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

## Documentation Changes

*March 2005*

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

Document Number: 252046-012

# Contents

# Revision History

| Version | Description | Date |
|---|---|---|
| -001 | • Initial Release | November 2002 |
| -002 | • Added 1-10 Documentation Changes.<br>• Removed old Documentation Changes items that already have been incorporated in the published Software Developer's manual | December 2002 |
| -003 | • Added 9 -17 Documentation Changes.<br>• Removed Documentation Change #6 - References to bits Gen and Len Deleted.<br>• Removed Documentation Change #4 - VIF Information Added to CLI Discussion. | February 2003 |
| -004 | • Removed Documentation changes 1-17.<br>• Added Documentation changes 1-24. | June 2003 |
| -005 | • Removed Documentation Changes 1-24.<br>• Added Documentation Changes 1-15. | September 2003 |
| -006 | • Added Documentation Changes 16- 34. | November 2003 |
| -007 | • Updated Documentation changes 14, 16, 17, and 28.<br>• Added Documentation Changes 35-45. | January 2004 |
| -008 | • Removed Documentation Changes 1-45.<br>• Added Documentation Changes 1-5. | March 2004 |
| -009 | • Added Documentation Changes 7-27. | May 2004 |
| -010 | • Removed Documentation Changes 1-27.<br>• Added Documentation Changes 1. | August 2004 |
| -011 | • Added Documentation Changes 2-28. | November 2004 |
| -012 | • Removed Documentation Changes 1-28.<br>• Added Documentation Changes 1-16. | March 2005 |

# Preface

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

## Affected Documents/Related Documents

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

## Nomenclature

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

# Summary Table of Changes

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

## Codes Used in Summary Table

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

## Summary Table of Documentation Changes

| Number | Documentation Changes |
|---|---|
| 1 | CALL Instruction Pseudocode Updated |
| 2 | Information Updated for Fast System Call Instructions: SYSENTER, SYSEXIT, SYSCALL, SYSRET |
| 3 | A Note on the PAE Mechanism Has Been Updated |
| 4 | Exception and Interrupt Priority Table Updated |
| 5 | Exceptions for MWAIT Instruction Updated |
| 6 | More Information on the Time-stamp Counter |
| 7 | Section on Serializing Instructions Has Been Updated |
| 8 | LAHF/SAHF Enabled on Some 64-bit Steppings |
| 9 | Updated Information on Disabling the APIC |
| 10 | Updated ENTER Instruction Pseudocode |
| 11 | Updated Information on the Usage of MSR_FSB_ESCR0 |
| 12 | Updated Footnotes in MOV, POP, STI Descriptions |
| 13 | Description Updated for CVTPI2PD Instruction |
| 14 | ROR/ROL Pseudocode Has Been Updated |
| 15 | Syntax for CPUID, CR and MSR Information |
| 16 | Inclusion of Intel® Extended Memory 64 Technology |
| 17 | Family Representation Within CPUID Context Updated |

# Documentation Changes

## 1. CALL Instruction Pseudocode Updated

"CALL — Call Procedure" in Chapter 3 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 2A*, has been updated to correct the inconsistency of the limit checking for code segments.

The pseudocode provided below has been updated to correct this problem.

