The same is true for the "NMI-window exiting" VM-execution control. Such VM exits occur with their normal priority. See Section 22.3.

...

26.15.7 Deactivating the Dual-Monitor Treatment

An SMM monitor may deactivate the dual monitor treatment and return the processor to default treatment of SMIs and SMM (see Section 26.14). It does this by executing a VM entry with the "deactivate dual-monitor treatment" VM-entry control set to 1.

As noted in Section 23.2.1.3 and Section 26.15.4.1, an attempt to deactivate the dual-monitor treatment fails in the following situations: (1) the processor is not in SMM; (2) the "entry to SMM" VM-entry control is 1; or (3) the executive-VMCS pointer does not contain the VMXON pointer (the VM entry is to VMX non-root operation).

As noted in Section 26.15.4.9, VM entries that deactivate the dual-monitor treatment ignore the SMI bit in the interruptibility-state field of the guest-state area. Instead, the blocking of SMIs following such a VM entry depends on whether the logical processor is in SMX operation:[1]

1. If the RSM is to VMX non-root operation and both the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls will be 1, CR0.PE and CR0.PG retain the values that were loaded from SMRAM regardless of what is reported in the capability MSR IA32_VMX_CR0_FIXED0.
2. "Unrestricted guest" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the "unrestricted guest" VM-execution control were 0. See Section 21.6.2.

- If the logical processor is in SMX operation, SMIs are blocked after VM entry.
- If the logical processor is outside SMX operation, SMIs are unblocked after VM entry.

...

25. Updates to Chapter 29, Volume 3B

Change bars show changes to Chapter 29 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

29.2 INTERRUPT HANDLING IN VMX OPERATION

The following bullets summarize VMX support for handling interrupts:

- **Control of processor exceptions**. The VMM can get control on specific guest exceptions through the exception-bitmap in the guest controlling VMCS. The exception bitmap is a 32-bit field that allows the VMM to specify processor behavior on specific exceptions (including traps, faults, and aborts). Setting a specific bit in the exception bitmap implies VM exits will be generated when the corresponding exception occurs. Any exceptions that are programmed not to cause VM exits are delivered directly to the guest through the guest IDT. The exception bitmap also controls execution of relevant instructions such as BOUND, INTO and INT3. VM exits on page-faults are treated in such a way the page-fault error code is qualified through the page-fault-error-code mask and match fields in the VMCS.
- **Control over triple faults**. If a fault occurs while attempting to call a double-fault handler in the guest and that fault is not configured to cause a VM exit in the exception bitmap, the resulting triple fault causes a VM exit.
- **Control of external interrupts**. VMX allows both host and guest control of external interrupts through the "external-interrupt exiting" VM execution control. If the control is 0, external-interrupts do not cause VM exits and the interrupt delivery is masked by the guest programmed RFLAGS.IF value.[1] If the control is 1, external-interrupts causes VM exits and are not masked by RFLAGS.IF. The VMM can identify VM exits due to external interrupts by checking the exit reason for an "external interrupt" (value = 1).

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENTER]. A logical processor is outside SMX operation if GETSEC[SENTER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENTER]. See Chapter 6, "Safer Mode Extensions Reference," in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.
1. This chapter uses the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For processors that do not support Intel 64 architecture, this notation refers to the 32-bit forms of those registers (EAX, EIP, ESP, EFLAGS, etc.).

- **Control of other events**. There is a pin-based VM-execution control that controls system behavior (exit or no-exit) for NMI events. Most VMM usages will need handling of NMI external events in the VMM and hence will specify host control of these events.

  Some processors also support a pin-based VM-execution control called "virtual NMIs." When this control is set, NMIs cause VM exits, but the processor tracks guest readiness for virtual NMIs. This control interacts with the "NMI-window exiting" VM-execution control (see below).

  INIT and SIPI events always cause VM exits.
- **Acknowledge interrupt on exit**. The "acknowledge interrupt on exit" VM-exit control in the controlling VMCS controls processor behavior for external interrupt acknowledgement. If the control is 1, the processor acknowledges the interrupt controller to acquire the interrupt vector upon VM exit, and stores the vector in the VM-exit interruption-information field. If the control is 0, the external interrupt is not acknowledged during VM exit. Since RFLAGS.IF is automatically cleared on VM exits due to external interrupts, VMM re-enabling of interrupts (setting RFLAGS.IF = 1) initiates the external interrupt acknowledgement and vectoring of the external interrupt through the monitor/host IDT.
- **Event-masking Support**. VMX captures the masking conditions of specific events while in VMX non-root operation through the interruptibility-state field in the guest-state area of the VMCS.

