• **SYSENTER Target EIP**—Holds the offset into the CS of the called procedure.

Figure 6-2 shows the register formats and their corresponding MSR IDs.

![Figure 6-2. SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP MSRs](image)

### 6.1.3 SWAPGS Instruction

The SWAPGS instruction provides a fast method for system software to load a pointer to system data structures. SWAPGS can be used upon entering system-software routines as a result of a SYSCALL instruction or as a result of an interrupt or exception. Before returning to application software, SWAPGS can restore an application data-structure pointer that was replaced by the system data-structure pointer.

SWAPGS exchanges the base-address value located in the KernelGSbase model-specific register (MSR address C000_0102h) with the base-address value located in the hidden portion of the GS selector register (GS.base). This exchange allows the system-kernel software to quickly access kernel data structures by using the GS segment-override prefix during memory references.

The need for SwapGS arises from the requirement that, upon entry to the OS kernel, the kernel needs to obtain a 64-bit pointer to its essential data structures. When using SYSCALL to implement system calls, no kernel stack exists at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures, from which the kernel stack pointer could be read. Thus, the kernel cannot save GPRs or reference memory. SwapGS does not require any GPR or memory operands, so no registers need to be saved before using it. Similarly, when the OS kernel is entered via an interrupt or exception (where the kernel stack is already set up), SwapGS can be used to quickly get a pointer to the kernel data structures.

See “FS and GS Registers in 64-Bit Mode” on page 74 for more information on using the GS.base register in 64-bit mode.

### 6.2 System Status and Control

System-status and system-control instructions are used to determine the features supported by a processor, gather information about the current execution state, and control the processor operating modes.
6.2.1 Processor Feature Identification (CPUID)

CPUID Instruction. The CPUID instruction provides complete information about the processor implementation and its capabilities. Software operating at any privilege level can execute the CPUID instruction to collect this information. System software normally uses the CPUID instruction to determine which optional features are available so the system can be configured appropriately. See Section 3.3, “Processor Feature Identification,” on page 64.

6.2.2 Accessing Control Registers

MOV CRn Instructions. The MOV CRn instructions can be used to copy data between the control registers and the general-purpose registers. These instructions are privileged and cause a general-protection exception (#GP) if non-privileged software attempts to execute them.

LMSW and SMSW Instructions. The machine status word is located in CR0 register bits 15:0. The load machine status word (LMSW) instruction writes only the least-significant four status-word bits (CR0[3:0]). All remaining status-word bits (CR0[15:4]) are left unmodified by the instruction. The instruction is privileged and causes a #GP to occur if non-privileged software attempts to execute it.

The store machine status word (SMSW) instruction stores all 16 status-word bits (CR0[15:0]) into the target GPR or memory location. The instruction is not privileged and can be executed by all software.

CLTS Instruction. The clear task-switched bit instruction (CLTS) clears CR0.TS to 0. The CR0.TS bit is set to 1 by the processor every time a task switch takes place. The bit is useful to system software in determining when the x87 and multimedia register state should be saved or restored. See “Task Switched (TS) Bit” on page 44 for more information on using CR0.TS to manage x87-instruction state. The CLTS instruction is privileged and causes a #GP to occur if non-privileged software attempts to execute it.

6.2.3 Accessing the RFLAGS Register

The RFLAGS register contains both application and system bits. This section describes the instructions used to read and write system bits. Descriptions of instruction effects on application flags can be found in “Flags Register” in Volume 1 and “Instruction Effects on rFLAGS” in Volume 3.

POPF and PUSHF Instructions. The pop and push rFLAGS instructions are used for moving data between the rFLAGS register and the stack. They are not strictly system instructions, but their behavior is mode-dependent.

CLI and STI Instructions. The clear interrupt (CLI) and set interrupt (STI) instructions modify only the RFLAGS.IF bit or RFLAGS.VIF bit. Clearing RFLAGS.IF to 0 causes the processor to ignore maskable interrupts. Setting RFLAGS.IF to 1 causes the processor to allow maskable interrupts.

See “Virtual Interrupts” on page 262 for more information on the operation of these instructions when virtual-8086 mode extensions are enabled (CR4.VME=1).
6.2.4 Accessing Debug Registers

The MOV DRn instructions are used to copy data between the debug registers and the general-purpose registers. These instructions are privileged and cause a general-protection exception (#GP) if non-privileged software attempts to execute them. See “Debug Registers” on page 356 for a detailed description of the debug registers.

6.2.5 Accessing Model-Specific Registers

RDMSR and WRMSR Instructions. The read/write model-specific register instructions (RDMSR and WRMSR) can be used by privileged software to access the 64-bit MSRs. See “Model-Specific Registers (MSRs)” on page 58 for details about the MSRs.

RDPMC Instruction. The read performance-monitoring counter instruction, RDPMC, is used to read the model-specific performance-monitoring counter registers.

RDTSC Instruction. The read time-stamp counter instruction, RDTSC, is used to read the model-specific time-stamp counter (TSC) register.

RDTSCP Instruction. The read time-stamp counter and processor ID instruction, RDTSCP, is used to read the model-specific time-stamp counter (TSC) register, as well as the low 32 bits of the TSC_AUX register (MSR C000_0103h).

6.3 Segment Register and Descriptor Register Access

The AMD64 architecture supports the legacy instructions that load and store segment registers and descriptor registers. In some cases the instruction capabilities are expanded to support long mode.

6.3.1 Accessing Segment Registers

MOV, POP, and PUSH Instructions. The MOV and POP instructions can be used to load a selector into a segment register from a general-purpose register or memory (MOV) or from the stack (POP). Any segment register, except the CS register, can be loaded with the MOV and POP instructions. The CS register must be loaded with a far-transfer instruction.

All segment register selectors can be stored in a general-purpose register or memory using the MOV instruction or pushed onto the stack using the PUSH instruction.

When a selector is loaded into a segment register, the processor automatically loads the corresponding descriptor-table entry into the hidden portion of the selector register. The hidden portion contains the base address, limit, and segment attributes.

Segment-load and segment-store instructions work normally in 64-bit mode. The appropriate entry is read from the system descriptor table (GDT or LDT) and is loaded into the hidden portion of the segment descriptor register. However, the contents of data-segment and stack-segment descriptor registers are ignored, except in the case of the FS and GS segment-register base fields—see “FS and GS Registers in 64-Bit Mode” on page 74 for more information.
The ability to use segment-load instructions allows a 64-bit operating system to set up segment registers for a compatibility-mode application before switching to compatibility mode.

6.3.2 Accessing Segment Register Hidden State

**WRMSR and RDMSR Instructions.** The base address field of the hidden state of the FS and GS registers are mapped to MSRs and may be read and written by privileged software when running in 64-bit mode.

**RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE Instructions.** When supported and enabled, these instructions allow software running at any privilege level to read and write the base address field of the hidden state of the FS and GS segment registers. These instructions are only defined in 64-bit mode.

6.3.3 Accessing Descriptor-Table Registers

**LGDT and LIDT Instructions.** The load GDTR (LGDT) and load IDTR (LIDT) instructions load a *pseudo-descriptor* from memory into the GDTR or IDTR registers, respectively.

**LLDT and LTR Instructions.** The load LDTR (LLDT) and load TR (LTR) instructions load a system-segment descriptor from the GDT into the LDTR and TR segment-descriptor registers (hidden portion), respectively.

**SGDT and SIDT Instructions.** The store GDTR (SGDT) and store IDTR (SIDT) instructions reverse the operation of the LGDT and LIDT instructions. SGDT and SIDT store a pseudo-descriptor from the GDTR or IDTR register into memory.

**SLDT and STR Instructions.** In all modes, the store LDTR (SLDT) and store TR (STR) instructions store the LDT or task selector from the visible portion of the LDTR or TR register into a general-purpose register or memory, respectively. The hidden portion of the LDTR or TR register is not stored.

6.4 Protection Checking

Several instructions are provided to allow software to determine the outcome of a protection check before performing a memory access that could result in a protection violation. By performing the checks before a memory access, software can avoid violations that result in a general-protection exception (#GP).

6.4.1 Checking Access Rights

**LAR Instruction.** The load access-rights (LAR) instruction can be used to determine if access to a segment is allowed, based on privilege checks and type checks. The LAR instruction uses a segment-selector in the source operand to reference a descriptor in the GDT or LDT. LAR performs a set of access-rights checks and, if successful, loads the segment-descriptor access rights into the destination register. Software can further examine the access-rights bits to determine if access into the segment is allowed.
6.4.2 Checking Segment Limits

**LSL Instruction.** The *load segment-limit* (LSL) instruction uses a segment-selector in the source operand to reference a descriptor in the GDT or LDT. LSL performs a set of preliminary access-rights checks and, if successful, loads the segment-descriptor limit field into the destination register. Software can use the limit value in comparisons with pointer offsets to prevent segment limit violations.

6.4.3 Checking Read/Write Rights

**VERR and VERW Instructions.** The *verify read-rights* (VERR) and *verify write-rights* (VERW) can be used to determine if a target code or data segment (not a system segment) can be read or written from the current privilege level (CPL). The source operand for these instructions is a pointer to the segment selector to be tested. If the tested segment (code or data) is readable from the current CPL, the VERR instruction sets RFLAGS.ZF to 1; otherwise, it is cleared to zero. Likewise, if the tested data segment is writable, the VERW instruction sets the RFLAGS.ZF to 1. A code segment cannot be tested for writability.

6.4.4 Adjusting Access Rights

**ARPL Instruction.** The *adjust RPL-field* (ARPL) instruction can be used by system software to prevent access into privileged-data segments by lower-privileged software. This can happen if an application passes a selector to system software and the selector RPL is less than (has greater privilege than) the calling-application CPL. To prevent this surrogate access, system software executes ARPL with the following operands:

- The destination operand is the data-segment selector passed to system software by the application.
- The source operand is the application code-segment selector (available on the system-software stack as a result of the CALL into system software by the application).

ARPL is not supported in 64-bit mode.

6.5 Processor Halt

The *processor halt* instruction (HLT) halts instruction execution, leaving the processor in the halt state. No registers or machine state are modified as a result of executing the HLT instruction. The processor remains in the halt state until one of the following occurs:

- A non-maskable interrupt (NMI).
- An enabled, maskable interrupt (INTR).
- Processor reset (RESET).
- Processor initialization (INIT).
- System-management interrupt (SMI).
6.6 Cache and TLB Management

Cache-management instructions are used by system software to maintain coherency within the memory hierarchy. Memory coherency and caches are discussed in Chapter 7, “Memory System.” Similarly, TLB-management instructions are used to maintain coherency between page translations cached in the TLB and the translation tables maintained by system software in memory. See “Translation-Lookaside Buffer (TLB)” on page 145 for more information.

6.6.1 Cache Management

Writeback and Invalidate (WBINVD) and Writeback No Invalidate (WBNOINVD) Instructions. The writeback and invalidate (WBINVD) and writeback no invalidate (WBNOINVD) instructions are used to write all modified cache lines to memory so that memory contains the most recent copy of data. After the writes are complete, the WBINVD instruction invalidates all cache lines, whereas the WBNOINVD instruction may leave the lines in the cache hierarchy in a non-modified state. These instructions operate on all caches in the memory hierarchy, including caches that are external to the processor. See the instructions' description in Volume 3 for further operational details.

Invalidate (INVD) Instruction. The invalidate (INVD) instruction is used to invalidate all cache lines in all caches in the memory hierarchy. Unlike the WBINVD instruction, no modified cache lines are written to memory. The INVD instruction should only be used in situations where memory coherency is not required.

6.6.2 TLB Invalidation

Invalidate TLB Entry (INVLPG) Instruction. The invalidate TLB entry (INVLPG) instruction can be used to invalidate specific entries within the TLB. The source operand is a virtual-memory address that specifies the TLB entry to be invalidated. Invalidating a TLB entry does not remove the associated page-table entry from the data cache. See “Translation-Lookaside Buffer (TLB)” on page 145 for more information.

Invalidate TLB Entry in a Specified ASID (INVLPGA) Instruction. The invalidate TLB entry in a Specified ASID instruction (INVLPGA) can be used to invalidate TLB entries associated with the specified ASID. See “Invalidate Page, Alternate ASID” on page 484.

Invalidate TLB with Broadcast (INVLPGB) Instruction. The invalidate TLB with Broadcast instruction (INVLPGB) can be used to invalidate a specified range of TLB entries on the local processor and broadcast the invalidation to remote processors. See “INVLPGB” in Volume 3.

Invalidate TLB entries in Specified PCID (INVLPCCID) Instruction. The invalidate TLB entries in Specified PCID instruction (INVPCID) can be used to invalidate TLB entries of the specified Processor Context ID. See “INVPCID” in Volume 3.
This chapter describes:

- Cache coherency mechanisms
- Cache control mechanisms
- Memory typing
- Memory mapped I/O
- Memory ordering rules
- Serializing instructions

Figure 7-1 on page 168 shows a conceptual picture of a processor and memory system, and how data and instructions flow between the various components. This diagram is not intended to represent a specific microarchitectural implementation but instead is used to illustrate the major memory-system components covered by this chapter.
The memory-system components described in this chapter are shown as *unshaded* boxes in Figure 7-1. Those items are summarized in the following paragraphs.

*Main memory* is external to the processor chip and is the memory-hierarchy level farthest from the processor execution units.

*Caches* are the memory-hierarchy levels closest to the processor execution units. They are much smaller and much faster than main memory, and can be either internal or external to the processor chip. Caches contain copies of the most frequently used instructions and data. By allowing fast access to frequently used data, software can run much faster than if it had to access that data from main memory. Figure 7-1 shows three caches, all internal to the processor:
• **L1 Data Cache**—The L1 (level-1) data cache holds the data most recently read or written by the software running on the processor.

• **L1 Instruction Cache**—The L1 instruction cache is similar to the L1 data cache except that it holds only the instructions executed most frequently. In some processor implementations, the L1 instruction cache can be combined with the L1 data cache to form a unified L1 cache.

• **L2 Cache**—The L2 (level-2) cache is usually several times larger than the L1 caches, but it is also slower. It is common for L2 caches to be implemented as a unified cache containing both instructions and data. Recently used instructions and data that do not fit within the L1 caches can reside in the L2 cache. The L2 cache can be exclusive, meaning it does not cache information contained in the L1 cache. Conversely, inclusive L2 caches contain a copy of the L1-cached information.

Memory-read operations from cacheable memory first check the cache to see if the requested information is available. A **read hit** occurs if the information is available in the cache, and a **read miss** occurs if the information is not available. Likewise, a **write hit** occurs if the memory write can be stored in the cache, and a **write miss** occurs if it cannot be stored in the cache.

Caches are divided into fixed-size blocks called **cache lines**. The cache allocates lines to correspond to regions in memory of the same size as the cache line, aligned on an address boundary equal to the cache-line size. For example, in a cache with 32-byte lines, the cache lines are aligned on 32-byte boundaries and byte addresses 0007h and 001Eh are both located in the same cache line. The size of a cache line is implementation dependent. Most implementations have either 32-byte or 64-byte cache lines. The implemented cache line size is reported by CPUID Fn8000_0005 and Fn8000_0006 for the various caches, as described in Appendix E of Volume 3.

The process of loading data into a cache is a **cache-line fill**. Even if only a single byte is requested, all bytes in a cache line are loaded from memory. Typically, a cache-line fill must remove (evict) an existing cache line to make room for the new line loaded from memory. This process is called **cache-line replacement**. If the existing cache line was modified before the replacement, the processor performs a cache-line **writeback** to main memory when it performs the cache-line fill.

Cache-line writebacks help maintain **coherency** between the caches and main memory. Internally, the processor can also maintain cache coherency by **internally probing** (checking) the other caches and write buffers for a more recent version of the requested data. External devices can also check processor caches for more recent versions of data by **externally probing** the processor. Throughout this document, the term **probe** is used to refer to external probes, while internal probes are always qualified with the word **internal**.

**Write buffers** temporarily hold data writes when main memory or the caches are busy with other memory accesses. The existence of write buffers is implementation dependent.

Implementations of the architecture can use **write-combining buffers** if the order and size of non-cacheable writes to main memory is not important to the operation of software. These buffers can combine multiple, individual writes to main memory and transfer the data in fewer bus transactions.
7.1 Single-Processor Memory Access Ordering

The flexibility with which memory accesses can be ordered is closely related to the flexibility in which a processor implementation can execute and retire instructions. Instruction execution creates results and status and determines whether or not the instruction causes an exception. Instruction retirement commits the results of instruction execution, in program order, to software-visible resources such as memory, caches, write-combining buffers, and registers, or it causes an exception to occur if instruction execution created one.

Implementations of the AMD64 architecture retire instructions in program order, but implementations can execute instructions in any order, subject only to data dependencies. Implementations can also speculatively execute instructions—executing instructions before knowing they are needed. Internally, implementations manage data reads and writes so that instructions complete in order. However, because implementations can execute instructions out of order and speculatively, the sequence of memory accesses performed by the hardware can appear to be out of program order. The following sections describe the rules governing memory accesses to which processor implementations adhere. These rules may be further restricted, depending on the memory type being accessed. Further, these rules govern single processor operation; see “Multiprocessor Memory Access Ordering” on page 172 for multiprocessor ordering rules.

7.1.1 Read Ordering

Generally, reads do not affect program order because they do not affect the state of software-visible resources other than register contents. However, some system devices might be sensitive to reads. In such a situation software can map a read-sensitive device to a memory type that enforces strong read-ordering, or use read/write barrier instructions to force strong read-ordering.

For cacheable memory types, the following rules govern read ordering:

- Out-of-order reads are allowed to the extent that they can be performed transparently to software, such that the appearance of in-order execution is maintained. Out-of-order reads can occur as a result of out-of-order instruction execution or speculative execution. The processor can read memory and perform cache refills out-of-order to allow out-of-order execution to proceed.
- Speculative reads are allowed. A speculative read occurs when the processor begins executing a memory-read instruction before it knows the instruction will actually complete. For example, the processor can predict a branch will occur and begin executing instructions following the predicted branch before it knows whether the prediction is valid. When one of the speculative instructions reads data from memory, the read itself is speculative. Cache refills may also be performed speculatively.
- Reads can be reordered ahead of writes. Reads are generally given a higher priority by the processor than writes because instruction execution stalls if the read data required by an instruction is not immediately available. Allowing reads ahead of writes usually maximizes software performance.
- A read cannot be reordered ahead of a prior write if the read is from the same location as the prior write. In this case, the read instruction stalls until the write instruction completes execution. The
read instruction requires the result of the write instruction for proper software operation. For cacheable memory types, the write data can be forwarded to the read instruction before it is actually written to memory.

- Instruction fetching constitutes a parallel, asynchronous stream of reads that is independent from and unordered with respect to the read accesses performed by loads in that instruction stream.

### 7.1.2 Write Ordering

Writes affect program order because they affect the state of software-visible resources. The following rules govern write ordering:

- Generally, out-of-order writes are not allowed. Write instructions executed out of order cannot commit (write) their result to memory until all previous instructions have completed in program order. The processor can, however, hold the result of an out-of-order write instruction in a private buffer (not visible to software) until that result can be committed to memory.

- It is possible for writes to write-combining memory types to appear to complete out of order, relative to writes into other memory types. See “Memory Types” on page 178 and “Write Combining” on page 184 for additional information.

- Speculative writes are not allowed. As with out-of-order writes, speculative write instructions cannot commit their result to memory until all previous instructions have completed in program order. Processors can hold the result in a private buffer (not visible to software) until the result can be committed.

- Write buffering is allowed. When a write instruction completes and commits its result, that result can be buffered until it is actually written to system memory in program order. Although the write buffer itself is not directly accessible by software, the results in the buffer are accessible by subsequent memory accesses to the locations that are buffered, including reads for which only a subset of bytes being accessed are in the buffer. For example, a doubleword read that overlaps a single modified byte in the write buffer can return the buffered value for that byte before that write has been committed to memory.

  In general, any read from cacheable memory returns the net result of all prior globally and locally visible writes to those bytes, as performed in program order. A given implementation may provide bytes from the write buffer to satisfy this, or may stall the read until any overlapping buffered writes have been committed to memory. For cacheable memory types, the write buffer can be read out-of-order and speculatively, just like memory.

- Write combining is allowed. In some situations software can relax the write-ordering rules through the use of a Write Combining memory type or non-temporal store instructions, and allow several writes to be combined into fewer writes to memory. When write-combining is used, it is possible for writes to other memory types to proceed ahead of (out-of-order) memory-combining writes, unless the writes are to the same address. Write-combining should be used only when the order of writes does not affect program order (for example, writes to a graphics frame buffer).
7.1.3 Read/Write Barriers

When the order of memory accesses must be strictly enforced, software can use read/write barrier instructions to force reads and writes to proceed in program order. Read/write barrier instructions enforce all prior reads or writes to complete before subsequent reads or writes are executed. The LFENCE, SFENCE, and MFENCE instructions are provided as dedicated read, write, and read/write barrier instructions (respectively). Serializing instructions, I/O instructions, and locked instructions (including the implicitly locked XCHG instruction) can also be used as read/write barriers. Barrier instructions are useful for controlling ordering between differing memory types as well as within one memory type; see Section 7.3.1, “Special Coherency Considerations,” on page 177 for details.

Table 7-1 on page 180 summarizes the memory-access ordering possible for each memory type supported by the AMD64 architecture.

7.2 Multiprocessor Memory Access Ordering

The term memory ordering refers to the sequence in which memory accesses are performed by the memory system, as observed by all processors or programs.

To improve performance of applications, AMD64 processors can speculatively execute instructions out of program order and temporarily hold out-of-order results. However, certain rules are followed with regard to normal cacheable accesses on naturally aligned boundaries to WB memory.

In the examples below, all memory values are initialized to zero.

From the point of view of a program, in ascending order of priority:

- All loads, stores and I/O operations from a single processor appear to occur in program order to the code running on that processor and all instructions appear to execute in program order.
- Successive stores from a single processor are committed to system memory and visible to other processors in program order. A store by a processor cannot be committed to memory before a read appearing earlier in the program has captured its targeted data from memory. In other words, stores from a processor cannot be reordered to occur prior to a load preceding it in program order.

In this context:

- Loads do not pass previous loads (loads are not reordered). Stores do not pass previous stores (stores are not reordered)

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store A ← 1</td>
<td>Load B</td>
</tr>
<tr>
<td>Store B ← 1</td>
<td>Load A</td>
</tr>
</tbody>
</table>

Load A cannot read 0 when Load B reads 1. (This rule may be violated in the case of loads as part of a string operation, in which one iteration of the string reads 0 for Load A while another iteration reads 1 for Load B.)

- Stores do not pass loads
Stores from a processor appear to be committed to the memory system in program order; however, stores can be delayed arbitrarily by store buffering while the processor continues operation. Therefore, stores from a processor may not appear to be sequentially consistent.

- Non-overlapping Loads may pass stores.

\[
\begin{array}{ll}
\text{Processor 0} & \text{Processor 1} \\
\text{Load A} & \text{Load B} \\
\text{Store B} & \text{Store A} \\
\end{array}
\]

Load A and Load B cannot both read 1.

Both Load A and Load B may read 1. Also, due to possible write combining one or both processors may not actually store a 1 at the designated location.

- Where sequential consistency is needed (for example in Dekker’s algorithm for mutual exclusion), an MFENCE instruction should be used between the store and the subsequent load, or a locked access, such as XCHG, should be used for the store.

\[
\begin{array}{ll}
\text{Processor 0} & \text{Processor 1} \\
\text{Store A} & \text{Store B} \\
\text{...} & \text{...} \\
\text{Load B} & \text{Load A} \\
\end{array}
\]

All combinations of values (00, 01, 10, and 11) may be observed by Processors 0 and 1.

- Loads that partially overlap prior stores may return the modified part of the load operand from the store buffer, combining globally visible bytes with bytes that are only locally visible. To ensure that such loads return only a globally visible value, an MFENCE or locked access must be used between the store and the dependent load, or the store or load must be performed with a locked operation such as XCHG.

- Stores to different locations in memory observed from two (or more) other processors will appear in the same order to all observers. Behavior such as that shown in this code example,
in which processor X sees store A from processor 0 before store B from processor 1, while processor Y sees store B from processor 1 before store A from processor 0, is not allowed.

- Dependent stores between different processors appear to occur in program order, as shown in the code example below.

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
<th>Processor X</th>
<th>Processor Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store A ← 1</td>
<td>Store B ← 1</td>
<td>Load A (1)</td>
<td>Load B (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Load B (0)</td>
<td>Load A (0)</td>
</tr>
</tbody>
</table>

If processor 1 reads a value from A (written by processor 0) before carrying out a store to B, and if processor 2 reads the updated value from B, a subsequent read of A must also be the updated value.

- The local visibility (within a processor) for a memory operation may differ from the global visibility (from another processor). Using a data bypass, a local load can read the result of a local store in a store buffer, before the store becomes globally visible. Program order is still maintained when using such bypasses.

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store A ← 1</td>
<td>Load A (1)</td>
<td>Load B (1)</td>
</tr>
<tr>
<td>Load r1 A</td>
<td>Store B ← 1</td>
<td>Load A (1)</td>
</tr>
<tr>
<td>Load r2 B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Load A in processor 0 can read 1 using the data bypass, while Load A in processor 1 can read 0. Similarly, Load B in processor 1 can read 1 while Load B in processor 0 can read 0. Therefore, the result r1 = 1, r2 = 0, r3 = 1 and r4 = 0 may occur. There are no constraints on the relative order of when the Store A of processor 0 is visible to processor 1 relative to when the Store B of processor 1 is visible to processor 0.

If a very strong memory ordering model is required that does not allow local store-load bypasses, an MFENCE instruction or a synchronizing instruction such as XCHG or a locked Read-modify-write should be used between the store and the subsequent load. This enforces a memory ordering stronger than total store ordering.

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store A ← 1</td>
<td>Store B ← 1</td>
</tr>
<tr>
<td>MFENCE</td>
<td>MFENCE</td>
</tr>
</tbody>
</table>
In this example, the MFENCE instruction ensures that any buffered stores are globally visible before the loads are allowed to execute, so the result \( r_1 = 1, r_2 = 0, r_3 = 1 \) and \( r_4 = 0 \) will not occur.

### 7.3 Memory Coherency and Protocol

Implementations that support caching support a cache-coherency protocol for maintaining coherency between main memory and the caches. The cache-coherency protocol is also used to maintain coherency between all processors in a multiprocessor system. The cache-coherency protocol supported by the AMD64 architecture is the **MOESI** (modified, owned, exclusive, shared, invalid) protocol. The states of the MOESI protocol are:

- **Invalid**—A cache line in the invalid state does not hold a valid copy of the data. Valid copies of the data can be either in main memory or another processor cache.

- **Exclusive**—A cache line in the exclusive state holds the most recent, correct copy of the data. The copy in main memory is also the most recent, correct copy of the data. No other processor holds a copy of the data.

- **Shared**—A cache line in the shared state holds the most recent, correct copy of the data. Other processors in the system may hold copies of the data in the shared state, as well. If no other processor holds it in the **owned** state, then the copy in main memory is also the most recent.

- **Modified**—A cache line in the modified state holds the most recent, correct copy of the data. The copy in main memory is stale (incorrect), and no other processor holds a copy.

- **Owned**—A cache line in the owned state holds the most recent, correct copy of the data. The owned state is similar to the shared state in that other processors can hold a copy of the most recent, correct data. Unlike the shared state, however, the copy in main memory can be stale (incorrect). Only one processor can hold the data in the owned state—all other processors must hold the data in the shared state.

Figure 7-2 on page 176 shows the general MOESI state transitions possible with various types of memory accesses. This is a logical software view, not a hardware view, of how cache-line state transitions. Instruction-execution activity and external-bus transactions can both be used to modify the cache MOESI state in multiprocessing or multi-mastering systems.
To maintain memory coherency, external bus masters (typically other processors with their own internal caches) need to acquire the most recent copy of data before caching it internally. That copy can be in main memory or in the internal caches of other bus-mastering devices. When an external master has a cache read-miss or write-miss, it probes the other mastering devices to determine whether the most recent copy of data is held in any of their caches. If one of the other mastering devices holds the most recent copy, it provides it to the requesting device. Otherwise, the most recent copy is provided by main memory.

**Figure 7-2. MOESI State Transitions**

To maintain memory coherency, external bus masters (typically other processors with their own internal caches) need to acquire the most recent copy of data before caching it internally. That copy can be in main memory or in the internal caches of other bus-mastering devices. When an external master has a cache read-miss or write-miss, it probes the other mastering devices to determine whether the most recent copy of data is held in any of their caches. If one of the other mastering devices holds the most recent copy, it provides it to the requesting device. Otherwise, the most recent copy is provided by main memory.
There are two general types of bus-master probes:

- Read probes indicate the external master is requesting the data for read purposes.
- Write probes indicate the external master is requesting the data for the purpose of modifying it.

Referring back to Figure 7-2 on page 176, the state transitions involving probes are initiated by other processors and external bus masters into the processor. Some read probes are initiated by devices that intend to cache the data. Others, such as those initiated by I/O devices, do not intend to cache the data. Some processor implementations do not change the data MOESI state if the read probe is initiated by a device that does not intend to cache the data.

State transitions involving read misses and write misses can cause the processor to generate probes into external bus masters and to read main memory.

Read hits do not cause a MOESI-state change. Write hits generally cause a MOESI-state change into the modified state. If the cache line is already in the modified state, a write hit does not change its state.

The specific operation of external-bus signals and transactions and how they influence a cache MOESI state are implementation dependent. For example, an implementation could convert a write miss to a WB memory type into two separate MOESI-state changes. The first would be a read-miss placing the cache line in the exclusive state. This would be followed by a write hit into the exclusive cache line, changing the cache-line state to modified.

### 7.3.1 Special Coherency Considerations

In some cases, data can be modified in a manner that is impossible for the memory-coherency protocol to handle due to the effects of instruction prefetching. In such situations software must use serializing instructions and/or cache-invalidation instructions to ensure subsequent data accesses are coherent.

An example of this type of a situation is a page-table update followed by accesses to the physical pages referenced by the updated page tables. The following sequence of events shows what can happen when software changes the translation of virtual-page $A$ from physical-page $M$ to physical-page $N$:

1. Software invalidates the TLB entry. The tables that translate virtual-page $A$ to physical-page $M$ are now held only in main memory. They are not cached by the TLB.
2. Software changes the page-table entry for virtual-page $A$ in main memory to point to physical-page $N$ rather than physical-page $M$.

During Step 3, software expects the processor to access the data from physical-page $N$. However, it is possible for the processor to prefetch the data from physical-page $M$ before the page table for virtual-page $A$ is updated in Step 2. This is because the physical-memory references for the page tables are different than the physical-memory references for the data. Because the physical-memory references are different, the processor does not recognize them as requiring coherency checking and believes it is safe to prefetch the data from virtual-page $A$, which is translated into a read from physical page $M$. Similar behavior can occur when instructions are prefetched from beyond the page table update instruction.
To prevent this problem, software must use an INVLPG or MOV CR3 instruction immediately after the page-table update to ensure that subsequent instruction fetches and data accesses use the correct virtual-page-to-physical-page translation. It is not necessary to perform a TLB invalidation operation preceding the table update.

### 7.3.2 Access Atomicity

Cacheable, naturally-aligned single loads or stores of up to a quadword are atomic on any processor model, as are misaligned loads or stores of less than a quadword that are contained entirely within a naturally-aligned quadword. Misaligned load or store accesses typically incur a small latency penalty. Model-specific relaxations of this quadword atomicity boundary, with respect to this latency penalty, may be found in a given processor's Software Optimization Guide.

Misaligned accesses can be subject to interleaved accesses from other processors or cache-coherent devices which can result in unintended behavior. Atomicity for misaligned accesses can be achieved where necessary by using the XCHG instruction or any suitable LOCK-prefixed instruction. Note that misaligned locked accesses may incur a significant performance penalty on various processor models.

### 7.4 Memory Types

Memory type is an attribute that can be associated with a specific region of virtual or physical memory. Memory type designates certain caching and ordering behaviors for loads and stores to addresses in that region. Most memory types are explicitly assigned, although some are inferred by the hardware from current processor state and instruction context.

The AMD64 architecture defines the following memory types:

- **Uncacheable (UC)**—Reads from, and writes to, UC memory are not cacheable. Reads from UC memory cannot be speculative. Write-combining to UC memory is not allowed. Reads from or writes to UC memory cause the write buffers to be written to memory and be invalidated prior to the access to UC memory.
  
  The UC memory type is useful for memory-mapped I/O devices where strict ordering of reads and writes is important. Note that this strong ordering is with respect to UC accesses only; reads to memory types which support speculative operation may bypass non-conflicting UC accesses.

- **Cache Disable (CD)**—The CD memory type is a form of uncachable memory type that is inferred when the L1 caches are disabled but not invalidated, or for certain conflicting memory type assignments from the Page Attribute Table (PAT) and Memory Type Range Register (MTRR) mechanisms. The former case occurs when caches are disabled by setting CR0.CD to 1 without invalidating the caches with either the INVD or WBINVD instruction for any reference to a region designated as cacheable. The latter case occurs when a specific type has been assigned to a virtual page via PAT, and a conflicting type has been assigned to the mapped physical page via an MTRR (see “Combined Effect of MTRRs and PAT” on page 207 and “Combining Memory Types, MTRRs” on page 505 for details).

  For the L1 data cache and the L2 cache, reads from, and writes to, CD memory that hit the cache, or any other caches in the system, cause the cache line(s) to be invalidated before accessing main
memory. If a cache line is in the modified state, the line is written to main memory prior to being invalidated. The access is allowed to proceed after any invalidations are complete.

For the L1 instruction cache, instruction fetches from CD memory that hit the cache read the cached instructions rather than access main memory. Instruction fetches that miss the cache access main memory and do not cause cache-line replacement. Writes to CD memory that hit in the instruction cache cause the line to be invalidated.

- **Write-Combining (WC)**—Reads from, and writes to, WC memory are not cacheable. Reads from WC memory can be speculative. Writes to this memory type can be combined internally by the processor and written to memory as a single write operation to reduce memory accesses. For example, four word writes to consecutive addresses can be combined by the processor into a single quadword write, resulting in one memory access instead of four.

The WC memory type is useful for graphics-display memory buffers where the order of writes is not important.

- **Write-Combining Plus (WC+)**—WC+ is an uncacheable memory type, and combines writes in write-combining buffers like WC. Unlike WC (but like the CD memory type), accesses to WC+ memory probe the caches on all processors (including the caches of the processor issuing the request) to maintain coherency. This ensures that cacheable writes are observed by WC+ accesses.

- **Write-Protect (WP)**—Reads from WP memory are cacheable and allocate cache lines on a read miss. Reads from WP memory can be speculative. Writes to WP memory that hit in the cache do not update the cache. Instead, all writes update memory (write to memory), and writes that hit in the cache invalidate the cache line. Write buffering of WP memory is allowed.

The WP memory type is useful for shadowed-ROM memory where updates must be immediately visible to all devices that read the shadow locations.

- **Write-through (WT)**—Reads from WT memory are cacheable and allocate cache lines on a read miss. Reads from WT memory can be speculative. All writes to WT memory update main memory, and writes that hit in the cache update the cache line (cache lines remain in the same state after a write that hits a cache line). Writes that miss the cache do not allocate a cache line. Write buffering of WT memory is allowed.

- **Write-back (WB)**—Reads from WB memory are cacheable and allocate cache lines on a read miss. Cache lines can be allocated in the shared, exclusive, or modified states. Reads from WB memory can be speculative. All writes that hit in the cache update the cache line and place the cache line in the modified state. Writes that miss the cache allocate a new cache line and place the cache line in the modified state. Writes to main memory only take place during writeback operations. Write buffering of WB memory is allowed.

The WB memory type provides the highest-possible performance and is useful for most software and data stored in system memory (DRAM).
Table 7-1 shows the memory access ordering possible for each memory type supported by the AMD64 architecture. Table 7-3 on page 182 shows the ordering behavior of various operations on various memory types in greater detail. Table 7-2 on page 180 shows the caching policy for the same memory types.

### Table 7-1. Memory Access by Memory Type

<table>
<thead>
<tr>
<th>Memory Access Allowed</th>
<th>Memory Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UC/CD</td>
</tr>
<tr>
<td>Read</td>
<td></td>
</tr>
<tr>
<td>Out-of-Order</td>
<td>no</td>
</tr>
<tr>
<td>Speculative</td>
<td>no</td>
</tr>
<tr>
<td>Reorder Before Write</td>
<td>no</td>
</tr>
<tr>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>Out-of-Order</td>
<td>no</td>
</tr>
<tr>
<td>Speculative</td>
<td>no</td>
</tr>
<tr>
<td>Buffering</td>
<td>no</td>
</tr>
<tr>
<td>Combining(^1)</td>
<td>no</td>
</tr>
</tbody>
</table>

**Note:**
1. Write-combining buffers are separate from write (store) buffers.

### Table 7-2. Caching Policy by Memory Type

<table>
<thead>
<tr>
<th>Caching Policy</th>
<th>Memory Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UC</td>
</tr>
<tr>
<td>Read Cacheable</td>
<td>no</td>
</tr>
<tr>
<td>Write Cacheable</td>
<td>no</td>
</tr>
<tr>
<td>Read Allocate</td>
<td>no</td>
</tr>
<tr>
<td>Write Allocate</td>
<td>no</td>
</tr>
<tr>
<td>Write Hits Update Memory</td>
<td>yes^2</td>
</tr>
</tbody>
</table>

**Note:**
1. For the L1 data cache and the L2 cache, if an access hits the cache, the cache line is invalidated. If the cache line is in the modified state, the line is written to main memory and then invalidated. For the L1 instruction cache, read (instruction fetch) hits access the cache rather than main memory.
2. The data is not cached, so a cache write hit cannot occur. However, memory is updated.
3. Write hits update memory and invalidate the cache line.

7.4.1 Instruction Fetching from Uncacheable Memory

Instruction fetches from an uncacheable memory type (including those for the CD type which don't hit in the instruction cache) may read a naturally-aligned block of memory no larger than an instruction cache line that contains multiple instructions, and may or may not repeat reads of a given block in the course of extracting instructions from it. In general, the exact sequence of read accesses is not deterministic, regardless of instruction stream contents, aside from the following constraints:

- instruction fetching of branch targets from uncacheable memory will only be done non-speculatively
• sequential instruction fetching will not transition speculatively from a cacheable memory type to an uncacheable memory type
• sequential instruction fetching will not speculatively cross more than one 4KB page boundary

It is recommended that MMIO devices that have read side-effects be separated from memory that's subject to uncacheable instruction fetches by at least one 4KB page.

### 7.4.2 Memory Barrier Interaction with Memory Types

Memory types other than WB may allow weaker ordering in certain respects. When the ordering of memory accesses to differing memory types must be strictly enforced, software can use the LFENCE, MFENCE or SFENCE barrier instructions to force loads and stores to proceed in program order.

Table 7-3 on page 182 summarizes the cases where a memory barrier must be inserted between two memory operations.

The table is read as follows: the ROW is the first memory operation in program order, followed by the COLUMN, which is the second memory operation in program order. Each cell represents the ordered combination of the two memory operations and the letters a, b, c, d, e, f, g, h, i, j, k, and l within the cell represent the applicable memory ordering rule for that combination. These symbols are described in the footnotes below the table. In the table and footnotes, the abbreviation nt stands for non-temporal (load or store), io stands for input / output, lf for LFENCE, sf for SFENCE, and mf for MFENCE.
Table 7-3. Memory Access Ordering Rules

<table>
<thead>
<tr>
<th>First Memory Operation</th>
<th>Second Memory Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load (wp, wt, wb)</strong></td>
<td>a f b (lf) c c c d d d</td>
</tr>
<tr>
<td><strong>Load (uc)</strong></td>
<td>a f b (lf) c c c d d d</td>
</tr>
<tr>
<td><strong>Load (wc, wc+)</strong></td>
<td>a f b (lf) c c c d d d</td>
</tr>
<tr>
<td><strong>Store (wp, wt, wb)</strong></td>
<td>e (mf) f e (mf) g g h (sf) d d d</td>
</tr>
<tr>
<td><strong>Store (uc)</strong></td>
<td>i f i g g h (sf) d d d</td>
</tr>
<tr>
<td><strong>Store (wc+, non-temporal)</strong></td>
<td>e (mf) f e (mf) j (sf) g, m h (sf) d d d</td>
</tr>
<tr>
<td><strong>Load/Store (io)</strong></td>
<td>k k k k k l d, k d, k d, k</td>
</tr>
<tr>
<td><strong>Lock (atomic)</strong></td>
<td>k k k k k k d, k d, k d, k</td>
</tr>
<tr>
<td><strong>Serialize instruction/Interrupts/Exceptions</strong></td>
<td>l l l l l l d, l d, l d, l</td>
</tr>
</tbody>
</table>

a — A load (wp, wt, wb) may not pass a previous load (wp, wt, wb, wc, wc+, uc).

b — A load (wc, wc+) may pass a previous load (wp, wt, wb, wc, wc+). To ensure memory order, an LFENCE instruction must be inserted between the two loads.

c — A store (wp, wt, wb, uc, wc, wc+, nt) may not pass a previous load (wp, wt, wb, uc, wc, wc+, nt).

d — All previous loads and stores complete to memory or I/O space before a memory access for an I/O, locked or serializing instruction is issued.

e — A load (wp, wt, wb, wc, wc+) may pass a previous non-conflicting store (wp, wt, wb, wc, wc+, nt). To ensure memory order, an MFENCE instruction must be inserted between the store and the load.

f — A load or store (uc) does not pass a previous load or store (wp, wt, wb, uc, wc, wc+, nt).

g — A store (wp, wt, wb, uc) does not pass a previous store (wp, wt, wb, uc).

h — A store (wc, wc+, nt) may pass a previous store (wp, wt, wb) or non-conflicting store (wc, wc+, nt). To ensure memory order, an SFENCE instruction must be inserted between these two stores. A store (wc, wc+, nt) does not pass a previous conflicting store (wc, wc+, nt).

i — A load (wp, wt, wb, wc, wc+) may pass a previous non-conflicting store (uc). To ensure memory order, an MFENCE instruction must be inserted between the store and the load.

j — A store (wp, wt, wb) may pass a previous store (wc, wc+, nt). To ensure memory order, an SFENCE instruction must be inserted between these two stores.

k — All loads and stores associated with the I/O and locked instructions complete to memory (no buffered stores) before a load or store from a subsequent instruction is issued.
1 — All loads and stores complete to memory for the serializing instruction before the subsequent instruction fetch is issued.

m — A store (uc) does not pass a previous store (wc, wc+).

### 7.5 Buffering and Combining Memory Writes

#### 7.5.1 Write Buffering

Writes to memory (main memory and caches) can be stored internally by the processor in *write buffers* (also known as store buffers) before actually writing the data into a memory location. System performance can be improved by buffering writes, as shown in the following examples:

- When higher-priority memory transactions, such as reads, compete for memory access with writes, writes can be delayed in favor of reads, which minimizes or eliminates an instruction-execution stall due to a memory-operand read.
- When the memory is busy, buffering writes while the memory is busy removes the writes from the instruction-execution pipeline, which frees instruction-execution resources.

The processor manages the write buffer so that it is transparent to software. Memory accesses check the write buffer, and the processor completes writes into memory from the buffer in program order. Also, the processor completely empties the write buffer by writing the contents to memory as a result of performing any of the following operations:

- **SFENCE Instruction**—Executing a store-fence (SFENCE) instruction forces all memory writes before the SFENCE (in program order) to be written into memory (or, for WB type, the cache) before memory writes that follow the SFENCE instruction. The memory-fence (MFENCE) instruction has a similar effect, but it forces the ordering of loads in addition to stores.
- **Serializing Instructions**—Executing a serializing instruction forces the processor to retire the serializing instruction (complete both instruction execution and result writeback) before the next instruction is fetched from memory.
- **I/O instructions**—Before completing an I/O instruction, all previous reads and writes must be written to memory, and the I/O instruction must complete before completing subsequent reads or writes. Writes to I/O-address space (OUT instruction) are never buffered.
- **Locked Instructions**—A locked instruction (an instruction executed using the LOCK prefix) or an XCHG instruction (which is implicitly locked) must complete *after* all previous reads and writes and *before* subsequent reads and writes. Locked writes are never buffered, although locked reads and writes are cacheable.
- **Interrupts and Exceptions**—Interrupts and exceptions are serializing events that force the processor to write all results from the write buffer to memory before fetching the first instruction from the interrupt or exception service routine.
- **UC Memory Reads**—UC memory reads are not reordered ahead of writes.
Write buffers can behave similarly to write-combining buffers because multiple writes may be collected internally before transferring the data to caches or main memory. See the following section for a description of write combining.

### 7.5.2 Write Combining

Write-combining memory uses a different buffering scheme than write buffering described above. Writes to write-combining (WC) memory can be combined internally by the processor in a buffer for more efficient transfer to main memory at a later time. For example, 16 doubleword writes to consecutive memory addresses can be combined in the WC buffers and transferred to main memory as a single burst operation rather than as individual memory writes.

The following instructions perform writes to WC memory:

- (V)MASKMOVDQU
- MASKMOVQ
- (V)MOVNTDQ
- MOVNTI
- (V)MOVNTPD
- (V)MOVNTPS
- MOVNTQ
- MOVNTSD
- MOVNTSS

WC memory is not cacheable. A WC buffer writes its contents only to main memory.

The size and number of WC buffers available is implementation dependent. The processor assigns an address range to an empty WC buffer when a WC-memory write occurs. The size and alignment of this address range is equal to the buffer size. All subsequent writes to WC memory that fall within this address range can be stored by the processor in the WC-buffer entry until an event occurs that causes the processor to write the WC buffer to main memory. After the WC buffer is written to main memory, the processor can assign a new address range on a subsequent WC-memory write.

Writes to consecutive addresses in WC memory are not required for the processor to combine them. The processor combines any WC memory write that falls within the active-address range for a buffer. Multiple writes to the same address overwrite each other (in program order) until the WC buffer is written to main memory.

It is possible for writes to proceed out of program order when WC memory is used. For example, a write to cacheable memory that follows a write to WC memory can be written into the cache before the WC buffer is written to main memory. For this reason, and the reasons listed in the previous paragraph, software that is sensitive to the order of memory writes should avoid using WC memory.

WC buffers are written to main memory under the same conditions as the write buffers, namely when:

- Executing a store-fence (SFENCE) instruction.
- Executing a serializing instruction.
- Executing an I/O instruction.
  - Executing an MMIO access (load or store to UC memory type)
- Executing a locked instruction (an instruction executed using the LOCK prefix).
- Executing an XCHG instruction
- An interrupt or exception occurs.

WC buffers are also written to main memory when:
- A subsequent non-write-combining operation has a write address that matches the WC-buffer active-address range.
- A write to WC memory falls outside the WC-buffer active-address range. The existing buffer contents are written to main memory, and a new address range is established for the latest WC write.

### 7.6 Memory Caches

The AMD64 architecture supports the use of internal and external caches. The size, organization, coherency mechanism, and replacement algorithm for each cache is implementation dependent. Generally, the existence of the caches is transparent to both application and system software. In some cases, however, software can use cache-structure information to optimize memory accesses or manage memory coherency. Such software can use the extended-feature functions of the CPUID instruction to gather information on the caching subsystem supported by the processor. For more information, see Section 3.3, “Processor Feature Identification,” on page 64.

#### 7.6.1 Cache Organization and Operation

Although the detailed organization of a processor cache depends on the implementation, the general constructs are similar. L1 caches—data and instruction, or unified—and L2 caches usually are implemented as n-way set-associative caches. Figure 7-3 on page 186 shows a typical logical organization of an n-way set-associative cache. The physical implementation of the cache can be quite different.
As shown in Figure 7-3, the cache is organized as an array of cache lines. Each cache line consists of three parts: a cache-data line (a fixed-size copy of a memory block), a tag, and other information. Rows of cache lines in the cache array are *sets*, and columns of cache lines are *ways*. In an *n*-way set-associative cache, each set is a collection of *n* lines. For example, in a four-way set-associative cache, each set is a collection of four cache lines, one from each way.
The cache is accessed using the physical address of the data or instruction being referenced. To access data within a cache line, the physical address is used to select the set, way, and byte from the cache. This is accomplished by dividing the physical address into the following three fields:

- **Index**—The *index field* selects the cache set (row) to be examined for a hit. All cache lines within the set (one from each way) are selected by the index field.

- **Tag**—The *tag field* is used to select a specific cache line from the cache set. The physical-address tag field is compared with each cache-line tag in the set. If a match is found, a cache hit is signalled, and the appropriate cache line is selected from the set. If a match is not found, a cache miss is signalled.

- **Offset**—The *offset field* points to the first byte in the cache line corresponding to the memory reference. The referenced data or instruction value is read from (or written to, in the case of memory writes) the selected cache line starting at the location selected by the offset field.

In Figure 7-3 on page 186, the physical-address index field is shown selecting Set 2 from the cache. The tag entry for each cache line in the set is compared with the physical-address tag field. The tag entry for Way 1 matches the physical-address tag field, so the cache-line data for Set 2, Way 1 is selected using the n:1 multiplexor. Finally, the physical-address offset field is used to point to the first byte of the referenced data (or instruction) in the selected cache line.

Cache lines can contain other information in addition to the data and tags, as shown in Figure 7-3 on page 186. MOESI state and the state bits associated with the cache-replacement algorithm are typical pieces of information kept with the cache line. Instruction caches can also contain pre-decode or branch-prediction information. The type of information stored with the cache line is implementation dependent.

**Self-Modifying Code.** Software that stores into its own pending instruction stream with the intent of then executing the modified instructions is classified as self-modifying code. To support self-modifying code, AMD64 processors will flush any lines from the instruction cache that such stores hit, and will additionally check whether an instruction being modified is already in decode or execution behind the store instruction. If so, it will flush the pipeline and restart instruction fetch to acquire and re-decode the updated instruction bytes. No special action is needed by software for such updates to be immediately recognized. As with cache coherency, the check for instructions that are in flight is performed using physical addresses to avoid aliasing issues that could arise with virtual (linear) addresses.

When the modified bytes are in cacheable memory, the data cache may retain a copy of the modified cache line in a shared state, and the instruction cache refill may be satisfied from any suitable place in the memory hierarchy in a model-dependent manner that maintains cache coherency.

**Cross-Modifying Code.** Software that stores into the active instruction stream of another executing thread with the intent that the other thread subsequently execute the modified instruction stream is classified as cross-modifying code. There are two approaches to consider: asynchronous modification and synchronous modification.
Asynchronous modification. This is done with a write to the target instruction stream with no particular coordination being done between the writing and receiving threads. The nature of the code being executed by the target thread is such that it is insensitive to the exact timing of the update, for example executing in a known loop until an update to a branch instruction's offset takes it down a new path (or an update to an immediate operand, or opcode, or other instruction field). Such modifications must be done via a single store to the target thread's instruction stream that is contained entirely within a naturally-aligned quadword, and is subject to the constraints given here. A key aspect is that, although the store is performed atomically, the affected quadword may be read more than once in the process of extracting instruction bytes from it. This can result in the following scenarios resulting from a single store:

1. An update to two successive instructions, A and B, to A' and B' may result in execution of an A-B' sequence rather than A'-B'. However it will not result in an A'-B sequence since stores become visible to instruction fetchers in program order, and instruction fetchers read memory sequentially between taken branches.

2. A modification to one instruction A that changes it to two instructions A'-B will only result in execution of A'-B.

3. A modification to two instructions A-B that combines them into one instruction A' may result in a sequence of A-X, where X starts at the point in A' where B previously started.

Note that since stores to the instruction stream are observed by the instruction fetcher in program order, one can do multiple modifications to an area of the target thread's code that is beyond reach of the thread's current control flow, followed by a final asynchronous update that alters the control flow to expose the modified code to fetching and execution.

If the desired action cannot be achieved within these constraints, a synchronous modification approach must be used for reliable operation.

Synchronous modification. This entails a producer-consumer approach to the modification, where the target thread waits on a signal from the modifying thread, such as changing the state of a shared variable, before executing the modified code. The modifying thread writes to the target instruction bytes in any desired manner, then writes the synchronizing variable to release the target thread. Upon release, the target thread must then execute a serializing instruction such as CPUID or MFENCE (a locked operation is not sufficient) before proceeding to the modified code to avoid executing a stale view of the instructions which may have been speculatively fetched. Note that such speculative fetching is a function of branch predictor operation which is completely beyond the control of software.

See Volume 1, Chapter 3, “Semaphores,” for a discussion of instructions that are useful for interprocessor synchronization.

7.6.2 Cache Control Mechanisms

The AMD64 architecture provides a number of mechanisms for controlling the cacheability of memory. These are described in the following sections.
**Cache Disable.** Bit 30 of the CR0 register is the cache-disable bit, CR0.CD. Caching is enabled when CR0.CD is cleared to 0, and caching is disabled when CR0.CD is set to 1. When caching is disabled, reads and writes access main memory.

Software can disable the cache while the cache still holds valid data (or instructions). If a read or write hits the L1 data cache or the L2 cache when CR0.CD=1, the processor does the following:

1. Writes the cache line back if it is in the modified or owned state.
2. Invalidates the cache line.
3. Performs a non-cacheable main-memory access to read or write the data.

If an instruction fetch hits the L1 instruction cache when CR0.CD=1, some processor models may read the cached instructions rather than access main memory. When CR0.CD=1, the exact behavior of L2 and L3 caches is model-dependent, and may vary for different types of memory accesses.

The processor also responds to cache probes when CR0.CD=1. Probes that hit the cache cause the processor to perform Step 1. Step 2 (cache-line invalidation) is performed only if the probe is performed on behalf of a memory write or an exclusive read.

**Writethrough Disable.** Bit 29 of the CR0 register is the *not writethrough* disable bit, CR0.NW. In early x86 processors, CR0.NW is used to control cache writethrough behavior, and the combination of CR0.NW and CR0.CD determines the cache operating mode.

In early x86 processors, clearing CR0.NW to 0 enables writeback caching for main memory, effectively disabling writethrough caching for main memory. When CR0.NW=0, software can disable writeback caching for specific memory pages or regions by using other cache control mechanisms. When software sets CR0.NW to 1, writeback caching is disabled for main memory, while writethrough caching is enabled.

In implementations of the AMD64 architecture, CR0.NW is not used to qualify the cache operating mode established by CR0.CD. Table 7-4 shows the effects of CR0.NW and CR0.CD on the AMD64 architecture cache-operating modes.

<table>
<thead>
<tr>
<th>CR0.CD</th>
<th>CR0.NW</th>
<th>Cache Operating Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Cache enabled with a writeback-caching policy.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Invalid setting—causes a general-protection exception (#GP).</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Cache disabled. See “Cache Disable” on page 189.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Page-Level Cache Disable.** Bit 4 of all paging data-structure entries controls page-level cache disable (PCD). When a data-structure-entry PCD bit is cleared to 0, the page table or physical page pointed to by that entry is cacheable, as determined by the CR0.CD bit. When the PCD bit is set to 1, the page table or physical page is not cacheable. The PCD bit in the paging data-structure base-register...
(bit 4 in CR3) controls the cacheability of the highest-level page table in the page-translation hierarchy.

**Page-Level Writethrough Enable.** Bit 3 of all paging data-structure entries is the page-level writethrough enable control (PWT). When a data-structure-entry PWT bit is cleared to 0, the page table or physical page pointed to by that entry has a writeback caching policy. When the PWT bit is set to 1, the page table or physical page has a writethrough caching policy. The PWT bit in the paging data-structure base-register (bit 3 in CR3) controls the caching policy of the highest-level page table in the page-translation hierarchy.

The corresponding PCD bit must be cleared to 0 (page caching enabled) for the PWT bit to have an effect.

**Memory Typing.** Two mechanisms are provided for software to control access to and cacheability of specific memory regions:

- The memory-type range registers (MTRRs) control cacheability based on physical addresses. See “MTRRs” on page 195 for more information on the use of MTRRs.
- The page-attribute table (PAT) mechanism controls cacheability based on virtual addresses. PAT extends the capabilities provided by the PCD and PWT page-level cache controls. See “Page-Attribute Table Mechanism” on page 204 for more information on the use of the PAT mechanism.

System software can combine the use of both the MTRRs and PAT mechanisms to maximize control over memory cacheability.

If the MTRRs are disabled in implementations that support the MTRR mechanism, the default memory type is set to uncacheable (UC). Memory accesses are not cached even if the caches are enabled by clearing CR0.CD to 0. Cacheable memory types must be established using the MTRRs in order for memory accesses to be cached.

**Cache Control Precedence.** The cache-control mechanisms are used to define the memory type and cacheability of main memory and regions of main memory. Taken together, the most restrictive memory type takes precedence in defining the caching policy of memory. The order of precedence is:

1. Uncacheable (UC)
2. Write-combining (WC)
3. Write-protected (WP)
4. Writethrough (WT)
5. Writeback (WB)

For example, assume a large memory region is designated a writethrough type using the MTRRs. Individual pages within that region can have caching disabled by setting the appropriate page-table PCD bits. However, no pages within that region can have a writeback caching policy, regardless of the page-table PWT values.
7.6.3 Cache and Memory Management Instructions

Data Prefetch. The prefetch instructions are used by software as a hint to the processor that the referenced data is likely to be used in the near future. The processor can preload the cache line containing the data in anticipation of its use. PREFETCH provides a hint that the data is to be read. PREFETCHW provides a hint that the data is to be written. The processor can mark the line as modified if it is preloaded using PREFETCHW.

Memory Ordering. Instructions are provided for software to enforce memory ordering (serialization) in weakly-ordered memory types. These instructions are:

- **SFENCE (store fence)**—forces all memory writes (stores) preceding the SFENCE (in program order) to be written into memory before memory writes following the SFENCE.
- **LFENCE (load fence)**—forces all memory reads (loads) preceding the LFENCE (in program order) to be read from memory before memory reads following the LFENCE.
- **MFENCE (memory fence)**—forces all memory accesses (reads and writes) preceding the MFENCE (in program order) to be written into or read from memory before memory accesses following the MFENCE.

Cache Line Writeback and Flush. The CLFLUSH instruction (writeback, if modified, and invalidate) takes the byte memory-address operand (a linear address), and checks to see if the address is cached. If the address is cached, the entire cache line containing the address is invalidated. If any portion of the cache line is dirty (in the modified or owned state), the entire line is written to main memory before it is invalidated. CLFLUSH affects all caches in the memory hierarchy—internal and external to the processor, and across all cores. The CLWB instruction operates in the same manner except it does not invalidate the cache line. The checking and invalidation process continues until the address has been updated in memory and, for CLFLUSH, invalidated in all caches.

In most cases, the underlying memory type assigned to the address has no effect on the behavior of this instruction. However, when the underlying memory type for the address is UC or WC (as defined by the MTRRs), the processor does not proceed with checking all caches to see if the address is cached. In both cases, the address is uncacheable, and invalidation is unnecessary. Write-combining buffers are written back to memory if the corresponding physical address falls within the buffer active-address range.

Cache Writeback and Invalidate. Unlike the CLFLUSH and CLWB instructions, the WBINVD and WBNOINVD instructions operate on the entire cache, rather than a single cache line. The WBINVD and WBNOINVD instructions first write back all cache lines that are dirty (in the modified or owned state) to main memory. After writeback is complete, the WBINVD instruction additionally invalidates all cache lines. The checking and invalidation process continues until all internal caches in the executing core's path to system memory are invalidated. In some implementations this may include caches in other branches of the system's cache hierarchy; see the description of these instructions in volume 3 for more detail. For either instruction, a special bus cycle is transmitted to higher-level external caches directing them to perform a writeback-and-invalidate operation.
Cache Invalidate. The INVD instruction is used to invalidate all cache lines. Unlike the WBINVD instruction, dirty cache lines are not written to main memory. The process continues until all internal caches have been invalidated. A special bus cycle is transmitted to higher-level external caches directing them to perform an invalidation.

The INVD instruction should only be used in situations where memory coherency is not required.

7.6.4 Serializing Instructions

Serializing instructions force the processor to retire the serializing instruction and all previous instructions before the next instruction is fetched. A serializing instruction is retired when the following operations are complete:

- The instruction has executed.
- All registers modified by the instruction are updated.
- All memory updates performed by the instruction are complete.
- All data held in the write buffers have been written to memory.

Serializing instructions can be used as a barrier between memory accesses to force strong ordering of memory operations. Care should be exercised in using serializing instructions because they modify processor state and may affect program flow. The instructions also force execution serialization, which can significantly degrade performance. When strongly-ordered memory accesses are required, but execution serialization is not, it is recommended that software use the memory-ordering instructions described on page 191.

The following are serializing instructions:

- **Non-Privileged Instructions**
  - CPUID
  - IRET
  - RSM
  - MFENCE

- **Privileged Instructions**
  - MOV CRn
  - MOV DRn
  - LGDT, LIDT, LLDT, LTR
  - SWAPGS
  - WRMSR
  - WBINVD, WBNOINVD, INVD
  - INVLPG
7.6.5 Cache and Processor Topology

Cache and processor topology information is useful in the optimal management of system and application resources. Exposing processor and cache topology information to the programmer allows software to make more efficient use of hardware multithreading resources delivering optimal performance. Shared resources in a specific cache and processor topology may require special consideration in the optimization of multiprocessing software performance.