---

**Operation**

```
IF near call
    THEN IF near relative call
        THEN
            IF OperandSize = 32
                THEN
                    tempEIP ← EIP + DEST; (* DEST is rel32 *)
                    IF tempEIP is not within code segment limit THEN #GP(0); FI;
                    IF stack not large enough for a 4-byte return address
                        THEN #SS(0); FI;
                    Push(EIP);
                    EIP ← tempEIP;
                ELSE (* OperandSize = 16 *)
                    tempEIP ← (EIP + DEST) AND 0000FFFFH; (* DEST is rel16 *)
                    IF tempEIP is not within code segment limit THEN #GP(0); FI;
                    IF stack not large enough for a 2-byte return address
                        THEN #SS(0); FI;
                    Push(IP);
                    EIP ← tempEIP;
            FI;
        FI;
    ELSE (* Near absolute call *)
        IF OperandSize = 32
            THEN
                tempEIP ← DEST; (* DEST is r/m32 *)
                IF tempEIP is not within code segment limit THEN #GP(0); FI;
                IF stack not large enough for a 4-byte return address
                    THEN #SS(0); FI;
                Push(EIP);
                EIP ← tempEIP;
            ELSE (* OperandSize = 16 *)
                tempEIP ← DEST AND 0000FFFFH; (* DEST is r/m16 *)
                IF tempEIP is not within code segment limit THEN #GP(0); FI;
                IF stack not large enough for a 2-byte return address
                    THEN #SS(0); FI;
                Push(IP);
                EIP ← tempEIP;
        FI;
```

```
    FI;
FI;

IF far call and (PE = 0 or (PE = 1 and VM = 1)) (* Real-address or virtual-8086 mode *)
    THEN
        IF OperandSize = 32
            THEN
                IF stack not large enough for a 6-byte return address
                    THEN #SS(0); FI;
                IF DEST[31:16] is not zero THEN #GP(0); FI;
                Push(CS); (* Padded with 16 high-order bits *)
                Push(EIP);
                CS ← DEST[47:32]; (* DEST is ptr16:32 or [m16:32] *)
                EIP ← DEST[31:0]; (* DEST is ptr16:32 or [m16:32] *)
            ELSE (* OperandSize = 16 *)
                IF stack not large enough for a 4-byte return address
                    THEN #SS(0); FI;
                Push(CS);
                Push(IP);
                CS ← DEST[31:16]; (* DEST is ptr16:16 or [m16:16] *)
                EIP ← DEST[15:0]; (* DEST is ptr16:16 or [m16:16]; clear upper 16 bits *)
        FI;
FI;

IF far call and (PE = 1 and VM = 0) (* Protected mode or Long Mode, not virtual-8086 mode*)
    THEN
        IF segment selector in target operand NULL
            THEN #GP(0); FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new code segment selector); FI;
        Read type and access rights of selected 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 or
                a 64-bit 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;
FI;

CONFORMING-CODE-SEGMENT:
    IF EFER.LMA = 1 and new code-segment descriptor has L=D=1
        THEN GP(new code segment selector); FI;
    IF DPL > CPL
        THEN #GP(new code segment selector); FI;
    IF segment not present
        THEN #NP(new code segment selector); FI;
    IF stack not large enough for return address
        THEN #SS(0); FI;
    tempEIP ← DEST(Offset);
    IF OperandSize = 16
        THEN
```

```
            tempEIP ← tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
    IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
    segment limit)
        THEN #GP(0); FI;
    IF tempEIP is non-canonical
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            Push(CS); (* Padded with 16 high-order bits *)
            Push(EIP);
            CS ← DEST(CodeSegmentSelector);
            (* Segment descriptor information also loaded *)
            CS(RPL) ← CPL;
            EIP ← tempEIP;
        ELSE
            IF OperandSize = 16
                THEN
                    Push(CS);
                    Push(IP);
                    CS ← DEST(CodeSegmentSelector);
                    (* Segment descriptor information also loaded *)
                    CS(RPL) ← CPL;
                    EIP ← tempEIP;
                ELSE (* OperandSize = 64 *)
                    Push(CS); (* Padded with 48 high-order bits *)
                    Push(RIP);
                    CS ← DEST(CodeSegmentSelector);
                    (* Segment descriptor information also loaded *)
                    CS(RPL) ← CPL;
                    RIP ← tempEIP;
            FI;
    FI;
END;

NONCONFORMING-CODE-SEGMENT:
    IF EFER.LMA = 1 and new code-segment descriptor has L=D=1
        THEN GP(new code segment selector); FI;
    IF (RPL > CPL) or (DPL ≠ CPL)
        THEN #GP(new code segment selector); FI;
    IF segment not present
        THEN #NP(new code segment selector); FI;
    IF stack not large enough for return address
        THEN #SS(0); FI;
    tempEIP ← DEST(Offset);
    IF OperandSize = 16
        THEN tempEIP ← tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
    IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
    segment limit)
        THEN #GP(0); FI;
    IF tempEIP is non-canonical
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            Push(CS); (* Padded with 16 high-order bits *)
            Push(EIP);
            CS ← DEST(CodeSegmentSelector);
            (* Segment descriptor information also loaded *)
            CS(RPL) ← CPL;
            EIP ← tempEIP;
        ELSE
            IF OperandSize = 16
                THEN
                    Push(CS);
                    Push(IP);
                    CS ← DEST(CodeSegmentSelector);
```

```
                        (* Segment descriptor information also loaded *)
                        CS(RPL) ← CPL;
                        EIP ← tempEIP;
                    ELSE (* OperandSize = 64 *)
                        Push(CS); (* Padded with 48 high-order bits *)
                        Push(RIP);
                        CS ← DEST(CodeSegmentSelector);
                        (* Segment descriptor information also loaded *)
                        CS(RPL) ← CPL;
                        RIP ← tempEIP;
                FI;
    FI;
END;

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(code segment selector); FI;
    Read code segment descriptor;
    IF code-segment segment descriptor does not indicate a code segment
    or code-segment segment descriptor DPL > CPL
        THEN #GP(code segment selector); FI;
    IF EFER.LMA = 1 AND (code-segment segment descriptor is not a 64-bit code segment
    or code-segment descriptor has both L-Bit and D-bit set)
        THEN #GP(code segment selector); FI;
    IF code segment not present
        THEN #NP(new code segment selector); FI;
    IF 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 TSS
        THEN
            TSSstackAddress ← new code segment (DPL ∗ 8) + 4;
            IF (TSSstackAddress + 7) > TSS limit
                THEN #TS(current TSS selector); FI;
            newSS ← TSSstackAddress + 4;
            newESP ← stack address;
        ELSE
            IF current TSS is 16-bit TSS
                THEN
                    TSSstackAddress ← new code segment (DPL ∗ 4) + 2;
                    IF (TSSstackAddress + 4) > TSS limit
                        THEN #TS(current TSS selector); FI;
                    newESP ← TSSstackAddress;
                    newSS ← TSSstackAddress + 2;
                ELSE (* TSS is 64-bit *)
                    TSSstackAddress ← new code segment (DPL ∗ 8) + 4;
                    IF (TSSstackAddress + 8) > TSS limit
```

```
                        THEN #TS(current TSS selector); FI;
                    newESP ← TSSstackAddress;
                    newSS ← NULL;
            FI;
FI;
IF EFER.LMA = 0 and stack segment selector = NULL
    THEN #TS(stack segment selector); FI;
Read code segment descriptor;
IF EFER.LMA = 0 and (stack segment selector's RPL ≠ DPL of code segment
or stack segment DPL ≠ DPL of code segment or stack segment is not a
writable data segment)
    THEN #TS(SS selector); FI
IF EFER.LMA = 0 and stack segment not present
    THEN #SS(SS selector); FI;
IF CallGateSize = 32
    THEN
        IF stack does not have room for parameters plus 16 bytes
            THEN #SS(SS selector); FI;
        IF CallGate(InstructionPointer) not within 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 stack does not have room for parameters plus 8 bytes
                    THEN #SS(SS selector); FI;
                IF (CallGate(InstructionPointer) AND FFFFH) not in 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 touches non-canonical addresses
                    THEN #SS(SS selector); FI;
                IF (CallGate(InstructionPointer) is non-canonical)
                    THEN #GP(0); FI;
                SS ← newSS; (* New SS 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;

SAME-PRIVILEGE:
    IF CallGateSize = 32
        THEN
            IF stack does not have room for 8 bytes
                THEN #SS(0); FI;
            IF CallGate(InstructionPointer) not within code segment limit
                THEN #GP(0); FI;
            CS:EIP ← CallGate(CS:EIP) (* Segment descriptor information also loaded *)
            Push(oldCS:oldEIP); (* Return address to calling procedure *)
        ELSE
            If CallGateSize = 16
                THEN
                    IF stack does not have room for 4 bytes
                        THEN #SS(0); FI;
                    IF CallGate(InstructionPointer) not within code segment limit
                        THEN #GP(0); FI;
                    CS:IP ← CallGate(CS:instruction pointer);
                    (* Segment descriptor information also loaded *)
                    Push(oldCS:oldIP); (* Return address to calling procedure *)
                ELSE (* CallGateSize = 64)
                    IF pushing 16 bytes on the stack touches non-canonical addresses
                        THEN #SS(0); FI;
                    IF RIP non-canonical
                        THEN #GP(0); FI;
                    CS:IP ← CallGate(CS:instruction pointer);
                    (* Segment descriptor information also loaded *)
                    Push(oldCS:oldIP); (* Return address to calling procedure *)
            FI;
    FI;
    CS(RPL) ← CPL
END;

TASK-GATE:
    IF task gate DPL < CPL or RPL
        THEN #GP(task gate selector); FI;
    IF task gate not present
        THEN #NP(task gate selector); FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
    or index not within GDT limits
        THEN #GP(TSS selector); FI;
    Access TSS descriptor in GDT;

    IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
        THEN #GP(TSS selector); FI;
    IF TSS not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
```

```
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;

TASK-STATE-SEGMENT:
    IF TSS DPL < CPL or RPL
    or TSS descriptor indicates TSS not available
        THEN #GP(TSS selector); FI;
    IF TSS is not present
        THEN #NP(TSS selector);  FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;
```

## 2. Information Updated for Fast System Call Instructions: SYSENTER, SYSEXIT, SYSCALL, SYSRET

SYSENTER and SYSEXIT sections covered in Chapter 4, *IA-32 Intel Architecture Software Developer's Manual, Volume 2B*, and SYSCALL and SYSRET sections covered in Chapter 3, *Intel® Extended Memory 64 Technology Software Developer's Guide, Volume 2*, have been updated and are now covered in *IA-32 Intel Architecture Software Developer's Manual, Volume 2B* (rev. 015).

Updated sections are reprinted below.

---

## SYSENTER—Fast System Call

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 34 | SYSENTER | Valid | Valid | Fast call to privilege level 0 system procedures. |

### Description

Executes a fast call to a level 0 system procedure or routine. SYSENTER is a companion instruction to SYSEXIT. The instruction is optimized to provide the maximum performance for system calls from user code running at privilege level 3 to operating system or executive procedures running at privilege level 0.

Prior to executing the SYSENTER instruction, software must specify the privilege level 0 code segment and code entry point, and the privilege level 0 stack segment and stack pointer by writing values to the following MSRs:

- IA32_SYSENTER_CS — Contains the 32-bit segment selector for the privilege level 0 code segment. This value is also used to compute the segment selector of the privilege level 0 stack segment.
- IA32_SYSENTER_EIP — Contains the 32-bit offset into the privilege level 0 code segment to the first instruction of the selected operating procedure or routine.
- IA32_SYSENTER_ESP — Contains the 32-bit stack pointer for the privilege level 0 stack.

These MSRs can be read from and written to using RDMSR/WRMSR. Register addresses are listed in Table 7-6. The addresses are defined to remain fixed for future IA-32 processors.

| MSR | Address |
|---|---|
| IA32_SYSENTER_CS | 174H |
| IA32_SYSENTER_ESP | 175H |
| IA32_SYSENTER_EIP | 176H |

When SYSENTER is executed, the processor:

1. Loads the segment selector from the IA32_SYSENTER_CS into the CS register.
2. Loads the instruction pointer from the IA32_SYSENTER_EIP into the EIP register.
3. Adds 8 to the value in IA32_SYSENTER_CS and loads it into the SS register.
4. Loads the stack pointer from the IA32_SYSENTER_ESP into the ESP register.
5. Switches to privilege level 0.
6. Clears the VM flag in the EFLAGS register, if the flag is set.
7. Begins executing the selected system procedure.

The processor does not save a return IP or other state information for the calling procedure.

The SYSENTER instruction always transfers program control to a protected-mode code segment with a DPL of 0. The instruction requires that the following conditions are met by the operating system:

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

The SYSENTER can be invoked from all operating modes except real-address mode.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. When executing a SYSENTER instruction, the processor does not save state information for the user code, and neither the SYSENTER nor the SYSEXIT instruction supports passing parameters on the stack.