  This feature allows proper virtualization of various interrupt blocking states, such as: (a) blocking of external interrupts for the instruction following STI; (b) blocking of interrupts for the instruction following a MOV-SS or POP-SS instruction; (c) SMI blocking of subsequent SMIs until the next execution of RSM; and (d) NMI/SMI blocking of NMIs until the next execution of IRET or RSM.

  INIT and SIPI events are treated specially. INIT assertions are always blocked in VMX root operation and while in SMM, and unblocked otherwise. SIPI events are always blocked in VMX root operation.

  The interruptibility state is loaded from the VMCS guest-state area on every VM entry and saved into the VMCS on every VM exit.
- **Event injection**. VMX operation allows injecting interruptions to a guest virtual machine through the use of VM-entry interrupt-information field in VMCS. Injectable interruptions include external interrupts, NMI, processor exceptions, software generated interrupts, and software traps. If the interrupt-information field indicates a valid interrupt, exception or trap event upon the next VM entry; the processor will use the information in the field to vector a virtual interruption through the guest IDT after all guest state and MSRs are loaded. Delivery through the guest IDT emulates vectoring in non-VMX operation by doing the normal privilege checks and pushing appropriate entries to the guest stack (entries may include RFLAGS, EIP and exception error code). A VMM with host control of NMI and external interrupts can use the event-injection facility to forward virtual interruptions to various guest virtual machines.
- **Interrupt-window exiting**. When set to 1, the "interrupt-window exiting" VM-execution control (Section 21.6.2) causes VM exits when guest RFLAGS.IF is 1 and no other conditions block external interrupts. A VM exit occurs at the beginning of any instruction at which RFLAGS.IF = 1 and on which the interruptibility state of the guest would allow delivery of an interrupt. For example: when the guest executes an STI instruction, RFLAGS = 1, and if at the completion of next instruction the interruptibility state masking due to STI is removed; a VM exit occurs if the "interrupt-window exiting" VM-execution control is 1. This feature allows a VMM to queue a

virtual interrupt to the guest when the guest is not in an interruptible state. The VMM can set the "interrupt-window exiting" VM-execution control for the guest and depend on a VM exit to know when the guest becomes interruptible (and, therefore, when it can inject a virtual interrupt). The VMM can detect such VM exits by checking for the basic exit reason "interrupt-window" (value = 7). If this feature is not used, the VMM will need to poll and check the interruptibility state of the guest to deliver virtual interrupts.

- **NMI-window exiting**. If the "virtual NMIs" VM-execution is set, the processor tracks virtual-NMI blocking. The "NMI-window exiting" VM-execution control (Section 21.6.2) causes VM exits when there is no virtual-NMI blocking. For example, after execution of the IRET instruction, a VM exit occurs if the "NMI-window exiting" VM-execution control is 1. This feature allows a VMM to queue a virtual NMI to a guest when the guest is not ready to receive NMIs. The VMM can set the "NMI-window exiting" VM-execution control for the guest and depend on a VM exit to know when the guest becomes ready for NMIs (and, therefore, when it can inject a virtual NMI). The VMM can detect such VM exits by checking for the basic exit reason "NMI window" (value = 8). If this feature is not used, the VMM will need to poll and check the interruptibility state of the guest to deliver virtual NMIs.
- **VM-exit information**. The VM-exit information fields provide details on VM exits due to exceptions and interrupts. This information is provided through the exit-qualification, VM-exit-interruption-information, instruction-length and interruption-error-code fields. Also, for VM exits that occur in the course of vectoring through the guest IDT, information about the event that was being vectored through the guest IDT is provided in the IDT-vectoring-information and IDT-vectoring-error-code fields. These information fields allow the VMM to identify the exception cause and to handle it properly.

...

