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

Documentation Changes

September 2010

Document Number: 252046-029

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
-029	• Removed Documentation Changes 1-29 • Added Documentation Changes 1-24	September 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

No.	DOCUMENTATION CHANGES
1	Updates to Chapter 3, Volume 1
2	Updates to Chapter 3, Volume 2A
3	Updates to Chapter 4, Volume 2B
4	Updates to Chapter 5, Volume 2B
5	Updates to Chapter 6, Volume 2B
6	Updates to Chapter 2, Volume 3A
7	Updates to Chapter 4, Volume 3A
8	Updates to Chapter 6, Volume 3A
9	Updates to Chapter 8, Volume 3A
10	Updates to Chapter 9, Volume 3A
11	Updates to Chapter 10, Volume 3A
12	Updates to Chapter 13, Volume 3A
13	Updates to Chapter 15, Volume 3A
14	Updates to Chapter 16, Volume 3A
15	Updates to Chapter 22, Volume 3B
16	Updates to Chapter 23, Volume 3B
17	Updates to Chapter 24, Volume 3B
18	Updates to Chapter 25, Volume 3B
19	Updates to Chapter 27, Volume 3B
20	Updates to Chapter 30, Volume 3B
21	Updates to Appendix B, Volume 3B
22	Updates to Appendix E, Volume 3B
23	Updates to Appendix G, Volume 3B

Documentation Changes

1. Updates to Chapter 3, Volume 1

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

...

3.7.5.1 Specifying an Offset in 64-Bit Mode

The offset part of a memory address in 64-bit mode can be specified directly as a static value or through an address computation made up of one or more of the following components:

- **Displacement —** An 8-bit or 32-bit value.
- **Base —** The value in a 32-bit (or 64-bit if REX.W is set) general-purpose register.
- **Index —** The value in a 32-bit (or 64-bit if REX.W is set) general-purpose register.
- **Scale factor —** A value of 2, 4, or 8 that is multiplied by the index value.

...

2. 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.

...

3.1.1.8 Operation Section

The "Operation" section contains an algorithm description (frequently written in pseudo-code) for the instruction. Algorithms are composed of the following elements:

- Comments are enclosed within the symbol pairs "(*" and "*)".
- Compound statements are enclosed in keywords, such as: IF, THEN, ELSE and FI for an if statement; DO and OD for a do statement; or CASE... OF for a case statement.
- A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that register. For example, ES:[DI] indicates the contents of the location whose ES segment relative address is in register DI. [SI] indicates the contents of the address contained in register SI relative to the SI register's default segment (DS) or the overridden segment.
- Parentheses around the "E" in a general-purpose register name, such as (E)SI, indicates that the offset is read from the SI register if the address-size attribute is 16, from the ESI register if the address-size attribute is 32. Parentheses around the "R" in a general-purpose register name, (R)SI, in the presence of a 64-bit register

definition such as (R)SI, indicates that the offset is read from the 64-bit RSI register if the address-size attribute is 64.

- Brackets are used for memory operands where they mean that the contents of the memory location is a segment-relative offset. For example, [SRC] indicates that the content of the source operand is a segment-relative offset.
- $A \leftarrow B$ indicates that the value of B is assigned to A.
- The symbols $=$, $\neq$, $>$, $<$, $\geq$, and $\leq$ are relational operators used to compare two values: meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as $A \leftarrow B$ is TRUE if the value of A is equal to B; otherwise it is FALSE.
- The expression "« COUNT" and "» COUNT" indicates that the destination operand should be shifted left or right by the number of bits indicated by the count operand.

The following identifiers are used in the algorithmic descriptions:

- **OperandSize and AddressSize** — The OperandSize identifier represents the operand-size attribute of the instruction, which is 16, 32 or 64-bits. The AddressSize identifier represents the address-size attribute, which is 16, 32 or 64-bits. For example, the following pseudo-code indicates that the operand-size attribute depends on the form of the MOV instruction used.

```
IF Instruction ← MOVW
    THEN OperandSize = 16;
ELSE
    IF Instruction ← MOVD
        THEN OperandSize = 32;
    ELSE
        IF Instruction ← MOVQ
            THEN OperandSize = 64;
        FI;
    FI;
FI;
```

See "Operand-Size and Address-Size Attributes" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for guidelines on how these attributes are determined.

...

AND—Logical AND

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
...					
REX + 80 /4 *ib*	AND *r/m8**, *imm8*	B	Valid	N.E.	*r/m8* AND *imm8.*
...					

...

CALL—Call Procedure

. . .

Operation

```
. . .
CALL-GATE:
    IF call gate (DPL < CPL) or (RPL > DPL)
         THEN #GP(call-gate selector); FI;
    IF call gate not present
         THEN #NP(call-gate selector); FI;
    IF call-gate code-segment selector is NULL
         THEN #GP(0); FI;
    IF call-gate code-segment selector index is outside descriptor table limits
         THEN #GP(call-gate code-segment selector); FI;
    Read call-gate code-segment descriptor;
    IF call-gate code-segment descriptor does not indicate a code segment
    or call-gate code-segment descriptor DPL > CPL
         THEN #GP(call-gate code-segment selector); FI;
    IF IA32_EFER.LMA = 1 AND (call-gate code-segment descriptor is
    not a 64-bit code segment or call-gate code-segment descriptor has both L-bit and D-bit set)
         THEN #GP(call-gate code-segment selector); FI;
    IF call-gate code segment not present
         THEN #NP(call-gate code-segment selector); FI;
    IF call-gate code segment is non-conforming and DPL < CPL
         THEN go to MORE-PRIVILEGE;
         ELSE go to SAME-PRIVILEGE;
    FI;
END;

MORE-PRIVILEGE:
    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
              IF current TSS is 16-bit
                   THEN
                        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);
                   ELSE (* current TSS is 64-bit *)
                        TSSstackAddress ← (new code-segment DPL ∗ 8) + 4;
                        IF (TSSstackAddress + 7) > current TSS limit
                             THEN #TS(current TSS selector); FI;
```

```
                NewSS ← new code-segment DPL; (* NULL selector with RPL = new CPL *)
                NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
        FI;
FI;
IF IA32_EFER.LMA = 0 and NewSS is NULL
    THEN #TS(NewSS); FI;
Read new code-segment descriptor and new stack-segment descriptor;
IF IA32_EFER.LMA = 0 and (NewSS RPL ≠ new code-segment DPL
or new stack-segment DPL ≠ new code-segment DPL or new stack segment is not a
writable data segment)
    THEN #TS(NewSS); FI
IF IA32_EFER.LMA = 0 and new stack segment not present
    THEN #SS(NewSS); FI;
IF CallGateSize = 32
    THEN
        IF new stack does not have room for parameters plus 16 bytes
            THEN #SS(NewSS); FI;
        IF CallGate(InstructionPointer) not within new code-segment limit
            THEN #GP(0); FI;
        SS ← newSS; (* Segment descriptor information also loaded *)
        ESP ← newESP;
        CS:EIP ← CallGate(CS:InstructionPointer);
        (* Segment descriptor information also loaded *)
        Push(oldSS:oldESP); (* From calling procedure *)
        temp ← parameter count from call gate, masked to 5 bits;
        Push(parameters from calling procedure's stack, temp)
        Push(oldCS:oldEIP); (* Return address to calling procedure *)
    ELSE
        IF CallGateSize = 16
            THEN
                IF new stack does not have room for parameters plus 8 bytes
                    THEN #SS(NewSS); FI;
                IF (CallGate(InstructionPointer) AND FFFFH) not in new code-segment limit
                    THEN #GP(0); FI;
                SS ← newSS; (* Segment descriptor information also loaded *)
                ESP ← newESP;
                CS:IP ← CallGate(CS:InstructionPointer);
                (* Segment descriptor information also loaded *)
                Push(oldSS:oldESP); (* From calling procedure *)
                temp ← parameter count from call gate, masked to 5 bits;
                Push(parameters from calling procedure's stack, temp)
                Push(oldCS:oldEIP); (* Return address to calling procedure *)
            ELSE (* CallGateSize = 64 *)
                IF pushing 32 bytes on the stack would use a non-canonical address
                    THEN #SS(NewSS); FI;
                IF (CallGate(InstructionPointer) is non-canonical)
                    THEN #GP(0); FI;
                SS ← NewSS; (* NewSS is NULL)
                RSP ← NewESP;
                CS:IP ← CallGate(CS:InstructionPointer);
```

```
                        (* Segment descriptor information also loaded *)
                        Push(oldSS:oldESP); (* From calling procedure *)
                        Push(oldCS:oldEIP); (* Return address to calling procedure *)
                FI;
        FI;
        CPL ← CodeSegment(DPL)
        CS(RPL) ← CPL
END;
```

...

CMOV*cc*—Conditional Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
...					
0F 40 */r*	CMOVO *r16, r/m16*	A	Valid	Valid	Move if overflow (OF=1).
0F 40 */r*	CMOVO *r32, r/m32*	A	Valid	Valid	Move if overflow (OF=1).
REX.W + 0F 40 */r*	CMOVO *r64, r/m64*	A	Valid	N.E.	Move if overflow (OF=1).
...					

...

CPUID—CPU Identification

...

INPUT EAX = 1: Returns Model, Family, Stepping Information

...

The Extended Family ID needs to be examined only when the Family ID is 0FH. Integrate the fields into a display using the following rule:

```
IF Family_ID ≠ 0FH
    THEN DisplayFamily = Family_ID;
    ELSE DisplayFamily = Extended_Family_ID + Family_ID;
    (* Right justify and zero-extend 4-bit field. *)
FI;
(* Show DisplayFamily as HEX field. *)
```

The Extended Model ID needs to be examined only when the Family ID is 06H or 0FH. Integrate the field into a display using the following rule:

```
IF (Family_ID = 06H or Family_ID = 0FH)
    THEN DisplayModel = (Extended_Model_ID « 4) + Model_ID;
    (* Right justify and zero-extend 4-bit field; display Model_ID as HEX field.*)
    ELSE DisplayModel = Model_ID;
FI;
(* Show DisplayModel as HEX field. *)
```

...

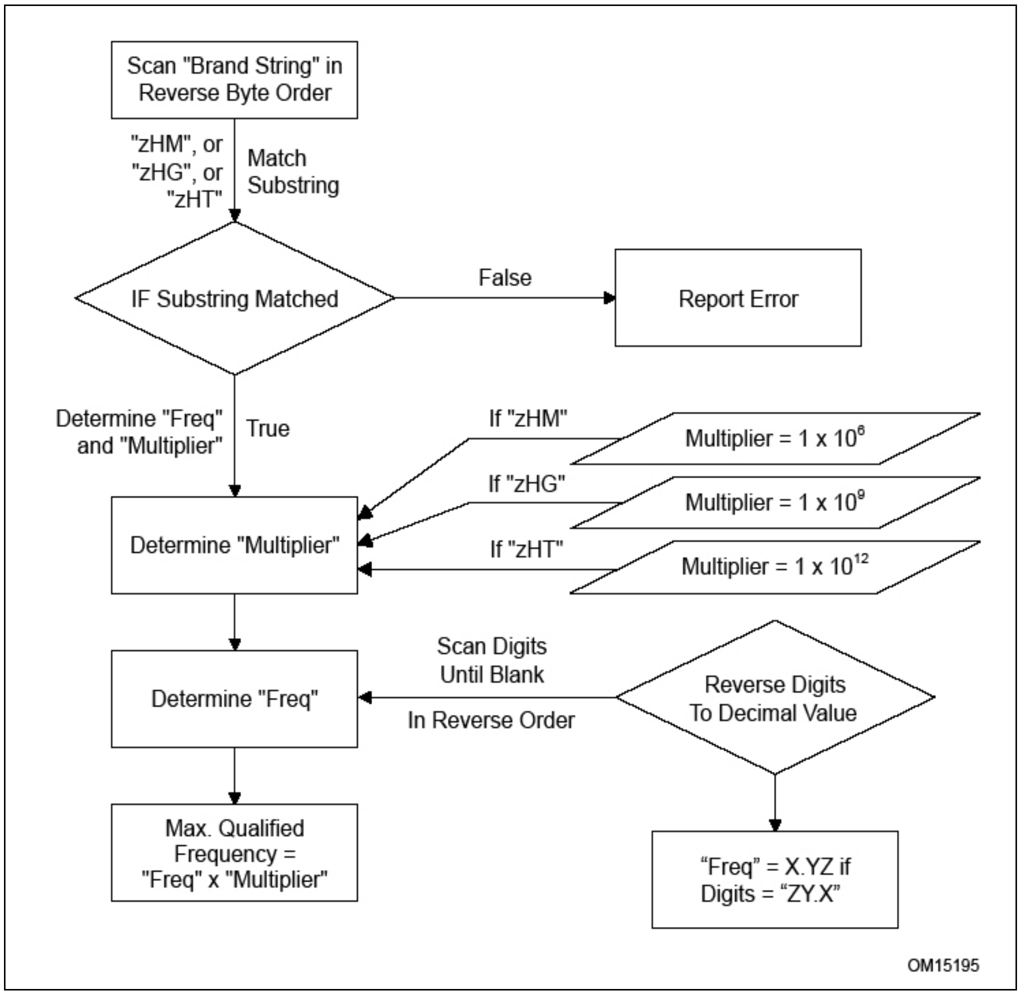


Figure 3-9. Algorithm for Extracting Maximum Processor Frequency

...

CVTPI2PD—Convert Packed Dword Integers to Packed Double-Precision FP Values

...

SIMD Floating-Point Exceptions

Precision

...

CVTSI2SD—Convert Dword Integer to Scalar Double-Precision FP Value

...

SIMD Floating-Point Exceptions

Precision

...

FXSAVE—Save x87 FPU, MMX Technology, and SSE State

...

The fields in Table 3-48 are defined in Table 3-49.

Table 3-49 Field Definitions

Field	Definition
FCW	x87 FPU Control Word (16 bits). See Figure 8-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the layout of the x87 FPU control word.
FSW	x87 FPU Status Word (16 bits). See Figure 8-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the layout of the x87 FPU status word.
Abridged FTW	x87 FPU Tag Word (8 bits). The tag information saved here is abridged, as described in the following paragraphs.
FOP	x87 FPU Opcode (16 bits). The lower 11 bits of this field contain the opcode, upper 5 bits are reserved. See Figure 8-8 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the layout of the x87 FPU opcode field.
FPU IP	x87 FPU Instruction Pointer Offset (32 bits). The contents of this field differ depending on the current addressing mode (32-bit or 16-bit) of the processor when the FXSAVE instruction was executed: 32-bit mode — 32-bit IP offset. 16-bit mode — low 16 bits are IP offset; high 16 bits are reserved. See "x87 FPU Instruction and Operand (Data) Pointers" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of the x87 FPU instruction pointer.
CS	x87 FPU Instruction Pointer Selector (16 bits).
FPU DP	x87 FPU Instruction Operand (Data) Pointer Offset (32 bits). The contents of this field differ depending on the current addressing mode (32-bit or 16-bit) of the processor when the FXSAVE instruction was executed: 32-bit mode — 32-bit DP offset. 16-bit mode — low 16 bits are DP offset; high 16 bits are reserved.
	See "x87 FPU Instruction and Operand (Data) Pointers" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of the x87 FPU operand pointer.
DS	x87 FPU Instruction Operand (Data) Pointer Selector (16 bits).

Table 3-49 Field Definitions (Continued)

Field	Definition
MXCSR	MXCSR Register State (32 bits). See Figure 10-3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the layout of the MXCSR register. If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save this register. This behavior is implementation dependent.
MXCSR_ MASK	MXCSR_MASK (32 bits). This mask can be used to adjust values written to the MXCSR register, ensuring that reserved bits are set to 0. Set the mask bits and flags in MXCSR to the mode of operation desired for SSE and SSE2 SIMD floating-point instructions. See "Guidelines for Writing to the MXCSR Register" in Chapter 11 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for instructions for how to determine and use the MXCSR_MASK value.
ST0/MM0 through ST7/MM7	x87 FPU or MMX technology registers. These 80-bit fields contain the x87 FPU data registers or the MMX technology registers, depending on the state of the processor prior to the execution of the FXSAVE instruction. If the processor had been executing x87 FPU instruction prior to the FXSAVE instruction, the x87 FPU data registers are saved; if it had been executing MMX instructions (or SSE or SSE2 instructions that operated on the MMX technology registers), the MMX technology registers are saved. When the MMX technology registers are saved, the high 16 bits of the field are reserved.
XMM0 through XMM7	XMM registers (128 bits per field). If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save these registers. This behavior is implementation dependent.

...

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

...

Description

The INT *n* instruction generates a call to the interrupt or exception handler specified with the destination operand (see the section titled "Interrupts and Exceptions" in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual*, *Volume 1*). The destination operand specifies an interrupt vector number from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each interrupt vector number provides an index to a gate descriptor in the IDT. The first 32 interrupt vector numbers are reserved by Intel for system use. Some of these interrupts are used for internally generated exceptions.

The INT *n* instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), interrupt vector number 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1. (The INTO instruction cannot be used in 64-bit mode.)

...

Operation

The following operational description applies not only to the INT *n* and INTO instructions, but also to external interrupts, nonmaskable interrupts (NMIs), and exceptions. Some of these events push onto the stack an error code.

The operational description specifies numerous checks whose failure may result in delivery of a nested exception. In these cases, the original event is not delivered.

The operational description specifies the error code delivered by any nested exception. In some cases, the error code is specified with a pseudofunction error_code(num,idt,ext), where idt and ext are bit values. The pseudofunction produces an error code as follows: (1) if idt is 0, the error code is (num & FCH) | ext; (2) if idt is 1, the error code is (num « 3) | 2 | ext.

In many cases, the pseudofunction error_code is invoked with a pseudovariable EXT. The value of EXT depends on the nature of the event whose delivery encountered a nested exception: if that event is a software interrupt, EXT is 0; otherwise, EXT is 1.

```
IF PE = 0
   THEN
       GOTO REAL-ADDRESS-MODE;
   ELSE (* PE = 1 *)
       IF (VM = 1 and IOPL < 3 AND INT n)
           THEN
                #GP(0); (* Bit 0 of error code is 0 because INT n *)
           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 « 2) + 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 in real-address mode*)
   CS ← IDT(Descriptor (vector_number « 2), selector));
   EIP ← IDT(Descriptor (vector_number « 2), 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(error_code(vector_number,1,EXT)); FI;
```

```
                (* idt operand to error_code set because vector is used *)
            IF software interrupt (* Generated by INT n, INT3, or INTO *)
                THEN
                    IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
                        THEN #GP(error_code(vector_number,1,0)); FI;
                        (* idt operand to error_code set because vector is used *)
                        (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
            FI;
            IF gate not present
                THEN #NP(error_code(vector_number,1,EXT)); FI;
                (* idt operand to error_code set because vector is used *)
            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 INTO and CS.L = 1 (64-bit mode)
                THEN #UD;
            FI;
            IF ((vector_number « 4) + 15) is not in IDT limits
            or selected IDT descriptor is not an interrupt-, or trap-gate type
                THEN #GP(error_code(vector_number,1,EXT));
                (* idt operand to error_code set because vector is used *)
            FI;
            IF software interrupt (* Generated by INT n, INT 3, or INTO *)
                THEN
                    IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
                        THEN #GP(error_code(vector_number,1,0));
                        (* idt operand to error_code set because vector is used *)
                        (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
                    FI;
            FI;
            IF gate not present
                THEN #NP(error_code(vector_number,1,EXT));
                (* idt operand to error_code set because vector is used *)
            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(error_code(TSS selector,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
                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,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
                IF TSS not present
                    THEN #NP(TSS selector,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
            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(EXT); FI;
                    Push(error code);
            FI;
            IF EIP not within code segment limit
                THEN #GP(EXT); 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(EXT); FI; (* Error code contains NULL selector *)
            IF new code-segment selector is not within its descriptor table limits
                THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            Read descriptor referenced by new code-segment selector;
            IF descriptor does not indicate a code segment or new code-segment DPL > CPL
                THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            IF new code-segment descriptor is not present,
                THEN #NP(error_code(new code-segment selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            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(error_code(new code-segment selector,0,EXT));
                                (* idt operand to error_code is 0 because selector is used *)
                            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(error_code(new code-segment selector,0,EXT));
                        (* idt operand to error_code is 0 because selector is used *)
                    IF new code segment is conforming or new code-segment DPL = CPL
                        THEN
                            GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
                        ELSE (* PE = 1, interrupt or trap gate, nonconforming code segment, DPL > CPL *)
                            #GP(error_code(new code-segment selector,0,EXT));
                            (* idt operand to error_code is 0 because selector is used *)
                    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 « 3) + 4;
                            IF (TSSstackAddress + 5) > current TSS limit
                                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                                (* idt operand to error_code is 0 because selector is used *)
                            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 « 2) + 2
                            IF (TSSstackAddress + 3) > current TSS limit
                                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                                (* idt operand to error_code is 0 because selector is used *)
                            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(error_code(NewSS,0,EXT)); FI;
                        (* idt operand to error_code is 0 because selector is used *)
                    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(error_code(NewSS,0,EXT)); FI;
                        (* idt operand to error_code is 0 because selector is used *)
                    IF NewSS is not present
                        THEN #SS(error_code(NewSS,0,EXT)); FI;
                        (* idt operand to error_code is 0 because selector is used *)
                ELSE (* IA-32e mode *)
                    IF IDT-gate IST = 0
                        THEN TSSstackAddress ← (new code-segment DPL « 3) + 4;
                        ELSE TSSstackAddress ← (IDT gate IST « 3) + 28;
                    FI;
                    IF (TSSstackAddress + 7) > current TSS limit
                        THEN #TS(error_code(current TSS selector,0,EXT); FI;
                        (* idt operand to error_code is 0 because selector is used *)
                    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(error_code(NewSS,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
            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(error_code(NewSS,0,EXT)); FI;
                        (* idt operand to error_code is 0 because selector is used *)
            ELSE (* 64-bit IDT gate*)
                IF StackAddress is non-canonical
                    THEN #SS(EXT); FI; (* Error code contains NULL selector *)
        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(EXT); FI; (* Error code contains NULL selector *)
        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(EXT); FI; (* Error code contains NULL selector *)
        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(error_code(current TSS selector,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
                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(error_code(current TSS selector,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
                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; (* Error code contains NULL selector *)
        IF NewSS index is not within its descriptor table limits
        or NewSS RPL ≠ 0
            THEN #TS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        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(error_code(NewSS,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
IF new stack segment not present
    THEN #SS(error_code(NewSS,0,EXT)); FI;
    (* idt operand to error_code is 0 because selector is used *)
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(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
    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(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
FI;
IF instruction pointer from IDT gate is not within new code-segment limits
    THEN #GP(EXT); FI; (* Error code contains NULL selector *)
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 « 3) + 28;
                    IF (TSSstackAddress + 7) > TSS limit
                        THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                        (* idt operand to error_code is 0 because selector is used *)
                    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(EXT); FI; (* Error code contains NULL selector *)
            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(EXT); FI; (* Error code contains NULL selector *)
            ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
                    IF NewRSP contains a non-canonical address
                        THEN #SS(EXT); (* Error code contains NULL selector *)
            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(EXT); FI; (* Error code contains NULL selector *)
            ELSE
                IF instruction pointer from IDT gate contains a non-canonical address
                    THEN #GP(EXT); FI; (* Error code contains NULL selector *)
                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;
```

...