To use the SYSENTER and SYSEXIT instructions as companion instructions for transitions between privilege level 3 code and privilege level 0 operating system procedures, the following conventions must be followed:

- The segment descriptors for the privilege level 0 code and stack segments and for the privilege level 3 code and stack segments must be contiguous in the global descriptor table. This convention allows the processor to compute the segment selectors from the value entered in the SYSENTER_CS_MSR MSR.
- The fast system call “stub” routines executed by user code (typically in shared libraries or DLLs) must save the required return IP and processor state information if a return to the calling procedure is required. Likewise, the operating system or executive procedures called with SYSENTER instructions must have access to and use this saved return and state information when returning to the user code.

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

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

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

### Operation

```
IF CR0.PE = 0 THEN #GP(0); FI;
IF SYSENTER_CS_MSR = 0 THEN #GP(0); FI;
EFLAGS.VM ← 0;                          (* Insures 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.CPL ← 0;
CS.BASE ← 0;                            (* Flat segment *)
CS.LIMIT ← FFFFH;                       (* 4-GByte limit *)
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.RPL ← 0;
CS.ARbyte.P ← 1;

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

ESP ← SYSENTER_ESP_MSR;
EIP ← SYSENTER_EIP_MSR;
```

### IA-32e Mode Operation

In IA-32e mode, SYSENTER executes a fast system calls from user code running at privilege level 3 (in compatibility mode or 64-bit mode) to 64-bit executive procedures running at privilege level 0. This instruction is a companion instruction to the SYSEXIT instruction.

In IA-32e mode, the IA32_SYSENTER_EIP and IA32_SYSENTER_ESP MSRs hold 64-bit addresses and must be in canonical form; IA32_SYSENTER_CS must not contain a NULL selector.

When SYSENTER transfers control, the following fields are generated and bits set:

- **Target code segment** — Reads non-NULL selector from IA32_SYSENTER_CS.
- **New CS attributes** — G-bit = 1, D-bit = 0, P-bit = 1, DPL = 0, CS_type = ER non-conforming code, L-bit = 1 (go to 64-bit mode); also CS base = 0, CS limit = FFFFFFFFH.
- **Target instruction** — Reads 64-bit canonical address from IA32_SYSENTER_EIP.
- **Stack segment** — Computed by adding 8 to the value from IA32_SYSENTER_CS.
- **Stack pointer** — Reads 64-bit canonical address from IA32_SYSENTER_ESP.
- **New SS attributes** — G-bit = 1, D-bit = 1, P-bit = 1, DPL = 0, SS_type = RW expand-up area; also SS base = 0, SS limit = FFFFFFFFH.

### Flags Affected

VM, IF, RF (see Operation above)

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If IA32_SYSENTER_CS = 0. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If protected mode is not enabled. |

### Virtual-8086 Mode Exceptions

Same exceptions as in Protected Mode.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

Same exceptions as in Protected Mode.

## SYSEXIT—Fast Return from Fast System Call

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 35 | SYSEXIT | Valid | Valid | Fast return to privilege level 3 user code. |
| REX.W + 0F 35 | SYSEXIT | Valid | Valid | Fast return to 64-bit mode privilege level 3 user code. |

### Description

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

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

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

The IA32_SYSENTER_CS MSR can be read from and written to using RDMSR/WRMSR. The register address is listed in Table 7-6. This address is defined to remain fixed for future IA-32 processors.

When SYSEXIT is executed, the processor:

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

See “SWAPGS—Swap GS Base Register” for information about using the SYSENTER and SYSEXIT instructions as companion call and return instructions.

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

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

The SYSENTER can be invoked from all operating modes except real-address mode.

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

```
IF CPUID SEP bit is set
    THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
```

```
        THEN
            SYSENTER/SYSEXIT_Not_Supported; FI;
        ELSE
            SYSENTER/SYSEXIT_Supported; FI;
FI;
```

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

### Operation

```
IF SYSENTER_CS_MSR = 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.BASE ← 0;                        (* Flat segment *)
CS.LIMIT ← FFFFH;                   (* 4-GByte limit *)
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.RPL ← 3;
CS.ARbyte.P ← 1;

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

ESP ← ECX;
EIP← EDX;
```

### IA-32e Mode Operation

In IA-32e mode, SYSEXIT executes a fast system calls from a 64-bit executive procedures running at privilege level 0 to user code running at privilege level 3 (in compatibility mode or 64-bit mode). This instruction is a companion instruction to the SYSENTER instruction.

In IA-32e mode, the IA32_SYSENTER_EIP and IA32_SYSENTER_ESP MSRs hold 64-bit addresses and must be in canonical form; IA32_SYSENTER_CS must not contain a NULL selector.

When the SYSEXIT instruction transfers control to 64-bit mode user code using REX.W, the following fields are generated and bits set:

- **Target code segment** — Computed by adding 32 to the value in the IA32_SYSENTER_CS.

- **New CS attributes** — G-bit = 1, D-bit = 0, P-bit = 1, DPL = 3, CS_type = ER accessed non-conforming code, L-bit = 1 (go to 64-bit mode).
- **Target instruction** — Reads 64-bit canonical address in RDX.
- **Stack segment** — Computed by adding 8 to the value of CS selector.
- **New SS attributes** — G-bit = 1, D-bit = 1, P-bit = 1, DPL = 3, SS_type = RW expand-up area.
- **Stack pointer** — Update RSP using 64-bit canonical address in RCX.

When SYSEXIT transfers control to compatibility mode user code when the operand size attribute is 32 bits, the following fields are generated and bits set:

- **Target code segment** — Computed by adding 16 to the value in IA32_SYSENTER_CS.
- **New CS attributes** — G-bit = 1, D-bit = 1, P-bit = 1, DPL = 3, CS_type = ER accessed non-conforming code, L-bit = 0 (go to compatibility mode); also CS base = 0, CS limit = FFFFFFFFH.
- **Target instruction** — Fetch the target instruction from 32-bit address in EDX.
- **Stack segment** — Computed by adding 24 to the value in IA32_SYSENTER_CS.
- **New SS attributes** — G-bit = 1, D-bit = 1, P-bit = 1, DPL = 3, SS_type = RW expand-up area; also SS base = 0, SS limit = FFFFFFFFH.
- **Stack pointer** — Update ESP from 32-bit address in ECX.

### Flags Affected

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If IA32_SYSENTER_CS = 0. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If protected mode is not enabled. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If IA32_SYSENTER_CS = 0. |

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If IA32_SYSENTER_CS = 0. |
| | If CPL ≠ 0. |
| | If ECX or EDX contains a non-canonical address. |

## SYSCALL—Fast System Call

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 05 | SYSCALL | Valid | Invalid | Fast call to privilege level 0 system procedures. |

### Description

SYSCALL saves the RIP of the instruction following SYSCALL to RCX and loads a new RIP from the IA32_LSTAR (64-bit mode). Upon return, SYSRET copies the value saved in RCX to the RIP.

SYSCALL saves RFLAGS (lower 32 bit only) in R11. It then masks RFLAGS with an OS-defined value using the IA32_FMASK (MSR C000_0084). The actual mask value used by the OS is the complement of the value written to the IA32_FMASK MSR. None of the bits in RFLAGS are automatically cleared (except for RF). SYSRET restores RFLAGS from R11 (the lower 32 bits only).

Software should not alter the CS or SS descriptors in a manner that violates the following assumptions made by SYSCALL/SYSRET:

- The CS and SS base and limit remain the same for all processes, including the operating system (the base is 0H and the limit is 0FFFFFFFFH).
- The CS of the SYSCALL target has a privilege level of 0.
- The CS of the SYSRET target has a privilege level of 3.

SYSCALL/SYSRET do not check for violations of these assumptions.