The processor topology allows software to determine which cores or logical processors are siblings in a compute unit, node, and processor package. For example, a scheduler can then choose to either compact or scatter threads (or processes) to cores in compute units, nodes, or across the cores in the entire physical package in order to optimize for a power and performance profile.

Topology extensions define processor topology at both the node, compute unit and cache level. Topology extensions include cache properties with sharing and the processor topology identified. The result is a simplified extension to the CPUID instruction that describes the processors cache topology and leverages existing industry cache properties folded into AMD’s topology extension description.

Topology extensions definition supports existing and future processors with varying degrees of cache level sharing. Topology extensions also support the description of a simple compute unit with one core or packages where the number of cores in a node and/or compute unit are not an even power of two.

**CPUID Function 8000_001D: Cache Topology Definition.** CPUID Function 8000_001D describes the hierarchical relationships of cache levels relative to the cores which share these resources. Function 8000_001D is defined to be called iteratively with the value 8000001Dh in EAX and an additional parameter in ECX. To gather information for all cache levels, software must call CPUID with 8000001Dh in EAX and ECX set to increasing values beginning with 0 until a value of 0 is returned from EAX[4:0], which indicates no more cache descriptions.

If software dynamically manages cache configuration, it will need to update any stored cache properties for the processor.

**CPUID Function 8000_001E: Processor Topology Definition.** CPUID Function 8000_001E describes processor topology with component identifiers. To read the processor topology, definition software calls the CPUID instruction with the value 8000001Eh in EAX. After execution the APIC ID is represented in EAX. EBX contains the compute unit description in the processor, while ECX contains system unique node identification. Software may read this information once for each core.

The following CPUID functions provide information about processor topology:

- CPUID Fn8000_0001_ECX
- CPUID Fn8000_0008_ECX
- CPUID Fn8000_001D_EAX, EBX, ECX, EDX
- CPUID Fn8000_001E_EAX, EBX, ECX

For more information using the CPUID instruction, see Section 3.3, “Processor Feature Identification,” on page 64.
7.7 Memory-Type Range Registers

The AMD64 architecture supports three mechanisms for software access-control and cacheability-control over memory regions. These mechanisms can be used in place of similar capabilities provided by external chipsets used with early x86 processors.

This section describes a control mechanism that uses a set of programmable model-specific registers (MSRs) called the memory-type-range registers (MTRRs). The MTRR mechanism provides system software with the ability to manage hardware-device memory mapping. System software can characterize physical-memory regions by type (e.g., ROM, flash, memory-mapped I/O) and assign hardware devices to the appropriate physical-memory type.

Another control mechanism is implemented as an extension to the page-translation capability and is called the page attribute table (PAT). It is described in “Page-Attribute Table Mechanism” on page 204. Like the MTRRs, PAT provides system software with the ability to manage hardware-device memory mapping. With PAT, however, system software can characterize physical pages and assign virtually-mapped devices to those physical pages using the page-translation mechanism. PAT may be used in conjunction with the MTRR mechanism to maximize flexibility in memory control.

Finally, control mechanisms are provided for managing memory-mapped I/O. These mechanisms employ extensions to the MTRRs and a separate feature called the top-of-memory registers. The MTRR extensions include additional MTRR type-field encodings for fixed-range MTRRs and variable-range I/O range registers (IORRs). These mechanisms are described in “Memory-Mapped I/O” on page 208.

7.7.1 MTRR Type Fields

The MTRR mechanism provides a means for associating a physical-address range with a memory type (see “Memory Types” on page 178). The MTRRs contain a type field used to specify the memory type in effect for a given physical-address range.

There are two variants of the memory type-field encodings: standard and extended. Both the standard and extended encodings use type-field bits 2:0 to specify the memory type. For the standard encodings, bits 7:3 are reserved and must be zero. For the extended encodings, bits 7:5 are reserved, but bits 4:3 are defined as the RdMem and WrMem bits. “Extended Fixed-Range MTRR Type-Field Encodings” on page 209 describes the function of these extended bits and how software enables them. Only the fixed-range MTRRs support the extended type-field encodings. Variable-range MTRRs use the standard encodings.

Table 7-5 on page 195 shows the memory types supported by the MTRR mechanism and their encoding in the MTRR type fields referenced throughout this section. Unless the extended type-field encodings are explicitly enabled, the processor uses the type values shown in Table 7-5.
If the MTRRs are disabled in implementations that support the MTRR mechanism, the default memory type is set to uncacheable (UC). *Memory accesses are not cached even if the caches are enabled by clearing CR0.CD to 0.* Cacheable memory types must be established using the MTRRs to enable memory accesses to be cached.

### 7.7.2 MTRRs

Both fixed-size and variable-size address ranges are supported by the MTRR mechanism. The fixed-size ranges are restricted to the lower 1 Mbyte of physical-address space, while the variable-size ranges can be located anywhere in the physical-address space.

Figure 7-4 on page 196 shows an example mapping of physical memory using the fixed-size and variable-size MTRRs. The areas shaded gray are not mapped by the MTRRs. Unmapped areas are set to the software-selected default memory type.

<table>
<thead>
<tr>
<th>Type Value</th>
<th>Type Name</th>
<th>Type Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>UC—Uncacheable</td>
<td>All accesses are uncacheable. Write combining is not allowed. Speculative accesses are not allowed</td>
</tr>
<tr>
<td>01h</td>
<td>WC—Write-Combining</td>
<td>All accesses are uncacheable. Write combining is allowed. Speculative reads are allowed</td>
</tr>
<tr>
<td>04h</td>
<td>WT—Writethrough</td>
<td>Reads allocate cache lines on a cache miss. Cache lines are not allocated on a write miss. Write hits update the cache and main memory.</td>
</tr>
<tr>
<td>05h</td>
<td>WP—Write-Protect</td>
<td>Reads allocate cache lines on a cache miss. All writes update main memory. Cache lines are not allocated on a write miss. Write hits invalidate the cache line and update main memory.</td>
</tr>
<tr>
<td>06h</td>
<td>WB—Writeback</td>
<td>Reads allocate cache lines on a cache miss, and can allocate to either the shared, exclusive, or modified state. Writes allocate to the modified state on a cache miss.</td>
</tr>
</tbody>
</table>
MTRRs are 64-bit model-specific registers (MSRs). They are read using the RDMSR instruction and written using the WRMSR instruction. See “Memory-Typing MSRs” on page 609 for a listing of the MTRR MSR numbers. The following sections describe the types of MTRRs and their function.

**Fixed-Range MTRRs.** The fixed-range MTRRs are used to characterize the first 1 Mbyte of physical memory. Each fixed-range MTRR contains eight type fields for characterizing a total of eight memory ranges. Fixed-range MTRRs support extended type-field encodings as described in “Extended Fixed-Range MTRR Type-Field Encodings” on page 209. The extended type field allows a fixed-range MTRR to be used as a fixed-range IORR. Figure 7-5 on page 197 shows the format of a fixed-range MTRR.
For the purposes of memory characterization, the first 1 Mbyte of physical memory is segmented into a total of 88 non-overlapping memory ranges, as follows:

- The 512 Kbytes of memory spanning addresses 00_0000h to 07_FFFFh are segmented into eight 64-Kbyte ranges. A single MTRR is used to characterize this address space.
- The 256 Kbytes of memory spanning addresses 08_0000h to 0B_FFFFh are segmented into 16 16-Kbyte ranges. Two MTRRs are used to characterize this address space.
- The 256 Kbytes of memory spanning addresses 0C_0000h to 0F_FFFFh are segmented into 64 4-Kbyte ranges. Eight MTRRs are used to characterize this address space.

Table 7-6 shows the address ranges corresponding to the type fields within each fixed-range MTRR. The gray-shaded heading boxes represent the bit ranges for each type field in a fixed-range MTRR. See Table 7-5 on page 195 for the type-field encodings.

Table 7-6. Fixed-Range MTRR Address Ranges

<table>
<thead>
<tr>
<th>Physical Address Range (in hexadecimal)</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>70000–7FFFF 60000–6FFFF 50000–5FFFF 40000–4FFFF 30000–3FFFF 20000–2FFFF 10000–1FFFF 00000–0FFFF</td>
<td>MTRRfix16K_80000</td>
</tr>
<tr>
<td>9C000–9FFFF 98000–9FFFF 94000–9FFFF 90000–9FFFF 8C000–8FFFF 88000–8FFFF 84000–8FFFF 80000–8FFFF</td>
<td>MTRRfix16K_A0000</td>
</tr>
<tr>
<td>BC000–BFFFF B8000–BFFFF B4000–BFFFF B0000–BFFFF A8000–AFFFF A4000–AFFFF A0000–AFFFF</td>
<td>MTRRfix4K_C0000</td>
</tr>
<tr>
<td>C7000–C7FFFF C6000–C6FFFF C5000–C5FFFF C4000–C4FFFF C3000–C3FFFF C2000–C2FFFF C1000–C1FFFF C0000–C0FFFF</td>
<td>MTRRfix4K_C8000</td>
</tr>
<tr>
<td>CF000–CFFFF CF000–CFFFF CD000–CDFFFF CC000–CCFFFF CB000–CBFFFF CA000–CAFFFF C9000–C9FFFF C8000–C8FFFF</td>
<td>MTRRfix4K_D0000</td>
</tr>
<tr>
<td>D7000–D7FFFF D6000–D6FFFF D5000–D5FFFF D4000–D4FFFF D3000–D3FFFF D2000–D2FFFF D1000–D1FFFF D0000–D0FFFF</td>
<td>MTRRfix4K_D8000</td>
</tr>
<tr>
<td>DF000–DFFFFF DF000–DFFFFF DC000–DCFFFF DB000–DBFFFF DA000–DAFFFF D9000–D9FFFF D8000–D8FFFF</td>
<td>MTRRfix4K_E0000</td>
</tr>
</tbody>
</table>
Variable-Range MTRRs. The variable-range MTRRs can be used to characterize any address range within the physical-memory space, including all of physical memory. Up to eight address ranges of varying sizes can be characterized using the MTRR. Two variable-range MTRRs are used to characterize each address range: MTRRphysBase<sub>n</sub> and MTRRphysMask<sub>n</sub> (n is the address-range number from 0 to 7). For example, address-range 3 is characterized using the MTRRphysBase3 and MTRRphysMask3 register pair.

Figure 7-6 shows the format of the MTRRphysBase<sub>n</sub> register and Figure 7-7 on page 199 shows the format of the MTRRphysMask<sub>n</sub> register. The fields within the register pair are read/write.

MTRRphysBase<sub>n</sub> Registers. The fields in these variable-range MTRRs, shown in Figure 7-6, are:

- **Type**—Bits 7:0. The memory type used to characterize the memory range. See Table 7-5 on page 195 for the type-field encodings. Variable-range MTRRs do not support the extended type-field encodings.

- **Range Physical Base-Address (PhysBase)**—Bits 51:12. The memory-range base-address in physical-address space. PhysBase is aligned on a 4-Kbyte (or greater) address in the 52-bit physical-address space supported by the AMD64 architecture. PhysBase represents the most-significant 40-address bits of the physical address. Physical-address bits 11:0 are assumed to be 0.

Note that a given processor may implement less than the architecturally-defined physical address size of 52 bits.
The fields in these variable-range MTRRs, shown in Figure 7-7, are:

- **Valid (V)**—Bit 11. Indicates that the MTRR pair is valid (enabled) when set to 1. When the valid bit is cleared to 0 the register pair is not used.

- **Range Physical Mask (PhysMask)**—Bits 51:12. The mask value used to specify the memory range. Like PhysBase, PhysMask is aligned on a 4-Kbyte physical-address boundary. Bits 11:0 of PhysMask are assumed to be 0.

![Figure 7-6. MTRRphysBasen Register](image)

![Figure 7-7. MTRRphysMaskn Register](image)

PhysMask and PhysBase are used together to determine whether a target physical-address falls within the specified address range. PhysMask is logically ANDed with PhysBase and separately ANDed with the upper 40 bits of the target physical-address. If the results of the two operations are identical, the target physical-address falls within the specified memory range. The pseudo-code for the operation is:
MaskBase = PhysMask AND PhysBase
MaskTarget = PhysMask AND Target_Address[51:12]
IF MaskBase == MaskTarget
    target address is in range
ELSE
    target address is not in range

Variable Range Size and Alignment. The size and alignment of variable memory-ranges (MTRRs) and I/O ranges (IORRs) are restricted as follows:

- The boundary on which a variable range is aligned must be equal to the range size. For example, a memory range of 16 Mbytes must be aligned on a 16-Mbyte boundary.
- The range size must be a power of 2 ($2^n$, $52 > n > 11$), with a minimum allowable size of 4 Kbytes. For example, 4 Mbytes and 8 Mbytes are allowable memory range sizes, but 6 Mbytes is not allowable.

PhysMask and PhysBase Values. Software can calculate the PhysMask value using the following procedure:

1. Subtract the memory-range physical base-address from the upper physical-address of the memory range.
2. Subtract the value calculated in Step 1 from the physical memory size.
3. Truncate the lower 12 bits of the result in Step 2 to create the PhysMask value to be loaded into the MTRRphysMask\textsubscript{n} register. Truncation is performed by right-shifting the value 12 bits.

For example, assume a 32-Mbyte memory range is specified within the 52-bit physical address space, starting at address 200_0000h. The upper address of the range is 3FF_FFFFh. Following the process outlined above yields:

1. 3FF_FFFFh–200_0000h = 1FF_FFFFh
2. F_FFFF_FFFF–1FF_FFFFh = F_FFFF_E000_0000h
3. Right shift (F_FFFF_E000_0000h) by 12 = FF_FFFF_E000h

In this example, the 40-bit value loaded into the PhysMask field is FF_FFFF_E000h.

Software must also truncate the lower 12 bits of the physical base-address before loading it into the PhysBase field. In the example above, the 40-bit PhysBase field is 00_0000_2000h.

Default-Range MTRRs. Physical addresses that are not within ranges established by fixed-range and variable-range MTRRs are set to a default memory-type using the MTRRdefType register. The format of this register is shown in Figure 7-8.

---

200 Memory System
The fields within the MTRRdefType register are read/write. These fields are:

- **Type**—Bits 7:0. The default memory-type used to characterize physical-memory space. See Table 7-5 on page 195 for the type-field encodings. The extended type-field encodings are not supported by this register.
- **Fixed-Range Enable (FE)**—Bit 10. All fixed-range MTRRs are enabled when FE is set to 1. Clearing FE to 0 disables all fixed-range MTRRs. Setting and clearing FE has no effect on the variable-range MTRRs. The FE bit has no effect unless the E bit is set to 1 (see below).
- **MTRR Enable (E)**—Bit 11. This is the MTRR memory typing enable bit. The memory typing capabilities of all fixed-range and variable-range MTRRs are enabled when E is set to 1. Clearing E to 0 disables the memory typing capabilities of all fixed-range and variable-range MTRRs and sets the default memory-type to uncacheable (UC) regardless of the value of the Type field. This bit does not affect the operation of the RdMem and WrMem fields.

### 7.7.3 Using MTRRs

**Identifying MTRR Features.** Software determines whether a processor supports the MTRR mechanism by executing the CPUID instruction with either function 0000_0001h or function 8000_0001h. If MTRRs are supported, bit 12 in the EDX register is set to 1 by CPUID. See “Processor Feature Identification” on page 64 for more information on the CPUID instruction.

The MTRR capability register (MTRRcap) is a read-only register containing information describing the level of MTRR support provided by the processor. Figure 7-9 shows the format of this register. If MTRRs are supported, software can read MTRRcap using the RDMSR instruction. Attempting to write to the MTRRcap register causes a general-protection exception (#GP).
The MTRRcap register field are:

- **Variable-Range Register Count (VCNT)**—Bits 7:0. The VCNT field contains the number of variable-range register pairs supported by the processor. For example, a processor supporting eight register pairs returns a 08h in this field.

- **Fixed-Range Registers (FIX)**—Bit 8. The FIX bit indicates whether or not the fixed-range registers are supported. If the processor returns a 1 in this bit, *all* fixed-range registers are supported. If the processor returns a 0 in this bit, *no* fixed-range registers are supported.

- **Write-Combining (WC)**—Bit 10. The WC bit indicates whether or not the write-combining memory type is supported. If the processor returns a 1 in this bit, WC memory is supported, otherwise it is not supported.

### 7.7.4 MTRRs and Page Cache Controls

When paging and the MTRRs are both enabled, the address ranges defined by the MTRR registers can span multiple pages, each of which can characterize memory with different types (using the PCD and PWT page bits). When caching is enabled (CR0.CD=0 and CR0.NW=0), the *effective memory type* is determined as follows:

1. If the page is defined as cacheable and writeback (PCD=0 and PWT=0), then the MTRR defines the effective memory-type.

2. If the page is defined as not cacheable (PCD=1), then UC is the effective memory-type.

3. If the page is defined as cacheable and writethrough (PCD=0 and PWT=1), then the MTRR defines the effective memory-type *unless* the MTRR specifies WB memory, in which case WT is the effective memory-type.
Table 7-7 lists the MTRR and page-level cache-control combinations and their combined effect on the final memory-type, if the PAT register holds the default settings.

### Table 7-7. Combined MTRR and Page-Level Memory Type with Unmodified PAT MSR

<table>
<thead>
<tr>
<th>MTRR Memory Type</th>
<th>Page PCD Bit</th>
<th>Page PWT Bit</th>
<th>Effective Memory-Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC</td>
<td>—</td>
<td>—</td>
<td>UC</td>
</tr>
<tr>
<td>WC</td>
<td>0</td>
<td>—</td>
<td>WC</td>
</tr>
<tr>
<td>WC</td>
<td>1</td>
<td>0</td>
<td>WC*</td>
</tr>
<tr>
<td>WC</td>
<td>1</td>
<td>1</td>
<td>UC</td>
</tr>
<tr>
<td>WP</td>
<td>0</td>
<td>—</td>
<td>WP</td>
</tr>
<tr>
<td>WP</td>
<td>1</td>
<td>—</td>
<td>UC</td>
</tr>
<tr>
<td>WT</td>
<td>0</td>
<td>—</td>
<td>WT</td>
</tr>
<tr>
<td>WT</td>
<td>1</td>
<td>—</td>
<td>UC</td>
</tr>
<tr>
<td>WB</td>
<td>0 0</td>
<td>0</td>
<td>WB</td>
</tr>
<tr>
<td>WB</td>
<td>0 1</td>
<td>—</td>
<td>WT</td>
</tr>
<tr>
<td>WB</td>
<td>1 —</td>
<td>—</td>
<td>UC</td>
</tr>
</tbody>
</table>

Note: 1. The effective memory-type resulting from the combination of PCD=1, PWT=0, and an MTRR WC memory type is implementation dependent.

### Large Page Sizes

When paging is enabled, software can use large page sizes (2 Mbytes and 4 Mbytes) in addition to the more typical 4-Kbyte page size. When large page sizes are used, it is possible for multiple MTRRs to span the memory range within a single large page. Each MTRR can characterize the regions within the page with different memory types. If this occurs, the effective memory-type used by the processor within the large page is undefined.

Software can avoid the undefined behavior in one of the following ways:

- Avoid using multiple MTRRs to characterize a single large page.
- Use multiple 4-Kbyte pages rather than a single large page.
- If multiple MTRRs must be used within a single large page, software can set the MTRR type fields to the same value.
- If the multiple MTRRs must have different type-field values, software can set the large page PCD and PWT bits to the most restrictive memory type defined by the multiple MTRRs.

### Overlapping MTRR Registers

If the address ranges of two or more MTRRs overlap, the following rules are applied to determine the memory type used to characterize the overlapping address range:

1. Fixed-range MTRRs, which characterize only the first 1 Mbyte of physical memory, have precedence over variable-range MTRRs.
2. If two or more variable-range MTRRs overlap, the following rules apply:
a. If the memory types are identical, then that memory type is used.
b. If at least one of the memory types is UC, the UC memory type is used.
c. If at least one of the memory types is WT, and the only other memory type is WB, then the WT memory type is used.
d. If the combination of memory types is not listed Steps A through C immediately above, then the memory type used is undefined.

7.7.5 MTRRs in Multi-Processing Environments

In multi-processing environments, the MTRRs located in all processors must characterize memory in the same way. Generally, this means that identical values are written to the MTRRs used by the processors. This also means that values CR0.CD and the PAT must be consistent across processors. Failure to do so may result in coherency violations or loss of atomicity. Processor implementations do not check the MTRR settings in other processors to ensure consistency. It is the responsibility of system software to initialize and maintain MTRR consistency across all processors.

7.8 Page-Attribute Table Mechanism

The page-attribute table (PAT) mechanism extends the page-table entry format and enhances the capabilities provided by the PCD and PWT page-level cache controls. PAT (and PCD, PWT) allow memory-type characterization based on the virtual (linear) address. The PAT mechanism provides the same memory-typing capabilities as the MTRRs but with the added flexibility of the paging mechanism. Software can use both the PAT and MTRR mechanisms to maximize flexibility in memory-type control.

7.8.1 PAT Register

Like the MTRRs, the PAT register is a 64-bit model-specific register (MSR). The format of the PAT registers is shown in Figure 7-10. See “Memory-Typing MSRs” on page 609 for more information on the PAT MSR number and reset value.

![Figure 7-10. PAT Register](image)

The PAT register contains eight page-attribute (PA) fields, numbered from PA0 to PA7. The PA fields hold the encoding of a memory type, as found in Table 7-8 on page 205. The PAT type-encodings
match the MTRR type-encodings, with the exception that PAT adds the 07h encoding. The 07h encoding corresponds to a UC− type. The UC− type (07h) is identical to the UC type (00h) except it can be overridden by an MTRR type of WC.

Software can write any supported memory-type encoding into any of the eight PA fields. An attempt to write anything but zeros into the reserved fields causes a general-protection exception (#GP). An attempt to write an unsupported type encoding into a PA field also causes a #GP exception.

The PAT register fields are initiated at processor reset to the default values shown in Table 7-9 on page 206.

Table 7-8. PAT Type Encodings

<table>
<thead>
<tr>
<th>Type Value</th>
<th>Type Name</th>
<th>Type Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>UC—Uncacheable</td>
<td>All accesses are uncacheable. Write combining is not allowed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speculative accesses are not allowed.</td>
</tr>
<tr>
<td>01h</td>
<td>WC—Write-Combining</td>
<td>All accesses are uncacheable. Write combining is allowed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speculative reads are allowed.</td>
</tr>
<tr>
<td>04h</td>
<td>WT—Writethrough</td>
<td>Reads allocate cache lines on a cache miss, but only to the shared state. Cache lines are not allocated on a write miss. Write hits update the cache and main memory.</td>
</tr>
<tr>
<td>05h</td>
<td>WP—Write-Protect</td>
<td>Reads allocate cache lines on a cache miss, but only to the shared state. All writes update main memory. Cache lines are not allocated on a write miss. Write hits invalidate the cache line and update main memory.</td>
</tr>
<tr>
<td>06h</td>
<td>WB—Writeback</td>
<td>Reads allocate cache lines on a cache miss, and can allocate to either the shared or exclusive state. Writes allocate to the modified state on a cache miss.</td>
</tr>
<tr>
<td>07h</td>
<td>UC— (UC minus)</td>
<td>All accesses are uncacheable. Write combining is not allowed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speculative accesses are not allowed. Can be overridden by an MTRR with the WC type.</td>
</tr>
</tbody>
</table>

7.8.2 PAT Indexing

PA fields in the PAT register are selected using three bits from the page-table entries. These bits are:

- **PAT (page attribute table)**—The PAT bit is bit 7 in 4-Kbyte PTEs; it is bit 12 in 2-Mbyte and 4-Mbyte PDEs. Page-table entries that don’t have a PAT bit (PML4 entries, for example) assume PAT = 0.

- **PCD (page cache disable)**—The PCD bit is bit 4 in all page-table entries. The PCD from the PTE or PDE is selected depending on the paging mode.

- **PWT (page writethrough)**—The PWT bit is bit 3 in all page-table entries. The PWT from the PTE or PDE is selected depending on the paging mode.

Table 7-9 on page 206 shows the various combinations of the PAT, PCD, and PWT bits used to select a PA field within the PAT register. Table 7-9 also shows the default memory-type values established in the PAT register by the processor after a reset. The default values correspond to the memory types.
established by the PCD and PWT bits alone in processor implementations that do not support the PAT mechanism. In such implementations, the PAT field in page-table entries is reserved and cleared to 0. See “Page-Translation-Table Entry Fields” on page 141 for more information on the page-table entries.

### Table 7-9. PAT-Register PA-Field Indexing

<table>
<thead>
<tr>
<th>Page Table Entry Bits</th>
<th>PAT Register Field</th>
<th>Default Memory Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>PA0</td>
<td>WB</td>
</tr>
<tr>
<td>0 0 1</td>
<td>PA1</td>
<td>WT</td>
</tr>
<tr>
<td>0 1 0</td>
<td>PA2</td>
<td>UC–1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>PA3</td>
<td>UC</td>
</tr>
<tr>
<td>1 0 0</td>
<td>PA4</td>
<td>WB</td>
</tr>
<tr>
<td>1 0 1</td>
<td>PA5</td>
<td>WT</td>
</tr>
<tr>
<td>1 1 0</td>
<td>PA6</td>
<td>UC–1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>PA7</td>
<td>UC</td>
</tr>
</tbody>
</table>

**Note:**
1. Can be overridden by WC memory type set by an MTRR.

### 7.8.3 Identifying PAT Support

Software determines whether a processor supports the PAT mechanism by executing the CPUID instruction with either function 0000_0001h or function 8000_0001h. If PAT is supported, bit 16 in the EDX register is set to 1 by CPUID. See Section 3.3, “Processor Feature Identification,” on page 64 for more information on the CPUID instruction.

If PAT is supported by a processor implementation, it is always enabled. The PAT mechanism cannot be disabled by software. Software can effectively avoid using PAT by:

- Not setting PAT bits in page-table entries to 1.
- Not modifying the reset values of the PA fields in the PAT register.

In this case, memory is characterized using the same types that are used by implementations that do not support PAT.

### 7.8.4 PAT Accesses

In implementations that support the PAT mechanism, all memory accesses that are translated through the paging mechanism use the PAT index bits to specify a PA field in the PAT register. The memory type stored in the specified PA field is applied to the memory access. The process is summarized as:

1. A virtual address is calculated as a result of a memory access.
2. The virtual address is translated to a physical address using the page-translation mechanism.
3. The PAT, PCD and PWT bits are read from the corresponding page-table entry during the virtual-address to physical-address translation.
4. The PAT, PCD and PWT bits are used to select a PA field from the PAT register.
5. The memory type is read from the appropriate PA field.
6. The memory type is applied to the physical-memory access using the translated physical address.

**Page-Translation Table Access.** The PAT bit exists only in the PTE (4K paging) or PDEs (2/4 Mbyte paging). In the remaining upper levels (PML4, PDP, and 4KB PDEs), only the PWT and PCD bits are used to index into the first 4 entries in the PAT register. The resulting memory type is used for the next lower paging level.

**7.8.5 Combined Effect of MTRRs and PAT**

The memory types established by the PAT mechanism can be combined with MTRR-established memory types to form an effective memory-type. The combined effect of MTRR and PAT memory types are shown in Figure 7-10. In the AMD64 architecture, reserved and undefined combinations of MTRR and PAT memory types result in undefined behavior. If the MTRRs are disabled in implementations that support the MTRR mechanism, the default memory type is set to uncacheable (UC).

**Table 7-10. Combined Effect of MTRR and PAT Memory Types**

<table>
<thead>
<tr>
<th>PAT Memory Type</th>
<th>MTRR Memory Type</th>
<th>Effective Memory Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC</td>
<td>UC</td>
<td>UC</td>
</tr>
<tr>
<td>UC</td>
<td>WC, WP, WT, WB</td>
<td>CD</td>
</tr>
<tr>
<td>UC−</td>
<td>UC</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td>WC</td>
<td>WC</td>
</tr>
<tr>
<td></td>
<td>WP, WT, WB</td>
<td>CD</td>
</tr>
<tr>
<td>WC</td>
<td>—</td>
<td>WC</td>
</tr>
<tr>
<td>WP</td>
<td>UC</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td>WC</td>
<td>CD</td>
</tr>
<tr>
<td></td>
<td>WP</td>
<td>WP</td>
</tr>
<tr>
<td></td>
<td>WT</td>
<td>CD</td>
</tr>
<tr>
<td></td>
<td>WB</td>
<td>WP</td>
</tr>
<tr>
<td>WT</td>
<td>UC</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td>WC, WP</td>
<td>CD</td>
</tr>
<tr>
<td></td>
<td>WT, WB</td>
<td>WT</td>
</tr>
<tr>
<td>WB</td>
<td>UC</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td>WC</td>
<td>WC</td>
</tr>
<tr>
<td></td>
<td>WP</td>
<td>WP</td>
</tr>
<tr>
<td></td>
<td>WT</td>
<td>WT</td>
</tr>
<tr>
<td></td>
<td>WB</td>
<td>WB</td>
</tr>
</tbody>
</table>
7.8.6 PATs in Multi-Processing Environments

In multi-processing environments, values of CR0.CD and the PAT must be consistent across all processors and the MTRRs in all processors must characterize memory in the same way. In other words, matching address ranges and cachability types are written to the MTRRs for each processor.

Failure to do so may result in coherency violations or loss of atomicity. Processor implementations do not check the MTRR, CR0.CD and PAT values in other processors to ensure consistency. It is the responsibility of system software to initialize and maintain consistency across all processors.

7.8.7 Changing Memory Type

A physical page should not have differing cacheability types assigned to it through different virtual mappings; they should be either all of a cacheable type (WB, WT, WP) or all of a non-cacheable type (UC, WC). Otherwise, this may result in a loss of cache coherency, leading to stale data and unpredictable behavior. For this reason, certain precautions must be taken when changing the memory type of a page. In particular, when changing from a cachable memory type to an uncachable type the caches must be flushed, because speculative execution by the processor may have resulted in memory being cached even though it was not programatically referenced. The following table summarizes the serialization requirements for safely changing memory types.

<table>
<thead>
<tr>
<th>Old Type</th>
<th>New Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>WB</td>
<td>WB</td>
</tr>
<tr>
<td>WB</td>
<td>–</td>
</tr>
<tr>
<td>WT</td>
<td>a</td>
</tr>
<tr>
<td>WP</td>
<td>a</td>
</tr>
<tr>
<td>UC</td>
<td>a</td>
</tr>
<tr>
<td>WC</td>
<td>a</td>
</tr>
</tbody>
</table>

**Note:**

a. Remove the previous mapping (make it not present in the page tables); Flush the TLBs including the TLBs of other processors that may have used the mapping, even speculatively; Create a new mapping in the page tables using the new type.

b. In addition to the steps described in note a, software should flush the page from the caches of any processor that may have used the previous mapping. This must be done after the TLB flushing in note a has been completed.

7.9 Memory-Mapped I/O

Processor implementations can independently direct reads and writes to either system memory or memory-mapped I/O. The method used for directing those memory accesses is implementation dependent. In some implementations, separate system-memory and memory-mapped I/O buses can be provided at the processor interface. In other implementations, system memory and memory-mapped I/O share common data and address buses, and system logic uses sideband signals from the processor to route accesses appropriately. Refer to AMD data sheets and application notes for more information about particular hardware implementations of the AMD64 architecture.
The I/O range registers (IORRs), and the top-of-memory registers allow system software to specify where memory accesses are directed for a given address range. The MTRR extensions are described in the following section. “IORRs” on page 210 describes the IORRs and “Top of Memory” on page 212 describes the top-of-memory registers. In implementations that support these features, the default action taken when the features are disabled is to direct memory accesses to memory-mapped I/O.

### 7.9.1 Extended Fixed-Range MTRR Type-Field Encodings

The fixed-range MTRRs support extensions to the type-field encodings that allow system software to direct memory accesses to system memory or memory-mapped I/O. The extended MTRR type-field encodings use previously reserved bits 4:3 to specify whether reads and writes to a physical-address range are to system memory or to memory-mapped I/O. The format for this encoding is shown in Figure 7-11 on page 209. The new bits are:

- **WrMem**—Bit 3. When set to 1, the processor directs write requests for this physical address range to system memory. When cleared to 0, writes are directed to memory-mapped I/O.
- **RdMem**—Bit 4. When set to 1, the processor directs read requests for this physical address range to system memory. When cleared to 0, reads are directed to memory-mapped I/O.

The type subfield (bits 2:0) allows the encodings specified in Table 7-5 on page 195 to be used for memory characterization.

![Figure 7-11. Extended MTRR Type-Field Format (Fixed-Range MTRRs)](image)

These extensions are enabled using the following bits in the SYSCFG MSR:

- **MtrrFixDramEn**—Bit 18. When set to 1, RdMem and WrMem attributes are enabled. When cleared to 0, these attributes are disabled. When disabled, accesses are directed to memory-mapped I/O space.
- **MtrrFixDramModEn**—Bit 19. When set to 1, software can read and write the RdMem and WrMem bits. When cleared to 0, writes do not modify the RdMem and WrMem bits, and reads return 0.

To use the MTRR extensions, system software must first set MtrrFixDramModEn=1 to allow modification to the RdMem and WrMem bits. After the attribute bits are properly initialized in the fixed-range registers, the extensions can be enabled by setting MtrrFixDramEn=1.

RdMem and WrMem allow the processor to independently direct reads and writes to either system memory or memory-mapped I/O. The RdMem and WrMem controls are particularly useful when shadowing ROM devices located in memory-mapped I/O space. It is often useful to shadow such devices in RAM system memory to improve access performance, but writes into the RAM location can...
corrupt the shadowed ROM information. The MTRR extensions solve this problem. System software can create the shadow location by setting WrMem = 1 and RdMem = 0 for the specified memory range and then copy the ROM location into itself. Reads are directed to the memory-mapped ROM, but writes go to the same physical addresses in system memory. After the copy is complete, system software can change the bit values to WrMem = 0 and RdMem = 1. Now reads are directed to the faster copy located in system memory, and writes are directed to memory-mapped ROM. The ROM responds as it would normally to a write, which is to ignore it.

Not all combinations of RdMem and WrMem are supported for each memory type encoded by bits 2:0. Table 7-12 on page 210 shows the allowable combinations. The behavior of reserved encoding combinations (shown as gray-shaded cells) is undefined and results in unpredictable behavior.

<table>
<thead>
<tr>
<th>RdMem</th>
<th>WrMem</th>
<th>Type</th>
<th>Implication or Potential Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 (UC)</td>
<td>UC I/O</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1 (WC)</td>
<td>WC I/O</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4 (WT)</td>
<td>WT I/O</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5 (WP)</td>
<td>WP I/O</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>6 (WB)</td>
<td>Reserved</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0 (UC)</td>
<td>Used while creating a shadowed ROM</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 (WC)</td>
<td>Used to access a shadowed ROM</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4 (WT)</td>
<td>Reserved</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>5 (WP)</td>
<td>WP Memory (Can be used to access shadowed ROM)</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>6 (WB)</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

### 7.9.2 IORRs

The IORRs operate similarly to the variable-range MTRRs. The IORRs specify whether reads and writes in any physical-address range map to system memory or memory-mapped I/O. Up to two
address ranges of varying sizes can be controlled using the IORRs. A pair of IORRs are used to control each address range: IORRBase \( n \) and IORRMask \( n \) (\( n \) is the address-range number from 0 to 1).

Figure 7-12 on page 211 shows the format of the IORRBase \( n \) registers and Figure 7-13 on page 212 shows the format of the IORRMask \( n \) registers. The fields within the register pair are read/write.

The intersection of the IORR range with the equivalent effective MTRR range follows the same type encoding table (Table 7-12) as the fixed-range MTRR, where the RdMem/WrMem and memory type are directly tied together.

**IORRBase \( n \) Registers.** The fields in these IORRs are:

- \( WrMem \)—Bit 3. When set to 1, the processor directs write requests for this physical address range to system memory. When cleared to 0, writes are directed to memory-mapped I/O.
- \( RdMem \)—Bit 4. When set to 1, the processor directs read requests for this physical address range to system memory. When cleared to 0, reads are directed to memory-mapped I/O.
- \( Range \ Physical-Base-Address \ (PhysBase) \)—Bits 51:12. The memory-range base-address in physical-address space. PhysBase is aligned on a 4-Kbyte (or greater) address in the 52-bit physical-address space supported by the AMD64 architecture. PhysBase represents the most-significant 40-address bits of the physical address. Physical-address bits 11:0 are assumed to be 0.

Note that a given processor may implement less than the architecturally-defined physical address size of 52 bits.

The format of these registers is shown in Figure 7-12.

![Figure 7-12. IORRBase \( n \) Register](image)

**IORRMask \( n \) Registers.** The fields in these IORRs are:
• **Valid (V)**—Bit 11. Indicates that the IORR pair is valid (enabled) when set to 1. When the valid bit is cleared to 0 the register pair is not used for memory-mapped I/O control (disabled).

• **Range Physical-Mask (PhysMask)**—Bits 51:12. The mask value used to specify the memory range. Like PhysBase, PhysMask is aligned on a 4-Kbyte physical-address boundary. Bits 11:0 of PhysMask are assumed to be 0.

The format of these registers is shown in Figure 7-13 on page 212.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:52</td>
<td>Reserved</td>
<td>Reserved, Must be Zero</td>
<td></td>
</tr>
<tr>
<td>51:12</td>
<td>PhysMask</td>
<td>Range Physical Mask</td>
<td>R/W</td>
</tr>
<tr>
<td>11</td>
<td>V</td>
<td>I/O Register Pair Enable (Valid)</td>
<td>R/W</td>
</tr>
<tr>
<td>10:0</td>
<td>Reserved</td>
<td>Reserved, Must be Zero</td>
<td></td>
</tr>
</tbody>
</table>

![Figure 7-13. IORRMaskn Register](image)

The operation of the PhysMask and PhysBase fields is identical to that of the variable-range MTRRs. See page 199 for a description of this operation.

### 7.9.3 IORR Overlapping

The use of overlapping IORRs is not recommended. If overlapping IORRs are specified, the resulting behavior is implementation-dependent.

### 7.9.4 Top of Memory

The *top-of-memory* registers, TOP_MEM and TOP_MEM2, allow system software to specify physical addresses ranges as memory-mapped I/O locations. Processor implementations can direct accesses to memory-mapped I/O differently than system I/O, and the precise method depends on the implementation. System software specifies memory-mapped I/O regions by writing an address into each of the top-of-memory registers. The memory regions specified by the TOP_MEM registers are aligned on 8-Mbyte boundaries as follows:

- Memory accesses from physical address 0 to one less than the value in TOP_MEM are directed to system memory.
- Memory accesses from the physical address specified in TOP_MEM to FFFF_FFFFh are directed to memory-mapped I/O.
• Memory accesses from physical address 1_0000_0000h to one less than the value in TOP_MEM2 are directed to system memory.
• Memory accesses from the physical address specified in TOP_MEM2 to the maximum physical address supported by the system are directed to memory-mapped I/O.

Figure 7-14 on page 213 shows how the top-of-memory registers organize memory into separate system-memory and memory-mapped I/O regions.

The intersection of the top-of-memory range with the equivalent effective MTRR range follows the same type encoding table (Table 7-12 on page 210) as the fixed-range MTRR, where the RdMem/WrMem and memory type are directly tied together.

Figure 7-14. Memory Organization Using Top-of-Memory Registers

Figure 7-15 shows the format of the TOP_MEM and TOP_MEM2 registers. Bits 51:23 specify an 8-Mbyte aligned physical address. All remaining bits are reserved and ignored by the processor. System software should clear those bits to zero to maintain compatibility with possible future extensions to the registers. The TOP_MEM registers are model-specific registers. See “Memory-Typing MSRs” on page 609 for information on the MSR address and reset values for these registers.
The TOP_MEM register is enabled by setting the MtrrVarDramEn bit in the SYSCFG MSR (bit 20) to 1 (one). The TOP_MEM2 register is enabled by setting the MtrrTom2En bit in the SYSCFG MSR (bit 21) to 1 (one). The registers are disabled when their respective enable bits are cleared to 0. When the top-of-memory registers are disabled, memory accesses default to memory-mapped I/O space.

Note that a given processor may implement fewer than the architecturally-defined number of physical address bits.

7.10 **Secure Memory Encryption**

Software running in non-virtualized (native) mode can utilize the Secure Memory Encryption (SME) feature to mark individual pages of memory as encrypted through the page tables. A page of memory marked encrypted will be automatically decrypted when read by software and automatically encrypted when written to DRAM. SME may therefore be used to protect the contents of DRAM from physical attacks on the system.

All memory encrypted using SME is encrypted with the same AES key which is created randomly each time a system is booted. The memory encryption key cannot be read or modified by software.

For details on using memory encryption in virtualized environments, please see Section 15.34, “Secure Encrypted Virtualization,” on page 539.

7.10.1 **Determining Support for Secure Memory Encryption**

Support for memory encryption features is reported in CPUID Fn8000_001F[EAX]. Bit 0 indicates support for Secure Memory Encryption. When this feature is present, CPUID Fn8000_001F[EBX] supplies additional information regarding the use of memory encryption such as which page table bit is used to mark pages as encrypted.

Additionally, in some implementations, the physical address size of the processor may be reduced when memory encryption features are enabled, for example from 48 to 43 bits. In this case the upper physical address bits are treated as reserved when the feature is enabled except where otherwise indicated. When memory encryption is supported in an implementation, CPUID Fn8000_001F[EBX] reports any physical address size reduction present. Bits reserved in this mode are treated the same as
other page table reserved bits, and will generate a page fault if found to be non-zero when used for address translation.

Complete CPUID details for encrypted memory features can be found in Volume 3, section E.4.17.

7.10.2 Enabling Memory Encryption Extensions

Prior to using SME, memory encryption features must be enabled by setting SYSCFG MSR bit 23 (MemEncryptionModEn) to 1. In implementations where the physical address size of the processor is reduced when memory encryption features are enabled, software must ensure it is executing from addresses where these upper physical address bits are 0 prior to setting SYSCFG[MemEncryptionModEn]. Memory encryption is then further controlled via the page tables.

Note that software should keep the value of SYSCFG[MemEncryptionModEn] consistent across all CPU cores in the system. Failure to do so may lead to unexpected results.

7.10.3 Supported Operating Modes

SME is supported in all CPU modes when CR4.PAE=1 and paging is enabled. This includes long mode as well as legacy PAE-enabled protected mode.

7.10.4 Page Table Support

Software utilizes the page tables to indicate if a memory page is encrypted or unencrypted. The location of the specific attribute bit (C-bit, or enCrypted bit) used is implementation-specific but may be determined by referencing CPUID Fn8000_001F[EBX] (see Volume 3, section E.4.17 for details). In some implementations, the bit used may be a physical address bit (e.g., address bit 47), especially in cases where the physical address size is reduced by hardware when memory encryption features are enabled.

To mark a memory page for encryption when stored in DRAM, software sets the C-bit to 1 for the page. If the C-bit is 0, the page is not encrypted when stored in DRAM. The C bit can be applied to translation table entries for any size of page - 4KB, 2MB, or 1GB.

Note that it is possible for the page tables themselves to be located in encrypted memory. For instance, if the C-bit is set in a PML4 entry, the PDP table it points to (and thus all PDPEs in that table) will be loaded from encrypted memory.
7.10.5 I/O Accesses

In implementations where the physical address size is reduced when memory encryption features are enabled, memory range checks (e.g. MTRR/TOM/IORR/etc.) to determine memory types or DRAM/MMIO are performed using the reduced physical address size. In particular, the C-bit is not considered a physical address bit and is masked by hardware for purposes of these checks.

Additionally, any pages corresponding to MMIO addresses must be configured with the C-bit clear. Encrypted I/O pages are not allowed and accesses with the C-bit set will result in a machine check error.

7.10.6 Restrictions

In some hardware implementations, coherency between the encrypted and unencrypted mappings of the same physical page are not enforced. In such a system, prior to changing the value of the C-bit for a page, software should flush the page from all CPU caches in the system. If a hardware implementation supports coherency across encryption domains as indicated by CPUID Fn8000_001F_EAX[10] then this flush is not required.

Simply changing the value of a C-bit on a page will not automatically encrypt the existing contents of a page, and any data in the page prior to the C-bit modification will become unintelligible. To set the C-bit on a page and cause its contents to become encrypted so the data remains accessible, see Section 7.10.8, “Encrypt-in-Place,” on page 217.
In legacy PAE mode, if the C-bit location is in the upper 32 bits of the page table entry, the first level page table (the PDP table) cannot be located in encrypted memory. This is because when the CPU is in 32-bit PAE mode, the CR3 value is only 32-bits in length.

### 7.10.7 SMM Interaction

SME is available when the processor is executing in SMM, once it has enabled paging. Any physical address bit restrictions that exist due to memory encryption features being enabled remain in place while in SMM.

### 7.10.8 Encrypt-in-Place

It is possible to perform an in-place encryption of data in physical memory. This technique is useful for setting the C-bit on a page while maintaining visibility to the page's contents such as during SME initialization. This is accomplished by creating two linear mappings of the same page where one mapping has the C-bit set to 0 and the other has the C-bit set to 1. To avoid possible data corruption, software should use the following algorithm for performing in-place encryption of memory:

1. Create two linear mappings X and Y that map to the same physical page. Mapping X has C-bit=0 and uses the WP (Write Protect) memory type. Mapping Y has C-bit=1 and uses the WB (Write-Back) memory type.
2. Perform a WBINVD on all cores in the system.
3. Copy N bytes from mapping X to a temporary buffer in conventionally-mapped memory (for which the C bit may or may not be set, as desired). N must be equal to the L1 cache line size as specified by CPUID Fn8000_0005[ECX].
4. Write N bytes from the temporary buffer to Y. Note that the initial cache refill of the line for this step will cause it to be decrypted, which corrupts the contents since it is not yet encrypted. This step restores the original contents. (If the line were evicted before this step was completed, the unwritten portion would get corrupted by the outgoing encryption, which is why the line can't be copied in-place, but rather must be copied from the temporary buffer.)
5. Repeat steps 3-4 until the entire page has been copied.
Exceptions and interrupts force control transfers from the currently-executing program to a system-software service routine that handles the interrupting event. These routines are referred to as exception handlers and interrupt handlers, or collectively as event handlers. Typically, interrupt events can be handled by the service routine transparently to the interrupted program. During the control transfer to the service routine, the processor stops executing the interrupted program and saves its return pointer. The system-software service routine that handles the exception or interrupt is responsible for saving the state of the interrupted program. This allows the processor to restart the interrupted program after system software has handled the event.

When an exception or interrupt occurs, the processor uses the interrupt vector number as an index into the interrupt-descriptor table (IDT). An IDT is used in all processor operating modes, including real mode (also called real-address mode), protected mode, and long mode.

Exceptions and interrupts come from three general sources:

- **Exceptions** occur as a result of software execution errors or other internal-processor errors. Exceptions also occur during non-error situations, such as program single stepping or address-breakpoint detection. Exceptions are considered synchronous events because they are a direct result of executing the interrupted instruction.

- **Software interrupts** occur as a result of executing interrupt instructions. Unlike exceptions and external interrupts, software interrupts allow intentional triggering of the interrupt-handling mechanism. Like exceptions, software interrupts are synchronous events.

- **External interrupts** are generated by system logic in response to an error or some other event outside the processor. They are reported over the processor bus using external signaling. External interrupts are asynchronous events that occur independently of the interrupted instruction.

Throughout this section, the term masking can refer to either disabling or delaying an interrupt. For example, masking external interrupts delays the interrupt, with the processor holding the interrupt as pending until it is unmasked. With floating-point exceptions (SSE and x87), masking prevents an interrupt from occurring and causes the processor to perform a default operation on the exception condition.

### 8.1 General Characteristics

Exceptions and interrupts have several different characteristics that depend on how events are reported and the implications for program restart.

#### 8.1.1 Precision

**Precision** describes how the exception is related to the interrupted program:

- **Precise** exceptions are reported on a predictable instruction boundary. This boundary is generally the first instruction that has not completed when the event occurs. All previous instructions (in
program order) are allowed to complete before transferring control to the event handler. The pointer to the instruction boundary is saved automatically by the processor. When the event handler completes execution, it returns to the interrupted program and restarts execution at the interrupted-instruction boundary.

- **Imprecise** exceptions are not guaranteed to be reported on a predictable instruction boundary. The boundary can be any instruction that has not completed when the interrupt event occurs. Imprecise events can be considered asynchronous, because the source of the interrupt is not necessarily related to the interrupted instruction. Imprecise exception and interrupt handlers typically collect machine-state information related to the interrupting event for reporting through system-diagnostic software. The interrupted program is not restartable.

### 8.1.2 Instruction Restart

As mentioned above, precise exceptions are reported on an instruction boundary. The instruction boundary can be reported in one of two locations:

- Most exceptions report the boundary *before* the instruction causing the exception. In this case, all previous instructions (in program order) are allowed to complete, but the interrupted instruction is not. *No program state is updated as a result of partially executing an interrupted instruction.*

- Some exceptions report the boundary *after* the instruction causing the exception. In this case, all previous instructions—including the one executing when the exception occurred—are allowed to complete. Program state can be updated when the reported boundary is after the instruction causing the exception. This is particularly true when the event occurs as a result of a task switch. In this case, the general registers, segment-selector registers, page-base address register, and LDTR are all updated by the hardware task-switch mechanism. The event handler cannot rely on the state of those registers when it begins execution and must be careful in validating the state of the segment-selector registers before restarting the interrupted task. This is not an issue in long mode, however, because the hardware task-switch mechanism is disabled in long mode.

### 8.1.3 Types of Exceptions

There are three types of exceptions, depending on whether they are precise and how they affect program restart:

- **Faults** are precise exceptions reported on the boundary before the instruction causing the exception. Generally, faults are caused by an error condition involving the faulted instruction. Any machine-state changes caused by the faulting instruction are discarded so that the instruction can be restarted. The saved rIP points to the faulting instruction.

- **Traps** are precise exceptions reported on the boundary following the instruction causing the exception. The trapped instruction is completed by the processor and all state changes are saved. The saved rIP points to the instruction following the faulting instruction.

- **Aborts** are imprecise exceptions. Because they are imprecise, aborts typically do not allow reliable program restart.
8.1.4 Masking External Interrupts

General Masking Capabilities. Software can mask the occurrence of certain exceptions and interrupts. Masking can delay or even prevent triggering of the exception-handling or interrupt-handling mechanism when an interrupt-event occurs. External interrupts are classified as maskable or nonmaskable:

- **Maskable interrupts** trigger the interrupt-handling mechanism only when RFLAGS.IF=1. Otherwise they are held pending for as long as the RFLAGS.IF bit is cleared to 0.
- **Nonmaskable interrupts** (NMI) are unaffected by the value of the RFLAGS.IF bit. However, the occurrence of an NMI masks further NMIs until an IRET instruction is executed to completion or, in the event of a task switch, to the completion of the outgoing TSS update. An exception raised during execution of the IRET prior to these points will result in NMI continuing to be masked for the duration of the exception handler, until the exception handler completes an IRET.

Masking During Stack Switches. The processor delays recognition of maskable external interrupts and debug exceptions during certain instruction sequences that are often used by software to switch stacks. The typical programming sequence used to switch stacks is:

1. Load a stack selector into the SS register.
2. Load a stack offset into the ESP register.

If an interrupting event occurs after the selector is loaded but before the stack offset is loaded, the interrupted-program stack pointer is invalid during execution of the interrupt handler.

To prevent interrupts from causing stack-pointer problems, the processor does not allow external interrupts or debug exceptions to occur until the instruction immediately following the MOV SS or POP SS instruction completes execution.

The recommended method of performing this sequence is to use the LSS instruction. LSS loads both SS and ESP, and the instruction inhibits interrupts until both registers are updated successfully.

8.1.5 Masking Floating-Point and Media Instructions

Any x87 floating-point exceptions can be masked and reported later using bits in the x87 floating-point status register (FSW) and the x87 floating-point control register (FCW). The floating-point exception-pending exception is used for unmasked x87 floating-point exceptions (see Section “#MF—x87 Floating-Point Exception-Pending (Vector 16)” on page 234).

The SIMD floating-point exception is used for unmasked SSE floating-point exceptions (see Section “#XF—SIMD Floating-Point Exception (Vector 19)” on page 236). SSE floating-point exceptions are masked using the MXCSR register. The exception mechanism is not triggered when these exceptions are masked. Instead, the processor handles the exceptions in a default manner.
8.1.6 Disabling Exceptions

Disabling an exception prevents the exception condition from being recognized, unlike masking an exception which prevents triggering the exception mechanism after the exception is recognized. Some exceptions can be disabled by system software running at CPL=0, using bits in the CR0 register or CR4 register:

- Alignment-check exception (see Section “#AC—Alignment-Check Exception (Vector 17)” on page 235).
- Device-not-available exception (see Section “#NM—Device-Not-Available Exception (Vector 7)” on page 228).
- Machine-check exception (see Section “#MC—Machine-Check Exception (Vector 18)” on page 236).

The debug-exception mechanism provides control over when specific breakpoints are enabled and disabled. See Section “Setting Breakpoints” on page 363 for more information on how breakpoint controls are used for triggering the debug-exception mechanism.

8.2 Vectors

Specific exception and interrupt sources are assigned a fixed vector-identification number (also called an “interrupt vector” or simply “vector”). The interrupt vector is used by the interrupt-handling mechanism to locate the system-software service routine assigned to the exception or interrupt. Up to 256 unique interrupt vectors are available. The first 32 vectors are reserved for predefined exception and interrupt conditions. Software-interrupt sources can trigger an interrupt using any available interrupt vector.

Table 8-1 on page 223 lists the supported interrupt vector numbers, the corresponding exception or interrupt name, the mnemonic, the source of the interrupt event, and a summary of the possible causes.
Table 8-1. Interrupt Vector Source and Cause

<table>
<thead>
<tr>
<th>Vector</th>
<th>Exception/Interrupt</th>
<th>Mnemonic</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide-by-Zero-Error</td>
<td>#DE</td>
<td>DIV, IDIV, AAM instructions</td>
</tr>
<tr>
<td>1</td>
<td>Debug</td>
<td>#DB</td>
<td>Instruction accesses and data accesses</td>
</tr>
<tr>
<td>2</td>
<td>Non-Maskable-Interrupt</td>
<td>#NMI</td>
<td>External NMI signal</td>
</tr>
<tr>
<td>3</td>
<td>Breakpoint</td>
<td>#BP</td>
<td>INT3 instruction</td>
</tr>
<tr>
<td>4</td>
<td>Overflow</td>
<td>#OF</td>
<td>INTO instruction</td>
</tr>
<tr>
<td>5</td>
<td>Bound-Range</td>
<td>#BR</td>
<td>BOUND instruction</td>
</tr>
<tr>
<td>6</td>
<td>Invalid-Opcode</td>
<td>#UD</td>
<td>Invalid instructions</td>
</tr>
<tr>
<td>7</td>
<td>Device-Not-Available</td>
<td>#NM</td>
<td>x87 instructions</td>
</tr>
<tr>
<td>8</td>
<td>Double-Fault</td>
<td>#DF</td>
<td>Exception during the handling of another exception or interrupt</td>
</tr>
<tr>
<td>9</td>
<td>Coprocessor-Segment-Overrun</td>
<td>—</td>
<td>Unsupported (Reserved)</td>
</tr>
<tr>
<td>10</td>
<td>Invalid-TSS</td>
<td>#TS</td>
<td>Task-state segment access and task switch</td>
</tr>
<tr>
<td>11</td>
<td>Segment-Not-Present</td>
<td>#NP</td>
<td>Segment register loads</td>
</tr>
<tr>
<td>12</td>
<td>Stack</td>
<td>#SS</td>
<td>SS register loads and stack references</td>
</tr>
<tr>
<td>13</td>
<td>General-Protection</td>
<td>#GP</td>
<td>Memory accesses and protection checks</td>
</tr>
<tr>
<td>14</td>
<td>Page-Fault</td>
<td>#PF</td>
<td>Memory accesses when paging enabled</td>
</tr>
<tr>
<td>15</td>
<td>Reserved</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>x87 Floating-Point Exception-Pending</td>
<td>#MF</td>
<td>x87 floating-point instructions</td>
</tr>
<tr>
<td>17</td>
<td>Alignment-Check</td>
<td>#AC</td>
<td>Misaligned memory accesses</td>
</tr>
<tr>
<td>18</td>
<td>Machine-Check</td>
<td>#MC</td>
<td>Model specific</td>
</tr>
<tr>
<td>19</td>
<td>SIMD Floating-Point</td>
<td>#XF</td>
<td>SSE floating-point instructions</td>
</tr>
<tr>
<td>20–27</td>
<td>Reserved</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Hypervisor Injection Exception</td>
<td>#HV</td>
<td>Event injection</td>
</tr>
<tr>
<td>29</td>
<td>VMM Communication Exception</td>
<td>#VC</td>
<td>Virtualization event</td>
</tr>
<tr>
<td>30</td>
<td>Security Exception</td>
<td>#SX</td>
<td>Security-sensitive event in host</td>
</tr>
<tr>
<td>31</td>
<td>Reserved</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>0–255</td>
<td>External Interrupts (Maskable)</td>
<td>#INTR</td>
<td>External interrupts</td>
</tr>
<tr>
<td>0–255</td>
<td>Software Interrupts</td>
<td>—</td>
<td>INTn instruction</td>
</tr>
</tbody>
</table>

Table 8-2 on page 224 shows how each interrupt vector is classified. Reserved interrupt vectors are indicated by the gray-shaded rows.
Table 8-2. Interrupt Vector Classification

<table>
<thead>
<tr>
<th>Vector</th>
<th>Interrupt (Exception)</th>
<th>Type</th>
<th>Precise</th>
<th>Class²</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide-by-Zero-Error</td>
<td>Fault</td>
<td>yes</td>
<td>Contributory</td>
</tr>
<tr>
<td>1</td>
<td>Debug</td>
<td>Fault or Trap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Non-Maskable-Interrupt</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Breakpoint</td>
<td>Trap</td>
<td>yes</td>
<td>Benign</td>
</tr>
<tr>
<td>4</td>
<td>Overflow</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Bound-Range</td>
<td>Fault</td>
<td>yes</td>
<td>Benign</td>
</tr>
<tr>
<td>6</td>
<td>Invalid-Opcode</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Device-Not-Available</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Double-Fault</td>
<td>Abort</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Coprocessor-Segment-Overrun</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Invalid-TSS</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Segment-Not-Present</td>
<td>Fault</td>
<td>yes</td>
<td>Contributory</td>
</tr>
<tr>
<td>12</td>
<td>Stack</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>General-Protection</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Page-Fault</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>x87 Floating-Point Exception-Pending</td>
<td>Fault</td>
<td>no</td>
<td>Benign</td>
</tr>
<tr>
<td>17</td>
<td>Alignment-Check</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Machine-Check</td>
<td>Abort</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>SIMD Floating-Point</td>
<td>Fault</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>20–27</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Hypervisor Injection Exception</td>
<td>—</td>
<td>—</td>
<td>Benign</td>
</tr>
<tr>
<td>29</td>
<td>VMM Communication Exception</td>
<td>Fault</td>
<td>yes</td>
<td>Contributory</td>
</tr>
<tr>
<td>30</td>
<td>Security Exception</td>
<td>—</td>
<td>yes</td>
<td>Contributory</td>
</tr>
<tr>
<td>31</td>
<td>Reserved</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>0–255</td>
<td>External Interrupts (Maskable)</td>
<td>—¹</td>
<td>—¹</td>
<td>Benign</td>
</tr>
<tr>
<td>0–255</td>
<td>Software Interrupts</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
</tbody>
</table>

Note:
1. External interrupts are not classified by type or whether or not they are precise.
2. See Section “#DF—Double-Fault Exception (Vector 8)” on page 228 for a definition of benign and contributory classes.

The following sections describe each interrupt in detail. The format of the error code reported by each interrupt is described in Section “Error Codes” on page 238.
8.2.1 #DE—Divide-by-Zero-Error Exception (Vector 0)

A #DE exception occurs when the denominator of a DIV instruction or an IDIV instruction is 0. A #DE also occurs if the result is too large to be represented in the destination.

#DE cannot be disabled.

Error Code Returned. None.

Program Restart. #DE is a fault-type exception. The saved instruction pointer points to the instruction that caused the #DE.

8.2.2 #DB—Debug Exception (Vector 1)

When the debug-exception mechanism is enabled, a #DB exception can occur under any of the following circumstances:

- Instruction execution.
- Instruction single stepping.
- Data read.
- Data write.
- I/O read.
- I/O write.
- Task switch.
- Debug-register access, or general detect fault (debug register access when DR7.GD=1).
- Executing the INT1 instruction (opcode 0F1h).

#DB conditions are enabled and disabled using the debug-control register, DR7 and RFLAGS.TF. Each #DB condition is described in more detail in Section “Setting Breakpoints” on page 363.

Error Code Returned. None. #DB information is returned in the debug-status register, DR6.

Program Restart. #DB can be either a fault-type or trap-type exception. In the following cases, the saved instruction pointer points to the instruction that caused the #DB:

- Instruction execution.
- Invalid debug-register access, or general detect.

In all other cases, the instruction that caused the #DB is completed, and the saved instruction pointer points to the instruction after the one that caused the #DB.