Protected Mode Exceptions

#GP(error_code)	If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.
	If the segment selector in the interrupt-, trap-, or task gate is NULL.
	If an interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.
	If the interrupt vector number is outside the IDT limits.
	If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.
	If an interrupt is generated by the INT *n*, INT 3, or INTO instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.
	If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.
#SS(error_code)	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.
	If the SS register is being loaded and the segment pointed to is marked not present.
	If pushing the return address, flags, error code, or stack segment pointer exceeds the bounds of the new stack segment when a stack switch occurs.

#NP(error_code)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(error_code)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.
	If DPL of the stack segment descriptor pointed to by the stack segment selector in the TSS is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
	If the stack segment selector in the TSS is NULL.
	If the stack segment for the TSS is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the interrupt vector number is outside the IDT limits.
#SS	If stack limit violation on push.
	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(error_code)	(For INT *n,* INTO, or BOUND instruction) If the IOPL is less than 3 or the DPL of the interrupt-, trap-, or task-gate descriptor is not equal to 3.
	If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.
	If the segment selector in the interrupt-, trap-, or task gate is NULL.
	If a interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.
	If the interrupt vector number is outside the IDT limits.
	If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.
	If an interrupt is generated by the INT *n* instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.
	If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
#SS(error_code)	If the SS register is being loaded and the segment pointed to is marked not present.
	If pushing the return address, flags, error code, stack segment pointer, or data segments exceeds the bounds of the stack segment.

#NP(error_code)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(error_code)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.
	If DPL of the stack segment descriptor for the TSS's stack segment is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
	If the stack segment selector in the TSS is NULL.
	If the stack segment for the TSS is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#BP	If the INT 3 instruction is executed.
#OF	If the INTO instruction is executed and the OF flag is set.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(error_code)	If the instruction pointer in the 64-bit interrupt gate or 64-bit trap gate is non-canonical.
	If the segment selector in the 64-bit interrupt or trap gate is NULL.
	If the interrupt vector number is outside the IDT limits.
	If the interrupt vector number points to a gate which is in non-canonical space.
	If the interrupt vector number points to a descriptor which is not a 64-bit interrupt gate or 64-bit trap gate.
	If the descriptor pointed to by the gate selector is outside the descriptor table limit.
	If the descriptor pointed to by the gate selector is in non-canonical space.
	If the descriptor pointed to by the gate selector is not a code segment.
	If the descriptor pointed to by the gate selector doesn't have the L-bit set, or has both the L-bit and D-bit set.
	If the descriptor pointed to by the gate selector has DPL > CPL.
#SS(error_code)	If a push of the old EFLAGS, CS selector, EIP, or error code is in non-canonical space with no stack switch.
	If a push of the old SS selector, ESP, EFLAGS, CS selector, EIP, or error code is in non-canonical space on a stack switch (either CPL change or no-CPL with IST).
#NP(error_code)	If the 64-bit interrupt-gate, 64-bit trap-gate, or code segment is not present.

#TS(error_code)	If an attempt to load RSP from the TSS causes an access to non-canonical space.
	If the RSP from the TSS is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
	If INTO.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

...

IRET/IRETD—Interrupt Return

...

Operation

```
IF PE = 0
    THEN
        GOTO REAL-ADDRESS-MODE;
    ELSE
        IF (IA32_EFER.LMA = 0)
                THEN (* Protected mode *)
                    GOTO PROTECTED-MODE;
                ELSE (* IA-32e mode *)
                    GOTO IA-32e-MODE;
        FI;
FI;
REAL-ADDRESS-MODE;
    IF OperandSize = 32
        THEN
            IF top 12 bytes of stack not within stack limits
                THEN #SS; FI;
            tempEIP ← 4 bytes at end of stack
            IF tempEIP[31:16] is not zero THEN #GP(0); FI;
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            tempEFLAGS ← Pop();
            EFLAGS ← (tempEFLAGS AND 257FD5H) OR (EFLAGS AND 1A0000H);
        ELSE (* OperandSize = 16 *)
            IF top 6 bytes of stack are not within stack limits
                THEN #SS; FI;
            EIP ← Pop(); (* 16-bit pop; clear upper 16 bits *)
            CS ← Pop(); (* 16-bit pop *)
            EFLAGS[15:0] ← Pop();
    FI;
    END;
PROTECTED-MODE:
    IF VM = 1 (* Virtual-8086 mode: PE = 1, VM = 1 *)
        THEN
```

```
                GOTO RETURN-FROM-VIRTUAL-8086-MODE; (* PE = 1, VM = 1 *)
        FI;
        IF NT = 1
            THEN
                GOTO TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
        FI;
        IF OperandSize = 32
            THEN
                IF top 12 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                tempEIP ← Pop();
                tempCS ← Pop();
                tempEFLAGS ← Pop();
            ELSE (* OperandSize = 16 *)
                IF top 6 bytes of stack are not within stack limits
                    THEN #SS(0); FI;
                tempEIP ← Pop();
                tempCS ← Pop();
                tempEFLAGS ← Pop();
                tempEIP ← tempEIP AND FFFFH;
                tempEFLAGS ← tempEFLAGS AND FFFFH;
        FI;
        IF tempEFLAGS(VM) = 1 and CPL = 0
            THEN
                GOTO RETURN-TO-VIRTUAL-8086-MODE;
            ELSE
                GOTO PROTECTED-MODE-RETURN;
        FI;
    IA-32e-MODE:
        IF NT = 1
            THEN #GP(0);
        ELSE IF OperandSize = 32
            THEN
                IF top 12 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                tempEIP ← Pop();
                tempCS ← Pop();
                tempEFLAGS ← Pop();
            ELSE IF OperandSize = 16
                THEN
                    IF top 6 bytes of stack are not within stack limits
                        THEN #SS(0); FI;
                    tempEIP ← Pop();
                    tempCS ← Pop();
                    tempEFLAGS ← Pop();
                    tempEIP ← tempEIP AND FFFFH;
                    tempEFLAGS ← tempEFLAGS AND FFFFH;
                FI;
            ELSE (* OperandSize = 64 *)
                THEN
```

```
                                tempRIP ← Pop();
                                tempCS ← Pop();
                                tempEFLAGS ← Pop();
                                tempRSP ← Pop();
                                tempSS ← Pop();
            FI;
            GOTO IA-32e-MODE-RETURN;
```

. . .

JMP—Jump

. . .

Operation

```
IF near jump
    IF 64-bit Mode
        THEN
            IF near relative jump
             THEN
                tempRIP ← RIP + DEST; (* RIP is instruction following JMP instruction*)
             ELSE (* Near absolute jump *)
                tempRIP ← DEST;
            FI;
        ELSE
            IF near relative jump
             THEN
                tempEIP ← EIP + DEST; (* EIP is instruction following JMP instruction*)
             ELSE (* Near absolute jump *)
                tempEIP ← DEST;
            FI;
    FI;
    IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode)
    and tempEIP outside code segment limit
        THEN #GP(0); FI
    IF 64-bit mode and tempRIP is not canonical
        THEN #GP(0);
    FI;
    IF OperandSize = 32
         THEN
            EIP ← tempEIP;
         ELSE
            IF OperandSize = 16
                THEN (* OperandSize = 16 *)
                    EIP ← tempEIP AND 0000FFFFH;
                 ELSE (* OperandSize = 64)
                    RIP ← tempRIP;
            FI;
     FI;
FI;
```

```
IF far jump and (PE = 0 or (PE = 1 AND VM = 1)) (* Real-address or virtual-8086 mode *)
    THEN
        tempEIP ← DEST(Offset); (* DEST is ptr16:32 or [m16:32] *)
        IF tempEIP is beyond code segment limit
            THEN #GP(0); FI;
        CS ← DEST(segment selector); (* DEST is ptr16:32 or [m16:32] *)
        IF OperandSize = 32
            THEN
                EIP ← tempEIP; (* DEST is ptr16:32 or [m16:32] *)
            ELSE (* OperandSize = 16 *)
                EIP ← tempEIP AND 0000FFFFH; (* Clear upper 16 bits *)
        FI;
FI;
IF far jump and (PE = 1 and VM = 0)
(* IA-32e mode or protected mode, not virtual-8086 mode *)
    THEN
        IF effective address in the CS, DS, ES, FS, GS, or SS segment is illegal
        or segment selector in target operand NULL
                THEN #GP(0); FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new selector); FI;
        Read type and access rights of segment descriptor;
        IF (EFER.LMA = 0)
            THEN
                IF segment type is not a conforming or nonconforming code
                segment, call gate, task gate, or TSS
                    THEN #GP(segment selector); FI;
            ELSE
                IF segment type is not a conforming or nonconforming code segment
                call gate
                    THEN #GP(segment selector); FI;
        FI;
        Depending on type and access rights:
            GO TO CONFORMING-CODE-SEGMENT;
            GO TO NONCONFORMING-CODE-SEGMENT;
            GO TO CALL-GATE;
            GO TO TASK-GATE;
            GO TO TASK-STATE-SEGMENT;
    ELSE
        #GP(segment selector);
FI;
```

...

LSL—Load Segment Limit

...

Real-Address Mode Exceptions

#UD	The LSL instruction cannot be executed in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The LSL instruction cannot be executed in virtual-8086 mode.

...

3. 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.

...

PMOVMSKB—Move Byte Mask

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F D7 /*r*	PMOVMSKB *reg, mm*	A	Valid	Valid	Move a byte mask of *mm* to *reg*. The upper bits of r32 or r64 are zeroed
66 0F D7 /*r*	PMOVMSKB *reg, xmm*	A	Valid	Valid	Move a byte mask of *xmm* to *reg*. The upper bits of r32 or r64 are zeroed

Instruction Operand Encoding

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

Description

Creates a mask made up of the most significant bit of each byte of the source operand (second operand) and stores the result in the low byte or word of the destination operand (first operand). The source operand is an MMX technology register or an XMM register; the destination operand is a general-purpose register. When operating on 64-bit operands, the byte mask is 8 bits; when operating on 128-bit operands, the byte mask is 16-bits.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

...

PUSH—Push Word, Doubleword or Quadword Onto the Stack

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
...					
68	PUSH *imm16*	C	Valid	Valid	Push *imm16. Stack pointer is decremented by the size of stack pointer.*
...					

...

Table 4-12 lists valid indices of the general-purpose and special-purpose performance counters according to the derived DisplayFamily_DisplayModel values of CPUID encoding for each processor family (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-M" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

Table 4-12 Valid General and Special Purpose Performance Counter Index Range for RDPMC

Processor Family	DisplayFamily_Display Model/ Other Signatures	Valid PMC Index Range	General-purpose Counters
P6	06H_01H, 06H_03H, 06H_05H, 06H_06H, 06H_07H, 06H_08H, 06H_0AH, 06H_0BH	0, 1	0, 1
Pentium® 4, Intel® Xeon processors	0FH_00H, 0FH_01H, 0FH_02H	≥ 0 and ≤ 17	≥ 0 and ≤ 17
Pentium 4, Intel Xeon processors	(0FH_03H, 0FH_04H, 0FH_06H) and (L3 is absent)	≥ 0 and ≤ 17	≥ 0 and ≤ 17
Pentium M processors	06H_09H, 06H_0DH	0, 1	0, 1
64-bit Intel Xeon processors with L3	0FH_03H, 0FH_04H) and (L3 is present)	≥ 0 and ≤ 25	≥ 0 and ≤ 17
Intel® Core™ Solo and Intel® Core™ Duo processors, Dual-core Intel® Xeon® processor LV	06H_0EH	0, 1	0, 1
Intel® Core™2 Duo processor, Intel Xeon processor 3000, 5100, 5300, 7300 Series - general-purpose PMC	06H_0FH	0, 1	0, 1
Intel Xeon processors 7100 series with L3	(0FH_06H) and (L3 is present)	≥ 0 and ≤ 25	≥ 0 and ≤ 17
Intel® Core™2 Duo processor family, Intel Xeon processor family - general-purpose PMC	06H_17H	0, 1	0, 1
Intel Xeon processors 7400 series	(06H_1DH)	≥ 0 and ≤ 9	0, 1
Intel® Atom™ processor family	06H_1CH	0, 1	0, 1
Intel® Core™i7 processor, Intel Xeon processors 5500 series	06H_1AH, 06H_1EH, 06H_1FH, 06H_2EH	0-3	0, 1, 2, 3

...

RSM—Resume from System Management Mode

...

Operation

```
ReturnFromSMM;
IF (IA-32e mode supported) or (CPUID DisplayFamily_DisplayModel = 06H_0CH )
    THEN
        ProcessorState ← Restore(SMMDump(IA-32e SMM STATE MAP));
    Else
        ProcessorState ← Restore(SMMDump(Non-32-Bit-Mode SMM STATE MAP));
FI
```

...

SYSENTER—Fast System Call

...

Operation

```
IF CR0.PE = 0 THEN #GP(0); FI;
IF SYSENTER_CS_MSR[15:2] = 0 THEN #GP(0); FI;
EFLAGS.VM ← 0;                          (* ensures protected mode execution *)
EFLAGS.IF ← 0;                          (* Mask interrupts *)
EFLAGS.RF ← 0;

CS.SEL ← SYSENTER_CS_MSR                (* Operating system provides CS *)
(* Set rest of CS to a fixed value *)
CS.SEL.RPL ← 0;
CS.BASE ← 0;                            (* Flat segment *)
CS.ARbyte.G ← 1;                        (* 4-KByte granularity *)
CS.ARbyte.S ← 1;
CS.ARbyte.TYPE ← 1011B;                 (* Execute + Read, Accessed *)
CS.ARbyte.D ← 1;                        (* 32-bit code segment*)
CS.ARbyte.DPL ← 0;
CS.ARbyte.P ← 1;
CS.LIMIT ← FFFFFH;                      (* with 4-KByte granularity, implies a 4-GByte limit *)
CPL ← 0;

SS.SEL ← CS.SEL + 8;
(* Set rest of SS to a fixed value *)
SS.SEL.RPL ← 0;
SS.BASE ← 0;                            (* Flat segment *)
SS.ARbyte.G ← 1;                        (* 4-KByte granularity *)
SS.ARbyte.S ← 1;
SS.ARbyte.TYPE ← 0011B;                 (* Read/Write, Accessed *)
SS.ARbyte.D ← 1;                        (* 32-bit stack segment*)
SS.ARbyte.DPL ← 0;
SS.ARbyte.P ← 1;
SS.LIMIT ← FFFFFH;                      (* with 4-KByte granularity, implies a 4-GByte limit *)
```

```
ESP ← SYSENTER_ESP_MSR;
EIP ← SYSENTER_EIP_MSR;
```

...

SYSEXIT—Fast Return from Fast System Call

...

Operation

```
IF SYSENTER_CS_MSR[15:2] = 0 THEN #GP(0); FI;
IF CR0.PE = 0 THEN #GP(0); FI;
IF CPL ≠ 0 THEN #GP(0); FI;

CS.SEL ← (SYSENTER_CS_MSR + 16);          (* Segment selector for return CS *)
(* Set rest of CS to a fixed value *)
CS.SEL.RPL ← 3;
CS.BASE ← 0;                              (* Flat segment *)
CS.ARbyte.G ← 1;                          (* 4-KByte granularity *)
CS.ARbyte.S ← 1;
CS.ARbyte.TYPE ← 1011B;                   (* Execute, Read, Non-Conforming Code *)
CS.ARbyte.D ← 1;                          (* 32-bit code segment*)
CS.ARbyte.DPL ← 3;
CS.ARbyte.P ← 1;
CS.LIMIT ← FFFFFH;                        (* with 4-KByte granularity, implies a 4-GByte limit *)
CPL ← 3;

SS.SEL ← (SYSENTER_CS_MSR + 24);          (* Segment selector for return SS *)
(* Set rest of SS to a fixed value *);
SS.SEL.RPL ← 3;
SS.BASE ← 0;                              (* Flat segment *)
SS.ARbyte.G ←1;                           (* 4-KByte granularity *)
SS.ARbyte.S ← 1;
SS.ARbyte.TYPE ← 0011B;                   (* Expand Up, Read/Write, Data *)
SS.ARbyte.D ← 1;                          (* 32-bit stack segment*)
SS.ARbyte.DPL ← 3;
SS.ARbyte.P ← 1;
SS.LIMIT ← FFFFFH;                        (* with 4-KByte granularity, implies a 4-GByte limit *)

ESP ← ECX;
EIP ← EDX;
```

...

4. 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.

VMXON—Enter VMX Operation

...

Operation

```
IF (register operand) 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[1]) or
        (bit 0 (lock bit) of IA32_FEATURE_CONTROL MSR is clear) or
        (in SMX operation[2] 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);
            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;
```

...

1. See Section 19.8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.
2. 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."

VMXON—Enter VMX Operation

. . .

Operation

```
IF (register operand) 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[1]) or
        (bit 0 (lock bit) of IA32_FEATURE_CONTROL MSR is clear) or
        (in SMX operation[2] 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);
            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;
```

1. See Section 19.8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.
2. 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."

...

5. 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[EXITAC]—Exit Authenticated Code Execution Mode

...

Operation

```
(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)
IF (CR4.SMXE=0)
    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 ( (in 64-bit mode) and ( RBX is non-canonical) )
    (CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or
    (ACMODEFLAG=0) or (IN_SMM=1)) or (EDX != 0))
    THEN #GP(0);
IF (OperandSize = 32)
    THEN tempEIP← EBX;
ELSIF (OperandSize = 64)
    THEN tempEIP← RBX;
ELSE
    tempEIP← EBX AND 0000FFFFH;
IF (tempEIP > code segment limit)
    THEN #GP(0);
Invalidate ACRAM contents;
Invalidate processor TLB(s);
Drain outgoing messages;
SignalTXTMsg(CloseLocality3);
SignalTXTMsg(LockSMRAM);
SignalTXTMsg(ProcessorRelease);
Unmask INIT;
IF (SENTERFLAG=0)
    THEN Unmask SMI, INIT, NMI, and A20M pin event;
ELSEIF (IA32_SMM_MONITOR_CTL[0] = 0)
    THEN Unmask SMI pin event;
ACMODEFLAG← 0;
EIP← tempEIP;
END;
```

...

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

...