### Operation

```
IF ((CS ≠ 1 ) or (IA32_EFER.SCE ≠ 1))
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
    THEN #UD; FI;
RCX ← RIP;
RIP ← LSTAR_MSR;
R11 ← EFLAGS;
EFLAGS ← (EFLAGS MASKED BY IA32_FMASK);
CPL ← 0;
CS(SEL) ← IA32_STAR_MSR[47:32];
CS(PL) ← 0;
CS(BASE) ← 0;
CS(LIMIT) ← 0xFFFFF;
CS(GRANULAR) ← 1;
SS(SEL) ← IA32_STAR_MSR[47:32] + 8;
SS(PL) ← 0;
SS(BASE) ← 0;
SS(LIMIT) ← 0xFFFFF;
SS(GRANULAR) ← 1;
```

### Flags Affected

All.

### Protected Mode Exceptions

#UD If Mode ≠ 64-bit.

### Real-Address Mode Exceptions

#UD Instruction is not recognized in this mode.

### Virtual-8086 Mode Exceptions

#UD Instruction is not recognized in this mode.

### Compatibility Mode Exceptions

#UD Instruction is not recognized in this mode.

### 64-Bit Mode Exceptions

#UD If IA32_EFER.SCE = 0.

## SYSRET—Return From Fast System Call

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 07 | SYSRET | Valid | Invalid | Return from fast system call |

### Description

SYSCALL saves the RIP of the instruction following the SYSCALL into RCX and loads the new RIP from the LSTAR (64-bit mode only). Upon return, SYSRET copies the value saved in RCX to the RIP.

In a return to 64-bit mode using Osize 64, SYSRET sets the CS selector value to MSR IA32_STAR[63:48] +16. The SS is set to IA32_STAR[63:48] + 8.

It is the responsibility of the OS to keep descriptors in the GDT/LDT that correspond to selectors loaded by SYSCALL/SYSRET consistent with the base, limit and attribute values forced by the these instructions.

Software should not alter the CS or SS descriptors in a manner that violates the following assumptions made by SYSCALL/SYSRET:

- CS and SS base and limit remain the same for all processes, including the operating system.
- CS of the SYSCALL target has a privilege level of 0.
- CS of the SYSRET target has a privilege level of 3.

SYSCALL/SYSRET do not check for violations of these assumptions.

### Operation

```
IF (CS.L ≠ 1 ) or (IA32_EFER.SCE ≠ 1)
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
    THEN #UD; FI;
IF (CPL ≠ 0)
    THEN #GP(0); FI;
IF (RCX ≠ CANONICAL_ADDRESS)
    THEN #GP(0); FI;
IF (OPERAND_SIZE = 64)
    THEN (* Return to 64-Bit Mode *)
        EFLAGS ← R11;
        CPL ← 0x3;
        CS(SEL) ← IA32_STAR[63:48] + 16;
        CS(PL) ← 0x3;
        SS(SEL) ← IA32_STAR[63:48] + 8;
        SS(PL) ← 0x3;
        RIP ← RCX;
    ELSE (* Return to Compatibility Mode *)
        #UD
FI;
```

### Flags Affected

VM, IF, RF.

### Protected Mode Exceptions

| | |
|---|---|
| #UD | If Mode ≠ 64-Bit. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #UD | Instruction not recognized in this mode. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #UD | Instruction not recognized in this mode. |

### Compatibility Mode Exceptions

| | |
|---|---|
| #UD | Instruction not recognized in this mode. |

### 64-Bit Mode Exceptions

| | |
|---|---|
| #UD | If IA32_EFER.SCE bit = 0. |
| #GP(0) | If CPL ≠ 0. |
| | If ECX contains a non-canonical address. |

## 3. A Note on the PAE Mechanism Has Been Updated

A note that discusses the PAE Mechanism has been updated. In the old version of the documentation, this note was in Section 3.8.4, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*. In the new version (Rev.015), the note is in Section 3.8.5.

The text of the updated section is provided below. See the change bars for the impacted area.

---

### 3.8.5 Page-Directory and Page-Table Entries With Extended Addressing Enabled

Figure 3-20 shows the format for the page-directory-pointer-table, page-directory, and page-table entries when 4-KByte pages and 36-bit extended physical addresses are being used. Figure 3-21 shows the format for the page-directory-pointer-table and page-directory entries when 2-MByte pages and 36-bit extended physical addresses are being used. The functions of the flags in these entries are the same as described in Section 3.7.6, "Page-Directory and Page-Table Entries". The major differences in these entries are as follows:

- A page-directory-pointer-table entry is added.
- The size of the entries are increased from 32 bits to 64 bits.
- The maximum number of entries in a page directory or page table is 512.
- The base physical address field in each entry is extended to 24 bits.

***Note:***

Older IA-32 processors that implement the PAE mechanism use uncached accesses when loading page-directory-pointer table entries. This behavior is model specific and not architectural. More recent IA-32 processors may cache page-directory-pointer table entries.

## 4. Exception and Interrupt Priority Table Updated

Information on the system response to simultaneous exceptions and interrupts has been updated. This update impacted Table 5-2, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*.

The updated table is reprinted below. See the change bars for the impacted area.

---

| Priority | Description |
|---|---|
| 1 (Highest) | Hardware Reset and Machine Checks<br>- RESET<br>- Machine Check |
| 2 | Trap on Task Switch<br>- T flag in TSS is set |
| 3 | External Hardware Interventions<br>- FLUSH<br>- STOPCLK<br>- SMI<br>- INIT |

| Priority | Description |
|---|---|
| 4 | Traps on the Previous Instruction<br>- Breakpoints<br>- Debug Trap Exceptions (TF flag set or data/I-O breakpoint) |
| 5 | Nonmaskable Interrupts (NMI) [1] |
| 6 | Maskable Hardware Interrupts [1] |
| 7 | Code Breakpoint Fault |
| 8 | Faults from Fetching Next Instruction<br>- Code-Segment Limit Violation<br>- Code Page Fault |
| 9 | Faults from Decoding the Next Instruction<br>- Instruction length > 15 bytes<br>- Invalid Opcode<br>- Coprocessor Not Available |
| 10 (Lowest) | Faults on Executing an Instruction<br>- Overflow<br>- Bound error<br>- Invalid TSS<br>- Segment Not Present<br>- Stack fault<br>- General Protection<br>- Data Page Fault |
| | - Alignment Check<br>- x87 FPU Floating-point exception<br>- SIMD floating-point exception |

**NOTE:**
1. The Intel486™ processor and earlier processors group nonmaskable and maskable interrupts in the same priority class.

## 5. Exceptions for MWAIT Instruction Updated

In the earlier documentation, this material was covered in the "MWAIT—Monitor Wait" section of Chapter 3 in the *Intel® Extended Memory 64 Technology Software Developer's Guide, Volume 2*. MWAIT is now covered in the updated version (Rev.015) of the *IA-32 Intel Architecture Software Developer's Manual, Volume 2A*.

The updated list of exceptions is reprinted below.

---

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If ECX ≠ 0. |
| #UD | If CPUID.01H:ECX.MONITOR[bit 3] = 0. |
| | If executed at privilege level 1 through 3 when the instruction is not available. |
| | If LOCK prefixes are used. |
| | If REPE, REPNE or operand size prefixes are used. |

### Real Address Mode Exceptions

| | |
|---|---|
| #GP(0) | For ECX has a value other than 0. |
| #UD | If CPUID.01H:ECX.MONITOR[bit 3] = 0. |
| | If LOCK prefix is used. |
| | If REPE, REPNE or operand size prefixes are used. |

### Virtual 8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | For ECX has a value other than 0. |
| #UD | If CPUID.01H:ECX.MONITOR[bit 3] = 0; or the instruction is executed at privilege level 1-2-3 when the instruction is not available. |
| | If LOCK prefix is used. |
| | If REPE, REPNE or operand size prefixes are used. |

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| | If RCX ≠ 0. |
| #UD | If CPUID.01H:ECX.MONITOR[bit 3] = 0. |
| | If the F3H, F2H, 66H or LOCK prefix is used. |

## 6. More Information on the Time-stamp Counter

The full 64-bit time-stamp counter is now writeable with WRMSR (for newer models). Table B-1 in *IA-32 Intel Architecture Software Developer's Manual, Volume 3* has been updated to reflect this. The updated table segment is reprinted below.