The RFLAGS.RF bit can be used to restart an instruction following an instruction breakpoint resulting in a #DB. In most cases, the processor clears RFLAGS.RF to 0 after every instruction is successfully executed. However, in the case of the IRET, JMP, CALL, and INTn (through a task gate) instructions, RFLAGS.RF is not cleared to 0 until the next instruction successfully executes.
When a non-debug exception occurs (or when a string instruction is interrupted), the processor normally sets RFLAGS.RF to 1 in the rFLAGS image that is pushed on the interrupt stack. A subsequent IRET back to the interrupted program pops the rFLAGS image off the stack and into the RFLAGS register, with RFLAGS.RF=1. The interrupted instruction executes without causing an instruction breakpoint, after which the processor clears RFLAGS.RF to 0.

However, when a #DB exception occurs, the processor clears RFLAGS.RF to 0 in the rFLAGS image that is pushed on the interrupt stack. The #DB handler has two options:

- Disable the instruction breakpoint completely.
- Set RFLAGS.RF to 1 in the interrupt-stack rFLAGS image. The instruction breakpoint condition is ignored immediately after the IRET, but reoccurs if the instruction address is accessed later, as can occur in a program loop.

8.2.3 NMI—Non-Maskable-Interrupt Exception (Vector 2)

An NMI exception occurs as a result of system logic signaling a non-maskable interrupt to the processor.

**Error Code Returned.** None.

**Program Restart.** NMI is an interrupt. The processor recognizes an NMI at an instruction boundary. The saved instruction pointer points to the instruction immediately following the boundary where the NMI was recognized.

**Masking.** NMI cannot be masked. However, when an NMI is recognized by the processor, recognition of subsequent NMIs are disabled until an IRET instruction is executed.

8.2.4 #BP—Breakpoint Exception (Vector 3)

A #BP exception occurs when an INT3 instruction is executed. The INT3 is normally used by debug software to set instruction breakpoints by replacing instruction-opcode bytes with the INT3 opcode.

#BP cannot be disabled.

**Error Code Returned.** None.

**Program Restart.** #BP is a trap-type exception. The saved instruction pointer points to the byte after the INT3 instruction. This location can be the start of the next instruction. However, if the INT3 is used to replace the first opcode bytes of an instruction, the restart location is likely to be in the middle of an instruction. In the latter case, the debug software must replace the INT3 byte with the correct instruction byte. The saved RIP instruction pointer must then be decremented by one before returning to the interrupted program. This allows the program to be restarted correctly on the interrupted-instruction boundary.
8.2.5 #OF—Overflow Exception (Vector 4)

An #OF exception occurs as a result of executing an INTO instruction while the overflow bit in RFLAGS is set to 1 (RFLAGS.OF=1).

#OF cannot be disabled.

**Error Code Returned.** None.

**Program Restart.** #OF is a trap-type exception. The saved instruction pointer points to the instruction following the INTO instruction that caused the #OF.

8.2.6 #BR—Bound-Range Exception (Vector 5)

A #BR exception can occur as a result of executing the BOUND instruction. The BOUND instruction compares an array index (first operand) with the lower bounds and upper bounds of an array (second operand). If the array index is not within the array boundary, the #BR occurs.

#BR cannot be disabled.

**Error Code Returned.** None.

**Program Restart.** #BR is a fault-type exception. The saved instruction pointer points to the BOUND instruction that caused the #BR.

8.2.7 #UD—Invalid-Opcode Exception (Vector 6)

A #UD exception occurs when an attempt is made to execute an invalid or undefined opcode. The validity of an opcode often depends on the processor operating mode. A #UD occurs under the following conditions:

- Execution of any reserved or undefined opcode in any mode.
- Execution of the UD0, UD1 or UD2 instructions.
- Use of the LOCK prefix on an instruction that cannot be locked.
- Use of the LOCK prefix on a lockable instruction with a non-memory target location.
- Execution of an instruction with an invalid-operand type.
- Execution of the SYSENTER or SYSEXIT instructions in long mode.
- Execution of any of the following instructions in 64-bit mode: AAA, AAD, AAM, AAS, BOUND, CALL (opcode 9A), DAA, DAS, DEC, INC, INTO, JMP (opcode EA), LDS, LES, POP (DS, ES, SS), POPA, PUSH (CS, DS, ES, SS), PUSHA, SALC.
- Execution of the ARPL, LAR, LLDT, LSL, LTR, SLDT, STR, VERR, or VERW instructions when protected mode is not enabled, or when virtual-8086 mode is enabled.
- Execution of any legacy SSE instruction when CR4.OSFXSR is cleared to 0. (For further information, see Section “FXSAVE/FXRSTOR Support (OSFXSR)” on page 50.
• Execution of any SSE instruction (uses YMM/XMM registers), or 64-bit media instruction (uses MMX™ registers) when CR0.EM = 1.
• Execution of any SSE floating-point instruction (uses YMM/XMM registers) that causes a numeric exception when CR4.OSXMMEXCPT = 0.
• Use of the DR4 or DR5 debug registers when CR4.DE = 1.
• Execution of RSM when not in SMM mode.

See the specific instruction description (in the other volumes) for additional information on invalid conditions.

#UD cannot be disabled.

**Error Code Returned.** None.

**Program Restart.** #UD is a fault-type exception. The saved instruction pointer points to the instruction that caused the #UD.

### 8.2.8 #NM—Device-Not-Available Exception (Vector 7)

A #NM exception occurs under any of the following conditions:

• An FWAIT/WAIT instruction is executed when CR0.MP=1 and CR0.TS=1.
• Any x87 instruction other than FWAIT is executed when CR0.EM=1.
• Any x87 instruction is executed when CR0.TS=1. The CR0.MP bit controls whether the FWAIT/WAIT instruction causes an #NM exception when TS=1.
• Any 128-bit or 64-bit media instruction when CR0.TS=1.

#NM can be enabled or disabled under the control of the CR0.MP, CR0.EM, and CR0.TS bits as described above. See Section 3.1.1 for more information on the CR0 bits used to control the #NM exception.

**Error Code Returned.** None.

**Program Restart.** #NM is a fault-type exception. The saved instruction pointer points to the instruction that caused the #NM.

### 8.2.9 #DF—Double-Fault Exception (Vector 8)

A #DF exception can occur when a second exception occurs during the handling of a prior (first) exception or interrupt handler.

Usually, the first and second exceptions can be handled sequentially without resulting in a #DF. In this case, the first exception is considered *benign*, as it does not harm the ability of the processor to handle the second exception.

In some cases, however, the first exception adversely affects the ability of the processor to handle the second exception. These exceptions contribute to the occurrence of a #DF, and are called *contributory*. 
exceptions. If a contributory exception is followed by another contributory exception, a double-fault exception occurs. Likewise, if a page fault is followed by another page fault or a contributory exception, a double-fault exception occurs.

Table 8-3 shows the conditions under which a #DF occurs. Page faults are either benign or contributory, and are listed separately. See the “Class” column in Table 8-2 on page 224 for information on whether an exception is benign or contributory.

### Table 8-3. Double-Fault Exception Conditions

<table>
<thead>
<tr>
<th>First Interrupting Event</th>
<th>Second Interrupting Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contributory Exceptions</td>
<td></td>
</tr>
<tr>
<td>• Divide-by-Zero-Error Exception</td>
<td>Invalid-TSS Exception</td>
</tr>
<tr>
<td>• Invalid-TSS Exception</td>
<td>Segment-Not-Present Exception</td>
</tr>
<tr>
<td>• Segment-Not-Present Exception</td>
<td>Stack Exception</td>
</tr>
<tr>
<td>• Stack Exception</td>
<td>General-Protection Exception</td>
</tr>
<tr>
<td>• General-Protection Exception</td>
<td></td>
</tr>
<tr>
<td>Page Fault Exception</td>
<td>Page Fault Exception</td>
</tr>
<tr>
<td></td>
<td>Invalid-TSS Exception</td>
</tr>
<tr>
<td></td>
<td>Segment-Not-Present Exception</td>
</tr>
<tr>
<td></td>
<td>Stack Exception</td>
</tr>
<tr>
<td></td>
<td>General-Protection Exception</td>
</tr>
</tbody>
</table>

If a third interrupting event occurs while transferring control to the #DF handler, the processor shuts down. Only an NMI, RESET, or INIT can restart the processor in this case. However, if the processor shuts down as it is executing an NMI handler, the processor can only be restarted with RESET or INIT.

#DF cannot be disabled.

**Error Code Returned.** Zero.

**Program Restart.** #DF is an abort-type exception. The saved instruction pointer is undefined, and the program cannot be restarted.

### 8.2.10 Coprocessor-Segment-Overrun Exception (Vector 9)

*This interrupt vector is reserved.* It is for a discontinued exception originally used by processors that supported external x87-instruction coprocessors. On those processors, the exception condition is caused by an invalid-segment or invalid-page access on an x87-instruction coprocessor-instruction operand. On current processors, this condition causes a general-protection exception to occur.

**Error Code Returned.** Not applicable.

**Program Restart.** Not applicable.
8.2.11 #TS—Invalid-TSS Exception (Vector 10)

A #TS exception occurs when an invalid reference is made to a segment selector as part of a task switch. A #TS also occurs during a privilege-changing control transfer (through a call gate or an interrupt gate), if a reference is made to an invalid stack-segment selector located in the TSS. Table 8-4 lists the conditions under which a #TS occurs and the error code returned by the exception mechanism.

#TS cannot be disabled.

Error Code Returned. See Table 8-4 for a list of error codes returned by the #TS exception.

Program Restart. #TS is a fault-type exception. If the exception occurs before loading the segment selectors from the TSS, the saved instruction pointer points to the instruction that caused the #TS. However, most #TS conditions occur due to errors with the loaded segment selectors. When an error is found with a segment selector, the hardware task-switch mechanism completes loading the new task state from the TSS, and then triggers the #TS exception mechanism. In this case, the saved instruction pointer points to the first instruction in the new task.

In long mode, a #TS cannot be caused by a task switch, because the hardware task-switch mechanism is disabled. A #TS occurs only as a result of a control transfer through a gate descriptor that results in an invalid stack-segment reference using an SS selector in the TSS. In this case, the saved instruction pointer always points to the control-transfer instruction that caused the #TS.

Table 8-4. Invalid-TSS Exception Conditions

<table>
<thead>
<tr>
<th>Selector Reference</th>
<th>Error Condition</th>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task-State Segment</td>
<td>TSS limit check on a task switch</td>
<td>TSS Selector Index</td>
</tr>
<tr>
<td></td>
<td>TSS limit check on an inner-level stack pointer</td>
<td></td>
</tr>
<tr>
<td>LDT Segment</td>
<td>LDT does not point to GDT</td>
<td>LDT Selector Index</td>
</tr>
<tr>
<td></td>
<td>LDT reference outside GDT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GDT entry is not an LDT descriptor</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDT descriptor is not present</td>
<td></td>
</tr>
<tr>
<td>Code Segment</td>
<td>CS reference outside GDT or LDT</td>
<td>CS Selector Index</td>
</tr>
<tr>
<td></td>
<td>Privilege check (conforming DPL &gt; CPL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Privilege check (non-conforming DPL ≠ CPL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Type check (CS not executable)</td>
<td></td>
</tr>
<tr>
<td>Data Segment</td>
<td>Data segment reference outside GDT or LDT</td>
<td>DS, ES, FS or GS Selector Index</td>
</tr>
<tr>
<td></td>
<td>Type check (data segment not readable)</td>
<td></td>
</tr>
<tr>
<td>Stack Segment</td>
<td>SS reference outside GDT or LDT</td>
<td>SS Selector Index</td>
</tr>
<tr>
<td></td>
<td>Privilege check (stack segment descriptor DPL ≠ CPL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Privilege check (stack segment selector RPL ≠ CPL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Type check (stack segment not writable)</td>
<td></td>
</tr>
</tbody>
</table>
8.2.12 #NP—Segment-Not-Present Exception (Vector 11)

An #NP occurs when an attempt is made to load a segment or gate with a clear present bit, as described in the following situations:

- Using the MOV, POP, LDS, LES, LFS, or LGS instructions to load a segment selector (DS, ES, FS, and GS) that references a segment descriptor containing a clear present bit (descriptor.P=0).
- Far transfer to a CS that is not present.
- Referencing a gate descriptor containing a clear present bit.
- Referencing a TSS descriptor containing a clear present bit. This includes attempts to load the TSS descriptor using the LTR instruction.
- Attempting to load a descriptor containing a clear present bit into the LDTR using the LLDT instruction.
- Loading a segment selector (CS, DS, ES, FS, or GS) as part of a task switch, with the segment descriptor referenced by the segment selector having a clear present bit. In long mode, an #NP cannot be caused by a task switch, because the hardware task-switch mechanism is disabled.

When loading a stack-segment selector (SS) that references a descriptor with a clear present bit, a stack exception (#SS) occurs. For information on the #SS exception, see the next section, “#SS—Stack Exception (Vector 12).”

#NP cannot be disabled.

Error Code Returned. The segment-selector index of the segment descriptor causing the #NP exception.

Program Restart. #NP is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that loaded the segment selector resulting in the #NP. See Section “Exceptions During a Task Switch” on page 238 for a description of the consequences when this exception occurs during a task switch.

8.2.13 #SS—Stack Exception (Vector 12)

An #SS exception can occur in the following situations:

- Implied stack references in which the stack address is not in canonical form. Implied stack references include all push and pop instructions, and any instruction using RSP or RBP as a base register.
- Attempting to load a stack-segment selector that references a segment descriptor containing a clear present bit (descriptor.P=0).
- Any stack access that fails the stack-limit check.

#SS cannot be disabled.

Error Code Returned. The error code depends on the cause of the #SS, as shown in Table 8-5 on page 232:
### Table 8-5. Stack Exception Error Codes

<table>
<thead>
<tr>
<th>Stack Exception Cause</th>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack-segment descriptor present bit is clear</td>
<td>SS Selector Index</td>
</tr>
<tr>
<td>Stack-limit violation</td>
<td>0</td>
</tr>
<tr>
<td>Stack reference using a non-canonical address</td>
<td>0</td>
</tr>
</tbody>
</table>

**Program Restart.** #SS is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that caused the #SS. See Section “Exceptions During a Task Switch” on page 238 for a description of the consequences when this exception occurs during a task switch.

### 8.2.14 #GP—General-Protection Exception (Vector 13)

Table 8-6 describes the general situations that can cause a #GP exception. The table is not an exhaustive, detailed list of #GP conditions, but rather a guide to the situations that can cause a #GP. If an invalid use of an AMD64 architectural feature results in a #GP, the specific cause of the exception is described in detail in the section describing the architectural feature.

#GP cannot be disabled.

**Error Code Returned.** As shown in Table 8-6, a selector index is reported as the error code if the #GP is due to a segment-descriptor access. In all other cases, an error code of 0 is returned.

**Program Restart.** #GP is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that caused the #GP. See Section “Exceptions During a Task Switch” on page 238 for a description of the consequences when this exception occurs during a task switch.

### Table 8-6. General-Protection Exception Conditions

<table>
<thead>
<tr>
<th>Error Condition</th>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any segment privilege-check violation, while loading a segment register.</td>
<td></td>
</tr>
<tr>
<td>Any segment type-check violation, while loading a segment register.</td>
<td></td>
</tr>
<tr>
<td>Loading a null selector into the CS, SS, or TR register.</td>
<td></td>
</tr>
<tr>
<td>Accessing a gate-descriptor containing a null segment selector.</td>
<td></td>
</tr>
<tr>
<td>Referencing an LDT descriptor or TSS descriptor located in the LDT.</td>
<td>Selector Index</td>
</tr>
<tr>
<td>Attempting a control transfer to a busy TSS (except IRET).</td>
<td></td>
</tr>
<tr>
<td>In 64-bit mode, loading a non-canonical base address into the GDTR or IDTR.</td>
<td></td>
</tr>
<tr>
<td>In long mode, accessing a system or call-gate descriptor whose extended type field is not 0.</td>
<td></td>
</tr>
<tr>
<td>In long mode, accessing a system descriptor containing a non-canonical base address.</td>
<td></td>
</tr>
<tr>
<td>In long mode, accessing a gate descriptor containing a non-canonical offset.</td>
<td></td>
</tr>
<tr>
<td>In long mode, accessing a gate descriptor that does not point to a 64-bit code segment.</td>
<td></td>
</tr>
<tr>
<td>In long mode, accessing a 16-bit gate descriptor.</td>
<td></td>
</tr>
<tr>
<td>In long mode, attempting a control transfer to a TSS or task gate.</td>
<td></td>
</tr>
</tbody>
</table>
8.2.15  #PF—Page-Fault Exception (Vector 14)

A #PF exception can occur during a memory access in any of the following situations:

- A page-translation-table entry or physical page involved in translating the memory access is not present in physical memory. This is indicated by a cleared present bit (P=0) in the translation-table entry.
- An attempt is made by the processor to load the instruction TLB with a translation for a non-executable page.
- The memory access fails the paging-protection checks (user/supervisor, read/write, or both).
- A reserved bit in one of the page-translation-table entries is set to 1. A #PF occurs for this reason only when CR4.PSE=1 or CR4.PAE=1.
- A data access to a user-mode address caused a protection key violation.

#PF cannot be disabled.

CR2 Register. The virtual (linear) address that caused the #PF is stored in the CR2 register. The legacy CR2 register is 32 bits long. The CR2 register in the AMD64 architecture is 64 bits long, as shown in Figure 8-1 on page 234. In AMD64 implementations, when either software or a page fault causes a write to the CR2 register, only the low-order 32 bits of CR2 are used in legacy mode; the processor clears the high-order 32 bits.

---

Table 8-6. General-Protection Exception Conditions (continued)

<table>
<thead>
<tr>
<th>Error Condition</th>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any segment limit-check or non-canonical address violation (except when using the SS register).</td>
<td></td>
</tr>
<tr>
<td>Accessing memory using a null segment register.</td>
<td></td>
</tr>
<tr>
<td>Writing memory using a read-only segment register.</td>
<td></td>
</tr>
<tr>
<td>Attempting to execute an SSE instruction specifying an unaligned memory operand.</td>
<td></td>
</tr>
<tr>
<td>Attempting to execute code that is past the CS segment limit or at a non-canonical RIP.</td>
<td></td>
</tr>
<tr>
<td>Executing a privileged instruction while CPL &gt; 0.</td>
<td></td>
</tr>
<tr>
<td>Executing an instruction that is more than 15 bytes long.</td>
<td></td>
</tr>
<tr>
<td>Writing a 1 into any register field that is reserved, must be zero (MBZ).</td>
<td></td>
</tr>
<tr>
<td>Using WRMSR to write a read-only MSR.</td>
<td>0</td>
</tr>
<tr>
<td>Using WRMSR to write a non-canonical value into an MSR that must be canonical.</td>
<td></td>
</tr>
<tr>
<td>Using WRMSR to set an invalid type encoding in an MTRR or the PAT MSR.</td>
<td>0</td>
</tr>
<tr>
<td>Enabling paging while protected mode is disabled.</td>
<td></td>
</tr>
<tr>
<td>Setting CR0.NW=1 while CR0.CD=0.</td>
<td></td>
</tr>
<tr>
<td>Any long-mode consistency-check violation.</td>
<td></td>
</tr>
</tbody>
</table>
Figure 8-1. Control Register 2 (CR2)

Error Code Returned. The page-fault error code is pushed onto the page-fault exception-handler stack. See Section “Page-Fault Error Code” on page 239 for a description of this error code.

Program Restart. #PF is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that caused the #PF. See Section “Exceptions During a Task Switch” on page 238 for a description of what can happen if this exception occurs during a task switch.

8.2.16 #MF—x87 Floating-Point Exception-Pending (Vector 16)

The #MF exception is used to handle unmasked x87 floating-point exceptions. An #MF occurs when all of the following conditions are true:

- CR0.NE=1.
- An unmasked x87 floating-point exception is pending. This is indicated by an exception bit in the x87 floating-point status-word register being set to 1
- The corresponding mask bit in the x87 floating-point control-word register is cleared to 0.
- The FWAIT/WAIT instruction or any waiting floating-point instruction is executed.

If there is an exception mask bit (in the FPU control word) set, the exception is not reported. Instead, the x87-instruction unit responds in a default manner and execution proceeds normally.

The x87 floating-point exceptions reported by the #MF exception are (including mnemonics):

- IE—Invalid-operation exception (also called #I), which is either:
  - IE alone—Invalid arithmetic-operand exception (also called #IA), or
  - SF and IE together—x87 Stack-fault exception (also called #IS).
- DE—Denormalized-operand exception (also called #D).
- ZE—Zero-divide exception (also called #Z).
- OE—Overflow exception (also called #O).
- UE—Underflow exception (also called #U).
- PE—Precision exception (also called #P or inexact-result exception).

Error Code Returned. None. Exception information is provided by the x87 status-word register. See “x87 Floating-Point Programming” in Volume 1 for more information on using this register.

Program Restart. #MF is a fault-type exception. The #MF exception is not precise, because multiple instructions and exceptions can occur before the #MF handler is invoked. Also, the saved instruction
pointer does not point to the instruction that caused the exception resulting in the #MF. Instead, the saved instruction pointer points to the x87 floating-point instruction or FWAIT/WAIT instruction that is about to be executed when the #MF occurs. The address of the last instruction that caused an x87 floating-point exception is in the x87 instruction-pointer register. See “x87 Floating-Point Programming” in Volume 1 for information on accessing this register.

**Masking.** Each type of x87 floating-point exception can be masked by setting the appropriate bits in the x87 control-word register. See “x87 Floating-Point Programming” in Volume 1 for more information on using this register.

#MF can also be disabled by clearing the CR0.NE bit to 0. See Section “Numeric Error (NE) Bit” on page 44 for more information on using this bit.

**8.2.17 #AC—Alignment-Check Exception (Vector 17)**

An #AC exception occurs when an unaligned-memory data reference is performed while alignment checking is enabled.

After a processor reset, #AC exceptions are disabled. Software enables the #AC exception by setting the following register bits:

- CR0.AM=1.
- RFLAGS.AC=1.

When the above register bits are set, an #AC can occur only when CPL=3. #AC never occurs when CPL < 3.

Table 8-7 lists the data types and the alignment boundary required to avoid an #AC exception when the mechanism is enabled.

<table>
<thead>
<tr>
<th>Supported Data Type</th>
<th>Required Alignment (Byte Boundary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word</td>
<td>2</td>
</tr>
<tr>
<td>Doubleword</td>
<td>4</td>
</tr>
<tr>
<td>Quadword</td>
<td>8</td>
</tr>
<tr>
<td>Bit string</td>
<td>2, 4 or 8 (depends on operand size)</td>
</tr>
<tr>
<td>256-bit media</td>
<td>16</td>
</tr>
<tr>
<td>128-bit media</td>
<td>16</td>
</tr>
<tr>
<td>64-bit media</td>
<td>8</td>
</tr>
<tr>
<td>Segment selector</td>
<td>2</td>
</tr>
<tr>
<td>32-bit near pointer</td>
<td>4</td>
</tr>
<tr>
<td>32-bit far pointer</td>
<td>2</td>
</tr>
<tr>
<td>48-bit far pointer</td>
<td>4</td>
</tr>
</tbody>
</table>
8.2.18 #MC—Machine-Check Exception (Vector 18)

The #MC exception is model specific. Processor implementations are not required to support the #MC exception, and those implementations that do support #MC can vary in how the #MC exception mechanism works.

The exception is enabled by setting CR4.MCE to 1. The machine-check architecture can include model-specific masking for controlling the reporting of some errors. Refer to Chapter 9, “Machine Check Architecture,” for more information.

Error Code Returned. None. Error information is provided by model-specific registers (MSRs) defined by the machine-check architecture.

Program Restart. #MC is an abort-type exception. There is no reliable way to restart the program. If the EIPV flag (EIP valid) is set in the MCG_Status MSR, the saved CS and rIP point to the instruction that caused the error. If EIP is clear, the CS:rIP of the instruction causing the failure is not known or the machine check is not related to a specific instruction.

8.2.19 #XF—SIMD Floating-Point Exception (Vector 19)

The #XF exception is used to handle unmasked SSE floating-point exceptions. A #XF exception occurs when all of the following conditions are true:

- A SSE floating-point exception occurs. The exception causes the processor to set the appropriate exception-status bit in the MXCSR register to 1.
- The exception-mask bit in the MXCSR that corresponds to the SSE floating-point exception is clear (=0).
- CR4.OSXMMEXCPT=1, indicating that the operating system supports handling of SSE floating-point exceptions.

The exception-mask bits are used by software to specify the handling of SSE floating-point exceptions. When the corresponding mask bit is cleared to 0, an exception occurs under the control of
the CR4.OSXMMEXCPT bit. However, if the mask bit is set to 1, the SSE floating-point unit responds in a default manner and execution proceeds normally.

The CR4.OSXMMEXCPT bit specifies the interrupt vector to be taken when an unmasked SSE floating-point exception occurs. When CR4.OSXMMEXCPT=1, the #XF interrupt vector is taken when an exception occurs. When CR4.OSXMMEXCPT=0, the #UD (undefined opcode) interrupt vector is taken when an exception occurs.

The SSE floating-point exceptions reported by the #XF exception are (including mnemonics):

• IE—Invalid-operation exception (also called #I).
• DE—Denormalized-operand exception (also called #D).
• ZE—Zero-divide exception (also called #Z).
• OE—Overflow exception (also called #O).
• UE—Underflow exception (also called #U).
• PE—Precision exception (also called #P or inexact-result exception).

Each type of SSE floating-point exception can be masked by setting the appropriate bits in the MXCSR register. #XF can also be disabled by clearing the CR4.OSXMMEXCPT bit to 0.

**Error Code Returned.** None. Exception information is provided by the SSE floating-point MXCSR register. See “Instruction Reference” in Volume 4 for more information on using this register.

**Program Restart.** #XF is a fault-type exception. Unlike the #MF exception, the #XF exception is precise. The saved instruction pointer points to the instruction that caused the #XF.

### 8.2.20 #HV—Hypervisor Injection Exception (Vector 28)

The #HV exception may be injected by the hypervisor into a secure guest VM to notify the VM of pending events. See Section 15.36.16 for details.

### 8.2.21 #VC—VMM Communication Exception (Vector 29)

The #VC exception is generated when certain events occur inside a secure guest VM. See Section 15.35.5 for more details.

### 8.2.22 #SX—Security Exception (Vector 30)

The #SX exception is generated by security-sensitive events under SVM. See Section 15.28 for details.

### 8.2.23 User-Defined Interrupts (Vectors 32–255)

User-defined interrupts can be initiated either by system logic or software. They occur when:

• System logic signals an external interrupt request to the processor. The signaling mechanism and the method of communicating the interrupt vector to the processor are implementation dependent.
• Software executes an INTn instruction. The INTn instruction operand provides the interrupt vector number.

Both methods can be used to initiate an interrupt into vectors 0 through 255. However, because vectors 0 through 31 are defined or reserved by the AMD64 architecture, software should not use vectors in this range for purposes other than their defined use.

**Error Code Returned.** None.

**Program Restart.** The saved instruction pointer depends on the interrupt source:

• External interrupts are recognized on instruction boundaries. The saved instruction pointer points to the instruction immediately following the boundary where the external interrupt was recognized.

• If the interrupt occurs as a result of executing the INTn instruction, the saved instruction pointer points to the instruction after the INTn.

**Masking.** The ability to mask user-defined interrupts depends on the interrupt source:

• External interrupts can be masked using the RFLAGS.IF bit. Setting RFLAGS.IF to 1 enables external interrupts, while clearing RFLAGS.IF to 0 inhibits them.

• Software interrupts (initiated by the INTn instruction) cannot be disabled.

### 8.3 Exceptions During a Task Switch

An exception can occur during a task switch while loading a segment selector. Page faults can also occur when accessing a TSS. In these cases, the hardware task-switch mechanism completes loading the new task state from the TSS, and then triggers the appropriate exception mechanism. No other checks are performed. When this happens, the saved instruction pointer points to the first instruction in the new task.

In long mode, an exception cannot occur during a task switch, because the hardware task-switch mechanism is disabled.

### 8.4 Error Codes

The processor exception-handling mechanism reports error and status information for some exceptions using an error code. The error code is pushed onto the stack by the exception-mechanism during the control transfer into the exception handler. The error code has two formats: a selector format for most error-reporting exceptions, and a page-fault format for page faults. These formats are described in the following sections.

#### 8.4.1 Selector-Error Code

Figure 8-2 shows the format of the selector-error code.
**Figure 8-2. Selector Error Code**

The information reported by the selector-error code includes:

- **EXT**—Bit 0. If this bit is set to 1, the exception source is external to the processor. If cleared to 0, the exception source is internal to the processor.

- **IDT**—Bit 1. If this bit is set to 1, the error-code selector-index field references a gate descriptor located in the interrupt-descriptor table (IDT). If cleared to 0, the selector-index field references a descriptor in either the global-descriptor table (GDT) or local-descriptor table (LDT), as indicated by the TI bit.

- **TI**—Bit 2. If this bit is set to 1, the error-code selector-index field references a descriptor in the LDT. If cleared to 0, the selector-index field references a descriptor in the GDT. The TI bit is relevant only when the IDT bit is cleared to 0.

- **Selector Index**—Bits 15:3. The selector-index field specifies the index into either the GDT, LDT, or IDT, as specified by the IDT and TI bits.

Some exceptions return a zero in the selector-error code.

**8.4.2 Page-Fault Error Code**

Figure 8-3 shows the format of the page-fault error code.

**Figure 8-3. Page-Fault Error Code**

The information reported by the page-fault error code includes:

- **P**—Bit 0. If this bit is cleared to 0, the page fault was caused by a not-present page. If this bit is set to 1, the page fault was caused by a page-protection violation.

- **R/W**—Bit 1. If this bit is cleared to 0, the access that caused the page fault is a memory read. If this bit is set to 1, the memory access that caused the page fault was a write. This bit does not necessarily indicate the cause of the page fault was a read or write violation.

- **U/S**—Bit 2. If this bit is cleared to 0, an access in supervisor mode (CPL=0, 1, or 2) caused the page fault. If this bit is set to 1, an access in user mode (CPL=3) caused the page fault. This bit does not necessarily indicate the cause of the page fault was a privilege violation.
• **RSV**—Bit 3. If this bit is set to 1, the page fault is a result of the processor reading a 1 from a reserved field within a page-translation-table entry. This type of page fault occurs only when CR4.PSE=1 or CR4.PAE=1. If this bit is cleared to 0, the page fault was not caused by the processor reading a 1 from a reserved field.

• **I/D**—Bit 4. If this bit is set to 1, it indicates that the access that caused the page fault was an instruction fetch. Otherwise, this bit is cleared to 0. This bit is only defined if no-execute feature is enabled (EFER.NXE=1 && CR4.PAE=1).

• **RMP**—Bit 31. If this bit is set to 1, the page fault is a result of the processor encountering an RMP violation. This type of page fault only occurs when SYSCFG[SecureNestedPagingEn]=1. If this bit is cleared to 0, the page fault was not caused by an RMP violation. See section 15.36.10 for additional information.

• **PK**—Bit 5. If this bit is set to 1, it indicates that a data access to a user-mode address caused a protection key violation. This fault only occurs if memory protection keys are enabled (CR4.PKE=1).

### 8.5 Priorities

To allow for consistent handling of multiple-interrupt conditions, simultaneous interrupts are prioritized by the processor. The AMD64 architecture defines priorities between groups of interrupts, and interrupt prioritization within a group is implementation dependent. Table 8-8 shows the interrupt priorities defined by the AMD64 architecture.

When simultaneous interrupts occur, the processor transfers control to the highest-priority interrupt handler. Lower-priority interrupts from external sources are held pending by the processor, and they are handled after the higher-priority interrupt is handled. Lower-priority interrupts that result from internal sources are discarded. Those interrupts reoccur when the high-priority interrupt handler completes and transfers control back to the interrupted instruction. Software interrupts are discarded as well, and reoccur when the software-interrupt instruction is restarted.

### Table 8-8. Simultaneous Interrupt Priorities

<table>
<thead>
<tr>
<th>Interrupt Priority</th>
<th>Interrupt Condition</th>
<th>Interrupt Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>(High) 0</td>
<td>Processor Reset</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Machine-Check Exception</td>
<td>18</td>
</tr>
<tr>
<td>1</td>
<td>External Processor Initialization (INIT)</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>SMI Interrupt</td>
<td>32–255</td>
</tr>
<tr>
<td></td>
<td>External Clock Stop (Stpclk)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Data, and I/O Breakpoint (Debug Register)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Single-Step Execution Instruction Trap (RFLAGS.TF=1)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Non-Maskable Interrupt</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Maskable External Interrupt (INTR)</td>
<td></td>
</tr>
</tbody>
</table>
8.5.1 Floating-Point Exception Priorities

Floating-point exceptions (SSE and x87 floating-point) can be handled in one of two ways:

- Unmasked exceptions are reported in the appropriate floating-point status register, and a software-interrupt handler is invoked. See Section “#MF—x87 Floating-Point Exception-Pending (Vector 16)” on page 234 and Section “#XF—SIMD Floating-Point Exception (Vector 19)” on page 236 for more information on the floating-point interrupts.
- Masked exceptions are also reported in the appropriate floating-point status register. Instead of transferring control to an interrupt handler, however, the processor handles the exception in a default manner and execution proceeds.

If the processor detects more than one exception while executing a single floating-point instruction, it prioritizes the exceptions in a predictable manner. When responding in a default manner to masked exceptions, it is possible that the processor acts only on the high-priority exception and ignores lower-
priority exceptions. In the case of vector (SIMD) floating-point instructions, priorities are set on sub-operations, not across all operations. For example, if the processor detects and acts on a QNaN operand in one sub-operation, the processor can still detect and act on a denormal operand in another sub-operation.

When reporting SSE floating-point exceptions before taking an interrupt or handling them in a default manner, the processor first classifies the exceptions as follows:

- **Input exceptions** include SNaN operand (#I), invalid operation (#I), denormal operand (#D), or zero-divide (#Z). Using a NaN operand with a maximum, minimum, compare, or convert instruction is also considered an input exception.
- **Output exceptions** include numeric overflow (#O), numeric underflow (#U), and precision (#P).

Using the above classification, the processor applies the following procedure to report the exceptions:

1. The exceptions for all sub-operations are prioritized.
2. The exception conditions for all sub-operations are logically ORed together to form a single set of exceptions covering all operations. For example, if two sub-operations produce a denormal result, only one denormal exception is reported.
3. If the set of exceptions includes any unmasked input exceptions, all input exceptions are reported in MCXSR, and no output exceptions are reported. Otherwise, all input and output exceptions are reported in MCXSR.
4. If any exceptions are unmasked, control is transferred to the appropriate interrupt handler.

Table 8-9 on page 242 lists the priorities for simultaneous floating-point exceptions.

<table>
<thead>
<tr>
<th>Exception Priority</th>
<th>Exception Condition</th>
<th>Exception Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(High) 0</td>
<td>SNaN Operand</td>
<td>#I</td>
</tr>
<tr>
<td></td>
<td>NaN Operand of Maximum, Minimum, Compare, and Convert Instructions (Vector Floating-Point)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stack Overflow (x87 Floating-Point)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stack Underflow (x87 Floating-Point)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>QNaN Operand</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>Invalid Operation (Remaining Conditions)</td>
<td>#I</td>
</tr>
<tr>
<td></td>
<td>Zero Divide</td>
<td>#Z</td>
</tr>
<tr>
<td>3</td>
<td>Denormal Operand</td>
<td>#D</td>
</tr>
<tr>
<td>4</td>
<td>Numeric Overflow</td>
<td>#O</td>
</tr>
<tr>
<td></td>
<td>Numeric Underflow</td>
<td>#U</td>
</tr>
<tr>
<td>5 (Low)</td>
<td>Precision</td>
<td>#P</td>
</tr>
</tbody>
</table>
8.5.2 External Interrupt Priorities

The AMD64 architecture allows software to define up to 15 external interrupt-priority classes. Priority classes are numbered from 1 to 15, with priority-class 1 being the lowest and priority-class 15 the highest. The organization of these priority classes is implementation dependent. A typical method is to use the upper four bits of the interrupt vector number to define the priority. Thus, interrupt vector 53h has a priority of 5 and interrupt vector 37h has a priority of 3.

A new control register (CR8) is introduced by the AMD64 architecture for managing priority classes. This register, called the task-priority register (TPR), uses its four low-order bits to specify a task priority. The remaining 60 bits are reserved and must be written with zeros. Figure 8-4 shows the format of the TPR.

The TPR is available only in 64-bit mode.

![Figure 8-4. Task Priority Register (CR8)](image)

System software can use the TPR register to temporarily block low-priority interrupts from interrupting a high-priority task. This is accomplished by loading TPR with a value corresponding to the highest-priority interrupt that is to be blocked. For example, loading TPR with a value of 9 (1001b) blocks all interrupts with a priority class of 9 or less, while allowing all interrupts with a priority class of 10 or more to be recognized. Loading TPR with 0 enables all external interrupts. Loading TPR with 15 (1111b) disables all external interrupts. The TPR is cleared to 0 on reset.

System software reads and writes the TPR using a MOV CR8 instruction. The MOV CR8 instruction requires a privilege level of 0. Programs running at any other privilege level cannot read or write the TPR, and an attempt to do so results in a general-protection exception (#GP).

A serializing instruction is not required after loading the TPR, because a new priority level is established when the MOV instruction completes execution. For example, assume two sequential TPR loads are performed, in which a low value is first loaded into TPR and immediately followed by a load of a higher value. Any pending, lower-priority interrupt enabled by the first MOV CR8 is recognized between the two MOVs.

The TPR is an architectural abstraction of the interrupt controller (IC), which prioritizes and manages external interrupt delivery to the processor. The IC can be an external system device, or it can be integrated on the chip like the local advanced programmable interrupt controller (APIC). Typically, the IC contains a priority mechanism similar, if not identical to, the TPR. The IC, however, is implementation dependent, and the underlying priority mechanisms are subject to change. The TPR, by contrast, is part of the AMD64 architecture.
**Effect of IC on TPR.** The features of the implementation-specific IC can impact the operation of the TPR. For example, the TPR might affect interrupt delivery only if the IC is enabled. Also, the mapping of an external interrupt to a specific interrupt priority is an implementation-specific behavior of the IC.

While the CR8 register provides the same functionality as the TPR at offset 80h of the local APIC, software should only use one mechanism to access the TPR. For example, updating the TPR with a write to the local APIC offset 0x80 but then reading it with a MOV CR8 is not guaranteed to return the same value that was written by the local APIC write.

### 8.6 Real-Mode Interrupt Control Transfers

In real mode, the IDT is a table of 4-byte entries, one entry for each of the 256 possible interrupts implemented by the system. The real mode IDT is often referred to as an *interrupt vector table*, or IVT. Table entries contain a far pointer (CS:IP pair) to an exception or interrupt handler. The base of the IDT is stored in the IDTR register, which is loaded with a value of 00h during a processor reset. Figure 8-5 on page 244 shows how the real-mode interrupt handler is located by the interrupt mechanism.

![Figure 8-5. Real-Mode Interrupt Control Transfer](image)

When an exception or interrupt occurs in real mode, the processor performs the following:

1. Pushes the FLAGS register (EFLAGS[15:0]) onto the stack.
2. Clears EFLAGS.IF to 0 and EFLAGS.TF to 0.
3. Saves the CS register and IP register (RIP[15:0]) by pushing them onto the stack.
4. Locates the interrupt-handler pointer (CS:IP) in the IDT by scaling the interrupt vector by four and adding the result to the value in the IDTR.

5. Transfers control to the interrupt handler referenced by the CS:IP in the IDT.

Figure 8-6 on page 245 shows the stack after control is transferred to the interrupt handler in real mode.

An IRET instruction is used to return to the interrupted program. When an IRET is executed, the processor performs the following:

1. Pops the saved CS value off the stack and into the CS register. The saved IP value is popped into RIP[15:0].

2. Pops the FLAGS value off of the stack and into EFLAGS[15:0].

3. Execution begins at the saved CS.IP location.

### 8.7 Legacy Protected-Mode Interrupt Control Transfers

In protected mode, the interrupt mechanism transfers control to an exception or interrupt handler through gate descriptors. In protected mode, the IDT is a table of 8-byte gate entries, one for each of the 256 possible interrupt vectors implemented by the system. Three gate types are allowed in the IDT:

- Interrupt gates.
- Trap gates.
- Task gates.

If a reference is made to any other descriptor type in the IDT, a general-protection exception (#GP) occurs.
Interrupt-gate control transfers are similar to CALLs and JMPs through call gates. The interrupt mechanism uses gates (interrupt, trap, and task) to establish protected entry-points into the exception and interrupt handlers.

The remainder of this chapter discusses control transfers through interrupt gates and trap gates. If the gate descriptor in the IDT is a task gate, a TSS-segment selector is referenced, and a task switch occurs. See Chapter 12, “Task Management,” for more information on the hardware task-switch mechanism.

8.7.1 Locating the Interrupt Handler

When an exception or interrupt occurs, the processor scales the interrupt vector number by eight and uses the result as an offset into the IDT. If the gate descriptor referenced by the IDT offset is an interrupt gate or a trap gate, it contains a segment-selector and segment-offset field (see Section “Legacy Segment Descriptors” on page 82 for a detailed description of the gate-descriptor format and fields). These two fields perform the same function as the pointer operand in a far control-transfer instruction. The gate-descriptor segment-selector field points to the target code-segment descriptor located in either the GDT or LDT. The gate-descriptor segment-offset field is the instruction-pointer offset into the interrupt-handler code segment. The code-segment base taken from the code-segment descriptor is added to the gate-descriptor segment-offset field to create the interrupt-handler virtual address (linear address).

Figure 8-7 on page 247 shows how the protected-mode interrupt handler is located by the interrupt mechanism.
8.7.2 Interrupt To Same Privilege

When a control transfer to an exception or interrupt handler at the same privilege level occurs (through an interrupt gate or a trap gate), the processor performs the following:

1. Pushes the EFLAGS register onto the stack.
2. Clears the TF, NT, RF, and VM bits in EFLAGS to 0.
3. The processor handles EFLAGS.IF based on the gate-descriptor type:
   - If the gate descriptor is an interrupt gate, EFLAGS.IF is cleared to 0.
   - If the gate descriptor is a trap gate, EFLAGS.IF is not modified.

4. Saves the return CS register and EIP register (RIP[31:0]) by pushing them onto the stack. The CS value is padded with two bytes to form a doubleword.

5. If the interrupt has an associated error code, the error code is pushed onto the stack.

6. The CS register is loaded from the segment-selector field in the gate descriptor, and the EIP is loaded from the offset field in the gate descriptor.

7. The interrupt handler begins executing with the instruction referenced by new CS:EIP.

Figure 8-8 shows the stack after control is transferred to the interrupt handler.

![Figure 8-8. Stack After Interrupt to Same Privilege Level](image)

8.7.3 Interrupt To Higher Privilege

When a control transfer to an exception or interrupt handler running at a higher privilege occurs (numerically lower CPL value), the processor performs a stack switch using the following steps:

1. The target CPL is read by the processor from the target code-segment DPL and used as an index into the TSS for selecting the new stack pointer (SS:ESP). For example, if the target CPL is 1, the processor selects the SS:ESP for privilege-level 1 from the TSS.

2. Pushes the return stack pointer (old SS:ESP) onto the new stack. The SS value is padded with two bytes to form a doubleword.

3. Pushes the EFLAGS register onto the new stack.

4. Clears the following EFLAGS bits to 0: TF, NT, RF, and VM.
5. The processor handles the EFLAGS.IF bit based on the gate-descriptor type:
   - If the gate descriptor is an interrupt gate, EFLAGS.IF is cleared to 0.
   - If the gate descriptor is a trap gate, EFLAGS.IF is not modified.

6. Saves the return-address pointer (CS:EIP) by pushing it onto the stack. The CS value is padded with two bytes to form a doubleword.

7. If the interrupt vector number has an error code associated with it, the error code is pushed onto the stack.

8. The CS register is loaded from the segment-selector field in the gate descriptor, and the EIP is loaded from the offset field in the gate descriptor.

9. The interrupt handler begins executing with the instruction referenced by new CS:EIP.

Figure 8-9 shows the new stack after control is transferred to the interrupt handler.

### Interrupt-Handler Stack

<table>
<thead>
<tr>
<th>With Error Code</th>
<th>With No Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return SS</td>
<td>Return SS</td>
</tr>
<tr>
<td>Return ESP</td>
<td>Return ESP</td>
</tr>
<tr>
<td>Return EFLAGS</td>
<td>Return EFLAGS</td>
</tr>
<tr>
<td>Return CS</td>
<td>Return CS</td>
</tr>
<tr>
<td>Return EIP</td>
<td>Return EIP</td>
</tr>
<tr>
<td>Error Code</td>
<td>Error Code</td>
</tr>
</tbody>
</table>

---

8.7.4 Privilege Checks

Before loading the CS register with the interrupt-handler code-segment selector (located in the gate descriptor), the processor performs privilege checks similar to those performed on call gates. The checks are performed when either conforming or nonconforming interrupt handlers are referenced:

1. The processor reads the gate DPL from the interrupt-gate or trap-gate descriptor. The gate DPL is the *minimum privilege-level* (numerically-highest value) needed by a program to access the gate. The processor compares the CPL with the gate DPL. The CPL must be numerically less-than or equal-to the gate DPL for this check to pass.
2. The processor compares the CPL with the interrupt-handler code-segment DPL. For this check to pass, the CPL must be numerically greater-than or equal-to the code-segment DPL. This check prevents control transfers to less-privileged interrupt handlers.

Unlike call gates, no RPL comparison takes place. This is because the gate descriptor is referenced in the IDT using the interrupt vector number rather than a selector, and no RPL field exists in the interrupt vector number.

Exception and interrupt handlers should be made reachable from software running at any privilege level that requires them. If the gate DPL value is too low (requiring more privilege), or the interrupt-handler code-segment DPL is too high (runs at lower privilege), the interrupt control transfer can fail the privilege checks. Setting the gate DPL=3 and interrupt-handler code-segment DPL=0 makes the exception handler or interrupt handler reachable from any privilege level.

Figure 8-10 on page 251 shows two examples of interrupt privilege checks. In Example 1, both privilege checks pass:

- The interrupt-gate DPL is at the lowest privilege (3), which means that software running at any privilege level (CPL) can access the interrupt gate.
- The interrupt-handler code segment is at the highest-privilege level, as indicated by DPL=0. This means software running at any privilege can enter the interrupt handler through the interrupt gate.
In Example 2, both privilege checks fail:

- The interrupt-gate DPL specifies that only software running at privilege-level 0 can access the gate. The current program does not have a high enough privilege level to access the interrupt gate, since its CPL is set at 2.

Figure 8-10. Privilege-Check Examples for Interrupts
• The interrupt handler has a lower privilege (DPL=3) than the currently-running software (CPL=2).
  Transitions from more-privileged software to less-privileged software are not allowed, so this
  privilege check fails as well.

Although both privilege checks fail, only one such failure is required to deny access to the interrupt
handler.

8.7.5 Returning From Interrupt Procedures

A return to an interrupted program should be performed using the IRET instruction. An IRET is a far
return to a different code segment, with or without a change in privilege level. The actions of an IRET
in both cases are described in the following sections.

IRET, Same Privilege. Before performing the IRET, the stack pointer must point to the return EIP. If
there was an error code pushed onto the stack as a result of the exception or interrupt, that error code
should have been popped off the stack earlier by the handler. The IRET reverses the actions of the
interrupt mechanism:

1. Pops the return pointer off of the stack, loading both the CS register and EIP register (RIP[31:0])
   with the saved values. The return code-segment RPL is read by the processor from the CS value
   stored on the stack to determine that an equal-privilege control transfer is occurring.
2. Pops the saved EFLAGS image off of the stack and into the EFLAGS register.
3. Transfers control to the return program at the target CS:EIP.

IRET, Less Privilege. If an IRET changes privilege levels, the return program must be at a lower
privilege than the interrupt handler. The IRET in this case causes a stack switch to occur:

1. The return pointer is popped off of the stack, loading both the CS register and EIP register
   (RIP[31:0]) with the saved values. The return code-segment RPL is read by the processor from
   the CS value stored on the stack to determine that a lower-privilege control transfer is occurring.
2. The saved EFLAGS image is popped off of the stack and loaded into the EFLAGS register.
3. The return-program stack pointer is popped off of the stack, loading both the SS register and ESP
   register (RSP[31:0]) with the saved values.
4. Control is transferred to the return program at the target CS:EIP.

8.8 Virtual-8086 Mode Interrupt Control Transfers

This section describes interrupt control transfers as they relate to virtual-8086 mode. Virtual-8086
mode is not supported by long mode. Therefore, the control-transfer mechanism described here is not
applicable to long mode.

When a software interrupt occurs (not external interrupts, INT1, or INT3) while the processor is
running in virtual-8086 mode (EFLAGS.VM=1), the control transfer that occurs depends on three
system controls:
• **EFLAGS.IOPL**—This field controls interrupt handling based on the CPL. See Section “I/O Privilege Level Field (IOPL) Field” on page 53 for more information on this field.

Setting IOPL<3 redirects the interrupt to the general-protection exception (#GP) handler.

• **CR4.VME**—This bit enables virtual-mode extensions. See Section “Virtual-8086 Mode Extensions (VME)” on page 48 for more information on this bit.

• **TSS Interrupt-Redirection Bitmap**—The TSS interrupt-redirection bitmap contains 256 bits, one for each possible INTn vector (software interrupt). When CR4.VME=1, the bitmap is used by the processor to direct interrupts to the handler provided by the currently-running 8086 program (bitmap entry is 0), or to the protected-mode operating-system interrupt handler (bitmap entry is 1). See Section “Legacy Task-State Segment” on page 341 for information on the location of this field within the TSS.

If IOPL<3, CR4.VME=1, and the corresponding interrupt redirection bitmap entry is 0, the processor uses the virtual-interrupt mechanism. See Section “Virtual Interrupts” on page 262 for more information on this mechanism.

Table 8-10 summarizes the actions of the above system controls on interrupts taken when the processor is running in virtual-8086 mode.

### Table 8-10. Virtual-8086 Mode Interrupt Mechanisms

<table>
<thead>
<tr>
<th>EFLAGS.IOPL</th>
<th>CR4.VME</th>
<th>TSS Interrupt Redirection Bitmap Entry</th>
<th>Interrupt Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 1, or 2</td>
<td>0</td>
<td>—</td>
<td>General-Protection Exception</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>Virtual Interrupt</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>Virtual 8086 Handler</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>—</td>
<td>Protected-Mode Handler</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

#### 8.8.1 Protected-Mode Handler Control Transfer

Control transfers to protected-mode handlers from virtual-8086 mode differ from standard protected-mode transfers in several ways. The processor follows these steps in making the control transfer:

1. Reads the CPL=0 stack pointer (SS:ESP) from the TSS.
2. Pushes the GS, FS, DS, and ES selector registers onto the stack. Each push is padded with two bytes to form a doubleword.
3. Clears the GS, FS, DS, and ES selector registers to 0. This places a null selector in each of the four registers
4. Pushes the return stack pointer (old SS:ESP) onto the new stack. The SS value is padded with two bytes to form a doubleword.
5. Pushes the EFLAGS register onto the new stack.

6. Clears the following EFLAGS bits to 0: TF, NT, RF, and VM.

7. Handles EFLAGS.IF based on the gate-descriptor type:
   - If the gate descriptor is an interrupt gate, EFLAGS.IF is cleared to 0.
   - If the gate descriptor is a trap gate, EFLAGS.IF is not modified.

8. Pushes the return-address pointer (CS:EIP) onto the stack. The CS value is padded with two bytes to form a doubleword.

9. If the interrupt has an associated error code, pushes the error code onto the stack.

10. Loads the segment-selector field from the gate descriptor into the CS register, and loads the offset field from the gate descriptor into the EIP register.

11. Begins execution of the interrupt handler with the instruction referenced by the new CS:EIP.

Figure 8-11 shows the new stack after control is transferred to the interrupt handler with an error code.

![Interrupt-Handler Stack]

**Figure 8-11. Stack After Virtual-8086 Mode Interrupt to Protected Mode**

An IRET from privileged protected-mode software (CPL=0) to virtual-8086 mode reverses the stack-build process. After the return pointer, EFLAGS, and return stack-pointer are restored, the processor restores the ES, DS, FS, and GS registers by popping their values off the stack.
8.8.2 Virtual-8086 Handler Control Transfer

When a control transfer to an 8086 handler occurs from virtual-8086 mode, the processor creates an interrupt-handler stack identical to that created when an interrupt occurs in real mode (see Figure 8-6 on page 245). The processor performs the following actions during a control transfer:

1. Pushes the FLAGS register (EFLAGS[15:0]) onto the stack.
2. Clears the EFLAGS.IF and EFLAGS.RF bits to 0.
3. Saves the CS register and IP register (RIP[15:0]) by pushing them onto the stack.
4. Locates the interrupt-handler pointer (CS:IP) in the 8086 IDT by scaling the interrupt vector by four and adding the result to the virtual (linear) address 0. The value in the IDTR is not used.
5. Transfers control to the interrupt handler referenced by the CS:IP in the IDT.

An IRET from the 8086 handler back to virtual-8086 mode reverses the stack-build process.

8.9 Long-Mode Interrupt Control Transfers

The long-mode architecture expands the legacy interrupt-mechanism to support 64-bit operating systems and applications. These changes include:

- All interrupt handlers are 64-bit code and operate in 64-bit mode.
- The size of an interrupt-stack push is fixed at 64 bits (8 bytes).
- The interrupt-stack frame is aligned on a 16-byte boundary.
- The stack pointer, SS:RSP, is pushed unconditionally on interrupts, rather than conditionally based on a change in CPL.
- The SS selector register is loaded with a null selector as a result of an interrupt, if the CPL changes.
- The IRET instruction behavior changes, to unconditionally pop SS:RSP, allowing a null SS to be popped.
- A new interrupt stack-switch mechanism, called the interrupt-stack table or IST, is introduced.

8.9.1 Interrupt Gates and Trap Gates

Only long-mode interrupt and trap gates can be referenced in long mode (64-bit mode and compatibility mode). The legacy 32-bit interrupt-gate and 32-bit trap-gate types (0Eh and 0Fh, as described in Section “System Descriptors” on page 92) are redefined in long mode as 64-bit interrupt-gate and 64-bit trap-gate types. 32-bit and 16-bit interrupt-gate and trap-gate types do not exist in long mode, and software is prohibited from using task gates. If a reference is made to any gate other than a 64-bit interrupt gate or a 64-bit trap gate, a general-protection exception (#GP) occurs.

The long-mode gate types are 16 bytes (128 bits) long. They are an extension of the legacy-mode gate types, allowing a full 64-bit segment offset to be stored in the descriptor. See Section “Legacy Segment Descriptors” on page 82 for a detailed description of the gate-descriptor format and fields.
8.9.2 Locating the Interrupt Handler

When an interrupt occurs in long mode, the processor multiplies the interrupt vector number by 16 and uses the result as an offset into the IDT. The gate descriptor referenced by the IDT offset contains a segment-selector and a 64-bit segment-offset field. The gate-descriptor segment-offset field contains the complete virtual address for the interrupt handler. The gate-descriptor segment-selector field points to the target code-segment descriptor located in either the GDT or LDT. The code-segment descriptor is only used for privilege-checking purposes and for placing the processor in 64-bit mode. The code segment-descriptor base field, limit field, and most attributes are ignored.

Figure 8-12 shows how the long-mode interrupt handler is located by the interrupt mechanism.

Figure 8-12. Long-Mode Interrupt Control Transfer
8.9.3 Interrupt Stack Frame

In long mode, the return-program stack pointer (SS:RSP) is always pushed onto the interrupt-handler stack, regardless of whether or not a privilege change occurs. Although the SS register is not used in 64-bit mode, SS is pushed to allow returns into compatibility mode. Pushing SS:RSP unconditionally presents operating systems with a consistent interrupt-stack-frame size for all interrupts, except for error codes. Interrupt service-routine entry points that handle interrupts generated by non-error-code interrupts can push an error code on the stack for consistency.

In long mode, when a control transfer to an interrupt handler occurs, the processor performs the following:

1. Aligns the new interrupt-stack frame by masking RSP with FFFF_FFFF_FFFF_FFF0h.
2. If IST field in interrupt gate is not 0, reads IST pointer into RSP.
3. If a privilege change occurs, the target DPL is used as an index into the long-mode TSS to select a new stack pointer (RSP).
4. If a privilege change occurs, SS is cleared to zero indicating a null selector.
5. Pushes the return stack pointer (old SS:RSP) onto the new stack. The SS value is padded with six bytes to form a quadword.
6. Pushes the 64-bit RFLAGS register onto the stack. The upper 32 bits of the RFLAGS image on the stack are written as zeros.
7. Clears the TF, NT, and RF bits in RFLAGS bits to 0.
8. Handles the RFLAGS.IF bit according to the gate-descriptor type:
   - If the gate descriptor is an interrupt gate, RFLAGS.IF is cleared to 0.
   - If the gate descriptor is a trap gate, RFLAGS.IF is not modified.
9. Pushes the return CS register and RIP register onto the stack. The CS value is padded with six bytes to form a quadword.
10. If the interrupt vector number has an error code associated with it, pushes the error code onto the stack. The error code is padded with four bytes to form a quadword.
11. Loads the segment-selector field from the gate descriptor into the CS register. The processor checks that the target code-segment is a 64-bit mode code segment.
12. Loads the offset field from the gate descriptor into the target RIP. The interrupt handler begins execution when control is transferred to the instruction referenced by the new RIP.

Figure 8-13 on page 258 shows the stack after control is transferred to the interrupt handler.
Exceptions and Interrupts

Interrupt-Stack Alignment. In legacy mode, the interrupt-stack pointer can be aligned at any address boundary. Long mode, however, aligns the stack on a 16-byte boundary. This alignment is performed by the processor in hardware before pushing items onto the stack frame. The previous RSP is saved unconditionally on the new stack by the interrupt mechanism. A subsequent IRET instruction automatically restores the previous RSP.

Aligning the stack on a 16-byte boundary allows optimal performance for saving and restoring the 16-byte XMM registers. The interrupt handler can save and restore the XMM registers using the faster 16-byte aligned loads and stores (MOVAPS), rather than unaligned loads and stores (MOVUPS). Although the RSP alignment is always performed in long mode, it is only of consequence when the interrupted program is already running at CPL=0, and it is generally used only within the operating-system kernel. The operating system should put 16-byte aligned RSP values in the TSS for interrupts that change privilege levels.

Stack Switch. In long mode, the stack-switch mechanism differs slightly from the legacy stack-switch mechanism (see Section “Interrupt To Higher Privilege” on page 248). When stacks are switched during a long-mode privilege-level change resulting from an interrupt, a new SS descriptor is not loaded from the TSS. Long mode only loads an inner-level RSP from the TSS. However, the SS selector is loaded with a null selector, allowing nested control transfers, including interrupts, to be handled properly in 64-bit mode. The SS.RPL is set to the new CPL value. See Section “Nested IRETs to 64-Bit Mode Procedures” on page 261 for additional information.

Figure 8-13. Long-Mode Stack After Interrupt—Same Privilege
The interrupt-handler stack that results from a privilege change in long mode looks identical to a long-mode stack when no privilege change occurs. Figure 8-14 shows the stack after the switch is performed and control is transferred to the interrupt handler.

![Interrupt-Handler Stack](image)

**Figure 8-14. Long-Mode Stack After Interrupt—Higher Privilege**

### 8.9.4 Interrupt-Stack Table

In long mode, a new interrupt-stack table (IST) mechanism is introduced as an alternative to the modified legacy stack-switch mechanism described above. The IST mechanism provides a method for specific interrupts, such as NMI, double-fault, and machine-check, to always execute on a known-good stack. In legacy mode, interrupts can use the hardware task-switch mechanism to set up a known-good stack by accessing the interrupt service routine through a task gate located in the IDT. However, the hardware task-switch mechanism is not supported in long mode.

When enabled, the IST mechanism unconditionally switches stacks. It can be enabled on an individual interrupt vector basis using a new field in the IDT gate-descriptor entry. This allows some interrupts to use the modified legacy mechanism, and others to use the IST mechanism. The IST mechanism is only available in long mode.

The IST mechanism uses new fields in the 64-bit TSS format and the long-mode interrupt-gate and trap-gate descriptors:

- Figure 12-8 on page 347 shows the format of the 64-bit TSS and the location of the seven IST pointers. The 64-bit TSS offsets from 24h to 5Bh provide space for seven IST pointers, each of which are 64 bits (8 bytes) long.
• The long-mode interrupt-gate and trap-gate descriptors define a 3-bit IST-index field in bits 2:0 of byte +4. Figure 4-24 on page 95 shows the format of long-mode interrupt-gate and trap-gate descriptors and the location of the IST-index field.

To enable the IST mechanism for a specific interrupt, system software stores a non-zero value in the interrupt gate-descriptor IST-index field. If the IST index is zero, the modified legacy stack-switching mechanism (described in the previous section) is used.

Figure 8-15 shows how the IST mechanism is used to create the interrupt-handler stack. When an interrupt occurs and the IST index is non-zero, the processor uses the index to select the corresponding IST pointer from the TSS. The IST pointer is loaded into the RSP to establish a new stack for the interrupt handler. The SS register is loaded with a null selector if the CPL changes and the SS.RPL is set to the new CPL value. After the stack is loaded, the processor pushes the old stack pointer, RFLAGS, the return pointer, and the error code (if applicable) onto the stack. Control is then transferred to the interrupt handler.