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);
IF (IA32_SMM_MONITOR_CTL[0] != ILP.IA32_SMM_MONITOR_CTL[0])
    THEN TXT-SHUTDOWN(#IllegalEvent)
IF (IA32_SMM_MONITOR_CTL[0] = 0)
    THEN Unmask SMI pin event;
ELSE
    Mask SMI pin event;
Mask 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;
```

...

6. Updates to Chapter 2, Volume 3A

Change bars show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

2.2 MODES OF OPERATION

...

The VM flag in the EFLAGS register determines whether the processor is operating in protected mode or virtual-8086 mode. Transitions between protected mode and virtual-8086 mode are generally carried out as part of a task switch or a return from an interrupt or exception handler. See also: Section 17.2.5, "Entering Virtual-8086 Mode."

The LMA bit (IA32_EFER.LMA[bit 10]) determines whether the processor is operating in IA-32e mode. When running in IA-32e mode, 64-bit or compatibility sub-mode operation is determined by CS.L bit of the code segment. The processor enters into IA-32e mode from protected mode by enabling paging and setting the LME bit (IA32_EFER.LME[bit 8]). See also: Chapter 9, "Processor Management and Initialization."

The processor switches to SMM whenever it receives an SMI while the processor is in real-address, protected, virtual-8086, or IA-32e modes. Upon execution of the RSM instruction, the processor always returns to the mode it was in when the SMI occurred.

...

7. Updates to Chapter 4, Volume 3A

Change bars show changes to Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

4.1.1 Three Paging Modes

If CR0.PG = 0, paging is not used. The logical processor treats all linear addresses as if they were physical addresses. CR4.PAE and IA32_EFER.LME are ignored by the processor, as are CR0.WP, CR4.PSE, and CR4.PGE, and IA32_EFER.NXE.

Paging is enabled if CR0.PG = 1. Paging can be enabled only if protection is enabled (CR0.PE = 1). If paging is enabled, one of three paging modes is used. The values of CR4.PAE and IA32_EFER.LME determine which paging mode is used:

- If CR0.PG = 1 and CR4.PAE = 0, **32-bit paging** is used. 32-bit paging is detailed in Section 4.3. 32-bit paging uses CR0.WP, CR4.PSE, and CR4.PGE as described in Section 4.1.3.
- If CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LME = 0, **PAE paging** is used. PAE paging is detailed in Section 4.4. PAE paging uses CR0.WP, CR4.PGE, and IA32_EFER.NXE as described in Section 4.1.3.
- If CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LME = 1, **IA-32e paging** is used.[1] IA-32e paging is detailed in Section 4.5. IA-32e paging uses CR0.WP, CR4.PGE, CR4.PCIDE, and IA32_EFER.NXE as described in Section 4.1.3. IA-32e paging is available only on processors that support the Intel 64 architecture.

The three paging modes differ with regard to the following details:

- Linear-address width. The size of the linear addresses that can be translated.
- Physical-address width. The size of the physical addresses produced by paging.
- Page size. The granularity at which linear addresses are translated. Linear addresses on the same page are translated to corresponding physical addresses on the same page.
- Support for execute-disable access rights. In some paging modes, software can be prevented from fetching instructions from pages that are otherwise readable.

Table Table 4-1. illustrates the key differences between the three paging modes.

Table 4-1. Properties of Different Paging Modes

Paging Mode	CR0.PG	CR4.PAE	LME in IA32_EFER	Linear-Address Width	Physical-Address Width[1]	Page Size(s)	Supports Execute-Disable?
None	0	N/A	N/A	32	32	N/A	No
32-bit	1	0	0[2]	32	Up to 40[3]	4-KByte 4-MByte[4]	No
PAE	1	1	0	32	Up to 52	4-KByte 2-MByte	Yes[5]

1. The LMA flag in the IA32_EFER MSR (bit 10) is a status bit that indicates whether the logical processor is in IA-32e mode (and thus using IA-32e paging). The processor always sets IA32_EFER.LMA to CR0.PG & IA32_EFER.LME. Software cannot directly modify IA32_EFER.LMA; an execution of WRMSR to the IA32_EFER MSR ignores bit 10 of its source operand.

Table 4-1. Properties of Different Paging Modes (Continued)

Paging Mode	CR0.PG	CR4.PAE	LME in IA32_EFER	Linear-Address Width	Physical-Address Width[1]	Page Size(s)	Supports Execute-Disable?
IA-32e	1	1	2	48	Up to 52	4-KByte 2-MByte 1-GByte[6]	Yes[5]

NOTES:

1. The physical-address width is always bounded by MAXPHYADDR; see Section 4.1.4.
2. The processor ensures that IA32_EFER.LME must be 0 if CR0.PG = 1 and CR4.PAE = 0.
3. 32-bit paging supports physical-address widths of more than 32 bits only for 4-MByte pages and only if the PSE-36 mechanism is supported; see Section 4.1.4 and Section 4.3.
4. 4-MByte pages are used with 32-bit paging only if CR4.PSE = 1; see Section 4.3.
5. Execute-disable access rights are applied only if IA32_EFER.NXE = 1; see Section 4.6.
6. Not all processors that support IA-32e paging support 1-GByte pages; see Section 4.1.4.

Because they are used only if IA32_EFER.LME = 0, 32-bit paging and PAE paging is used only in legacy protected mode. Because legacy protected mode cannot produce linear addresses larger than 32 bits, 32-bit paging and PAE paging translate 32-bit linear addresses.

Because it is used only if IA32_EFER.LME = 1, IA-32e paging is used only in IA-32e mode. (In fact, it is the use of IA-32e paging that defines IA-32e mode.) IA-32e mode has two sub-modes:

- Compatibility mode. This mode uses only 32-bit linear addresses. IA-32e paging treats bits 47:32 of such an address as all 0.
- 64-bit mode. While this mode produces 64-bit linear addresses, the processor ensures that bits 63:47 of such an address are identical.[1] IA-32e paging does not use bits 63:48 of such addresses.

4.1.2 Paging-Mode Enabling

If CR0.PG = 1, a logical processor is in one of three paging modes, depending on the values of CR4.PAE and IA32_EFER.LME. Figure Figure 4-1. illustrates how software can enable these modes and make transitions between them. The following items identify certain limitations and other details:

- IA32_EFER.LME cannot be modified while paging is enabled (CR0.PG = 1). Attempts to do so using WRMSR cause a general-protection exception (#GP(0)).
- Paging cannot be enabled (by setting CR0.PG to 1) while CR4.PAE = 0 and IA32_EFER.LME = 1. Attempts to do so using MOV to CR0 cause a general-protection exception (#GP(0)).
- CR4.PAE cannot be cleared while IA-32e paging is active (CR0.PG = 1 and IA32_EFER.LME = 1). Attempts to do so using MOV to CR4 cause a general-protection exception (#GP(0)).

1. Such an address is called **canonical**. Use of a non-canonical linear address in 64-bit mode produces a general-protection exception (#GP(0)); the processor does not attempt to translate non-canonical linear addresses using IA-32e paging.

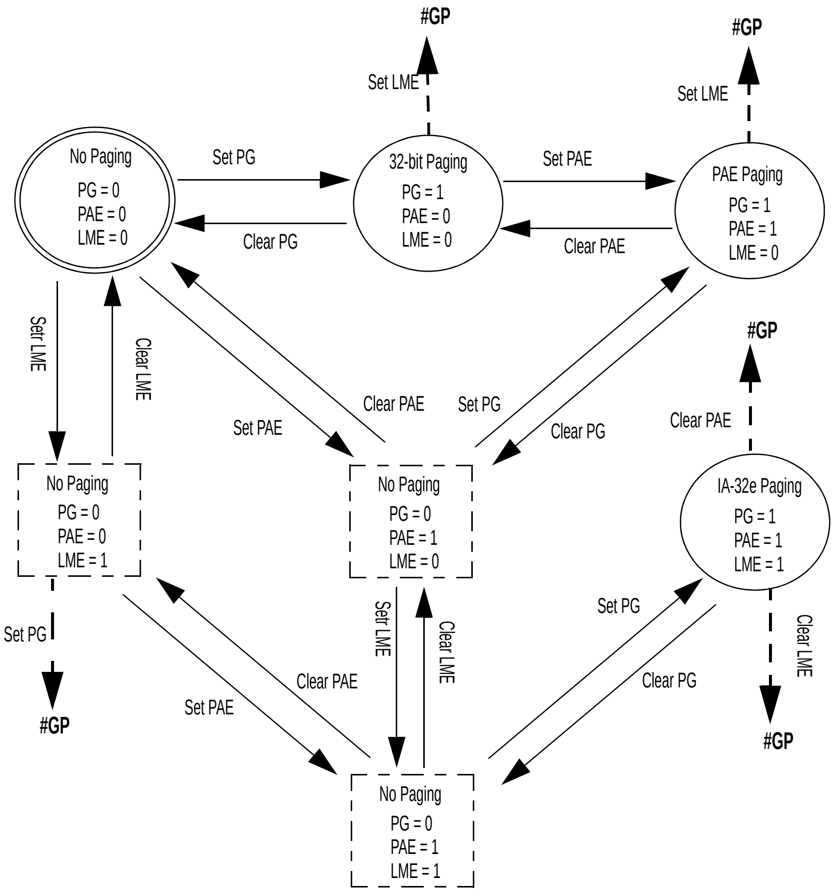


Figure 4-1. Enabling and Changing Paging Modes

- Regardless of the current paging mode, software can disable paging by clearing CR0.PG with MOV to CR0.[1]
- Software can make transitions between 32-bit paging and PAE paging by changing the value of CR4.PAE with MOV to CR4.
- Software cannot make transitions directly between IA-32e paging and either of the other two paging modes. It must first disable paging (by clearing CR0.PG with MOV to CR0), then set CR4.PAE and IA32_EFER.LME to the desired values (with MOV to CR4 and WRMSR), and then re-enable paging (by setting CR0.PG with MOV to CR0). As noted earlier, an attempt to clear either CR4.PAE or IA32_EFER.LME cause a general-protection exception (#GP(0)).
- VMX transitions allow transitions between paging modes that are not possible using MOV to CR or WRMSR. This is because VMX transitions can load CR0, CR4, and IA32_EFER in one operation. See Section 4.11.1.

1. If CR4.PCIDE = 1, an attempt to clear CR0.PG causes a general-protection exception (#GP); software should clear CR4.PCIDE before attempting to disable paging.

4.1.3 Paging-Mode Modifiers

Details of how each paging mode operates are determined by the following control bits:

- The WP flag in CR0 (bit 16).
- The PSE, PGE, and PCIDE flags in CR4 (bit 4, bit 7, and bit 17, respectively).
- The NXE flag in the IA32_EFER MSR (bit 11).

CR0.WP allows pages to be protected from supervisor-mode writes. If CR0.WP = 0, software operating with CPL < 3 (supervisor mode) can write to linear addresses with read-only access rights; if CR0.WP = 1, it cannot. (Software operating with CPL = 3 — user mode — cannot write to linear addresses with read-only access rights, regardless of the value of CR0.WP.) Section 4.6 explains how access rights are determined.

CR4.PSE enables 4-MByte pages for 32-bit paging. If CR4.PSE = 0, 32-bit paging can use only 4-KByte pages; if CR4.PSE = 1, 32-bit paging can use both 4-KByte pages and 4-MByte pages. See Section 4.3 for more information. (PAE paging and IA-32e paging can use multiple page sizes regardless of the value of CR4.PSE.)

CR4.PGE enables global pages. If CR4.PGE = 0, no translations are shared across address spaces; if CR4.PGE = 1, specified translations may be shared across address spaces. See Section 4.10.2.4 for more information.

CR4.PCIDE enables process-context identifiers (PCIDs) for IA-32e paging (CR4.PCIDE can be 1 only when IA-32e paging is in use). PCIDs allow a logical processor to cache information for multiple linear-address spaces. See Section 4.10.1 for more information.

IA32_EFER.NXE enables execute-disable access rights for PAE paging and IA-32e paging. If IA32_EFER.NXE = 0, software may fetch instructions from any linear address that paging allows the software to read; if IA32_EFER.NXE = 1, instructions fetches can be prevented from specified linear addresses (even if data reads from the addresses are allowed). Section 4.6 explains how access rights are determined. (32-bit paging always allows software to fetch instructions from any linear address that may be read; IA32_EFER.NXE has no effect with 32-bit paging. Software that wants to limit instruction fetches from readable pages must use either PAE paging or IA-32e paging.)

4.1.4 Enumeration of Paging Features by CPUID

Software can discover support for different paging features using the CPUID instruction:

- PSE: page-size extensions for 32-bit paging.
  If CPUID.01H:EDX.PSE [bit 3] = 1, CR4.PSE may be set to 1, enabling support for 4-MByte pages with 32-bit paging (see Section 4.3).
- PAE: physical-address extension.
  If CPUID.01H:EDX.PAE [bit 6] = 1, CR4.PAE may be set to 1, enabling PAE paging (this setting is also required for IA-32e paging).
- PGE: global-page support.
  If CPUID.01H:EDX.PGE [bit 13] = 1, CR4.PGE may be set to 1, enabling the global-page feature (see Section 4.10.2.4).
- PAT: page-attribute table.
  If CPUID.01H:EDX.PAT [bit 16] = 1, the 8-entry page-attribute table (PAT) is supported. When the PAT is supported, three bits in certain paging-structure entries select a memory type (used to determine type of caching used) from the PAT (see Section 4.9.2).
- PSE-36: 36-Bit page size extension.
  If CPUID.01H:EDX.PSE-36 [bit 17] = 1, the PSE-36 mechanism is supported,

indicating that translations using 4-MByte pages with 32-bit paging may produce physical addresses with more than 32 bits (see Section 4.3).

- PCID: process-context identifiers.
  If CPUID.01H:ECX.PCID [bit 17] = 1, CR4.PCIDE may be set to 1, enabling process-context identifiers (see Section 4.10.1).
- NX: execute disable.
  If CPUID.80000001H:EDX.NX [bit 20] = 1, IA32_EFER.NXE may be set to 1, allowing PAE paging and IA-32e paging to disable execute access to selected pages (see Section 4.6). (Processors that do not support CPUID function 80000001H do not allow IA32_EFER.NXE to be set to 1.)
- Page1GB: 1-GByte pages.
  If CPUID.80000001H:EDX.Page1GB [bit 26] = 1, 1-GByte pages are supported with IA-32e paging (see Section 4.5).
- LM: IA-32e mode support.
  If CPUID.80000001H:EDX.LM [bit 29] = 1, IA32_EFER.LME may be set to 1, enabling IA-32e paging. (Processors that do not support CPUID function 80000001H do not allow IA32_EFER.LME to be set to 1.)
- CPUID.80000008H:EAX[7:0] reports the physical-address width supported by the processor. (For processors that do not support CPUID function 80000008H, the width is generally 36 if CPUID.01H:EDX.PAE [bit 6] = 1 and 32 otherwise.) This width is referred to as MAXPHYADDR. MAXPHYADDR is at most 52.
- CPUID.80000008H:EAX[15:8] reports the linear-address width supported by the processor. Generally, this value is 48 if CPUID.80000001H:EDX.LM [bit 29] = 1 and 32 otherwise. (Processors that do not support CPUID function 80000008H, support a linear-address width of 32.)

4.2 HIERARCHICAL PAGING STRUCTURES: AN OVERVIEW

All three paging modes translate linear addresses use **hierarchical paging structures**. This section provides an overview of their operation. Section 4.3, Section 4.4, and Section 4.5 provide details for the three paging modes.

Every paging structure is 4096 Bytes in size and comprises a number of individual **entries**. With 32-bit paging, each entry is 32 bits (4 bytes); there are thus 1024 entries in each structure. With PAE paging and IA-32e paging, each entry is 64 bits (8 bytes); there are thus 512 entries in each structure. (PAE paging includes one exception, a paging structure that is 32 bytes in size, containing 4 64-bit entries.)

The processor uses the upper portion of a linear address to identify a series of paging-structure entries. The last of these entries identifies the physical address of the region to which the linear address translates (called the **page frame**). The lower portion of the linear address (called the **page offset**) identifies the specific address within that region to which the linear address translates.

Each paging-structure entry contains a physical address, which is either the address of another paging structure or the address of a page frame. In the first case, the entry is said to **reference** the other paging structure; in the latter, the entry is said to **map a page**.

The first paging structure used for any translation is located at the physical address in CR3. A linear address is translated using the following iterative procedure. A portion of the linear address (initially the uppermost bits) select an entry in a paging structure (initially the one located using CR3). If that entry references another paging structure, the process continues with that paging structure and with the portion of the linear address immediately below that just used. If instead the entry maps a page, the process completes: the physical address in the entry is that of the page frame and the remaining lower portion of the linear address is the page offset.

The following items give an example for each of the three paging modes (each example locates a 4-KByte page frame):

- With 32-bit paging, each paging structure comprises 1024 = 2^{10} entries. For this reason, the translation process uses 10 bits at a time from a 32-bit linear address. Bits 31:22 identify the first paging-structure entry and bits 21:12 identify a second. The latter identifies the page frame. Bits 11:0 of the linear address are the page offset within the 4-KByte page frame. (See Figure Figure 4-2 for an illustration.)
- With PAE paging, the first paging structure comprises only 4 = 2^2 entries. Translation thus begins by using bits 31:30 from a 32-bit linear address to identify the first paging-structure entry. Other paging structures comprise 512 =2^9 entries, so the process continues by using 9 bits at a time. Bits 29:21 identify a second paging-structure entry and bits 20:12 identify a third. This last identifies the page frame. (See Figure 4-5 for an illustration.)
- With IA-32e paging, each paging structure comprises 512 = 2^9 entries and translation uses 9 bits at a time from a 48-bit linear address. Bits 47:39 identify the first paging-structure entry, bits 38:30 identify a second, bits 29:21 a third, and bits 20:12 identify a fourth. Again, the last identifies the page frame. (See Figure 4-8 for an illustration.)

The translation process in each of the examples above completes by identifying a page frame. However, the paging structures may be configured so that translation terminates before doing so. This occurs if process encounters a paging-structure entry that is marked “not present” (because its P flag — bit 0 — is clear) or in which a reserved bit is set. In this case, there is no translation for the linear address; an access to that address causes a page-fault exception (see Section 4.7).

In the examples above, a paging-structure entry maps a page with 4-KByte page frame when only 12 bits remain in the linear address; entries identified earlier always reference other paging structures. That may not apply in other cases. The following items identify when an entry maps a page and when it references another paging structure:

- If more than 12 bits remain in the linear address, bit 7 (PS — page size) of the current paging-structure entry is consulted. If the bit is 0, the entry references another paging structure; if the bit is 1, the entry maps a page.
- If only 12 bits remain in the linear address, the current paging-structure entry always maps a page (bit 7 is used for other purposes).

If a paging-structure entry maps a page when more than 12 bits remain in the linear address, the entry identifies a page frame larger than 4 KBytes. For example, 32-bit paging uses the upper 10 bits of a linear address to locate the first paging-structure entry; 22 bits remain. If that entry maps a page, the page frame is 2^{22} Bytes = 4 MBytes. 32-bit paging supports 4-MByte pages if CR4.PSE = 1. PAE paging and IA-32e paging support 2-MByte pages (regardless of the value of CR4.PSE). IA-32e paging may support 1-GByte pages (see Section 4.1.4).

Paging structures are given different names based their uses in the translation process. Table Table 4-2 gives the names of the different paging structures. It also provides, for each structure, the source of the physical address used to locate it (CR3 or a different paging-structure entry); the bits in the linear address used to select an entry from the structure; and details of about whether and how such an entry can map a page.

Table 4-2 Paging Structures in the Different Paging Modes

Paging Structure	Entry Name	Paging Mode	Physical Address of Structure	Bits Selecting Entry	Page Mapping
PML4 table	PML4E	32-bit, PAE	N/A		
		IA-32e	CR3	47:39	N/A (PS must be 0)
Page-directory-pointer table	PDPTE	32-bit	N/A		
		PAE	CR3	31:30	N/A (PS must be 0)
		IA-32e	PML4E	38:30	1-GByte page if PS=1[1]
Page directory	PDE	32-bit	CR3	31:22	4-MByte page if PS=1[2]
		PAE, IA-32e	PDPTE	29:21	2-MByte page if PS=1
Page table	PTE	32-bit	PDE	21:12	4-KByte page
		PAE, IA-32e		20:12	4-KByte page

NOTES:

1. Not all processors allow the PS flag to be 1 in PDPTEs; see Section 4.1.4 for how to determine whether 1-GByte pages are supported.
2. 32-bit paging ignores the PS flag in a PDE (and uses the entry to reference a page table) unless CR4.PSE = 1. Not all processors allow CR4.PSE to be 1; see Section 4.1.4 for how to determine whether 4-MByte pages are supported with 32-bit paging.

4.3 32-BIT PAGING

A logical processor uses 32-bit paging if CR0.PG = 1 and CR4.PAE = 0. 32-bit paging translates 32-bit linear addresses to 40-bit physical addresses.[1] Although 40 bits corresponds to 1 TByte, linear addresses are limited to 32 bits; at most 4 GBytes of linear-address space may be accessed at any given time.

32-bit paging uses a hierarchy of paging structures to produce a translation for a linear address. CR3 is used to locate the first paging-structure, the page directory. Table 4-3 illustrates how CR3 is used with 32-bit paging.

32-bit paging may map linear addresses to either 4-KByte pages or 4-MByte pages. Figure Figure 4-2 illustrates the translation process when it uses a 4-KByte page; Figure Figure 4-3 covers the case of a 4-MByte page. The following items describe the 32-bit paging process in more detail as well has how the page size is determined:

1. Bits in the range 39:32 are 0 in any physical address used by 32-bit paging except those used to map 4-MByte pages. If the processor does not support the PSE-36 mechanism, this is true also for physical addresses used to map 4-MByte pages. If the processor does support the PSE-36 mechanism and MAXPHYADDR < 40, bits in the range 39:MAXPHYADDR are 0 in any physical address used to map a 4-MByte page. (The corresponding bits are reserved in PDEs.) See Section 4.1.4 for how to determine MAXPHYADDR and whether the PSE-36 mechanism is supported.

- A 4-KByte naturally aligned page directory is located at the physical address specified in bits 31:12 of CR3 (see Table 4-3). A page directory comprises 1024 32-bit entries (PDEs). A PDE is selected using the physical address defined as follows:
  - — Bits 39:32 are all 0.
  - — Bits 31:12 are from CR3.
  - — Bits 11:2 are bits 31:22 of the linear address.
  - — Bits 1:0 are 0.

Because a PDE is identified using bits 31:22 of the linear address, it controls access to a 4-Mbyte region of the linear-address space. Use of the PDE depends on CR.PSE and the PDE's PS flag (bit 7):

- If CR4.PSE = 1 and the PDE's PS flag is 1, the PDE maps a 4-MByte page (see Table Table 4-4). The final physical address is computed as follows:
  - — Bits 39:32 are bits 20:13 of the PDE.
  - — Bits 31:22 are bits 31:22 of the PDE.[1]
  - — Bits 21:0 are from the original linear address.
- If CR4.PSE = 0 or the PDE's PS flag is 0, a 4-KByte naturally aligned page table is located at the physical address specified in bits 31:12 of the PDE (see Table Table 4-5). A page table comprises 1024 32-bit entries (PTEs). A PTE is selected using the physical address defined as follows:
  - — Bits 39:32 are all 0.
  - — Bits 31:12 are from the PDE.
  - — Bits 11:2 are bits 21:12 of the linear address.
  - — Bits 1:0 are 0.
- Because a PTE is identified using bits 31:12 of the linear address, every PTE maps a 4-KByte page (see Table Table 4-6). The final physical address is computed as follows:
  - — Bits 39:32 are all 0.
  - — Bits 31:12 are from the PTE.
  - — Bits 11:0 are from the original linear address.

If a paging-structure entry's P flag (bit 0) is 0 or if the entry sets any reserved bit, the entry is used neither to reference another paging-structure entry nor to map a page. A reference using a linear address whose translation would use such a paging-structure entry causes a page-fault exception (see Section 4.7).

With 32-bit paging, there are reserved bits only if CR4.PSE = 1:

- If the P flag and the PS flag (bit 7) of a PDE are both 1, the bits reserved depend on MAXPHYADDR whether the PSE-36 mechanism is supported:[2]
  - — If the PSE-36 mechanism is not supported, bits 21:13 are reserved.
  - — If the PSE-36 mechanism is supported, bits 21:(M–19) are reserved, where M is the minimum of 40 and MAXPHYADDR.

1. The upper bits in the final physical address do not all come from corresponding positions in the PDE; the physical-address bits in the PDE are not all contiguous.
2. See Section 1.1.5 for how to determine MAXPHYADDR and whether the PSE-36 mechanism is supported.

- If the PAT is not supported:[1]
  - If the P flag of a PTE is 1, bit 7 is reserved.
  - If the P flag and the PS flag of a PDE are both 1, bit 12 is reserved.

(If CR4.PSE = 0, no bits are reserved with 32-bit paging.)

A reference using a linear address that is successfully translated to a physical address is performed only if allowed by the access rights of the translation; see Section 4.6.

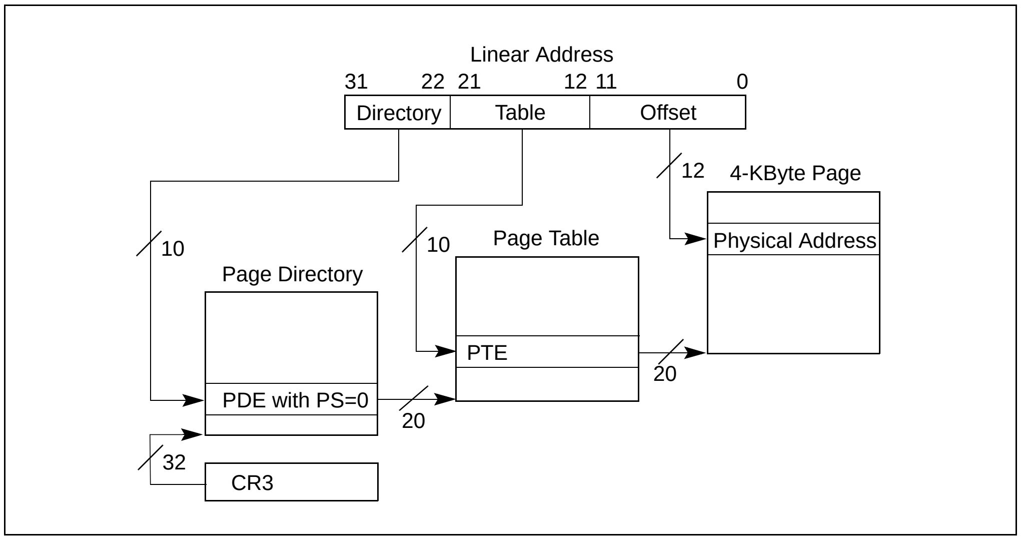


Figure 4-2 Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

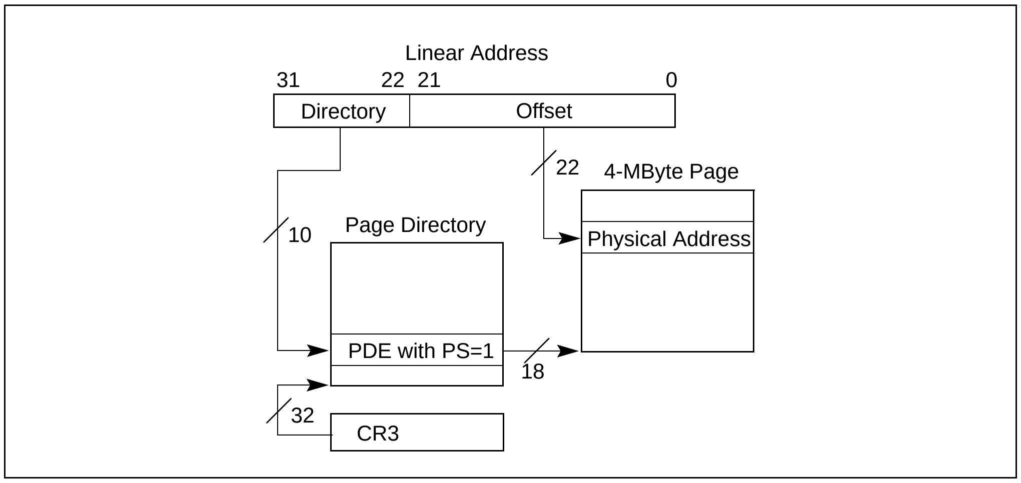


Figure 4-3 Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

1. See Section 4.1.4 for how to determine whether the PAT is supported.

Figure Figure 4-4 gives a summary of the formats of CR3 and the paging-structure entries with 32-bit paging. For the paging structure entries, it identifies separately the format of entries that map pages, those that reference other paging structures, and those that do neither because they are “not present”; bit 0 (P) and bit 7 (PS) are high-lighted because they determine how such an entry is used..

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 2MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

Figure 4-4 Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

NOTES:

1. CR3 has 64 bits on processors supporting the Intel-64 architecture. These bits are ignored with 32-bit paging.
2. This example illustrates a processor in which MAXPHYADDR is 36. If this value is larger or smaller, the number of bits reserved in positions 20:13 of a PDE mapping a 4-MByte will change.

Table 4-3 Use of CR3 with 32-Bit Paging

Bit Position(s)	Contents
2:0	Ignored
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page directory during linear-address translation (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page directory during linear-address translation (see Section 4.9)
11:5	Ignored
31:12	Physical address of the 4-KByte aligned page directory used for linear-address translation
63:32	Ignored (these bits exist only on processors supporting the Intel-64 architecture)

Table 4-4 Format of a 32-Bit Page-Directory Entry that Maps a 4-MByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-MByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte page referenced by this entry (depends on CPL and CR0.WP; see Section 4.6)
2 (U/S)	User/supervisor; if 0, accesses with CPL=3 are not allowed to the 4-MByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-MByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-MByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-MByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-MByte page referenced by this entry (see Section 4.8)
7 (PS)	Page size; must be 1 (otherwise, this entry references a page table; see Table Table 4-5)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored

Table 4-4 Format of a 32-Bit Page-Directory Entry that Maps a 4-MByte Page

Bit Position(s)	Contents
12 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-MByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)[1]
(M–20):13	Bits (M–1):32 of physical address of the 4-MByte page referenced by this entry[2]
21:(M–19)	Reserved (must be 0)
31:22	Bits 31:22 of physical address of the 4-MByte page referenced by this entry

NOTES:

1. See Section 4.1.4 for how to determine whether the PAT is supported.
2. If the PSE-36 mechanism is not supported, M is 32, and this row does not apply. If the PSE-36 mechanism is supported, M is the minimum of 40 and MAXPHYADDR (this row does not apply if MAXPHYADDR = 32). See Section 4.1.4 for how to determine MAXPHYADDR and whether the PSE-36 mechanism is supported.

Table 4-5 Format of a 32-Bit Page-Directory Entry that References a Page Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (depends on CPL and CR0.WP; see Section 4.6)
2 (U/S)	User/supervisor; if 0, accesses with CPL=3 are not allowed to the 4-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	If CR4.PSE = 1, must be 0 (otherwise, this entry maps a 4-MByte page; see Table Table 4-4); otherwise, ignored
11:8	Ignored
31:12	Physical address of 4-KByte aligned page table referenced by this entry

Table 4-6 Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (depends on CPL and CR0.WP; see Section 4.6)
2 (U/S)	User/supervisor; if 0, accesses with CPL=3 are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)[1]
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
31:12	Physical address of the 4-KByte page referenced by this entry

NOTES:
1. See Section 4.1.4 for how to determine whether the PAT is supported.

4.4 PAE PAGING

A logical processor uses PAE paging if CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LME = 0. PAE paging translates 32-bit linear addresses to 52-bit physical addresses.[1] Although 52 bits corresponds to 4 PBytes, linear addresses are limited to 32 bits; at most 4 GBytes of linear-address space may be accessed at any given time.

With PAE paging, a logical processor maintains a set of four (4) PDPTE registers, which are loaded from an address in CR3. Linear address are translated using 4 hierarchies of in-memory paging structures, each located using one of the PDPTE registers. (This is different from the other paging modes, in which there is one hierarchy referenced by CR3.)

1. If MAXPHYADDR < 52, bits in the range 51:MAXPHYADDR will be 0 in any physical address used by PAE paging. (The corresponding bits are reserved in the paging-structure entries.) See Section 4.1.4 for how to determine MAXPHYADDR.

Section 4.4.1 discusses the PDPTE registers. Section 4.4.2 describes linear-address translation with PAE paging.

4.4.1 PDPTE Registers

When PAE paging is used, CR3 references the base of a 32-Byte **page-directory-pointer table**. Table Table 4-7 illustrates how CR3 is used with PAE paging.

Table 4-7 Use of CR3 with PAE Paging

Bit Position(s)	Contents
4:0	Ignored
31:5	Physical address of the 32-Byte aligned page-directory-pointer table used for linear-address translation
63:32	Ignored (these bits exist only on processors supporting the Intel-64 architecture)

The page-directory-pointer-table comprises four (4) 64-bit entries called PDPTEs. Each PDPTE controls access to a 1-GByte region of the linear-address space. Corresponding to the PDPTEs, the logical processor maintains a set of four (4) internal, non-architectural PDPTE registers, called PDPTE0, PDPTE1, PDPTE2, and PDPTE3.

The logical processor loads these registers from the PDPTEs in memory as part of certain executions the MOV to CR instruction:

- If PAE paging would be in use following an execution of MOV to CR0 or MOV to CR4 (see Section 4.1.1) and the instruction is modifying any of CR0.CD, CR0.NW, CR0.PG, CR4.PAE, CR4.PGE, or CR4.PSE; then the PDPTEs are loaded from the address in CR3.
- If MOV to CR3 is executed while the logical processor is using PAE paging, the PDPTEs are loaded from the address being loaded into CR3.
- If PAE paging is in use and a task switch changes the value of CR3, the PDPTEs are loaded from the address in the new CR3 value.
- Certain VMX transitions load the PDPTE registers. See Section 4.11.1.

Unless the caches are disabled, the processor uses the WB memory type to load the PDPTEs from memory.[1]

Table 4-8 gives the format of a PDPTE. If any of the PDPTEs sets both the P flag (bit 0) and any reserved bit, the MOV to CR instruction causes a general-protection exception (#GP(0)) and the PDPTEs are not loaded.[2] As show in Table 4-8, bits 2:1, 8:5, and 63:MAXPHYADDR are reserved in the PDPTEs.

Table 4-8 Format of a PAE Page-Directory-Pointer-Table Entry (PDPTE)

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page directory