29.3.1 Virtualization of Interrupt Vector Space

The Intel 64 and IA-32 architectures use 8-bit vectors of which 244 (20H – FFH) are available for external interrupts. Vectors are used to select the appropriate entry in the interrupt descriptor table (IDT). VMX operation allows each guest to control its own IDT. Host vectors refer to vectors delivered by the platform to the processor during the interrupt acknowledgement cycle. Guest vectors refer to vectors programmed by a guest to select an entry in its guest IDT. Depending on the I/O resource management models supported by the VMM design, the guest vector space may or may not overlap with the underlying host vector space.

...

29.3.2.3 Local APIC Virtualization

The local APIC is responsible for the local interrupt sources, interrupt acceptance, dispensing interrupts to the logical processor, and generating inter-processor interrupts. Software interacts with the local APIC by reading and writing its memory-mapped registers residing within a 4-KByte uncached memory region with base address stored in the IA32_APIC_BASE MSR. Since the local APIC registers are memory-mapped, the VMM can utilize memory virtualization techniques (such as page-table virtualization) to trap guest accesses to the page frame hosting the virtual local APIC registers.

Local APIC virtualization in the VMM needs to emulate the various local APIC operations and registers, such as: APIC identification/format registers, the local vector table (LVT), the interrupt command register (ICR), interrupt capture registers (TMR, IRR and ISR), task and processor priority registers (TPR, PPR), the EOI register and the APIC-timer register. Since local APICs are designed to operate with non-specific EOI, local APIC emulation also needs to emulate broadcast of EOI to the guest's virtual I/O APICs for level triggered virtual interrupts.

A local APIC allows interrupt masking at two levels: (1) mask bit in the local vector table entry for local interrupts and (2) raising processor priority through the TPR registers for masking lower priority external interrupts. The VMM needs to comprehend these virtual local APIC mask settings as programmed by the guest in addition to the guest virtual processor interruptibility state (when injecting APIC routed external virtual interrupts to a guest VM).

VMX provides several features which help the VMM to virtualize the local APIC. These features allow many of guest TPR accesses (using CR8 only) to occur without VM exits to the VMM:

- The VMCS contains a "virtual-APIC address" field. This 64-bit field is the physical address of the 4-KByte virtual APIC page (4-KByte aligned). The virtual-APIC page contains a TPR shadow, which is accessed by the MOV CR8 instruction. The TPR shadow comprises bits 7:4 in byte 80H of the virtual-APIC page.
- The TPR threshold: bits 3:0 of this 32-bit field determine the threshold below which the TPR shadow cannot fall. A VM exit will occur after an execution of MOV CR8 that reduces the TPR shadow below this value.
- The processor-based VM-execution controls field contains a "use TPR shadow" bit and a "CR8-store exiting" bit. If the "use TPR shadow" VM-execution control is 1 and the "CR8-store exiting" VM-execution control is 0, then a MOV from CR8 reads from the TPR shadow. If the "CR8-store exiting" VM-execution control is 1, then MOV from CR8 causes a VM exit; the "use TPR shadow" VM-execution control is ignored in this case.
- The processor-based VM-execution controls field contains a "CR8-load exiting" bit. If the "use TPR shadow" VM-execution control is set and the "CR8-load exiting" VM-execution control is clear, then MOV to CR8 writes to the "TPR shadow". A VM exit will occur after this write if the value written is below the TPR threshold. If the "CR8-load exiting" VM-execution control is set, then MOV to CR8 causes a VM exit; the "use TPR shadow" VM-execution control is ignored in this case.

...

29.3.3.2 Processor Treatment of External Interrupt

Interrupts are automatically masked by hardware in the processor on VM exit by clearing RFLAGS.IF. The exit-reason field in VMCS is set to 1 to indicate an external interrupt as the exit reason.

If the VMM is utilizing the acknowledge-on-exit feature (by setting the "acknowledge interrupt on exit" VM-exit control), the processor acknowledges the interrupt, retrieves the host vector, and saves the interrupt in the VM-exit-interruption-information field (in the VM-exit information region of the VMCS) before transitioning control to the VMM.