Figure 8-15. Long-Mode IST Mechanism

Software must make sure that an interrupt or exception handler using an IST pointer doesn't take another exception using the same IST pointer, as this will result in the first stack exception frame being overwritten.
8.9.5 Returning From Interrupt Procedures

As with legacy mode, a return to an interrupted program in long mode should be performed using the IRET instruction. However, in long mode, the IRET semantics are different from legacy mode:

- **In 64-bit mode**, IRET pops the return-stack pointer unconditionally off the interrupt-stack frame and into the SS:RSP registers. This reverses the action of the long-mode interrupt mechanism, which saves the stack pointer whether or not a privilege change occurs. IRET also allows a null selector to be popped off the stack and into the SS register. See Section “Nested IRETs to 64-Bit Mode Procedures” on page 261 for additional information.

- **In compatibility mode**, IRET behaves as it does in legacy mode. The SS:ESP is popped off the stack only if a control transfer to less privilege (numerically greater CPL) is performed. Otherwise, it is assumed that a stack pointer is not present on the interrupt-handler stack.

The long-mode interrupt mechanism always uses a 64-bit stack when saving values for the interrupt handler, and the interrupt handler is always entered in 64-bit mode. To work properly, an IRET used to exit the 64-bit mode interrupt-handler requires a series of eight-byte pops off the stack. This is accomplished by using a 64-bit operand-size prefix with the IRET instruction. The default stack size assumed by an IRET in 64-bit mode is 32 bits, so a 64-bit REX prefix is needed by 64-bit mode interrupt handlers.

**Nested IRETs to 64-Bit Mode Procedures.** In long mode, an interrupt causes a null selector to be loaded into the SS register if the CPL changes (this is the same action taken by a far CALL in long mode). If the interrupt handler performs a far call, or is itself interrupted, the null SS selector is pushed onto the stack frame, and another null selector is loaded into the SS register. Using a null selector in this way allows the processor to properly handle returns nested within 64-bit-mode procedures and interrupt handlers.

The null selector enables the processor to properly handle nested returns to 64-bit mode (which do not use the SS register), and returns to compatibility mode (which do use the SS register). Normally, an IRET that pops a null selector into the SS register causes a general-protection exception (#GP) to occur. However, in long mode, the null selector indicates the existence of nested interrupt handlers and/or privileged software in 64-bit mode. Long mode allows an IRET to pop a null selector into SS from the stack under the following conditions:

- The target mode is 64-bit mode.
- The target CPL<3.

In this case, the processor does not load an SS descriptor, and the null selector is loaded into SS without causing a #GP exception.
8.10 Virtual Interrupts

The term virtual interrupts includes two classes of extensions to the interrupt-handling mechanism:

- **Virtual-8086 Mode Extensions (VME)**—These allow virtual interrupts and interrupt redirection in virtual-8086 mode. VME has no effect on protected-mode programs.
- **Protected-Mode Virtual Interrupts (PVI)**—These allow virtual interrupts in protected mode when CPL=3. Interrupt redirection is not available in protected mode. PVI has no effect on virtual-8086-mode programs.

Because virtual-8086 mode is not supported in long mode, VME extensions are not supported in long mode. PVI extensions are, however, supported in long mode.

8.10.1 Virtual-8086 Mode Extensions

The virtual-8086-mode extensions (VME) enable performance enhancements for 8086 programs running as protected tasks in virtual-8086 mode. These extensions are enabled by setting CR4.VME (bit 0) to 1. The extensions enabled by CR4.VME are:

- Virtualizing control and notification of maskable external interrupts with the EFLAGS VIF (bit 19) and VIP (bit 20) bits.
- Selective interception of software interrupts (INT\_n instructions) using the TSS interrupt redirection bitmap (IRB).

**Background.** Legacy-8086 programs expect to have full access to the EFLAGS interrupt flag (IF) bit, allowing programs to enable and disable maskable external interrupts. When those programs run in virtual-8086 mode under a multitasking protected-mode environment, it can disrupt the operating system if programs enable or disable interrupts for their own purposes. This is particularly true if interrupts associated with one program can occur during execution of another program. For example, a program could request that an area of memory be copied to disk. System software could suspend the program before external hardware uses an interrupt to acknowledge that the block has been copied. System software could subsequently start a second program which enables interrupts. This second program could receive the external interrupt indicating that the memory block of the first program has been copied. If that were to happen, the second program would probably be unprepared to handle the interrupt properly.

Access to the IF bit must be managed by system software on a task-by-task basis to prevent corruption of system resources. In order to completely manage the IF bit, system software must be able to interrupt all instructions that can read or write the bit. These instructions include STI, CLI, PUSHF, POPF, INT\_n, and IRET. These instructions are part of an instruction class that is IOPL-sensitive. The processor takes a general-protection exception (#GP) whenever an IOPL-sensitive instruction is executed and the EFLAGS.IOPL field is less than the CPL. Because all virtual-8086 programs run at CPL=3, system software can interrupt all instructions that modify the IF bit by setting IOPL<3.

System software maintains a virtual image of the IF bit for each virtual-8086 program by emulating the actions of IOPL-sensitive instructions that modify the IF bit. When an external maskable-interrupt
occurs, system software checks the state of the IF image for the current virtual-8086 program to determine whether the program is masking interrupts. If the program is masking interrupts, system software saves the interrupt information until the virtual-8086 program attempts to re-enable interrupts. When the virtual-8086 program unmaskes interrupts with an IOPL-sensitive instruction, system software traps the action with the #GP handler.

The performance of a processor can be significantly degraded by the overhead of trapping and emulating IOPL-sensitive instructions, and the overhead of maintaining images of the IF bit for each virtual-8086 program. This performance loss can be eliminated by running virtual-8086 programs with IOPL set to 3, thus allowing changes to the real IF flag from any privilege level. Unfortunately, this can leave critical system resources unprotected.

In addition to the performance problems caused by virtualizing the IF bit, software interrupts (INTn instructions) cannot be masked by the IF bit or virtual copies of the IF bit. The IF bit only affects maskable external interrupts. Software interrupts in virtual-8086 mode are normally directed to the real mode interrupt vector table (IVT), but it can be desirable to redirect certain interrupts to the protected-mode interrupt-descriptor table (IDT).

The virtual-8086-mode extensions are designed to support both external interrupts and software interrupts, with mechanisms that preserve high performance without compromising protection. Virtualization of external interrupts is supported using two bits in the EFLAGS register: the virtual-interrupt flag (VIF) bit and the virtual-interrupt pending (VIP) bit. Redirection of software interrupts is supported using the interrupt-redirection bitmap (IRB) in the TSS. A separate TSS can be created for each virtual-8086 program, allowing system software to control interrupt redirection independently for each virtual-8086 program.

**VIF and VIP Extensions for External Interrupts.** When VME extensions are enabled, the IF-modifying instructions normally trapped by system software are allowed to execute. However, instead of modifying the IF bit, they modify the EFLAGS VIF bit. This leaves control over maskable interrupts to the system software. It can also be used as an indicator to system software that the virtual-8086 program is able to, or is expecting to, receive external interrupts.

When an unmasked external interrupt occurs, the processor transfers control from the virtual-8086 program to a protected-mode interrupt handler. If the interrupt handler determines that the interrupt is for the virtual-8086 program, it can check the state of the VIF bit in the EFLAGS value pushed on the stack for the virtual-8086 program. If the VIF bit is set (indicating the virtual-8086 program attempted to unmask interrupts), system software can allow the interrupt to be handled by the appropriate virtual-8086 interrupt handler.

If the VIF bit is clear (indicating the virtual-8086 program attempted to mask interrupts) and the interrupt is for the virtual-8086 program, system software can hold the interrupt pending. System software holds an interrupt pending by saving appropriate information about the interrupt, such as the interrupt vector, and setting the virtual-8086 program's VIP bit in the EFLAGS image on the stack. When the virtual-8086 program later attempts to set IF, the previously set VIP bit causes a general-protection exception (#GP) to occur. System software can then pass the saved interrupt information to the virtual-8086 interrupt handler.
To summarize, when the VME extensions are enabled (CR4.VME=1), the VIF and VIP bits are set and cleared as follows:

- **VIF Bit**—This bit is set and cleared by the processor in virtual-8086 mode in response to an attempt by a virtual-8086 program to set and clear the EFLAGS.IF bit. VIF is used by system software to determine whether a maskable external interrupt should be passed on to the virtual-8086 program, emulated by system software, or held pending. VIF is also cleared during software interrupts through interrupt gates, with the original VIF value preserved in the EFLAGS image on the stack.

- **VIP Bit**—System software sets and clears this bit in the EFLAGS image saved on the stack after an interrupt. It can be set when an interrupt occurs for a virtual-8086 program that has a clear VIF bit. The processor examines the VIP bit when an attempt is made by the virtual-8086 program to set the IF bit. If VIP is set when the program attempts to set IF, a general-protection exception (#GP) occurs before execution of the IF-setting instruction. System software must clear VIP to avoid repeated #GP exceptions when returning to the interrupted instruction.

The VIF and VIP bits can be used by system software to minimize the overhead associated with managing maskable external interrupts because virtual copies of the IF flag do not have to be maintained by system software. Instead, VIF and VIP are maintained during context switches along with the remaining EFLAGS bits.

Table 8-11 on page 266 shows how the behavior of instructions that modify the IF bit are affected by the VME extensions.

**Interrupt Redirection of Software Interrupts.** In virtual-8086 mode, software interrupts (INT\(n\) instructions) are trapped using a #GP exception handler if the IOPL is less than 3 (the CPL for virtual-8086 mode). This allows system software to interrupt and emulate 8086-interrupt handlers. System software can set the IOPL to 3, in which case the INT\(n\) instruction is vectored through a gate descriptor in the protected-mode IDT. System software can use the gate to control access to the virtual-8086 mode interrupt vector table (IVT), or to redirect the interrupt to a protected-mode interrupt handler.

When VME extensions are enabled, for INT\(n\) instructions to execute normally, vectoring directly to a virtual-8086 interrupt handler through the virtual-8086 IVT (located at address 0 in the virtual-address space of the task). For security or performance reasons, however, it can be necessary to intercept INT\(n\) instructions on a vector-specific basis to allow servicing by protected-mode interrupt handlers. This is performed by using the interrupt-redirection bitmap (IRB), located in the TSS and enabled when CR4.VME=1. The IRB is available only in virtual-8086 mode.

Figure 12-6 on page 342 shows the format of the TSS, with the interrupt redirection bitmap located near the top. The IRB contains 256 bits, one for each possible software-interrupt vector. The most-significant bit of the IRB controls interrupt vector 255, and is located immediately before the IOPB base. The least-significant bit of the IRB controls interrupt vector 0.

The bits in the IRB function as follows:
• When set to 1, the INTn instruction behaves as if the VME extensions are not enabled. The interrupt is directed through the IDT to a protected-mode interrupt handler if IOPL=3. If IOPL<3, the INTn causes a #GP exception.
• When cleared to 0, the INTn instruction is directed through the IVT for the virtual-8086 program to the corresponding virtual-8086 interrupt handler.

Only software interrupts can be redirected using the IRB mechanism. External interrupts are asynchronous events that occur outside the context of a virtual-8086 program. Therefore, external interrupts require system-software intervention to determine the appropriate context for the interrupt. The VME extensions described in Section “VIF and VIP Extensions for External Interrupts” on page 263 are provided to assist system software with external-interrupt intervention.

8.10.2 Protected Mode Virtual Interrupts

The protected-mode virtual-interrupt (PVI) bit in CR4 enables support for interrupt virtualization in protected mode. When enabled, the processor maintains program-specific VIF and VIP bits similar to the manner defined by the virtual-8086 mode extensions (VME). However, unlike VME, only the STI and CLI instructions are affected by the PVI extension. When a program is running at CPL=3, it can use STI and CLI to set and clear its copy of the VIF flag without causing a general-protection exception. The last section of Table 8-11 on page 266 describes the behavior of instructions that modify the IF bit when PVI extensions are enabled.

The interrupt redirection bitmap (IRB) defined by the VME extensions is not supported by the PVI extensions.

8.10.3 Effect of Instructions that Modify EFLAGS.IF

Table 8-11 on page 266 shows how the behavior of instructions that modify the IF bit are affected by the VME and PVI extensions. The table columns specify the following:

- Operating Mode—the processor mode in effect when the instruction is executed.
- Instruction—the IF-modifying instruction.
- IOPL—the value of the EFLAGS.IOPL field.
- VIP—the value of the EFLAGS.VIP bit.
- #GP—indicates whether the conditions in the first four columns cause a general-protection exception (#GP) to occur.
- Effect on IF Bit—indicates the effect the conditions in the first four columns have on the EFLAGS.IF bit and the image of EFLAGS.IF on the stack.
- Effect on VIF Bit—indicates the effect the conditions in the first four columns have on the EFLAGS.VIF bit and the image of EFLAGS.VIF on the stack.
### Table 8-11. Effect of Instructions that Modify the IF Bit

<table>
<thead>
<tr>
<th>Operating Mode</th>
<th>Instruction</th>
<th>IOPL</th>
<th>VIP</th>
<th>#GP</th>
<th>Effect on IF Bit</th>
<th>Effect on VIF Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Real Mode</strong></td>
<td>CLI</td>
<td></td>
<td></td>
<td>no</td>
<td>IF = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STI</td>
<td></td>
<td></td>
<td></td>
<td>IF = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PUSHF</td>
<td></td>
<td></td>
<td></td>
<td>EFLAGS.IF Stack Image = IF</td>
<td>IF = EFLAGS.IF stack image</td>
</tr>
<tr>
<td></td>
<td>POPF</td>
<td></td>
<td></td>
<td></td>
<td>EFLAGS.IF Stack Image = IF</td>
<td>IF = 0</td>
</tr>
<tr>
<td></td>
<td>intn</td>
<td></td>
<td></td>
<td></td>
<td>IF = EFLAGS.IF Stack Image</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IRET</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Protected Mode</strong></td>
<td>CLI</td>
<td>≥CPL</td>
<td>&lt;CPL</td>
<td>no</td>
<td>IF = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STI</td>
<td>≥CPL</td>
<td>&lt;CPL</td>
<td></td>
<td>IF = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PUSHF</td>
<td>x</td>
<td></td>
<td>yes</td>
<td>EFLAGS.IF Stack Image = IF</td>
<td>IF = EFLAGS.IF Stack Image</td>
</tr>
<tr>
<td></td>
<td>POPF</td>
<td>≥CPL</td>
<td>&lt;CPL</td>
<td></td>
<td>No Change</td>
<td></td>
</tr>
<tr>
<td></td>
<td>intn gate</td>
<td>x</td>
<td></td>
<td>no</td>
<td>EFLAGS.IF Stack Image = IF</td>
<td>IF = 0</td>
</tr>
<tr>
<td></td>
<td>IRET</td>
<td></td>
<td></td>
<td></td>
<td>IF = EFLAGS.IF Stack Image</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IRETD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Virtual-8086 Mode</strong></td>
<td>CLI</td>
<td>3</td>
<td></td>
<td>no</td>
<td>IF = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STI</td>
<td>3</td>
<td></td>
<td>yes</td>
<td>IF = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PUSHF</td>
<td>3</td>
<td></td>
<td>yes</td>
<td>EFLAGS.IF Stack Image = IF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>POPF</td>
<td>3</td>
<td></td>
<td>yes</td>
<td>EFLAGS.IF Stack Image = IF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>intn gate</td>
<td>3</td>
<td></td>
<td>yes</td>
<td>EFLAGS.IF Stack Image = IF</td>
<td>IF = 0</td>
</tr>
<tr>
<td></td>
<td>IRET</td>
<td>3</td>
<td></td>
<td>yes</td>
<td>IF = EFLAGS.IF Stack Image</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IRETD</td>
<td>3</td>
<td></td>
<td>yes</td>
<td>IF = EFLAGS.IF Stack Image</td>
<td></td>
</tr>
</tbody>
</table>

**Note:**

Gray-shaded boxes indicate the bits are unsupported (ignored) in the specified operating mode.

“x” indicates the value of the bit is a “don’t care”.

“—” indicates the instruction causes a general-protection exception (#GP).

**Note:**

1. If the EFLAGS.IF stack image is 0, no #GP exception occurs and the IRET instruction is executed.
2. If the EFLAGS.IF stack image is 1, the IRET is not executed, and a #GP exception occurs.
### Table 8-11. Effect of Instructions that Modify the IF Bit (continued)

<table>
<thead>
<tr>
<th>Operating Mode with VME Extensions</th>
<th>Instruction</th>
<th>IOPL</th>
<th>VIP</th>
<th>#GP</th>
<th>Effect on IF Bit</th>
<th>Effect on VIF Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLI</td>
<td>3</td>
<td>x</td>
<td>no</td>
<td>IF = 0</td>
<td>No Change</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;3</td>
<td></td>
<td></td>
<td></td>
<td>No Change</td>
<td>VIF = 0</td>
</tr>
<tr>
<td>STI</td>
<td>3</td>
<td>x</td>
<td>no</td>
<td>IF = 1</td>
<td>No Change</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;3</td>
<td>0</td>
<td>no</td>
<td>No Change</td>
<td>VIF = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>yes</td>
<td>—</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSHF</td>
<td>3</td>
<td>x</td>
<td>no</td>
<td>EFLAGS.IF Stack Image = IF</td>
<td>Not Pushed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;3</td>
<td></td>
<td></td>
<td></td>
<td>Not Pushed</td>
<td>EFLAGS.IF Stack Image = VIF</td>
</tr>
<tr>
<td>PUSHFD</td>
<td>3</td>
<td>x</td>
<td>no</td>
<td>EFLAGS.IF Stack Image = IF</td>
<td>EFLAGS.VIF Stack Image = VIF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;3</td>
<td></td>
<td>yes</td>
<td>—</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POPF</td>
<td>3</td>
<td>x</td>
<td>no</td>
<td>IF = EFLAGS.IF Stack Image</td>
<td>No Change</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;3</td>
<td>0</td>
<td>no</td>
<td>No Change</td>
<td>VIF = EFLAGS.IF Stack Image</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>yes</td>
<td>—</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POPFD</td>
<td>3</td>
<td>x</td>
<td>no</td>
<td>IF = EFLAGS.IF Stack Image</td>
<td>No Change</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;3</td>
<td></td>
<td>yes</td>
<td>—</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT ( ) gate</td>
<td>3</td>
<td>x</td>
<td>no</td>
<td>EFLAGS.IF Stack Image = IF</td>
<td>No Change</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IF = 0</td>
<td></td>
<td>VIF = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No Change</td>
<td>EFLAGS.IF Stack Image = VIF</td>
<td></td>
</tr>
<tr>
<td>IRET</td>
<td>3</td>
<td>x</td>
<td>no</td>
<td>IF = EFLAGS.IF Stack Image</td>
<td>No Change</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;3</td>
<td>0</td>
<td>no</td>
<td>No Change</td>
<td>VIF = EFLAGS.IF Stack Image</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>no^1</td>
<td>No Change</td>
<td>VIF = EFLAGS.IF Stack Image</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>yes^2</td>
<td>—</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRETD</td>
<td>3</td>
<td>x</td>
<td>no</td>
<td>IF = EFLAGS.IF Stack Image</td>
<td>VIF = EFLAGS.IF Stack Image</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;3</td>
<td></td>
<td>yes</td>
<td>—</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:**

Gray-shaded boxes indicate the bits are unsupported (ignored) in the specified operating mode.

"x" indicates the value of the bit is a “don’t care”.

“—” indicates the instruction causes a general-protection exception (#GP).

**Note:**

1. If the EFLAGS.IF stack image is 0, no #GP exception occurs and the IRET instruction is executed.
2. If the EFLAGS.IF stack image is 1, the IRET is not executed, and a #GP exception occurs.
Table 8-11. Effect of Instructions that Modify the IF Bit (continued)

<table>
<thead>
<tr>
<th>Operating Mode</th>
<th>Instruction</th>
<th>IOPL</th>
<th>VIP</th>
<th>#GP</th>
<th>Effect on IF Bit</th>
<th>Effect on VIF Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protected Mode</td>
<td>CLI</td>
<td>3</td>
<td>x</td>
<td>no</td>
<td>IF = 0</td>
<td>No Change</td>
</tr>
<tr>
<td>with PVI</td>
<td></td>
<td>&lt;3</td>
<td></td>
<td></td>
<td>No Change</td>
<td>VIF = 0</td>
</tr>
<tr>
<td>Extensions</td>
<td>STI</td>
<td>3</td>
<td>x</td>
<td>no</td>
<td>IF = 1</td>
<td>No Change</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;3</td>
<td>0</td>
<td>no</td>
<td>No Change</td>
<td>VIF = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>yes</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>PUSHF</td>
<td>3</td>
<td></td>
<td>x</td>
<td>EFLAGS.IF Stack Image = IF</td>
<td>Not Pushed</td>
</tr>
<tr>
<td></td>
<td>PUSHFD</td>
<td></td>
<td></td>
<td></td>
<td>EFLAGS.VIF Stack Image = VIF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>POPF</td>
<td>x</td>
<td></td>
<td>no</td>
<td>EFLAGS.IF Stack Image = IF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>POPFD</td>
<td></td>
<td></td>
<td></td>
<td>VIF = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTn gate</td>
<td></td>
<td></td>
<td></td>
<td>EFLAGS.IF Stack Image = IF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IRET</td>
<td></td>
<td></td>
<td></td>
<td>No Change</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IRETD</td>
<td></td>
<td></td>
<td></td>
<td>VIF = EFLAGS.VIF Stack Image</td>
<td></td>
</tr>
</tbody>
</table>

Note:
Gray-shaded boxes indicate the bits are unsupported (ignored) in the specified operating mode.
“x” indicates the value of the bit is a “don’t care”.
“—” indicates the instruction causes a general-protection exception (#GP).

Note:
1. If the EFLAGS.IF stack image is 0, no #GP exception occurs and the IRET instruction is executed.
2. If the EFLAGS.IF stack image is 1, the IRET is not executed, and a #GP exception occurs.
9 Machine Check Architecture

The AMD64 Machine Check Architecture (MCA) plays a vital role in the reliability, availability, and serviceability (RAS) of AMD processors, as well as the RAS of the computer systems in which they are embedded. MCA defines the facilities by which processor and system hardware errors are logged and reported to system software. This allows system software to serve a strategic role in recovery from and diagnosis of hardware errors.

Error checking hardware is configured and information about detected error conditions is conveyed via an architecturally-defined set of registers. The system programming interface of MCA is described below in Section 9.3 “Machine Check Architecture MSRs” on page 273.

9.1 Introduction

All computer systems are susceptible to errors—results that are contrary to the system design. Errors can be categorized as soft or hard. Soft errors are caused by transient interference and are not necessarily indicative of any damage to the computer circuitry. These external events include noise from electromagnetic radiation and the incursion of sub-atomic particles that cause bit cell storage capacitors to change state.

Hard errors are repeatable malfunctions that are generally attributable to physical damage to computer circuitry. Damage may be caused by external forces (for example, voltage surges) or wear processes inherent in the circuit technology. Damaged circuit elements can manifest symptoms similar to those that are caused by soft error processes. An increase in the frequency of errors attributable to one circuit element may indicate that the element has sustained damage or is wearing-out and may, in the future, cause a hard error.

9.1.1 Reliability, Availability, and Serviceability

This section describes the concepts of reliability, availability, and serviceability (RAS) and shows how they are interrelated.

The rate at which errors occur in a computer system is a measure of the system’s reliability. Availability is the percentage of time that the system is available to do useful work. Errors that prevent a computer system from continued operation result in down-time, that is, periods of unavailability. Down-time includes the amount of time required to restore the system to operation. This may include the time to diagnose a failure, determine the field replaceable unit (FRU) containing the faulty circuitry, carry out the repair action required to replace the identified FRU, and restart the system. This time directly impacts the system’s availability and is a measure of the system’s serviceability.

The availability of a computer system can be increased without decreasing performance or significantly increasing cost through the judicious addition of data and control path redundancy in concert with dedicated error-checking hardware. Together, redundancy and error checking detect and
often correct hardware errors. When errors are corrected by hardware, system operation continues without any perceptible disruption or loss in performance.

Another important technique that can prevent downtime is error containment. Error containment limits the propagation of an erroneous data. This enhances system availability by limiting the effects of errors to a subset of software or hardware resources. System software may either correct the error and resume the interrupted program or, if the error cannot be corrected, terminate software processes that cannot continue due to the error.

Error logging enhances serviceability by providing information that is used to identify the FRU that contains the failed circuitry. The mechanical design of the computer system can enhance serviceability (and thus availability) by making the task of physically replacing a failed FRU quicker and easier.

9.1.2 Error Detection, Logging, and Reporting

Error detection requires specific error-checking hardware that compares the actual result of some data transfer or transformation to the expected result. Any disparity indicates that an error has occurred. Error detection is controlled through implementation-specific means. Disabling detection is normally only appropriate when hardware is being debugged in the laboratory.

When an error is detected, hardware autonomously acts to either correct the error or contain the propagation of the corrupting effects of an uncorrected error. For some error sources, hardware action can be disabled by software through the MCA interface.

As hardware acts to correct or contain a detected error, it gathers information about the error to aid in recovery, diagnosis, and repair. The architecture provides software control of error logging and reporting. The following describes the characteristics of each:

• Logging
  Logging involves saving information about the error in specific MCA registers. If the error reporting bank associated with the error source is enabled, logging occurs; if disabled, error information is generally discarded (there are implementation-specific exceptions).

• Reporting
  An uncorrected error may be reported to system software via a machine-check exception, if error reporting for the specific error source is enabled.

Reporting is the hardware-initiated action of interrupting the processor using a machine-check exception (#MC). Reporting for each specific error type can be enabled or disabled by system software though the MCA register interface. Even if reporting for an error type is disabled, logging may continue.

Disabling reporting can negatively impact both error containment and error recovery (see the next section) and should be avoided.

Hardware categorizes errors into three classes. These are:

• corrected
The following sections describe the characteristics of each of these error classes:

If an error can be corrected by hardware, no immediate action by software is required. In this case, information is logged, if enabled, to aid in later diagnosis and possible repair.

If correction is not possible, the error is classified as uncorrected. The occurrence of an uncorrected error requires immediate action by system software to either correct the error and resume the interrupted program or, if software-based correction is not possible, to determine the extent of the impact of the uncorrected error to any executing instruction stream or the architectural state of the processor or system and take actions to contain the error condition by terminating corrupted software processes.

For errors that are not corrected, but have no immediate impact on the architectural state of the system, processor core, or any current thread of execution, the error may be classified by hardware as a deferred error. Information about deferred errors is logged, if enabled, but not reported via a machine-check exception. Instead hardware monitors the error and escalates the error classification to uncorrected at the point in time where the error condition is about to impact the execution of an instruction stream or cause the corruption of the processor core or system architectural state.

This escalation results in a #MC exception, assuming that reporting for that error source is enabled. If software can correct the error, it may be possible to resume the affected program. If not, software can terminate the affected program rather than bringing down the entire system. This is referred to as error localization.

A common example of deferred error processing and localization is the conversion of globally uncorrected DRAM errors to process-specific consumed memory errors. In this example, uncorrected ECC-protected data that has not yet been consumed by any processor core is tagged as “poison.” Hardware reports the uncorrected data as a localized error via a #MC exception when it is about to be used (“consumed”) by an instruction execution stream.

In contrast, an error that cannot be contained and is of such severity that it has compromised the continued operation of a processor core requires immediate action to terminate system processing and may result in a hardware-enforced shutdown. In the shutdown state, the execution of instructions by that processor core is halted. See Section 8.2.9 “#DF—Double-Fault Exception (Vector 8)” on page 228 for a description of the shutdown processor state.

If supported, system software can chose to configure and enable hardware to generate an interrupt when a deferred error is first detected. Corrected errors may be counted as they are logged. If supported and enabled, exceeding a software-configured count threshold may be signalled via an interrupt. These notification mechanisms are independent of machine-check reporting.

Specific details on hardware error detection, logging, and reporting are implementation-dependent and are described in the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual applicable to your product.
### 9.1.3 Error Recovery

When errors cannot be corrected by hardware, *error recovery* comes into play. Error recovery, as defined by MCA, always involves software intervention. Logged information about the uncorrected error condition that caused the exception allows system software to take actions to either correct the error and resume the interrupted execution stream or terminate software processes (or higher-level software constructs) that are known to be affected by the uncorrected error.

From a system perspective, all errors are either recoverable or unrecoverable. The following outlines the characteristics of each:

- **Recoverable**—Hardware has determined that the architectural state of the processor experiencing the uncorrected error has not been compromised. Software execution can continue if system software can determine the extent of the error and take actions to either:
  - correct the error and *resume* the interrupted stream of execution or,
  - if this is not possible, *terminate* software processes that have incurred a loss of architectural state and *continue* other software processes that are unaffected by the error.

- **Unrecoverable**—Hardware has determined that the architectural state of the processor experiencing the uncorrected error has been corrupted. Software execution cannot reliably continue.
  
  Software saves any diagnostic information that it may be able to gather and halts.

The fact that an error is recoverable does not mean that recovery software will be able to resume program execution. If it is unable to determine the extent of the corruption or if it determines that essential state information has been lost, it may only be able to save information about the error and halt processing.

System software has many options to recover from an uncorrected error. The following is a partial list of possible actions that system software might take:

- If it can be determined that the corruption caused by the uncorrected error is contained within a software process, software can kill the process.
- If the uncorrected error has corrupted the architectural state of a virtual machine, the VMM can rebuild the container (using only hardware resources that are known to be good) and reboot the guest operating system.
- If the uncorrected error is a part of a block of data being transferred to or from an I/O device, the data transfer can be flushed and retried or terminated with an error.
- If the uncorrected error is due to a hard link failure, software can reconfigure the network to route information around the failed link.
- If the uncorrected error is in a cache and the cache line containing the uncorrected (known bad) data is in the shared state, software can invalidate the line so that it will be reloaded from memory or another cache that has the line in the owned state.
Many more error scenarios are recoverable depending on the effectiveness of hardware error containment, the logging capabilities of the system, and the sophistication of the recovery software that acts on the information conveyed through the MCA reporting structure.

If recovery software is unable to restore a valid system architectural state at some level of software abstraction (process, guest operating system, virtual machine, or virtual machine monitor), the uncorrected error is considered system fatal. In this situation, system software must halt the execution of instructions. A system reset is required to restore the system to a known-good architectural state.

9.2 Determining Machine-Check Architecture Support

Support for the machine-check architecture is implementation-dependent. System software executes the CPUID instruction to determine whether a processor implements the machine-check exception (#MC) and the global MCA MSRs. The CPUID Fn0000_0001_EDX[MCE] feature bit indicates support for the machine-check exception and the CPUID Fn0000_0001_EDX[MCA] feature bit indicates support for the base set of global machine-check MSRs.

Once system software determines that the base set of MCA MSRs is available, it determines the implemented number of machine-check reporting banks by reading the machine-check capabilities register (MCG_CAP), which is the first of the global MCA MSRs.

For a processor implementation to provide an architecturally compliant MCA interface, it must provide support for the machine-check exception, the global machine-check MSRs, the watchdog timer (see “CPU Watchdog Timer Register” on page 276.), and at least one bank of the machine-check reporting registers.

Support for the deferred reporting and software-based containment of uncorrected data errors is indicated by the feature bit CPUID Fn8000_0007_EBX[SUCCOR]. See “Machine-Check Recovery” on page 279.

Support for recoverable MCA overflow conditions is indicated by feature bit CPUID Fn8000_0007_EBX[McaOverflowRecov]. See the discussion of recoverable status overflow in Section “MCA Overflow” on page 278.

Implementation-specific information concerning the machine-check mechanism can be found in the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual applicable to your product. For more information on using the CPUID instruction, see Section 3.3, “Processor Feature Identification,” on page 64.

9.3 Machine Check Architecture MSRs

The AMD64 Machine-Check Architecture defines the set of model-specific registers (MCA MSRs) used to log and report hardware errors. These registers are:

- Global status and control registers:
  - Machine-check global-capabilities register (MCG_CAP)
- Machine-check global-status register (MCG_STATUS)
- Machine-check global-control register (MCG_CTL)

- One or more error-reporting register banks, each containing:
  - Machine-check control register (MCi_CTL)
  - Machine-check status register (MCi_STATUS)
  - Machine-check address register (MCi_ADDR)
  - At least one machine-check miscellaneous error-information register (MCi_MISC0)

Each error-reporting register bank is associated with a specific processor unit (or group of processor units).

- CPU Watchdog Timer register (CPU_WATCHDOG_TIMER)

The error-reporting registers retain their values through a warm reset. (A warm reset occurs while power to the processor is stable. This in contrast to a cold reset, which occurs during the application of power after a period of power loss.) This preservation of error information allows the platform firmware or other system-boot software to recover and report information associated with the error when the processor is forced into a shutdown state.

The RDMSR and WRMSR instructions are used to read and write the machine-check MSRs. See “Machine-Check MSRs” on page 611 for a listing of the machine-check MSR numbers and their reset values. The following sections describe each MCA MSR and its function.

### 9.3.1 Global Status and Control Registers

The global status and control MSRs are the MCG_CAP, MCG_STATUS, and MCG_CTL registers.

#### Machine-Check Global-Capabilities Register

Figure 9-1 shows the format of the machine-check global-capabilities register (MCG_CAP). MCG_CAP is a read-only register that specifies the machine-check mechanism capabilities supported by the processor implementation.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:9</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>CTLP</td>
<td>MCG_CTL register present</td>
<td>R</td>
</tr>
<tr>
<td>7:0</td>
<td>BANK_CNT</td>
<td>Number of reporting banks</td>
<td>R</td>
</tr>
</tbody>
</table>

**Figure 9-1. MCG_CAP Register**
The fields within the MCG_CAP register are:

- **BANK_CNT (MCi Bank Count)**—Bits 7:0. This field specifies how many error-reporting register banks are supported by the processor implementation.

- **CTLP (MCG_CTL Register Present)**—Bit 8. This bit specifies whether or not the Machine-Check Global-Control (MCG_CTL) Register is supported by the processor. When the bit is set to 1, the register is supported. When the bit is cleared to 0, the register is unsupported. The MCG_CTL register is described on page 276.

All remaining bits in the MCG_CAP register are reserved. Writing values to the MCG_CAP register produces undefined results.

**Machine-Check Global-Status Register.** Figure 9-2 shows the format of the machine-check global-status register (MCG_STATUS). MCG_STATUS provides basic information about the processor state after the occurrence of a machine-check error.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:3</td>
<td>Reserved</td>
<td></td>
<td>R/W</td>
</tr>
<tr>
<td>2</td>
<td>MCIP</td>
<td>Machine Check In-Progress</td>
<td>R/W</td>
</tr>
<tr>
<td>1</td>
<td>EIPV</td>
<td>Error IP Valid Flag</td>
<td>R/W</td>
</tr>
<tr>
<td>0</td>
<td>RIPV</td>
<td>Restart IP Valid Flag</td>
<td>R/W</td>
</tr>
</tbody>
</table>

**Figure 9-2. MCG_STATUS Register**

The fields within the MCG_STATUS register are:

- **Restart-IP Valid (RIPV)**—Bit 0. When this bit is set to 1, the interrupted program can be reliably restarted at the instruction addressed by the instruction pointer pushed onto the stack by the machine-check error mechanism. If this bit is cleared to 0, the interrupted program cannot be reliably restarted.

- **Error-IP Valid (EIPV)**—Bit 1. When this bit is set to 1, the instruction that is referenced by the instruction pointer pushed onto the stack by the machine-check error mechanism is responsible for the machine-check error. If this bit is cleared to 0, it is possible that the instruction referenced by the instruction pointer is not responsible for the machine-check error.

- **Machine Check In-Progress (MCIP)**—Bit 2. When this bit is set to 1, it indicates that a machine-check error is in progress. If another machine-check error occurs while this bit is set, the processor
enters the shutdown state. The processor sets this bit whenever a machine check exception is generated. Software is responsible for clearing it after the machine check exception is handled.

All remaining bits in the MCG_STATUS register are reserved.

**Machine-Check Global-Control Register.** Figure 9-3 shows the format of the machine-check global-control register (MCG_CTL). MCG_CTL is used by software to enable or disable the logging and reporting of machine-check errors from the implemented error-reporting banks. Depending on the implementation, detected errors from some error sources associated with a reporting bank that is disabled are still logged. Setting all bits to 1 in this register enables all implemented error-reporting register banks to log errors.

![Figure 9-3. MCG_CTL Register](image)

**CPU Watchdog Timer Register.** The CPU watchdog timer is used to generate a machine check condition when an instruction does not complete within a time period specified by the CPU Watchdog Timer register. The timer restarts the count each time an instruction completes, when enabled by the CPU Watchdog Timer Enable bit. The time period is determined by the Count Select and Time Base fields. The timer does not count during halt or stop-grant.

The format of the CPU watchdog timer is shown in Figure 9-4.

![Figure 9-4. CPU Watchdog Timer Register Format](image)
CPU Watchdog Timer Enable (EN) - Bit 0. This bit specifies whether the CPU Watchdog Timer is enabled. When the bit is set to 1, the timer increments and generates a machine check when the timer expires. When cleared to 0, the timer does not increment and no machine check is generated.

CPU Watchdog Timer Time Base (TB) - Bits 2:1. Specifies the time base for the time-out period indicated in the Count Select field. The allowable time base values are provided in Table 9-1.

### Table 9-1. CPU Watchdog Timer Time Base

<table>
<thead>
<tr>
<th>TB[1:0]</th>
<th>Time Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>00b</td>
<td>1 millisecond</td>
</tr>
<tr>
<td>01b</td>
<td>1 microsecond</td>
</tr>
<tr>
<td>10b</td>
<td>Reserved</td>
</tr>
<tr>
<td>11b</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

CPU Watchdog Timer Count Select (CS) - Bits 6:3. Specifies the time period required for the CPU Watchdog Timer to expire. The time period is this value times the time base specified in the Time Base field. The allowable values are shown in Table 9-2.

### Table 9-2. CPU Watchdog Timer Count Select

<table>
<thead>
<tr>
<th>CS[3:0]</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b</td>
<td>4095</td>
</tr>
<tr>
<td>0001b</td>
<td>2047</td>
</tr>
<tr>
<td>0010b</td>
<td>1023</td>
</tr>
<tr>
<td>0011b</td>
<td>511</td>
</tr>
<tr>
<td>0100b</td>
<td>255</td>
</tr>
<tr>
<td>0101b</td>
<td>127</td>
</tr>
<tr>
<td>0110b</td>
<td>63</td>
</tr>
<tr>
<td>0111b</td>
<td>31</td>
</tr>
<tr>
<td>1000b</td>
<td>8191</td>
</tr>
<tr>
<td>1001b</td>
<td>16383</td>
</tr>
<tr>
<td>1010b~</td>
<td></td>
</tr>
<tr>
<td>1111b</td>
<td></td>
</tr>
</tbody>
</table>

9.3.2 Error-Reporting Register Banks

Each error-reporting register bank contains the following registers:

- Machine-check control register (MCi_CTL).
- Machine-check status register (MCi_STATUS).
- Machine-check address register (MCi_ADDR).
- Machine-check miscellaneous error-information register 0 (MCi_MISC0).
The $i$ in each register name corresponds to the number of a supported register bank. Each error-reporting register bank is normally associated with a specific execution unit. The number of error-reporting register banks is implementation-specific. For more information, see the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product.

Software reads the MCG_CAP register to determine the number of supported register banks. The first error-reporting register (MC0_CTL) always starts with MSR address 400h, followed by MC0_STATUS (401h), MC0_ADDR (402h), and MC0_MISC0 (403h). The addresses of any additional error-reporting MSRs are assigned sequentially starting at 404h through the remaining supported register banks.

**MCA Overflow.** If an error occurs within an error reporting bank while the status register for that bank contains valid data ($\text{MC}_i\_\text{STATUS}[\text{VAL}] = 1$), an MCA overflow condition results. In this situation, information about the new error will either be discarded or will replace the information about the prior error.

Hardware sets the $\text{MC}_i\_\text{STATUS}[\text{OVER}]$ bit to indicate this condition has occurred and follows a set of rules to determine whether to overwrite the previously logged error information or discard the new error information. These rules are shown in Table 9-3 below.

![Table 9-3. Error Logging Priorities](image)

If the VAL bit is not set, hardware writes the appropriate logging registers based on the type of error (writing the $\text{MC}_i\_\text{STATUS}$ register last) and then sets the VAL bit to indicate to software that the information currently contained in the $\text{MC}_i\_\text{STATUS}$ register is valid. Software clears the VAL bit after reading the contents of this register (after reading and saving valid information stored in any of the other logging registers) to indicate to hardware that it has saved the information, making the registers available to log the next error.

If survivable MCA overflow is supported by the implementation (as indicated by CPUID Fn8000_0007_EBX[\text{McaOverflowRecov}] = 1), the state of the $\text{MC}_i\_\text{STATUS}[\text{PCC}]$ bit indicates whether system execution can continue. If a particular processor does not support survivable MCA overflow and overflow occurs, software must halt instruction execution on that processor core regardless of the state of the PCC bit because critical information may have been lost as a result of the
overflow. See the description of the Machine-Check Status registers below for more information on the PCC bit.

**Machine-Check Recovery.** Machine Check Recovery is a feature allowing recovery of the system when the hardware cannot correct an error. Machine Check Recovery is supported when Fn8000_0007_EBX[SUCCOR]=1.

When Machine Check Recovery is supported and an uncorrected error has been detected that the hardware can contain to the task or process to which the machine check has been delivered, it logs a context-synchronous uncorrectable error (MCi_STATUS[UC]=1, MCi_STATUS[PCC]=0). The rest of the system is unaffected and may continue running if supervisory software can terminate only the affected process context.

**Machine-Check Control Registers.** The machine-check control registers (MCi_CTL), as shown in Figure 9-5, contain an enable bit for each error source within an error-reporting register bank. Setting an enable bit to 1 enables error reporting for the specific feature controlled by the bit, and clearing the bit to 0 disables error reporting for the feature. It is recommended that the value FFFF_FFFF_FFFF_FFFFh be programmed into each MCi_CTL register.

Disabling the reporting of errors from error sources that are capable of detecting uncorrected errors can compromise future error recovery and is not recommended. Other implementation-specific values are documented in the product’s BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual.

63 2 1 0

<table>
<thead>
<tr>
<th>E</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

Error-Reporting Register-Bank Enable Bits

---

Figure 9-5. MCi_CTL Register

**Machine-Check Status Registers.** Each error-reporting register bank includes a machine-check status register (MCi_STATUS) that the processor uses to log error information. Hardware writes the status register bits when an error is detected, and sets the VAL bit of the register to 1, indicating that the status information is valid. Error reporting for the error source associated with the detected error does not need to be enabled in the MCi_CTL Register for the processor to write the status register. Error reporting must be enabled for the error to be reported via a #MC exception. Software is responsible for clearing the status register after the exception has been handled. Attempting to write a value other than 0 to an MCi_STATUS register will raise a general-protection (#GP) exception.

Figure 9-6 on page 280 shows the format of the MCi_STATUS register.
The fields within the MC\textsubscript{i} STATUS register are:

- **MCA Error Code**—Bits 15:0. This field encodes information about the error, including:
  - The type of transaction that caused the error.
  - The memory-hierarchy level involved in the error.
  - The type of request that caused the error.
  - Other information concerning the transaction type.

See the *BIOS and Kernel Developer's Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product for information on the format and encoding of the MCA error code.

- **Model-Specific Extended Error Code**—Bits 31:16. This field encodes model-specific information about the error. For further information, see the documentation for particular implementations of the architecture.
• **Implementation-specific Information**—Bits 56, 54:45, 42:32. These bit ranges hold model-specific error information. Software should not rely on the field definitions in these ranges being consistent between processor implementations. For details see the BKDG or PPR for desired implementations of the architecture.

• **Poison**—Bit 43. When set to 1, this bit indicates that the uncorrected error condition being reported is due to the attempted use of data that was previous detected as in error (and could not be corrected) and marked as known-bad.

• **Deferred**—Bit 44. When set to 1, this bit indicates that hardware has determined that the error condition being logged has not affected the execution of any instruction stream and that action by system software to prevent or correct an error is not required. No machine-check exception is signalled. Hardware will monitor the error and log an uncorrected error when the execution of any thread of execution is impacted.

• **TCC**—Bit 55. When set to 1, this bit indicates that the hardware context of the process thread to which the error was reported may have been corrupted. Continued operation of the thread may have unpredictable results. When this bit is cleared, the hardware context of the process thread to which the error was reported is not corrupted and recovery of the process thread is possible. This bit is only meaningful when MCA_STATUS[PCC]=0.

• **PCC**—Bit 57. When set to 1, this bit indicates that the processor state is likely to be corrupt due to an uncorrected error. In this case, it is possible that software cannot reliably continue execution. When this bit is cleared, the processor state is not corrupted and recovery is still possible. If the PCC bit is set in any error bank, the processor will clear RIPV and EIPV in the MCG_STATUS register.

• **ADDRV**—Bit 58. When set to 1, this bit indicates that the contents of the corresponding error-reporting address register (MCi_ADDR) are valid. When this bit is cleared, the contents of MCi_ADDR are not valid.

• **MISCV**—Bit 59. When set to 1, this bit indicates that additional information about the error is saved in the corresponding error-reporting miscellaneous register (MCi_MISC0). When cleared, this bit indicates that the contents of the MCi_MISC0 register are not valid.

• **EN**—Bit 60. When set to 1, this bit indicates that the error condition is enabled in the corresponding error-reporting control register (MCi_CTL). Errors disabled by MCi_CTL do not cause a machine-check exception.

• **UC**—Bit 61. When set to 1, this bit indicates that the logged error status is for an uncorrected error. When cleared, the error class is determined by looking at the Deferred bit; the error is a Corrected error if the Deferred bit is clear or a Deferred error if the Deferred bit is set. (See Section 9.1.2, “Error Detection, Logging, and Reporting,” on page 270, for more detail on these error classes.)

• **OVER**—Bit 62. This bit is set to 1 by the processor if the VAL bit is already set to 1 as the processor attempts to write error information into MCi_STATUS. In this situation, the machine-check mechanism handles the contents of MCi_STATUS as follows:
  - For processor implementations that log errors for disabled reporting banks, status for an enabled error replaces status for a disabled error.
  - Status for a deferred error replaces status for a corrected error.
- Status for an uncorrected error replaces status for a corrected or deferred error.
- Status for an enabled uncorrected error is never replaced.

See Section “MCA Overflow” on page 278 for more information on this field.

- **VAL**—Bit 63. This bit is set to 1 by the processor if the contents of MCI_STATUS are valid.
  Software should clear the VAL bit after reading the MCI_STATUS register, otherwise a subsequent machine-check error sets the OVER bit as described above.

When a machine-check error occurs, the processor writes an error code into the appropriate MCI_STATUS register MCA error-code field. The MCI_STATUS[VAL] bit is set to 1, indicating that the MCI_STATUS register contents are valid.

MCA error-codes are used to report errors in the memory hierarchy, the system bus, and the system-interconnection logic. Error-codes are divided into subfields that are used to describe the cause of an error. The information is implementation-specific. For further information, see the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual applicable to your product.

**Machine-Check Address Registers.** Each error-reporting register bank includes a machine-check address register (MCI_ADDR) that the processor uses to report the address or location associated with the logged error. The address field can hold a virtual (linear) address, a physical address, or a value indicating an internal physical location, depending on the type of error. For further information, see the documentation for particular implementations of the architecture. The contents of this register are valid only if the ADDRV bit in the corresponding MCI_STATUS register is set to 1.

**Machine-Check Miscellaneous-Error Information Register 0(MCI_MISC0).** Each error-reporting register bank includes the Machine-Check Miscellaneous 0 register that the processor uses to report additional error information.

In some implementations, the MCI_MISC0 register is used for error thresholding. Thresholding is a mechanism provided by hardware to:

- count detected errors, and
- (optionally) generate an APIC-based interrupt when a programmed number of errors has been counted.

Processor hardware counts detected errors and ensures that multiple error sources do not share the same thresholding register. Software can use corrected error counts to help predict which components might soon fail (begin generating uncorrectable errors) and schedule their replacement.

Threshold counters increment for error sources that are enabled for logging.

The MCI_MISC0[BlkPtr] field is used to point to any additional MCI_MISCj registers, where $j > 0$. If this field is zero, no additional MCI_MISC registers are implemented. If this field is one, and Fn8000_0007_EBX[ScalableMca]=1, additional MCI_MISC registers are implemented.

**Additional Machine-Check Miscellaneous-Error Information Registers (MCI_MISCj).** If the MCI_MISC0[BlkPtr] field is non-zero and Fn8000_0007_EBX[ScalableMca]=0, up to 8 additional
MCi_MISCj registers can be implemented for the error-reporting bank \( i \) (for a total of 9). These registers are allocated in contiguous blocks of 8, with MCi_MISC1 addressed by:

\[
\text{MCi}_i \_\text{MISC1 address} = \text{C000}_0400h + (\text{MCi}_i \_\text{MISC0}[\text{BlkPtr}] << 3)
\]

This is illustrated in Figure 9-7 below.

![Figure 9-7. MCi_MISC1 Addressing](image)

The format of implemented MCi_MISCj registers depends upon their use and use can vary from one implementation to another. Figure 9-8 below illustrates the format of a miscellaneous error information register when used as an error thresholding register.

All miscellaneous error information registers will contain the VAL field in bit position 63. MCi_MISC0 must contain the BLKP field in bits 31:24.
The fields within the \textit{MC}_i\_MISC_j register are:

- \textit{Valid (VAL)}—Bit 63. When set to 1, indicates that the counter present (CTRP) and block pointer (BLKP) fields in this register are valid.
- \textit{Counter Present (CTRP)}—Bit 62. When set to 1, indicates the presence of a threshold counter.
- \textit{Locked (LKD)}—Bit 61. When set to 1, indicates that the threshold counter is not available for OS use. If this is the case, writes to bits 60:0 of this register are ignored and do not generate a fault. Software must check the Locked bit before writing into the thresholding register. This field is write-enabled by MSR C001_0015h Hardware Configuration Register [MCSTATUSWrEn].
- \textit{IntP (Thresholding Interrupt Supported)}—Bit 60. When set, this bit indicates that the reporting of threshold overflow via interrupt is supported. Interrupt type is determined by the setting of the INTT field.
- \textit{LVT Offset (LVTOFF)}—Bits 55:52. This field specifies the address of the APIC LVT entry to deliver the threshold counter interrupt. Software must initialize the APIC LVT entry before enabling the threshold counter to generate the APIC interrupt; otherwise, undefined behavior may result.

\[
\text{APIC LVT address} = (\text{MC}_i\_MISC_j[\text{LvtOff}] \ll 4) + 500h
\]
• **Counter Enable (CNTE)—**Bit 51. When set to 1, counting of implementation-dependent errors is enabled; otherwise, counting is disabled.

• **Interrupt Type (INTT)—**Bits 50:49. The value of this field specifies the type of interrupt signaled when the value of the overflow bit changes from 0 to 1.
  - 00b = No interrupt
  - 01b = APIC-based interrupt
  - 10b = Reserved
  - 11b = Reserved

• **Overflow (OF)—**Bit 48. The value of this field is maintained through a warm reset. This bit is set by hardware when the error counter increments to its maximum implementation-supported value (from FFFFe to FFFFh for the maximum implementation-supported value). This is defined as the threshold level. When the overflow bit is set, the interrupt selected by the interrupt type field is generated. Software must reset this bit to zero in the interrupt handler routine when they update the error counter.

• **Error Counter (ERRCT)—**Bits 47:32. This field is maintained through a warm reset. The size of the threshold counter is implementation-dependent. Implementations with less than 16 bits fill the most significant unimplemented bits with zeros.

  Software enumerates the counter bits to discover the size of the counter and the threshold level (when counter increments to the maximum count implemented). Software sets the starting error count as follows:

  \[
  \text{Starting error count} = \text{threshold level} - \text{desired software error count to cause overflow}
  \]

  The error counter is incremented by hardware when errors for the associated error counter are logged. When this counter overflows, it stays at the maximum error count (with no rollover).

• **Block pointer for additional MISC registers (BLKP)—**Bits 31:24. This field is only valid when valid (VAL) bit is set. When non-zero, this field is used to indicate the presence of additional MCi_MISC registers.

Other formats for miscellaneous information registers are implementation-dependent, see the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product for more details.

### 9.4 Initializing the Machine-Check Mechanism

Following a processor reset, all machine-check error-reporting enable bits are disabled. System software must enable these bits before machine-check errors can be reported. Generally, system software should initialize the machine-check mechanism using the following process:

• Execute the CPUID instruction and verify that the processor supports the machine-check exception (MCE) and machine-check registers (MCA). Software should not proceed with initializing the machine-check mechanism if the machine-check registers are not supported.

• If the machine-check registers are supported, system software should take the following steps:
- Check to see if the CTLP bit in the MCG_CAP register is set to 1. If it is, then the MCG_CTL register is supported by the processor. If the MCG_CTL register is supported, software should set its enable bits to 1 for the machine-check features it uses. Software can load MCG_CTL with all 1s to enable all available machine-check reporting banks.

- Read the COUNT field from the MCG_CAP register to determine the number of error-reporting register banks supported by the processor. For each error-reporting register bank, software should set the enable bits to 1 in the MCi_CTL register for the error types it wants the processor to report. Software can write each MCi_CTL with all 1s to enable all error-reporting mechanisms.

Not enabling reporting banks that may be involved in the reporting of uncorrected errors can lead to the loss of system reliability and error recoverability.

- Check the VAL bit on each implemented MCi_STATUS register. It is possible that valid error-status information has already been logged in the MCi_STATUS registers at the time software is attempting to initialize them. The status can reflect errors logged prior to a warm reset or errors recorded during the system power-up and boot process. Before clearing the MCi_STATUS registers, software should examine their contents and log any errors found.

- After saving any valid error information contained in the MCi_STATUS, MCi_ADDR, and any implemented miscellaneous error information registers for each implemented reporting bank, software should clear all status fields in the MCi_STATUS register for each bank by writing all 0s to the register.

As a final step in the initialization process, system software should enable the machine-check exception by setting CR4[MCE] to 1.

A machine-check condition that occurs while CR4[MCE] is cleared will result in the processor core entering the shutdown state.

### 9.5 Using MCA Features

System software can detect and handle logged errors using three methods:

1. **Polling**
   
   Software can periodically examine the machine-check status registers for errors, and save any error information found. Uncorrected errors found during polling will require some type of immediate response to initiate recovery or shutdown.

2. **Enabling machine-check reporting**
   
   When reporting is enabled, any uncorrected error that occurs causes control to be transferred to the machine-check exception handler. The exception handler can be designed for a specific processor implementation or can be generalized to work on multiple implementations.

3. **Setting up and enabling interrupts for deferred and corrected errors**
   
   In many implementations, MCA hardware can be configured to generate an interrupt hardware on the detection of a deferred error or when a programmed corrected error threshold is reached.
These methods are not mutually exclusive.

9.5.1 Determining the Scope of Detected Errors

Table 9-4 details the actions that recovery software should take and the level of recovery possible based on status information returned in the MCi_STATUS and MCG_STATUS registers.

<table>
<thead>
<tr>
<th>MCI_STATUS</th>
<th>Error Scope</th>
<th>PCC</th>
<th>TCC</th>
<th>UC</th>
<th>Deferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1</td>
<td>System fatal error. Error has corrupted the processor core architectural state. System processing must be terminated.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>Recoverable error. If software can correct the error, the interrupted program can be resumed.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>Containable error. The interrupted instruction stream cannot be resumed. System-level recovery may be possible if software can localize the error and terminate any affected software processes.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>Deferred error. Immediate software action is not required. A latent error has been discovered, but not yet consumed. Error handling software may attempt to correct this data error, or prevent access by processes which map the data, or make the physical resource containing the data inaccessible.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>Hardware corrected error. No software action is required. Error information should be saved for analysis.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

9.5.2 Handling Machine Check Exceptions

The processor uses the interrupt control-transfer mechanism to invoke an exception handler after a machine-check exception occurs. This requires system software to initialize the interrupt-descriptor table (IDT) with either an interrupt gate or a trap gate that references the interrupt handler. See “Legacy Protected-Mode Interrupt Control Transfers” on page 245 and “Long-Mode Interrupt Control Transfers” on page 255 for more information on interrupt control transfers.

At a minimum, the machine-check exception handler must be capable of logging errors for later examination. This can be a sufficient implementation for some handlers. More thorough exception-handler implementations can analyze the error to determine if it is unrecoverable, and whether it can be recovered in software.

Machine-check exception handlers that attempt recovery must be thorough in their analysis and their corrective actions. The following guidelines should be used when writing such a handler:

- The status registers in all the enabled error-reporting register banks must be examined to identify the cause of the machine-check exception. Read the COUNT field from MCG_CAP to determine the number of status registers supported by the processor.
- Check the valid bit in each status register (MCi_STATUS[VAL]). The MCi_STATUS register does not need to be examined when its valid bit is clear.
Check the valid MCi_STATUS registers to see if error recovery is possible. Error recovery is not possible when:
- The processor-context corrupt bit (MCi_STATUS[PCC]) is set to 1.
- The error-overflow status bit (MCi_STATUS[OVER]) is set and the processor does not support recoverable MCi_STATUS overflow (as indicated by feature bit CPUID Fn8000_0007_EBX[McaOverflowRecov] = 0).
- The processor does not support Machine Check Recovery as indicated by feature bit CPUID Fn8000_0007_EBX[SUCCOR].

If error recovery is not possible, the handler should log the error information and return to the system software responsible for shutting down the processor core.

Check the MCi_STATUS[UC] bit to see if the processor corrected the error. If UC is set, the processor did not correct the error and the exception handler must correct the error before restarting the interrupted program.
- If MCA Recovery is supported:
  • Check MCA_STATUS[TCC].
    • If TCC is set, the context of the process thread executing on the interrupted logical core may be corrupt and the thread cannot be recovered. The rest of the system is unaffected; it is possible to terminate only the affected process thread.
    • If TCC is clear, the context of the process thread executing on the interrupted logical core is not corrupt. Recovery of the process thread may be possible, but only if the uncorrected error condition is first corrected by software; otherwise, the interrupted process thread must be terminated.

If the handler cannot correct the error or the MCG_STATUS[RIPV] bit is cleared, it should not return control to the interrupted program, but should log the error information and terminate the software process that was about to consume the uncorrected data. If the error has compromised the state of a guest operating system, the guest should be restarted. If the state of the virtual machine has been corrupted, the virtual machine must be reinitialized.

When identifying the error condition, portable exception handlers should examine only the architecturally defined fields of the MCi_STATUS register.

If the MCG_STATUS[RIPV] bit is set, the interrupted program can be restarted reliably at the instruction pointer address pushed onto the exception handler stack. If RIPV = 0, the interrupted program cannot be restarted reliably at that location, although it can be restarted at that location for debugging purposes.

When logging errors, particularly those that are not recoverable, check the MCG_STATUS[EIPV] bit to see if the instruction-pointer address pushed onto the exception handler stack is related to the machine-check error. If EIPV = 0, the address is not guaranteed to be related to the error.

Before exiting the machine-check handler, clear the MCG_STATUS[MCIP] bit. MCIP indicates a machine-check exception occurred. If this bit is set when another machine-check exception occurs, the processor enters the shutdown state.
• When an exception handler is able to, at a minimum, successfully log an error condition, the
MCi_STATUS registers should be cleared before exiting the machine-check handler. Software is
responsible for clearing at least the MCi_STATUS[VAL] bits.

• Additional machine-check exception-handler portability can be added by having the handler use
the CPUID instruction to identify the processor and its capabilities. Implementation-specific
software can be added to the machine-check exception handler based on the processor information
reported by CPUID.

9.5.3 Reporting Corrected Errors

Machine-check exceptions do not occur if the error is corrected by the processor. If system software
wishes to detect and save information concerning corrected machine-check errors, a system-service
routine must be provided to check the contents of the machine-check status registers for corrected
errors. The service routine can be invoked by system software on a periodic basis, or by an error-
thresholding interrupt.

A service routine that gathers error information for corrected errors should perform the following:

• Examine the status register (MCi_STATUS) in each of the enabled error-reporting register banks.
For each MCi_STATUS register with a set valid bit (VAL=1), the service routine should:
  - Save the contents of the MCi_STATUS register.
  - Save the contents of the corresponding MCi_ADDR register if MCi_STATUS[ADDRV] = 1.
  - Save the contents of the corresponding MCi_MISC register if MCi_STATUS[MISCV] = 1.
• Once the information found in the error-reporting register banks is saved, the MCi_STATUS
register should be cleared. This allows the processor to properly report any subsequent errors in the
MCi_STATUS registers.
• The service routine can save the time-stamp counter with each error logged. This can help in
determining how frequently errors occur. For further information, see “Time-Stamp Counter” on
page 377.
• In multiprocessor configurations, the service routine can save the processor-node identifier. This
can help locate a failing multiprocessor-system component, which can then be isolated from the
rest of the system. For further information, see the documentation for particular implementations
of the architecture.
10 System-Management Mode

System-management mode (SMM) is an operating mode designed for system-control activities like power management. Normally, these activities are transparent to conventional operating systems and applications. SMM is used by platform firmware and specialized low-level device drivers, rather than the operating system.