---

| Register Address | | Register Name Fields and Flags | Model Avail-ability | Shared/ Unique[1] | Bit Description |
|---|---|---|---|---|---|
| Hex | Dec | | | | |
| ... more table cells.... | | | | | |
| 10H | 16 | IA32_TIME_STAMP_ COUNTER | 0, 1, 2, 3, 4 | Unique | Time Stamp Counter. See Section 15.8, “Time-Stamp Counter” |
| | | 63:0 | | | **Timestamp Count Value.** A 64-bit register accessed when referenced as a qword through a RDMSR, WRMSR or RDTSC instruction. Returns the current time stamp count value. All 64 bits are readable.<br><br>On earlier processors, only the lower 32 bits are writeable. On any write to the lower 32 bits, the upper 32 bits are cleared. For processor family 0FH, models 3 and 4: all 64 bits are writeable. |
| ... more table cells.... | | | | | |

---

Sections 15.7 and 15.9.9 in the *IA-32 Intel Architecture Software Developer’s Manual, Volume 3* have been updated to improve the description of clock-tick operation. In the new version of the documentation (Rev.015), these sections are numbered 15.8 and 15.10.9.

The updated documentation is reprinted below.

---

# Time-Stamp Counter

The IA-32 architecture (beginning with the Pentium processor) defines 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 IA-32 processor implementation 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, and Intel Xeon processors) is a 64-bit counter that is set to 0 following a RESET of the processor. Following a RESET, the counter will increment 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.

Members of the 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]): 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 frequency at which the processor is booted. 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 15.10.9 and Appendix A in this manual 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]: all 64 bits are writeable.

## 15.10.9 Counting Clocks

The count of cycles, also known as clockticks, forms a the basis for measuring how long a program takes to execute. Clockticks are also used as part of efficiency ratios like cycles per instruction (CPI). Processor clocks may stop ticking under circumstances like the following:

- The processor is halted when there is nothing for the CPU to do. For example, the processor may halt to save power while the computer is servicing an I/O request. When Hyper-Threading

Technology is enabled, both logical processors must be halted for performance-monitoring counters to be powered down.

- The processor is asleep as a result of being halted or because of a power-management scheme. There are different levels of sleep. In the some deep sleep levels, the time-stamp counter stops counting.

There are three ways to count processor clock cycles to monitor performance. These are:

- **Non-halted clockticks —** Measures clock cycles in which the specified logical processor is not halted and is not in any power-saving state. When Hyper-Threading Technology is enabled, these ticks can be measured on a per-logical-processor basis.
- **Non-sleep clockticks —** Measures clock cycles in which the specified physical processor is not in a sleep mode or in a power-saving state. These ticks can**n**ot be measured on a logical-processor basis.
- **Time-stamp counter —** Measures clock cycles in which the physical processor is not in deep sleep. These ticks cannot be measured on a logical-processor basis. Some processor models permit clock cycles to be measured when the physical processor is not in deep sleep (by using the time-stamp counter and the RDTSC instruction). Note that such sticks cannot be measured on a per-logical-processor basis. See Section  for detail on processor capabilities.

The first two methods use performance counters and can be set up to cause an interrupt upon overflow (for sampling). They may also be useful where it is easier for a tool to read a performance counter than to use a time stamp counter (the timestamp counter is accessed using the RDTSC instruction).

For applications with a significant amount of I/O, there are two ratios of interest:

- **Non-halted CPI —** Non-halted clockticks/instructions retired measures the CPI for phases where the CPU was being used. This ratio can be measured on a logical-processor basis when Hyper-Threading Technology is enabled.
- **Nominal CPI —** Time-stamp counter ticks/instructions retired measures the CPI over the duration of a program, including those periods when the machine halts while waiting for I/O.

### 15.10.9.3 Incrementing the Time-Stamp Counter

The time-stamp counter increments when the clock signal on the system bus is active and when the sleep pin is not asserted. The counter value can be read with the RDTSC instruction.

The time-stamp counter and the non-sleep clockticks count may not agree in all cases and for all processors. See Section 10.8 for more information on counter operation.

## 7. Section on Serializing Instructions Has Been Updated

The information on SFENCE, LFENCE, MFENCE has been updated in Section 7.4, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*.

The updated material is reprinted below. Note the change bars.

---

# Serializing Instructions

The IA-32 architecture defines 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 insures 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 Pentium 4, Intel Xeon, and P6 family processors constrain speculative execution because the results of speculatively executed instructions are discarded. The following instructions are serializing instructions:

- **Privileged serializing instructions** — MOV (to control register), MOV (to debug register), WRMSR, INVD, INVLPG, WBINVD, LGDT, LLDT, LIDT, and LTR.
- **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.

The following instructions are memory ordering instructions, not serializing instructions. These drain the data memory subsystem. They do not effect the instruction execution stream:

- **Non-privileged memory ordering instructions** — SFENCE, LFENCE, and MFENCE.

The SFENCE, LFENCE, and MFENCE instructions provide more granularity in controlling the serialization of memory loads and stores (see Section 7.2.4, “Strengthening or Weakening the Memory Ordering Model”).

The following additional information is worth noting regarding serializing instructions:

- The processor does not writeback the contents of modified data in its data cache to external memory when it serializes instruction execution. Software can force modified data to be written back by executing the WBINVD instruction, which is a serializing instruction. It should be noted that frequent use of the WBINVD instruction will seriously reduce system performance.
- When an instruction is executed that enables or disables paging (that is, changes the PG flag in control register CR0), the instruction should be followed by a jump instruction. The target instruction of the jump instruction is fetched with the new setting of the PG flag (that is, paging is enabled or disabled), but the jump instruction itself is fetched with the previous setting. The Pentium 4, Intel Xeon, and P6 family processors do not require the jump operation following the move to register CR0 (because any use of the MOV instruction in a Pentium 4, Intel Xeon, or P6 family processor to write to CR0 is completely serializing). However, to maintain backwards and forward compatibility with code written to run on other IA-32 processors, it is recommended that the jump operation be performed.

- Whenever an instruction is executed to change the contents of CR3 while paging is enabled, the next instruction is fetched using the translation tables that correspond to the new value of CR3. Therefore the next instruction and the sequentially following instructions should have a mapping based upon the new value of CR3. (Global entries in the TLBs are not invalidated, see Section 10.9, “Invalidating the Translation Lookaside Buffers (TLBs)”.)
- The Pentium 4, Intel Xeon, P6 family, and Pentium processors use branch-prediction techniques to improve performance by prefetching the destination of a branch instruction before the branch instruction is executed. Consequently, instruction execution is not deterministically serialized when a branch instruction is executed.

## 8. LAHF/SAHF Enabled on Some 64-bit Steppings

LAHF/SAHF are now enabled on some 64-Bit steppings. The corrected information is reprinted below. For coverage of these instructions, see *IA-32 Intel Architecture Software Developer’s Manual, Volumes 2A & 2B*.

---

## LAHF—Load Status Flags into AH Register

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 9F | LAHF | Invalid* | Valid | Load: AH ← EFLAGS(SF:ZF:0:AF:0:PF:1:CF). |

**NOTE:**
* Valid in specific steppings. See Description section.

### Description

Moves the low byte of the EFLAGS register (which includes status flags SF, ZF, AF, PF, and CF) to the AH register. Reserved bits 1, 3, and 5 of the EFLAGS register are set in the AH register as shown in the “Operation” section below.

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

### Operation

```
IF 64-Bit Mode
    THEN
        IF CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1;
            THEN AH ← RFLAGS(SF:ZF:0:AF:0:PF:1:CF);
            ELSE #UD;
        FI;
    ELSE
        AH ← EFLAGS(SF:ZF:0:AF:0:PF:1:CF);
FI;
```

### Flags Affected

None. The state of the flags in the EFLAGS register is not affected.

### Protected Mode Exceptions

None.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

None.

### Compatibility Mode Exceptions

None.

### 64-Bit Mode Exceptions

#UD If CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 0.