1. Older IA-32 processors used the UC memory type when loading the PDPTEs. This behavior is model-specific and not architectural.

Table 4-8 Format of a PAE Page-Directory-Pointer-Table Entry (PDPTE) (Continued)

Bit Position(s)	Contents
2:1	Reserved (must be 0)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9)
8:5	Reserved (must be 0)
11:9	Ignored
(M–1):12	Physical address of 4-KByte aligned page directory referenced by this entry[1]
63:M	Reserved (must be 0)

NOTES:
1. M is an abbreviation for MAXPHYADDR, which is at most 52; see Section 4.1.4.

4.4.2 Linear-Address Translation with PAE Paging

PAE paging may map linear addresses to either 4-KByte pages or 2-MByte pages. Figure 4-5 illustrates the translation process when it produces a 4-KByte page; Figure 4-6 covers the case of a 2-MByte page. The following items describe the PAE paging process in more detail as well has how the page size is determined:

- Bits 31:30 of the linear address select a PDPTE register (see Section 4.4.1); this is PDPTE*i*, where *i* is the value of bits 31:30.[1] Because a PDPTE register is identified using bits 31:30 of the linear address, it controls access to a 1-GByte region of the linear-address space. If the P flag (bit 0) of PDPTE*i* is 0, the processor ignores bits 63:1, and there is no mapping for the 1-GByte region controlled by PDPTE*i*. A reference using a linear address in this region causes a page-fault exception (see Section 4.7).
- If the P flag of PDPTE*i* is 1, 4-KByte naturally aligned page directory is located at the physical address specified in bits 51:12 of PDPTE*i* (see Table 4-8 in Section 4.4.1) A page directory comprises 512 64-bit entries (PDEs). A PDE is selected using the physical address defined as follows:
  - — Bits 51:12 are from PDPTE*i*.
  - — Bits 11:3 are bits 29:21 of the linear address.
  - — Bits 2:0 are 0.

Because a PDE is identified using bits 31:21 of the linear address, it controls access to a 2-Mbyte region of the linear-address space. Use of the PDE depends on its PS flag (bit 7):

2. On some processors, reserved bits are checked even in PDPTEs in which the P flag (bit 0) is 0.
1. With PAE paging, the processor does not use CR3 when translating a linear address (as it does the other paging modes). It does not access the PDPTEs in the page-directory-pointer table during linear-address translation.

- If the PDE’s PS flag is 1, the PDE maps a 2-MByte page (see Table 4-9). The final physical address is computed as follows:
  - — Bits 51:21 are from the PDE.
  - — Bits 20:0 are from the original linear address.
- If the PDE’s PS flag is 0, a 4-KByte naturally aligned page table is located at the physical address specified in bits 51:12 of the PDE (see Table 4-10). A page directory comprises 512 64-bit entries (PTEs). A PTE is selected using the physical address defined as follows:
  - — Bits 51:12 are from the PDE.
  - — Bits 11:3 are bits 20:12 of the linear address.
  - — Bits 2:0 are 0.
- Because a PTE is identified using bits 31:12 of the linear address, every PTE maps a 4-KByte page (see Table 4-11). The final physical address is computed as follows:
  - — Bits 51:12 are from the PTE.
  - — Bits 11:0 are from the original linear address.

If the P flag (bit 0) of a PDE or a PTE is 0 or if a PDE or a PTE sets any reserved bit, the entry is used neither to reference another paging-structure entry nor to map a page. A reference using a linear address whose translation would use such a paging-structure entry causes a page-fault exception (see Section 4.7).

The following bits are reserved with PAE paging:

- If the P flag (bit 0) of a PDE or a PTE is 1, bits 62:MAXPHYADDR are reserved.
- If the P flag and the PS flag (bit 7) of a PDE are both 1, bits 20:13 are reserved.
- If IA32_EFER.NXE = 0 and the P flag of a PDE or a PTE is 1, the XD flag (bit 63) is reserved.
- If the PAT is not supported:[1]
  - — If the P flag of a PTE is 1, bit 7 is reserved.
  - — If the P flag and the PS flag of a PDE are both 1, bit 12 is reserved.

A reference using a linear address that is successfully translated to a physical address is performed only if allowed by the access rights of the translation; see Section 4.6.

...

1. See Section 4.1.4 for how to determine whether the PAT is supported.

4.5 IA-32E PAGING

A logical processor uses IA-32e paging if CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LME = 1. With IA-32e paging, linear address are translated using a hierarchy of in-memory paging structures located using the contents of CR3. IA-32e paging translates 48-bit linear addresses to 52-bit physical addresses.[1] Although 52 bits corresponds to 4 PBytes, linear addresses are limited to 48 bits; at most 256 TBytes of linear-address space may be accessed at any given time.

IA-32e paging uses a hierarchy of paging structures to produce a translation for a linear address. CR3 is used to locate the first paging-structure, the PML4 table. Use of CR3 with IA-32e paging depends on whether process-context identifiers (PCIDs) have been enabled by setting CR4.PCIDE:

- Table Table 4-12 illustrates how CR3 is used with IA-32e paging if CR4.PCIDE = 0.

Table 4-12 Use of CR3 with IA-32e Paging and CR4.PCIDE = 0

Bit Position(s)	Contents
2:0	Ignored
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the PML4 table during linear-address translation (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the PML4 table during linear-address translation (see Section 4.9.2)
11:5	Ignored
M–1:12	Physical address of the 4-KByte aligned PML4 table used for linear-address translation[1]
63:M	Reserved (must be 0)

NOTES:
1. M is an abbreviation for MAXPHYADDR, which is at most 52; see Section 4.1.4.

- Table Table 4-13 illustrates how CR3 is used with IA-32e paging if CR4.PCIDE = 1.

Table 4-13 Use of CR3 with IA-32e Paging and CR4.PCIDE = 1

Bit Position(s)	Contents
11:0	PCID (see Section 4.10.1)[1]
M–1:12	Physical address of the 4-KByte aligned PML4 table used for linear-address translation[2]
63:M	Reserved (must be 0)[3]

1. If MAXPHYADDR < 52, bits in the range 51:MAXPHYADDR will be 0 in any physical address used by IA-32e paging. (The corresponding bits are reserved in the paging-structure entries.) See Section 4.1.4 for how to determine MAXPHYADDR.

NOTES:

1. Section 4.9.2 explains how the processor determines the memory type used to access the PML4 table during linear-address translation with CR4.PCIDE = 1.
2. M is an abbreviation for MAXPHYADDR, which is at most 52; see Section 4.1.4.
3. See Section 4.10.4.1 for use of bit 63 of the source operand of the MOV to CR3 instruction.

After software modifies the value of CR4.PCIDE, the logical processor immediately begins using CR3 as specified for the new value. For example, if software changes CR4.PCIDE from 1 to 0, the current PCID immediately changes from CR3[11:0] to 000H (see also Section 4.10.4.1). In addition, the logical processor subsequently determines the memory type used to access the PML4 table using CR3.PWT and CR3.PCD, which had been bits 4:3 of the PCID.

IA-32e paging may map linear addresses to 4-KByte pages, 2-MByte pages, or 1-GByte pages.[1] Figure 4-8 illustrates the translation process when it produces a 4-KByte page; Figure 4-9 covers the case of a 2-MByte page, and Figure 4-10 the case of a 1-GByte page.

...

8. 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.

...

6.13 ERROR CODE

When an exception condition is related to a specific segment selector or IDT vector, the processor pushes an error code onto the stack of the exception handler (whether it is a procedure or task). The error code has the format shown in Figure Figure 6-6.. The error code resembles a segment selector; however, instead of a TI flag and RPL field, the error code contains 3 flags:

EXT **External event (bit 0)** — When set, indicates that the exception occurred during delivery of an event external to the program, such as an interrupt or an earlier exception.

IDT **Descriptor location (bit 1)** — When set, indicates that the index portion of the error code refers to a gate descriptor in the IDT; when clear, indicates that the index refers to a descriptor in the GDT or the current LDT.

TI **GDT/LDT (bit 2)** — Only used when the IDT flag is clear. When set, the TI flag indicates that the index portion of the error code refers to a segment or gate descriptor in the LDT; when clear, it indicates that the index refers to a descriptor in the current GDT.

1. Not all processors support 1-GByte pages; see Section 4.1.4.

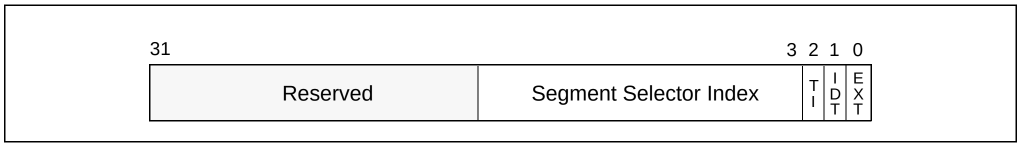


Figure 6-6. Error Code

The segment selector index field provides an index into the IDT, GDT, or current LDT to the segment or gate selector being referenced by the error code. In some cases the error code is null (all bits are clear except possibly EXT). A null error code indicates that the error was not caused by a reference to a specific segment or that a null segment descriptor was referenced in an operation.

The format of the error code is different for page-fault exceptions (#PF). See the "Interrupt 14—Page-Fault Exception (#PF)" section in this chapter.

The error code is pushed on the stack as a doubleword or word (depending on the default interrupt, trap, or task gate size). To keep the stack aligned for doubleword pushes, the upper half of the error code is reserved. Note that the error code is not popped when the IRET instruction is executed to return from an exception handler, so the handler must remove the error code before executing a return.

Error codes are not pushed on the stack for exceptions that are generated externally (with the INTR or LINT[1:0] pins) or the INT *n* instruction, even if an error code is normally produced for those exceptions.

...

Interrupt 17—Alignment Check Exception (#AC)

...

Exception Error Code

Yes. The error code is null; all bits are clear except possibly bit 0 — EXT; see Section 6.13. EXT is set if the #AC is recognized during delivery of an event other than a software interrupt (see "INT n/INTO/INT 3—Call to Interrupt Procedure" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A)*.

...

9. 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.

...

8.2.3.2 Neither Loads Nor Stores Are Reordered with Like Operations

The Intel-64 memory-ordering model allows neither loads nor stores to be reordered with the same kind of operation. That is, it ensures that loads are seen in program order and that stores are seen in program order. This is illustrated by the following example:

Example 8-1. Stores Are Not Reordered with Other Stores

Processor 0	Processor 1
mov [_x], 1 mov [_y], 1	mov r1, [_y] mov r2, [_x]
Initially x = y = 0 r1 = 1 and r2 = 0 is not allowed	

The disallowed return values could be exhibited only if processor 0's two stores are reordered (with the two loads occurring between them) or if processor 1's two loads are reordered (with the two stores occurring between them).

If r1 = 1, the store to y occurs before the load from y. Because the Intel-64 memory-ordering model does not allow stores to be reordered, the earlier store to x occurs before the load from y. Because the Intel-64 memory-ordering model does not allow loads to be reordered, the store to x also occurs before the later load from x. This r2 = 1.

8.2.3.3 Stores Are Not Reordered With Earlier Loads

The Intel-64 memory-ordering model ensures that a store by a processor may not occur before a previous load by the same processor. This is illustrated by the following example:

Example 8-2. Stores Are Not Reordered with Older Loads

Processor 0	Processor 1
mov r1, [_x] mov [_y], 1	mov r2, [_y] mov [_x], 1
Initially x = y = 0 r1 = 1 and r2 = 1 is not allowed	

Assume r1 = 1.

- Because r1 = 1, processor 1's store to x occurs before processor 0's load from x.
- Because the Intel-64 memory-ordering model prevents each store from being reordered with the earlier load by the same processor, processor 1's load from y occurs before its store to x.
- Similarly, processor 0's load from x occurs before its store to y.
- Thus, processor 1's load from y occurs before processor 0's store to y, implying r2 = 0.

8.2.3.4 Loads May Be Reordered with Earlier Stores to Different Locations

The Intel-64 memory-ordering model allows a load to be reordered with an earlier store to a different location. However, loads are not reordered with stores to the same location.

The fact that a load may be reordered with an earlier store to a different location is illustrated by the following example:

Example 8-3. Loads May be Reordered with Older Stores

Processor 0	Processor 1
mov [_x], 1 mov r1, [_y]	mov [_y], 1 mov r2, [_x]
Initially x = y = 0 r1 = 0 and r2 = 0 is allowed	

At each processor, the load and the store are to different locations and hence may be reordered. Any interleaving of the operations is thus allowed. One such interleaving has the two loads occurring before the two stores. This would result in each load returning value 0.

The fact that a load may not be reordered with an earlier store to the same location is illustrated by the following example:

Example 8-4. Loads Are not Reordered with Older Stores to the Same Location

Processor 0
mov [_x], 1 mov r1, [_x]
Initially x = 0 r1 = 0 is not allowed

The Intel-64 memory-ordering model does not allow the load to be reordered with the earlier store because the accesses are to the same location. Therefore, r1 = 1 must hold.

8.2.3.5 Intra-Processor Forwarding Is Allowed

The memory-ordering model allows concurrent stores by two processors to be seen in different orders by those two processors; specifically, each processor may perceive its own store occurring before that of the other. This is illustrated by the following example:

Example 8-5. Intra-Processor Forwarding is Allowed

Processor 0	Processor 1
mov [_x], 1 mov r1, [_x] mov r2, [_y]	mov [_y], 1 mov r3, [_y] mov r4, [_x]
Initially x = y = 0 r2 = 0 and r4 = 0 is allowed	

The memory-ordering model imposes no constraints on the order in which the two stores appear to execute by the two processors. This fact allows processor 0 to see its store before seeing processor 1's, while processor 1 sees its store before seeing processor 0's. (Each processor is self consistent.) This allows r2 = 0 and r4 = 0.

In practice, the reordering in this example can arise as a result of store-buffer forwarding. While a store is temporarily held in a processor's store buffer, it can satisfy the processor's own loads but is not visible to (and cannot satisfy) loads by other processors.

8.2.3.6 Stores Are Transitively Visible

The memory-ordering model ensures transitive visibility of stores; stores that are causally related appear to all processors to occur in an order consistent with the causality relation. This is illustrated by the following example:

Example 8-6. Stores Are Transitively Visible

Processor 0	Processor 1	Processor 2
mov [_x], 1	mov r1, [_x] mov [_y], 1	 mov r2, [_y] mov r3, [_x]
Initially x = y = 0 r1 = 1, r2 = 1, r3 = 0 is not allowed		

Assume that r1 = 1 and r2 = 1.

- Because r1 = 1, processor 0's store occurs before processor 1's load.
- Because the memory-ordering model prevents a store from being reordered with an earlier load (see Section 8.2.3.3), processor 1's load occurs before its store. Thus, processor 0's store causally precedes processor 1's store.
- Because processor 0's store causally precedes processor 1's store, the memory-ordering model ensures that processor 0's store appears to occur before processor 1's store from the point of view of all processors.
- Because r2 = 1, processor 1's store occurs before processor 2's load.
- Because the Intel-64 memory-ordering model prevents loads from being reordered (see Section 8.2.3.2), processor 2's load occur in order.
- The above items imply that processor 0's store to x occurs before processor 2's load from x. This implies that r3 = 1.

8.2.3.7 Stores Are Seen in a Consistent Order by Other Processors

As noted in Section 8.2.3.5, the memory-ordering model allows stores by two processors to be seen in different orders by those two processors. However, any two stores must appear to execute in the same order to all processors other than those performing the stores. This is illustrated by the following example:

Example 8-7. Stores Are Seen in a Consistent Order by Other Processors

Processor 0	Processor 1	Processor 2	Processor 3
mov [_x], 1	mov [_y], 1	mov r1, [_x] mov r2, [_y]	mov r3, [_y] mov r4, [_x]
Initially x = y =0 r1 = 1, r2 = 0, r3 = 1, r4 = 0 is not allowed			

By the principles discussed in Section 8.2.3.2,

- processor 2's first and second load cannot be reordered,
- processor 3's first and second load cannot be reordered.
- If r1 = 1 and r2 = 0, processor 0's store appears to precede processor 1's store with respect to processor 2.

- Similarly, r3 = 1 and r4 = 0 imply that processor 1's store appears to precede processor 0's store with respect to processor 1.

Because the memory-ordering model ensures that any two stores appear to execute in the same order to all processors (other than those performing the stores), this set of return values is not allowed

8.2.3.8 Locked Instructions Have a Total Order

The memory-ordering model ensures that all processors agree on a single execution order of all locked instructions, including those that are larger than 8 bytes or are not naturally aligned. This is illustrated by the following example:

Example 8-8. Locked Instructions Have a Total Order

Processor 0	Processor 1	Processor 2	Processor 3
xchg [_x], r1	xchg [_y], r2		
		mov r3, [_x]	mov r5, [_y]
		mov r4, [_y]	mov r6, [_x]
Initially r1 = r2 = 1, x = y = 0			
r3 = 1, r4 = 0, r5 = 1, r6 = 0 is not allowed			

Processor 2 and processor 3 must agree on the order of the two executions of XCHG. Without loss of generality, suppose that processor 0's XCHG occurs first.

- If r5 = 1, processor 1's XCHG into y occurs before processor 3's load from y.
- Because the Intel-64 memory-ordering model prevents loads from being reordered (see Section 8.2.3.2), processor 3's loads occur in order and, therefore, processor 1's XCHG occurs before processor 3's load from x.
- Since processor 0's XCHG into x occurs before processor 1's XCHG (by assumption), it occurs before processor 3's load from x. Thus, r6 = 1.

A similar argument (referring instead to processor 2's loads) applies if processor 1's XCHG occurs before processor 0's XCHG.

8.2.3.9 Loads and Stores Are Not Reordered with Locked Instructions

The memory-ordering model prevents loads and stores from being reordered with locked instructions that execute earlier or later. The examples in this section illustrate only cases in which a locked instruction is executed before a load or a store. The reader should note that reordering is prevented also if the locked instruction is executed after a load or a store.

The first example illustrates that loads may not be reordered with earlier locked instructions:

Example 8-9. Loads Are not Reordered with Locks

Processor 0	Processor 1
xchg [_x], r1	xchg [_y], r3
mov r2, [_y]	mov r4, [_x]
Initially x = y = 0, r1 = r3 = 1	
r2 = 0 and r4 = 0 is not allowed	

As explained in Section 8.2.3.8, there is a total order of the executions of locked instructions. Without loss of generality, suppose that processor 0's XCHG occurs first.

Because the Intel-64 memory-ordering model prevents processor 1's load from being reordered with its earlier XCHG, processor 0's XCHG occurs before processor 1's load. This implies r4 = 1.

A similar argument (referring instead to processor 2's accesses) applies if processor 1's XCHG occurs before processor 0's XCHG.

The second example illustrates that a store may not be reordered with an earlier locked instruction:

Example 8-10. Stores Are not Reordered with Locks

Processor 0	Processor 1
xchg [_x], r1	mov r2, [_y]
mov [_y], 1	mov r3, [_x]
Initially x = y = 0, r1 = 1	
r2 = 1 and r3 = 0 is not allowed	

Assume r2 = 1.

- Because r2 = 1, processor 0's store to y occurs before processor 1's load from y.
- Because the memory-ordering model prevents a store from being reordered with an earlier locked instruction, processor 0's XCHG into x occurs before its store to y. Thus, processor 0's XCHG into x occurs before processor 1's load from y.
- Because the memory-ordering model prevents loads from being reordered (see Section 8.2.3.2), processor 1's loads occur in order and, therefore, processor 1's XCHG into x occurs before processor 1's load from x. Thus, r3 = 1.

...

8.2.4.2 Examples Illustrating Memory-Ordering Principles for String Operations

The following examples uses the same notation and convention as described in Section 8.2.3.1.

In Example Example 8-11., processor 0 does one round of (128 iterations) doubleword string store operation via rep:stosd, writing the value 1 (value in EAX) into a block of 512 bytes from location _x (kept in ES:EDI) in ascending order. Since each operation stores a doubleword (4 bytes), the operation is repeated 128 times (value in ECX). The block of memory initially contained 0. Processor 1 is reading two memory locations that are part of the memory block being updated by processor 0, i.e, reading locations in the range _x to (_x+511).

Example 8-11. Stores Within a String Operation May be Reordered

Processor 0	Processor 1
rep:stosd [_x]	mov r1, [_z]
	mov r2, [_y]

Example 8-11. Stores Within a String Operation May be Reordered

Processor 0	Processor 1
Initially on processor 0: EAX = 1, ECX=128, ES:EDI =_x	
Initially [_x] to 511[_x]= 0, _x <= _y < _z < _x+512	
r1 = 1 and r2 = 0 is allowed	

It is possible for processor 1 to perceive that the repeated string stores in processor 0 are happening out of order. Assume that fast string operations are enabled on processor 0.

In Example Example 8-12., processor 0 does two separate rounds of rep stosd operation of 128 doubleword stores, writing the value 1 (value in EAX) into the first block of 512 bytes from location _x (kept in ES:EDI) in ascending order. It then writes 1 into a second block of memory from (_x+512) to (_x+1023). All of the memory locations initially contain 0. The block of memory initially contained 0. Processor 1 performs two load operations from the two blocks of memory.

Example 8-12. Stores Across String Operations Are not Reordered

Processor 0	Processor 1
rep:stosd [_x]	
	mov r1, [_z]
mov ecx, $128	
	mov r2, [_y]
rep:stosd 512[_x]	
Initially on processor 0: EAX = 1, ECX=128, ES:EDI =_x	
Initially [_x] to 1023[_x]= 0, _x <= _y < _x+512 < _z < _x+1024	
r1 = 1 and r2 = 0 is not allowed	

It is not possible in the above example for processor 1 to perceive any of the stores from the later string operation (to the second 512 block) in processor 0 before seeing the stores from the earlier string operation to the first 512 block.

The above example assumes that writes to the second block (_x+512 to _x+1023) does not get executed while processor 0's string operation to the first block has been interrupted. If the string operation to the first block by processor 0 is interrupted, and a write to the second memory block is executed by the interrupt handler, then that change in the second memory block will be visible before the string operation to the first memory block resumes.

In Example Example 8-13., processor 0 does one round of (128 iterations) doubleword string store operation via rep:stosd, writing the value 1 (value in EAX) into a block of 512 bytes from location _x (kept in ES:EDI) in ascending order. It then writes to a second memory location outside the memory block of the previous string operation. Processor 1 performs two read operations, the first read is from an address outside the 512-byte block but to be updated by processor 0, the second ready is from inside the block of memory of string operation.

Example 8-13. String Operations Are not Reordered with later Stores

Processor 0	Processor 1
rep:stosd [_x] mov [_z], $1	mov r1, [_z] mov r2, [_y]
Initially on processor 0: EAX = 1, ECX=128, ES:EDI =_x Initially [_y] = [_z] = 0, [_x] to 511[_x]= 0, _x <= _y < _x+512, _z is a separate memory location r1 = 1 and r2 = 0 is not allowed	

Processor 1 cannot perceive the later store by processor 0 until it sees all the stores from the string operation. Example Example 8-13. assumes that processor 0's store to [_z] is not executed while the string operation has been interrupted. If the string operation is interrupted and the store to [_z] by processor 0 is executed by the interrupt handler, then changes to [_z] will become visible before the string operation resumes.

Example Example 8-14. illustrates the visibility principle when a string operation is interrupted.

Example 8-14. Interrupted String Operation

Processor 0	Processor 1
rep:stosd [_x] // interrupted before es:edi reach _y mov [_z], $1 // interrupt handler	mov r1, [_z] mov r2, [_y]
Initially on processor 0: EAX = 1, ECX=128, ES:EDI =_x Initially [_y] = [_z] = 0, [_x] to 511[_x]= 0, _x <= _y < _x+512, _z is a separate memory location r1 = 1 and r2 = 0 is allowed	

In Example Example 8-14., processor 0 started a string operation to write to a memory block of 512 bytes starting at address _x. Processor 0 got interrupted after k iterations of store operations. The address _y has not yet been updated by processor 0 when processor 0 got interrupted. The interrupt handler that took control on processor 0 writes to the address _z. Processor 1 may see the store to _z from the interrupt handler, before seeing the remaining stores to the 512-byte memory block that are executed when the string operation resumes.

Example Example 8-15. illustrates the ordering of string operations with earlier stores. No store from a string operation can be visible before all prior stores are visible.

Example 8-15. String Operations Are not Reordered with Earlier Stores

Processor 0	Processor 1
mov [_z], $1 rep:stosd [_x]	mov r1, [_y] mov r2, [_z]
Initially on processor 0: EAX = 1, ECX=128, ES:EDI =_x Initially [_y] = [_z] = 0, [_x] to 511[_x]= 0, _x <= _y < _x+512, _z is a separate memory location r1 = 1 and r2 = 0 is not allowed	

...

8.3 SERIALIZING INSTRUCTIONS

The Intel 64 and IA-32 architectures define several **serializing instructions**. These instructions force the processor to complete all modifications to flags, registers, and memory by previous instructions and to drain all buffered writes to memory before the next instruction is fetched and executed. For example, when a MOV to control register instruction is used to load a new value into control register CR0 to enable protected mode, the processor must perform a serializing operation before it enters protected mode. This serializing operation ensures that all operations that were started while the processor was in real-address mode are completed before the switch to protected mode is made.

The concept of serializing instructions was introduced into the IA-32 architecture with the Pentium processor to support parallel instruction execution. Serializing instructions have no meaning for the Intel486 and earlier processors that do not implement parallel instruction execution.

It is important to note that executing of serializing instructions on P6 and more recent processor families constrain speculative execution because the results of speculatively executed instructions are discarded. The following instructions are serializing instructions:

- **Privileged serializing instructions** — INVD, INVEPT, INVLPG, INVVPID, LGDT, LIDT, LLDT, LTR, MOV (to control register, with the exception of MOV CR8[1]), MOV (to debug register), WBINVD, and WRMSR[2].
- **Non-privileged serializing instructions** — CPUID, IRET, and RSM.

When the processor serializes instruction execution, it ensures that all pending memory transactions are completed (including writes stored in its store buffer) before it executes the next _instruction. Nothing can pass a serializing instruction and a serializing instruction cannot pass any other instruction (read, write, instruction fetch, or I/O). For example, CPUID can be executed at any privilege level to serialize instruction execution with no effect on program flow, except that the EAX, EBX, ECX, and EDX registers are modified.

...

8.7.11 MICROCODE UPDATE Resources

In an Intel processor supporting Intel Hyper-Threading Technology, the microcode update facilities are shared between the logical processors; either logical processor can initiate an update. Each logical processor has its own BIOS signature MSR (IA32_BIOS_SIGN_ID at MSR address 8BH). When a logical processor performs an update for the physical processor, the IA32_BIOS_SIGN_ID MSRs for resident logical processors are updated with identical information. If logical processors initiate an update simultaneously, the processor core provides the necessary synchronization needed to ensure that only one update is performed at a time.

NOTE

Some processors (prior to the introduction of Intel 64 Architecture and based on Intel NetBurst microarchitecture) do not support simultaneous

1. MOV CR8 is not defined architecturally as a serializing instruction.
2. WRMSR to the IA32_TSC_DEADLINE MSR (MSR index 6E0H) and the X2APIC MSRs (MSR indices 802H to 83FH) are not serializing..

loading of microcode update to the sibling logical processors in the same core. All other processors support logical processors initiating an update simultaneously. Intel recommends a common approach that the microcode loader use the sequential technique described in Section 9.11.6.3.

...

8.8.5 MICROCODE UPDATE Resources

Microcode update facilities are shared between two logical processors sharing a processor core if the physical package supports Intel Hyper-Threading Technology. They are not shared between logical processors in different cores or different physical packages. Either logical processor that has access to the microcode update facility can initiate an update.

Each logical processor has its own BIOS signature MSR (IA32_BIOS_SIGN_ID at MSR address 8BH). When a logical processor performs an update for the physical processor, the IA32_BIOS_SIGN_ID MSRs for resident logical processors are updated with identical information.

NOTE

Some processors (prior to the introduction of Intel 64 Architecture and based on Intel NetBurst microarchitecture) do not support simultaneous loading of microcode update to the sibling logical processors in the same core. All other processors support logical processors initiating an update simultaneously. Intel recommends a common approach that the microcode loader use the sequential technique described in Section 9.11.6.3.

...

Example 8-22. Compute the Number of Packages, Cores, and Processor Relationships in a MP System

a) Assemble lists of PACKAGE_ID, CORE_ID, and SMT_ID of each enabled logical processors