29.3.3.3 Processing of External Interrupts by VMM

Upon VM exit, the VMM can determine the exit cause of an external interrupt by checking the exit-reason field (value = 1) in VMCS. If the acknowledge-interrupt-on-exit control (see Section 21.7.1) is enabled, the VMM can use the saved host vector (in the exit-interruption-information field) to switch to the appropriate interrupt handler. If the "acknowledge interrupt on exit" VM-exit control is 0, the VMM may re-enable interrupts (by setting RFLAGS.IF) to allow vectoring of external interrupts through the monitor/ host IDT.

...

29.4.3 MCA Error Handling Guidelines for VMM

Section 29.4.2 covers general requirements for VMMs to handle machine-check exceptions, when normal operation of the guest machine and/or the VMM is no longer possible. enhancements of machine check architecture in newer processors may support software recovery of uncorrected MC errors (UCR) signaled through either machine-check exceptions or corrected machine-check interrupt (CMCI). Section 15.5 and Section 15.6 describes details of these more recent enhancements of machine check architecture.

In general, Virtual Machine Monitor (VMM) error handling should follow the recommendations for OS error handling described in Section 15.3, Section 15.6, Section 15.9, and Section 15.10. This section describes additional guidelines for hosted and native hypervisor-based VMM implementations to support corrected MC errors and recoverable uncorrected MC errors.

Because a hosted VMM provides virtualization services in the context of an existing standard host OS, the host OS controls platform hardware through the host OS services such as the standard OS device drivers. In hosted VMMs. MCA errors will be handled by the host OS error handling software.

In native VMMs, the hypervisor runs on the hardware directly, and may provide only a limited set of platform services for guest VMs. Most platform services may instead be provided by a "control OS". In hypervisor-based VMMs, MCA errors will either be delivered directly to the VMM MCA handler (when the error is signaled while in the VMM context) or cause by a VM exit from a guest VM or be delivered to the MCA intercept handler. There are two general approaches the hypervisor can use to handle the MCA error: either within the hypervisor itself or by forwarding the error to the control OS.

...

26. Updates to Chapter 30, Volume 3B

Change bars show changes to Chapter 30 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

30.5 PERFORMANCE MONITORING (PROCESSORS BASED ON INTEL® ATOM™ MICROARCHITECTURE)

Intel Atom processor family supports architectural performance monitoring capability with version ID 3 (see Section 30.2.2.2) and a host of non-architectural monitoring capabilities. The initial implementation of Intel Atom processor family provides two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2).

Non-architectural performance monitoring in Intel Atom processor family uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events is listed in Table A-9.

Architectural and non-architectural performance monitoring events in Intel Atom processor family support thread qualification using bit 21 of IA32_PERFEVTSELx MSR.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 30-6 and described in Section 30.2.1.1 and Section 30.2.2.2.

Valid event mask (Umask) bits are listed in Appendix A. The UMASK field may contain sub-fields that provide the same qualifying actions like those listed in Table 30-2, Table 30-3, Table 30-4, and Table 30-5. One or more of these sub-fields may apply to specific events on an event-by-event basis. Details are listed in Table A-9 in Appendix A, "Performance-Monitoring Events." Precise Event Based Monitoring is supported using IA32_PMC0 (see also Section 16.4.9, "BTS and DS Save Area").

...

Table 30-15 MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 Bit Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
WB	3	(R/W). Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
OTHER	7	(R/W). Counts one of the following transaction types, including L3 invalidate, I/O, full or partial writes, WC or non-temporal stores, CLFLUSH, Fences, lock, unlock, split lock.
UNCORE_HIT	8	(R/W). L3 Hit: local or remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
OTHER_CORE_HIT_SNP	9	(R/W). L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoop where no modified copies were found (clean).

Bit Name	Offset	Description
OTHER_CORE_HITM	10	(R/W). L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoop where modified copies were found (HITM).
Reserved	11	Reserved
REMOTE_CACHE_FWD	12	(R/W). L3 Miss: local homed requests that missed the L3 cache and was serviced by forwarded data following a cross package snoop where no modified copies found. (Remote home requests are not counted)
REMOTE_DRAM	13	(R/W). L3 Miss: remote home requests that missed the L3 cache and were serviced by remote DRAM.
LOCAL_DRAM	14	(R/W). L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.
NON_DRAM	15	(R/W). Non-DRAM requests that were serviced by IOH.

...