The SMM interrupt-handling mechanism differs substantially from the standard interrupt-handling mechanism described in Chapter 8, “Exceptions and Interrupts.” SMM is entered using a special external interrupt called the system-management interrupt (SMI). After an SMI is received by the processor, the processor saves the processor state in a separate address space, called SMRAM. The SMM-handler software and data structures are also located in the SMRAM space. Interrupts and exceptions that ordinarily cause control transfers to the operating system are disabled when SMM is entered. The processor exits SMM, restores the saved processor state, and resumes normal execution by using a special instruction, RSM.

In SMM, address translation is disabled and addressing is similar to real mode. SMM programs can address up to 4 Gbytes of physical memory. See “SMM Operating-Environment” on page 301 for additional information on memory addressing in SMM.

The following sections describe the components of the SMM mechanism:

- “SMM Resources” on page 292—this section describes SMRAM, the SMRAM save-state area used to hold the processor state, and special SMRAM save-state entries used in support of SMM.
- “Using SMM” on page 301—this section describes the mechanism of entering and exiting SMM. It also describes SMM memory allocation, addressing, and interrupts and exceptions.

Of these mechanisms, only the format of the SMRAM save-state area differs between the AMD64 architecture and the legacy architecture.

Note: Model-independent aspects of SMM operation are described here; see the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual of a given processor family for possible model-specific details.

10.1 SMM Differences

There are functional differences between the SMM support in the AMD64 architecture and the SMM support found in previous architectures. These are:

- The SMRAM state-save area layout is changed to hold the 64-bit processor state.
- The initial processor state upon entering SMM is expanded to reflect the 64-bit nature of the processor.
- New conditions exist that can cause a processor shutdown while in SMM.
The auto-halt restart and I/O-instruction restart entries in the SMRAM state-save area are one byte each instead of two bytes each.

SMRAM caching considerations are modified because the legacy FLUSH# external signal (writeback, if modified, and invalidate) is not supported on implementations of the AMD64 architecture.

Some previous AMD x86 processors saved and restored the CR2 register in the SMRAM state-save area. This register is not saved by the SMM implementation in the AMD64 architecture. SMM handlers that save and restore CR2 must perform the operation in software.

10.2 SMM Resources

The SMM resources supported by the processor consist of SMRAM, the SMRAM state-save area, and special entries within the SMRAM state-save area. In addition to the save-state area, SMRAM includes space for the SMM handler.

10.2.1 SMRAM

SMRAM is the memory-address space accessed by the processor when in SMM. The default size of SMRAM is 64 Kbytes and can range in size between 32 Kbytes and 4 Gbytes. System logic can use physically separate SMRAM and main memory, directing memory transactions to SMRAM after recognizing SMM is entered, and redirecting memory transactions back to system memory after recognizing SMM is exited. When separate SMRAM and main memory are used, the system designer needs to provide a method of mapping SMRAM into main memory so that the SMI handler and data structures can be loaded.

Figure 10-1 on page 293 shows the default SMRAM memory map. The default SMRAM code-segment (CS) has a base address of 0003_0000h (the base address is automatically scaled by the processor using the CS-selector register, which is set to the value 3000h). This default SMRAM-base address is known as $SMBASE$. A 64-Kbyte memory region, addressed from 0003_0000h to 0003_FFFFh, makes up the default SMRAM memory space. The top 32 Kbytes (0003_8000h to 0003_FFFFh) must be supported by system logic, with physical memory covering that entire address range. The top 512 bytes (0003_FE00h to 0003_FFFFh) of this address range are the default SMM state-save area. The default entry point for the SMM interrupt handler is located at 0003_8000h.
10.2.2 SMBASE Register

The format of the SMBASE register is shown in Figure 10-2. SMBASE is an internal processor register that holds the value of the SMRAM-base address. SMBASE is set to 30000h after a processor reset.

![Figure 10-2. SMBASE Register](image)

In some operating environments, relocation of SMRAM to a higher memory area can provide more low memory for legacy software. SMBASE relocation is supported when the SMM-base relocation bit in the SMM-revision identifier (bit 17) is set to 1. In processors implementing the AMD64 architecture, SMBASE relocation is always supported.

Software can only modify SMBASE (relocate the SMRAM-base address) by entering SMM, modifying the SMBASE image stored in the SMRAM state-save area, and exiting SMM. The SMM-
handler entry point must be loaded at the new memory location specified by SMBASE+8000h. The next time SMM is entered, the processor saves its state in the new state-save area at SMBASE+0FE00h, and begins executing the SMM handler at SMBASE+8000h. The new SMBASE address is used for every SMM until it is changed, or a hardware reset occurs.

When SMBASE is used to relocate SMRAM to an address above 1 Mbyte, 32-bit address-size-override prefixes must be used to access this memory. This is because addressing in SMM behaves as it does in real mode, with a 16-bit default operand size and address size. The values in the 16-bit segment-selector registers are left-shifted four bits to form a 20-bit segment-base address. Without using address-size overrides, the maximum computable address is 10FFEFh.

Because SMM memory-addressing is similar to real-mode addressing, the SMBASE address must be less than 4 Gbytes.

### 10.2.3 SMRAM State-Save Area

When an SMI occurs, the processor saves its state in the 512-byte SMRAM state-save area during the control transfer into SMM. The format of the state-save area defined by the AMD64 architecture is shown in Table 10-1. This table shows the offsets in the SMRAM state-save area relative to the SMRAM-base address. The state-save area is located between offset 0_FE00h (SMBASE+0_FE00h) and offset 0_FFFFh (SMBASE+0_FFFFh). Software should not modify offsets specified as read-only or reserved, otherwise unpredictable results can occur.

**Table 10-1. AMD64 Architecture SMM State-Save Area**

<table>
<thead>
<tr>
<th>Offset (Hex) from SMBASE</th>
<th>Contents</th>
<th>Size</th>
<th>Allowable Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>FE00h</td>
<td>ES</td>
<td>Selector</td>
<td>Word</td>
</tr>
<tr>
<td>FE02h</td>
<td></td>
<td>Attributes</td>
<td>Word</td>
</tr>
<tr>
<td>FE04h</td>
<td></td>
<td>Limit</td>
<td>Doubleword</td>
</tr>
<tr>
<td>FE08h</td>
<td></td>
<td>Base</td>
<td>Quadword</td>
</tr>
<tr>
<td>FE10h</td>
<td>CS</td>
<td>Selector</td>
<td>Word</td>
</tr>
<tr>
<td>FE12h</td>
<td></td>
<td>Attributes</td>
<td>Word</td>
</tr>
<tr>
<td>FE14h</td>
<td></td>
<td>Limit</td>
<td>Doubleword</td>
</tr>
<tr>
<td>FE18h</td>
<td></td>
<td>Base</td>
<td>Quadword</td>
</tr>
<tr>
<td>FE20h</td>
<td>SS</td>
<td>Selector</td>
<td>Word</td>
</tr>
<tr>
<td>FE22h</td>
<td></td>
<td>Attributes</td>
<td>Word</td>
</tr>
<tr>
<td>FE24h</td>
<td></td>
<td>Limit</td>
<td>Doubleword</td>
</tr>
<tr>
<td>FE28h</td>
<td></td>
<td>Base</td>
<td>Quadword</td>
</tr>
</tbody>
</table>

*Note: The offset for the SMM-revision identifier is compatible with previous implementations.*
### Table 10-1. AMD64 Architecture SMM State-Save Area (continued)

<table>
<thead>
<tr>
<th>Offset (Hex) from SMBASE</th>
<th>Contents</th>
<th>Size</th>
<th>Allowable Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>FE30h</td>
<td>DS</td>
<td>Selector</td>
<td>Word</td>
</tr>
<tr>
<td>FE32h</td>
<td>Attributes</td>
<td>Word</td>
<td></td>
</tr>
<tr>
<td>FE34h</td>
<td>Limit</td>
<td>Doubleword</td>
<td></td>
</tr>
<tr>
<td>FE38h</td>
<td>Base</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FE40h</td>
<td>FS</td>
<td>Selector</td>
<td>Word</td>
</tr>
<tr>
<td>FE42h</td>
<td>Attributes</td>
<td>Word</td>
<td></td>
</tr>
<tr>
<td>FE44h</td>
<td>Limit</td>
<td>Doubleword</td>
<td></td>
</tr>
<tr>
<td>FE48h</td>
<td>Base</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FE50h</td>
<td>GS</td>
<td>Selector</td>
<td>Word</td>
</tr>
<tr>
<td>FE52h</td>
<td>Attributes</td>
<td>Word</td>
<td></td>
</tr>
<tr>
<td>FE54h</td>
<td>Limit</td>
<td>Doubleword</td>
<td></td>
</tr>
<tr>
<td>FE58h</td>
<td>Base</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FE60h–FE63h</td>
<td>GDTR</td>
<td>Reserved</td>
<td>4 Bytes</td>
</tr>
<tr>
<td>FE64h</td>
<td>Limit</td>
<td>Word</td>
<td></td>
</tr>
<tr>
<td>FE66h–FE67h</td>
<td>Reserved</td>
<td>2 Bytes</td>
<td></td>
</tr>
<tr>
<td>FE68h</td>
<td>Base</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FE70h</td>
<td>LDTR</td>
<td>Selector</td>
<td>Word</td>
</tr>
<tr>
<td>FE72h</td>
<td>Attributes</td>
<td>Word</td>
<td></td>
</tr>
<tr>
<td>FE74h</td>
<td>Limit</td>
<td>Doubleword</td>
<td></td>
</tr>
<tr>
<td>FE78h</td>
<td>Base</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FE80h–FEB3h</td>
<td>IDTR</td>
<td>Reserved</td>
<td>4 Bytes</td>
</tr>
<tr>
<td>FEB4h</td>
<td>Limit</td>
<td>Word</td>
<td></td>
</tr>
<tr>
<td>FEB6h–FEB7h</td>
<td>Reserved</td>
<td>2 Bytes</td>
<td></td>
</tr>
<tr>
<td>FE88h</td>
<td>Base</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FE90h</td>
<td>TR</td>
<td>Selector</td>
<td>Word</td>
</tr>
<tr>
<td>FE92h</td>
<td>Attributes</td>
<td>Word</td>
<td></td>
</tr>
<tr>
<td>FE94h</td>
<td>Limit</td>
<td>Doubleword</td>
<td></td>
</tr>
<tr>
<td>FE98h</td>
<td>Base</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FEA0h</td>
<td>I/O Instruction Restart RIP</td>
<td>Quadword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FEA8h</td>
<td>I/O Instruction Restart RCX</td>
<td>Quadword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FEB0h</td>
<td>I/O Instruction Restart RSI</td>
<td>Quadword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FEB8h</td>
<td>I/O Instruction Restart RDI</td>
<td>Quadword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FEC0h</td>
<td>I/O Instruction Restart Dword</td>
<td>Doubleword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FEC4h–FEC7h</td>
<td>Reserved</td>
<td>4 Bytes</td>
<td></td>
</tr>
</tbody>
</table>

**Note:**
1. The offset for the SMM-revision identifier is compatible with previous implementations.
Table 10-1. AMD64 Architecture SMM State-Save Area (continued)

<table>
<thead>
<tr>
<th>Offset (Hex) from SMBASE</th>
<th>Contents</th>
<th>Size</th>
<th>Allowable Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEC8h</td>
<td>I/O Instruction Restart</td>
<td>Byte</td>
<td>Read/Write</td>
</tr>
<tr>
<td>FEC9h</td>
<td>Auto-Halt Restart</td>
<td>Byte</td>
<td></td>
</tr>
<tr>
<td>FECAh—FECFh</td>
<td>Reserved</td>
<td>5 Bytes</td>
<td></td>
</tr>
<tr>
<td>FED0h</td>
<td>EFER</td>
<td>Quadword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FED8h</td>
<td>SVM Guest</td>
<td>Quadword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FEE0h</td>
<td>SVM Guest VMCB Physical Address</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FEE8h</td>
<td>SVM Guest Virtual Interrupt</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FEF0h—FEF8h</td>
<td>Reserved</td>
<td>10 Bytes</td>
<td></td>
</tr>
<tr>
<td>FEFCh</td>
<td>SMM-Revision Identifier(^1)</td>
<td>Doubleword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FF00h</td>
<td>SMBASE</td>
<td>Doubleword</td>
<td>Read/Write</td>
</tr>
<tr>
<td>FF04h—FF1Fh</td>
<td>Reserved</td>
<td>27 Bytes</td>
<td></td>
</tr>
<tr>
<td>FF20h</td>
<td>SVM Guest PAT</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FF28h</td>
<td>SVM Host EFER</td>
<td>Quadword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FF30h</td>
<td>SVM Host CR4</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FF38h</td>
<td>SVM Host CR3</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FF40h</td>
<td>SVM Host CR0</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FF48h</td>
<td>CR4</td>
<td>Quadword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FF50h</td>
<td>CR3</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FF58h</td>
<td>CR0</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FF60h</td>
<td>DR7</td>
<td>Quadword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FF68h</td>
<td>DR6</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FF70h</td>
<td>RFLAGS</td>
<td>Quadword</td>
<td>Read/Write</td>
</tr>
<tr>
<td>FF78h</td>
<td>RIP</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FF80h</td>
<td>R15</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FF88h</td>
<td>R14</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FF90h</td>
<td>R13</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FF98h</td>
<td>R12</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FAA0h</td>
<td>R11</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FAA8h</td>
<td>R10</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FFB0h</td>
<td>R9</td>
<td>Quadword</td>
<td></td>
</tr>
<tr>
<td>FFB8h</td>
<td>R8</td>
<td>Quadword</td>
<td></td>
</tr>
</tbody>
</table>

**Note:**

1. The offset for the SMM-revision identifier is compatible with previous implementations.
A number of other registers are not saved or restored automatically by the SMM mechanism. See “Saving Additional Processor State” on page 303 for information on using these registers in SMM.

As a reference for legacy processor implementations, the legacy SMM state-save area format is shown in Table 10-2. *Implementations of the AMD64 architecture do not use this format.*

### Table 10-2. Legacy SMM State-Save Area (Not used by AMD64 Architecture)

<table>
<thead>
<tr>
<th>Offset (Hex) from SMBASE</th>
<th>Contents</th>
<th>Size</th>
<th>Allowable Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>FE00h—FEF7h</td>
<td>Reserved</td>
<td>248 Bytes</td>
<td>—</td>
</tr>
<tr>
<td>FEF8h</td>
<td>SMBASE</td>
<td>Doubleword</td>
<td>Read/Write</td>
</tr>
<tr>
<td>FEFCh</td>
<td>SMM-Revision Identifier</td>
<td>Doubleword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FF00h</td>
<td>I/O Instruction Restart Word</td>
<td>Word</td>
<td>Read/Write</td>
</tr>
<tr>
<td>FF02h</td>
<td>Auto-Halt Restart</td>
<td>Word</td>
<td></td>
</tr>
<tr>
<td>FF04h—FF87h</td>
<td>Reserved</td>
<td>132 Bytes</td>
<td>—</td>
</tr>
<tr>
<td>FF88h</td>
<td>GDT Base</td>
<td>Doubleword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FF8Ch—FF93h</td>
<td>Reserved</td>
<td>Quadword</td>
<td>—</td>
</tr>
<tr>
<td>FF94h</td>
<td>IDT Base</td>
<td>Doubleword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FF98h—FFA7h</td>
<td>Reserved</td>
<td>16 Bytes</td>
<td>—</td>
</tr>
</tbody>
</table>

*Note:*
1. The offset for the SMM-revision identifier is compatible with previous implementations.
Table 10-2. Legacy SMM State-Save Area (Not used by AMD64 Architecture) (continued)

<table>
<thead>
<tr>
<th>Offset (Hex) from SMBASE</th>
<th>Contents</th>
<th>Size</th>
<th>Allowable Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFA8h</td>
<td>ES</td>
<td>Doubleword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FFACh</td>
<td>CS</td>
<td>Doubleword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FFB0h</td>
<td>SS</td>
<td>Doubleword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FFB4h</td>
<td>DS</td>
<td>Doubleword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FFB8h</td>
<td>FS</td>
<td>Doubleword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FFBCh</td>
<td>GS</td>
<td>Doubleword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FFC0h</td>
<td>LDT Base</td>
<td>Doubleword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FFC4h</td>
<td>TR</td>
<td>Doubleword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FFC8h</td>
<td>DR7</td>
<td>Doubleword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FFCCh</td>
<td>DR6</td>
<td>Doubleword</td>
<td>Read-Only</td>
</tr>
<tr>
<td>FFD0h</td>
<td>EAX</td>
<td>Doubleword</td>
<td>Read/Write</td>
</tr>
<tr>
<td>FFD4h</td>
<td>ECX</td>
<td>Doubleword</td>
<td>Read/Write</td>
</tr>
<tr>
<td>FFD8h</td>
<td>EDX</td>
<td>Doubleword</td>
<td>Read/Write</td>
</tr>
<tr>
<td>FFDCh</td>
<td>EBX</td>
<td>Doubleword</td>
<td>Read/Write</td>
</tr>
<tr>
<td>FFE0h</td>
<td>ESP</td>
<td>Doubleword</td>
<td>Read/Write</td>
</tr>
<tr>
<td>FFE4h</td>
<td>EBP</td>
<td>Doubleword</td>
<td>Read/Write</td>
</tr>
<tr>
<td>FFE8h</td>
<td>ESI</td>
<td>Doubleword</td>
<td>Read/Write</td>
</tr>
<tr>
<td>FFECh</td>
<td>EDI</td>
<td>Doubleword</td>
<td>Read/Write</td>
</tr>
<tr>
<td>FFF0h</td>
<td>EIP</td>
<td>Doubleword</td>
<td>Read/Write</td>
</tr>
<tr>
<td>FFF4h</td>
<td>EFLAGS</td>
<td>Doubleword</td>
<td>Read/Write</td>
</tr>
<tr>
<td>FFF8h</td>
<td>CR3</td>
<td>Doubleword</td>
<td>Read/Only</td>
</tr>
<tr>
<td>FFFCh</td>
<td>CR0</td>
<td>Doubleword</td>
<td>Read/Only</td>
</tr>
</tbody>
</table>

Note:
1. The offset for the SMM-revision identifier is compatible with previous implementations.

10.2.4 SMM-Revision Identifier

The SMM-revision identifier specifies the SMM version and the available SMM extensions implemented by the processor. Software reads the SMM-revision identifier from offset FEFCh in the SMM state-save area of SMRAM. This offset location is compatible with earlier versions of SMM. Software must not write to this location. Doing so can produce undefined results. Figure 10-3 on page 299 shows the format of the SMM-revision identifier.
System-Management Mode

The fields within the SMM-revision identifier are:

- **SMM-revision Level**—Bits 15:0. Specifies the version of SMM supported by the processor. The SMM-revision level is of the form 0_xx64h, where xx starts with 00 and is incremented for later revisions to the SMM mechanism.
- **I/O Instruction Restart**—Bit 16. When set to 1, the processor supports restarting I/O instructions that are interrupted by an SMI. This bit is always set to 1 by implementations of the AMD64 architecture. See “I/O Instruction Restart” on page 305 for information on using this feature.
- **SMM Base Relocation**—Bit 17. When set to 1, the processor supports relocation of SMRAM. This bit is always set to 1 by implementations of the AMD64 architecture. See “SMBASE Register” on page 293 for information on using this feature.

All remaining bits in the SMM-revision identifier are reserved.

### 10.2.5 SMRAM Protected Areas

Two areas are provided as safe areas for SMM code and data that are not readily accessible by non-SMM applications. The SMI handler can be located in one of these two ranges, or it can be located outside of these ranges. The handler is placed in the desired range by setting SMBASE accordingly.

The ASeg range is located at a fixed address from A_0000h to B_FFFFh. The TSeg range is located at a variable base specified by the SMM_ADDR MSR with a variable size specified by the SMM_MASK MSR. These ranges must never overlap.

Each CPU memory access is in the TSeg range if the following is true:

\[
\]
For example, if the TSeg range spans 256 Kbytes starting at address 10_0000h, then SMM_ADDR = 0010_0000h and SMM_MASK = FFFC_0000h. This results in a TSeg address range from 0010_0000 to 0013_FFFFh. The TSeg range must be aligned to a 128 Kbyte boundary and the minimum TSeg size is 128 Kbytes.

### Figure 10-4. SMM_ADDR Register Format

- **SMM TSeg Base Address (BASE)**—Bits 51:17. Specifies the base address of the TSeg range of protected addresses.

### Figure 10-5. SMM_MASK Register Format

- **ASeg Address Range Enable (AE)**—Bit 0. Specifies whether the ASeg address range is enabled for protection. When the bit is set to 1, the ASeg address range is enabled for protection. When cleared to 0, the ASeg address range is disabled for protection.
• **TSeg Address Range Enable (TE)**—Bit 1. Specifies whether the TSeg address range is enabled for protection. When the bit is set to 1, the TSeg address range is enabled for protection. When cleared to 0, the TSeg address range is disabled for protection.

• **TSeg Mask (MASK)**—Bits 51:17. Specifies the mask used to determine the TSeg range of protected addresses. The physical address is in the TSeg range if the following is true:

\[
\]

Note that a processor is not required to implement all 52 bits of the physical address.

## 10.3 Using SMM

### 10.3.1 System-Management Interrupt (SMI)

SMM is entered using the system-management interrupt (SMI). SMI is an external non-maskable interrupt that operates differently from and independently of other interrupts. SMI has priority over all other external interrupts, including NMI (see “Priorities” on page 240 for a list of the interrupt priorities). SMIs are disabled when in SMM, which prevents reentrant calls to the SMM handler.

When an SMI is received by the processor, the processor stops fetching instructions and waits for currently-executing instructions to complete and write their results. The SMI also waits for all buffered memory writes to update the caches or system memory. When these activities are complete, the processor uses implementation-dependent external signaling to acknowledge back to the system that it has received the SMI.

### 10.3.2 SMM Operating-Environment

The SMM operating-environment is similar to real mode, except that the segment limits in SMM are 4 Gbytes rather than 64 Kbytes. This allows an SMM handler to address memory in the range from 0h to 0xFFFF_FFFFh. As with real mode, segment-base addresses are restricted to 20 bits in SMM, and the default operand-size and address-size is 16 bits. To address memory locations above 1 Mbyte, the SMM handler must use the 32-bit operand-size-override and address-size-override prefixes.

After saving the processor state in the SMRAM state-save area, a processor running in SMM sets the segment-selector registers and control registers into a state consistent with real mode. Other registers are also initialized upon entering SMM, as shown in Table 10-3.

### Table 10-3. SMM Register Initialization

<table>
<thead>
<tr>
<th>Register</th>
<th>Initial SMM Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS Selectors</td>
<td>SMBASE right-shifted 4 bits</td>
</tr>
<tr>
<td>Base</td>
<td>SMBASE</td>
</tr>
<tr>
<td>Limit</td>
<td>$FFFF_{16}$</td>
</tr>
<tr>
<td>Attr</td>
<td>Read-Write-Execute</td>
</tr>
</tbody>
</table>
### 10.3.3 Exceptions and Interrupts

All hardware interrupts are disabled upon entering SMM, but exceptions and software interrupts are not disabled. If necessary, the SMM handler can re-enable hardware interrupts. Software that handles interrupts in SMM should consider the following:

- **SMI**—If an SMI occurs while the processor is in SMM, it is latched by the processor. The latched SMI occurs when the processor leaves SMM.

- **NMI**—If an NMI occurs while the processor is in SMM, it is latched by the processor, but the NMI handler is not invoked until the processor leaves SMM with the execution of an RSM instruction. A pending NMI causes the handler to be invoked immediately after the RSM completes and before the first instruction in the interrupted program is executed.

  An SMM handler can unmask NMI interrupts by simply executing an IRET. Upon completion of the IRET instruction, the processor recognizes the pending NMI, and transfers control to the NMI handler. Once an NMI is recognized within SMM using this technique, subsequent NMIs are recognized until SMM is exited. Later SMIs cause NMIs to be masked, until the SMM handler unmaskes them.

- **Exceptions**—Exceptions (internal processor interrupts) are not disabled and can occur while in SMM. Therefore, the SMM-handler software should be written to avoid generating exceptions.

- **Software Interrupts**—The software-interrupt instructions (BOUND, INT\textit{n}, INT3, and INTO) can be executed while in SMM. However, it is not recommended that the SMM handler use these instructions.

- **Maskable Interrupts**—RFLAGS.IF is cleared to 0 by the processor when SMM is entered. Software can re-enable maskable interrupts while in SMM, but it must follow the guidelines listed below for handling interrupts.

- **Debug Interrupts**—The processor disables the debug interrupts when SMM is entered by clearing DR7 to 0 and clearing RFLAGS.TF to 0. The SMM handler can re-enable the debug facilities while in SMM, but it must follow the guidelines listed below for handling interrupts.

---

### Table 10-3. SMM Register Initialization (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Initial SMM Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS, ES, FS, GS, SS</td>
<td>Selector: 0000h, Base: 0000_0000_0000_0000h, Limit: FFFF_FFFFh, Attr: Read-Write</td>
</tr>
<tr>
<td>RIP</td>
<td>0000_0000_0000_8000h</td>
</tr>
<tr>
<td>RFLAGS</td>
<td>0000_0000_0000_0002h</td>
</tr>
<tr>
<td>CR0</td>
<td>PE, EM, TS, PG bits cleared to 0. All other bits are unmodified.</td>
</tr>
<tr>
<td>CR4</td>
<td>0000_0000_0000_0000h</td>
</tr>
<tr>
<td>DR7</td>
<td>0000_0000_0000_0400h</td>
</tr>
<tr>
<td>EFER</td>
<td>0000_0000_0000_0000h</td>
</tr>
</tbody>
</table>
• **INIT**—The processor does not recognize INIT while in SMM.

Because the RFLAGS.IF bit is cleared when entering SMM, the HLT instruction should not be executed in SMM without first setting the RFLAGS.IF bit to 1. Setting this bit to 1 allows the processor to exit the halt state by using an external maskable interrupt.

In the cases where an SMM handler must accept and handle interrupts and exceptions, several guidelines must be followed:

• Interrupt handlers must be loaded and accessible before enabling interrupts.

• A real-mode interrupt vector table located at virtual (linear) address 0 is required.

• Segments accessed by the interrupt handler cannot have a base address greater than 20 bits because of the real-mode addressing used in SMM. In SMM, the 16-bit value stored in the segment-selector register is left-shifted four bits to form the 20-bit segment-base address, like real mode.

• Only the IP (rIP[15:0]) is pushed onto the stack as a result of an interrupt in SMM, because of the real-mode addressing used in SMM. If the SMM handler is interrupted at a code-segment offset above 64 Kbytes, then the return address on the stack must be adjusted by the interrupt-handler, and a RET instruction with a 32-bit operand-size override must be used to return to the SMM handler.

• If the interrupt-handler is located below 1 Mbyte, and the SMM handler is located above 1 Mbyte, a RET instruction cannot be used to return to the SMM handler. In this case, the interrupt handler can adjust the return pointer on the stack, and use a far CALL to transfer control back to the SMM handler.

### 10.3.4 Invalidating the Caches

The processor can cache SMRAM-memory locations. If the system implements physically separate SMRAM and system memory, it is possible for SMRAM and system memory locations to alias into identical cache locations. In some processor implementations, the cache contents must be written to memory and invalidated when SMM is entered and exited. This prevents the processor from using previously-cached main-memory locations as aliases for SMRAM-memory locations when SMM is entered, and vice-versa when SMM is exited.

Implementations of the AMD64 architecture do not require cache invalidation when entering and exiting SMM. Internally, the processor keeps track of SMRAM and system-memory accesses separately and properly handles situations where aliasing occurs. Cached system memory and SMRAM locations can persist across SMM mode changes. Removal of the requirement to writeback and invalidate the cache simplifies SMM entry and exit and allows SMM code to execute more rapidly.

### 10.3.5 Saving Additional Processor State

Several registers are not saved or restored automatically by the SMM mechanism. These are:

• The 128-bit media instruction registers.

• The 64-bit media instruction registers.
- The x87 floating-point registers.
- The page-fault linear-address register (CR2).
- The task-priority register (CR8).
- The debug registers, DR0, DR1, DR2, and DR3.
- The memory-type range registers (MTRRs).
- Model-specific registers (MSRs).

These registers are not saved because SMM handlers do not normally use or modify them. If an SMI results in a processor reset (due to powering down the processor, for example) or the SMM handler modifies the contents of the unsaved registers, the SMM handler should save and restore the original contents of those registers. The unsaved registers, along with those stored in the SMRAM state-save area, need to be saved in a non-volatile storage location if a processor reset occurs. The SMM handler should execute the CPUID instruction to determine the feature set available in the processor, and be able to save and restore the registers required by those features. For more information on using the CPUID instruction, see Section 3.3, “Processor Feature Identification,” on page 64.

The SMM handler can execute any of the 128-bit media, 64-bit media, or x87 instructions. A simple method for saving and restoring those registers is to use the FXSAVE and FXRSTOR instructions, respectively, if it is supported by the processor. See “Saving Media and x87 Execution Unit State” on page 316 for information on saving and restoring those registers.

Floating-point exceptions can occur when the SMM handler uses media or x87 floating-point instructions. If the SMM handler uses floating-point exception handlers, they must follow the usage guidelines established in “Exceptions and Interrupts” on page 302. A simple method for dealing with floating-point exceptions while in SMM is to simply mask all exception conditions using the appropriate floating-point control register. When the exceptions are masked, the processor handles floating-point exceptions internally in a default manner, and allows execution to continue uninterrupted.

### 10.3.6 Operating in Protected Mode and Long Mode

Software can enable protected mode from SMM and it can also enable and activate long mode. An SMM handler can use this capability to enter 64-bit mode and save additional processor state that cannot be accessed from outside 64-bit mode (for example, the most-significant 32 bits of CR2).

### 10.3.7 Auto-Halt Restart

The auto-halt restart entry is located at offset FEC9h in the SMM state-save area. The size of this field is one byte, as compared with two bytes in previous versions of SMM.

When entering SMM, the processor loads the auto-halt restart entry to indicate whether SMM was entered from the halt state, as follows:

- Bit 0 indicates the processor state upon entering SMM:
  - When set to 1, the processor entered SMM from the halt state.
When cleared to 0, the processor did not enter SMM from the halt state.

- Bits 7:1 are cleared to 0.

The SMM handler can write the auto-halt restart entry to specify whether the return from SMM should take the processor back to the halt state or to the instruction-execution state specified by the SMM state-save area. The values written are:

- **Clear to 00h**—The processor returns to the state specified by the SMM state-save area.
- **Set to any non-zero value**—The processor returns to the halt state.

If the return from SMM takes the processor back to the halt state, the HLT instruction is not re-executed. However, the halt special bus-cycle is driven on the processor bus after the RSM instruction executes.

The result of entering SMM from a non-halt state and returning to a halt state is not predictable.

### 10.3.8 I/O Instruction Restart

The I/O-instruction restart entry is located at offset FEC8h in the SMM state-save area. The size of this field is one byte, as compared with two bytes in previous versions of SMM. The I/O-instruction restart mechanism is supported when the I/O-instruction restart bit (bit 16) in the SMM-revision identifier is set to 1. This bit is always set to 1 in the AMD64 architecture.

When an I/O instruction is interrupted by an SMI, the I/O-instruction restart entry specifies whether the interrupted I/O instruction should be re-executed following an RSM that returns from SMM. Re-executing a trapped I/O instruction is useful, for example, when an I/O write is performed to a powered-down disk drive. When this occurs, the system logic monitoring the access can issue an SMI to have the SMM handler power-up the disk drive and retry the I/O write. The SMM handler does this by querying system logic and detecting the failed I/O write, asking system logic to initiate the disk-drive power-up sequence, enabling the I/O instruction restart mechanism, and returning from SMM. Upon returning from SMM, the I/O write to the disk drive is restarted.

When an SMI occurs, the processor always clears the I/O-instruction restart entry to 0. If the SMI interrupted an I/O instruction, then the SMM handler can modify the I/O-instruction restart entry as follows:

- **Clear to 00h (default value)**—The I/O instruction is not restarted, and the instruction following the interrupted I/O-instruction is executed. When a REP (repeat) prefix is used with an I/O instruction, it is possible that the next instruction to be executed is the next I/O instruction in the repeat loop.
- **Set to any non-zero value**—The I/O instruction is restarted.

While in SMM, the handler must determine the cause of the SMI and examine the processor state at the time the SMI occurred to determine whether or not an I/O instruction was interrupted. Implementations provide state information in the SMM save-state area to assist in this determination:

- **I/O Instruction Restart DWORD**—indicates whether the SMI interrupted an I/O instruction, and saves extra information describing the I/O instruction.
• I/O Instruction Restart RIP—the RIP of the interrupted I/O instruction.
• I/O Instruction Restart RCX—the RCX of the interrupted I/O instruction.
• I/O Instruction Restart RSI—the RSI of the interrupted I/O instruction.
• I/O Instruction Restart RDI—the RDI of the interrupted I/O instruction.

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORT</td>
<td>RESERVED</td>
<td>A64</td>
<td>A32</td>
<td>A16</td>
<td>S63</td>
<td>S32</td>
<td>S16</td>
<td>S8</td>
<td>REP</td>
<td>STR</td>
<td>VAL</td>
<td>TYPE</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 10-6. I/O Instruction Restart Dword**

The fields in the I/O Instruction Restart DWORD are as follows:
• PORT—Intercepted I/O port
• A64—64-bit address
• A32—32-bit address
• A16—16-bit address
• SZ32—32-bit I/O port size
• SZ16—16-bit I/O port size
• SZ8—8-bit I/O port size
• REP—Repeated port access
• STR—String based port access (INS, OUTS)
• VAL—Valid (SMI was detected during an I/O instruction.)
• TYPE—Access type (0 = OUT instruction, 1 = IN instruction).

### 10.4 Leaving SMM

Software leaves SMM and returns to the interrupted program by executing the RSM instruction. RSM causes the processor to load the interrupted state from the SMRAM state-save area and then transfer control back to the interrupted program. RSM cannot be executed in any mode other than SMM, otherwise an invalid-opcode exception (#UD) occurs.

An RSM causes a processor shutdown if an invalid-state condition is found in the SMRAM state-save area. Only an external reset, external processor-initialization, or non-maskable external interrupt (NMI) can cause the processor to leave the shutdown state. The invalid SMRAM state-save-area conditions that can cause a processor shutdown during an RSM are:
• CR0.PE=0 and CR0.PG=1.
• CR0.CD=0 and CR0.NW=1.
• Certain reserved bits are set to 1, including:
- Any CR0 bit in the range 63:32 is set to 1.
- Any unsupported bit in CR3 is set to 1.
- Any unsupported bit in CR4 is set to 1.
- Any DR6 bit or DR7 bit in the range 63:32 is set to 1.
- Any unsupported bit in EFER is set to 1.

Invalid returns to long mode, including:
- EFER.LME=1, CR0.PG=1, and CR4.PAE=0.
- EFER.LME=1, CR0.PG=1, CR4.PAE=1, CS.L=1, and CS.D=1.

The SSM revision identifier is modified.

Some SMRAM state-save-area conditions are ignored, and the registers, or bits within the registers, are restored in a default manner by the processor. This avoids a processor shutdown when an invalid condition is stored in SMRAM. The default conditions restored by the processor are:

- The EFER.LMA register bit is set to the value obtained by logically ANDing the SMRAM values of EFER.LME, CR0.PG, and CR4.PAE.
- The RFLAGS.VM register bit is set to the value obtained by logically ANDing the SMRAM values of RFLAGS.VM, CR0.PE, and the inverse of EFER.LMA.
- The base values of FS, GS, GDTR, IDTR, LDTR, and TR are restored in canonical form. Those values are sign-extended to bit 63 using the most-significant implemented bit.
- Unimplemented segment-base bits in the CS, DS, ES, and SS registers are cleared to 0.

10.5 Multiprocessor Considerations

For multiprocessor operation, each logical processor must be given a separate SMBASE value so that the save-state areas do not overlap. For systems with fewer than 64 logical processors it is sufficient to stagger the SMBASE values by 512 bytes. Note that this also offsets the SMI entry point by the same amount for each processor. With 64 or more logical processors, the entry points will start to collide with the save-state areas. Staggering the SMBASE values by 1024 bytes results in 512-byte entry point areas interleaved with the 512-byte state-save areas, and so provides scaling beyond 63 logical processors.

Further details on multiprocessor aspects of SMM may be found in the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual for a given processor family.
11  SSE, MMX, and x87 Programming

This chapter describes the system-software implications of supporting applications that use the Streaming SIMD Extensions (SSE), MMX™, and x87 instructions. Throughout this chapter, these instructions are collectively referred to as *media and x87* (media/x87) instructions. A complete listing of the instructions that fall in this category—and the detailed operation of each instruction—can be found in volumes 4 and 5. Refer to Volume 1 for information on using these instructions in application software.

The SSE instruction set is comprised of the *legacy SSE* instruction set which includes the SSE1, SSE2, SSE3, SSSE3, SSE4A, SSE4.1, and SSE4.2 subsets and the *extended SSE* instruction set which includes the AVX, FMA4, and XOP subsets. Many of the extended SSE instructions support both 128-bit and 256-bit data types.

11.1  Overview of System-Software Considerations

Processor implementations can support different combinations of the SSE, MMX, and x87 instruction sets. Two sets of registers— independent of the general-purpose registers—support these instructions. The SSE instructions operate on the YMM/XMM registers, and the 64-bit media and x87-instructions operate on the aliased MMX/x87 registers. The SSE and x87 floating-point instruction sets have distinct status registers, control registers, exception vectors, and system-software control bits for managing the operating environment. System software that supports use of these instructions must be able to manage these resources properly including:

• Detecting support for the instruction set, and enabling any optional features, as necessary.
• Saving and restoring the processor media or x87 state.
• Execution of floating-point instructions (media or x87) can produce exceptions. System software must supply exception handlers for all unmasked floating-point exceptions.

11.2  Determining Media and x87 Feature Support

Support for the architecturally defined subsets within the media and x87 instructions is implementation dependent. System software executes the CPUID instruction to determine whether a processor implements any of these features (see Section 3.3, “Processor Feature Identification,” on page 64 for more information on using the CPUID instruction). After CPUID is executed feature support can be determined by examining specific bit fields returned in the EAX, ECX, and EDX registers.

The following table summarizes the architecturally defined SSE subsets and state management instructions and gives the feature bits returned by the CPUID function. If the indicated bit is set, the feature is supported by the processor.
Some instructions may be listed in more than one subset. If software attempts to execute an instruction belonging to an unsupported instruction subset, an invalid-opcode exception (#UD) occurs. Refer to Appendix D, “Instruction Subsets and CPUID Feature Flags” in Volume 3 for specific information.
11.3 Enabling SSE Instructions

Use of the 256-bit and 128-bit media instructions by application software requires system software support. System software must determine which SSE subsets are supported, enable those that are to be used, and supply code to handle the various exceptions that may occur during the execution of these instructions. The legacy SSE instructions and the extended SSE instructions often require unique exception handling.

11.3.1 Enabling Legacy SSE Instruction Execution

When legacy SSE instructions are supported, system software must set CR4.OSFXSR to let the processor know that the software supports the FXSAVE/FXRSTOR instructions. When the processor detects CR4.OSFXSR = 1, it allows execution of the legacy SSE instructions. If system software does not set CR4.OSFXSR, any attempt to execute these instructions causes an invalid-opcode exception (#UD). System software must also clear the CR0.EM (emulate coprocessor) bit to 0, otherwise an attempt to execute a legacy SSE instruction causes a #UD exception. An attempt to execute either FXSAVE or FXRSTOR when CR0.EM is set results in a #NM exception.

System software should also set the CR0.MP (monitor coprocessor) bit to 1. When CR0.EM=0 and CR0.MP=1, all media instructions, x87 instructions, and the FWAIT/WAIT instructions cause a device-not-available exception (#NM) when the CR0.TS bit is set. System software can use the #NM exception to perform lazy context switching, saving and restoring media and x87 state only when necessary after a task switch. See “CR0 Register” on page 42 for more information.

11.3.2 Enabling Extended SSE Instruction Execution

After the steps specified above are completed to enable legacy SSE instruction execution, additional steps are required to enable the extended SSE instructions and state management. System software must carry out the following process:

- Confirm that the hardware supports the XSAVE, XRSTOR, XSETBV, and XGETBV instructions and the XCR0 register (XFEATURE_ENABLED_MASK) by executing the CPUID instruction function 0000_0001h. If CPUID Fn0000_0001_ECX[XSAVE] is set, hardware support is verified.
- Optionally confirm hardware support of the XSAVEOPT instruction by executing CPUID function 0000_000Dh, sub-function 1 (ECX = 1). If CPUID Fn0000_000D_EAX_x1[XSAVEOPT] is set, the processor supports the XSAVEOPT instruction. XSAVEOPT is a performance optimized version of XSAVE.
- Confirm that hardware supports the extended SSE instructions by verifying XFeatureSupportedMask[2:0] = 111b. XFeatureSupportedMask is accessed via the CPUID instruction function 0000_000Dh, sub-function 0 (ECX = 0). XFeatureSupportedMask[31:0] is returned in the EAX register.

If CPUID Fn0000_000D_EAX_x0[2:0] = 111b, hardware supports x87, legacy SSE, and extended SSE instructions. Bit 0 of EAX signifies x87 floating-point and MMX support, bit 1 signifies legacy SSE support, and bit 2 signifies extended SSE support. Support for both x87 and legacy SSE instructions are required for processors that support the extended SSE instructions.
• Set CR4[OSXSAVE] (bit 18) to enable the use of the XSETBV and XGETBV instructions. 
  XSETBV is a privileged instruction that writes the XCRn registers. XCR0 is the 
  XFEATURE_ENABLED_MASK used to manage media and x87 processor state using the 
  XSAVE, XSAVEOPT, and XRSTOR instructions.

• Enable the x87/MMX, legacy SSE, and extended SSE instructions and processor state 
  management by setting the x87, SSE, and YMM bits of XCR0 
  (XFEATURE_ENABLED_MASK). This is done via the privileged instruction XSETBV. 
  Enabling extended SSE capabilities without enabling legacy SSE capabilities is not allowed. The 
  x87 flag (bit 0) of the XFEATURE_ENABLED_MASK must be set when writing XCR0.

• Determine the XSAVE/XRSTOR memory save area size requirement. The field 
  XFeatureEnabledSizeMax specifies the size requirement in bytes based on the currently enabled 
  extended features and is returned in the EBX register after execution of CPUID Function 
  0000_000Dh, sub-function 0 (ECX = 0).

• Allocate the save/restore area based on the information obtained in the previous step.

For a detailed description of the XSETBV and XGETBV instructions, see individual instruction 
reference pages in Volume 4. See the section entitled “XFEATURE_ENABLED_MASK” in Volume 4 
for details on the field definitions for XFEATURE_ENABLED_MASK.

For more information on using the CPUID instruction to obtain processor feature information, see 
Section 3.3, “Processor Feature Identification,” on page 64.

11.3.3 SIMD Floating-Point Exception Handling

System software must supply an exception handler if unmasked SSE floating-point exceptions are 
allowed to occur. When an unmasked exception is detected, the processor transfers control to the 
SIMD floating-point exception (#XF) handler provided by the operating system. System software 
must let the processor know that the #XF handler is available by setting CR4.OSXMMEXCPT to 1. If 
this bit is set to 1, the processor transfers control to the #XF handler when it detects an unmasked 
exception, otherwise a #UD exception occurs. When the processor detects a masked exception, it 
handles it in a default manner regardless of the CR4.OSXMMEXCPT value.

11.4 Media and x87 Processor State

The media and x87 processor state includes the contents of the registers used by SSE, MMX, and x87 
instructions. System software that supports such applications must be capable of saving and restoring 
these registers.

11.4.1 SSE Execution Unit State

Figure 11-1 shows the registers whose contents are affected by execution of SSE instructions. These 
include:

• YMM/XMM0–15—Sixteen 256-bit/128-bit SSE registers. In legacy and compatibility modes, 
  software access is limited to the first eight registers.
• MXCSR—The 32-bit Media eXtensions Control and Status Register.

All of these registers are visible to application software. Refer to “Streaming SIMD Extensions Media and Scientific Programming” in Volume 1 for more information on these registers.

![Figure 11-1. SSE Execution Unit State](513-314 ymm.png)

### 11.4.2 MMX Execution Unit State

Figure 11-2 on page 314 shows the register contents that are affected by execution of 64-bit media instructions. These registers include:

- *mmx0–mmx7*—Eight 64-bit media registers.
- *FSW*—Two fields (TOP and ES) in the 16-bit x87 status word register.
• *FTW*—The 16-bit x87 tag word.

![MMX Registers Diagram](v2_MMX_regs.eps)

![x87 Status Word Diagram](v2_MMX_regs.eps)

**Figure 11-2. MMX Execution Unit State**

The 64-bit media instructions and x87 floating-point instructions share the same physical data registers. Figure 11-2 shows how the 64-bit registers (MMX0–MMX7) are aliased onto the low 64 bits of the 80-bit x87 floating-point physical data registers (FPR0–FPR7). Refer to “64-Bit Media Programming” in Volume 1 for more information on these registers.

Of the registers shown in Figure 11-2, only the eight 64-bit MMX registers are visible to 64-bit media application software. The processor maintains the contents of the two fields of the x87 status word—top-of-stack-pointer (TOP) and exception summary (ES)—and the 16-bit x87 tag word during execution of 64-bit media instructions, as described in “Actions Taken on Executing 64-Bit Media Instructions” in Volume 1.

64-bit media instructions do not generate x87 floating-point exceptions, nor do they set any status flags. However, 64-bit media instructions can trigger an unmasked floating-point exception caused by a previously executed x87 instruction. 64-bit media instructions do this by reading the x87 FSW.ES bit to determine whether such an exception is pending.

**11.4.3 x87 Execution Unit State**

Figure 11-3 on page 316 shows the registers whose contents are affected by execution of x87 floating-point instructions. These registers include:
- `fpr0–fpr7`—Eight 80-bit floating-point physical registers.
- `FCW`—The 16-bit x87 control word register.
- `FSW`—The 16-bit x87 status word register.
- `FTW`—The 16-bit x87 tag word.
- `Last x87 Instruction Pointer`—This value is a pointer (32-bit, 48-bit, or 64-bit, depending on effective operand size and mode) to the last non-control x87 floating-point instruction executed.
- `Last x87 Data Pointer`—The pointer (32-bit, 48-bit, or 64-bit, depending on effective operand size and mode) to the data operand referenced by the last non-control x87 floating-point instruction executed, if that instruction referenced memory; if it did not, then this value is implementation dependent.
- `Last x87 Opcode`—An 11-bit permutation of the instruction opcode from the last non-control x87 floating-point instruction executed.

Of the registers shown in Figure 11-3 on page 316, only `FPR0–FPR7`, `FCW`, and `FSW` are directly updated by x87 application software. The processor maintains the contents of the `FTW`, instruction and data pointers, and opcode registers during execution of x87 instructions. Refer to “Registers” in Volume 1 for more information on these registers.

The 11-bit instruction opcode register holds a permutation of the two-byte instruction opcode from the last non-control x87 instruction executed by the processor. (For a definition of non-control x87 instruction, see “Control” in Chapter 6 of Volume 1.) The opcode field is formed as follows:

- Opcode Register Field[10:8] = First x87 opcode byte[2:0].
- Opcode Register Field[7:0] = Second x87 opcode byte[7:0].

For example, the x87 opcode D9 F8h is stored in the opcode register as 001_1111_1000b. The low-order three bits of the first opcode byte, D9h (1101_1001b), are stored in opcode-register bits 10:8. The second opcode byte, F8h (1111_1000b), is stored in bits 7:0 of the opcode register. The high-order five bits of the first opcode byte (1101_1b) are not needed because they are identical for all x87 instructions.
11.4.4 Saving Media and x87 Execution Unit State

In most cases, operating systems, exception handlers, and device drivers should save and restore the media and/or x87 processor state between task switches or other interventions in the execution of 128-bit, 64-bit, or x87 procedures. Application programs are also free to save and restore state at any time.

In general, system software should use the FXSAVE and FXRSTOR instructions to save and restore the entire media and x87 processor state. The FSAVE/FNSAVE and FRSTOR instructions can be used for saving and restoring the x87 state. Because the 64-bit media registers are physically aliased onto the x87 registers, the FSAVE/FNSAVE and FRSTOR instructions can also be used to save and restore the 64-bit media state. However, FSAVE/FNSAVE and FRSTOR do not save or restore the 128-bit media state.
FSAVE/FNSAVE and FRSTOR Instructions. The FSAVE/FNSAVE and FRSTOR instructions save and restore the entire register state for 64-bit media instructions and x87 floating-point instructions. The FSAVE instruction stores the register state, but only after handling any pending unmasked-x87 floating-point exceptions. The FNSAVE instruction stores the register state but skips the reporting and handling of these exceptions. The state of all MMX/FPR registers is saved, as well as all other x87 state (the control word register, status word register, tag word, instruction pointer, data pointer, and last opcode). After saving this state, the tag state for all MMX/FPR registers is changed to empty and is thus available for a new procedure.

Starting on page 318, Figure 11-4 through Figure 11-7 show the memory formats used by the FSAVE/FNSAVE and FRSTOR instructions when storing the x87 state in various processor modes and using various effective-operand sizes. This state includes:

- **x87 Data Registers**
  - FPR0–FPR7 80-bit physical data registers.

- **x87 Environment**
  - FCW: x87 control word register
  - FSW: x87 status word register
  - FTW: x87 tag word
  - Last x87 instruction pointer
  - Last x87 data pointer
  - Last x87 opcode

The eight data registers are stored in the 80 bytes following the environment information. Instead of storing these registers in their physical order (FPR0–FPR7), the processor stores the registers in their stack order, ST(0)–ST(7), beginning with the top-of-stack, ST(0).
### Figure 11-4. FSAVE/FNSAVE Image (32-Bit, Protected Mode)

<table>
<thead>
<tr>
<th>Bit Offset</th>
<th>Byte Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ST(7)[79:48]</td>
<td>+68h</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>ST(1)[15:0]</td>
<td>ST(0)[79:64]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>ST(0)[63:32]</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>ST(0)[31:0]</td>
<td>+1Ch</td>
</tr>
<tr>
<td>Reserved, IGN</td>
<td>Data DS Selector[15:0]</td>
</tr>
<tr>
<td>Data Offset[31:0]</td>
<td></td>
</tr>
<tr>
<td>00000b</td>
<td>Instruction Opcode[10:0]</td>
</tr>
<tr>
<td>Instruction Offset[31:0]</td>
<td></td>
</tr>
<tr>
<td>Reserved, IGN</td>
<td>x87 Tag Word (FTW)</td>
</tr>
<tr>
<td>Reserved, IGN</td>
<td>x87 Status Word (FSW)</td>
</tr>
<tr>
<td>Reserved, IGN</td>
<td>x87 Control Word (FCW)</td>
</tr>
</tbody>
</table>
### Figure 11-5. FSAVE/FNSAVE Image (32-Bit, Real/Virtual-8086 Modes)

<table>
<thead>
<tr>
<th>Bit Offset</th>
<th>Byte Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 16 15 0</td>
<td></td>
</tr>
</tbody>
</table>

- **ST(7)[79:48]**: Bit Offset +68h
- **...**: Bit Offset...
- **ST(1)[15:0]**: Bit Offset...
- **ST(0)[79:64]**: Bit Offset...
- **ST(0)[63:32]**: Bit Offset...
- **ST(0)[31:0]**: Bit Offset +1Ch

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b</td>
<td>0000 0000 0000b</td>
<td>0</td>
<td>0</td>
<td>0000b</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Reserved, IGN</td>
<td>Reserved, IGN</td>
<td>Reserved, IGN</td>
<td>Reserved, IGN</td>
<td>Reserved, IGN</td>
<td>Reserved, IGN</td>
<td>Reserved, IGN</td>
</tr>
</tbody>
</table>

- **Reserved, IGN**: Bit Offset +18h
- **Data Offset[15:0]**: Bit Offset +14h
- **Instruction Offset[15:0]**: Bit Offset +10h
- **Instruction Opcode[10:0]**: Bit Offset +0Ch
- **x87 Tag Word (FTW)**: Bit Offset +08h
- **x87 Status Word (FSW)**: Bit Offset +04h
- **x87 Control Word (FCW)**: Bit Offset +00h
<table>
<thead>
<tr>
<th>Bit Offset</th>
<th>Byte Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>16 15 0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td><em>Not Part of x87 State</em></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td><strong>ST(7)[79:64]</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>+5Ch</strong></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td><strong>ST(0)[79:48]</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>+14h</strong></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td><strong>ST(0)[47:16]</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>+10h</strong></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td><strong>ST(0)[15:0]</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Data DS Selector[15:0]</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>+0Ch</strong></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td><strong>Data Offset[15:0]</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Instruction CS Selector[15:0]</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>+08h</strong></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td><strong>Instruction Offset[15:0]</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>x87 Tag Word (FTW)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>+04h</strong></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td><strong>x87 Status Word (FSW)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>x87 Control Word (FCW)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>+00h</strong></td>
</tr>
</tbody>
</table>

Figure 11-6.  FSAVE/FNSAVE Image (16-Bit, Protected Mode)
The `FLDENV/FNLDENV` and `FSTENV` instructions load and store only the x87 floating-point environment. These instructions, unlike the `FSAVE/FNSAVE` and `FRSTOR` instructions, do not save or restore the x87 data registers. The `FLDENV/FSTENV` instructions do not save the full 64-bit data and instruction pointers. 64-bit applications should use `FXSAVE/FXRSTOR`, rather than `FLDENV/FSTENV`. The format of the saved x87 environment images for protected mode and real/virtual mode are the same as those of the first 14-bytes of the `FSAVE/FNSAVE` images for 16-bit operands or 32/64-bit operands, respectively. See Figure 11-4 on page 318, Figure 11-5 on page 319, Figure 11-6 on page 320, and Figure 11-7.

The `FXSAVE` and `FXRSTOR` instructions save and restore the entire 128-bit media, 64-bit media, and x87 state. These instructions usually execute faster than `FSAVE/FNSAVE` and `FRSTOR` because they do not normally save and restore the x87 exception pointers (last-instruction pointer, last data-operand pointer, and last opcode). The only case in which they do save the exception pointers is the relatively rare case in which the exception-summary bit in

---

**Table 11-7. FSAVE/FNSAVE Image (16-Bit, Real/Virtual-8086 Modes)**

<table>
<thead>
<tr>
<th>Bit Offset</th>
<th>Byte Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-16-15-0</td>
<td></td>
</tr>
<tr>
<td>ST(7)</td>
<td>+5Ch</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>ST(0)[79:48]</td>
<td>+14h</td>
</tr>
<tr>
<td>ST(0)[47:16]</td>
<td>+10h</td>
</tr>
<tr>
<td>ST(0)[15:0]</td>
<td>+0Ch</td>
</tr>
<tr>
<td>Data [19:16]</td>
<td>0000 0000 0000b</td>
</tr>
<tr>
<td>Data Offset [15:0]</td>
<td>Instruction [19:16]</td>
</tr>
<tr>
<td>Instruction Offset [15:0]</td>
<td>x87 Tag Word (FTW)</td>
</tr>
<tr>
<td>x87 Status Word (FSW)</td>
<td>x87 Control Word (FCW)</td>
</tr>
</tbody>
</table>
the x87 status word (FSW.ES) is set to 1, indicating that an unmasked exception has occurred. The FXSAVE and FXRSTOR memory format contains fields for storing these values.

Unlike FSAVE and FNSAVE, the FXSAVE instruction does not alter the x87 tag word. Therefore, the contents of the shared 64-bit MMX and 80-bit FPR registers can remain valid after an FXSAVE instruction (or any other value the tag bits indicated before the save). Also, FXSAVE (like FNSAVE) does not check for pending unmasked-x87 floating-point exceptions.

Figure 11-9 on page 329 shows the memory format of the media x87 state in long mode. If a 32-bit operand size is used in 64-bit mode, the memory format is the same, except that RIP and RDS are stored as sel:offset pointers, as shown in Figure 11-10 on page 330.

For more information on the FXSAVE and FXRSTOR instructions, see individual instruction listings in "64-Bit Media Instruction Reference" of Volume 5.
11.5 XSAVE/XRSTOR Instructions

The XSAVE, XSAVEOPT, XRSTOR, XGETBV, and XSETBV instructions and associated data structures extend the FXSAVE/FXRSTOR memory image used to manage processor states and provide additional functionality. These instructions do not obviate the FXSAVE/FXRSTOR instructions. For more information about FXSAVE/FXRSTOR, see “FXSAVE and FXRSTOR Instructions” in Volume 2. For detailed descriptions of FXSAVE and FXRSTOR, see individual instruction listings in AMD64 Architecture Programmer’s Manual “Volume 5: 64-Bit Media and x87 Floating-Point Instructions.”

The CPUID instruction is used to identify features supported in processor hardware. Extended control registers are used to enable and disable the handling of processor states associated with supported hardware features and to communicate to an application whether an operating system supports a particular feature that has a processor state specific to it.

11.5.1 CPUID Enhancements

• CPUID Fn0000_00001_ECX[XSAVE] indicates that the processor supports XSAVE/XRSTOR instructions and at least one XCR.
• CPUID Fn0000_00001_ECX[OSXSAVE] indicates whether the operating system has enabled extensible state management and supports processor extended state management.
• CPUID Fn0000_0000D enumerates processor states (including legacy x87 FPU states, SSE states, and processor extended states), the offset, and the size of the save area for each processor extended state. Sub-functions (ECX > 0) provide details concerning features and support of processor states enumerated in the root function.

11.5.2 XFEATURE_ENABLED_MASK

XFEATURE_ENABLED_MASK is set up by privileged software to enable the saving and restoring of extended processor architectural state information supported by a specific processor. Clearing defined bit fields in this mask inhibits the XSAVE instruction from saving (and XRSTOR from restoring) this state information.

XFEATURE_ENABLED_MASK is addressed as XCR0 in the extended control register space and is accessed via the XSETBV and XGETBV instructions.

XFEATURE_ENABLED_MASK is defined as follows:

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>61</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>LWP</td>
<td>Reserved</td>
<td>MPK</td>
<td>Reserved</td>
<td>YMM</td>
<td>SSE</td>
<td>x87</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hardware initializes XCR0 to 0000_0000_0000_0001h. On writing this register, software must insure that XCR0[63:3] is clear, XCR0[0] is set, and that XCR0[2:1] is not equal to 10b. An attempt to write data that violates these rules results in a #GP.

### 11.5.3 Extended Save Area

The XSAVE/XRSTOR save area extends the legacy 512-byte FXSAVE/FXRSTOR memory image to provide a compatible register state management environment as well as an upward migration path. The save area is architecturally defined to be extendable and enumerated by the sub-functions of CPUID Fn 0000_000Dh. Figure 11-2 shows the format of the XSAVE/XRSTOR area.

#### Table 11-2. Extended Save Area Format

<table>
<thead>
<tr>
<th>Save Area</th>
<th>Offset (Byte)</th>
<th>Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPU/SSE Save Area</td>
<td>0</td>
<td>512</td>
</tr>
<tr>
<td>Header</td>
<td>512</td>
<td>64</td>
</tr>
<tr>
<td>Reserved, (Ext_Save_Area_2)</td>
<td>CPUID Fn 0000_000D_EBX_x02</td>
<td>CPUID Fn 0000_000D_EAX_x02</td>
</tr>
<tr>
<td>Reserved, (Ext_Save_Area_3)</td>
<td>CPUID Fn 0000_000D_EBX_x03</td>
<td>CPUID Fn 0000_000D_EAX_x03</td>
</tr>
<tr>
<td>Reserved, (Ext_Save_Area_4)</td>
<td>CPUID Fn 0000_000D_EBX_x04</td>
<td>CPUID Fn 0000_000D_EAX_x04</td>
</tr>
<tr>
<td>Reserved, (…)</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

*Note: Bytes 464–511 are available for software use. XRSTOR ignores bytes 464–511 of an XSAVE image.*

The register fields of the first 512 bytes of the XSAVE/XRSTOR area are the same as those of the FXSAVE/FXRSTOR area, but the 512-byte area is organized as x87 FPU states, MXCSR (including MXCSR_MASK), and XMM registers. The layout of the save area is fixed and may contain non-
contiguous individual save areas because a processor does not support certain extended states or because system software does not support certain processor extended states. The save area is not compacted when features are not saved or are not supported by the processor or by system software.

For more information on using the CPUID instruction to obtain processor implementation information, see Section 3.3, “Processor Feature Identification,” on page 64.

11.5.4 Instruction Functions

CR4.OSXSAVE and XCR0 can be read at all privilege levels but written only at ring 0.

- XGETBV reads XCR0.
- XSETBV writes XCR0, ring 0 only.
- XRSTOR restores states specified by bitwise AND of a mask operand in EDX:EAX with XCR0.
- XSAVE (and XSAVEOPT) saves states specified by bitwise AND of a mask operand in EDX:EAX with XCR0.

11.5.5 YMM States and Supported Operating Modes

Extended instructions operate on YMM states by means of extended (XOP/VEX) prefix encoding. When a processor supports YMM states, the states exist in all operating modes, but interfaces to access the YMM states may vary by mode. Processor support for extended prefix encoding is independent of processor support of YMM states.