```
//The BIOS and/or OS may limit the number of logical processors available to applications
// after system boot. The below algorithm will compute topology for the processors visible
// to the thread that is computing it.

// Extract the 3-levels of IDs on every processor
// SystemAffinity is a bitmask of all the processors started by the OS. Use OS specific APIs to
// obtain it.
// ThreadAffinityMask is used to affinitize the topology enumeration thread to each processor
using OS specific APIs.
// Allocate per processor arrays to store the Package_ID, Core_ID and SMT_ID for every started
// processor.

    ThreadAffinityMask = 1;
   ProcessorNum = 0;
    while (ThreadAffinityMask != 0 && ThreadAffinityMask <= SystemAffinity) {
          // Check to make sure we can utilize this processor first.
```

```
        if (ThreadAffinityMask & SystemAffinity){
            Set thread to run on the processor specified in ThreadAffinityMask
            Wait if necessary and ensure thread is running on specified processor

            APIC_ID = GetAPIC_ID(); // 32 bit ID in Example 8-19 or 8-bit ID in Example 8-20
            Extract the Package_ID, Core_ID and SMT_ID as explained in three level extraction
                algorithm of Example 8-21
            PackageID[ProcessorNUM] = PACKAGE_ID;
            CoreID[ProcessorNum] = CORE_ID;
            SmtID[ProcessorNum] = SMT_ID;
            ProcessorNum++;
        }
        ThreadAffinityMask <<= 1;
    }
    NumStartedLPs = ProcessorNum;
```

b) Using the list of PACKAGE_ID to count the number of physical packages in a MP system and construct, for each package, a multi-bit mask corresponding to those logical processors residing in the same package.

```
    // Compute the number of packages by counting the number of processors
    // with unique PACKAGE_IDs in the PackageID array.
    // Compute the mask of processors in each package.

    PackageIDBucket is an array of unique PACKAGE_ID values. Allocate an array of
    NumStartedLPs count of entries in this array.
    PackageProcessorMask is a corresponding array of the bit mask of processors belonging to
    the same package, these are processors with the same PACKAGE_ID
    The algorithm below assumes there is symmetry across package boundary if more than
    one socket is populated in an MP system.
    // Bucket Package IDs and compute processor mask for every package.

    PackageNum = 1;
    PackageIDBucket[0] = PackageID[0];
    ProcessorMask = 1;
    PackageProcessorMask[0] = ProcessorMask;
    For (ProcessorNum = 1; ProcessorNum < NumStartedLPs; ProcessorNum++) {
        ProcessorMask << = 1;
        For (i=0; i < PackageNum; i++) {
            // we may be comparing bit-fields of logical processors residing in different
            // packages, the code below assume package symmetry
            If (PackageID[ProcessorNum] = PackageIDBucket[i]) {
                PackageProcessorMask[i] |= ProcessorMask;
                Break; // found in existing bucket, skip to next iteration
            }
        }
        if (i =PackageNum) {
            //PACKAGE_ID did not match any bucket, start new bucket
            PackageIDBucket[i] = PackageID[ProcessorNum];
            PackageProcessorMask[i] = ProcessorMask;
            PackageNum++;
        }
    }
    // PackageNum has the number of Packages started in OS
    // PackageProcessorMask[] array has the processor set of each package
```

c) Using the list of CORE_ID to count the number of cores in a MP system and construct, for each core, a multi-bit mask corresponding to those logical processors residing in the same core.

Processors in the same core can be determined by bucketing the processors with the same PACKAGE_ID and CORE_ID. Note that code below can BIT OR the values of PACKGE and CORE ID because they have not been shifted right.
The algorithm below assumes there is symmetry across package boundary if more than one socket is populated in an MP system.

```
//Bucketing PACKAGE and CORE IDs and computing processor mask for every core
    CoreNum = 1;
    CoreIDBucket[0] = PackageID[0] | CoreID[0];
    ProcessorMask = 1;
    CoreProcessorMask[0] = ProcessorMask;
    For (ProcessorNum = 1; ProcessorNum < NumStartedLPs; ProcessorNum++) {
        ProcessorMask << = 1;
        For (i=0; i < CoreNum; i++) {
            // we may be comparing bit-fields of logical processors residing in different
            // packages, the code below assume package symmetry
            If ((PackageID[ProcessorNum] | CoreID[ProcessorNum]) = CoreIDBucket[i]) {
                CoreProcessorMask[i] |= ProcessorMask;
                Break; // found in existing bucket, skip to next iteration
            }
        }
        if (i = CoreNum) {
            //Did not match any bucket, start new bucket
            CoreIDBucket[i] = PackageID[ProcessorNum] | CoreID[ProcessorNum];
            CoreProcessorMask[i] = ProcessorMask;
            CoreNum++;
        }
    }
    // CoreNum has the number of cores started in the OS
    // CoreProcessorMask[] array has the processor set of each core
```

Other processor relationships such as processor mask of sibling cores can be computed from set operations of the PackageProcessorMask[] and CoreProcessorMask[].

The algorithm shown above can be adapted to work with earlier generations of single-core IA-32 processors that support Intel Hyper-Threading Technology and in situations that the deterministic cache parameter leaf is not supported (provided CPUID supports initial APIC ID). A reference code example is available (see *Intel® 64 Architecture Processor Topology Enumeration*).

...

10. Updates to Chapter 9, Volume 3A

Change bars show changes to Chapter 9 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

9.9.2 Switching Back to Real-Address Mode

The processor switches from protected mode back to real-address mode if software clears the PE bit in the CR0 register with a MOV CR0 instruction. A procedure that re-enters real-address mode should perform the following steps:

1. Disable interrupts. A CLI instruction disables maskable hardware interrupts. NMI interrupts can be disabled with external circuitry.
2. If paging is enabled, perform the following operations:
   - Transfer program control to linear addresses that are identity mapped to physical addresses (that is, linear addresses equal physical addresses).
   - Insure that the GDT and IDT are in identity mapped pages.
   - Clear the PG bit in the CR0 register.
   - Move 0H into the CR3 register to flush the TLB.
3. Transfer program control to a readable segment that has a limit of 64 KBytes (FFFFH). This operation loads the CS register with the segment limit required in real-address mode.
4. Load segment registers SS, DS, ES, FS, and GS with a selector for a descriptor containing the following values, which are appropriate for real-address mode:
   - Limit = 64 KBytes (0FFFFH)
   - Byte granular (G = 0)
   - Expand up (E = 0)
   - Writable (W = 1)
   - Present (P = 1)
   - Base = any value

   The segment registers must be loaded with non-null segment selectors or the segment registers will be unusable in real-address mode. Note that if the segment registers are not reloaded, execution continues using the descriptor attributes loaded during protected mode.
5. Execute an LIDT instruction to point to a real-address mode interrupt table that is within the 1-MByte real-address mode address range.
6. Clear the PE flag in the CR0 register to switch to real-address mode.
7. Execute a far JMP instruction to jump to a real-address mode program. This operation flushes the instruction queue and loads the appropriate base-address value in the CS register.
8. Load the SS, DS, ES, FS, and GS registers as needed by the real-address mode code. If any of the registers are not going to be used in real-address mode, write 0s to them.

9. Execute the STI instruction to enable maskable hardware interrupts and perform the necessary hardware operation to enable NMI interrupts.

NOTE

All the code that is executed in steps 1 through 9 must be in a single page and the linear addresses in that page must be identity mapped to physical addresses.

...

Example 9-5. Pseudo Code to Validate the Processor Signature

```
ProcessorSignature ← CPUID(1):EAX

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

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

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

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

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

...

Example 9-6. Pseudo Code Example of Processor Flags Test

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

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

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

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

...

Example 9-7. Pseudo Code Example of Checksum Test

```
N ← 512

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

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

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

...

Example 9-10. Pseudo Code to Authenticate the Update

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

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

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

```
Else
   Fail

...
```

Example 9-11. Pseudo Code, Checks Required Prior to Loading an Update

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

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

...
```

Example 9-12. INT 15 D042 Calling Program Pseudo-code

```
//
// We must be in real mode
//
```

```
If the system is not in Real mode exit
//
// Detect presence of Genuine Intel processor(s) that can be updated
// using(CPUID)
//
If no Intel processors exist that can be updated exit
//
//  Detect the presence of the Intel microcode update extensions
//
If the BIOS fails the PresenceTestexit
//
//  If the APIC is enabled, see if any other processors are out there
//
Read IA32_APICBASE
If APIC enabled
{
    Send Broadcast Message to all processors except self via APIC
    Have all processors execute CPUID, record the Processor Signature
    (i.e.,Extended Family, Extended Model, Type, Family, Model, Stepping)
    Have all processors read IA32_PLATFORM_ID[52:50], record Platform
     Id Bits

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

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