27. Updates to Appendix B, Volume 3B

Change bars show changes to Appendix B of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

This appendix lists MSRs provided in Intel® Core™ 2 processor family, Intel® Atom™, Intel® Core™ Duo, Intel® Core™ Solo, Pentium® 4 and Intel® Xeon® processors, P6 family processors, and Pentium® processors in TablesB-12, B17 and B-18, respectively. All MSRs listed can be read with the RDMSR and written with the WRMSR instructions.

...

Table B-1 CPUID Signature Values of DisplayFamily_DisplayModel

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_2AH	Next Generation Intel Core Processor
06_2DH	Next Generation Intel Xeon Processor
...	

...

Table B-2 IA-32 Architectural MSRs

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Introduced as Architectural MSR
Hex	Decimal			
...				
19AH	410	IA32_CLOCK_MODULATION	Clock Modulation Control (R/W) See Section 14.5.3, "Software Controlled Clock Modulation."	0F_0H

		0	Extended On-Demand Clock Modulation Duty Cycle:	If CPUID.06H:EAX[5] = 1
		3:1	On-Demand Clock Modulation Duty Cycle: Specific encoded values for target duty cycle modulation	
		4	On-Demand Clock Modulation Enable: Set 1 to enable modulation	
		63:5	Reserved	
...				
19BH	411	IA32_THERM_INTERRUPT	Thermal Interrupt Control (R/W) Enables and disables the generation of an interrupt on temperature transitions detected with the processor's thermal sensors and thermal monitor. See Section 14.5.2, "Thermal Monitor."	0F_0H
		0	High-Temperature Interrupt Enable	
		1	Low-Temperature Interrupt Enable	
		2	PROCHOT# Interrupt Enable	
		3	FORCEPR# Interrupt Enable	
		4	Critical Temperature Interrupt Enable	
		7:5	Reserved	
		14:8	Threshold #1 Value	
		15	Threshold #1 Interrupt Enable	
		22:16	Threshold #2 Value	
		23	Threshold #2 Interrupt Enable	
		24	Power Limit Notification Enable	If CPUID.06H:EAX[4] = 1
		63:25	Reserved	
...				

19CH	412	IA32_THERM_STATUS	Thermal Status Information (RO) Contains status information about the processor's thermal sensor and automatic thermal monitoring facilities. See Section 14.5.2, "Thermal Monitor"	0F_0H
		0	Thermal Status (RO):	
		1	Thermal Status Log (R/W):	
		2	PROCHOT # or FORCEPR# event (RO)	
		3	PROCHOT # or FORCEPR# log (R/WC0)	
		4	Critical Temperature Status (RO)	
		5	Critical Temperature Status log (R/WC0)	
		6	Thermal Threshold #1 Status (RO)	If CPUID.01H:ECX[8] = 1
		7	Thermal Threshold #1 log (R/WC0)	If CPUID.01H:ECX[8] = 1
		8	Thermal Threshold #2 Status (RO)	If CPUID.01H:ECX[8] = 1
		9	Thermal Threshold #1 log (R/WC0)	If CPUID.01H:ECX[8] = 1
		10	Power Limitation Status (RO)	If CPUID.06H:EAX[4] = 1
		11	Power Limitation log (R/WC0)	If CPUID.06H:EAX[4] = 1
		15:12	Reserved	
		22:16	Digital Readout (RO)	If CPUID.06H:EAX[0] = 1
		26:23	Reserved	
		30:27	Resolution in Degrees Celsius (RO)	If CPUID.06H:EAX[0] = 1

		31	Reading Valid (RO)	If CPUID.06H:EAX[0] = 1
		63:32	Reserved	
...				
1B1H	433	IA32_PACKAGE_THERM_STATUS	Package Thermal Status Information (RO) Contains status information about the package's thermal sensor. See Section 14.6, "Package Level Thermal Management."	06_2AH
		0	Pkg Thermal Status (RO):	
		1	Pkg Thermal Status Log (R/W):	
		2	Pkg PROCHOT # event (RO)	
		3	Pkg PROCHOT # log (R/WC0)	
		4	Pkg Critical Temperature Status (RO)	
		5	Pkg Critical Temperature Status log (R/WC0)	
		6	Pkg Thermal Threshold #1 Status (RO)	
		7	Pkg Thermal Threshold #1 log (R/WC0)	
		8	Pkg Thermal Threshold #2 Status (RO)	
		9	Pkg Thermal Threshold #1 log (R/WC0)	
		10	Pkg Power Limitation Status (RO)	
		11	Pkg Power Limitation log (R/WC0)	
		15:12	Reserved	
		22:16	Pkg Digital Readout (RO)	
		63:23	Reserved	