Instructions that use extended prefix encoding are generally supported in long and protected modes, but are not supported in real or virtual 8086 modes, or when entering SMM mode. Bits 255:128 of the YMM register state are maintained across transitions into and out of these modes. The XSAVE/XRSTOR instructions function in all operating modes; XRSTOR can modify YMM register state in any operating mode, using state information from the XSAVE/XRSTOR area.

11.5.6 Extended SSE Execution State Management

Operating system software must use the XSAVE/XRSTOR instructions for extended SSE execution state management. XSAVEOPT, a performance optimized version of XSAVE, may be used instead of XSAVE once the XSAVE/XRSTOR save area is initialized. In the following discussion XSAVEOPT may be substituted for the instruction XSAVE. The instructions also provide an interface to manage XMM/MXCSR states and x87 FPU states in conjunction with processor extended states. An operating system must enable extended SSE execution state management prior to the execution of extended SSE instructions. Attempting to execute an extended SSE instruction without enabling execution state management causes a #UD exception.

11.5.6.1 Enabling Extended SSE Instruction Execution

To enable extended SSE instruction execution and state management, system software must carry out the following process:
• Confirm that the hardware supports the XSAVE, XRSTOR, XSETBV, and XGETBV instructions and the XCR0 register (XFEATURE_ENABLED_MASK) by executing the CPUID instruction function 0000_0001h. If CPUID Fn0000_0001_ECX[XSAVE] is set, hardware support is verified.

• Optionally confirm hardware support of the XSAVEOPT instruction by executing CPUID function 0000_000Dh, sub-function 1 (ECX = 1). If CPUID Fn0000_000D_EAX_x1[XSAVEOPT] is set, the processor supports the XSAVEOPT instruction. XSAVEOPT is a performance optimized version of XSAVE. (SDCR-3580)

• Confirm that hardware supports the extended SSE instructions by verifying XFeatureSupportedMask[2:0] = 111b. XFeatureSupportedMask is accessed via the CPUID instruction function 0000_000Dh, sub-function 0 (ECX = 0).

  If CPUID Fn0000_000D_EAX_x0[2:0] = 111b, hardware supports x87, legacy SSE, and extended SSE instructions. Bit 0 of EAX signifies x87 floating-point and MMX support, bit 1 signifies legacy SSE support, and bit 2 signifies extended SSE support. Support for both x87 and legacy SSE instructions are required for processors that support the extended SSE instructions.

• Set CR4[OSXSA VE] (bit 18) to enable the use of the XSETBV and XGETBV instructions. XSETBV is a privileged instruction that writes the XCRn registers. XCR0 is the XFEATURE_ENABLED_MASK used to manage media and x87 processor state using the XSAVE, XSAVEOPT, and XRSTOR instructions.

• Enable the x87/MMX, legacy SSE, and extended SSE instructions and processor state management by setting the x87, SSE, and YMM bits of XCR0 (XFEATURE_ENABLED_MASK). Enabling extended SSE capabilities without enabling legacy SSE capabilities is not allowed. The x87 flag (bit 0) of the XFEATURE_ENABLED_MASK must be set when writing XCR0.

• Determine the XSAVE/XRSTOR memory save area size requirement. The field XFeatureEnabledSizeMax specifies the size requirement in bytes based on the currently enabled extended features and is returned in the EAX register after execution of CPUID Function 0000_000Dh, sub-function 0 (ECX = 0).

• Allocate the save/restore area based on the information obtained in the previous step.

For more information on the XSETBV and XGETBV instructions, see individual instruction descriptions in Volume 4. XFEATURE_ENABLED_MASK fields are defined in Section 11.5.2 above.

For more information on using the CPUID instruction to obtain processor implementation information, see Section 3.3, “Processor Feature Identification,” on page 64.
11.5.7 Saving Processor State

The XSTATE header starts at byte offset 512 in the save area. XSTATE_BV is the first 64-bit field in the header. The order of bit vectors in XSTATE_BV matches the order of bit vectors in XCR0. The XSAVE instruction sets bits in the XSTATE_BV vector field when it writes the corresponding processor extended state to a save area in memory. XSAVE modifies only bits for processor states specified by bitwise AND of the XSAVE bit mask operand in EDX:EAX with XCR0. If software modifies the save area image of a particular processor state component directly, it must also set the corresponding bit of XSTATE_BV. If the bit is not set, directly modified state information in a save area image may be ignored by XRSTOR.

XSAVEOPT, a performance optimized version of the XSAVE instruction, may be used (if supported) in lieu of the XSAVE instruction once the XSAVE/XRSTOR save area has been initialized via the execution of the XSAVE instruction.

11.5.8 Restoring Processor State

When XRSTOR is executed, processor state components are updated only if the corresponding bits in the mask operand (EDX:EAX) and XCR0 are both set. For each updated component, when the corresponding bit in the XSTATE_BV field in the save area header is set, the component is loaded from the save area in memory. When the XSTATE_BV bit is cleared, the state is set to the hardware-specified initial values shown in Table 11-3.

<table>
<thead>
<tr>
<th>Component</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x87</td>
<td>FCW = 037Fh</td>
</tr>
<tr>
<td></td>
<td>FSW = 0000h</td>
</tr>
<tr>
<td></td>
<td>Empty/Full = 00h (FTW = FFFFh)</td>
</tr>
<tr>
<td></td>
<td>x87 Error Pointers = 0</td>
</tr>
<tr>
<td></td>
<td>ST0 - ST7 = 0</td>
</tr>
<tr>
<td>XMM</td>
<td>XMM0 - XMM15 = 0, if 64-bit mode</td>
</tr>
<tr>
<td></td>
<td>XMM0 - XMM7 = 0, if !64-bit mode</td>
</tr>
<tr>
<td>YMM_HI</td>
<td>YMM_HI0 - Y MM_HI15 = 0, if 64-bit mode</td>
</tr>
<tr>
<td></td>
<td>YMM_HI0-YMM_HI7 = 0, if !64-bit mode</td>
</tr>
<tr>
<td>LWP</td>
<td>LWP disabled</td>
</tr>
</tbody>
</table>

11.5.9 MXCSR State Management

The MXCSR has no hardware-specified initial state; it is read from the save area in memory whenever either XMM or YMM_HI are updated.

11.5.10 Mode-Specific XSAVE/XRSTOR State Management

Some state is conditionally saved or updated, depending on processor state:
• On processors where CPUID Fn8000_0008_EBX[2] is 0, the x87 error pointers are not saved or restored if the state saved or loaded from memory doesn't have a pending #MF. On processors where CPUID Fn8000_0008_EBX[2] is 1, the error pointers are always restored from the save area (and if in 64-bit mode the CS and DS portions of the error pointer registers are zeroed), and the error pointer fields in the save area are zeroed if there is no pending #MF, else the error pointer offset registers are written to the save area.

• XMM8–XMM15 are not saved or restored in non 64-bit mode.

• YMM_HI8–YMM_HI15 are not saved or restored in non 64-bit mode.
**Figure 11-9. FXSAVE and FXRSTOR Image (64-bit Mode)**

<table>
<thead>
<tr>
<th>F E D C B A 9 8 7 6 5 4 3 2 1 0</th>
<th>Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved, IGN</td>
<td>+1F0h</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Reserved, IGN</td>
<td>+1A0h</td>
</tr>
<tr>
<td>XMM15</td>
<td>+190h</td>
</tr>
<tr>
<td>XMM14</td>
<td>+180h</td>
</tr>
<tr>
<td>XMM13</td>
<td>+170h</td>
</tr>
<tr>
<td>XMM12</td>
<td>+160h</td>
</tr>
<tr>
<td>XMM11</td>
<td>+150h</td>
</tr>
<tr>
<td>XMM10</td>
<td>+140h</td>
</tr>
<tr>
<td>XMM9</td>
<td>+130h</td>
</tr>
<tr>
<td>XMM8</td>
<td>+120h</td>
</tr>
<tr>
<td>XMM7</td>
<td>+110h</td>
</tr>
<tr>
<td>XMM6</td>
<td>+100h</td>
</tr>
<tr>
<td>XMM5</td>
<td>+F0h</td>
</tr>
<tr>
<td>XMM4</td>
<td>+E0h</td>
</tr>
<tr>
<td>XMM3</td>
<td>+D0h</td>
</tr>
<tr>
<td>XMM2</td>
<td>+C0h</td>
</tr>
<tr>
<td>XMM1</td>
<td>+B0h</td>
</tr>
<tr>
<td>XMM0</td>
<td>+A0h</td>
</tr>
</tbody>
</table>

| Reserved, IGN                 | ST(7)  |
| Reserved, IGN                 | ST(6)  |
| Reserved, IGN                 | ST(5)  |
| Reserved, IGN                 | ST(4)  |
| Reserved, IGN                 | ST(3)  |
| Reserved, IGN                 | ST(2)  |
| Reserved, IGN                 | ST(1)  |
| Reserved, IGN                 | ST(0)  |

<table>
<thead>
<tr>
<th>MXCSR_MASK</th>
<th>MXCSR</th>
<th>RDP¹</th>
<th>RIP¹</th>
<th>FOP</th>
<th>0</th>
<th>FTW</th>
<th>FSW</th>
<th>FCW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Stored as sel:offset if operand size is 32 bits. 32bit sel:offset format of the pointers is shown in figure 11-10.
Software can read and write all fields within the FXSAVE and FXRSTOR memory image. These fields include:

- **FCW**—Bytes 01h–00h. x87 control word.
- **FSW**—Bytes 03h–02h. x87 status word.
- **FTW**—Byte 04h. x87 tag word. See “FXSAVE Format for x87 Tag Word” on page 331 for additional information on the FTW format saved by the FXSAVE instruction.
- (Byte 05h contains the value 00h.)
- **FOP**—Bytes 07h–06h. last x87 opcode.
- **Last x87 Instruction Pointer**—A pointer to the last non-control x87 floating-point instruction executed by the processor:

```
<table>
<thead>
<tr>
<th>Byte</th>
<th>Reserved, IGN</th>
<th>...</th>
<th>Reserved, IGN</th>
<th>XMM7</th>
<th>...</th>
<th>XMM0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>110h</td>
<td></td>
<td>100h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01h</td>
<td>111h</td>
<td></td>
<td>101h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02h</td>
<td>112h</td>
<td></td>
<td>102h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>03h</td>
<td>113h</td>
<td></td>
<td>103h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>04h</td>
<td>114h</td>
<td></td>
<td>104h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>05h</td>
<td>115h</td>
<td></td>
<td>105h</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Figure 11-10. FXSAVE and FXRSTOR Image (Non-64-bit Mode)**
- **RIP (64-bit format)**—Bytes 0Fh–08h. 64-bit offset into the code segment (used without a CS selector).
- **EIP (32-bit format)**—Bytes 0Bh–08h. 32-bit offset into the code segment.
- **CS (32-bit format)**—Bytes 0Dh–0Ch. Segment selector portion of the pointer.

**Last x87 Data Pointer**—If the last non-control x87 floating point instruction referenced memory, this value is a pointer to the data operand referenced by the last non-control x87 floating-point instruction executed by the processor:
- **RDP (64-bit format)**—Bytes 17h–10h. 64-bit offset into the data segment (used without a DS selector).
- **DP (32-bit format)**—Bytes 13h–10h. 32-bit offset into the data segment.
- **DS (32-bit format)**—Bytes 15h–14h. Segment selector portion of the pointer.

If the last non-control x87 instruction did not reference memory, then the value in the pointer is implementation dependent.

**MXCSR**—Bytes 1Bh–18h. 128-bit media-instruction control and status register. This register is saved only if CR4.OSFXSR is set to 1.

**MXCSR_MASK**—Bytes 1Fh–1Ch. Set bits in MXCSR_MASK indicate supported feature bits in MXCSR. For example, if bit 6 (the DAZ bit) in the returned MXCSR_MASK field is set to 1, the DAZ mode and the DAZ flag in MXCSR are supported. Cleared bits in MXCSR_MASK indicate reserved bits in MXCSR. If software attempts to set a reserved bit in the MXCSR register, a #GP exception will occur. To avoid this exception, after software clears the FXSAVE memory image and executes the FXSAVE instruction, software should use the value returned by the processor in the MXCSR_MASK field when writing a value to the MXCSR register, as follows:

- **MXCSR_MASK = 0**: If the processor writes a zero value into the MXCSR_MASK field, the denormals-are-zeros (DAZ) mode and the DAZ flag in MXCSR are not supported. Software should use the default mask value, 0000_FFBFh (bit 6, the DAZ bit, and bits 31:16 cleared to 0), to mask any value it writes to the MXCSR register to ensure that all reserved bits in MXCSR are written with 0, thus avoiding a #GP exception.
- **MXCSR_MASK ≠ 0**: If the processor writes a non-zero value into the MXCSR_MASK field, software should AND this value with any value it writes to the MXCSR register.

**MMXn/FPRn**—Bytes 9Fh–20h. Shared 64-bit media and x87 floating-point registers. As in the case of the x87 FSAVE instruction, these registers are stored in stack order ST(0)–ST(7). The upper six bytes in the memory image for each register are reserved.

**XMMn**—Bytes 11Fh–A0h. 128-bit media registers. These registers are saved only if CR4.OSFXSR is set to 1.

**FXSAVE Format for x87 Tag Word.** Rather than saving the entire x87 tag word, FXSAVE saves a single-byte encoded version. FXSAVE encodes each of the eight two-bit fields in the x87 tag word as follows:

- Two-bit values of 00, 01, and 10 are encoded as a 1, indicating the corresponding x87 FPRn register holds a value.
• A two-bit value of 11 is encoded as a 0, indicating the corresponding x87 FPR\textsubscript{n} is empty.

For example, assume an FSAVE instruction saves an x87 tag word with the value 83F1\text{h}. This tag-word value describes the x87 FPR\textsubscript{n} contents as follows:

<table>
<thead>
<tr>
<th>x87 Register</th>
<th>FPR7</th>
<th>FPR6</th>
<th>FPR5</th>
<th>FPR4</th>
<th>FPR3</th>
<th>FPR2</th>
<th>FPR1</th>
<th>FPR0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag Word Value (hex)</td>
<td>8</td>
<td>3</td>
<td>F</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag Value (binary)</td>
<td>10</td>
<td>00</td>
<td>00</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>Meaning</td>
<td>Special</td>
<td>Valid</td>
<td>Valid</td>
<td>Empty</td>
<td>Empty</td>
<td>Empty</td>
<td>Valid</td>
<td>Zero</td>
</tr>
</tbody>
</table>

When an FXSAVE is used to write the x87 tag word to memory, it encodes the value as E3\text{h}. This encoded version describes the x87 FPR\textsubscript{n} contents as follows:

<table>
<thead>
<tr>
<th>x87 Register</th>
<th>FPR7</th>
<th>FPR6</th>
<th>FPR5</th>
<th>FPR4</th>
<th>FPR3</th>
<th>FPR2</th>
<th>FPR1</th>
<th>FPR0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoded Tag Byte (hex)</td>
<td>E</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag Value (binary)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Meaning</td>
<td>Valid</td>
<td>Valid</td>
<td>Valid</td>
<td>Empty</td>
<td>Empty</td>
<td>Empty</td>
<td>Valid</td>
<td>Valid</td>
</tr>
</tbody>
</table>

If necessary, software can decode the single-bit FXSAVE tag-word fields into the two-bit field FSAVE uses by examining the contents of the corresponding FPR registers saved by FXSAVE. Table 11-4 on page 333 shows how the FPR contents are used to find the equivalent FSAVE tag-field value. The fraction column refers to fraction portion of the extended-precision significand (bits 62:0). The integer bit column refers to the integer-portion of the significand (bit 63). See Chapter 11, “SSE, MMX, and x87 Programming,” on page 309 for more information on floating-point numbering formats.
When system software supports multi-tasking, it must be able to save the processor state for one task and load the state for another. For performance reasons, the media and/or x87 processor state is usually saved and loaded only when necessary. System software can save and load this state at the time a task switch occurs. However, if the new task does not use the state, loading the state is unnecessary and reduces performance.

The task-switch bit (CR0.TS) is provided as a lazy context-switch mechanism that allows system software to save and load the processor state only when necessary. When CR0.TS=1, a device-not-available exception (#NM) occurs when an attempt is made to execute a 128-bit media, 64-bit media, or x87 instruction. System software can use the #NM exception handler to save the state of the previous task, and restore the state of the current task. Before returning from the exception handler to the media or x87 instruction, system software must clear CR0.TS to 0 to allow the instruction to be executed. Using this approach, the processor state is saved only when the registers are used.

In legacy mode, the hardware task-switch mechanism sets CR0.TS=1 during a task switch (see “Task Switched (TS) Bit” on page 44 for more information). In long mode, the hardware task-switching is not supported, and the CR0.TS bit is not set by the processor. Instead, the architecture assumes that system software handles all task-switching and state-saving functions. If CR0.TS is to be used in long mode for controlling the save and restore of media or x87 state, system software must set and clear it explicitly.

### Table 11-4. Deriving FSAVE Tag Field from FXSAVE Tag Field

<table>
<thead>
<tr>
<th>Encoded FXSAVE Tag Field</th>
<th>Exponent</th>
<th>Integer Bit(^2)</th>
<th>Fraction(^1)</th>
<th>Type of Value</th>
<th>Equivalent FSAVE Tag Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (Valid)</td>
<td>All 0s</td>
<td>0</td>
<td>All 0s</td>
<td>Zero</td>
<td>01 (Zero)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>Not all 0s</td>
<td>Denormal</td>
<td>00 (Valid)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>All 0s</td>
<td>Pseudo Denormal</td>
<td>10 (Special)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Not all 0s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neither all 0s nor all 1s</td>
<td>0</td>
<td>don't care</td>
<td></td>
<td>Unnormal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>Normal</td>
<td></td>
</tr>
<tr>
<td>0 (Empty)</td>
<td>don't care</td>
<td></td>
<td></td>
<td>Empty</td>
<td>11 (Empty)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>All 0s</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Not all 0s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:**
1. Bits 62:0 of the significand. Bit 62, the most-significant bit of the fraction, is also called the M bit.
2. Bit 63 of the significand, also called the J bit.

**Performance Considerations.** When system software supports multi-tasking, it must be able to save the processor state for one task and load the state for another. For performance reasons, the media and/or x87 processor state is usually saved and loaded only when necessary. System software can save and load this state at the time a task switch occurs. However, if the new task does not use the state, loading the state is unnecessary and reduces performance.
12 Task Management

This chapter describes the hardware task-management features. All of the legacy x86 task-management features are supported by the AMD64 architecture in legacy mode, but most features are not available in long mode. Long mode, however, requires system software to initialize and maintain certain task-management resources. The details of these resource-initialization requirements for long mode are discussed in “Task-Management Resources” on page 336.

12.1 Hardware Multitasking Overview

A task (also called a process) is a program that the processor can execute, suspend, and later resume executing at the point of suspension. During the time a task is suspended, other tasks are allowed to execute. Each task has its own execution space, consisting of:

- Code segment and instruction pointer.
- Data segments.
- Stack segments for each privilege level.
- General-purpose registers.
- rFLAGS register.
- Local-descriptor table.
- Task register, and a link to the previously-executed task.
- I/O-permission and interrupt-permission bitmaps.
- Pointer to the page-translation tables (CR3).

The state information defining this execution space is stored in the task-state segment (TSS) maintained for each task.

Support for hardware multitasking is provided in legacy mode. Hardware multitasking provides automated mechanisms for switching tasks, saving the execution state of the suspended task, and restoring the execution state of the resumed task. When hardware multitasking is used to switch tasks, the processor takes the following actions:

- Suspends execution of the task, allowing any executing instructions to complete and save their results.
- Saves the task execution state in the task TSS.
- Loads the execution state for the new task from its TSS.
- Begins executing the new task at the location specified in the new task TSS.

Software can switch tasks by branching to a new task using the CALL or JMP instructions. Exceptions and interrupts can also switch tasks if the exception or interrupt handlers are themselves separate tasks. IRET can be used to return to an earlier task.
12.2 Task-Management Resources

The hardware-multitasking features are available when protected mode is enabled (CR0.PE=1). Protected-mode software execution, by definition, occurs as part of a task. While system software is not required to use the hardware-multitasking features, it is required to initialize certain task-management resources for at least one task (the current task) when running in protected mode. This single task is needed to establish the protected-mode execution environment. The resources that must be initialized are:

- **Task-State Segment (TSS)**—A segment that holds the processor state associated with a task.
- **TSS Descriptor**—A segment descriptor that defines the task-state segment.
- **TSS Selector**—A segment selector that references the TSS descriptor located in the GDT.
- **Task Register**—A register that holds the TSS selector and TSS descriptor for the current task.

Figure 12-1 on page 337 shows the relationship of these resources to each other in both 64-bit and 32-bit operating environments.
A fifth resource is available in legacy mode for use by system software that uses the hardware-multitasking mechanism to manage more than one task:

- **Task-Gate Descriptor**—This form of gate descriptor holds a reference to a TSS descriptor and is used to control access between tasks.
The task-management resources are described in the following sections.

### 12.2.1 TSS Selector

TSS selectors are selectors that point to task-state segment descriptors in the GDT. Their format is identical to all other segment selectors, as shown in Figure 12-2.

![Figure 12-2. Task-Segment Selector](image)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:3</td>
<td>Selector Index</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TI</td>
<td>Table Indicator</td>
</tr>
<tr>
<td>1:0</td>
<td>RPL</td>
<td>Requestor Privilege Level</td>
</tr>
</tbody>
</table>

The selector format consists of the following fields:

**Selector Index.** Bits 15:3. The selector-index field locates the TSS descriptor in the global-descriptor table.

**Table Indicator (TI) Bit.** Bit 2. The TI bit must be cleared to 0, which indicates that the GDT is used. TSS descriptors cannot be located in the LDT. If a reference is made to a TSS descriptor in the LDT, a general-protection exception (#GP) occurs.

**Requestor Privilege-Level (RPL) Field.** Bits 1:0. RPL represents the privilege level (CPL) the processor is operating under at the time the TSS selector is loaded into the task register.

### 12.2.2 TSS Descriptor

The TSS descriptor is a system-segment descriptor, and it can be located only in the GDT. The format for an 8-byte, legacy-mode and compatibility-mode TSS descriptor can be found in “System Descriptors” on page 87. The format for a 16-byte, 64-bit mode TSS descriptor can be found in “System Descriptors” on page 92.

The fields within a TSS descriptor (all modes) are described in “Descriptor Format” on page 82. The following additional information applies to TSS descriptors:

- **Segment Limit**—A TSS descriptor must have a segment limit value of at least 67h, which defines a minimum TSS size of 68h (104 decimal) bytes. If the limit is less than 67h, an invalid-TSS exception (#TS) occurs during the task switch. When an I/O-permission bitmap, interrupt-redirection bitmap, or additional state information is included in the TSS, the limit must be set to a value large enough to enclose that information. In this case, if the TSS limit is not large enough to
hold the additional information, a #GP exception occurs when an attempt is made to access beyond the TSS limit. No check for the larger limit is performed during the task switch.

- **Type**—Four system-descriptor types are defined as TSS types, as shown in Table 4-5 on page 87. Bit 9 is used as the descriptor busy bit (B). This bit indicates that the task is busy when set to 1, and available when cleared to 0. Busy tasks are the currently running task and any previous (outer) tasks in a nested-task hierarchy. Task recursion is not supported, and a #GP exception occurs if an attempt is made to transfer control to a busy task. See “Nesting Tasks” on page 353 for additional information.

In long mode, the 32-bit TSS types (available and busy) are redefined as 64-bit TSS types, and only 64-bit TSS descriptors can be used. Loading the task register with an available 64-bit TSS causes the processor to change the TSS descriptor type to indicate a busy 64-bit TSS. Because long mode does not support task switching, the TSS-descriptor busy bit is never cleared by the processor to indicate an available 64-bit TSS.

Sixteen-bit TSS types are illegal in long mode. A general-protection exception (#GP) occurs if a reference is made to a 16-bit TSS.

### 12.2.3 Task Register

The **task register** (TR) points to the TSS location in memory, defines its size, and specifies its attributes. As with the other descriptor-table registers, the TR has two portions. A visible portion holds the TSS selector, and a hidden portion holds the TSS descriptor. When the TSS selector is loaded into the TR, the processor automatically loads the TSS descriptor from the GDT into the hidden portion of the TR.

The TR is loaded with a new selector using the LTR instruction. The TR is also loaded during a task switch, as described in “Switching Tasks” on page 349.

Figure 12-3 shows the format of the TR in legacy mode.

![Figure 12-3. TR Format, Legacy Mode](image)
Figure 12-4 shows the format of the TR in long mode (both compatibility mode and 64-bit mode).

The AMD64 architecture expands the TSS-descriptor base-address field to 64 bits so that system software running in long mode can access a TSS located anywhere in the 64-bit virtual-address space. The processor ignores the 32 high-order base-address bits when running in legacy mode. Because the TR is loaded from the GDT, the system-segment descriptor format has been expanded to 16 bytes by the AMD64 architecture in support of 64-bit mode. See “System Descriptors” on page 92 for more information on this expanded format. The high-order base-address bits are only loaded from 64-bit mode using the LTR instruction. Figure 12-5 shows the relationship between the TSS and GDT.

Figure 12-5. Relationship between the TSS and GDT
Long mode requires the use of a 64-bit TSS type, and this type must be loaded into the TR by executing the LTR instruction in 64-bit mode. Executing the LTR instruction in 64-bit mode loads the TR with the full 64-bit TSS base address from the 16-byte TSS descriptor format (compatibility mode can only load 8-byte system descriptors). A processor running in either compatibility mode or 64-bit mode uses the full 64-bit TR.base address.

### 12.2.4 Legacy Task-State Segment

The task-state segment (TSS) is a data structure in memory that the processor uses to save and restore the execution state for a task when a task switch occurs. Figure 12-6 on page 342 shows the format of a legacy 32-bit TSS.
Figure 12-6. Legacy 32-bit TSS
The 32-bit TSS contains three types of fields:

- **Static fields** are read by the processor during a task switch when a new task is loaded, but are not written by the processor when a task is suspended.
- **Dynamic fields** are read by the processor during a task switch when a new task is loaded, and are written by the processor when a task is suspended.
- **Software-defined fields** are read and written by software, but are not read or written by the processor. All but the first 104 bytes of a TSS can be defined for software purposes, minus any additional space required for the optional I/O-permission bitmap and interrupt-redirection bitmap.

TSS fields are not read or written by the processor when the LTR instruction is executed. The LTR instruction loads the TSS descriptor into the TR and marks the task as busy, but it does not cause a task switch.

The TSS fields used by the processor in legacy mode are:

- **Link**—Bytes 01h–00h, dynamic field. Contains a copy of the task selector from the previously-executed task. See “Nesting Tasks” on page 353 for additional information.
- **Stack Pointers**—Bytes 1Bh–04h, static field. Contains the privilege 0, 1, and 2 stack pointers for the task. These consist of the stack-segment selector (SS<sub>n</sub>) and the stack-segment offset (ESP<sub>n</sub>).
- **CR3**—Bytes 1Fh–1Ch, static field. Contains the page-translation-table base-address (CR3) register for the task.
- **EIP**—Bytes 23h–20h, dynamic field. Contains the instruction pointer (EIP) for the next instruction to be executed when the task is restored.
- **EFLAGS**—Bytes 27h–24h, dynamic field. Contains a copy of the EFLAGS image at the point the task is suspended.
- **General-Purpose Registers**—Bytes 47h–28h, dynamic field. Contains a copy of the EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI values at the point the task is suspended.
- **Segment-Selector Registers**—Bytes 59h–48h, dynamic field. Contains a copy of the ES, CS, SS, DS, FS, and GS, values at the point the task is suspended.
- **LDT Segment-Selector Register**—Bytes 63h–60h, static field. Contains the local-descriptor-table segment selector for the task.
- **T (Trap) Bit**—Bit 0 of byte 64h, static field. This bit, when set to 1, causes a debug exception (#DB) to occur on a task switch. See “Breakpoint Instruction (INT3)” on page 368 for additional information.
- **I/O-Permission Bitmap Base Address**—Bytes 67h–66h, static field. This field represents a 16-bit offset into the TSS. This offset points to the beginning of the I/O-permission bitmap, and the end of the interrupt-redirection bitmap.
- **I/O-Permission Bitmap**—Static field. This field specifies protection for I/O-port addresses (up to the 64K ports supported by the processor), as follows:
  - Whether the port can be accessed at any privilege level.
  - Whether the port can be accessed outside the privilege level established by EFLAGS.IOPL.
- Whether the port can be accessed when the processor is running in virtual-8086 mode.

Because one bit is used per 8-byte I/O-port, this bitmap can take up to 8 Kbytes of TSS space. The bitmap can be located anywhere within the first 64 Kbytes of the TSS, as long as it is above byte 103. The last byte of the bitmap must contain all ones (0FFh). See “I/O-Permission Bitmap” on page 344 for more information.

- **Interrupt-Redirection Bitmap**—Static field. This field defines how each of the 256-possible software interrupts is directed in a virtual-8086 environment. One bit is used for each interrupt, for a total bitmap size of 32 bytes. The bitmap can be located anywhere above byte 103 within the first 64 Kbytes of the TSS. See “Interrupt Redirection of Software Interrupts” on page 264 for information on using this field.

The TSS can be paged by system software. System software that uses the hardware task-switch mechanism must guarantee that a page fault does not occur during a task switch. Because the processor only reads and writes the first 104 TSS bytes during a task switch, this restriction only applies to those bytes. The simplest approach is to align the TSS on a page boundary so that all critical bytes are either present or not present. Then, if a page fault occurs when the TSS is accessed, it occurs before the first byte is read. If the page fault occurs after a portion of the TSS is read, the fault is unrecoverable.

**I/O-Permission Bitmap.** The I/O-permission bitmap (IOPB) allows system software to grant less-privileged programs access to individual I/O ports, overriding the effect of RFLAGS.IOPL for those devices. When an I/O instruction is executed, the processor checks the IOPB only if the processor is in virtual x86 mode or the CPL is greater than the RFLAGS.IOPL field. Each bit in the IOPB corresponds to a byte I/O port. A word I/O port corresponds to two consecutive IOPB bits, and a doubleword I/O port corresponds to four consecutive IOPB bits. Access is granted to an I/O port of a given size when all IOPB bits corresponding to that port are clear. If any bits are set, a #GP occurs.

The IOPB is located in the TSS, as shown by the example in Figure 12-7 on page 345. Each TSS can have a different copy of the IOPB, so access to individual I/O devices can be granted on a task-by-task basis. The I/O-permission bitmap base-address field located at byte 66h in the TSS is an offset into the TSS locating the start of the IOPB. If all 64K IO ports are supported, the IOPB base address must not be greater than 0DFFFh, otherwise accesses to the bitmap cause a #GP to occur. An extra byte must be present after the last IOPB byte. This byte must have all bits set to 1 (0FFh). This allows the processor to read two IOPB bytes each time an I/O port is accessed. By reading two IOPB bytes, the processor can check all bits when unaligned, multi-byte I/O ports are accessed.
Bits in the IOPB sequentially correspond to I/O port addresses. The example in Figure 12-7 shows bits 12 through 15 in the second doubleword of the IOPB cleared to 0. Those bit positions correspond to byte I/O ports 44h through 47h, or alternatively, doubleword I/O port 44h. Because the bits are cleared to zero, software running at any privilege level can access those I/O ports.

By adjusting the TSS limit, it may happen that some ports in the I/O-address space have no corresponding IOPB entry. Ports not represented by the IOPB will cause a #GP exception. Referring again to Figure 12-7, the last IOPB entry is at bit 23 in the fourth IOPB doubleword, which corresponds to I/O port 77h. In this example, all ports from 78h and above will cause a #GP exception, as if their permission bit was set to 1.

### 12.2.5 64-Bit Task State Segment

Although the hardware task-switching mechanism is not supported in long mode, a 64-bit task state segment (TSS) must still exist. System software must create at least one 64-bit TSS for use after activating long mode, and it must execute the LTR instruction, *in 64-bit mode*, to load the TR register with a pointer to the 64-bit TSS that serves both 64-bit-mode programs and compatibility-mode programs.

The legacy TSS contains several fields used for saving and restoring processor-state information. The legacy fields include general-purpose register, EFLAGS, CR3 and segment-selector register state, among others. Those legacy fields are not supported by the 64-bit TSS. System software must save and restore the necessary processor-state information required by the software-multitasking implementation (if multitasking is supported). Figure 12-8 on page 347 shows the format of a 64-bit TSS.

The 64-bit TSS holds several pieces of information important to long mode that are not directly related to the task-switch mechanism:

- *RSPn*—Bytes 1Bh–04h. The full 64-bit canonical forms of the stack pointers (RSP) for privilege levels 0 through 2.
• *ISTn*—Bytes 5Bh–24h. The full 64-bit canonical forms of the interrupt-stack-table (IST) pointers. See “Interrupt-Stack Table” on page 259 for a description of the IST mechanism.

• *I/O Map Base Address*—Bytes 67h–66h. The 16-bit offset to the I/O-permission bit map from the 64-bit TSS base. The function of this field is identical to that in a legacy 32-bit TSS. See “I/O-Permission Bitmap” on page 344 for more information.
**Figure 12-8. Long Mode TSS Format**

<table>
<thead>
<tr>
<th>Bit Offset</th>
<th>Byte Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 16 15 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I/O-Permission Bitmap (IOPB) (Up to 8 Kbytes)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I/O Map Base Address</th>
<th>Reserved, IGN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved, IGN</td>
<td>+64h</td>
</tr>
<tr>
<td>Reserved, IGN</td>
<td>+60h</td>
</tr>
<tr>
<td>Reserved, IGN</td>
<td>+5Ch</td>
</tr>
<tr>
<td>IST7[63:32]</td>
<td>+58h</td>
</tr>
<tr>
<td>IST7[31:0]</td>
<td>+54h</td>
</tr>
<tr>
<td>IST6[63:32]</td>
<td>+50h</td>
</tr>
<tr>
<td>IST6[31:0]</td>
<td>+4Ch</td>
</tr>
<tr>
<td>IST5[63:32]</td>
<td>+48h</td>
</tr>
<tr>
<td>IST5[31:0]</td>
<td>+44h</td>
</tr>
<tr>
<td>IST4[63:32]</td>
<td>+40h</td>
</tr>
<tr>
<td>IST4[31:0]</td>
<td>+3Ch</td>
</tr>
<tr>
<td>IST3[63:32]</td>
<td>+38h</td>
</tr>
<tr>
<td>IST3[31:0]</td>
<td>+34h</td>
</tr>
<tr>
<td>IST2[63:32]</td>
<td>+30h</td>
</tr>
<tr>
<td>IST2[31:0]</td>
<td>+2Ch</td>
</tr>
<tr>
<td>IST1[63:32]</td>
<td>+28h</td>
</tr>
<tr>
<td>IST1[31:0]</td>
<td>+24h</td>
</tr>
<tr>
<td>Reserved, IGN</td>
<td>+20h</td>
</tr>
<tr>
<td>RSP2[63:32]</td>
<td>+1Ch</td>
</tr>
<tr>
<td>RSP2[31:0]</td>
<td>+18h</td>
</tr>
<tr>
<td>RSP1[63:32]</td>
<td>+14h</td>
</tr>
<tr>
<td>RSP1[31:0]</td>
<td>+10h</td>
</tr>
<tr>
<td>RSP0[63:32]</td>
<td>+0Ch</td>
</tr>
<tr>
<td>RSP0[31:0]</td>
<td>+08h</td>
</tr>
<tr>
<td>Reserved, IGN</td>
<td>+04h</td>
</tr>
<tr>
<td>Reserved, IGN</td>
<td>+00h</td>
</tr>
</tbody>
</table>
12.2.6 Task Gate Descriptor (Legacy Mode Only)

Task-gate descriptors hold a selector reference to a TSS and are used to control access between tasks. Unlike a TSS descriptor or other gate descriptors, a task gate can be located in any of the three descriptor tables (GDT, LDT, and IDT). Figure 12-9 shows the format of a task-gate descriptor.

![Figure 12-9. Task-Gate Descriptor, Legacy Mode Only](image)

The task-gate descriptor fields are:

- **System (S) and Type**—Bits 12 and 11:8 (respectively) of byte +4. These bits are encoded by software as 00101b to indicate a task-gate descriptor type.

- **Present (P)**—Bit 15 of byte +4. The segment-present bit indicates the segment referenced by the gate descriptor is loaded in memory. If a reference is made to a segment when P=0, a segment-not-present exception (#NP) occurs. This bit is set and cleared by system software and is never altered by the processor.

- **Descriptor Privilege-Level (DPL)**—Bits 14:13 of byte +4. The DPL field indicates the gate-descriptor privilege level. DPL can be set to any value from 0 to 3, with 0 specifying the most privilege and 3 the least privilege.

12.3 Hardware Task-Management in Legacy Mode

This section describes the operation of the task-switch mechanism when the processor is running in legacy mode. None of these features are supported in long mode (either compatibility mode or 64-bit mode).

12.3.1 Task Memory-Mapping

The hardware task-switch mechanism gives system software a great deal of flexibility in managing the sharing and isolation of memory—both virtual (linear) and physical—between tasks.

**Segmented Memory.** The segmented memory for a task consists of the segments that are loaded during a task switch and any segments that are later accessed by the task code. The hardware task-switch mechanism allows tasks to either share segments with other tasks, or to access segments in isolation from one another. Tasks that share segments actually share a virtual-address (linear-address) space, but they do not necessarily share a physical-address space. When paging is enabled, the virtual-to-physical mapping for each task can differ, as is described in the following section. Shared segments
do share physical memory when paging is disabled, because virtual addresses are used as physical addresses.

A number of options are available to system software that shares segments between tasks:

- Sharing segment descriptors using the GDT. All tasks have access to the GDT, so it is possible for segments loaded in the GDT to be shared among tasks.

- Sharing segment descriptors using a single LDT. Each task has its own LDT, and that LDT selector is automatically saved and restored in the TSS by the processor during task switches. Tasks, however, can share LDTs simply by storing the same LDT selector in multiple TSSs. Using the LDT to manage segment sharing and segment isolation provides more flexibility to system software than using the GDT for the same purpose.

- Copying shared segment descriptors into multiple LDTs. Segment descriptors can be copied by system software into multiple LDTs that are otherwise not shared between tasks. Allowing segment sharing at the segment-descriptor level, rather than the LDT level or GDT level, provides the greatest flexibility to system software.

In all three cases listed above, the actual data and instructions are shared between tasks only when the tasks’ virtual-to-physical address mappings are identical.

**Paged Memory.** Each task has its own page-translation table base-address (CR3) register, and that register is automatically saved and restored in the TSS by the processor during task switches. This allows each task to point to its own set of page-translation tables, so that each task can translate virtual addresses to physical addresses independently. Page translation must be enabled for changes in CR3 values to have an effect on virtual-to-physical address mapping. When page translation is disabled, the tables referenced by CR3 are ignored, and virtual addresses are equivalent to physical addresses.

### 12.3.2 Switching Tasks

The hardware task-switch mechanism transfers program control to a new task when any of the following occur:

- A CALL or JMP instruction with a selector operand that references a task gate is executed. The task gate can be located in either the LDT or GDT.

- A CALL or JMP instruction with a selector operand that references a TSS descriptor is executed. The TSS descriptor must be located in the GDT.

- A software-interrupt instruction (INTn) is executed that references a task gate located in the IDT.

- An exception or external interrupt occurs, and the vector references a task gate located in the IDT.

- An IRET is executed while the EFLAGS.NT bit is set to 1, indicating that a return is being performed from an inner-level task to an outer-level task. The new task is referenced using the selector stored in the current-task link field. See “Nesting Tasks” on page 353 for additional information. The RET instruction cannot be used to switch tasks.

When a task switch occurs, the following operations are performed automatically by the processor:
The processor performs privilege-checking to determine whether the currently-executing program is allowed to access the target task. If this check fails, the task switch is aborted without modifying the processor state, and a general-protection exception (#GP) occurs. The privilege checks performed depend on the cause of the task switch:

- If the task switch is initiated by a CALL or JMP instruction through a TSS descriptor, the processor checks that both the currently-executing program CPL and the TSS-selector RPL are numerically less-than or equal-to the TSS-descriptor DPL.
- If the task switch takes place through a task gate, the CPL and task-gate RPL are compared with the task-gate DPL, and no comparison is made using the TSS-descriptor DPL. See “Task Switches Using Task Gates” on page 351.
- Software interrupts, hardware interrupts, and exceptions all transfer control without checking the task-gate DPL.
- The IRET instruction transfers control without checking the TSS-descriptor DPL.

The processor performs limit-checking on the target TSS descriptor to verify that the TSS limit is greater than or equal to 67h (at least 104 bytes). If this check fails, the task switch is aborted without modifying the processor state, and an invalid-TSS exception (#TS) occurs.

The current-task state is saved in the TSS. This includes the next-instruction pointer (EIP), EFLAGS, the general-purpose registers, and the segment-selector registers.

Up to this point, any exception that occurs aborts the task switch without changing the processor state. From this point forward, any exception that occurs does so in the context of the new task. If an exception occurs in the context of the new task during a task switch, the processor finishes loading the new-task state without performing additional checks. The processor transfers control to the #TS handler after this state is loaded, but before the first instruction is executed in the new task. When a #TS occurs, it is possible that some of the state loaded by the processor did not participate in segment access checks. The #TS handler must verify that all segments are accessible before returning to the interrupted task.

The task register (TR) is loaded with the new-task TSS selector, and the hidden portion of the TR is loaded with the new-task descriptor. The TSS now referenced by the processor is that of the new task.

The current task is marked as busy. The previous task is marked as available or remains busy, based on the type of linkage. See “Nesting Tasks” on page 353 for more information.

CR0.TS is set to 1. This bit can be used to save other processor state only when it becomes necessary. For more information, see the next section, “Saving Other Processor State.”

The new-task state is loaded from the TSS. This includes the next-instruction pointer (EIP), EFLAGS, the general-purpose registers, and the segment-selector registers. The processor clears the segment-descriptor present (P) bits (in the hidden portion of the segment registers) to prevent access into the new segments, until the task switch completes successfully.

The LDTR and CR3 registers are loaded from the TSS, changing the virtual-to-physical mapping from that of the old task to the new task. Because this is done in the middle of accessing the new TSS, system software must guarantee that TSS addresses are translated identically in all tasks.
The descriptors for all previously-loaded segment selectors are loaded into the hidden portion of the segment registers. This sets or clears the P bits for the segments as specified by the new descriptor values.

If the above steps complete successfully, the processor begins executing instructions in the new task beginning with the instruction referenced by the CS:EIP far pointer loaded from the new TSS. The privilege level of the new task is taken from the new CS segment selector’s RPL.

**Saving Other Processor State.** The processor does not automatically save the registers used by the media or x87 instructions. Instead, the processor sets CR0.TS to 1 during a task switch. Later, when an attempt is made to execute any of the media or x87 instructions while TS=1, a device-not-available exception (#NM) occurs. System software can then save the previous state of the media and x87 registers and clear the CR0.TS bit to 0 before executing the next media/x87 instruction. As a result, the media and x87 registers are saved only when necessary after a task switch.

### 12.3.3 Task Switches Using Task Gates

When a control transfer to a new task occurs through a task gate, the processor reads the task-gate DPL (DPL\textsubscript{G}) from the task-gate descriptor. Two privilege checks, both of which must pass, are performed on DPL\textsubscript{G} before the task switch can occur successfully:

- The processor compares the CPL with DPL\textsubscript{G}. The CPL must be numerically less than or equal to DPL\textsubscript{G} for this check to pass. In other words, the following expression must be true: CPL ≤ DPL\textsubscript{G}.

- The processor compares the RPL in the task-gate selector with DPL\textsubscript{G}. The RPL must be numerically less than or equal to DPL\textsubscript{G} for this check to pass. In other words, the following expression must be true: RPL ≤ DPL\textsubscript{G}.

Unlike call-gate control transfers, the processor does not read the DPL from the target TSS descriptor (DPL\textsubscript{S}) and compare it with the CPL when a task gate is used.

Figure 12-10 on page 352 shows two examples of task-gate privilege checks. In Example 1, the privilege checks pass:

- The task-gate DPL (DPL\textsubscript{G}) is at the lowest privilege (3), specifying that software running at any privilege level (CPL) can access the gate.

- The selector referencing the task gate passes its privilege check because the RPL is numerically less than or equal to DPL\textsubscript{G}.

In Example 2, both privilege checks fail:

- The task-gate DPL (DPL\textsubscript{G}) specifies that only software at privilege-level 0 can access the gate. The current program does not have enough privilege to access the task gate, because its CPL is 2.

- The selector referencing the task-gate descriptor does not have a high enough privilege to complete the reference. Its RPL is numerically greater than DPL\textsubscript{G}.

Although both privilege checks failed in the example, if only one check fails, access into the target task is denied.
Because the legacy task-switch mechanism is not supported in long mode, *software cannot use task gates in long mode*. Any attempt to transfer control to another task using a task gate in long mode causes a general-protection exception (#GP) to occur.

**Figure 12-10. Privilege-Check Examples for Task Gates**
12.3.4 Nesting Tasks

The hardware task-switch mechanism supports task nesting through the use of EFLAGS nested-task (NT) bit and the TSS link-field. The manner in which these fields are updated and used during a task switch depends on how the task switch is initiated:

- The JMP instruction does not update EFLAGS.NT or the TSS link-field. Task nesting is not supported by the JMP instruction.
- The CALL instruction, INTn instructions, interrupts, and exceptions can only be performed from outer-level tasks to inner-level tasks. All of these operations set the EFLAGS.NT bit for the new task to 1 during a task switch, and copy the selector for the previous task into the new-task link field.
- An IRET instruction which returns to another task only occurs when the EFLAGS.NT bit for the current task is set to 1, and only can be performed from an inner-level task to an outer-level task. When an IRET results in a task switch, the new task is referenced using the selector stored in the current-TSS link field. The EFLAGS.NT bit for the current task is cleared to 0 during the task switch.

Table 12-1 summarizes the effect various task-switch initiators have on EFLAGS.NT, the TSS link-field, and the TSS-busy bit. (For more information on the busy bit, see the next section, “Preventing Recursion.”)

Table 12-1. Effects of Task Nesting

<table>
<thead>
<tr>
<th>Task-Switch Initiator</th>
<th>Old Task</th>
<th>New Task</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EFLAGS.NT</td>
<td>Link (Selector)</td>
</tr>
<tr>
<td>JMP</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CALL</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>INTn Interrupt</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Note: “—” indicates no change is made.

Programs running at any privilege level can set EFLAGS.NT to 1 and execute the IRET instruction to transfer control to another task. System software can keep control over improperly nested-task switches by initializing the link field of all TSSs that it creates. That way, improperly nested-task switches always transfer control to a known task.

**Preventing Recursion.** Task recursion is not allowed by the hardware task-switch mechanism. If recursive-task switches were allowed, they would replace a previous task-state image with a newer image, discarding the previous information. To prevent recursion from occurring, the processor uses
the busy bit located in the TSS-descriptor type field (bit 9 of byte +4). Use of this bit depends on how
the task switch is initiated:

- The JMP instruction clears the busy bit in the old task to 0 and sets the busy bit in the new task to 1. A general-protection exception (#GP) occurs if an attempt is made to JMP to a task with a set busy bit.
- The CALL instruction, INTn instructions, interrupts, and exceptions set the busy bit in the new task to 1. The busy bit in the old task remains set to 1, preventing recursion through task-nesting levels. A general-protection exception (#GP) occurs if an attempt is made to switch to a task with a set busy bit.
- An IRET to another task (EFLAGS.NT must be 1) clears the busy bit in the old task to 0. The busy bit in the new task is not altered, because it was already set to 1.

Table 12-1 on page 353 summarizes the effect various task-switch initiators have on the TSS-busy bit.
13 Software Debug and Performance Resources

Testing, debug, and performance optimization consume a significant portion of the time needed to develop a new computer or software product and move it successfully into production. To stay competitive, product developers need tools that allow them to rapidly detect, isolate, and correct problems before a product is shipped. The goal of the debug and performance features incorporated into processor implementations of the AMD64 architecture is to support the tool chain solutions used in software and hardware product development.

The debug and performance resources that can be supported by AMD64 architecture implementations include:

- **Software Debug**—software-debug facilities include the debug registers (DR0–DR7), debug exception, and breakpoint exception. Additional features are provided using model-specific registers (MSRs). These registers are used to set breakpoints on branches, interrupts, and exceptions and to single step from one branch to the next. The software-debug capability is described in the following section.

- **Performance Monitoring Counters**—Performance monitoring counters (PMCs) are provided to count specific processor hardware events. A set of control registers allow the selection of events to be monitored and a corresponding set of counter registers track the frequency of monitored events. These counters are described in Section 13.2 “Performance Monitoring Counters” on page 370.

- **Instruction-Based Sampling**—Instruction-based sampling is a hardware-based facility that enables system software to capture specific data concerning instruction fetch and instruction execution operation based on random sampling. This facility is described in Section 13.3 “Instruction-Based Sampling” on page 379.

- **Lightweight Profiling**—AMD64 architecture provides instructions that allow user-level programs to manage the gathering of instruction statistics using very little overhead. This facility is described in Section 13.4 “Lightweight Profiling” on page 392.

Although a subset of the facilities listed are available in all processor implementations, the remainder are optional. Support for optional facilities is indicated via CPUID feature bits. The means of determining support for each architected facility is described along with the facility in the sections that follow.

A given processor product may include additional debug and performance monitoring capabilities beyond those which are architecturally-defined. For details see the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual applicable to your product.
13.1 Software-Debug Resources

Software can program breakpoints into the debug registers, causing a debug exception (#DB) when matches occur on instruction-memory addresses, data-memory addresses, or I/O addresses. The breakpoint exception (#BP) is also supported to allow software to set breakpoints by placing INT3 instructions in the instruction memory for a program. Program control is transferred to the breakpoint exception (#BP) handler when an INT3 instruction is executed.

In addition to the debug features supported by the debug registers (DR0–DR7), the processor also supports features supported by model-specific registers (MSRs). Together, these capabilities provide a rich set of breakpoint conditions, including:

- **Breakpoint On Address Match**—Breakpoints occur when the address stored in a address-breakpoint register matches the address of an instruction or data reference. Up to four address-match breakpoint conditions can be set by software.

- **Single Step All Instructions**—Breakpoints can be set to occur on every instruction, allowing a debugger to examine the contents of registers as a program executes.

- **Single Step Control Transfers**—Breakpoints can be set to occur on control transfers, such as calls, jumps, interrupts, and exceptions. This can allow a debugger to narrow a problem search to a specific section of code before enabling single stepping of all instructions.

- **Breakpoint On Any Instruction**—Breakpoints can be set on any specific instruction using either the address-match breakpoint condition or using the INT3 instruction to force a breakpoint when the instruction is executed.

- **Breakpoint On Task Switch**—Software forces a #DB exception to occur when a task switch is performed to a task with the T bit in the TSS set to 1. Debuggers can use this capability to enable or disable debug conditions for a specific task.

Problem areas can be identified rapidly using the information supplied by the debug registers when breakpoint conditions occur:

- Special conditions that cause a #DB exception are recorded in the DR6 debug-status register, including breakpoints due to task switches and single stepping. The DR6 register also identifies which address-breakpoint register (DR0–DR3) caused a #DB exception due to an address match. When combined with the DR7 debug-control register settings, the cause of a #DB exception can be identified.

- To assist in analyzing the instruction sequence a processor follows in reaching its current state, the source and destination addresses of control-transfer events are saved by the processor. These include branches (calls and jumps), interrupts, and exceptions. Debuggers can use this information to narrow a problem search to a specific section of code before single stepping all instructions.

13.1.1 Debug Registers

The AMD64 architecture supports the legacy debug registers, DR0–DR7. These registers are expanded to 64 bits by the AMD64 architecture. In legacy mode and in compatibility mode, only the lower 32 bits are used. In these modes, writes to a debug register fill the upper 32 bits with zeros, and
reads from a debug register return only the lower 32 bits. In 64-bit mode, all 64 bits of the debug registers are read and written. Operand-size prefixes are ignored.

The debug registers can be read and written only when the current-protection level (CPL) is 0 (most privileged). Attempts to read or write the registers at a lower-privilege level (CPL>0) cause a general-protection exception (#GP).

Several debug registers described below are model-specific registers (MSRs). See “Software-Debug MSRs” on page 612 for a listing of the debug-MSR numbers and their reset values. Some processor implementations include additional MSRs used to support implementation-specific software debug features. For more information on these registers and their capabilities, see the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual applicable to your product.

### 13.1.1.1 Address-Breakpoint Registers (DR0-DR3)

Figure 13-1 shows the format of the four address-breakpoint registers, DR0-DR3. Software can load a virtual (linear) address into any of the four registers, and enable breakpoints to occur when the address matches an instruction or data reference. The MOV DRn instructions *do not* check that the virtual addresses loaded into DR0–DR3 are in canonical form. Breakpoint conditions are enabled using the debug-control register, DR7 (see “Debug-Control Register (DR7)” on page 359).

![Figure 13-1. Address-Breakpoint Registers (DR0–DR3)](image)

### 13.1.1.2 Reserved Debug Registers (DR4, DR5)

The DR4 and DR5 registers are reserved and should not be used by software. These registers are aliased to the DR6 and DR7 registers, respectively. When the debug extensions are enabled (CR4[DE] = 1) attempts to access these registers cause an invalid-opcode exception (#UD).
13.1.1.3 Debug-Status Register (DR6)

Figure 13-2 on page 358 shows the format of the debug-status register, DR6. Debug status is loaded into DR6 when an enabled debug condition is encountered that causes a #DB exception.

![Figure 13-2. Debug-Status Register (DR6)](image)

Bits 15:13 of the DR6 register are not cleared by the processor and must be cleared by software after the contents have been read. Register fields are:

- **Breakpoint-Condition Detected (B3–B0)**—Bits 3:0. The processor updates these four bits on every debug breakpoint or general-detect condition. A bit is set to 1 if the corresponding address-breakpoint register detects an enabled breakpoint condition, as specified by the DR7 Ln, Gn, R/Wn and LENn controls, and is cleared to 0 otherwise. For example, B1 (bit 1) is set to 1 if an address-breakpoint condition is detected by DR1.
- **Debug-Register-Access Detected (BD)**—Bit 13. The processor sets this bit to 1 if software accesses any debug register (DR0–DR7) while the general-detect condition is enabled (DR7[GD] = 1).
- **Single Step (BS)**—Bit 14. The processor sets this bit to 1 if the #DB exception occurs as a result of single-step mode (rFLAGS[TF] = 1). Single-step mode has the highest-priority among debug exceptions. Other status bits within the DR6 register can be set by the processor along with the BS bit.
- **Task-Switch (BT)**—Bit 15. The processor sets this bit to 1 if the #DB exception occurred as a result of task switch to a task with a TSS T-bit set to 1.
All remaining bits in the DR6 register are reserved. Reserved bits 31:16 and 11:4 must all be set to 1, while reserved bit 12 must be cleared to 0. In 64-bit mode, the upper 32 bits of DR6 are reserved and must be written with zeros. Writing a 1 to any of the upper 32 bits results in a general-protection exception, #GP(0).

13.1.1.4 Debug-Control Register (DR7)

Figure 13-3 shows the format of the debug-control register, DR7. DR7 is used to establish the breakpoint conditions for the address-breakpoint registers (DR0–DR3) and to enable debug exceptions for each address-breakpoint register individually. DR7 is also used to enable the general-detect breakpoint condition.
The fields within the DR7 register are all read/write. These fields are:

- **Local-Breakpoint Enable (L3–L0)**—Bits 6, 4, 2, and 0 (respectively). Software individually sets these bits to 1 to enable debug exceptions to occur when the corresponding address-breakpoint register (DRn) detects a breakpoint condition while executing the current task. For example, if L1 (bit 2) is set to 1 and an address-breakpoint condition is detected by DR1, a #DB exception occurs. These bits are cleared to 0 by the processor when a hardware task-switch occurs.

- **Global-Breakpoint Enable (G3–G0)**—Bits 7, 5, 3, and 1 (respectively). Software sets these bits to 1 to enable debug exceptions to occur when the corresponding address-breakpoint register (DRn) detects a breakpoint condition while executing any task. For example, if G1 (bit 3) is set to 1 and an address-breakpoint condition is detected by DR1, a #DB exception occurs. These bits are never cleared to 0 by the processor.

- **Local-Enable (LE)**—Bit 8. Software sets this bit to 1 in legacy implementations to enable exact breakpoints while executing the current task. This bit is ignored by implementations of the AMD64 architecture. All breakpoint conditions, except certain string operations preceded by a repeat prefix, are exact.

- **Global-Enable (GE)**—Bit 9. Software sets this bit to 1 in legacy implementations to enable exact breakpoints while executing any task. This bit is ignored by implementations of the AMD64 architecture. All breakpoint conditions, except certain string operations preceded by a repeat prefix, are exact.

- **General-Detect Enable (GD)**—Bit 13. Software sets this bit to 1 to cause a debug exception to occur when an attempt is made to execute a MOV DRn instruction to any debug register (DR0–DR7). This bit is cleared to 0 by the processor when the #DB handler is entered, allowing the handler to read and write the DRn registers. The #DB exception occurs before executing the instruction, and DR6[BD] is set by the processor. Software debuggers can use this bit to prevent the currently-executing program from interfering with the debug operation.

- **Read/Write (R/W3–R/W0)**—Bits 29:28, 25:24, 21:20, and 17:16 (respectively). Software sets these fields to control the breakpoint conditions used by the corresponding address-breakpoint registers (DRn). For example, control-field R/W1 (bits 21:20) controls the breakpoint conditions for the DR1 register. The R/Wn control-field encodings specify the following conditions for an address-breakpoint to occur:
  - 00—Only on instruction execution.
  - 01—Only on data write.
  - 10—This encoding is further qualified by CR4[DE] as follows:
    - CR4[DE] = 0—Condition is undefined.
    - CR4[DE] = 1—Only on I/O read or I/O write.
  - 11—Only on data read or data write.

- **Length (LEN3–LEN0)**—Bits 31:30, 27:26, 23:22, and 19:18 (respectively). Software sets these fields to control the range used in comparing a memory address with the corresponding address-breakpoint register (DRn). For example, control-field LEN1 (bits 23:22) controls the breakpoint-comparison range for the DR1 register.
The value in DR\textsubscript{n} defines the low-end of the address range used in the comparison. LEN\textsubscript{n} is used to mask the low-order address bits in the corresponding DR\textsubscript{n} register so that they are not used in the address comparison. To work properly, breakpoint boundaries must be aligned on an address corresponding to the range size specified by LEN\textsubscript{n}. The LEN\textsubscript{n} control-field encodings specify the following address-breakpoint-comparison ranges:

- 00—1 byte.
- 01—2 byte, must be aligned on a word boundary.
- 10—8 byte, must be aligned on a quadword boundary. (Long mode only; otherwise undefined.)
- 11—4 byte, must be aligned on a doubleword boundary.

If the R/W\textsubscript{n} field is used to specify instruction breakpoints (R/W\textsubscript{n}=00), the corresponding LEN\textsubscript{n} field must be set to 00. Setting LEN\textsubscript{n} to any other value produces undefined results.

All remaining bits in the DR7 register are reserved. Reserved bits 15:14 and 12:11 must all be cleared to 0, while reserved bit 10 must be set to 1. In 64-bit mode, the upper 32 bits of DR7 are reserved and must be written with zeros. Writing a 1 to any of the upper 32 bits results in a general-protection #GP(0) exception.

### 13.1.1.5 64-Bit-Mode Extended Debug Registers

In 64-bit mode, additional encodings for debug registers are available. The R bit of the REX prefix is used to modify the ModRM\textsubscript{reg} field when that field encodes a control register. These additional encodings enable the processor to address DR8–DR15.

Access to the DR8–DR15 registers is implementation-dependent. The architecture does not require any of these extended debug registers to be implemented. Any attempt to access an unimplemented register results in an invalid-opcode exception (#UD).

### 13.1.1.6 Debug-Control MSR (DebugCtl)

Figure 13-4 on page 362 shows the format of the debug-control MSR (DebugCtl). DebugCtl provides additional debug controls over control-transfer recording and single stepping, and external-breakpoint reporting and trace messages. DebugCtl is read and written using the RDMSR and WRMSR instructions.
Figure 13-4. Debug-Control MSR (DebugCtl)

The fields within the DebugCtl register are:

- **Last-Branch Record (LBR)**—Bit 0, read/write. Software sets this bit to 1 to cause the processor to record the source and target addresses of the last control transfer taken before a debug exception occurs. The recorded control transfers include branch instructions, interrupts, and exceptions. See “Control-Transfer Breakpoint Features” on page 368 for more details on the registers. See Figure 13-5 on page 363 for the format of the control-transfer recording MSRs.

- **Branch Single Step (BTF)**—Bit 1, read/write. Software uses this bit to change the behavior of the rFLAGS[TF] bit. When this bit is cleared to 0, the rFLAGS[TF] bit controls instruction single stepping, (normal behavior). When this bit is set to 1, the rFLAGS[TF] bit controls single stepping on control transfers. The single-stepped control transfers include branch instructions, interrupts, and exceptions. Control-transfer single stepping requires both BTF = 1 and rFLAGS[TF] = 1. See “Control-Transfer Breakpoint Features” on page 368 for more details on control-transfer single stepping.

- **Performance-Monitoring/Breakpoint Pin-Control (PBi)**—Bits 5:2, read/write. Software uses these bits to control the type of information reported by the four external performance-monitoring/breakpoint pins on the processor. When a PBi bit is cleared to 0, the corresponding external pin (BPi) reports performance-monitor information. When a PBi bit is set to 1, the corresponding external pin (BPi) reports breakpoint information.

All remaining bits in the DebugCtl register are reserved.
13.1.1.7 Control-Transfer Recording MSRs

Figure 13-5 on page 363 shows the format of the 64-bit control-transfer recording MSRs: LastBranchToIP, LastBranchFromIP, LastIntToIP, and LastIntFromIP. These registers are loaded automatically by the processor when the DebugCtl[LBR] bit is set to 1. These MSRs are read-only.

![Figure 13-5. Control-Transfer Recording MSRs](image)

13.1.2 Setting Breakpoints

Breakpoints can be set to occur on either instruction addresses or data addresses using the breakpoint-address registers, DR0–DR3 (DRn). The values loaded into these registers represent the breakpoint-location virtual address. The debug-control register, DR7, is used to enable the breakpoint registers and to specify the type of access and the range of addresses that can trigger a breakpoint.

Software enables the DRn registers using the corresponding local-breakpoint enable (Ln) or global-breakpoint enable (Gn) found in the DR7 register. Ln is used to enable breakpoints only while the current task is active, and it is cleared by the processor when a task switch occurs. Gn is used to enable breakpoints for all tasks, and it is never cleared by the processor.

The R/Wn fields in DR7, along with the CR4[DE] bit, specify the type of access required to trigger a breakpoint when an address match occurs on the corresponding DRn register. Breakpoints can be set to occur on instruction execution, data reads and writes, and I/O reads and writes. The R/Wn and CR4[DE] encodings used to specify the access type are described on page 360 of “Debug-Control Register (DR7).”

The LENn fields in DR7 specify the size of the address range used in comparison with data or instruction addresses. LENn is used to mask the low-order address bits in the corresponding DRn register so that they are not used in the address comparison. Breakpoint boundaries must be aligned on an address corresponding to the range size specified by LENn. Assuming the access type matches the
type specified by R/W\textsubscript{n}, a breakpoint occurs if any accessed byte falls within the range specified by LEN\textsubscript{n}. For instruction breakpoints, LEN\textsubscript{n} must specify a single-byte range. The LEN\textsubscript{n} encodings used to specify the address range are described on page 360 of “Debug-Control Register (DR7).”

Table 13-1 shows several examples of data accesses, and whether or not they cause a #DB exception to occur based on the breakpoint address in DR\textsubscript{n} and the breakpoint-address range specified by LEN\textsubscript{n}. In this table, R/W\textsubscript{n} always specifies read/write access.

### Table 13-1. Breakpoint-Setting Examples

<table>
<thead>
<tr>
<th>Data-Access Address (hexadecimal)</th>
<th>Access Size (bytes)</th>
<th>Byte-Addresses in Data-Access (hexadecimal)</th>
<th>Breakpoint-Address Range (hexadecimal)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DR\textsubscript{n}=F000, LEN\textsubscript{n}=00 (1 Byte)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EFFB</td>
<td>8</td>
<td>EFFB, EFFC, EFFD, EFFE, EFFF, F000, F001</td>
<td></td>
<td>#DB</td>
</tr>
<tr>
<td>EFFE</td>
<td>2</td>
<td>EFFE, EFFF</td>
<td></td>
<td>—</td>
</tr>
<tr>
<td>F000</td>
<td>1</td>
<td>F000</td>
<td></td>
<td>#DB</td>
</tr>
<tr>
<td>F001</td>
<td>2</td>
<td>F001, F002</td>
<td></td>
<td>—</td>
</tr>
<tr>
<td>F005</td>
<td>4</td>
<td>F005, F006, F007, F008</td>
<td></td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DR\textsubscript{n}=F004, LEN\textsubscript{n}=11 (4 Bytes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EFFB</td>
<td>8</td>
<td>EFFB, EFFC, EFFD, EFFE, EFFF, F000, F001</td>
<td></td>
<td>—</td>
</tr>
<tr>
<td>EFFE</td>
<td>2</td>
<td>EFFE, EFFF</td>
<td></td>
<td>—</td>
</tr>
<tr>
<td>EFFE</td>
<td>4</td>
<td>EFFE, EFFF, F000, F001</td>
<td></td>
<td>#DB</td>
</tr>
<tr>
<td>F000</td>
<td>1</td>
<td>F000</td>
<td></td>
<td>—</td>
</tr>
<tr>
<td>F001</td>
<td>2</td>
<td>F001, F002</td>
<td></td>
<td>—</td>
</tr>
<tr>
<td>F005</td>
<td>4</td>
<td>F005, F006, F007, F008</td>
<td></td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DR\textsubscript{n}=F005, LEN\textsubscript{n}=10 (8 Bytes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EFFB</td>
<td>8</td>
<td>EFFB, EFFC, EFFD, EFFE, EFFF, F000, F001</td>
<td></td>
<td>#DB</td>
</tr>
<tr>
<td>EFFE</td>
<td>2</td>
<td>EFFE, EFFF</td>
<td></td>
<td>—</td>
</tr>
<tr>
<td>EFFE</td>
<td>4</td>
<td>EFFE, EFFF, F000, F001</td>
<td></td>
<td>#DB</td>
</tr>
<tr>
<td>F000</td>
<td>1</td>
<td>F000</td>
<td></td>
<td>—</td>
</tr>
<tr>
<td>F001</td>
<td>2</td>
<td>F001, F002</td>
<td></td>
<td>—</td>
</tr>
<tr>
<td>F005</td>
<td>4</td>
<td>F005, F006, F007, F008</td>
<td></td>
<td>—</td>
</tr>
</tbody>
</table>

**Note:** "—" indicates no #DB occurs.
13.1.3 Using Breakpoints

A debug exception (#DB) occurs when an enabled-breakpoint condition is encountered during program execution. The debug-handler must check the debug-status register (DR6), the conditions enabled by the debug-control register (DR7), and the debug-control MSR (DebugCtl), to determine the #DB cause. The #DB exception corresponds to interrupt vector 1. See “#DB—Debug Exception (Vector 1)” on page 225.

Instruction breakpoints and general-detect conditions cause the #DB exception to occur before the instruction is executed, while all other breakpoint and single-stepping conditions cause the #DB exception to occur after the instruction is executed. Table 13-2 summarizes where the #DB exception occurs based on the breakpoint condition.

<table>
<thead>
<tr>
<th>Breakpoint Condition</th>
<th>Breakpoint Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Before Instruction is Executed</td>
</tr>
<tr>
<td>General Detect</td>
<td></td>
</tr>
<tr>
<td>Data Write Only</td>
<td></td>
</tr>
<tr>
<td>Data Read or Data Write</td>
<td>After Instruction is Executed¹</td>
</tr>
<tr>
<td>I/O Read or I/O Write</td>
<td></td>
</tr>
<tr>
<td>Single Step¹</td>
<td>After Instruction is Executed</td>
</tr>
<tr>
<td>Task Switch</td>
<td></td>
</tr>
</tbody>
</table>

Note:
1. Repeated operations (REP prefix) can breakpoint between iterations.

Instruction breakpoints and general-detect conditions have a lower interrupt-priority than the other breakpoint and single-stepping conditions (see “Priorities” on page 240). Data-breakpoint conditions on the previous instruction occur before an instruction-breakpoint condition on the next instruction. However, if instruction and data breakpoints can occur as a result of executing a single instruction, the instruction breakpoint occurs first (before the instruction is executed), followed by the data breakpoint (after the instruction is executed).

13.1.3.1 Instruction Breakpoints

Instruction breakpoints are set by loading a breakpoint-address register (DRn) with the desired instruction virtual-address, and then setting the corresponding DR7 fields as follows:

- \( L_n \) or \( G_n \) is set to 1 to enable the breakpoint for either the local task or all tasks, respectively.
- \( R/W_n \) is set to 00b to specify that the contents of DRn are to be compared only with the virtual address of the next instruction to be executed.
- \( LEN_n \) must be set to 00b.

When a #DB exception occurs due to an instruction breakpoint-address in DRn, the corresponding Bn field in DR6 is set to 1 to indicate that a breakpoint condition occurred. The breakpoint occurs before
the instruction is executed, and the breakpoint-instruction address is pushed onto the debug-handler stack. If multiple instruction breakpoints are set, the debug handler can use the $B_n$ field to identify which register caused the breakpoint.

Returning from the debug handler causes the breakpoint instruction to be executed. Before returning from the debug handler, the rFLAGS[RF] bit should be set to 1 to prevent a reoccurrence of the #DB exception due to the instruction-breakpoint condition. The processor ignores instruction-breakpoint conditions when rFLAGS[RF] = 1, until after the next instruction (in this case, the breakpoint instruction) is executed. After the next instruction is executed, the processor clears rFLAGS[RF].

### 13.1.3.2 Data Breakpoints

Data breakpoints are set by loading a breakpoint-address register (DR$n$) with the desired data virtual-address, and then setting the corresponding DR7 fields as follows:

- $L_n$ or $G_n$ is set to 1 to enable the breakpoint for either the local task or all tasks, respectively.
- $R/W_n$ is set to 01b to specify that the data virtual-address is compared with the contents of DR$n$ only during a memory-write. Setting this field to 11b specifies that the comparison takes place during both memory reads and memory writes.
- $LEN_n$ is set to 00b, 01b, 11b, or 10b to specify an address-match range of one, two, four, or eight bytes, respectively. Long mode must be active to set $LEN_n$ to 10b.

When a #DB exception occurs due to a data breakpoint address in DR$n$, the corresponding $B_n$ field in DR6 is set to 1 to indicate that a breakpoint condition occurred. The breakpoint occurs after the data-access instruction is executed, which means that the original data is overwritten by the data-access instruction. If the debug handler needs to report the previous data value, it must save that value before setting the breakpoint.

Because the breakpoint occurs after the data-access instruction is executed, the address of the instruction following the data-access instruction is pushed onto the debug-handler stack. Repeated string instructions, however, can trigger a breakpoint before all iterations of the repeat loop have completed. When this happens, the address of the string instruction is pushed onto the stack during a #DB exception if the repeat loop is not complete. A subsequent IRET from the #DB handler returns to the string instruction, causing the remaining iterations to be executed. Most implementations cannot report breakpoints exactly for repeated string instructions, but instead report the breakpoint on an iteration later than the iteration where the breakpoint occurred.

### 13.1.3.3 I/O Breakpoints

I/O breakpoints are set by loading a breakpoint-address register (DR$n$) with the I/O-port address to be trapped, and then setting the corresponding DR7 fields as follows:

- $L_n$ or $G_n$ is set to 1 to enable the breakpoint for either the local task or all tasks, respectively.
- $R/W_n$ is set to 10b to specify that the I/O-port address is compared with the contents of DR$n$ only during execution of an I/O instruction. This encoding of $R/W_n$ is valid only when debug extensions are enabled (CR4[DE] = 1).
• LEN\(n\) is set to 00b, 01b, or 11b to specify the breakpoint occurs on a byte, word, or doubleword I/O operation, respectively.

The I/O-port address specified by the I/O instruction is zero extended by the processor to 64 bits before comparing it with the DR\(n\) registers.

When a #DB exception occurs due to an I/O breakpoint in DR\(n\), the corresponding B\(n\) field in DR6 is set to 1 to indicate that a breakpoint condition occurred. The breakpoint occurs after the instruction is executed, which means that the original data is overwritten by the breakpoint instruction. If the debug handler needs to report the previous data value, it must save that value before setting the breakpoint.

Because the breakpoint occurs after the instruction is executed, the address of the instruction following the I/O instruction is pushed onto the debug-handler stack, in most cases. In the case of INS and OUTS instructions that use the repeat prefix, however, the breakpoint occurs after the first iteration of the repeat loop. When this happens, the I/O-instruction address can be pushed onto the stack during a #DB exception if the repeat loop is not complete. A subsequent return from the debug handler causes the next I/O iteration to be executed. If the breakpoint condition is still set, the #DB exception reoccurs after that iteration is complete.

13.1.3.4 Task-Switch Breakpoints

Breakpoints can be set in a task TSS to raise a #DB exception after a task switch. Software enables a task breakpoint by setting the T bit in the TSS to 1. When a task switch occurs into a task with the T bit set, the processor completes loading the new task state. Before the first instruction is executed, the #DB exception occurs, and the processor sets DR6[BT] to 1, indicating that the #DB exception occurred as a result of task breakpoint.

The processor does not clear the T bit in the TSS to 0 when the #DB exception occurs. Software must explicitly clear this bit to disable the task breakpoint. Software should never set the T-bit in the debug-handler TSS if a separate task is used for #DB exception handling, otherwise the processor loops on the debug handler.

13.1.3.5 General-Detect Condition

General-detect is a special debug-exception condition that occurs when software running at any privilege level attempts to access any of the DR\(n\) registers while DR7[GD] is set to 1. When a #DB exception occurs due to the general-detect condition, the processor clears DR7[GD] and sets DR6[BD] to 1. Clearing DR7[GD] allows the debug handler to access the DR\(n\) registers without causing infinite #DB exceptions.

A debugger enables general detection to prevent other software from accessing and interfering with the debug registers while they are in use by the debugger. The exception is taken before executing the MOV DR\(n\) instruction so that the DR\(n\) contents are not altered.
13.1.4 Single Stepping

Single-step breakpoints are enabled by setting the rFLAGS[TF] bit to 1. TF may be set by the IRET, POPF or SYSRET instructions, with an IRET executed by a debugger being the typical use case. When IRET sets TF, it causes a #DB exception to be taken immediately after the target of the IRET is executed, returning control to the debugger and thereby single-stepping the target instruction. Setting TF with a POPF instruction also causes a one-instruction delayed #DB exception. When TF is set by SYSRET however, a #DB exception is taken before the target instruction executes, hence SYSRET does not provide the single-stepping behavior of IRET.

When a #DB exception occurs due to single stepping, the processor clears rFLAGS[TF] before entering the debug handler, so that the debug handler itself is not single stepped. The processor also sets DR6[BS] to 1, which indicates that the #DB exception occurred as a result of single stepping. The rFLAGS image pushed onto the debug-handler stack has the TF bit set, and single stepping resumes when a subsequent IRET pops the stack image into the rFLAGS register.

Single-step breakpoints have a higher priority than external interrupts. If an external interrupt occurs during single stepping, control is transferred to the #DB handler first, causing the rFLAGS[TF] bit to be cleared. Next, before the first instruction in the debug handler is executed, the processor transfers control to the pending-interrupt handler. This allows external interrupts to be handled outside of single-step mode.

The INTn, INT3, and INTO instructions clear the rFLAGS[TF] bit when they are executed. If a debugger is used to single-step software that contains these instructions, it must emulate them instead of executing them.

The single-step mechanism can also be set to single step only control transfers, rather than single step every instruction. See “Single Stepping Control Transfers” on page 369 for additional information.

13.1.5 Breakpoint Instruction (INT3)

The INT3 instruction, or the INTn instruction with an operand of 3, can be used to set breakpoints that transfer control to the breakpoint-exception (#BP) handler rather than the debug-exception handler. When a debugger uses the breakpoint instructions to set breakpoints, it does so by replacing the first bytes of an instruction with the breakpoint instruction. The debugger replaces the breakpoint instructions with the original-instruction bytes to clear the breakpoint.

INT3 is a single-byte instruction while INTn with an operand of 3 is a two-byte instruction. The instructions have slightly different effects on the breakpoint exception-handler stack. See “#BP—Breakpoint Exception (Vector 3)” on page 226 for additional information on this exception.

13.1.6 Control-Transfer Breakpoint Features

A control transfers is accomplished by using one of following instructions:

- JMP, CALL, RET
- Jcc, JrCXZ, LOOPcc
- JMPF, CALLF, RETF
- INTn, INT 3, INTO, ICEBP
- Exceptions, IRET
- SYSCALL, SYSRET, SYSENTER, SYSEXIT
- INTR, NMI, SMI, RSM

### 13.1.6.1 Recording Control Transfers

Software enables control-transfer recording by setting DebugCtl[LBR] to 1. When this bit is set, the processor updates the recording MSRs automatically when control transfers occur:

- **LastBranchFromIP and LastBranchToIP Registers**—On branch instructions, the LastBranchFromIP register is loaded with the segment offset of the branch instruction, and the LastBranchToIP register is loaded with the first instruction to be executed after the branch. On interrupts and exceptions, the LastBranchFromIP register is loaded with the segment offset of the interrupted instruction, and the LastBranchToIP register is loaded with the offset of the interrupt or exception handler.

- **LastIntFromIP and LastIntToIP Registers**—The processor loads these from the LastBranchFromIP register and the LastBranchToIP register, respectively, when most interrupts and exceptions are taken. These two registers are not updated, however, when #DB or #MC exceptions are taken, or the ICEBP instruction is executed.

The processor automatically disables control-transfer recording when a debug exception (#DB) occurs by clearing DebugCtl[LBR] to 0. The contents of the control-transfer recording MSRs are not altered by the processor when the #DB occurs. Before exiting the debug-exception handler, software can set DebugCtl[LBR] to 1 to re-enable the recording mechanism.

Debuggers can trace a control transfer backward from a bug to its source using the recording MSRs and the breakpoint-address registers. The debug handler does this by updating the breakpoint registers from the recording MSRs after a #DB exception occurs, and restarting the program. The program takes a #DB exception on the previous control transfer, and this process can be repeated. The debug handler cannot simply copy the contents of the recording MSR into the breakpoint-address register. The recording MSRs hold segment offsets, while the debug registers hold virtual (linear) addresses. The debug handler must calculate the virtual address by reading the code-segment selector (CS) from the interrupt-handler stack, then reading the segment-base address from the CS descriptor, and adding that base address to the offset in the recording MSR. The calculated virtual-address can then be used as a breakpoint address.

### 13.1.6.2 Single Stepping Control Transfers

Software can enable control-transfer single stepping by setting DebugCtl[BTF] to 1 and rFLAGS[TF] to 1. The processor automatically disables control-transfer single stepping when a debug exception (#DB) occurs by clearing DebugCtl[BTF]. rFLAGS[TF] is also cleared when a #DB exception occurs. Before exiting the debug-exception handler, software must set both DebugCtl[BTF] and rFLAGS[TF] to 1 to restart single stepping.
When enabled, this single-step mechanism causes a #DB exception to occur on every branch instruction, interrupt, or exception. Debuggers can use this capability to perform a “coarse” single step across blocks of code (bound by control transfers), and then, as the problem search is narrowed, switch into a “fine” single-step mode on every instruction (DebugCtl[BT] = 0 and rFLAGS[TF] = 1).

Debuggers can use both the single-step mechanism and recording mechanism to support full backward and forward tracing of control transfers.

### 13.1.7 Debug Breakpoint Address Masking

The Breakpoint Address Extension feature extends the DR[0-3] breakpoint capabilities. Processors with this extension support address mask registers corresponding to each of the DR[0-3] breakpoint registers, in the form of DR[0-3]_ADDR_MASK MSRs. These masks may be used to increase the range of breakpoints by excluding address bits from the breakpoint match. Each bit set to one excludes the corresponding address bit from the breakpoint comparison. Mask bits 11:0 may be used to expand instruction fetch breakpoint ranges up to a 4KB page, while mask bits 31:12 have no effect on instruction breakpoints. For DR0 only, the full mask field (31:0) may be used to qualify data breakpoint matches. This extension is signified by CPUID Fn8000_0001_ECX[26]=1.

An additional extension expands the data breakpoint masking capability of DR0 to the other breakpoint registers, and extends instruction breakpoint masking to bits 31:0 for all registers. This is signified by CPUID Fn8000_0001_ECX[26]=1.

### 13.2 Performance Monitoring Counters

The AMD64 architecture supports a set of hardware-based performance-monitoring counters (PMCs) that can be utilized to measure the frequency or duration of certain hardware events. MSRs allow the selection of events to be monitored and include a set of corresponding counter registers that accumulate a count of monitored events.

Software tools can use these counters to identify performance bottlenecks, such as sections of code that have high cache-miss rates or frequently mispredicted branches. This information can then be used as a guide for improving overall performance or eliminating performance problems through software optimizations or hardware-design improvements.

Software performance analysis tools often require a means to time-stamp an event or measure elapsed time between two events. The time-stamp counter provides this capability. See Section 13.2.4 “Time-Stamp Counter” on page 377.

The registers used in support of performance monitoring are model-specific registers (MSRs). See “Model-Specific Registers (MSRs)” on page 58 for a general discussion of MSRs and “Performance-Monitoring MSRs” on page 613 for a listing of the performance-monitoring MSR numbers and their reset values.
13.2.1 Performance Counter MSRs

The legacy architecture defines four performance counters (PerfCtrn) and corresponding event-select registers (PerfEvtSeln). Extensions add northbridge and L2 cache performance monitoring counters. Each *PerfCtr register counts events selected by the corresponding *PerfEvtSel register.

An architectural extension augments the number of performance and event-select registers by adding two more processor counter / event-select pairs. Further extensions add four counter / event-select pairs dedicated to counting northbridge (NB) events and four counter / event-select pairs dedicated to counting L2 cache (L2I) events.

Core logic includes instruction execution pipelines, execution units, and caches closest to the execution hardware. The NB includes logic that routes data traffic between caches, external I/O devices, and a system memory controller which reads and writes system memory (usually implemented as external DRAM). The L2 cache is a cache that is further away from the processor core than the L1 cache or caches. This cache is normally larger than the L1 cache(s) and requires more processor cycles to access. An L2 cache may be shared between physical processor cores.

All implementations support the base set of four performance counter / event-select pairs. Support for the extended performance monitoring registers and the performance-related events selectable via the *PerfEvtSel registers vary by implementation and are described in the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual for that processor.

Core performance counters are used to count processor core events, such as data-cache misses, or the duration of events, such as the number of clocks it takes to return data from memory after a cache miss. During event counting, hardware increments a counter each time it detects an occurrence of a specified event. During duration measurement, hardware counts the number of processor clock cycles required to complete a specific hardware function.

NB performance counters are used to count events that occur within the northbridge. The L2I performance counters are used to count events associated with accessing the L2 cache.

Performance counters and event-select registers are implemented as machine-specific registers (MSRs). The base set of four PerfCtr and PerfEvtSel registers are accessed via a legacy set of MSRs and the extended set of six core PerfCtr / PerfEvtSel registers are accessed via a different set. Extended core PerfCtr / PerfEvtSel registers 0–3 alias the legacy set.

Support for the extended set of core PerfCtr registers and associated PerfEvtSel registers, as well as the sets of northbridge and L2 cache counter / event-select pairs are indicated by CPUID feature bits. See “Detecting Hardware Support for Performance Counters” on page 377. The MSR address assignments for the legacy and extended performance / event-select pairs are listed in Appendix A, Section A.6, “Performance-Monitoring MSRs” on page 613.

The length, in bits, of the performance counters is implementation-dependent, but the maximum length supported is 64 bits. Figure 13-6 shows the format of the performance counter registers.
For a given processor, all implemented performance counter registers can be read and written by system software running at CPL = 0 using the RDMSR and WRMSR instructions, respectively. The architecture also provides an instruction, RDPMC, which may be employed by user-mode software to read the architected core, northbridge, and L2 performance counters.

The RDPMC instruction loads the contents of the architected performance counter register specified by the index value contained in the ECX register, into the EDX register and the EAX register. The high 32 bits are returned in EDX, and the low 32 bits are returned in EAX. RDPMC can be executed only at CPL = 0, unless system software enables use of the instruction at all privilege levels. RDPMC can be enabled for use at all privilege levels by setting CR4[PCE] (the performance-monitor counter-enable bit) to 1. When CR4[PCE] = 0 and CPL > 0, attempts to execute RDPMC result in a general-protection exception (#GP). For more information on the RDPMC instruction, see the instruction reference page in Volume 3 of this manual.

Writing the performance counters can be useful if software wants to count a specific number of events, and then trigger an interrupt when that count is reached. An interrupt can be triggered when a performance counter overflows (see “Counter Overflow” on page 377 for additional information). Software should use the WRMSR instruction to load the count as a two’s-complement negative number into the performance counter. This causes the counter to overflow after counting the appropriate number of times.

The performance counters are not guaranteed to produce identical measurements each time they are used to measure a particular instruction sequence, and they should not be used to take measurements of very small instruction sequences. The RDPMC instruction is not serializing, and it can be executed out-of-order with respect to other instructions around it. Even when bound by serializing instructions, the system environment at the time the instruction is executed can cause events to be counted before the counter value is loaded into EDX:EAX. The following sections describe the core performance event-select and the northbridge performance event-select registers.

**Core Performance Event-Select Registers**

The core performance event-select registers (PerfEvtSel[n]) are 64-bit registers used to specify the events counted by the core performance counters, and to control other aspects of their operation. Each performance counter supported by the implementation has a corresponding event-select register that controls its operation. Figure 13-7 below shows the format of the core PerfEvtSel register.
The fields shown in Figure 13-7 above are further described below:

- HG_ONLY (Host/Guest Only): read/write. This field qualifies events to be counted based on virtualization operating mode (guest or host). The following table defines how HG_ONLY qualifies the counting of events:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:42</td>
<td>—</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>41:40</td>
<td>HG_ONLY</td>
<td>Host/Guest Only</td>
<td>R/W</td>
</tr>
<tr>
<td>39:36</td>
<td>—</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>31:24</td>
<td>CNT_MASK</td>
<td>Counter Mask</td>
<td>R/W</td>
</tr>
<tr>
<td>23</td>
<td>INV</td>
<td>Invert Comparison</td>
<td>R/W</td>
</tr>
<tr>
<td>22</td>
<td>EN</td>
<td>Counter Enable</td>
<td>R/W</td>
</tr>
<tr>
<td>21</td>
<td>—</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>INT</td>
<td>Interrupt Enable</td>
<td>R/W</td>
</tr>
<tr>
<td>19</td>
<td>—</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>EDGE</td>
<td>Edge Detect</td>
<td>R/W</td>
</tr>
<tr>
<td>17</td>
<td>OS</td>
<td>Operating-System Mode</td>
<td>R/W</td>
</tr>
<tr>
<td>16</td>
<td>USR</td>
<td>User Mode</td>
<td>R/W</td>
</tr>
<tr>
<td>15:8</td>
<td>UNIT_MASK</td>
<td>Unit Mask</td>
<td>R/W</td>
</tr>
<tr>
<td>7:0</td>
<td>EVENT_SELECT[7:0]</td>
<td>Event select bits 7:0</td>
<td>R/W</td>
</tr>
</tbody>
</table>

**Figure 13-7. Core Performance Event-Select Register (PerfEvtSeln)**
• CNT_MASK (Counter Mask): read/write. Used with INV bit to control the counting of multiple events that occur within one clock cycle. The table below describes this:

<table>
<thead>
<tr>
<th>CNT_MASK</th>
<th>INV</th>
<th>Increment Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>–</td>
<td>Corresponding PerfCtr[n] register is incremented by the number of events occurring in a clock cycle. If the number of events is equal to or greater than 32, the count register is incremented by 32.</td>
</tr>
<tr>
<td>FFh:01h¹</td>
<td>0</td>
<td>Corresponding PerfCtr[n] register is incremented by 1, if the number of events occurring in a clock cycle is greater than or equal to the CNT_MASK value.</td>
</tr>
<tr>
<td>FFh:01h¹</td>
<td>1</td>
<td>Corresponding PerfCtr[n] register is incremented by 1, if the number of events occurring in a clock cycle is less than the CNT_MASK value.</td>
</tr>
</tbody>
</table>

Note 1: Maximum CNT_MASK value (in the range FFh:01h is implementation dependent. Consult applicable BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual.

• INV (Invert Comparison): read/write. Used with CNT_MASK field to control the counting of multiple events within one clock cycle. See table above.

• EN (Counter Enable): read/write. Software sets this bit to 1 to enable the PerfEvtSel[n] register, and counting in the corresponding PerfCtr[n] register. Clearing this bit to 0 disables the register pair.

• INT (Interrupt Enable): read/write. Software sets this bit to 1 to enable an interrupt to occur when the performance counter overflows (see “Counter Overflow” on page 377 for additional information). Clearing this bit to 0 disables the triggering of the interrupt.

• EDGE (Edge Detect): read/write. Software sets this bit to 1 to count the number of edge transitions from the negated to asserted state. This feature is useful when coupled with event-duration monitoring, as it can be used to calculate the average time spent in an event. Clearing this bit to 0 disables edge detection.

• OS (Operating-System Mode) and USR (User Mode): read/write. Software uses these bits to control the privilege level at which event counting is performed according to Table 13-5.

<table>
<thead>
<tr>
<th>OS (Bit 17)</th>
<th>USR (Bit 16)</th>
<th>Event Counting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No counting.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Only at CPL &gt; 0.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Only at CPL = 0.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>At all privilege levels.</td>
</tr>
</tbody>
</table>

• UNIT_MASK (Unit Mask): read/write. This field further specifies or qualifies the event specified by the EVENT_SELECT field. Depending on implementation, it may be used to specify a sub-event within the class specified by the EVENT_SELECT field or it may act as bit mask and be used to specify a number of events within the class to be monitored simultaneously.
• EVENT_SELECT[7:0] (Event Select [7:0]): read/write. This field concatenated with
EVENT_SELECT[11:8] specifies the event or event duration to be counted by the corresponding
PerfCtr[n] register. The events that can be monitored are implementation dependent. In some
implementations, support for a specific EVENT_SELECT value may restricted to a subset of the
available performance counters. For more information, see the BIOS and Kernel Developer’s
Guide (BKDG) or Processor Programming Reference Manual applicable to your product.

The core performance event-select registers can be read and written only by system software running
at CPL = 0 using the RDMSR and WRMSR instructions, respectively. Any attempt to read or write
these registers at CPL > 0 causes a general-protection exception to occur.

Northbridge (NB) Performance Event-Select Registers

The NB performance event-select registers (NB_PerfEvtSeln) are 64-bit registers used to specify the
events counted by the northbridge performance counters, and to control other aspects of their
operation. Each performance counter supported by the implementation has a corresponding event-
select register that controls its operation. Figure 13-8 below shows the format of the NB_PerfEvtSeln
register.

![Northbridge Performance Event-Select Register (NB_PerfEvtSeln)]

The northbridge performance event-select registers can be read and written only by system software running
at CPL = 0 using the RDMSR and WRMSR instructions, respectively. Any attempt to read or write
these registers at CPL > 0 causes a general-protection exception to occur.
For more information on the defined fields within the NB_PerfEvtSel$n registers, see the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual applicable to your product.

**L2 Cache (L2I) Performance Event-Select Registers**

The L2 cache performance event-select registers (L2I_PerfEvtSel$n) are 64-bit registers used to specify the events counted by the L2 cache performance counters, and to control other aspects of their operation. Each performance counter supported by the implementation has a corresponding event-select register that controls its operation. Figure 13-9 below shows the format of the L2I_PerfEvtSel$n register.

![Figure 13-9. L2 Cache Performance Event-Select Register (L2I_PerfEvtSel$n)](image)

The L2 cache performance event-select registers can be read and written only by system software running at CPL = 0 using the RDMSR and WRMSR instructions, respectively. Any attempt to read or write these registers at CPL > 0 causes a general-protection exception to occur.

For more information on the defined fields within the L2I_PerfEvtSel$n registers, see the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual applicable to your product.

**Instructions Retired Performance counter**

This is a dedicated counter that is always counting instructions retired. It exists at MSR address C000_00E9. It is enabled by setting a 1 to HWCR[30] and its support is indicated by CPUID Fn8000_0008_EBX[1].
13.2.2 Detecting Hardware Support for Performance Counters

Support for extended core, northbridge, and L2 cache performance counters is implementation-dependent. Support on a given processor implementation can be verified using the CPUID instruction.

CPUID Fn8000_0001_ECX[PerfCtrExtCore] = 1 indicates support for the six architecturally defined extended core performance counters and their associated event-select registers. CPUID Fn8000_0001_ECX[PerfCtrExtNB] = 1 indicates support for the four architecturally defined northbridge performance counter / event-select pairs and CPUID Fn8000_0001_ECX[PerfCtrExtL2I] = 1 indicates support for the four architecturally defined L2 cache performance counter / event-select pairs.

See Section 3.3, “Processor Feature Identification,” on page 64 for more information on using the CPUID instruction.

A given processor may implement other performance measurement MSRs with similar capabilities even if one of the optional architected facilities are not.

13.2.3 Using Performance Counters

13.2.3.1 Starting and Stopping

Performance measurement using the PerfCtrn, NB_PerfCtrn, and L2I_PerfCtrn registers is initiated by setting the corresponding *PerfEvtSeln[EN] bit to 1. Counting is stopped by clearing the *PerfEvtSeln[EN] bit. Software must initialize the remaining *PerfEvtSeln fields with the appropriate setup information before or at the same time EN is set. Counting begins when the WRMSR instruction that sets *PerfEvtSeln[EN] to 1 completes execution. Counting stops when the WRMSR instruction that clears the EN bit completes execution.

13.2.3.2 Counter Overflow

Some processor implementations support an interrupt-on-overflow capability that allows an interrupt to occur when one of the *PerfCtrn registers overflows. The source and type of interrupt is implementation dependent. Some implementations cause a debug interrupt to occur, while others make use of the local APIC to specify the interrupt vector and trigger the interrupt when an overflow occurs. Software enables or disables the triggering of an interrupt on counter overflow by setting or clearing the *PerfEvtSeln[INT] bit.

If system software makes use of the interrupt-on-overflow capability, an interrupt handler must be provided that can record information relevant to the counter overflow. Before returning from the interrupt handler, the performance counter can be re-initialized to its previous state so that another interrupt occurs when the appropriate number of events are counted.

13.2.4 Time-Stamp Counter

The time-stamp counter (TSC) is used to count processor-clock cycles. The TSC is cleared to 0 after a processor reset. After a reset, the TSC is incremented at a rate corresponding to the baseline frequency
of the processor (which may differ from actual processor frequency in low power modes of operation). Each time the TSC is read, it returns a monotonically-larger value than the previous value read from the TSC. When the TSC contains all ones, it wraps to zero. The TSC in a 1-GHz processor counts for almost 600 years before it wraps. Figure 13-10 shows the format of the 64-bit time-stamp counter (TSC).

![Time-Stamp Counter (TSC)](image)

**Figure 13-10. Time-Stamp Counter (TSC)**

The TSC is a model-specific register that can also be read using one of the special *read time-stamp counter* instructions, RDTSC (Read Time-Stamp Counter) or RDTSCP (Read Time-Stamp Counter and Processor ID). The RDTSC and RDTSCP instructions load the contents of the TSC into the EDX register and the EAX register. The high 32 bits are loaded into EDX, and the low 32 bits are loaded into EAX. The RDTSC and RDTSCP instructions can be executed at any privilege level and from any processor mode. However, system software can disable the RDTSC or RDTSCP instructions for programs that run at CPL > 0 by setting CR4[TSD] (the *time-stamp disable* bit) to 1. When CR4[TSD] = 1 and CPL > 0, attempts to execute RDSTC or RDSTCP result in a general-protection exception (#GP).

The TSC register can be read and written using the RDMSR and WRMSR instructions, respectively. The programmer should use the CPUID instruction to determine whether these features are supported. If EDX bit 4 (as returned by CPUID function 1) is set, then the processor supports TSC, the RDTSC instruction and CR4[TSD]. If EDX bit 27 returned by CPUID function 8000_0001h is set, then the processor supports the RDTSCP instruction.

The TSC register can be used by performance-analysis applications, along with the performance-monitoring registers, to help determine the relative frequency of an event or its duration. Software can also use the TSC to time software routines to help identify candidates for optimization. In general, the TSC should not be used to take very short time measurements, because the resulting measurement is not guaranteed to be identical each time it is made. The RDTSC instruction (unlike the RDTSCP instruction) is not serializing, and can be executed out-of-order with respect to other instructions around it. Even when bound by serializing instructions, the system environment at the time the instruction is executed can cause additional cycles to be counted before the TSC value is loaded into EDX:EAX.

When using the TSC to measure elapsed time, programmers must be aware that for some implementations, the rate at which the TSC is incremented varies based on the processor power management state (Pstate). For other implementations, the TSC increment rate is fixed and is not subject to power-management related changes in processor frequency. CPUID Fn 8000_0007h_EDX[TscInvariant] = 1 indicates that the TSC increment rate is a constant.