```
{
   //
   // Load each Update
   //
   Issue the WriteUpdate function

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

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

   If any other error returned
   {
      Display Diagnostic
      exit
   }

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

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

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

...
```

11. 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.

...

10.3 THE INTEL® 82489DX EXTERNAL APIC, THE APIC, THE XAPIC, AND THE X2APIC

The local APIC in the P6 family and Pentium processors is an architectural subset of the Intel® 82489DX external APIC. See Section 19.27.1, "Software Visible Differences Between the Local APIC and the 82489DX."

The APIC architecture used in the Pentium 4 and Intel Xeon processors (called the xAPIC architecture) is an extension of the APIC architecture found in the P6 family processors. The primary difference between the APIC and xAPIC architectures is that with the xAPIC architecture, the local APICs and the I/O APIC communicate through the system bus. With the APIC architecture, they communication through the APIC bus (see Section 10.2, "System Bus Vs. APIC Bus"). Also, some APIC architectural features have been extended and/or modified in the xAPIC architecture. These extensions and modifications are described in Section 10.4 through Section 10.10.

The basic operating mode of the xAPIC is **xAPIC mode**. The x2APIC architecture is an extension of the xAPIC architecture, primarily to increase processor addressability. The x2APIC architecture provides backward compatibility to the xAPIC architecture and forward extendability for future Intel platform innovations. These extensions and modifications are supported by a new mode of execution (**x2APIC mode**) are detailed in Section 10.12.

...

10.4.1 The Local APIC Block Diagram

Figure 10-4 gives a functional block diagram for the local APIC. Software interacts with the local APIC by reading and writing its registers. APIC registers are memory-mapped to a 4-KByte region of the processor's physical address space with an initial starting address of FEE00000H. For correct APIC operation, this address space must be mapped to an area of memory that has been designated as strong uncacheable (UC). See Section 11.3, "Methods of Caching Available."

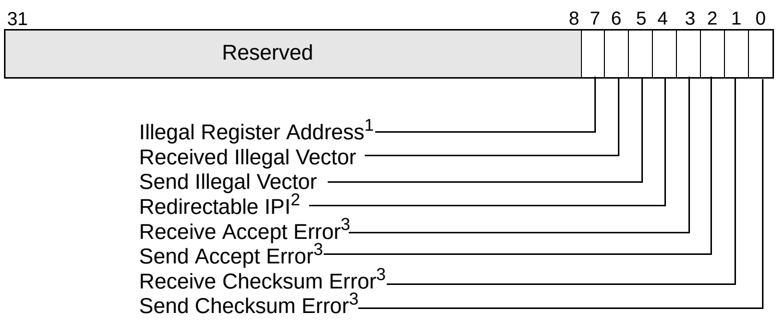
In MP system configurations, the APIC registers for Intel 64 or IA-32 processors on the system bus are initially mapped to the same 4-KByte region of the physical address space. Software has the option of changing initial mapping to a different 4-KByte region for all the local APICs or of mapping the APIC registers for each local APIC to its own 4-KByte region. Section 10.4.5, "Relocating the Local APIC Registers," describes how to relocate the base address for APIC registers.

On processors supporting x2APIC architecture (indicated by CPUID.01H:ECX[21] = 1), the local APIC supports operation both in xAPIC mode and (if enabled by software) in x2APIC mode. x2APIC mode provides extended processor addressability (see Section 10.12).

...

10.5.3 Error Handling

The local APIC records errors detected during interrupt handling in the error status register (ESR). The format of the ESR is given in Figure Figure 10-9.; it contains the following flags:



Address: FEE0 0280H
Value after reset: 0H

NOTES:

1. Used only by Intel Core, Pentium 4, Intel Xeon, and P6 family processors; reserved on the Pentium processor.
2. Used only by some Intel Core and Intel Xeon processors; reserved on other processors.
3. Used only by the P6 family and Pentium processors; reserved on Intel Core, Pentium 4 and Intel Xeon processors.

Figure 10-9. Error Status Register (ESR)

- Bit 0: Send Checksum Error.
  Set when the local APIC detects a checksum error for a message that it sent on the APIC bus. Used only on P6 family and Pentium processors.
- Bit 1: Receive Checksum Error.
  Set when the local APIC detects a checksum error for a message that it received on the APIC bus. Used only on P6 family and Pentium processors.
- Bit 2: Send Accept Error.
  Set when the local APIC detects that a message it sent was not accepted by any APIC on the APIC bus. Used only on P6 family and Pentium processors.
- Bit 3: Receive Accept Error.
  Set when the local APIC detects that the message it received was not accepted by any APIC on the APIC bus, including itself. Used only on P6 family and Pentium processors.
- Bit 4: Redirectable IPI.
  Set when the local APIC detects an attempt to send an IPI with the lowest-priority delivery mode and the local APIC does not support the sending of such IPIs. This bit is used on some Intel Core and Intel Xeon processors. As noted in Section 10.6.2, the ability of a processor to send a lowest-priority IPI is model-specific and should be avoided.

- Bit 5: Send Illegal Vector.
  Set when the local APIC detects an illegal vector (one in the range 0 to 15) in the message that it is sending. This occurs as the result of a write to the ICR (in both xAPIC and x2APIC modes) or to SELF IPI register (x2APIC mode only) with an illegal vector.

  If the local APIC does not support the sending of lowest-priority IPIs and software writes the ICR to send a lowest-priority IPI with an illegal vector, the local APIC sets only the “redirectible IPI” error bit. The interrupt is not processed and hence the “Send Illegal Vector” bit is not set in the ESR.
- Bit 6: Receive Illegal Vector.
  Set when the local APIC detects an illegal vector (one in the range 0 to 15) in an interrupt message it receives or in an interrupt generated locally from the local vector table or via a self IPI. Such interrupts are not be delivered to the processor; the local APIC will never set an IRR bit in the range 0 to 15.
- Bit 7: Illegal Register Address
  Set when the local APIC is in xAPIC mode and software attempts to access a register that is reserved in the processor's local-APIC register-address space; see Table 10-1. (The local-APIC register-address space comprises the 4 KBytes at the physical address specified in the IA32_APIC_BASE MSR.) Used only on Intel Core, Intel Atom™, Pentium 4, Intel Xeon, and P6 family processors.

  In x2APIC mode, software accesses the APIC registers using the RDMSR and WRMSR instructions. Use of one of these instructions to access a reserved register cause a general-protection exception (see Section 10.12.1.3). They do not set the “Illegal Register Access” bit in the ESR.

The ESR is a write/read register. Before attempt to read from the ESR, software should first write to it. (The value written does not affect the values read subsequently; only zero may be written in x2APIC mode.) This write clears any previously logged errors and updates the ESR with any errors detected since the last write to the ESR.

The LVT Error Register (see Section 10.5.1) allows specification of the vector of the interrupt to be delivered to the processor core when APIC error is detected. The register also provides a means of masking an APIC-error interrupt. This masking only prevents delivery of APIC-error interrupts; the APIC continues to record errors in the ESR.

...

10.5.4.1 TSC-Deadline Mode

The mode of operation of the local-APIC timer is determined by the LVT Timer Register. Specifically, if CPUID.01H:ECX.TSC_Deadline[bit 24] = 0, the mode is determined by bit 17 of the register; if CPUID.01H:ECX.TSC_Deadline[bit 24] = 1, the mode is determined by bits 18:17. See Figure 10-8. (If CPUID.01H:ECX.TSC_Deadline[bit 24] = 0, 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 modes are given in Table 10-2. The three modes of the local APIC timer are mutually exclusive.

Table 10-2 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 single interrupt at an absolute time. In TSC-deadline mode, writes to the initial-count register are ignored; and current-count register always reads 0. Instead, timer behavior is controlled using the IA32_TSC_DEADLINE MSR.

The IA32_TSC_DEADLINE MSR (MSR address 6E0H) is a per-logical processor MSR that specifies the time at which a timer interrupt should occur. Writing a non-zero 64-bit value into IA32_TSC_DEADLINE arms the timer. An interrupt is generated when the logical processor's time-stamp counter equals or exceeds the target value in the IA32_TSC_DEADLINE MSR.[1] When the timer generates an interrupt, it disarms itself and clears the IA32_TSC_DEADLINE MSR. Thus, each write to the IA32_TSC_DEADLINE MSR generates at most on timer interrupt.

In TSC-deadline mode, writing 0 to the IA32_TSC_DEADLINE MSR disarms the local-APIC timer. Transitioning between TSC-deadline mode and other timer modes also disarms the timer.

The hardware reset value of the IA32_TSC_DEADLINE MSR is 0. In other timer modes (LVT bit 18 = 0), the IA32_TSC_DEADLINE MSR reads zero and writes are ignored.

Software can configure the TSC-deadline timer to deliver a single interrupt using the following algorithm:

1. Detect support for TSC-deadline mode by verifying CPUID.1:ECX.24 = 1.
2. Select the TSC-deadline mode by programming bits 18:17 of the LVT Timer register with 10b.
3. Program the IA32_TSC_DEADLINE MSR with the target TSC value at which the timer interrupt is desired. This causes the processor to arm the timer.
4. The processor generates a timer interrupt when the value of time-stamp counter is greater than or equal to that of IA32_TSC_DEADLINE. It then disarms the timer and clear the IA32_TSC_DEADLINE MSR. (Both the time-stamp counter and the IA32_TSC_DEADLINE MSR are 64-bit unsigned integers.)
5. Software can re-arm the timer by repeating step 3.

The following are usage guidelines for TSC-deadline mode:

- Writes to the IA32_TSC_DEADLINE MSR are not serialized. Therefore, system software should not use WRMSR to the IA32_TSC_DEADLINE MSR as a serializing

1. If the logical processor is in VMX non-root operation, a read of the time-stamp counter (using either RDMSR, RDTSC, or RDTSCP) may not return the actual value of the time-stamp counter; see Chapter 22 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. It is the responsibility of software operating in VMX root operation to coordinate the virtualization of the time-stamp counter and the IA32_TSC_DEADLINE MSR.

instruction. Read and write accesses to the IA32_TSC_DEADLINE and other MSR registers will occur in program order.

- Software can disarm the timer at any time by writing 0 to the IA32_TSC_DEADLINE MSR.
- If timer is armed, software can change the deadline (forward or backward) by writing a new value to the IA32_TSC_DEADLINE MSR.
- If software disarms the timer or postpones the deadline, race conditions may result in the delivery of a spurious timer interrupt. Software is expected to detect such spurious interrupts by checking the current value of the time-stamp counter to confirm that the interrupt was desired.[1]
- In xAPIC mode (in which the local-APIC registers are memory-mapped), software must serialize between the memory-mapped write to the LVT entry and the WRMSR to IA32_TSC_DEADLINE. In x2APIC mode, no serialization is required between the two writes (by WRMSR) to the LVT and IA32_TSC_DEADLINE MSRs.

The following is a sample algorithm for serializing writes in xAPIC mode:

1. Memory-mapped write to LVT Timer Register, setting bits 18:17 to 10b.
2. WRMSR to the IA32_TSC_DEADLINE MSR a value much larger than current time-stamp counter.
3. If RDMSR of the IA32_TSC_DEADLINE MSR returns zero, go to step 2.
4. WRMSR to the IA32_TSC_DEADLINE MSR the desired deadline.

...

10.6.2.2 Logical Destination Mode

...

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

NOTES

All processors that have their APIC software enabled (using the spurious vector enable/disable bit) must have their DFRs (Destination Format Registers) programmed identically.

The default mode for DFR is flat mode. If you are using cluster mode, DFRs must be programmed before the APIC is software enabled. Since some chipsets do not accurately track a system view of the logical mode, program DFRs as soon as possible after starting the processor.

1. If the logical processor is in VMX non-root operation, a read of the time-stamp counter (using either RDMSR, RDTSC, or RDTSCP) may not return the actual value of the time-stamp counter; see Chapter 22 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. It is the responsibility of software operating in VMX root operation to coordinate the virtualization of the time-stamp counter and the IA32_TSC_DEADLINE MSR.

...

10.12 EXTENDED XAPIC (X2APIC)

The x2APIC architecture extends the xAPIC architecture (described in Section 9.4) in a backward compatible manner and provides forward extendability for future Intel platform innovations. Specifically, the x2APIC architecture does the following:

- Retains all key elements of compatibility to the xAPIC architecture:
  - — delivery modes,
  - — interrupt and processor priorities,
  - — interrupt sources,
  - — interrupt destination types;
- Provides extensions to scale processor addressability for both the logical and physical destination modes;
- Adds new features to enhance performance of interrupt delivery;
- Reduces complexity of logical destination mode interrupt delivery on link based platform architectures.
- Uses MSR programming interface to access APIC registers in x2APIC mode instead of memory-mapped interfaces. Memory-mapped interface is supported when operating in xAPIC mode.

...

12. Updates to Chapter 13, Volume 3A

Change bars show changes to Chapter 13 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

13.1.5 Providing Non-Numeric Exception Handlers for Exceptions Generated by the SSE/SSE2/SSE3/SSSE3/SSE4 Instructions

...

- System Exceptions:
  - — Invalid-opcode exception (#UD). This exception is generated when executing SSE/SSE2/SSE3/SSSE3 instructions under the following conditions:
    - SSE/SSE2/SSE3/SSSE3/SSE4_1/SSE4_2 feature flags returned by CPUID are set to 0. This condition does not affect the CLFLUSH instruction, nor POPCNT.
    - The CLFSH feature flag returned by the CPUID instruction is set to 0. This exception condition only pertains to the execution of the CLFLUSH instruction.
    - The POPCNT feature flag returned by the CPUID instruction is set to 0. This exception condition only pertains to the execution of the POPCNT instruction.

- The EM flag (bit 2) in control register CR0 is set to 1, regardless of the value of TS flag (bit 3) of CR0. This condition does not affect the PAUSE, PREFETCH*h*, MOVNTI, SFENCE, LFENCE, MFENCE, CLFLUSH, CRC32 and POPCNT instructions.
- The OSFXSR flag (bit 9) in control register CR4 is set to 0. This condition does not affect the PAVGB, PAVGW, PEXTRW, PINSRW, PMAXSW, PMAXUB, PMINSW, PMINUB, PMOVMSKB, PMULHUW, PSADBW, PSHUFW, MASKMOVQ, MOVNTQ, MOVNTI, PAUSE, PREFETCH*h*, SFENCE, LFENCE, MFENCE, CLFLUSH, CRC32 and POPCNT instructions.
- Executing a instruction that causes a SIMD floating-point exception when the OSXMMEXCPT flag (bit 10) in control register CR4 is set to 0. See Section 13.5.1, “Using the TS Flag to Control the Saving of the x87 FPU, MMX, SSE, SSE2, SSE3 SSSE3 and SSE4 State.”

— Device not available (#NM). This exception is generated by executing a SSE/SSE2/SSE3/SSSE3/SSE4 instruction when the TS flag (bit 3) of CR0 is set to 1.

...

13. Updates to Chapter 15, Volume 3A

Change bars show changes to Chapter 15 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A:* System Programming Guide, Part 1.

...

15.1 MACHINE-CHECK ARCHITECTURE

The Pentium 4, Intel Xeon, and P6 family processors implement a machine-check architecture that provides a mechanism for detecting and reporting hardware (machine) errors, such as: system bus errors, ECC errors, parity errors, cache errors, and TLB errors. It consists of a set of model-specific registers (MSRs) that are used to set up machine checking and additional banks of MSRs used for recording errors that are detected.

The processor signals the detection of an uncorrected machine-check error by generating a machine-check exception (#MC), which is an abort class exception. The implementation of the machine-check architecture does not ordinarily permit the processor to be restarted reliably after generating a machine-check exception. However, the machine-check-exception handler can collect information about the machine-check error from the machine-check MSRs.

Starting with 45nm Intel 64 processor on which CPUID reports DisplayFamily_DisplayModel as 06H_1AH (see CPUID instruction in Chapter 3, “Instruction Set Reference, A-M” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*), the processor can report information on corrected machine-check errors and deliver a programmable interrupt for software to respond to MC errors, referred to as corrected machine-check error interrupt (CMCI). See Section 15.5 for detail.

Intel 64 processors supporting machine-check architecture and CMCI may also support an additional enhancement, namely, support for software recovery from certain uncorrected recoverable machine check errors. See Section 15.6 for detail.

...

15.3.2.1 IA32_MCi_CTL MSRs

...

NOTE

For P6 family processors, processors based on Intel Core microarchitecture (excluding those on which on which CPUID reports DisplayFamily_DisplayModel as 06H_1AH and onward): the operating system or executive software must not modify the contents of the IA32_MC0_CTL MSR. This MSR is internally aliased to the EBL_CR_POWERON MSR and controls platform-specific error handling features. System specific firmware (the BIOS) is responsible for the appropriate initialization of the IA32_MC0_CTL MSR. P6 family processors only allow the writing of all 1s or all 0s to the IA32_MC*i*_CTL MSR.

15.3.2.2 IA32_MCi_STATUS MSRS

Each IA32_MCi_STATUS MSR contains information related to a machine-check error if its VAL (valid) flag is set (see Figure 15-5). Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.

NOTE

Figure 15-5 depicts the IA32_MCi_STATUS MSR when IA32_MCG_CAP[24] = 1, IA32_MCG_CAP[11] = 1 and IA32_MCG_CAP[10] = 1. When IA32_MCG_CAP[24] = 0 and IA32_MCG_CAP[11] = 1, bits 56:55 is reserved and bits 54:53 for threshold-based error reporting. When IA32_MCG_CAP[11] = 0, bits 56:53 are part of the "Other Information" field. The use of bits 54:53 for threshold-based error reporting began with Intel Core Duo processors, and is currently used for cache memory. See Section 15.4, "Enhanced Cache Error reporting," for more information. When IA32_MCG_CAP[10] = 0, bits 52:38 are part of the "Other Information" field. The use of bits 52:38 for corrected MC error count is introduced with Intel 64 processor on which CPUID reports DisplayFamily_DisplayModel as 06H_1AH.

...

15.6 RECOVERY OF UNCORRECTED RECOVERABLE (UCR) ERRORS

Recovery of uncorrected recoverable machine check errors is an enhancement in machine-check architecture. The first processor that supports this feature is 45nm Intel 64 processor on which CPUID reports DisplayFamily_DisplayModel as 06H_2EH (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-M" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). This allow system software to perform recovery action on certain class of uncorrected errors and continue execution.

...

15.10.2 Pentium Processor Machine-Check Exception Handling

Machine-check exception handler on P6 family and later processor families, should follow the guidelines described in Section 15.10.1 and Example 15-2 that check the processor's support of MCA.

NOTE

On processors that support MCA (CPUID.1.EDX.MCA = 1) reading the P5_MC_TYPE and P5_MC_ADDR registers may produce invalid data.

...

15.10.4.1 Machine-Check Exception Handler for Error Recovery

When writing a machine-check exception (MCE) handler to support software recovery from Uncorrected Recoverable (UCR) errors, consider the following:

- When IA32_MCG_CAP [24] is zero, there are no recoverable errors supported and all machine-check are fatal exceptions. The logging of status and error information is therefore a baseline implementation requirement.
- When IA32_MCG_CAP [24] is 1, certain uncorrected errors called uncorrected recoverable (UCR) errors may be software recoverable. The handler can analyze the reported error information, and in some cases attempt to recover from the uncorrected error and continue execution.
- For processors on which CPUID reports DisplayFamily_DisplayModel as 06H_0EH and onward, an MCA signal is broadcast to all logical processors in the system (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-M" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). Due to the potentially shared machine check MSR resources among the logical processors on the same package/core, the MCE handler may be required to synchronize with the other processors that received a machine check error and serialize access to the machine check registers when analyzing, logging and clearing the information in the machine check registers.
- The VAL (valid) flag in each IA32_MCi_STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank do not contain valid error information and should not be checked.
- The MCE handler is primarily responsible for processing uncorrected errors. The UC flag in each IA32_MCi_Status register indicates whether the reported error was corrected (UC=0) or uncorrected (UC=1). The MCE handler can optionally log and clear the corrected errors in the MC banks if it can implement software algorithm to avoid the undesired race conditions with the CMCI or CMC polling handler.
- For uncorrectable errors, the EIPV flag in the IA32_MCG_STATUS register indicates (when set) that the instruction pointed to by the instruction pointer pushed onto the stack when the machine-check exception is generated is directly associated with the error. When this flag is cleared, the instruction pointed to may not be associated with the error.
- The MCIP flag in the IA32_MCG_STATUS register indicates whether a machine-check exception was generated. When a machine check exception is generated, it is expected that the MCIP flag in the IA32_MCG_STATUS register is set to 1. If it is not

set, this machine check was generated by either an INT 18 instruction or some piece of hardware signaling an interrupt with vector 18.

...

14. 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.

...

16.2.4 Debug Control Register (DR7)

...

NOTES

For Pentium® 4 and Intel® Xeon® processors with a CPUID signature corresponding to family 15 (model 3, 4, and 6), break point conditions permit specifying 8-byte length on data read/write with an of encoding 10B in the LEN*n* field.

Encoding 10B is also supported in processors based on Intel Core microarchitecture or enhanced Intel Core microarchitecture, the respective CPUID signatures corresponding to family 6, model 15, and family 6, DisplayModel value 23 (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-M" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). The Encoding 10B is supported in processors based on Intel® Atom™ microarchitecture, with CPUID signature of family 6, DisplayModel value 28. The encoding 10B is undefined for other processors.

...

16.4.8 LBR Stack

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported across Intel 64 and IA-32 processor families. However, the number of MSRs in the LBR stack and the valid range of TOS pointer value can vary between different processor families. Table 16-3 lists the LBR stack size and TOS pointer range for several processor families according to the CPUID signatures of DisplayFamily_DisplayModel encoding (see CPUID instruction in Chapter 3 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

...

The last branch recording mechanism tracks not only branch instructions (like JMP, Jcc, LOOP and CALL instructions), but also other operations that cause a change in the instruction pointer (like external interrupts, traps and faults). The branch recording mechanisms generally employs a set of MSRs, referred to as last branch record (LRB) stack. The size and exact locations of the LRB stack are generally model-specific (see Appendix B, "Model-Specific Registers (MSRs)" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B* for model-specific MSR addresses).

...

16.11 TIME-STAMP COUNTER

The Intel 64 and IA-32 architectures (beginning with the Pentium processor) define a time-stamp counter mechanism that can be used to monitor and identify the relative time occurrence of processor events. The counter's architecture includes the following components:

- **TSC flag —** A feature bit that indicates the availability of the time-stamp counter. The counter is available in an if the function CPUID.1:EDX.TSC[bit 4] = 1.
- **IA32_TIME_STAMP_COUNTER MSR** (called TSC MSR in P6 family and Pentium processors) **—** The MSR used as the counter.
- **RDTSC instruction —** An instruction used to read the time-stamp counter.
- **TSD flag —** A control register flag is used to enable or disable the time-stamp counter (enabled if CR4.TSD[bit 2] = 1).

The time-stamp counter (as implemented in the P6 family, Pentium, Pentium M, Pentium 4, Intel Xeon, Intel Core Solo and Intel Core Duo processors and later processors) is a 64-bit counter that is set to 0 following a RESET of the processor. Following a RESET, the counter increments even when the processor is halted by the HLT instruction or the external STPCLK# pin. Note that the assertion of the external DPSLP# pin may cause the time-stamp counter to stop.

Processor families increment the time-stamp counter differently:

- For Pentium M processors (family [06H], models [09H, 0DH]); for Pentium 4 processors, Intel Xeon processors (family [0FH], models [00H, 01H, or 02H]); and for P6 family processors: the time-stamp counter increments with every internal processor clock cycle.

  The internal processor clock cycle is determined by the current core-clock to bus-clock ratio. Intel® SpeedStep® technology transitions may also impact the processor clock.
- For Pentium 4 processors, Intel Xeon processors (family [0FH], models [03H and higher]); for Intel Core Solo and Intel Core Duo processors (family [06H], model [0EH]); for the Intel Xeon processor 5100 series and Intel Core 2 Duo processors (family [06H], model [0FH]); for Intel Core 2 and Intel Xeon processors (family [06H], DisplayModel [17H]); for Intel Atom processors (family [06H], DisplayModel [1CH]): the time-stamp counter increments at a constant rate. That rate may be set by the maximum core-clock to bus-clock ratio of the processor or may be set by the maximum resolved frequency at which the processor is booted. The maximum resolved frequency may differ from the maximum qualified frequency of the processor, see Section 30.10.5 for more detail.