---

## SAHF—Store AH into Flags

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 9E | SAHF | Invalid* | Valid | Loads SF, ZF, AF, PF, and CF from AH into EFLAGS register. |

* Valid in specific steppings. See Description section.

### Description

Loads the SF, ZF, AF, PF, and CF flags of the EFLAGS register with values from the corresponding bits in the AH register (bits 7, 6, 4, 2, and 0, respectively). Bits 1, 3, and 5 of register AH are ignored; the corresponding reserved bits (1, 3, and 5) in the EFLAGS register remain as shown in the “Operation” section below.

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

### Operation

```
IF IA-64 Mode
    THEN
        IF CPUID.80000001.ECX[0] = 1;
            THEN
                RFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;
            ELSE
                #UD;
        FI
    ELSE
        EFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;
FI;
```

### Flags Affected

The SF, ZF, AF, PF, and CF flags are loaded with values from the AH register. Bits 1, 3, and 5 of the EFLAGS register are unaffected, with the values remaining 1, 0, and 0, respectively.

### Protected Mode Exceptions

None.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

None.

### Compatibility Mode Exceptions

None.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #UD | If CPUID.80000001:ECX[0] = 0. |

## 9. Updated Information on Disabling the APIC

Sections 8.4.3 and 8.4.4 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 3* have been updated to more accurately describe the effect of disabling the APIC.

The updated text is reprinted below.

---

## 8.4.3 Enabling or Disabling the Local APIC

The local APIC can be enabled or disabled in either of two ways:

1. Using the APIC global enable/disable flag in the IA32_APIC_BASE MSR (MSR address 1BH). See Figure 8-5.
   - When IA32_APIC_BASE[11] is 0, the processor is functionally equivalent to an IA-32 processor without an on-chip APIC. The CPUID feature flag for the APIC (see Section 8.4.2, "Presence of the Local APIC") is also set to 0.
   - After IA32_APIC_BASE[11] is set to 0, processor APICs based on the 3-wire APIC bus cannot be generally re-enabled until a system hardware reset. The three 3-wire bus looses track of arbitration that would necessary for complete re-enabling. Certain APIC functionality can be enabled (performance and thermal monitoring interrupt generation for example). However, delivery of interrupts across the 3-wire bus cannot.
   - For processors that use Front Side Bus delivery (FSB) of interrupts, software may disable or enable the APIC by setting and resetting IA32_APIC_BASE[11]. A hardware reset is not required to re-start APIC functionality.
2. Using the APIC software enable/disable flag in the spurious-interrupt vector register. See Figure 8-23.

— If IA32_APIC_BASE[11] is 1, software can temporarily disable a local APIC at any time by clearing the APIC software enable/disable flag in the spurious-interrupt vector register (see Figure 8-23). The state of the local APIC when in this software-disabled state is described in Section 8.4.7.2, “Local APIC State After It Has Been Software Disabled”.

— When the local APIC is in the software-disabled state, it can be re-enabled at any time by setting the APIC software enable/disable flag to 1.

For the Pentium processor, the APICEN pin (which is shared with the PICD1 pin) is used during power-up or RESET to disable the local APIC.

Note that each entry in the LVT has a mask bit that can be used to inhibit interrupts from being delivered to the processor from selected local interrupt sources (the LINT0 and LINT1 pins, the APIC timer, the performance-monitoring counters, the thermal sensor, and/or the internal APIC error detector).

### 8.4.4 Local APIC Status and Location

The status and location of the local APIC are contained in the IA32_APIC_BASE MSR (see Figure 8-5). MSR bit functions are described below:

- **BSP flag, bit 8** — Indicates if the processor is the bootstrap processor (BSP). See Section 7.5, “Multiple-Processor (MP) Initialization”. Following a power-up or RESET, this flag is set to 1 for the processor selected as the BSP and set to 0 for the remaining processors (APs).
- **APIC Global Enable flag, bit 11** — Enables or disables the local APIC (see Section 8.4.3, *Enabling or Disabling the Local APIC*). This flag is available in the Pentium 4, Intel Xeon, and P6 family processors. It is not guaranteed to be available or available at the same location in future IA-32 processors.
- **APIC Base field, bits 12 through 35** — Specifies the base address of the APIC registers. This 24-bit value is extended by 12 bits at the low end to form the base address. This automatically aligns the address on a 4-KByte boundary. Following a power-up or RESET, the field is set to FEE0 0000H.
- Bits 0 through 7, bits 9 and 10, and bits 36 through 63 in the IA32_APIC_BASE MSR are reserved.

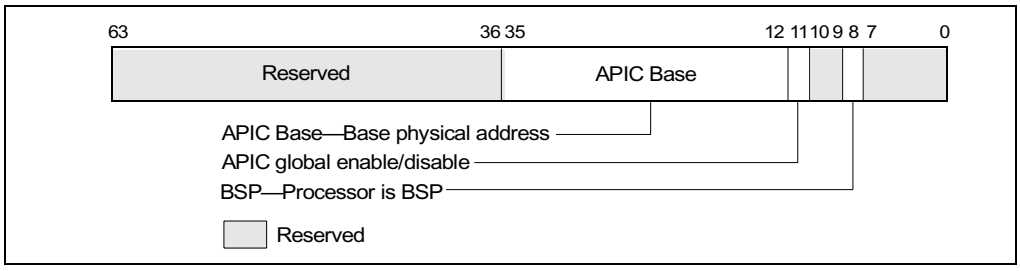


**Figure 8-5. IA32_APIC_BASE MSR (APIC_BASE_MSR in P6 Family)**

## 10. Updated ENTER Instruction Pseudocode

“ENTER—Make Stack Frame for Procedure Parameters” section in Chapter 3 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 2B* has been updated. The updated section is reprinted below.

---

## ENTER—Make Stack Frame for Procedure Parameters

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| C8 *iw* 00 | ENTER *imm16*, 0 | Valid | Valid | Create a stack frame for a procedure. |
| C8 *iw* 01 | ENTER *imm16*,1 | Valid | Valid | Create a nested stack frame for a procedure. |
| C8 *iw* ib | ENTER *imm16, imm8* | Valid | Valid | Create a nested stack frame for a procedure. |

### Description

Creates a stack frame for a procedure. The first operand (size operand) specifies the size of the stack frame (that is, the number of bytes of dynamic storage allocated on the stack for the procedure). The second operand (nesting level operand) gives the lexical nesting level (0 to 31) of the procedure. The nesting level determines the number of stack frame pointers that are copied into the "display area" of the new stack frame from the preceding frame. Both of these operands are immediate values.

The stack-size attribute determines whether the BP (16 bits), EBP (32 bits), or RBP (64 bits) register specifies the current frame pointer and whether SP (16 bits), ESP (32 bits), or RSP (64 bits) specifies the stack pointer. In 64-bit mode, stack-size attribute is always 64-bits.

The ENTER and companion LEAVE instructions are provided to support block structured languages. The ENTER instruction (when used) is typically the first instruction in a procedure and is used to set up a new stack frame for a procedure. The LEAVE instruction is then used at the end of the procedure (just before the RET instruction) to release the stack frame.

If the nesting level is 0, the processor pushes the frame pointer from the BP/EBP/RBP register onto the stack, copies the current stack pointer from the SP/ESP/RSP register into the BP/EBP/RBP register, and loads the SP/ESP/RSP register with the current stack-pointer value minus the value in the size operand. For nesting levels of 1 or greater, the processor pushes additional frame pointers on the stack before adjusting the stack pointer. These additional frame pointers provide the called procedure with access points to other nested frames on the stack. See "Procedure Calls for Block-Structured Languages" in Chapter 6 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for more information about the actions of the ENTER instruction.

In 64-bit mode, default operation size is 64 bits; 32-bit operation size cannot be encoded.