1B2H	434	IA32_PACKAGE_THERM_INTERRUPT	Pkg Thermal Interrupt Control (R/W) Enables and disables the generation of an interrupt on temperature transitions detected with the package's thermal sensor. See Section 14.6, "Package Level Thermal Management."	06_2AH
		0	Pkg High-Temperature Interrupt Enable	
		1	Pkg Low-Temperature Interrupt Enable	
		2	Pkg PROCHOT# Interrupt Enable	
		3	Reserved	
		4	Pkr Overheat Interrupt Enable	
		7:5	Reserved	
		14:8	Pkg Threshold #1 Value	
		15	Pkg Threshold #1 Interrupt Enable	
		22:16	Pkg Threshold #2 Value	
		23	Pkg Threshold #2 Interrupt Enable	
		24	Pkg Power Limit Notification Enable	
		63:25	Reserved	
...				
40CH	1036	IA32_MC3_CTL	MC3_CTL	P6 Family Processors
40DH	1037	IA32_MC3_STATUS	MC3_STATUS	P6 Family Processors
40EH	1038	IA32_MC3_ADDR[1]	MC3_ADDR	P6 Family Processors
40FH	1039	IA32_MC3_MISC	MC3_MISC	P6 Family Processors
410H	1040	IA32_MC4_CTL	MC4_CTL	P6 Family Processors
411H	1041	IA32_MC4_STATUS	MC4_STATUS	P6 Family Processors
412H	1042	IA32_MC4_ADDR[1]	MC4_ADDR	P6 Family Processors

413H	1043	IA32_MC4_MISC	MC4_MISC	P6 Family Processors
...				

...

Table B-5 MSRs in Processors Based on Intel Microarchitecture codename Nehalem

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
...				
1A6H	422	MSR_OFFCORE_RSP_0	Thread	Offcore Response Event Select Register (R/W)
...				

...

Table B-8 Additional MSRs supported by Intel Processors (Intel microarchitecture codename Westmere)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1A7H	423	MSR_OFFCORE_RSP_1	Thread	Offcore Response Event Select Register (R/W)
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	see Table B-2

...

B.6 MSRS IN NEXT GENERATION INTEL® PROCESSOR FAMILY (CODENAME SANDY BRIDGE)

Table B-9 lists selected model-specific registers (MSRs) that are common to next generation for Intel® processor family (codename Sandy Bridge). All architectural MSRs listed in Table B-2 are supported. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2AH, 06_2DH, see Table B-1. Additional MSRs specific to 06_2AH are listed in Table B-10.

Table B-9 Selected MSRs supported by Next Generation Intel Processors (Intel microarchitecture codename Sandy Bridge)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1B1H	433	IA32_PACKAGE_THERM_STATUS	Package	see Table B-2
1B2H	434	IA32_PACKAGE_THERM_INTERRUPT	Package	see Table B-2
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces** (R/O) See Section 14.7.1, “RAPL Interfaces.”
610H	1552	MSR_PKG_RAPL_POWER_LIMIT	Package	**PKG RAPL Power Limit Control** (R/W) See Section 14.7.3, “Package RAPL Domain.”
611H	1553	MSR_PKG_ENERY_STATUS	Package	**PKG Energy Status** (R/O) See Section 14.7.3, “Package RAPL Domain.”
613H	1555	MSR_PKG_PERF_STATUS	Package	**PKG Performance Throttling Status** (R/O) See Section 14.7.3, “Package RAPL Domain.”
614H	1556	MSR_PKG_POWER_INFO	Package	**PKG RAPL Parameters** (R/W) See Section 14.7.3, “Package RAPL Domain.”
638H	1592	MSR_PP0_POWER_LIMIT	Package	**PP0 RAPL Power Limit Control** (R/W) See Section 14.7.4, “PP0/PP1 RAPL Domains.”
639H	1593	MSR_PP0_ENERY_STATUS	Package	**PP0 Energy Status** (R/O) See Section 14.7.4, “PP0/PP1 RAPL Domains.”
63AH	1594	MSR_PP0_POLICY	Package	**PP0 Balance Policy** (R/W) See Section 14.7.4, “PP0/PP1 RAPL Domains.”
63BH	1595	MSR_PP0_PERF_STATUS	Package	**PP0 Performance Throttling Status** (R/O) See Section 14.7.4, “PP0/PP1 RAPL Domains.”