  The specific processor configuration determines the behavior. Constant TSC behavior ensures that the duration of each clock tick is uniform and supports the use of the TSC as a wall clock timer even if the processor core changes frequency. This is the architectural behavior moving forward.

NOTE

To determine average processor clock frequency, Intel recommends the use of EMON logic to count processor core clocks over the period of time for which the average is required. See Section 30.10, "Counting Clocks," and Appendix A, "Performance-Monitoring Events," for more information.

The RDTSC instruction reads the time-stamp counter and is guaranteed to return a monotonically increasing unique value whenever executed, except for a 64-bit counter wraparound. Intel guarantees that the time-stamp counter will not wraparound within 10 years after being reset. The period for counter wrap is longer for Pentium 4, Intel Xeon, P6 family, and Pentium processors.

Normally, the RDTSC instruction can be executed by programs and procedures running at any privilege level and in virtual-8086 mode. The TSD flag allows use of this instruction to be restricted to programs and procedures running at privilege level 0. A secure operating system would set the TSD flag during system initialization to disable user access to the time-stamp counter. An operating system that disables user access to the time-stamp counter should emulate the instruction through a user-accessible programming interface.

The RDTSC instruction is not serializing or ordered with other instructions. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDTSC instruction operation is performed.

The RDMSR and WRMSR instructions read and write the time-stamp counter, treating the time-stamp counter as an ordinary MSR (address 10H). In the Pentium 4, Intel Xeon, and P6 family processors, all 64-bits of the time-stamp counter are read using RDMSR (just as with RDTSC). When WRMSR is used to write the time-stamp counter on processors before family [0FH], models [03H, 04H]: only the low-order 32-bits of the time-stamp counter can be written (the high-order 32 bits are cleared to 0). For family [0FH], models [03H, 04H, 06H]; for family [06H]], model [0EH, 0FH]; for family [06H]], DisplayModel [17H, 1AH, 1CH, 1DH]: all 64 bits are writable.

...

15. 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.

...

22.4 CHANGES TO INSTRUCTION BEHAVIOR IN VMX NON-ROOT OPERATION

The behavior of some instructions is changed in VMX non-root operation. Some of these changes are determined by the settings of certain VM-execution control fields. The following items detail such changes:

- **CLTS.** Behavior of the CLTS instruction is determined by the bits in position 3 (corresponding to CR0.TS) in the CR0 guest/host mask and the CR0 read shadow:
  - — If bit 3 in the CR0 guest/host mask is 0, CLTS clears CR0.TS normally (the value of bit 3 in the CR0 read shadow is irrelevant in this case), unless CR0.TS is fixed to 1 in VMX operation (see Section 20.8), in which case CLTS causes a general-protection exception.

— If bit 3 in the CR0 guest/host mask is 1 and bit 3 in the CR0 read shadow is 0, CLTS completes but does not change the contents of CR0.TS.

— If the bits in position 3 in the CR0 guest/host mask and the CR0 read shadow are both 1, CLTS causes a VM exit (see Section 22.1.3).

- **IRET.** Behavior of IRET with regard to NMI blocking (see Table 21-3) is determined by the settings of the "NMI exiting" and "virtual NMIs" VM-execution controls:

  — If the "NMI exiting" VM-execution control is 0, IRET operates normally and unblocks NMIs. (If the "NMI exiting" VM-execution control is 0, the "virtual NMIs" control must be 0; see Section 23.2.1.1.)

  — If the "NMI exiting" VM-execution control is 1, IRET does not affect blocking of NMIs. If, in addition, the "virtual NMIs" VM-execution control is 1, the logical processor tracks virtual-NMI blocking. In this case, IRET removes any virtual-NMI blocking.

  The unblocking of NMIs or virtual NMIs specified above occurs even if IRET causes a fault.

...

16. 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.

...

23.2 CHECKS ON VMX CONTROLS AND HOST-STATE AREA

If the checks in Section 23.1 do not cause VM entry to fail, the control and host-state areas of the VMCS are checked to ensure that they are proper for supporting VMX non-root operation, that the VMCS is correctly configured to support the next VM exit, and that, after the next VM exit, the processor's state is consistent with the Intel 64 and IA-32 architectures.

VM entry fails if any of these checks fail. When such failures occur, control is passed to the next instruction, RFLAGS.ZF is set to 1 to indicate the failure, and the VM-instruction error field is loaded with an error number that indicates whether the failure was due to the controls or the host-state area (see Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*).

These checks may be performed in any order. Thus, an indication by error number of one cause (for example, host state) does not imply that there are not also other errors. Different processors may thus give different error numbers for the same VMCS. Some checks prevent establishment of settings (or combinations of settings) that are currently reserved. Future processors may allow such settings (or combinations) and may not perform the corresponding checks. The correctness of software should not rely on VM-entry failures resulting from the checks documented in this section.

The checks on the controls and the host-state area are presented in Section 23.2.1 through Section 23.2.4. These sections reference VMCS fields that correspond to processor state. Unless otherwise stated, these references are to fields in the host-state area.

...

23.3.1 Checks on the Guest State Area

This section describes checks performed on fields in the guest-state area. These checks may be performed in any order. Some checks prevent establishment of settings (or combinations of settings) that are currently reserved. Future processors may allow such settings (or combinations) and may not perform the corresponding checks. The correctness of software should not rely on VM-entry failures resulting from the checks documented in this section.

The following subsections reference fields that correspond to processor state. Unless otherwise stated, these references are to fields in the guest-state area.

...

23.6.1 Interruptibility State

The interruptibility-state field in the guest-state area (see Table 21-3) contains bits that control blocking by STI, blocking by MOV SS, and blocking by NMI. This field impacts event blocking after VM entry as follows:

- If the VM entry is vectoring, there is no blocking by STI or by MOV SS following the VM entry, regardless of the contents of the interruptibility-state field.
- If the VM entry is not vectoring, the following apply:
  - — Events are blocked by STI if and only if bit 0 in the interruptibility-state field is 1. This blocking is cleared after the guest executes one instruction or incurs an exception (including a debug exception made pending by VM entry; see Section 23.6.3).
  - — Events are blocked by MOV SS if and only if bit 1 in the interruptibility-state field is 1. This may affect the treatment of pending debug exceptions; see Section 23.6.3. This blocking is cleared after the guest executes one instruction or incurs an exception (including a debug exception made pending by VM entry).
- The blocking of non-maskable interrupts (NMIs) is determined as follows:
  - — If the “virtual NMIs” VM-execution control is 0, NMIs are blocked if and only if bit 3 (blocking by NMI) in the interruptibility-state field is 1. If the “NMI exiting” VM-execution control is 0, execution of the IRET instruction removes this blocking (even if the instruction generates a fault). If the “NMI exiting” control is 1, IRET does not affect this blocking.
  - — The following items describe the use of bit 3 (blocking by NMI) in the interruptibility-state field if the “virtual NMIs” VM-execution control is 1:
    - The bit’s value does not affect the blocking of NMIs after VM entry. NMIs are not blocked in VMX non-root operation (except for ordinary blocking for other reasons, such as by the MOV SS instruction, the wait-for-SIPI state, etc.)
    - The bit’s value determines whether there is virtual-NMI blocking after VM entry. If the bit is 1, virtual-NMI blocking is in effect after VM entry. If the bit is 0, there is no virtual-NMI blocking after VM entry unless the VM entry is injecting an NMI (see Section 23.5.1.1). Execution of IRET removes virtual-NMI blocking (even if the instruction generates a fault).

  If the “NMI exiting” VM-execution control is 0, the “virtual NMIs” control must be 0; see Section 23.2.1.1.

- Blocking of system-management interrupts (SMIs) is determined as follows:
  - — If the VM entry was not executed in system-management mode (SMM), SMI blocking is unchanged by VM entry.
  - — If the VM entry was executed in SMM, SMIs are blocked after VM entry if and only if the bit 2 in the interruptibility-state field is 1.

...

23.6.4 VMX-Preemption Timer

If the “activate VMX-preemption timer” VM-execution control is 1, VM entry starts the VMX-preemption timer with the unsigned value in the VMX-preemption timer-value field.

It is possible for the VMX-preemption timer to expire during VM entry (e.g., if the value in the VMX-preemption timer-value field is zero). If this happens (and if the VM entry was not to the shutdown state or the wait-for-SIPI state), a VM exit occurs with its normal priority after any event injection and before execution of any instruction following VM entry. For example, any pending debug exceptions established by VM entry (see Section 23.6.3) take priority over a timer-induced VM exit. (The timer-induced VM exit will occur after delivery of the debug exception, unless that exception or its delivery causes a different VM exit.)

See Section 22.7.1 for details of the operation of the VMX-preemption timer in VMX non-root operation, including the blocking and priority of the VM exits that it causes.

...

17. 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.

...

24.3.4 Saving Non-Register State

Information corresponding to guest non-register state is saved as follows:

...

- If the “save VMX-preemption timer value” VM-exit control is 1, the value of timer is saved into the VMX-preemption timer-value field. This is the value loaded from this field on VM entry as subsequently decremented (see Section 22.7.1). VM exits due to timer expiration save the value 0. Other VM exits may also save the value 0 if the timer expired during VM exit. (If the “save VMX-preemption timer value” VM-exit control is 0, VM exit does not modify the value of the VMX-preemption timer-value field.)

...

18. 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.

...

25.3.2 Creating and Using Cached Translation Information

...

The following items detail the use of the various mappings:

- If EPT is not in use (e.g., when outside VMX non-root operation), a logical processor may use cached mappings as follows:
  - For accesses using linear addresses, it may use linear mappings associated with the current VPID and the current PCID. It may also use global TLB entries (linear mappings) associated with the current VPID and any PCID.
  - No guest-physical or combined mappings are used while EPT is not in use.
- If EPT is in use, a logical processor may use cached mappings as follows:
  - For accesses using linear addresses, it may use combined mappings associated with the current VPID, the current PCID, and the current EP4TA. It may also use global TLB entries (combined mappings) associated with the current VPID, the current EP4TA, and any PCID.
  - For accesses using guest-physical addresses, it may use guest-physical mappings associated with the current EP4TA.
  - No linear mappings are used while EPT is in use.

...

19. Updates to Chapter 27, Volume 3B

Change bars show changes to Chapter 27 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

27.2.1 Using Unrestricted Guest Mode

Processors which support the “unrestricted guest” VM-execution control allow VM software to run in real-address mode and unpaged protected mode. Since these modes do not use paging, VMM software must virtualize guest memory using EPT.

Special notes for 64-bit VMM software using the 1-setting of the “unrestricted guest” VM-execution control:

- It is recommended that 64-bit VMM software use the 1-settings of the "load IA32_EFER" VM entry control and the "save IA32_EFER" VM-exit control. If VM entry is establishing CR0.PG=0 and if the "IA-32e mode guest" and "load IA32_EFER" VM entry controls are both 0, VM entry leaves IA32_EFER.LME unmodified (i.e., the host value will persist in the guest).

- It is not necessary for VMM software to track guest transitions into and out of IA-32e mode for the purpose of maintaining the correct setting of the "IA-32e mode guest" VM entry control. This is because VM exits on processors supporting the 1-setting of the "unrestricted guest" VM-execution control save the (guest) value of IA32_EFER.LMA into the "IA-32e mode guest" VM entry control.

...

27.13 USE OF THE VMX-PREEMPTION TIMER

The VMX-preemption timer allows VMM software to preempt guest VM execution after a specified amount of time. Typical VMX-preemption timer usage is to program the initial VM quantum into the timer, save the timer value on each successive VM-exit (using the VM-exit control “save preemption timer value”) and run the VM until the timer expires.

In an alternative scenario, the VMM may use another timer (e.g. the TSC) to track the amount of time the VM has run while still using the VMX-preemption timer for VM preemption. In this scenario the VMM would not save the VMX-preemption timer on each VM-exit but instead would reload the VMX-preemption timer with initial VM quantum less the time the VM has already run. This scenario includes all the VM-entry and VM-exit latencies in the VM run time.

In both scenarios, on each successive VM-entry the VMX-preemption timer contains a smaller value until the VM quantum ends. If the VMX-preemption timer is loaded with a value smaller than the VM-entry latency then the VM will not execute any instructions before the timer expires. The VMM must ensure the initial VM quantum is greater than the VM-entry latency; otherwise the VM will make no forward progress.

...

20. 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.

...

30.6.1.1 Precise Event Based Sampling (PEBS)

...

In IA-32e mode, the full 64-bit value is written to the register. If the processor is not operating in IA-32e mode, 32-bit value is written to registers with bits 63:32 zeroed. Registers not defined when the processor is not in IA32e mode are written to zero.

Bytes 0xAF:0x90 are enhancement to the PEBS record format. Support for this enhanced PEBS record format is indicated by IA32_PERF_CAPABILITIES[11:8] encoding of 0001B.

...

30.14 PERFORMANCE MONITORING ON L3 AND CACHING BUS CONTROLLER SUB-SYSTEMS

The Intel Xeon processor 7400 series and Dual-Core Intel Xeon processor 7100 series employ a distinct L3/caching bus controller sub-system. These sub-system have a unique set of performance monitoring capability and programming interfaces that are largely common between these two processor families.

Intel Xeon processor 7400 series are based on 45nm enhanced Intel Core microarchitecture. The CPUID signature is indicated by DisplayFamily_DisplayModel value of 06_1DH (see CPUID instruction in Chapter 3, “Instruction Set Reference, A-M” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*). Intel Xeon processor 7400 series have six processor cores that share an L3 cache.

...

21. 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 Tables B-12, B-17, and B-18, respectively. All MSRs listed can be read with the RDMSR and written with the WRMSR instructions.

Register addresses are given in both hexadecimal and decimal. The register name is the mnemonic register name and the bit description describes individual bits in registers.

Model specific registers and its bit-fields may be supported for a finite range of processor families/models. To distinguish between different processor family and/or models, software must use CPUID.01H leaf function to query the combination of DisplayFamily and DisplayModel to determine model-specific availability of MSRs (see CPUID instruction in Chapter 3, “Instruction Set Reference, A-M” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*). Table Table B-1 lists the signature values of DisplayFamily and DisplayModel for various processor families or processor number series.

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
06_1AH	Intel Core i7 Processor, Intel Xeon Processor 5500 series
06_1EH, 06_1FH	Intel Core i7 and i5 Processor,
06_2EH	Intel Xeon Processor 7500 series
06_25H, 06_2CH	Intel Xeon Processor 5600 series, Intel Core i7, i5 and i3 Processor
06_1DH	Intel Xeon Processor MP 7400 series
06_17H	Intel Xeon Processor 5200, 5400 series, Intel Core 2 Quad processors 8000, 9000 series

Table B-1 CPUID Signature (Continued)Values of DisplayFamily_DisplayModel

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_0FH	Intel Xeon Processor 3000, 3200, 5100, 5300, 7300 series, Intel Core 2 Quad processor 6000 series, Intel Core 2 Extreme 6000 series, Intel Core 2 Duo 4000, 5000, 6000, 7000 series processors, Intel Pentium dual-core processors
06_0EH	Intel Core Duo, Intel Core Solo processors
06_0DH	Intel Pentium M processor
06_1CH	Intel Atom processor
0F_06H	Intel Xeon processor 7100, 5000 Series, Intel Xeon Processor MP, Intel Pentium 4, Pentium D processors
0F_03H, 0F_04H	Intel Xeon Processor, Intel Xeon Processor MP, Intel Pentium 4, Pentium D processors
06_09H	Intel Pentium M processor
0F_02H	Intel Xeon Processor, Intel Xeon Processor MP, Intel Pentium 4 processors
0F_0H, 0F_01H	Intel Xeon Processor, Intel Xeon Processor MP, Intel Pentium 4 processors
06_7H, 06_08H, 06_0AH, 06_0BH	Intel Pentium III Xeon Processor, Intel Pentium III Processor
06_03H, 06_05H	Intel Pentium II Xeon Processor, Intel Pentium II Processor
06_01H	Intel Pentium Pro Processor
05_01H, 05_02H, 05_04H	Intel Pentium Processor, Intel Pentium Processor with MMX Technology

...

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			
...				
1A0H	416	IA32_MISC_ENABLE	**Enable Misc. Processor Features. (R/W)** Allows a variety of processor functions to be enabled and disabled.	
		0	**Fast-Strings Enable.** When set, the fast-strings feature (for REP MOVS and REP STORS) is enabled (default); when clear, fast-strings are disabled.	0F_0H
		2:1	Reserved.	

		3	**Automatic Thermal Control Circuit Enable. (R/W)** 1 = Setting this bit enables the thermal control circuit (TCC) portion of the Intel Thermal Monitor feature. This allows the processor to automatically reduce power consumption in response to TCC activation. 0 = Disabled (default). Note: In some products clearing this bit might be ignored in critical thermal conditions, and TM1, TM2 and adaptive thermal throttling will still be activated.	0F_0H
		6:4	Reserved	
		7	**Performance Monitoring Available. (R)** 1 = Performance monitoring enabled 0 = Performance monitoring disabled	0F_0H
		10:8	Reserved	
		11	**Branch Trace Storage Unavailable. (RO)** 1 = Processor doesn't support branch trace storage (BTS) 0 = BTS is supported	0F_0H
...				
3F1H	1009	IA32_PEBS_ENABLE	PEBS Control (R/W)	
		0	Enable PEBS on IA32_PMC0	06_0FH
		1-3	Reserved or Model specific	
		31:4	Reserved	
		35-32	Reserved or Model specific	
		63:36	Reserved	
...				
6E0H	1760	IA32_TSC_DEADLINE	**TSC Target of Local APIC's TSC Deadline Mode. (R/W)**	If(CPUID.01H:ECX.[bit 25] = 1

...				
802H	2050	IA32_X2PIC_APICID	**x2APIC ID Register. (R/O)** See x2APIC Specification	If (CPUID.01H:ECX.[bit 21] = 1)
803H	2051	IA32_X2APIC_VERSION	**x2APIC Version Register. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
808H	2056	IA32_X2APIC_TPR	**x2APIC Task Priority Register. (R/W)**	If (CPUID.01H:ECX.[bit 21] = 1)
80AH	2058	IA32_X2APIC_PPR	**x2APIC Processor Priority Register. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
80BH	2059	IA32_X2APIC_EOI	**x2APIC EOI Register. (W/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
80DH	2061	IA32_X2APIC_LDR	**x2APIC Logical Destination Register. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
80FH	2063	IA32_X2APIC_SIVR	**x2APIC Spurious Interrupt Vector Register. (R/W)**	If (CPUID.01H:ECX.[bit 21] = 1)
810H	2064	IA32_X2APIC_ISR0	**x2APIC In-Service Register Bits 31:0. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
811H	2065	IA32_X2APIC_ISR1	**x2APIC In-Service Register Bits 63:32. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
812H	2066	IA32_X2APIC_ISR2	**x2APIC In-Service Register Bits 95:64. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
813H	2067	IA32_X2APIC_ISR3	**x2APIC In-Service Register Bits 127:96. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
814H	2068	IA32_X2APIC_ISR4	**x2APIC In-Service Register Bits 159:128. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
815H	2069	IA32_X2APIC_ISR5	**x2APIC In-Service Register Bits 191:160. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
816H	2070	IA32_X2APIC_ISR6	**x2APIC In-Service Register Bits 223:192. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
817H	2071	IA32_X2APIC_ISR7	**x2APIC In-Service Register Bits 255:224. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)

818H	2072	IA32_X2APIC_TMR0	**x2APIC Trigger Mode Register Bits 31:0. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
819H	2073	IA32_X2APIC_TMR1	**x2APIC Trigger Mode Register Bits 63:32. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
81AH	2074	IA32_X2APIC_TMR2	**x2APIC Trigger Mode Register Bits 95:64. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
81BH	2075	IA32_X2APIC_TMR3	**x2APIC Trigger Mode Register Bits 127:96. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
81CH	2076	IA32_X2APIC_TMR4	**x2APIC Trigger Mode Register Bits 159:128 (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
81DH	2077	IA32_X2APIC_TMR5	**x2APIC Trigger Mode Register Bits 191:160 (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
81EH	2078	IA32_X2APIC_TMR6	**x2APIC Trigger Mode Register Bits 223:192 (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
81FH	2079	IA32_X2APIC_TMR7	**x2APIC Trigger Mode Register Bits 255:224 (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
820H	2080	IA32_X2APIC_IRR0	**x2APIC Interrupt Request Register Bits 31:0. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
821H	2081	IA32_X2APIC_IRR1	**x2APIC Interrupt Request Register Bits 63:32. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
822H	2082	IA32_X2APIC_IRR2	**x2APIC Interrupt Request Register Bits 95:64. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
823H	2083	IA32_X2APIC_IRR3	**x2APIC Interrupt Request Register Bits 127:96. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
824H	2084	IA32_X2APIC_IRR4	**x2APIC Interrupt Request Register Bits 159:128. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
825H	2085	IA32_X2APIC_IRR5	**x2APIC Interrupt Request Register Bits 191:160. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
826H	2086	IA32_X2APIC_IRR6	**x2APIC Interrupt Request Register Bits 223:192. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)

827H	2087	IA32_X2APIC_IRR7	**x2APIC Interrupt Request Register Bits 255:224. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
828H	2088	IA32_X2APIC_ESR	**x2APIC Error Status Register. (R/W)**	If (CPUID.01H:ECX.[bit 21] = 1)
82FH	2095	IA32_X2APIC_LVT_CMCI	**x2APIC LVT Corrected Machine Check Interrupt Register. (R/W)**	If (CPUID.01H:ECX.[bit 21] = 1)
830H	2096	IA32_X2APIC_ICR	**x2APIC Interrupt Command Register. (R/W)**	If (CPUID.01H:ECX.[bit 21] = 1)
832H	2098	IA32_X2APIC_LVT_TIMER	**x2APIC LVT Timer Interrupt Register. (R/W)**	If (CPUID.01H:ECX.[bit 21] = 1)
833H	2099	IA32_X2APIC_LVT_THERMAL	**x2APIC LVT Thermal Sensor Interrupt Register. (R/W)**	If (CPUID.01H:ECX.[bit 21] = 1)
834H	2100	IA32_X2APIC_LVT_PMI	**x2APIC LVT Performance Monitor Interrupt Register. (R/W)**	If (CPUID.01H:ECX.[bit 21] = 1)
835H	2101	IA32_X2APIC_LVT_LINT0	**x2APIC LVT LINT0 Register. (R/W)**	If (CPUID.01H:ECX.[bit 21] = 1)
836H	2102	IA32_X2APIC_LVT_LINT1	**x2APIC LVT LINT1 Register. (R/W)**	If (CPUID.01H:ECX.[bit 21] = 1)
837H	2103	IA32_X2APIC_LVT_ERROR	**x2APIC LVT Error Register. (R/W)**	If (CPUID.01H:ECX.[bit 21] = 1)
838H	2104	IA32_X2APIC_INIT_COUNT	**x2APIC Initial Count Register. (R/W)**	If (CPUID.01H:ECX.[bit 21] = 1)
839H	2105	IA32_X2APIC_CUR_COUNT	**x2APIC Current Count Register. (R/O)**	If (CPUID.01H:ECX.[bit 21] = 1)
83EH	2110	IA32_X2APIC_DIV_CONF	**x2APIC Divide Configuration Register. (R/W)**	If (CPUID.01H:ECX.[bit 21] = 1)
83FH	2111	IA32_X2APIC_SELF_IPI	**x2APIC Self IPI Register. (W/O)**	If (CPUID.01H:ECX.[bit 21] = 1)

...

Table B-3. MSRs in Processors Based on Intel Core Microarchitecture

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
...				
1C9H	457	MSR_ LASTBRANCH_ TOS	Unique	**Last Branch Record Stack TOS. (R)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H).
...				
3F1H	1009	MSR_PEBS_ ENABLE	Unique	see Table Table B-2.. See Section 30.4.4, “Precise Event Based Sampling (PEBS).”
		0		Enable PEBS on IA32_PMC0. (R/W)
...				

...

Table B-4. MSRs in Intel Atom Processor Family

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
...				
1C9H	457	MSR_ LASTBRANCH_ TOS	Unique	**Last Branch Record Stack TOS. (R)** Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H).
...				
3F1H	1009	MSR_PEBS_ ENABLE	Unique	see Table Table B-2.. See Section 30.4.4, “Precise Event Based Sampling (PEBS).”
		0		Enable PEBS on IA32_PMC0. (R/W)
...				

...

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

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
...				
1C9H	457	MSR_ LASTBRANCH_ TOS	Thread	**Last Branch Record Stack TOS. (R)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 680H).
...				
3F1H	1009	MSR_PEBS_ ENABLE	Thread	see See Section 30.6.1.1, “Precise Event Based Sampling (PEBS).”
		0		Enable PEBS on IA32_PMC0. (R/W)
		1		Enable PEBS on IA32_PMC1. (R/W)
		2		Enable PEBS on IA32_PMC2. (R/W)
		3		Enable PEBS on IA32_PMC3. (R/W)
		31:4		Reserved
		32		Enable Load Latency on IA32_PMC0. (R/W)
		33		Enable Load Latency on IA32_PMC1. (R/W)
		34		Enable Load Latency on IA32_PMC2. (R/W)
		35		Enable Load Latency on IA32_PMC3. (R/W)
		63:36		Reserved
3F6H	1014	MSR_PEBS_ LD_LAT	Thread	see See Section 30.6.1.2, “Load Latency Performance Monitoring Facility.”
		15:0		Minimum threshold latency value of tagged load operation that will be counted. (R/W)