### Operation

```
NestingLevel ← NestingLevel MOD 32
IF 64-Bit Mode (StackSize = 64)
    THEN
        Push(RBP);
        FrameTemp ← RSP;
    ELSE IF StackSize = 32
        THEN
            Push(EBP);
            FrameTemp ← ESP; FI;
    ELSE (* StackSize = 16 *)
```

```
            Push(BP);
            FrameTemp ← SP;
FI;
IF NestingLevel = 0
    THEN GOTO CONTINUE; FI;
IF (NestingLevel > 0)
    FOR i ← 1 to (NestingLevel − 1)
        DO
            IF 64-Bit Mode (StackSize = 64)
                THEN
                    RBP ← RBP − 8;
                    Push([RBP]); (* Quadword push *)
                ELSE IF OperandSize = 32
                    THEN
                        IF StackSize = 32
                            EBP ← EBP − 4;
                            Push([EBP]); (* Doubleword push *)
                        ELSE (* StackSize = 16 *)
                            BP ← BP − 4;
                            Push([BP]); (* Doubleword push *)
                        FI;
                    ELSE (* OperandSize = 16 *)
                        IF StackSize = 32
                            THEN
                                EBP ← EBP − 2;
                                Push([EBP]); (* Word push *)
                            ELSE (* StackSize = 16 *)
                                BP ← BP − 2;
                                Push([BP]); (* Word push *)
                        FI;
                FI;
    OD;
    IF 64-Bit Mode (StackSize = 64)
        THEN
            Push(FrameTemp); (* Quadword push *)
        ELSE IF OperandSize = 32
            THEN
                Push(FrameTemp); FI; (* Doubleword push *)
        ELSE (* OperandSize = 16 *)
                Push(FrameTemp); (* Word push *)
    FI;
    GOTO CONTINUE;
FI;
CONTINUE:
IF 64-Bit Mode (StackSize = 64)
    THEN
            RBP ← FrameTemp;
            RSP ← RSP − Size;
    ELSE IF StackSize = 32
        THEN
            EBP ← FrameTemp;
            ESP ← ESP − Size; FI;
    ELSE (* StackSize = 16 *)
```

```
            BP ← FrameTemp;
            SP ← SP − Size;
FI;
END;
```

### Flags Affected

None.

### Protected Mode Exceptions

| | |
|---|---|
| #SS(0) | If the new value of the SP or ESP register is outside the stack segment limit. |
| #PF(fault-code) | If a page fault occurs. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #SS(0) | If the new value of the SP or ESP register is outside the stack segment limit. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #SS(0) | If the new value of the SP or ESP register is outside the stack segment limit. |
| #PF(fault-code) | If a page fault occurs. |

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |

## 11. Updated Information on the Usage of MSR_FSB_ESCR0

The description of the IOQ_allocation event has been updated to better reflect the usage of using MSR_FSB_ESCR0. See Table A-1 in *IA-32 Intel Architecture Software Developer's Manual, Volume 3*.

The change is reproduced in the table segment below (items 3 and 4a; see the change bars).

---

| Event Name | Event Parameters | Parameter Value | Description |
|---|---|---|---|
| *... additional table cells before this...* | | | |
| IOQ_allocation | .... | .... | ..... |

| | | | |
|---|---|---|---|
| | Event Specific Notes | | 1. If PREFETCH bit is cleared, sectors fetched using prefetch are excluded in the counts. If PREFETCH bit is set, all sectors or chunks read are counted.<br>2. Specify the edge trigger in CCCR to avoid double counting.<br>3. The mapping of interpreted bit field values to transaction types may differ with different processor model implementations of the Pentium 4 processor family. Applications that program performance monitoring events should use CPUID to determine processor models when using this event. The logic equations that trigger the event are model-specific (see 4a and 4b below). |
| | | | 4a. For Pentium 4 and Xeon Processors starting with CPUID Model field encoding equal to 2 or greater, this event is triggered by evaluating the logical expression ((Request type) and (Bit 5 or Bit 6) and (Memory type) and (Source agent)).<br>4b. For Pentium 4 and Xeon Processors with CPUID Model field encoding less than 2, this event is triggered by evaluating the logical expression [((Request type) or Bit 5 or Bit 6) or (Memory type)] and (Source agent). Note that event mask bits for memory type are ignored if either ALL_READ or ALL_WRITE is specified.<br>5. This event is known to ignore CPL in early implementations of Pentium 4 and Xeon Processors. Both user requests and OS requests are included in the count. This behavior is fixed starting with Pentium 4 and Xeon Processors with CPUID signature 0xF27 (Family 15, Model 2, Stepping 7). |
| | | | 6. For write-through (WT) and write-protected (WP) memory types, this event counts reads as the number of 64-byte sectors. Writes are counted by individual chunks. |
| | | | 7. For uncacheable (UC) memory types, this events counts the number of 8-byte chunks allocated. |
| | | | 8. For Pentium 4 and Xeon Processors with CPUID Signature less than 0xf27, only MSR_FSB_ESCR0 is available. |
| *... ... additional table cells follow ... ...* | | | |

## 12. Updated Footnotes in MOV, POP, STI Descriptions

---

"MOV—Move" section in Chapter 3 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 2A* is reprinted below.

1. If a code instruction breakpoint (for debug) is placed on an instruction located immediately after a MOV SS instruction, the breakpoint may not be triggered. However, in a sequence of instructions that load the SS register, only the first instruction in the sequence is guaranteed to delay an interrupt.
In the following sequence, interrupts may be recognized before MOV ESP, EBP executes:
MOV SS, EDX
MOV SS, EAX
MOV ESP, EBP

"POP—Pop a Value from the Stack" in Chapter 4 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 2B* is reprinted below.

1. If a code instruction breakpoint (for debug) is placed on an instruction located immediately after a POP SS instruction, the breakpoint may not be triggered. However, in a sequence of instructions that POP the SS register, only the first instruction in the sequence is guaranteed to delay an interrupt.
In the following sequence, interrupts may be recognized before POP ESP executes:
POP SS
POP SS
POP ESP

"STI—Set Interrupt Flag" in Chapter 4 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 2B* is reprinted below.

1. The STI instruction delays recognition of interrupts only if it is executed with EFLAGS.IF = 0. In a sequence of STI instructions, only the first instruction in the sequence is guaranteed to delay interrupts.
In the following instruction sequence, interrupts may be recognized before RET executes:
STI
STI
RET

## 13. Description Updated for CVTPI2PD Instruction

Chapter 3 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 2A* has been updated to correctly describe the operation of the instruction when the second operand is a memory operand. The corrected text is provided below.

---

## CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 2A /r | CVTPI2PD *xmm*, *mm/ m64** | Valid | Valid | Convert two packed signed doubleword integers from *mm/mem64* to two packed double-precision floating-point values in *xmm*. |

**NOTES:**
* Operation is different for different operand sets; see the Description section.

### Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed double-precision floating-point values in the destination operand (first operand).

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. In addition, depending on the operand configuration:

- **For operands *xmm*, *mm*:** the instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PD instruction is executed.
- **For operands *xmm*, *m64*:** the instruction does not cause a transition to MMX technology and does not take x87 FPU exceptions.

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

## 14. ROR/ROL Pseudocode Has Been Updated

"RCL/RCR/ROL/ROR—Rotate" section in Chapter 4 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 2B* has been updated to correct ROR/ROL's treatment of [EFLAGS.CF].

The corrected code segments are reprinted below.

---

*... ... ... more code ... ... ...*

```
(* ROL instruction operation *)
IF (tempCOUNT > 0) (* Prevents updates to CF *)
    WHILE (tempCOUNT ≠ 0)
        DO
            tempCF ← MSB(DEST);
            DEST ← (DEST ∗ 2) + tempCF;
            tempCOUNT ← tempCOUNT – 1;
        OD;
    ELIHW;
    CF ← LSB(DEST);
    IF COUNT = 1
        THEN OF ← MSB(DEST) XOR CF;
        ELSE OF is undefined;
    FI;
FI;

(* ROR instruction operation *)
IF tempCOUNT > 0) (* Prevent updates to CF *)
    WHILE (tempCOUNT ≠ 0)
        DO
            tempCF ← LSB(SRC);
            DEST ← (DEST / 2) + (tempCF ∗ 2^SIZE);
            tempCOUNT ← tempCOUNT – 1;
        OD;
    ELIHW;
```

```
    CF ← MSB(DEST);
    IF COUNT = 1
        THEN OF ← MSB(DEST) XOR MSB – 1(DEST);
        ELSE OF is undefined;
    FI;
FI;
```

## 15. Syntax for CPUID, CR and MSR Information

We are implementing a consistent syntax for expressing CPUID, CR and MSR values in *IA-32 Intel Architecture Software Developer's Manual*. The syntax is described in the first chapter of each of the volumes. The applicable section (from Volume 1, Chapter 1) is reprinted below.