B.6.1 MSRs In Next Generation Intel® Core Processor Family (codename Sandy Bridge)

Table B-10 lists selected model-specific registers (MSRs) that are specific to next generation for Intel® Core processor family (codename Sandy Bridge). These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2AH, see Table B-1.

Table B-10 Selected MSRs supported by Next Generation Intel Core Processors (Intel microarchitecture codename Sandy Bridge)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
640H	1600	MSR_PP1_POWER_LIMIT	Package	**PP1 RAPL Power Limit Control** (R/W) See Section 14.7.4, “PP0/PP1 RAPL Domains.”
641H	1601	MSR_PP1_ENERY_STATUS	Package	**PP1 Energy Status** (R/O) See Section 14.7.4, “PP0/PP1 RAPL Domains.”

Table B-10 Selected MSRs supported by Next Generation Intel Core Processors (Continued)(Intel microarchitecture codename Sandy Bridge)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
642H	1602	MSR_PP1_POLICY	Package	**PP1 Balance Policy** (R/W) See Section 14.7.4, "PP0/PP1 RAPL Domains."

B.6.2 MSRs In Next Generation Intel® Xeon Processor Family (codename Sandy Bridge)

Table B-11 lists selected model-specific registers (MSRs) that are specific to next generation for Intel® Xeon processor family (codename Sandy Bridge). These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2DH, see Table B-1.

Table B-11 Selected MSRs supported by Next Generation Intel Xeon Processors (Intel microarchitecture codename Sandy Bridge)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control** (R/W) See Section 14.7.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERY_STATUS	Package	**DRAM Energy Status** (R/O) See Section 14.7.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status** (R/O) See Section 14.7.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters** (R/W) See Section 14.7.5, "DRAM RAPL Domain."

...

Table B-17 MSRs in the P6 Family Processors

Register Address		Register Name	Bit Description
Hex	Dec		
...			
401H	1025	MC0_STATUS	
		15:0	MC_STATUS_MCACOD
		31:16	MC_STATUS_MSCOD
		57	MC_STATUS_DAM
		58	MC_STATUS_ADDRV
		59	MC_STATUS_MISCV
		60	MC_STATUS_EN. (Note: For MC0_STATUS only, this bit is hardcoded to 1.)

		61	MC_STATUS_UC
		62	MC_STATUS_O
		63	MC_STATUS_V
...			

...

28. Updates to Appendix E, Volume 3B

Change bars show changes to Appendix E of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

Table E-11 Incremental Memory Controller Error Codes of Machine Check for IA32_MC8_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA error codes[1]	0-15	MCACOD	Memory error format: 1MMMCCCC
Model specific errors			
	16	Read ECC error	if 1, ECC occurred on a read
	17	RAS ECC error	If 1, ECC occurred on a scrub
	18	Write parity error	If 1, bad parity on a write
	19	Redundancy loss	if 1, Error in half of redundant memory
	20	Reserved	Reserved
	21	Memory range error	If 1, Memory access out of range
	22	RTID out of range	If 1, Internal ID invalid
	23	Address parity error	If 1, bad address parity
	24	Byte enable parity error	If 1, bad enable parity
Other information	37-25	Reserved	Reserved
	52:38	CORE_ERR_CNT	Corrected error count
	56-53	Reserved	Reserved
Status register validity indicators[1]	57-63		

NOTES:
1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

...

29. Updates to Appendix H, Volume 3B

Change bars show changes to Appendix H of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

Table H-1 Encoding for 16-Bit Control Fields (0000_00xx_xxxx_xxx0B)

Field Name	Index	Encoding
Virtual-processor identifier (VPID)[1]	000000000B	00000000H

NOTES:

1. This field exists only on processors that support the 1-setting of the "enable VPID" VM-execution control.

...