		63:36		Reserved
3F8H	1016	MSR_PKG_C3_RES IDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.
3F9H	1017	MSR_PKG_C6_RES IDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

		63:0		Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.
3FAH	1018	MSR_PKG_C7_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C7 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C7 states. Count at the same frequency as the TSC.
3FCH	1020	MSR_CORE_C3_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C3 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.
3FDH	1021	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C6 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C6 states. Count at the same frequency as the TSC.
...				
802H	2050	IA32_X2PIC_APICID	Thread	x2APIC ID register (R/O) see x2APIC specification
803H	2051	IA32_X2APIC_VERSION	Thread	x2APIC Version register (R/O)
808H	2056	IA32_X2APIC_TPR	Thread	x2APIC Task Priority register (R/W)
80AH	2058	IA32_X2APIC_PPR	Thread	x2APIC Processor Priority register (R/O)
80BH	2059	IA32_X2APIC_EOI	Thread	x2APIC EOI register (W/O)
80DH	2061	IA32_X2APIC_LDR	Thread	x2APIC Logical Destination register (R/O)
80FH	2063	IA32_X2APIC_SIVR	Thread	x2APIC Spurious Interrupt Vector register (R/W)
810H	2064	IA32_X2APIC_ISR0	Thread	x2APIC In-Service register bits [31:0] (R/O)
811H	2065	IA32_X2APIC_ISR1	Thread	x2APIC In-Service register bits [63:32] (R/O)
812H	2066	IA32_X2APIC_ISR2	Thread	x2APIC In-Service register bits [95:64] (R/O)

813H	2067	IA32_X2APIC_ISR 3	Thread	x2APIC In-Service register bits [127:96] (R/O)
814H	2068	IA32_X2APIC_ISR 4	Thread	x2APIC In-Service register bits [159:128] (R/O)
815H	2069	IA32_X2APIC_ISR 5	Thread	x2APIC In-Service register bits [191:160] (R/O)
816H	2070	IA32_X2APIC_ISR 6	Thread	x2APIC In-Service register bits [223:192] (R/O)
817H	2071	IA32_X2APIC_ISR 7	Thread	x2APIC In-Service register bits [255:224] (R/O)
818H	2072	IA32_X2APIC_TMR0	Thread	x2APIC Trigger Mode register bits [31:0] (R/O)
819H	2073	IA32_X2APIC_TMR1	Thread	x2APIC Trigger Mode register bits [63:32] (R/O)
81AH	2074	IA32_X2APIC_TMR2	Thread	x2APIC Trigger Mode register bits [95:64] (R/O)
81BH	2075	IA32_X2APIC_TMR3	Thread	x2APIC Trigger Mode register bits [127:96] (R/O)
81CH	2076	IA32_X2APIC_TMR4	Thread	x2APIC Trigger Mode register bits [159:128] (R/O)
81DH	2077	IA32_X2APIC_TMR5	Thread	x2APIC Trigger Mode register bits [191:160] (R/O)
81EH	2078	IA32_X2APIC_TMR6	Thread	x2APIC Trigger Mode register bits [223:192] (R/O)
81FH	2079	IA32_X2APIC_TMR7	Thread	x2APIC Trigger Mode register bits [255:224] (R/O)
820H	2080	IA32_X2APIC_IRR 0	Thread	x2APIC Interrupt Request register bits [31:0] (R/O)
821H	2081	IA32_X2APIC_IRR 1	Thread	x2APIC Interrupt Request register bits [63:32] (R/O)
822H	2082	IA32_X2APIC_IRR 2	Thread	x2APIC Interrupt Request register bits [95:64] (R/O)
823H	2083	IA32_X2APIC_IRR 3	Thread	x2APIC Interrupt Request register bits [127:96] (R/O)
824H	2084	IA32_X2APIC_IRR 4	Thread	x2APIC Interrupt Request register bits [159:128] (R/O)
825H	2085	IA32_X2APIC_IRR 5	Thread	x2APIC Interrupt Request register bits [191:160] (R/O)
826H	2086	IA32_X2APIC_IRR 6	Thread	x2APIC Interrupt Request register bits [223:192] (R/O)
827H	2087	IA32_X2APIC_IRR 7	Thread	x2APIC Interrupt Request register bits [255:224] (R/O)
828H	2088	IA32_X2APIC_ESR	Thread	x2APIC Error Status register (R/W)

82FH	2095	IA32_X2APIC_LVT_CMCI	Thread	x2APIC LVT Corrected Machine Check Interrupt register (R/W)
830H	2096	IA32_X2APIC_ICR	Thread	x2APIC Interrupt Command register (R/W)
832H	2098	IA32_X2APIC_LVT_TIMER	Thread	x2APIC LVT Timer Interrupt register (R/W)
833H	2099	IA32_X2APIC_LVT_THERMAL	Thread	x2APIC LVT Thermal Sensor Interrupt register (R/W)
834H	2100	IA32_X2APIC_LVT_PMI	Thread	x2APIC LVT Performance Monitor register (R/W)
835H	2101	IA32_X2APIC_LVT_LINT0	Thread	x2APIC LVT LINT0 register (R/W)
836H	2102	IA32_X2APIC_LVT_LINT1	Thread	x2APIC LVT LINT1 register (R/W)
837H	2103	IA32_X2APIC_LVT_ERROR	Thread	x2APIC LVT Error register (R/W)
838H	2104	IA32_X2APIC_INIT_COUNT	Thread	x2APIC Initial Count register (R/W)
839H	2105	IA32_X2APIC_CUR_COUNT	Thread	x2APIC Current Count register (R/O)
83EH	2110	IA32_X2APIC_DIV_CONF	Thread	x2APIC Divide Configuration register (R/W)
83FH	2111	IA32_X2APIC_SELF_IPI	Thread	x2APIC Self IPI register (W/O)

...

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			
...				
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	**C-State Configuration Control** (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

		2:0		**Package C-State limit. (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-sate support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		**Reserved.**
		10		**I/O MWAIT Redirection Enable. (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		**Reserved.**
		15		**CFG Lock. (R/WO)** When set, lock bits 15:0 of this register until next reset
		24:16		**Reserved.**
		25		**C3 state auto demotion enable. (R/W)** When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information
		26		**C1 state auto demotion enable. (R/W)** When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information
		27		**Enable C3 undemotion (R/W)** When set, enables undemotion from demoted C3
		28		**Enable C1 undemotion (R/W)** When set, enables undemotion from demoted C1
		63:29		Reserved.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Core	**Power Management IO Redirection in C-state** (R/W)

		15:0		**LVL_2 Base Address. (R/W)** Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software
		18:16		**C-state Range. (R/W)** Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PMG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include 001b - C6 is the max C-State to include 010b - C7 is the max C-State to include
		63:19		Reserved.
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	see Table Table B-2.
1B2H	434	IA32_PACKAGE_THERM_INTERRUPT	Package	see Table B-2.
1FCH	508	POWER_CTL	Core	Power Control Register
280H	640	IA32_MC0_CTL2	Core	see Table B-2.
281H	641	IA32_MC1_CTL2	Core	see Table B-2.
282H	642	IA32_MC2_CTL2	Core	see Table B-2.
283H	643	IA32_MC3_CTL2	Core	see Table B-2.
284H	644	MSR_MC4_CTL2	Package	Always 0 (CMCI not supported)
400H	1024	IA32_MC0_CTL	Core	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
401H	1025	IA32_MC0_STATUS	Core	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
402H	1026	IA32_MC0_ADDR	Core	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
403H	1027	IA32_MC0_MISC	Core	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
404H	1028	IA32_MC1_CTL	Core	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
405H	1029	IA32_MC1_STATUS	Core	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
406H	1030	IA32_MC1_ADDR	Core	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
407H	1031	IA32_MC1_MISC	Core	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
409H	1033	IA32_MC2_STATUS	Core	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
40AH	1034	IA32_MC2_ADDR	Core	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
40BH	1035	IA32_MC2_MISC	Core	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”

40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
40DH	1037	IA32_MC3_ STATUS	Core	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
40FH	1039	IA32_MC3_MISC	Core	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
410H	1040	MSR_MC4_CTL	Core	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
		0		**PCU Hardware Error. (R/W)** When set, enables signaling of PCU hardware detected errors.
		1		**PCU Controller Error. (R/W)** When set, enables signaling of PCU controller detected errors
		2		**PCU Firmware Error. (R/W)** When set, enables signaling of PCU firmware detected errors
		63:2		Reserved.
411H	1041	IA32_MC4_ STATUS	Core	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
606H	1542	MSR_RAPL_POWE R_UNIT	Package	**Unit Multipliers used in RAPL Interfaces** (R/O) See Section 14.7.1, “RAPL Interfaces.”
60AH	1546	MSR_PKGC3_IRTL	Package	**Package C3 Interrupt Response Limit** (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit. (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C3 state.
		12:10		**Time Unit. (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		**Reserved.**
		15		**Valid. (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.

		63:16		Reserved.
60BH	1547	MSR_PKGC6_IRTL	Package	**Package C6 Interrupt Response Limit** (R/W) This MSR defines the budget allocated for the package to exit from C6 to a C0 state, where interrupt request can be delivered to the core and serviced. Additional core-exit latency amy be applicable depending on the actual C-state the core is in. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit. (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C6 state.
		12:10		**Time Unit. (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		**Reserved.**
		15		**Valid. (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
60CH	1548	MSR_PKGC7_IRTL	Package	**Package C7 Interrupt Response Limit** (R/W) This MSR defines the budget allocated for the package to exit from C7 to a C0 state, where interrupt request can be delivered to the core and serviced. Additional core-exit latency amy be applicable depending on the actual C-state the core is in. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit. (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C7 state.

		12:10		**Time Unit. (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		**Reserved.**
		15		**Valid. (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
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.”
6E0H	1760	IA32_TSC_DEADLINE	Thread	See Table Table B-2.

...

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			
285H	645	IA32_MC5_CTL2	Package	see Table Table B-2.

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
286H	646	IA32_MC6_CTL2	Package	see Table Table B-2.
287H	647	IA32_MC7_CTL2	Package	see Table Table B-2.
288H	648	IA32_MC8_CTL2	Package	see Table Table B-2.
289H	649	IA32_MC9_CTL2	Package	see Table Table B-2.
28AH	650	IA32_MC10_CTL2	Package	see Table Table B-2.
28BH	651	IA32_MC11_CTL2	Package	see Table Table B-2.
28CH	652	IA32_MC12_CTL2	Package	see Table Table B-2.
28DH	653	IA32_MC13_CTL2	Package	see Table Table B-2.
28EH	654	IA32_MC14_CTL2	Package	see Table Table B-2.
28FH	655	IA32_MC15_CTL2	Package	see Table Table B-2.
290H	656	IA32_MC16_CTL2	Package	see Table Table B-2.
291H	657	IA32_MC17_CTL2	Package	see Table Table B-2.
292H	658	IA32_MC18_CTL2	Package	see Table Table B-2.
293H	659	IA32_MC19_CTL2	Package	see Table Table B-2.
414H	1044	MSR_MC5_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
415H	1045	MSR_MC5_ STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
416H	1046	MSR_MC5_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
417H	1047	MSR_MC5_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
418H	1048	MSR_MC6_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
419H	1049	MSR_MC6_ STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
41AH	1050	MSR_MC6_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
41BH	1051	MSR_MC6_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
41CH	1052	MSR_MC7_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
41DH	1053	MSR_MC7_ STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
41EH	1054	MSR_MC7_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
41FH	1055	MSR_MC7_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
420H	1056	MSR_MC8_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
421H	1057	MSR_MC8_ STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
422H	1058	MSR_MC8_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
423H	1059	MSR_MC8_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
424H	1060	MSR_MC9_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
425H	1061	MSR_MC9_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
426H	1062	MSR_MC9_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
427H	1063	MSR_MC9_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
428H	1064	MSR_MC10_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
429H	1065	MSR_MC10_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
42AH	1066	MSR_MC10_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
42BH	1067	MSR_MC10_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
42CH	1068	MSR_MC11_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
42DH	1069	MSR_MC11_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
42EH	1070	MSR_MC11_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
42FH	1071	MSR_MC11_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
430H	1072	MSR_MC12_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
431H	1073	MSR_MC12_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
432H	1074	MSR_MC12_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
433H	1075	MSR_MC12_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
434H	1076	MSR_MC13_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
435H	1077	MSR_MC13_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
436H	1078	MSR_MC13_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
437H	1079	MSR_MC13_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
438H	1080	MSR_MC14_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
439H	1081	MSR_MC14_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
43AH	1082	MSR_MC14_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
43BH	1083	MSR_MC14_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
43CH	1084	MSR_MC15_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
43DH	1085	MSR_MC15_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
43EH	1086	MSR_MC15_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
43FH	1087	MSR_MC15_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
440H	1088	MSR_MC16_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
441H	1089	MSR_MC16_STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
442H	1090	MSR_MC16_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
443H	1091	MSR_MC16_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
444H	1092	MSR_MC17_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
445H	1093	MSR_MC17_ STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
446H	1094	MSR_MC17_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
447H	1095	MSR_MC17_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
448H	1096	MSR_MC18_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
449H	1097	MSR_MC18_ STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
44AH	1098	MSR_MC18_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
44BH	1099	MSR_MC18_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
44CH	1100	MSR_MC19_CTL	Package	See Section 15.3.2.1, “IA32_MCi_CTL MSRs.”
44DH	1101	MSR_MC19_ STATUS	Package	See Section 15.3.2.2, “IA32_MCi_STATUS MSRS.” and Appendix E.
44EH	1102	MSR_MC19_ADDR	Package	See Section 15.3.2.3, “IA32_MCi_ADDR MSRs.”
44FH	1103	MSR_MC19_MISC	Package	See Section 15.3.2.4, “IA32_MCi_MISC MSRs.”
618H	1560	MSR_DRAM_POWE R_LIMIT	Package	**DRAM RAPL Power Limit Control** (R/W) See Section 14.7.5, “DRAM RAPL Domain.”
619H	1561	MSR_DRAM_ENER Y_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_POWE R_INFO	Package	**DRAM RAPL Parameters** (R/W) See Section 14.7.5, “DRAM RAPL Domain.”

...

Table B-15. MSRs in Intel Core Solo, Intel Core Duo Processors, and Dual-Core Intel Xeon Processor LV

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
...				
1C9H	457	MSR_ LASTBRANCH_ TOS	Unique	**Last Branch Record Stack TOS. (R)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H)
...				

...

Table B-16. MSRs in Pentium M Processors

Register Address		Register Name	Bit Description
Hex	Dec		
...			
1C9H	457	MSR_LASTBRANCH_TOS	**Last Branch Record Stack TOS. (R)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See also: ▪ MSR_LASTBRANCH_0_FROM_IP (at 40H) ▪ Section 16.9, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)"
...			

...

22. 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.

...

E.4 INCREMENTAL DECODING INFORMATION: PROCESSOR FAMILY WITH CPUID DISPLAYFAMILY_DISPLAYMODEL SIGNATURE 06_2DH, MACHINE ERROR CODES FOR MACHINE CHECK

Table E-8 through Table E-12 provide information for interpreting additional model-specific fields for memory controller errors relating to the processor family with CPUID DisplayFamily_DisplaySignature 06_2DH, which supports Intel QuickPath Interconnect links. Incremental MC error codes related to the Intel QPI links are reported in the register banks IA32_MC6 and IA32_MC7, incremental error codes for internal machine check error from PCU controller is reported in the register bank IA32_MC4, and incremental error codes for the memory controller unit is reported in the register banks IA32_MC8-IA32_MC11.

E.4.1 Internal Machine Check Errors

Table E-13. Machine Check Error codes for IA32_MC4_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA error codes[1]	0-15	MCACOD	
Model specific errors	19:16	Reserved except for the following	0000b - No Error 0001b - Non_IMem_Sel 0010b - I_Parity_Error 0011b - Bad_OpCode 0100b - I_Stack_Underflow 0101b - I_Stack_Overflow 0110b - D_Stack_Underflow 0111b - D_Stack_Overflow 1000b - Non-DMem_Sel 1001b - D_Parity_Error
	23-20	Reserved	Reserved
	31-24	Reserved except for the following	00h - No Error 20h - MC_RCLK_PLL_LOCK_TIMEOUT 21h - MC_PCIE_PLL_LOCK_TIMEOT 22h - MC_BOOT_VID_SET_TIMEOUT 23h - MC_BOOT_FREQUENCY_SET_TIMEOUT 24h - MC_START_IA_CORES_TIMEOUT 26h - MC_PCIE_RCOMP_TIMEOUT 27h - MC_PMA_DNS_COMMAND_TIMEOUT 28h - MC_MESSAGE_CHANNEL_TIMEOUT 29h - MC_GVFSM_BGF_PROGRAM_TIMEOUT 2Ah - MC_MC_PLL_LOCK_TIMEOUT 2Bh - MC_MS_BGF_PROGRAM_TIMEOUT
	56-32	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.

E.4.2 Intel QPI Machine Check Errors

Table E-14. Intel QPI MC Error codes for IA32_MC6_STATUS and IA32_MC7_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA error codes[1]	0-15	MCACOD	Bus error format: 1PPTRRRRIILL
Model specific errors			
	56-16	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.

E.4.3 Integrated Memory Controller Machine Check Errors

MC error codes associated with integrated memory controllers are reported in the MSRs IA32_MC8_STATUS-IA32_MC11_STATUS. The supported error codes are follows the architectural MCACOD definition type 1MMMCCCC (see Chapter 15, "Machine-Check Architecture,").

...

23. Updates to Appendix G, Volume 3B

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

...

G.3.1 Pin-Based VM-Execution Controls

The IA32_VMX_PINBASED_CTLS MSR (index 481H) reports on the allowed settings of **most** of the pin-based VM-execution controls (see Section 21.6.1):

- Bits 31:0 indicate the **allowed 0-settings** of these controls. VM entry allows control X (bit X of the pin-based VM-execution controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.

  Exceptions are made for the pin-based VM-execution controls in the default1 class (see Appendix G.2). These are bits 1, 2, and 4; the corresponding bits of the IA32_VMX_PINBASED_CTLS MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32_VMX_BASIC MSR:

  — If bit 55 in the IA32_VMX_BASIC MSR is read as 0, VM entry fails if any pin-based VM-execution control in the default1 class is 0.

  - — If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_PINBASED_CTLS MSR (see below) reports which of the pin-based VM-execution controls in the default1 class can be 0 on VM entry.
- Bits 63:32 indicate the **allowed 1-settings** of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_PINBASED_CTLS MSR (index 48DH) reports on the allowed settings of **all** of the pin-based VM-execution controls:

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the pin-based VM-execution controls:

- If bit 55 in the IA32_VMX_BASIC MSR is read as 0, all information about the allowed settings of the pin-based VM-execution controls is contained in the IA32_VMX_PINBASED_CTLS MSR. (The IA32_VMX_TRUE_PINBASED_CTLS MSR is not supported.)
- If bit 55 in the IA32_VMX_BASIC MSR is read as 1, all information about the allowed settings of the pin-based VM-execution controls is contained in the IA32_VMX_TRUE_PINBASED_CTLS MSR. Assuming that software knows that the default1 class of pin-based VM-execution controls contains bits 1, 2, and 4, there is no need for software to consult the IA32_VMX_PINBASED_CTLS MSR.

G.3.2 Primary Processor-Based VM-Execution Controls

The IA32_VMX_PROCBASED_CTLS MSR (index 482H) reports on the allowed settings of **most** of the primary processor-based VM-execution controls (see Section 21.6.2):

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X (bit X of the primary processor-based VM-execution controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.

  Exceptions are made for the primary processor-based VM-execution controls in the default1 class (see Appendix G.2). These are bits 1, 4–6, 8, 13–16, and 26; the corresponding bits of the IA32_VMX_PROCBASED_CTLS MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32_VMX_BASIC MSR:

  - — If bit 55 in the IA32_VMX_BASIC MSR is read as 0, VM entry fails if any of the primary processor-based VM-execution controls in the default1 class is 0.
  - — If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_PROCBASED_CTLS MSR (see below) reports which of the primary processor-based VM-execution controls in the default1 class can be 0 on VM entry.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

If bit 55 in the IA32_VMX_BASIC MSR is read as 1,
the IA32_VMX_TRUE_PROCBASED_CTLS MSR (index 48EH) reports on the allowed settings of **all** of the primary processor-based VM-execution controls:

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the primary processor-based VM-execution controls:

- If bit 55 in the IA32_VMX_BASIC MSR is read as 0, all information about the allowed settings of the primary processor-based VM-execution controls is contained in the IA32_VMX_PROCBASED_CTLS MSR. (The IA32_VMX_TRUE_PROCBASED_CTLS MSR is not supported.)
- If bit 55 in the IA32_VMX_BASIC MSR is read as 1, all information about the allowed settings of the processor-based VM-execution controls is contained in the IA32_VMX_TRUE_PROCBASED_CTLS MSR. Assuming that software knows that the default1 class of processor-based VM-execution controls contains bits 1, 4–6, 8, 13–16, and 26, there is no need for software to consult the IA32_VMX_PROCBASED_CTLS MSR.

G.3.3 Secondary Processor-Based VM-Execution Controls

The IA32_VMX_PROCBASED_CTLS2 MSR (index 48BH) reports on the allowed settings of the secondary processor-based VM-execution controls (see Section 21.6.2).
VM entries perform the following checks:

- Bits 31:0 indicate the allowed 0-settings of these controls. These bits are always 0. This fact indicates that VM entry allows each bit of the secondary processor-based VM-execution controls to be 0.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X (bit X of the secondary processor-based VM-execution controls) to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X and the “activate secondary controls” primary processor-based VM-execution control are both 1.

The IA32_VMX_PROCBASED_CTLS2 MSR exists only on processors that support the 1-setting of the “activate secondary controls” VM-execution control (only if bit 63 of the IA32_VMX_PROCBASED_CTLS MSR is 1).

G.4 VM-EXIT CONTROLS

The IA32_VMX_EXIT_CTLS MSR (index 483H) reports on the allowed settings of **most** of the VM-exit controls (see Section 21.7.1):

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X (bit X of the VM-exit controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.

  Exceptions are made for the VM-exit controls in the default1 class (see Appendix G.2). These are bits 0–8, 10, 11, 13, 14, 16, and 17; the corresponding bits of the

IA32_VMX_EXIT_CTLS MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32_VMX_BASIC MSR:

— If bit 55 in the IA32_VMX_BASIC MSR is read as 0, VM entry fails if any VM-exit control in the default1 class is 0.

— If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_EXIT_CTLS MSR (see below) reports which of the VM-exit controls in the default1 class can be 0 on VM entry.

- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control 32+X to be 1 if bit X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_EXIT_CTLS MSR (index 48FH) reports on the allowed settings of **all** of the VM-exit controls:

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the VM-exit controls:

- If bit 55 in the IA32_VMX_BASIC MSR is read as 0, all information about the allowed settings of the VM-exit controls is contained in the IA32_VMX_EXIT_CTLS MSR. (The IA32_VMX_TRUE_EXIT_CTLS MSR is not supported.)
- If bit 55 in the IA32_VMX_BASIC MSR is read as 1, all information about the allowed settings of the VM-exit controls is contained in the IA32_VMX_TRUE_EXIT_CTLS MSR. Assuming that software knows that the default1 class of VM-exit controls contains bits 0–8, 10, 11, 13, 14, 16, and 17, there is no need for software to consult the IA32_VMX_EXIT_CTLS MSR.

G.5 VM-ENTRY CONTROLS

The IA32_VMX_ENTRY_CTLS MSR (index 484H) reports on the allowed settings of **most** of the VM-entry controls (see Section 21.8.1):

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X (bit X of the VM-entry controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.

  Exceptions are made for the VM-entry controls in the default1 class (see Appendix G.2). These are bits 0–8 and 12; the corresponding bits of the IA32_VMX_ENTRY_CTLS MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32_VMX_BASIC MSR:

  — If bit 55 in the IA32_VMX_BASIC MSR is read as 0, VM entry fails if any VM-entry control in the default1 class is 0.

  — If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_ENTRY_CTLS MSR (see below) reports which of the VM-entry controls in the default1 class can be 0 on VM entry.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry fails if bit X is 1 in the VM-entry controls and bit 32+X is 0 in this MSR.

If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_ENTRY_CTLS MSR (index 490H) reports on the allowed settings of **all** of the VM-entry controls:

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control 32+X to be 1 if bit X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the VM-entry controls:

- If bit 55 in the IA32_VMX_BASIC MSR is read as 0, all information about the allowed settings of the VM-entry controls is contained in the IA32_VMX_ENTRY_CTLS MSR. (The IA32_VMX_TRUE_ENTRY_CTLS MSR is not supported.)
- If bit 55 in the IA32_VMX_BASIC MSR is read as 1, all information about the allowed settings of the VM-entry controls is contained in the IA32_VMX_TRUE_ENTRY_CTLS MSR. Assuming that software knows that the default1 class of VM-entry controls contains bits 0–8 and 12, there is no need for software to consult the IA32_VMX_ENTRY_CTLS MSR.

...