---

### 1.3.7 A New Syntax for CPUID, CR, and MSR Values

Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a new syntax to represent this information. See Figure 1-2.

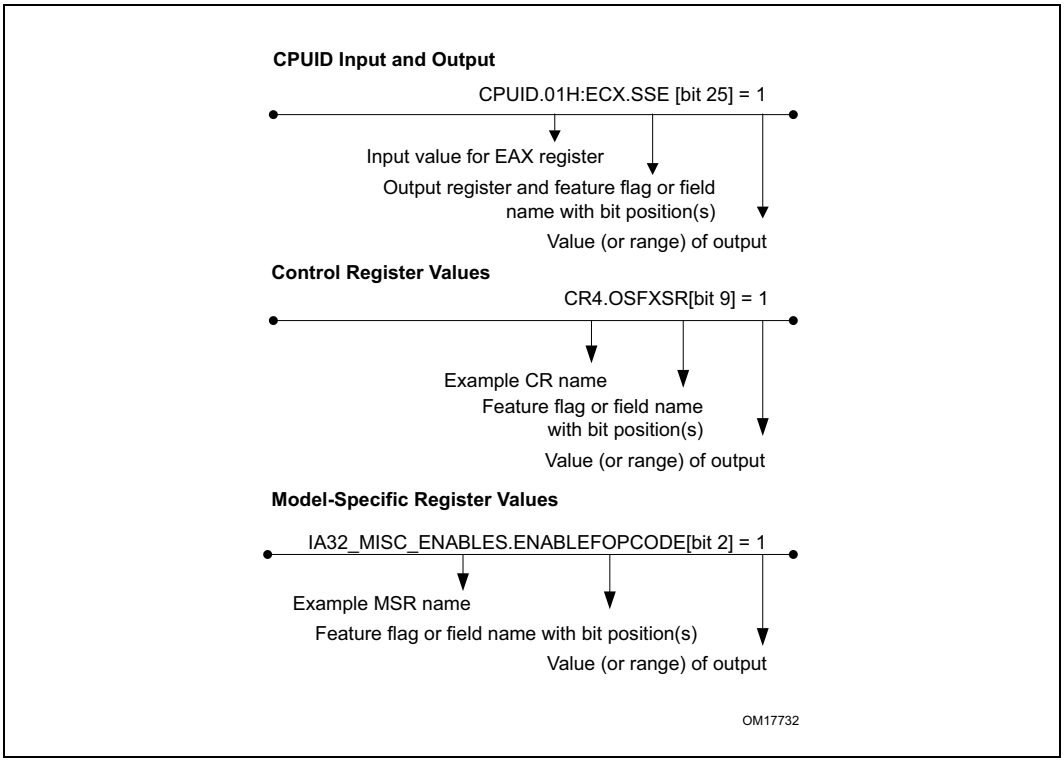


**Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation**

## 16. Inclusion of Intel® Extended Memory 64 Technology

Additional information on Intel® Extended Memory 64 Technology (Intel® EM64T) has been added to Software Developer's Manuals. For specific topics, see the table of contents and index provided in each volume.

The section shown below is reproduced from *IA-32 Intel Architecture Software Developer's Manual, Volume 1*.

---

### 2.2.5 Intel® Extended Memory 64 Technology

Intel® Extended Memory 64 Technology (Intel® EM64T) increases the linear address space for software to 64 bits and supports physical address space up to 40 bits. The technology also introduces a new operating mode referred to as IA-32e mode.

IA-32e mode operates in one of two sub-modes: (1) compatibility mode enables a 64-bit operating system to run most legacy 32-bit software unmodified, (2) 64-bit mode enables a 64-bit operating system to run applications written to access 64-bit address space.

In the 64-bit mode of Intel EM64T, applications may access:

- 64-bit flat linear addressing
- 8 additional general-purpose registers (GPRs)
- 8 additional registers for streaming SIMD extensions (SSE, SSE2 and SSE3)
- 64-bit-wide GPRs and instruction pointers
- uniform byte-register addressing
- fast interrupt-prioritization mechanism
- a new instruction-pointer relative-addressing mode.

A processor with Intel EM64T supports existing IA-32 software because it is able to run in non-64-bit legacy modes. Most existing IA-32 applications also run in compatibility mode.

## 17. Family Representation Within CPUID Context Updated

"CPUID—CPU Identification" section in Chapter 3 of the *IA-32 Intel Architecture Software Developer's Manual, Volume 2A* has been updated to correct the subsection on family and model information (the previous update contained an error).

The corrected text is reproduced below. See the change bars within the context of the section. Pay particular attention to the extended-model ID formula.

---

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

When CPUID executes with EAX set to 1, version information is returned in EAX (see Figure 1-1). For example: model, family, and processor type for the first processor in the Intel Pentium 4 family is returned as follows:

- Model — 0000B

- Family — 1111B
- Processor Type — 00B

See Table 7-6 for available processor type values. Stepping IDs are provided as needed.

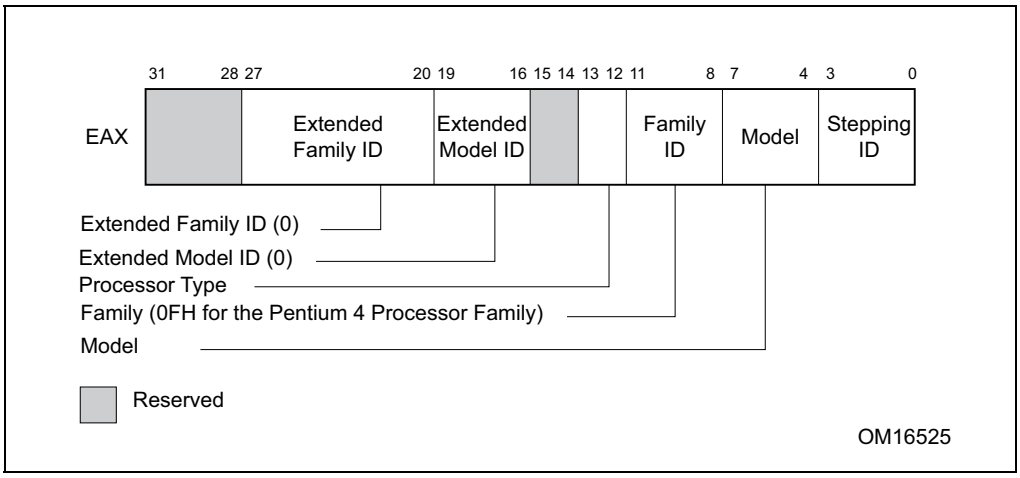


**Figure 1-1. Version Information Returned by CPUID in EAX**

**Table 7-6. Processor Type Field**

| Type | Encoding |
|---|---|
| Original OEM Processor | 00B |
| Intel OverDrive® Processor | 01B |
| Dual processor (not applicable to Intel486 processors) | 10B |
| Intel reserved | 11B |

***Note:***

See AP-485, *Intel Processor Identification and the CPUID Instruction* (Order Number 241618) and Chapter 13 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*, for information on identifying earlier IA-32 processors.

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 Displayed_Family = Family_ID;
    ELSE Displayed_Family = Extended_Family_ID + Family_ID;
    (* Right justify and zero-extend 4-bit field. *)
FI;
(* Show Display_Family 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 Displayed_Model = (Extended_Model_ID << 4) + Model_ID;
```

```
        (* Right justify and zero-extend Extended_Model_ID and Model_ID. *)
        ELSE Displayed_Model = Model_ID;
FI;
(* Show Display_Model as HEX field. *)
```