For more information on using the CPUID instruction to obtain processor implementation information, see Section 3.3, “Processor Feature Identification,” on page 64.

13.3 Instruction-Based Sampling

Instruction-Based Sampling (IBS) is a hardware facility that can be used to gather specific metrics related to processor instruction fetch and instruction execution activity. Data capture is performed by hardware at a sampling interval specified by values programmed in IBS sampling control registers. The IBS facility can be utilized by software to perform code profiling based on statistical sampling.

There are two independent data gathering components of IBS: instruction fetch sampling and instruction execution sampling. Instruction fetch sampling provides information about instruction address translation look-aside buffer (ITLB) and instruction cache behavior for a randomly selected fetch block, under the control of the IBS Fetch Control Register. Instruction execution sampling provides information about instruction execution behavior by tracking the execution of a single micro-operation (op) that is randomly selected, under the control of the IBS Execution Control Register.

When the programmed interval for fetch sampling has expired, the fetch sampling component of IBS selects and tags the next fetch block. IBS hardware records specific performance information about the tagged fetch. In a similar manner, when the programmed interval for op sampling has expired, the op sampling component of IBS selects and tags the next op being dispatched for execution.

When data collection for the tagged fetch or op is complete, the hardware signals an interrupt. An interrupt handler can then read the performance information that was captured for the fetch or op in IBS MSRs, save it, and re-enable the hardware to take the next sample.

More information about the IBS facility and how software can use it to perform code profiling can be found in the Software Optimization Guide for your specific product. The Software Optimization Guide for AMD Family 15h Processors is order #47414.

Support for the IBS feature is indicated by the CPUID Fn 8000_0001h_ECX[IBS]. For more information on using the CPUID instruction to obtain processor implementation information, see Section 3.3, “Processor Feature Identification,” on page 64.

13.3.1 IBS Fetch Sampling

When a processor fetches an instruction, it is actually reading a contiguous range of instruction bytes that contains the instruction from memory or from cache. This range of bytes loaded by the processor in one operation is called a fetch block. The size and address-alignment characteristics of the fetch block are implementation-dependent. In the following discussion, the term instruction fetch or simply fetch refers to this operation of reading a fetch block.

Instruction fetch sampling records the following performance information for each tagged fetch:

- If the fetch completed or was aborted
- The number of core clock cycles spent on the fetch
• If the fetch hit or missed the instruction cache
• If the instruction fetch hit or missed the instruction TLBs
• The fetch address translation page size
• The linear and physical address associated with the fetch

IBS selects and tags a fetch at a programmable rate. When enabled by the IBS Fetch Control Register (IbsFetchEn = 1 and IbsFetchVal = 0), an internal 20-bit fetch interval counter increments for every successful completion of a fetch operation. When the value in bits 19:4 of the fetch counter equal the value in the IbsFetchMaxCnt field of the IBS Fetch Control Register, the next fetch block is tagged for data collection.

When the tagged fetch completes or is aborted, the status of the fetch is written to the IBS Fetch Control Register and the associated linear address and physical address are written in the IBS Fetch Linear Address Register and IBS Fetch Physical Address Register, respectively. The IbsFetchVal bit is set in the IBS Fetch Control Register and an interrupt is generated as specified by the local APIC.

The interrupt service routine saves the performance information stored in the IBS fetch registers. Software can then initiate another sample by resetting the IbsFetchVal bit in the IBS Fetch Control Register. Hardware initializes bits 19:4 of the internal fetch interval counter with the value in the IbsFetchCnt field. If the IbsFetchCtl[IbsRandEn] bit is set, bits 3:0 of the fetch interval counter are re-initialized by hardware with a pseudo-random value; otherwise bits 3:0 are cleared.

### 13.3.2 IBS Fetch Sampling Registers

The IBS fetch sampling registers consist of the status and control register (IBS Fetch Control Register) and the associated fetch address registers (IBS Fetch Linear Address Register and IBS Fetch Physical Address Register). The IBS fetch sampling registers are accessed using the RDMSR and WRMSR instructions.

#### IBS Fetch Control Register

<table>
<thead>
<tr>
<th>Bit 63</th>
<th>Bit 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved, MBZ</td>
<td>IbsFetchLat</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 58</th>
<th>Bit 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>IbsRandEn</td>
<td>IbsFetchCnt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 57</th>
<th>Bit 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>IbsL1TlbMiss</td>
<td>IbsFetchMaxCnt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 56</th>
<th>Bit 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>IbsL1TlbPgsz</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 55</th>
<th>Bit 14</th>
</tr>
</thead>
<tbody>
<tr>
<td>IbsPhyAddrValid</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 54</th>
<th>Bit 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>IbsMiss</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 53</th>
<th>Bit 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>IbsFetchComp</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 52</th>
<th>Bit 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>IbsFetchVal</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 51</th>
<th>Bit 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>IbsFetchEn</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 50</th>
<th>Bit 9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 49</th>
<th>Bit 8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 48</th>
<th>Bit 7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 47</th>
<th>Bit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 46</th>
<th>Bit 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 45</th>
<th>Bit 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 44</th>
<th>Bit 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 43</th>
<th>Bit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 42</th>
<th>Bit 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 41</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The fields shown in Figure 13-11 are further described below:

- **IbsRandEn (IBS Randomize Tagging Enable)—**Bit 57, read/write. Software sets this bit to 1 to add variability to the interval at which fetch operations are selected for tagging. When set, bits 3:0 of the fetch interval counter are set to a pseudo-random value when the IbsFetchCtl register is written. Clearing this bit causes bits 3:0 of the fetch interval counter to be reset to zero.

- **IbsL1TlbMiss (IBS Fetch L1 TLB Miss)—**Bit 55, read/write. This bit is set if the tagged fetch missed in the L1 TLB.

- **IbsL1TlbPgSz[1:0] (IBS Fetch L1 TLB Page Size)—**Bits 54:53, read/write. This field indicates the page size of the translation in the L1 TLB for the tagged fetch. This field is valid only if IbsPhyAddrVal = 1. The table below defines the encoding of this two-bit field:

<table>
<thead>
<tr>
<th>Value</th>
<th>Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00b</td>
<td>4 Kbyte</td>
</tr>
<tr>
<td>01b</td>
<td>2 Mbyte</td>
</tr>
<tr>
<td>10b</td>
<td>1 Gbyte</td>
</tr>
<tr>
<td>11b</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Some implementations might not support all page sizes. Note: The page size in the L1 TLB might not match the page size in the page table.

- **IbsPhyAddrValid (IBS Fetch Physical Address Valid)—**Bit 52, read/write. This bit is set if the physical address of the tagged fetch is valid. When this bit is set, the IbsL1TlbPgSz field and the contents of the IBS Fetch Physical Address Register (see definition of this register below) are both valid.

- **IbsIcMiss (IBS Instruction Cache Miss)—**Bit 51, read/write. This bit is set if the tagged fetch missed in the instruction cache.
• IbsFetchComp (IBS Fetch Complete)—Bit 50, read/write. This bit is set if the tagged fetch completes and data is available for use by the instruction decoder.

• IbsFetchVal (IBS Fetch Valid)—Bit 49, read/write. This bit is set if the tagged fetch either completes or is aborted. When the bit is set, captured data for the tagged fetch is available and the fetch interval counter stops. An interrupt is generated as specified by the local APIC. The interrupt handler should read and save the captured performance data before clearing the bit.

• IbsFetchEn (IBS Fetch Enable)—Bit 48, read/write. Software sets this bit to enable fetch sampling. Clearing this bit to 0 disables fetch sampling.

• IbsFetchLat[15:0] (IBS Fetch Latency)—Bits 47:32, read/write. This 16-bit field indicates the number of core clock cycles from the initiation of the fetch to the delivery of the instruction bytes to the core. If the fetch is aborted before it completes, this field returns the number of clock cycles from the initiation of the fetch to its abortion.

• IbsFetchCnt[15:0] (IBS Fetch Count)—Bits 31:16, read/write. This 16-bit field returns the current value of bits 19:4 of the fetch interval counter on a read. Bits 19:4 of the fetch interval counter are set to this value on a write.

• IbsFetchMaxCnt[15:0] (IBS Fetch Maximum Count)—Bits 15:0, read/write. This 16-bit field specifies the maximum count value of bits 19:4 of the fetch interval counter. When the value in bits 19:4 of the fetch counter equals the value in this field, the next fetch block is tagged for profiling.

### IBS Fetch Linear Address Register

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Field Mnemonic</th>
<th>Descriptive Name</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:0</td>
<td>IbsFetchLinAd</td>
<td>IBS Fetch Linear Address</td>
<td>RO</td>
</tr>
</tbody>
</table>

**Figure 13-12. IBS Fetch Linear Address Register (IbsFetchLinAd)**

This is a read-only register. Reading the IbsFetchLinAd MSR returns the 64-bit linear address of the tagged fetch. This address may correspond to the first byte of an AMD64 instruction or the start of the fetch block. The address is valid only if the IbsFetchVal bit is set. The address is in canonical form.
IBS Fetch Physical Address Register

This is a read-only register. Reading the IbsFetchPhysAd MSR returns the 52-bit physical address of the tagged fetch. This address may correspond to the first byte of an AMD64 instruction or the start of the fetch block. The address is valid only if both the IbsPhyAddrValid and the IbsFetchVal bits of the IbsFetchCtl register are set. Otherwise, the contents of this register are undefined. The indicated size of 52 bits is an architectural limit. Specific processors may implement fewer bits.

13.3.3 IBS Execution Sampling

Instruction execution performance is measured by tagging an op associated with an instruction. The tagged op joins other ops in a queue waiting to be dispatched and executed. Instructions that decode to more than one op may return different performance data depending upon which op associated with the instruction is tagged. IBS returns the following performance information for each retired tagged op:

- Branch status for branch ops.
- For a load or store op:
  - Whether the load or store missed in the data cache.
  - Whether the load or store address hit or missed in the TLBs.
  - The linear and physical address of the data operand associated with the load or store operation.
  - Source information for cache, DRAM, MMIO, or I/O accesses.

IBS selects and tags an op at a programmable rate. When enabled by the IBS Execution Control Register (IbsOpEn = 1 and IbsOpVal = 0), an internal 27-bit op interval counter increments either once for every core clock cycle, if IbsOpCntCtl is cleared, or once for every dispatched op, if IbsOpCntCtl is set.

When the value in bits 26:4 of the op counter equals the value in the IbsOpMaxCnt field of the IBS Execution Control Register, an op is tagged in the next cycle. When the op is retired, the execution status of the op is written to the IBS execution registers, and IbsOpVal bit of the IBS Execution Control
Register is set. When this is complete, an interrupt is signalled to the local APIC. The local APIC should be programmed to deliver this interrupt to the processor core.

The interrupt service routine must save the performance information stored in IBS execution registers. Software can then initiate another sample by resetting the IbsOpVal bit in the IBS Execution Control Register.

Aborted ops do not produce an IBS execution sample. If the tagged op aborts (i.e., does not retire), hardware resets bits 26:7 of the op interval counter to zero, and bits 6:0 to a random value. The op counter continues to increment and another op is selected when the value in bits 26:4 of the op interval counter equals the value in the IbsOpMaxCnt field.

**Randomization of sampling interval:** A degree of randomization of the sampling interval is necessary to ensure fairness in sampling, especially for loop-intensive code. For execution sampling this must be done by software. This is accomplished when writing the IbsOpCtl register to clear the IbsOpVal bit and initiate a new sampling interval. At that time software can provide a small random number (4-6 bits) in the IbsOpCurCnt field to offset the starting count, thereby randomizing the point at which the count reaches the IbsOpMaxCnt value and triggers a sample.

### 13.3.4 IBS Execution Sampling Registers

The IBS execution sampling registers consist of the control register (IBS Execution Control Register), the linear address register (IBS Op Linear Address Register), and three execution data registers (IBS Op Data 1–3). The IBS execution sampling registers are accessed using the RDMSR and WRMSR instructions.

**IBS Execution Control Register (IbsOpCtl)**

<table>
<thead>
<tr>
<th>63</th>
<th>59</th>
<th>58</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved, MBZ</td>
<td>IbsOpCurCnt[26:0]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>27</th>
<th>26</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
</table>
The fields shown in Figure 13-14 are further described below:

- **IbsOpCurCnt[26:0]** (IBS Op Current Count)—Bits 58:32, read/write. This field returns the current value of the op counter, and provides the starting value of the counter when software writes this register to clear the IbsOpVal bit and start another sampling interval.

- **IbsOpMaxCnt[26:20]** (IBS Op Maximum Count[26:20])—Bits 26:20, read/write. This field is used to specify the most significant 7 bits of the IbsOpMaxCnt.

- **IbsOpCntCtl** (IBS Op Counter Control)—Bit 19, read/write. This bit controls op tagging. When this bit is zero, IBS counts core clock cycles in order to select an op for tagging. When this bit is one, IBS counts dispatched ops in order to select an op for tagging.

- **IbsOpVal** (IBS Op Sample Valid)—Bit 18, read/write. This bit is set when a tagged op retires and indicates that new instruction execution data is available. The op counter stops counting. An interrupt is generated as specified by the local APIC. The software interrupt handler captures the performance data before clearing the bit to enable the hardware to take another sample.

- **IbsOpEn** (IBS Op Sampling Enable)—Bit read/write. Software sets this bit to enable IBS execution sampling. Clearing this bit disables IBS execution sampling.

- **IbsOpMaxCnt[19:4]** (IBS Op Maximum Count[19:4]): read/write. This field specifies the maximum count value for bits 19:4 of the op interval counter. When the value in bits 26:4 of the op interval counter equal the value specified by the concatenation of the IbsOpMaxCnt[26:20] field with this field, the next op is tagged for profiling.
IBS Op Linear Address Register (IbsOpRip)

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Field Mnemonic</th>
<th>Descriptive Name</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:0</td>
<td>IbsOpRip</td>
<td>IBS Op Linear Address</td>
<td>R/W</td>
</tr>
</tbody>
</table>

Figure 13-15. IBS Op Linear Address Register (IbsOpRip)

IbsOpRip[63:0] (IBS Op Linear Address): read/write. Specifies the linear address for the instruction from which the tagged op was issued. The address is valid only if the IbsOpCtl[IbsOpVal] bit is set and the IbsOpData1[IbsRipInvalid] bit is cleared. The address is in canonical form.
IBS Op Data 1 Register (IbsOpData1)

The IBS Op Data 1 Register provides core cycle counts for tagged ops and performance data for tagged ops which perform a branch.

### Figure 13-16.  IBS Op Data 1 Register (IbsOpData1)

The fields shown in Figure 13-16 are further described below:

- **IbsRipInvalid (IbsOpRip Register Invalid)**—Bit 38, read/write. If this bit is set, the contents of the IbsOpRip register are not valid.
- **IbsOpBrnRet (IBS Op Branch.Retired)**—Bit 37, read/write. This bit is set if the tagged op performs a branch that retired.
- **IbsOpBrnMisp (IBS Op Branch.Mispredicted)**—Bit 36, read/write. This bit is set if the tagged op performs a retired mispredicted branch.
- **IbsOpBrnTaken (IBS Op Branch.Taken)**—Bit 35, read/write. This bit is set if the tagged op performs a retired taken branch.
- **IbsOpReturn (IBS Op RET)**—Bit 34, read/write. This bit is set if the tagged op performs a retired subroutine return (RET).
IBS Op Data 2 Register (IbsOpData2)

The IBS Op Data 2 Register captures northbridge-related performance data. The information captured is implementation-dependent. See the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product for details.
IBS Op Data 3 Register (IbsOpData3)

Data Cache (first-level cache) performance data is captured in IBS Op Data 3 Register. If a load or store operation crosses a 128-bit boundary, the data returned in this register is the data for the access to the data below the 128-bit boundary.

![IBS Op Data 3 Register Diagram](image)

The fields shown in Figure 13-17 are further described below:
- IbsDcMissLat[15:0] (IBS Data Cache Miss Latency)—Bits 47:32, read/write. This field indicates the number of core clock cycles from when a miss is detected in the data cache to when the data is delivered to the core. The value is not valid for data cache store operations.

- IbsDePhyAddrValid (IBS Data Cache Physical Address Valid)—Bit 18, read/write. This bit is set if the physical address in the IBS DC Physical Address Register is valid for a load or store operation.

- IbsDcLinAddrValid (IBS Data Cache Linear Address Valid)—Bit 17, read/write. This bit is set if the linear address in the IBS DC Linear Address Register is valid for a load or store operation.

- IbsDcLockedOp (IBS Data Cache Locked Op)—Bit 15, read/write. This bit is set if the tagged load or store operation was a locked operation.

- IbsDcUcMemAcc (IBS Data Cache UC Memory Access)—Bit 14, read/write. This bit is set if the tagged load or store operation accessed uncacheable memory.

- IbsDcWcMemAcc (IBS Data Cache WC Memory Access)—Bit 13, read/write. This bit is set if the tagged load or store operation accessed write combining memory.

- IbsDcMisAcc (IBS Data Cache Misaligned Access Penalty)—Bit 8, read/write. This bit is set if a tagged load or store operation incurred a performance penalty due to a misaligned access.

- IbsDcMiss (IBS Data Cache Miss)—Bit 7, read/write. This bit is set if the cache line used by the tagged load or store operation was not present in the data cache.

- IbsDcL1tlbHit1G (IBS Data Cache L1 TLB Hit 1-Gbyte Page)—Bit 5, read/write. This bit is set if the physical address for the tagged load or store operation was present in a 1-Gbyte page table entry in the data cache L1 TLB.

- IbsDcL1tlbHit2M (IBS Data Cache L1 TLB Hit 2-Mbyte Page)—Bit 4, read/write. This bit is set if the physical address for the tagged load or store operation was present in a 2-Mbyte page table entry in the data cache L1 TLB.

- IbsDcL1tlbMiss (IBS Data Cache L1 TLB Miss)—Bit 2, read/write. This bit is set if the physical address for the tagged load or store operation was not present in the data cache L1 TLB.

- IbsStOp (IBS Store Op)—Bit 1, read/write. This bit is set if the tagged op was a store.

- IbsLdOp (IBS Load Op)—Bit 0, read/write. This bit is set if the tagged op was a load.

**IBS Data Cache Linear Address Register (IbsDcLinAd)**

<table>
<thead>
<tr>
<th>63</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>IbsDcLinAd[63:32]</td>
<td></td>
</tr>
</tbody>
</table>
**Figure 13-18. IBS Data Cache Linear Address Register (IbsDcLinAd)**

IbsDcLinAd[63:0] (IBS Data Cache Linear Address): read/write. Specifies the linear address of the tagged op's memory operand. The address is valid only if IbsOpData3[IbsDcLinAdVal] is set. The address is in canonical form.

**IBS Data Cache Physical Address Register (IbsDcPhysAd)**

IbsDcPhysAd (IBS Data Cache Physical Address): read/write. Specifies the physical address of the tagged op's memory operand. The address is valid only if IbsOpData3[IbsDcPhysAdVal] is set.

**Figure 13-19. IBS Data Cache Physical Address Register (IbsDcPhysAd)**

IbsDcPhysAd (IBS Data Cache Physical Address): read/write. Specifies the physical address of the tagged op's memory operand. The address is valid only if IbsOpData3[IbsDcPhysAdVal] is set.
IbBrTarget (IBS Branch Target): read/write. Specifies the 64-bit linear address for the branch target. The address is in canonical form. The branch target address is valid if it is non-zero. For a conditional branch not taken, the value supplied in this register is the fall-through address.

## 13.4 Lightweight Profiling

Lightweight Profiling (LWP) is an AMD64 extension that allows user mode processes to gather performance data about themselves with very low overhead. LWP is supported in both long mode and legacy mode. Modules such as managed runtime environments and dynamic optimizers can use LWP to monitor the running program with high accuracy and high resolution. They can quickly discover performance problems and opportunities and immediately act on this information.

LWP allows a program to gather performance data and examine it either by polling or by taking an occasional interrupt. It introduces minimal additional state to the CPU and the process. LWP differs from the existing performance counters and from Instruction Based Sampling (IBS) because it collects large quantities of data before taking an interrupt. This substantially reduces the overhead of using performance feedback. An application can avoid the need to enable and process interrupts by polling the LWP data.

A program can control LWP data collection entirely in user mode. It can start, stop, and reconfigure profiling without calling the kernel.

LWP runs within the context of a thread, so it can be used by multiple processes in a system at the same time without interference. This also means that if one thread is using LWP and another is not, the latter thread incurs no profiling overhead.

LWP can be programmed to run in one of two modes: synchronized mode or continuous mode. In synchronized mode the recording of events stops when the buffer set up to hold event records becomes full. In continuous mode, the storing of events wraps in the buffer overwriting older records.

### 13.4.1 Overview

When enabled, LWP hardware monitors one or more events during the execution of user-mode code and periodically inserts event records into a ring buffer in the address space of the running process. If performance timestamping is supported and enabled, each event record captured is timestamped using
the value read from the performance timestamp counter (PTSC). Timestamping is enabled by setting the Flags.PTSC bit of the Lightweight Profiling Control Block (LWPCB). When the ring buffer is filled beyond a user-specified threshold, the hardware can cause an interrupt which the operating system (OS) uses to signal a process to empty the ring buffer. With proper OS support, the interrupt can even be delivered to a separate process or thread.

LWP only counts instructions that retire in user mode (CPL = 3). Instructions that change to CPL 3 from some other level are not counted, since the instruction address is not an address in user mode space. LWP does not count hardware events while the processor is in system management mode (SMM) and while entering or leaving SMM.

Once LWP is enabled, each user-mode thread uses the LLWPCB and SLWPCB instructions to control LWP operation. These instructions refer to a data structure in application memory called the Lightweight Profiling Control Block, or LWPCB, to specify the profiling parameters and to interact with the LWP hardware. The LWPCB in turn points to a buffer in memory in which LWP stores event records.

Each thread in a multi-threaded process must configure LWP separately. A thread has its own ring buffer and counters which are context switched with the rest of the thread state. However, a single monitor thread could collect and process LWP data from multiple other threads.

LWP may be set up to run in one of two modes:

- **Synchronized Mode**
  LWP runs in synchronized mode when it is started with LWPCB.Flags.CONT = 0. In this mode, LWP will not advance the ring buffer pointer when the event ring buffer is full. It simply increments LWPCB.MissedEvents to count the number of missed event records. In synchronized mode, a thread can remove event records from the ring buffer by advancing the ring buffer tail pointer without stopping LWP in the executing thread. If the buffer had been full, event records will again be written and the ring buffer pointer will be advanced.

- **Continuous Mode**
  LWP runs in continuous mode when it is started with LWPCB.Flags.CONT = 1. In this mode, LWP will store an event record even when the event ring buffer is full, wrapping around in the ring buffer and overwriting the oldest event record. In continuous mode, LWPCB.MissedEvents counts the number of times that such wrapping has occurred. The only reliable way to read events from the ring buffer when LWP is in continuous mode is to stop LWP in the running thread before accessing the LWPCB and the ring buffer contents. Support for continuous mode is indicated by CPUID Fn8000_001C_EAX[LwpCont].

During profiling, the LWP hardware monitors and reports on one or more types of events. Following are the steps in this process:
1. **Count**—Each time an instruction is retired, LWP decrements its internal event counters for all of the events associated with the instruction. An instruction can cause zero, one, or multiple events. For instance, an indirect jump through a pointer in memory counts as an instruction retired, a branch retired, and may also cause up to two DCache misses (or more, if there is a TLB miss) and up to two ICache misses.

   - Some events may have filters or conditions on them that regulate counting. For instance, the application may configure LWP so that only cache miss events with latency greater than a specified minimum are eligible to be counted.

2. **Gather**—When an event counter becomes negative, the event should be reported. LWP gathers an event record and, if enabled, samples the value in the PTSC to be included in the record as the TimeStamp value. The event’s counter may continue to count below zero until the record is written to the event ring buffer.

   For most events, such as instructions retired, LWP gathers an event record describing the instruction that caused the counter to become negative. However, it is valid for LWP to gather event record data for the next instruction that causes the event, or to take other measures to capture a record. Some of these options are described with the individual events.

   - An implementation can choose to gather event information on one or many events at any one time. If multiple event counters become negative, an advanced LWP implementation might gather one event record per event and write them sequentially. A basic LWP implementation may choose one of the eligible events. Other events continue counting but wait until the first event record is written. LWP picks the next eligible instructions for the waiting events. This situation should be extremely uncommon if software chooses large event interval values.

   - LWP may discard an event occurrence. For instance, if the LWPCB or the event ring buffer needs to be paged in from disk, LWP might choose not to preserve the pending event data. If an event is discarded, LWP gathers an event record for the next instruction to cause the event.

   - Similarly, if LWP needs to replay an instruction to gather a complete event record, the replay may abort instead of retiring. The event counter continues counting below zero and LWP gathers an event record for the next instruction to cause the event.

3. **Store**—When a complete event record is gathered, LWP stores it into the event ring buffer in the process’ address space and advances the ring buffer head pointer.

   - LWP checks to see if the ring buffer is full, i.e., if advancing the ring buffer head pointer would make it equal to the tail pointer. If the buffer is full, LWP increments the 64-bit counter LWPCB.MissedEvents. If LWP is running in synchronized mode, it does not advance the head pointer. If LWP is running in continuous mode, it always advances the head pointer and LWPCB.MissedEvents counts the number of times that the buffer wrapped.

   - If more than one event record reaches the Store stage simultaneously, only one need be stored. Though LWP might store all such event records, it may delay storing some event records or it may discard the information and proceed to choose the next eligible instruction for the discarded event type(s). This behavior is implementation dependent.

   - The store need not complete synchronously with the instruction retiring. In other words, if LWP buffers the event record contents, the Store stage (and subsequent stages) may complete
some number of cycles after the tagged instruction retires. The data about the event and the instruction are precise, but the Report and Reset steps (below) may complete later.

4. **Report**—If LWP threshold interrupts are enabled and the space used in the event ring buffer exceeds a user-defined threshold, LWP initiates an interrupt. The OS can use this to signal the process to empty the ring buffer. Note that the interrupt may occur significantly later than the event that caused the threshold to be reached.

5. **Reset**—For each event that was stored, the counter is reset to its programmed interval. If requested by the application, LWP applies randomization to the low order bits of the interval. Counting for that event continues. Reset happens if the ring buffer head pointer was advanced or if the missed event counter was incremented. If the event counter went below -1, indicating that additional events occurred between the selected event and the time it was reported, that overrun value reduces the reset value so as to preserve the statistical distribution of events.

For all events except the LWPVAL instruction, the hardware may impose a minimum on the reset value of an event counter. This prevents the system from spending too much time storing samples rather than making forward progress on the application. Any minimum imposed by the hardware can be detected by examining the EventInterval fields in the LWPCB after enabling LWP.

An application should periodically remove event records from the ring buffer and advance the tail pointer. (If the application does not process the event records quickly enough or often enough, the LWP hardware will detect that the ring buffer is full and will miss events.) There are two ways to process the gathered events: interrupts or polling.

The application can wait until a threshold interrupt occurs to process the event records in the ring buffer. This requires OS or driver support. (As a consequence, interrupts can only be enabled if a kernel mode routine allows it; see “LWP_CFG—LWP Configuration MSR” on page 410) One usage model is to associate the LWP interrupt with a semaphore or mutex. When the interrupt occurs, the OS or driver signals the associated object. A thread waiting on the object wakes up and empties the ring buffer. Other models are possible, of course.

Alternatively, the application can have a thread that periodically polls the ring buffer. The polling thread need not be part of the process that is using LWP. It can be in a separate process that shares the memory containing the LWP control block and ring buffer.

Access to the ring buffer uses a lockless protocol between the LWP hardware and the application. The hardware owns the head pointer and the area in the ring buffer from the head pointer up to (but not including) the tail pointer. The application must not modify the head pointer nor rely on any data in the area of the ring buffer owned by the hardware. If the application has a stale value for the head pointer, it may miss an existing event record but it will never read invalid data. When the application is done emptying the ring buffer, it should refresh its copy of the head pointer to see if the LWP hardware has added any new event records.

Similarly, the application owns the tail pointer and the area in the ring buffer from the tail pointer up to (but not including) the head pointer. The hardware will never modify the tail pointer or overwrite the data in that region of the ring buffer. If the hardware has a stale value for the tail pointer, it may
consider that the ring buffer is full or at its threshold, but it will never overwrite valid data. Instead, it refreshes its copy of the tail pointer and rechecks to see if the full or threshold condition still applies.

When LWP is in continuous mode, this lockless protocol does not work, since the LWP hardware may overwrite the event records in the ring buffer when it advances the head pointer past the tail pointer. Because of this, the application must stop LWP before removing event records from the ring buffer. This prevents the hardware from wrapping through the ring buffer asynchronously from the application’s attempt to remove data from it.

To use continuous mode properly, the application should set LWPCB.MissedEvents to 0 and set the head and tail pointers to the start of the ring buffer before starting LWP. To empty the ring buffer, the application should stop LWP. If LWPCB.MissedEvents is 0, the buffer did not wrap and there are event records starting at the tail pointer and continuing up to (but not including) the head pointer. If MissedEvents is not 0, the buffer wrapped and there are event records starting with the oldest one pointed to by the head pointer and continuing (possibly wrapping) all the way around to the newest one just before the head pointer.

13.4.2 Events and Event Records

When a monitored event overflows its event counter, LWP puts an event record into the LWP event ring buffer. If event timestamping is supported and enabled, each event record will include a TimeStamp value. This value is a copy of the contents of Performance Timestamp Counter (PTSC) zero-extended if necessary to 64 bits.

The PTSC is a free-running counter that increments at a constant rate of 100MHz and is synchronized across all cores on a node to within +/-1. This counter starts when the processor is initialized and cannot be reset or modified. It is at least 40 bits wide. Privileged code can read the PTSC value via the RDMSR instruction. The size of the counter is indicated by the 2-bit field CPUID Fn8000_0008_ECX[PerfTscSize]. A value of 00b means that the PTSC is 40 bits wide; 01b means 48 bits, 10b means 56 bits, and 11b indicates a full 64 bits.

The PTSC can be correlated to the architectural TSC that runs at the P0 frequency. An application can read the TSC and PTSC, wait a 1000 clock periods or so, then read them again. The ratio of the differences is the scaling factor for the counters.

The event record size is fixed but may vary based on implementation. The event record size for a given processor is discovered by executing CPUID Fn8000_001C and extracting the value of the LwpEventSize field. (See “Detecting LWP Capabilities” on page 407). Current implementations fix the record size at 32 bytes and this size is used in the record format specifications below.

Reserved fields and fields that are not defined for a particular event are set to zero when LWP writes an event record.
Table 13-6 below lists the event identifiers for the events defined in version 1 of LWP. They are described in detail in the following sections.

Table 13-6. EventId Values

<table>
<thead>
<tr>
<th>EventId</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved – invalid event</td>
</tr>
<tr>
<td>1</td>
<td>Programmed value sample</td>
</tr>
<tr>
<td>2</td>
<td>Instructions retired</td>
</tr>
<tr>
<td>3</td>
<td>Branches retired</td>
</tr>
<tr>
<td>4</td>
<td>DCache misses</td>
</tr>
</tbody>
</table>
13.4.2.1 Programmed Value Sample

LWP decrements the event counter each time the program executes the LWPVAL instruction (see “LWPVAL—Insert Value Sample in LWP Ring Buffer” on page 415). When the counter becomes negative, it stores an event record with an EventId of 1. The data in the event record come from the operands to the instruction as detailed in the instruction description.

![Figure 13-22. Programmed Value Sample Event Record](image)

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>EventId</td>
<td>Event identifier = 1</td>
</tr>
<tr>
<td>1</td>
<td>CoreId</td>
<td>CPU core identifier from LWP_CFG</td>
</tr>
<tr>
<td>3–2</td>
<td>Flags</td>
<td>Immediate value (bottom 16 bits)</td>
</tr>
<tr>
<td>7–4</td>
<td>Data1</td>
<td>Reg/mem value</td>
</tr>
<tr>
<td>15–8</td>
<td>InstructionAddress</td>
<td>Instruction address of LWPVAL instruction</td>
</tr>
<tr>
<td>23–16</td>
<td>Data2</td>
<td>Reg value (zero extended if running in legacy mode)</td>
</tr>
<tr>
<td>31–24</td>
<td>TimeStamp</td>
<td>Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.</td>
</tr>
</tbody>
</table>

13.4.2.2 Instructions Retired

LWP decrements the event counter each time an instruction retires. When the counter becomes negative, it stores a generic event record with an EventId of 2.

Instructions are counted if they execute entirely in user mode (CPL = 3). Instructions that change to CPL 3 from some other level are not counted, since the instruction address is not an address in user mode space. All user mode instructions are counted, including LWPVAL and LWPINS.
LWP decrements the event counter each time a transfer of control retires, regardless of whether or not it is taken. When the counter becomes negative, it stores an event record with an EventId of 3.

Control transfer instructions that are counted are:

- JMP (near), Jcc, JCXZ, JEXCZ, and JRCXZ
- LOOP, LOOPE, and LOOPNE
- CALL (near) and RET (near)

LWP does not count JMP (far), CALL (far), RET (far), traps, or interrupts (whether synchronous or asynchronous), nor does it count operations that switch to or from ring 3, SMM, or SVM, such as SYSCALL, SYSENTER, SYSEXIT, SYSRET, VMMCALL, INT, or INTO.

Some implementations of the AMD64 architecture perform an optimization called “fusing” when a compare operation (or other operation that sets the condition codes) is followed immediately by a conditional branch. The processor fuses these into a single operation internally before they are executed. While this is invisible to the programmer, the address of the actual branch is not available for LWP to report when the (fused) instruction retires. In this case, LWP sets the FUS bit in the event record and reports the address of the operation that set the condition codes. If FUS is set, software can find the address of the actual branch by decoding the instruction at the reported InstructionAddress and
adding its length to that address. (Note that fused instructions do count as 2 instructions for the Instructions Retired event, since there were 2 x86 instructions originally.)
Figure 13-24. Branch Retired Event Record

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Bits</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7:0</td>
<td>EventId</td>
<td>Event identifier = 3</td>
</tr>
<tr>
<td>1</td>
<td>7:0</td>
<td>CoreId</td>
<td>CPU core identifier from LWP_CFG</td>
</tr>
<tr>
<td>3–2</td>
<td>11:0</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>FUS</td>
<td>1—Fused operation. InstructionAddress points to a compare operation (or other operation that sets the condition codes) immediately preceding the branch. 0—InstructionAddress points to the branch instruction.</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>PRV</td>
<td>1—PRD bit is valid 0—Prediction information is not available Some implementations of LWP may be unable to capture branch prediction information on some or all branches.</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>PRD</td>
<td>1—Branch was predicted correctly 0—Mispredicted If PRV = 0, the value of PRD is unpredictable and should be ignored. For unconditional branches, PRD=1 if PRV=1.</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>TKN</td>
<td>1—Branch was taken 0—Branch not taken Always 1 for unconditional branches.</td>
</tr>
<tr>
<td>7–4</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>15–8</td>
<td></td>
<td>InstructionAddress</td>
<td>Instruction address</td>
</tr>
<tr>
<td>23–16</td>
<td></td>
<td>TargetAddress</td>
<td>Address of instruction executed after branch. This is the target if the branch was taken and the fall-through address if the branch was a not-taken conditional branch. TargetAddress is the Effective Address value before adding in the CS base address.</td>
</tr>
<tr>
<td>31–24</td>
<td></td>
<td>TimeStamp</td>
<td>Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.</td>
</tr>
</tbody>
</table>
13.4.2.4 DCache Misses

LWP decrements the event counter each time a load from memory causes a DCache miss whose latency exceeds the LwpCacheLatency threshold and/or whose data come from a level of the cache or memory hierarchy that is selected for counting. When the counter becomes negative, LWP stores an event record with an EventId of 4.

A misaligned access that causes two misses on a single load decrements the event counter by 1 and, if it reports an event, the data are for the lowest address that missed. LWP only counts loads directly caused by the instruction. It does not count cache misses that are indirectly due to TLB walks, LDT or GDT references, TLB misses, etc. Cache misses caused by LWP itself accessing the LWPCB or the event ring buffer are not counted.

Measuring Latency

The x86 architecture allows multiple loads to be outstanding simultaneously. An implementation of LWP might not have a full latency counter for every load that is waiting for a cache miss to be resolved. Therefore, an implementation may apply any of the following simplifications. Software using LWP should be prepared for this.

- The implementation may round the latency to a multiple of $2^j$. This is a small power of 2, and the value of $j$ must be 1 to 4. For example, in the rest of this section, assume that $j = 4$, so $2^4 = 16$. The low 4 bits of latency reported in the event record will be 0. The actual latency counter is incremented by 16 every 16 cycles of waiting. The value of $j$ is returned as LwpLatencyRnd (see “Detecting LWP Capabilities” on page 407).

- The implementation may do an approximation when starting to count latency. If counting is in increments of 16, the 16 cycles need not start when the load begins to wait. The implementation may bump the latency value from 0 to 16 any time during the first 16 cycles of waiting.

- The implementation may cap total latency to $2^n-16$ (where $n >= 10$). The latency counter is thus a saturating counter that stops counting when it reaches its maximum value. For example, if $n = 10$, the latency value will count from 0 to 1008 in steps of 16 and then stop at 1008. (If $n = 10$, each counter is only 6 bits wide.) The value of $n$ is returned as LwpLatencyMax (see “Detecting LWP Capabilities” on page 407).

Note that the latency threshold used to filter events is a multiple of 16. This value is used in the comparison that decides whether a cache miss event is eligible to be counted.

Reporting the DCache Miss Data Address

The event record for a DCache miss reports the linear address of the data (after adding in the segment base address, if any). The way an implementation records the linear address affects the exact event that is reported and the amount of time it takes to report a cache miss event. The implementation may report the event immediately, report the next eligible event once the counter becomes negative, or replay the instruction.
### 13.4.2.5 CPU Clocks not Halted

LWP decrements the event counter each clock cycle that the CPU is not in a halted state (due to STPCLK or a HLT instruction). When the counter becomes negative, it stores a generic event record with an EventId of 5. This counter varies in real-time frequency as the core clock frequency changes.

---

#### Figure 13-25. DCache Miss Event Record

<table>
<thead>
<tr>
<th>Byte 7</th>
<th>Byte 6</th>
<th>Byte 5</th>
<th>Byte 4</th>
<th>Byte 3</th>
<th>Byte 2</th>
<th>Byte 1</th>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>Reserved</td>
<td>CoreId</td>
<td>EventId=4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>InstructionAddress</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataAddress</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TimeStamp</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Bits</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7:0</td>
<td>EventId</td>
<td>Event identifier = 4</td>
</tr>
<tr>
<td>1</td>
<td>7:0</td>
<td>CoreId</td>
<td>CPU identifier from LWP_CFG</td>
</tr>
<tr>
<td>2–3</td>
<td>11:0</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>DAV</td>
<td>1—DataAddress is valid  0—Address is unavailable</td>
</tr>
<tr>
<td>3</td>
<td>5:7</td>
<td>SRC</td>
<td>Data source for the requested data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0—No valid status  1—Local L3 cache  2—Remote CPU or L3 cache  3—DRAM  4—Reserved (for Remote cache)  5—Reserved  6—Reserved  7—Other (MMIO/Config/PCI/API)</td>
</tr>
<tr>
<td>7–4</td>
<td>Latency</td>
<td>Total latency of cache miss (in cycles)</td>
<td></td>
</tr>
<tr>
<td>15–8</td>
<td>InstructionAddress</td>
<td>Instruction address</td>
<td></td>
</tr>
<tr>
<td>23–16</td>
<td>DataAddress</td>
<td>Address of memory reference (if flag bit 28 = 1)</td>
<td></td>
</tr>
<tr>
<td>31–24</td>
<td>TimeStamp</td>
<td>Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.</td>
<td></td>
</tr>
</tbody>
</table>
13.4.2.6 CPU Reference Clocks not Halted

LWP decrements the event counter each reference clock cycle that the CPU is not in a halted state (due to STPCLK or a HLT instruction). When the counter becomes negative, it stores a generic event record with an EventId of 6.

The reference clock runs at a constant frequency that is independent of the core frequency and of the performance state. The reference clock frequency is processor dependent. The processor may implement this event by subtracting the ratio of (reference clock frequency / core clock frequency) each core clock cycle.

---

**Figure 13-26. CPU Clocks not Halted Event Record**

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Bits</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7:0</td>
<td>EventId</td>
<td>Event identifier = 5</td>
</tr>
<tr>
<td>1</td>
<td>7:0</td>
<td>CoreId</td>
<td>CPU identifier from LWP_CFG</td>
</tr>
<tr>
<td>7–2</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>15–8</td>
<td></td>
<td>InstructionAddress</td>
<td>Instruction address</td>
</tr>
<tr>
<td>23–16</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>31–24</td>
<td></td>
<td>TimeStamp</td>
<td>Performance Time Stamp Counter value if LWP was started with LWPCB.Flags.PTSC = 1, zero otherwise.</td>
</tr>
</tbody>
</table>

---
Figure 13.27. CPU Reference Clocks not Halted Event Record

13.4.2.7 Programmed Event

When a program executes the LWPINS instruction (see “LWPINS—Insert User Event Record in LWP Ring Buffer” on page 416), the processor stores an event record with an event identifier of 255. The data in the event record come from the operands to the instruction as detailed in the instruction description.
13.4.2.8 Other Events

The overall design of LWP allows easy extension to the list of events that it can monitor. The following are possibilities for events that may be added in future versions of LWP:

- DTLB misses
- FPU operations
- ICache misses
- ITLB misses

13.4.3 Detecting LWP

An application uses the CPUID instruction to identify whether Lightweight Profiling is present and which of its capabilities are available for use. An operating system uses CPUID to determine whether LWP is supported on the hardware and to determine which features of LWP are supported and can be made available to applications.
13.4.3.1 Detecting LWP Presence

LWP is supported on a processor if CPUID Fn8000_0001_ECX[LWP] (bit 15) is set. This bit is identical to the value of CPUID Fn0000_000D_EDX_x0[bit 30], which is bit 62 of the XFeatureSupportedMask and indicates XSAVE support for LWP. A system can check either of those bits to determine if LWP is supported. Since LWP requires XSAVE, software can assume that this bit being set implies that CPUID Fn0000_0001_ECX[XSAVE] (bit 26) is also set.

13.4.3.2 Detecting LWP XSAVE Area

The size of the LWP extended state save area used by XSAVE/XRSTOR is 128 bytes (080h). This value is returned by CPUID Fn0000_000D_EAX_x3E (ECX=62).

The offset of the LWP save area from the beginning of the XSAVE/XRSTOR area is 832 bytes (340h). This value is returned by CPUID Fn0000_000D_EBX_x3E (ECX=62).

The size of the LWP save area is included in the XFeatureSupportedSizeMax value returned by CPUID Fn0000_000D_ECX_x0 (ECX=0).

If LWP is enabled in the XFEATURE_ENABLED_MASK, the size of the LWP save area is included in the XFeatureEnabledSizeMax value returned by CPUID Fn0000_000D_EBX_x0 (ECX=0).

13.4.3.3 Detecting LWP Capabilities

The values returned by CPUID Fn8000_001C indicate the capabilities of LWP. See Table 13-7, “Lightweight Profiling CPUID Values” for a listing of the returned values.

Bit 0 of EAX is a copy of bit 62 from XFEATURE_ENABLED_MASK and indicates whether LWP is available for use by applications. If it is 1, the processor supports LWP and the operating system has enabled LWP for applications.

Bits 31:1 returned in EAX are taken from the LWP_CFG MSR and reflect the LWP features that are available for use. These are a subset of the bits returned in EDX, which reflect the full capabilities of LWP on current processor. The operating system can make a subset of LWP available if it cannot handle all supported features. For instance, if the OS cannot handle an LWP threshold interrupt, it can disable the feature. User-mode software must assume that the bits in EAX describe the features it can use. Operating systems should use the bits from EDX to determine the supported capabilities of LWP and make all or some of those features available.

Under SVM, if a VMM allows the migration of guests among processors that all support LWP, it must arrange for CPUID to report the logical AND of the supported feature bits over all processors in the migration pool. Other CPUID values must also be reported as the “least common denominator” among the processors.
### Table 13-7. Lightweight Profiling CPUID Values

<table>
<thead>
<tr>
<th>Reg</th>
<th>Bits</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>0</td>
<td>LwpAvail</td>
<td>1—LWP is available to application programs. The hardware and the operating system support LWP. 0—LWP is not available. This bit is a copy of bit 62 of the XFEATURE_ENABLED_MASK register (XCR0).</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>LwpVAL</td>
<td>LWPVAL instruction (EventId = 1) is available.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>LwpIRE</td>
<td>Instructions retired event (EventId = 2) is available.</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>LwpBRE</td>
<td>Branch retired event (EventId = 3) is available.</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>LwpDME</td>
<td>DCache miss event (EventId = 4) is available.</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>LwpCNH</td>
<td>CPU clocks not halted event (EventId = 5) is available.</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>LwpRNH</td>
<td>CPU reference clocks not halted event (EventId = 6) is available.</td>
</tr>
<tr>
<td></td>
<td>28:7</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>LwpCont</td>
<td>Sampling in continuous mode is available.</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>LwpPTSC</td>
<td>Performance Time Stamp Counter in event records is available.</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>LwpInt</td>
<td>Interrupt on threshold overflow is available.</td>
</tr>
<tr>
<td>EBX</td>
<td>7:0</td>
<td>LwpCbSize</td>
<td>Size in quadwords of the LWPCB. This value is at least (LwpEventOffset / 8) + LwpMaxEvents but an implementation may require a larger control block.</td>
</tr>
<tr>
<td></td>
<td>15:8</td>
<td>LwpEventSize</td>
<td>Size in bytes of an event record in the LWP event ring buffer. (32 for LWP Version 1.)</td>
</tr>
<tr>
<td></td>
<td>23:16</td>
<td>LwpMaxEvents</td>
<td>Maximum supported EventId value (not including EventId 255 used by the LWPINS instruction). Not all events between 1 and LwpMaxEvents are necessarily supported.</td>
</tr>
<tr>
<td></td>
<td>31:24</td>
<td>LwpEventOffset</td>
<td>Offset from the start of the LWPCB to the EventInterval1 field. Software uses this value to locate the area of the LWPCB that describes events to be sampled. This permits expansion of the initial fixed region of the LWPCB. LwpEventOffset is always a multiple of 8.</td>
</tr>
</tbody>
</table>
Table 13-7. Lightweight Profiling CPUID Values

<table>
<thead>
<tr>
<th>Reg</th>
<th>Bits</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:0</td>
<td>LwpLatencyMax</td>
<td>Number of bits in cache latency counters (10 to 31). 0 if DCache miss event is not supported (EDX[LwpDME] = 0).</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>LwpDataAddress</td>
<td>1—Cache miss event records report the data address of the reference. 0—Data address is not reported. 0 if DCache miss event is not supported (EDX[LwpDME] = 0).</td>
<td></td>
</tr>
<tr>
<td>8:6</td>
<td>LwpLatencyRnd</td>
<td>The amount by which cache latency is rounded. The bottom LwpLatencyRnd bits of latency information will be zero. The actual number of bits implemented for the counter is (LwpLatencyMax – LwpLatencyRnd). Must be 0 to 4. 0 if DCache miss event is not supported (EDX[LwpDME] = 0).</td>
<td></td>
</tr>
<tr>
<td>15:9</td>
<td>LwpVersion</td>
<td>Version of LWP implementation. (1 for LWP Version 1.)</td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td>LwpMinBufferSize</td>
<td>Minimum size of the LWP event ring buffer, in units of 32 event records. At least 32 * LwpMinBufferSize records must be allocated for the LWP event ring buffer, and hence the size of the ring buffer must be at least 32 * LwpMinBufferSize * LwpEventSize bytes. If 0, there is no minimum.</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>LwpBranchPrediction</td>
<td>1—Branches Retired events can be filtered based on whether the branch was predicted properly. The values of NMB and NPB in the LWPCB enable filtering based on prediction. 0—NMB and NPB fields of the LWPCB are ignored. 0 if Branches Retired event is not supported (EDX[LwpBRE] = 0).</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>LwpIpFiltering</td>
<td>1—IP filtering is supported. 0—IP filtering is not supported. The IPI, IPF, BaseIP, and LimitIP fields of the LWPCB are ignored.</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>LwpCacheLevels</td>
<td>1—Cache-related events can be filtered by the cache level that returned the data. The value of CLF in the LWPCB enables cache level filtering. 0—CLF is ignored. An implementation must support filtering either by latency or by cache level. It may support both. 0 if DCache miss event is not supported (EDX[LwpDME] = 0).</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>LwpCacheLatency</td>
<td>1—Cache-related events can be filtered by latency. The value of MinLatency in the LWPCB controls filtering. 0—MinLatency is ignored. An implementation must support filtering either by latency or by cache level. It may support both. 0 if DCache miss event is not supported (EDX[LwpDME] = 0).</td>
<td></td>
</tr>
</tbody>
</table>
13.4.4  LWP Registers

The XFEATURE_ENABLED_MASK register (extended control register XCR0) and the LWP model-specific registers describe and control the LWP hardware. The MSRs are available if CPUID Fn8000_0001_ECX[LWP] (bit 15) is set. LWP can only be used if the system has made support for LWP state management available in XFEATURE_ENABLED_MASK.

13.4.4.1  XFEATURE_ENABLED_MASK Support

LWP requires that the processor support the XSAVE/XRSTOR instructions to manage LWP state, along with the XSETBV/XGETBV instructions that manage the enabled state mask. An operating system uses XSETBV to set bit 62 of XFEATURE_ENABLED_MASK to indicate that it supports management of LWP state and allows applications to use LWP. When the system makes LWP available by setting bit 62 of XFEATURE_ENABLED_MASK, LWP is initially disabled (LWP_CBADDR is zero).

See “Guidelines for Operating Systems” on page 433 for details on how to implement LWP support in an operating system.

13.4.4.2  LWP_CFG—LWP Configuration MSR

LWP_CFG (MSR C000_0105h) controls which features of LWP are available on the processor. The operating system loads LWP_CFG at start-up time (or at the time an LWP driver is loaded) to indicate

<table>
<thead>
<tr>
<th>Reg</th>
<th>Bits</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDX</td>
<td>0</td>
<td>LwpAvail</td>
<td>LWP is supported. If 0, the remainder of the data returned by CPUID should be ignored. This bit is a copy of CPUID Fn8000_0001_ECX[LWP] (bit 15).</td>
</tr>
<tr>
<td>1</td>
<td>LwpVAL</td>
<td>LWPVAL instruction (EventId = 1) is supported.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>LwpIRE</td>
<td>Instructions retired event (EventId = 2) is supported.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>LwpBRE</td>
<td>Branch retired event (EventId = 3) is supported.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>LwpDME</td>
<td>DCache miss event (EventId = 4) is supported.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>LwpCNH</td>
<td>CPU clocks not halted event (EventId = 5) is supported.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>LwpRNH</td>
<td>CPU reference clocks not halted event (EventId = 6) is supported.</td>
<td></td>
</tr>
<tr>
<td>28:7</td>
<td>LwpCont</td>
<td>Sampling in continuous mode is supported.</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>LwpPTSC</td>
<td>Performance Time Stamp Counter in event records is supported.</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>LwpInt</td>
<td>Interrupt on threshold overflow is supported.</td>
<td></td>
</tr>
</tbody>
</table>
its level of support for LWP. Only bits for supported features (those that are set in CPUID Fn8000_001C_EDX) can be turned on in LWP_CFG. Attempting to set other bits causes a #GP fault.

User code can examine LWP_CFG bits 31:1 by reading CPUID Fn8000_001C_EAX.

Bits 39:32 of LWP_CFG contains the COREID value that LWP will store into the CoreId field of every event record written by this core. The operating system should initialize this value to be the local APIC number, obtained from CPUID Fn0000_0001_EBX[LocalApicId] (bits 31:24). COREID is present so that when LWP is used in a virtualized environment, it has access to the core number without needing to enter the hypervisor. On systems that support x2APIC, local APIC numbers may be more than 8 bits wide. The operating system may then assign LWP COREID values that are small and identify the core within a cluster. If the system has more than 256 cores, there will be unavoidable duplication of COREID values.

Bits 47:40 of LWP_CFG specify the vector number that LWP will use when it signals a ring buffer threshold interrupt.

The reset value of LWP_CFG is 0.

Figure 13-29. LWP_CFG—Lightweight Profiling Features MSR
13.4.4.3 **LWP_CBADDR—LWPCB Address MSR**

LWP_CBADDR (MSR C000_0106h) provides access to the internal copy of the LWPCB linear address.

RDMSR from this register returns the current LWPCB address without performing any of the operations described for the SLWPCB instruction.

WRMSR to this register with a non-zero value generates a #GP fault; use LLWPCB or XRSTOR to load an LWPCB address.

Writing a zero to LWP_CBADDR immediately disables LWP, discarding any internal state. For instance, an operating system can write a zero to stop LWP when it terminates a thread.

Note that LWP_CBADDR contains the linear address of the control block. All references to the LWPCB that are made by microcode during the normal operation of LWP ignore the DS segment register.

The reset value of LWP_CBADDR is 0. This means that when the system sets bit 62 of XFEATURE_ENABLED_MASK to make LWP available, it is initially disabled.

13.4.5 **LWP Instructions**

This section describes the instructions included in the AMD64 architecture to support LWP. These instructions raise #UD if LWP is not supported or if bit 62 of XFEATURE_ENABLED_MASK is 0 indicating that LWP is not available.

The LLWPCB instruction enables or disables Lightweight Profiling and controls the events being profiled. The SLWPCB instruction queries the current state of Lightweight Profiling.

LWP provides two instructions for inserting user data into the event ring buffer. The LWPINS instruction unconditionally stores an event record into the ring buffer, while the LWPVAL instruction uses an LWP event counter to sample program values at defined intervals.

The instructions LLWPCB, SLWPCB, LWPINS, and LWPVAL are also described in the chapter "General-Purpose Instruction Reference" of Volume 3. Refer to reference pages for the individual instruction for information on instruction encoding, flags affected, and exception behavior.

13.4.5.1 **LLWPCB—Load LWPCB Address**

 Parses the Lightweight Profiling Control Block at the address contained in the specified register. If the LWPCB is valid, writes the address into the LWP_CBADDR MSR and enables Lightweight Profiling.

The LWPCB must be in memory that is readable and writable in user mode. For better performance, it should be aligned on a 64-byte boundary in memory and placed so that it does not cross a page boundary, though neither of these suggestions is required.
Action

1. If LWP is not available or if the machine is not in protected mode, LLWPCB immediately causes a #UD exception.

2. If LWP is already enabled, the processor flushes the LWP state to memory in the old LWPCB. See “SLWPCB—Store LWPCB Address” on page 414 for details on saving the active LWP state.
   
   If the flush causes a #PF exception, LWP remains enabled with the old LWPCB still active. Note that the flush is done before LWP attempts to access the new LWPCB.

3. If the specified LWPCB address is 0, LWP is disabled and the execution of LLWPCB is complete.

4. The LWPCB address is non-zero. LLWPCB validates it as follows:
   - If any part of the LWPCB or the ring buffer is beyond the data segment limit, LLWPCB causes a #GP exception.
   - If the ring buffer size is below the implementation’s minimum ring buffer size, LLWPCB causes a #GP exception.
   - While doing these checks, LWP reads and writes the LWPCB, which may cause a #PF exception.

   If any of these exceptions occurs, LLWPCB aborts and LWP is left disabled. Usually, the operating system will handle a #PF exception by making the memory available and returning to retry the LLWPCB instruction. The #GP exceptions indicate application programming errors.

5. LWP converts the LWPCB address and the ring buffer address to linear address form by adding the DS base address and stores the addresses internally.

6. LWP examines the LWPCB.Flags field to determine which events should be enabled and whether threshold interrupts should be taken. It clears the bits for any features that are not available and stores the result back to LWPCB.Flags to inform the application of the actual LWP state.

7. For each event being enabled, LWP examines the EventInterval[n] value and, if necessary, sets it to an implementation-defined minimum. (The minimum event interval for LWPVAL is zero.) It loads its internal counter for the event from the value in EventCounter[n]. A zero or negative value in EventCounter[n] means that the next event of that type will cause an event record to be stored. To count every j\textsuperscript{th} event, a program should set EventInterval[n] to j-1 and EventCounter[n] to some starting value (where j-1 is a good initial count). If the counter value is larger than the interval, the first event record will be stored after a larger number of events than subsequent records.

8. LWP is started. The execution of LLWPCB is complete.

Notes

If none of the bits in the LWPCB.Flags specifies an available event, LLWPCB still enables LWP to allow the use of the LWPINS instruction. However, no other event records will be stored.

A program can temporarily disable LWP by executing SLWPCB to obtain the current LWPCB address, saving that value, and then executing LLWPCB with a register containing 0. It can later re-enable LWP by executing LLWPCB with a register containing the saved address.
When LWP is enabled, it is typically an error to execute LLWPCB with the address of the active LWPCB. When the hardware flushes the existing LWP state into the LWPCB, it may overwrite fields that the application may have set to new LWP parameter values. The flushed values will then be loaded as LWP is restarted. To reuse an LWPCB, an application should stop LWP by passing a zero to LLWPCB, then prepare the LWPCB with new parameters and execute LLWPCB again to restart LWP.

Internally, LWP keeps the linear address of the LWPCB and the ring buffer. If the application changes the value of DS, LWP will continue to collect samples even if the new DS value would no longer allows it to access the LWPCB or the ring buffer. However, a #GP fault will occur if the application uses XRSTOR to restore LWP state saved by XSAVE. Programs should avoid using XSAVE/XRSTOR on LWP state if DS has changed. This only applies when the CPL ≠ 0; kernel mode operation of XRSTOR is unaffected by changes to DS. See “XSAVE/XRSTOR” on page 426 for details.

Operating system and hypervisor code that runs when the CPL ≠ 3 should use XSAVE and XRSTOR to control LWP rather than using LLWPCB (see below). Use WRMSR to write 0 to LWP_CBADDR to immediately stop LWP without saving its current state (see “LWP_CBADDR—LWPCB Address MSR” on page 412).

It is possible to execute LLWPCB when the CPL ≠ 3 or when SMM is active, but the system software must ensure that the LWPCB and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Furthermore, if LWP is enabled when a kernel executes LLWPCB, both the old and new control blocks and ring buffers must be accessible. Using LLWPCB in these situations is not recommended.

13.4.5.2 SLWPCB—Store LWPCB Address

Flushes LWP state to memory and returns the current effective address of the LWPCB in the specified register.

If LWP is not currently enabled, SLWPCB sets the specified register to zero.

The flush operation stores the internal event counters for active events and the current ring buffer head pointer into the LWPCB. If there is an unwritten event record pending, it is written to the event ring buffer.

If LWP_CBADDR is not zero, the value returned is an effective address that is calculated by subtracting the current DS.Base address from the linear address kept in LWP_CBADDR. Note that if DS has changed between the time LLWPCB was executed and the time SLWPCB is executed, this might result in an address that is not currently accessible by the application.

SLWPCB generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not available.

It is possible to execute SLWPCB when the CPL ≠ 3 or when SMM is active, but if the LWPCB pointer is not zero, the system software must ensure that the LWPCB and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF fault. Using SLWPCB in these situations is not recommended.
13.4.5.3 LWPVAL—Insert Value Sample in LWP Ring Buffer

Decrement the event counter associated with the Programmed Value Sample event (see “Programmed Value Sample” on page 398). If the resulting counter value is negative, inserts an event record into the LWP event ring buffer in memory and advances the ring buffer pointer. If the counter is not negative and the ModRM operand specifies a memory location, that location is not accessed.

The event record has an EventId of 1. The value in the register specified by vvvv (first operand) is stored in the Data2 field at bytes 23–16 (zero extended if the operand size is 32). The value in a register or memory location (second operand) is stored in the Data1 field at bytes 7–4. The immediate value (third operand) is truncated to 16 bits and stored in the Flags field at bytes 3–2. See Figure 13-22 on page 398.

If the ring buffer is not full or if LWP is running in continuous mode, the head pointer is advanced and the event counter is reset to the interval for the event (subject to randomization). If the ring buffer threshold is exceeded and threshold interrupts are enabled, an interrupt is signaled. If LWP is in continuous mode and the new head pointer equals the tail pointer, the MissedEvents counter is incremented to indicate that the buffer wrapped.

If the ring buffer is full and LWP is running in synchronized mode, the event record overwrites the last record in the buffer, the MissedEvents counter in the LWPCB is incremented, and the head pointer is not advanced.

LWPVAL generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not available.

LWPVAL does nothing if LWP is not enabled or if the Programmed Value Sample event is not enabled in LWPCB.Flags. This allows LWPVAL instructions to be harmlessly ignored if profiling is turned off.

It is possible to execute LWPVAL when the CPL ≠ 3 or when SMM is active, but the system software must ensure that the memory operand (if present), the LWPCB, and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Using LWPVAL in these situations is not recommended.

LWPVAL can be used by a program to perform value profiling. This is the technique of sampling the value of some program variable at a predetermined frequency. For example, a managed runtime might use LWPVAL to sample the value of the divisor for a frequently executed divide instruction in order to determine whether to generate specialized code for a common division. It might sample the target location of an indirect branch or call to see if one destination is more frequent than others. Since LWPVAL does not modify any registers or condition codes, it can be inserted harmlessly between any instructions.

**Note**

When LWPVAL completes (whether or not it stored an event record in the event ring buffer), it counts as an instruction retired. If the Instructions Retired event is active, this might cause that counter to become negative and immediately store an event record. If LWPVAL also stored an event record, the buffer will contain two records with the same instruction address (but different EventId values).
13.4.5.4 LWPINS—Insert User Event Record in LWP Ring Buffer

Inserts a record into the LWP event ring buffer in memory and advances the ring buffer pointer.

The record has an EventId of 255. The value in the register specified by vvvv (first operand) is stored in the Data2 field at bytes 23–16 (zero extended if the operand size is 32). The value in a register or memory location (second operand) is stored in the Data1 field at bytes 7–4. The immediate value (third operand) is truncated to 16 bits and stored in the Flags field at bytes 3–2. See Figure 13-28 on page 406.

If the ring buffer is not full or if LWP is running in continuous mode, the head pointer is advanced and the CF flag is cleared. If the ring buffer threshold is exceeded and threshold interrupts are enabled, an interrupt is signaled. If LWP is in continuous mode and the new head pointer equals the tail pointer, the MissedEvents counter is incremented to indicate that the buffer wrapped.

If the ring buffer is full and LWP is running in synchronized mode, the event record overwrites the last record in the buffer, the MissedEvents counter in the LWPCB is incremented, the head pointer is not advanced, and the CF flag is set.

LWPINS generates an invalid opcode exception (#UD) if the machine is not in protected mode or if LWP is not available.

LWPINS simply clears CF if LWP is not enabled. This allows LWPINS instructions to be harmlessly ignored if profiling is turned off.

It is possible to execute LWPINS when the CPL ≠ 3 or when SMM is active, but the system software must ensure that the memory operand (if present), the LWPCB, and the entire ring buffer are properly mapped into writable memory in order to avoid a #PF or #GP fault. Using LWPINS in these situations is not recommended.

LWPINS can be used by a program to mark significant events in the ring buffer as they occur. For instance, a program might capture information on changes in the process’ address space such as library loads and unloads, or changes in the execution environment such as a change in the state of a user-mode thread of control.

Note that when the LWPINS instruction finishes writing a event record in the event ring buffer, it counts as an instruction retired. If the Instructions Retired event is active, this might cause that counter to become negative and immediately store another event record with the same instruction address (but different EventId values).

13.4.6 LWP Control Block

An application uses the LWP Control Block (LWPCB) to specify the details of Lightweight Profiling operation. It is an interactive region of memory in which some fields are controlled and modified by the LWP hardware and others are controlled and modified by the software that processes the LWP event records.
Most of the fields in the LWPCB are constant for the duration of a LWP session (the time between enabling LWP and disabling it). This means that they are loaded into the LWP hardware when it is enabled, and may be periodically reloaded from the same location as needed. The contents of the constant fields must not be changed during a LWP run or results will be unpredictable. Changing the LWPCB memory to read-only or unmapped will cause an exception the next time LWP attempts to access it. To change values in the LWPCB, disable LWP, change the LWPCB (or create a new one), and re-enable LWP.

A few fields are modified by the LWP hardware to communicate progress to the software that is emptying the event ring buffer. Software may read them but should never modify them during an LWP session. Other fields are for software to modify to indicate that progress has been made in emptying the ring buffer. Software writes these fields and the LWP hardware reads them as needed.

For efficiency, some of the LWPCB fields may be shadowed internally in the LWP hardware unit when profiling is enabled. LWP refreshes these fields from (or flushes them to) memory as needed to allow software to make progress. For more information, refer to “LWPCB Access” on page 432.

The BufferTailOffset field is at offset 64 in the LWPCB in order to place it in a separate cache line on most implementations, assuming that the LWPCB itself is aligned properly. This allows the software thread that is emptying the ring buffer to retain write ownership of that cache line without colliding with the changes made by LWP when writing BufferHeadOffset. In addition, most implementations will use a value of 128 as the offset to the EventInterval1 field, since that places the event information in a separate cache line.

All fields in the LWPCB (as shown in Figure 13-30) that are marked as “Reserved” (or “Rsvd”) should be zero.
The R/W column in Table 13-8 below indicates how a field is used while LWP is enabled:

<table>
<thead>
<tr>
<th>Byte 7</th>
<th>Byte 6</th>
<th>Byte 5</th>
<th>Byte 4</th>
<th>Byte 3</th>
<th>Byte 2</th>
<th>Byte 1</th>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>BufferSize</td>
<td></td>
<td>Flags</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BufferBase</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reserved</td>
<td>BufferHeadOffset</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MissedEvents</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Filters</td>
<td>Threshold</td>
<td></td>
<td></td>
<td></td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BaseIP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LimitIP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>48</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>56</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reserved</td>
<td>BufferTailOffset</td>
<td></td>
<td></td>
<td></td>
<td>64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reserved for software</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>72</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reserved for software</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>80</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td>88</td>
</tr>
</tbody>
</table>

\[ E = LwpEventOffset \]
\[ E + 8 \]
\[ N = LwpMaxEvents \]

**Figure 13-30. LWPCB—Lightweight Profiling Control Block**
• LWP—hardware modifies the field; software may read it, but must not change it
• Init—hardware reads and modifies the field while executing LLWPCB; the field must then remain unchanged as long as the LWPCB is in use
• SW—software may modify the field; hardware may read it, but does not change it
• No—field must remain unchanged as long as the LWPCB is in use

Table 13-8. LWPCB—Lightweight Profiling Control Block Fields

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Bits</th>
<th>Field</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>3–0</td>
<td></td>
<td>Flags</td>
<td>Flags indicating which events should be or are being counted (see Figure 13-31, “LWPCB Flags”) and whether threshold interrupts should be enabled. Before executing LLWPCB, the application sets Flags to a bit mask of the events (and interrupt) that should be enabled. LLWPCB does a logical “and” of this mask with the available feature bits in LWP_CFG and rewrites Flags with the mask of features actually enabled.</td>
<td>Init</td>
</tr>
<tr>
<td>7–4</td>
<td>27:0</td>
<td>BufferSize</td>
<td>Total size of the event ring buffer (in bytes). Must be a multiple of the event record size LwpEventSize (the value used internally will be rounded down if not). BufferSize must be at least (32 * LwpMinBufferSize * LwpEventSize).</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>7:4</td>
<td>Random</td>
<td>Number of bits of randomness to use in counters. Each time a counter is loaded from an interval to start counting down to the next event to record, the bottom Random bits are set to a random value. This avoids fixed patterns in events.</td>
<td>No</td>
</tr>
<tr>
<td>15–8</td>
<td></td>
<td>BufferBase</td>
<td>The Effective Address of the event ring buffer. Should be aligned on a 64-byte boundary for reasonable performance. Software is encouraged to align the ring buffer to a page boundary for best performance. If the default address size is less than 64 bits, the upper bits of BufferBase must be zero. LLWPCB converts BufferBase to a linear address and stores it internally. LWPCB.BufferBase is not modified.</td>
<td>No</td>
</tr>
<tr>
<td>19–16</td>
<td></td>
<td>BufferHeadOffset</td>
<td>Unsigned offset from BufferBase specifying where the LWP hardware will store the next event record. When BufferHeadOffset == BufferTailOffset, the ring buffer is empty. BufferHeadOffset must always be less than BufferSize; LWP will use a value of 0 if BufferHeadOffset is too large. Also, it must always be a multiple of LwpEventSize; LWP will round it down if not.</td>
<td>LWP</td>
</tr>
<tr>
<td>23–20</td>
<td></td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31–24</td>
<td></td>
<td>MissedEvents</td>
<td>The 64-bit count of the number of events that were missed. A missed event occurs when LWP stores an event record, attempts to advance BufferHeadOffset, and discovers that it would be equal to BufferTailOffset. In this case, LWP leaves BufferHeadOffset unchanged and instead increments the MissedEvents counter. Thus, when the ring buffer is full, the last event record is overwritten.</td>
<td>LWP</td>
</tr>
</tbody>
</table>
### Table 13-8. LWPCB—Lightweight Profiling Control Block Fields (continued)

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Bits</th>
<th>Field</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
</table>
| 35–32 |      | Threshold     | Threshold for signaling an interrupt to indicate that the ring buffer is filling up. If threshold interrupts are enabled in Flags, then when LWP advances BufferHeadOffset, it computes the space used as 
|       |      |               | ((BufferHeadOffset – BufferTailOffset) % BufferSize). If the space used equals or exceeds Threshold, LWP causes an interrupt.                                                                                   | No  |
|       |      |               | If Threshold is greater than BufferSize, no interrupt will ever be taken. If Threshold is zero, an interrupt will be taken every time an event record is stored in the ring buffer.                             |
|       |      |               | Threshold is an unsigned integer multiple of LwpEventSize (the value used internally will be rounded down if not).                                                                                           |
|       |      |               | Ignored if threshold interrupts are not available in LWP_CFG or if they are not enabled in Flags                                                                                                        |
| 39–36 |      | Filters       | Filters to qualify which events are eligible to be counted. This field includes bits to filter branch events by type and prediction status, and bits and values to filter cache events by type and latency. See Figure 13-32, “LWPCB Filters” for details. | No  |
| 47–40 |      | BaseIP        | Low limit of the IP filtering range. An instruction must start at a location greater than or equal to BaseIP to be in range.                                                                                   | No  |
|       |      |               | Ignored if IPF is zero or if the CPUID LwpIpFiltering bit is 0 to indicate that IP filtering is not supported.                                                                                           |
| 55–48 |      | LimitIP       | High limit of the IP filtering range. An instruction must start at a location less than or equal to LimitIP to be in range.                                                                                   | No  |
|       |      |               | Ignored if IPF is zero or if the CPUID LwpIpFiltering bit is 0 to indicate that IP filtering is not supported.                                                                                           |
| 63–56 |      | Reserved      |                                                                                                                                                                                                           |     |
| 67–64 |      | BufferTailOffset | Unsigned offset from BufferBase to the oldest event record in the ring buffer. BufferTailOffset is maintained by software and must always be less than BufferSize and a multiple of LwpEventSize. If software stores a value of BufferTailOffset into the LWPCB that violates these rules, the LWP hardware might not detect ring buffer overflow or threshold conditions properly. | SW  |
| 71–68 |      | Reserved      |                                                                                                                                                                                                           |     |
| 72–87 |      | Reserved      | Reserved for software use. These bytes are never read or written by the LWP hardware                                                                                                                     | SW  |
| (E-1)–88 | | Reserved area between the fixed portion of the LWPCB and the event specifiers. Should be zero. The EventInterval1 field is at offset E = LwpEventOffset. |     |
Table 13-8. LWPCB—Lightweight Profiling Control Block Fields (continued)

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Bits</th>
<th>Field</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E+3)–</td>
<td>25:0</td>
<td>EventInterval1</td>
<td>Reset value for counting events of type EventId = 1 (Programmed Value Sample). A value of ( n ) specifies that after ( n+1 ) (modified by Random) LWPVAL instructions, LWP will store an event record in the ring buffer. EventInterval1 is a signed value. If it is negative, LLWPCB will use zero and will store zero into EventInterval1 in the LWPCB. The Programmed Value Sample event is the only one which allows an interval to be below the implementation minimum interval value.</td>
<td>Init</td>
</tr>
<tr>
<td>E+3</td>
<td>7:2</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(E+7)–</td>
<td>25:0</td>
<td>EventCounter1</td>
<td>Starting (LLWPCB) or current (SLWPCB) value of counter. This is a signed number. LLWPCB treats a negative value as zero.</td>
<td>LWP</td>
</tr>
<tr>
<td>(E+4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E+7</td>
<td>7:2</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(E+11)–</td>
<td>25:0</td>
<td>EventInterval2</td>
<td>Reset value for counting events of type EventId = 2 (Instructions Retired). A value of ( n ) specifies that after ( n+1 ) (modified by Random) instructions are retired, LWP will store an event record in the ring buffer. EventInterval2 is a signed value. If it is negative or is below the implementation minimum, LLWPCB will use the minimum and will store that value into EventInterval2 in the LWPCB.</td>
<td>Init</td>
</tr>
<tr>
<td>(E+8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E+11</td>
<td>7:2</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(E+15)–</td>
<td>57:32</td>
<td>EventCounter2</td>
<td>Starting (LLWPCB) or current (SLWPCB) value of counter. This is a signed number. LLWPCB treats a negative value as zero.</td>
<td>LWP</td>
</tr>
<tr>
<td>(E+12)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E+15</td>
<td>7:2</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Event3… | Repeat event configuration similar to EventInterval2 and EventCounter2 for EventId values from 3 to LwpMaxEvents. |       |

The LLWPCB instruction reads the Flags word from the LWPCB to determine which events to profile and whether threshold interrupts should be enabled. LLWPCB writes the Flags word after turning off bits corresponding to features which are not currently available.
Event counting can be filtered by a number of conditions which are specified in the Filters word of the LLWPCB. The IP filtering applies to all events. Cache level filtering applies to all events that interact with the caches. Branch filtering applies to the Branches Retired event.

### Figure 13-31. LWPCB Flags

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Input to LLWPCB</th>
<th>Value after LLWPCB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>VAL</td>
<td>Enable LWPVAL instruction</td>
<td>LWPVAL instruction enabled</td>
</tr>
<tr>
<td>2</td>
<td>IRE</td>
<td>Enable Instructions Retired event</td>
<td>Instructions Retired event enabled</td>
</tr>
<tr>
<td>3</td>
<td>BRE</td>
<td>Enable Branches Retired event</td>
<td>Branches Retired event enabled</td>
</tr>
<tr>
<td>4</td>
<td>DME</td>
<td>Enable DCache miss event</td>
<td>DCache Miss event enabled</td>
</tr>
<tr>
<td>5</td>
<td>CNH</td>
<td>Enable CPU clocks not halted event</td>
<td>CPU Clocks Not Halted event enabled</td>
</tr>
<tr>
<td>6</td>
<td>RNH</td>
<td>Enable CPU reference clocks not halted event</td>
<td>CPU Reference Clocks Not Halted event enabled</td>
</tr>
<tr>
<td>28:7</td>
<td>Reserved</td>
<td>1—Use continuous mode. If the ring buffer overflows, LWP continues to store events and advance BufferHead. Software must stop LWP in order to empty the ring buffer. 0—Use synchronized mode. LWP operates in continuous mode if input bit is set and continuous mode is available. Otherwise, LWP operates in synchronous mode.</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>CONT</td>
<td>1—Store the Performance Time Stamp Counter (PTSC) in the TimeStamp field of each event record, if PTSC is available. 0—Store 0 in the TimeStamp field. Performance Time Stamp Counter value will be stored if input bit is set and PTSC feature is available. Otherwise 0 is stored.</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>PTSC</td>
<td>Enable threshold interrupts.</td>
<td>Threshold interrupts are enabled</td>
</tr>
<tr>
<td>31</td>
<td>INT</td>
<td>Enable threshold interrupts.</td>
<td></td>
</tr>
</tbody>
</table>
The following table provides detailed descriptions of the fields in the Filters word.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>MinLatency</td>
<td>Minimum latency for a cache-related event</td>
</tr>
<tr>
<td>8</td>
<td>CLF</td>
<td>Cache level filtering</td>
</tr>
<tr>
<td>9</td>
<td>NBC</td>
<td>Northbridge cache events</td>
</tr>
<tr>
<td>10</td>
<td>RDC</td>
<td>Remote data cache events</td>
</tr>
<tr>
<td>11</td>
<td>RAM</td>
<td>DRAM cache events</td>
</tr>
<tr>
<td>12</td>
<td>OTH</td>
<td>Other cache events</td>
</tr>
<tr>
<td>24:13</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>NMB</td>
<td>No mispredicted branches</td>
</tr>
<tr>
<td>26</td>
<td>NPB</td>
<td>No predicted branches</td>
</tr>
<tr>
<td>27</td>
<td>NAB</td>
<td>No absolute branches</td>
</tr>
<tr>
<td>28</td>
<td>NCB</td>
<td>No conditional branches</td>
</tr>
<tr>
<td>29</td>
<td>NRB</td>
<td>No unconditional relative branches</td>
</tr>
<tr>
<td>30</td>
<td>IPI</td>
<td>IP filtering invert</td>
</tr>
<tr>
<td>31</td>
<td>IPF</td>
<td>IP filtering</td>
</tr>
</tbody>
</table>

**Figure 13-32. LWPCB Filters**
<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>MinLatency</td>
<td>Minimum latency for a cache-related event to be eligible for LWP counting. Applies to all cache-related events being monitored. MinLatency is multiplied by 16 to get the actual latency in cycles, providing less resolution but a larger range for filtering. An implementation may have a maximum for the latency value. If MinLatency*16 exceeds this maximum value, the maximum is used instead. A value of 0 disables filtering by latency. Ignored if no cache latency event is enabled or if the CPUID LwpCacheLatency bit is 0 to indicate that the implementation does not filter by latency (use the CLF bits to get a similar effect). At least one of these mechanisms is supported if any cache miss events are supported.</td>
</tr>
<tr>
<td>8</td>
<td>CLF</td>
<td>Cache level filtering. 1—Enables filtering cache-related events by the cache level or memory level that returned the data. It enables the next 4 bits. Cache-related events are only eligible for counting if the bit describing the memory level is on. 0—Disables cache level filtering. The next 4 bits are ignored, and any cache or memory level is eligible. Ignored if no cache latency event is enabled or if the CPUID LwpCacheLevels bit is 0 to indicate that the implementation does not filter by cache level (use the MinLatency field to get a similar effect). At least one of these mechanisms is supported if any cache miss events are supported.</td>
</tr>
<tr>
<td>9</td>
<td>NBC</td>
<td>Northbridge cache events. 1—Count cache-related events that are satisfied from data held in a cache that resides on the northbridge. 0—Ignore northbridge cache events Ignored if CLF is 0.</td>
</tr>
<tr>
<td>10</td>
<td>RDC</td>
<td>Remote data cache events. 1—Count cache-related events that are satisfied from data held in a remote data cache. 0—Ignore remote cache events. Ignored if CLF is 0.</td>
</tr>
<tr>
<td>11</td>
<td>RAM</td>
<td>DRAM cache events. 1—Count cache-related events that are satisfied from DRAM. 0—Ignore DRAM cache events. Ignored if CLF is 0.</td>
</tr>
<tr>
<td>12</td>
<td>OTH</td>
<td>Other cache events. 1—Count cache-related events that are satisfied from other sources, such as MMIO, Config space, PCI space, or APIC. 0—Ignore such cache events Ignored if CLF is 0.</td>
</tr>
<tr>
<td>24:13</td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Table 13.9. LWPCB Filters Fields (continued)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>
| 25   | NMB   | No mispredicted branches.  
|      |       | 1—Mispredicted branches will not be counted.  
|      |       | 0—Mispredicted branches will be counted if not suppressed by other filter conditions.  
|      |       | Caution: If NMB and NPB are both set, no branches will be counted.  
|      |       | Ignored if the Branches Retired event is not enabled or if the CPUID LwpBranchPrediction bit is 0 to indicate that the implementation does not filter by prediction. |
| 26   | NPB   | No predicted branches.  
|      |       | 1—Correctly predicted branches will not be counted. Note that since direct branches are always predicted correctly, this is a superset of the NDB filter.  
|      |       | 0—Correctly predicted branches will be counted if not suppressed by other filter conditions.  
|      |       | Caution: If NMB and NPB are both set, no branches will be counted.  
|      |       | Ignored if the Branches Retired event is not enabled or if the CPUID LwpBranchPrediction bit is 0 to indicate that the implementation does not filter by prediction. |
| 27   | NAB   | No absolute branches.  
|      |       | 1—Absolute branches will not be counted. This only applies to jumps through a register or memory (JMP opcode FF /4) and calls through a register or memory (CALL opcode FF /2). Relative branches (both conditional and unconditional) are counted normally if not disabled via the NRB or NCB bits.  
|      |       | 0—Absolute branches will be counted if not suppressed by other filter conditions.  
|      |       | Caution: If NRB, NCB, and NAB are all set, no branches will be counted.  
|      |       | Ignored if the Branches Retired event is not enabled. |
| 28   | NCB   | No conditional branches.  
|      |       | 1—Conditional branches will not be counted. This only applies to conditional jumps (Jcc) and loops (LOOPcc). Unconditional relative branches, indirect jumps through a register or memory, and returns are counted normally if not disabled via the NRB or NAB bits.  
|      |       | 0—Conditional branches will be counted if not suppressed by other filter conditions.  
|      |       | Caution: If NRB, NCB, and NAB are all set, no branches will be counted.  
|      |       | Ignored if the Branches Retired event is not enabled. |
13.4.7 XSAVE/XRSTOR

LWP requires that the processor support the XSAVE/XRSTOR instructions for managing extended processor state components.

13.4.7.1 Configuration

The processor uses bit 62 of XFEATURE_ENABLED_MASK (register XCR0) to indicate whether LWP state can be saved and restored, and thus whether LWP is available to applications. The LWP XSAVE area length and offset from the beginning of the XSAVE area are available from the CPUID instruction (see “Detecting LWP XSAVE Area” on page 407). In Version 1 of LWP, the LWP XSAVE area is 128 (080h) bytes long and the offset is 832 (340h) bytes.

13.4.7.2 XSAVE Area

Figure 13-33 below shows the layout of the XSAVE area for LWP. It is large enough to allow for future expansion of the number of event counters. Details of the fields are in Table 13-10.
All fields in the XSAVE area that are marked as “Reserved” (or “Rsvd”) must be zero.

![Figure 13-33. XSAVE Area for LWP](image)

<table>
<thead>
<tr>
<th>Byte 7</th>
<th>Byte 06</th>
<th>Byte 05</th>
<th>Byte 04</th>
<th>Byte 03</th>
<th>Byte 02</th>
<th>Byte 01</th>
<th>Byte 00</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWPCBAAddress</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BufferHeadOffset</td>
<td>Counter Flags (Reserved)</td>
<td>Cntr Flags</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BufferBase</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filters</td>
<td>31</td>
<td>28</td>
<td>27</td>
<td>Rsvd</td>
<td>BufferSize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Saved Event Record</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EventCounter2</td>
<td>EventCounter1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EventCounter4</td>
<td>EventCounter3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EventCounter6</td>
<td>EventCounter5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved for EventCounter8</td>
<td>Reserved for EventCounter7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved for EventCounter10</td>
<td>Reserved for EventCounter9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved for EventCounter12</td>
<td>Reserved for EventCounter11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved for EventCounter14</td>
<td>Reserved for EventCounter13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved for EventCounter16</td>
<td>Reserved for EventCounter15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
13.4.7.3 XSAVE operation

If LWP is not currently enabled (i.e., if LWP_CBADDR = 0), no state needs to be stored. XSAVE sets bit 62 in XSAVE.HEADER.XSTATE_BV to 0 so that an attempt to restore state from this save area will use the processor supplied values. See “Processor supplied values” on page 430.

If LWP is enabled, XSAVE stores the various internal LWP values into the XSAVE area with no checking or conversion and sets bit 62 in XSAVE.HEADER.XSTATE_BV to 1.

13.4.7.4 XRSTOR operation

If bit 62 in XFEATURE_ENABLED_MASK (XCR0) is 0 or if bit 62 of EDX:EAX (EDX[30]) is 0, XRSTOR does not alter the LWP state.

If the above bits are 1 but bit 62 in XSAVE.HEADER.XSTATE_BV is 0, XRSTOR writes the LWP state using the processor supplied values, disabling LWP. See “Processor supplied values” on page 430.

If all of the above bits are 1, XRSTOR loads LWP state from the XSAVE area as follows:

1. The internal pointers and sizes are loaded.
- If BufferSize is below the implementation minimum, LWP is disabled and XRSTOR of LWP state terminates.
- If BufferSize is not a multiple of the event record size, it is rounded down.
- If BufferHeadOffset is greater than (BufferSize - LwpEventSize), a value of 0 is used instead.
- If BufferHeadOffset is not a multiple of the event record size, it is rounded down.

2. For each bit that is set in the Flags field that corresponds to an available event (as currently set in the LWP_CFG MSR), the corresponding event is enabled and the event counter is loaded from the EventCountern field. All other events are disabled.

3. If the EventId field in the SavedEventRecord is non-zero, there was a pending event when XSAVE was executed. XRSTOR loads the event record into hardware. LWP will store it into the event ring buffer as soon as possible once the CPL is 3.

Software should not alter the SavedEventRecord field. An implementation may ignore a saved event record if it was not constructed by XSAVE. Storing an event into SavedEventRecord and then executing XRSTOR is not a reliable way of injecting an event into the ring buffer.

Note that if LWP is already enabled when executing XRSTOR, the old LWP state is overwritten without being saved.

No interrupt is generated by XRSTOR if the restored value of BufferHeadOffset results in a buffer that is filled beyond the threshold. The interrupt will occur the next time an event record is stored.

XRSTOR may not restore all of the state necessary for LWP to operate. The LWP hardware will read additional state from the LWPCB when it stores then next event record.

If the CPL = 0, XRSTOR simply reloads the LWPCB address and the ring buffer address from the XSAVE area. Kernel software is trusted not to alter the area in such a way as to allow access to memory that the application could not otherwise read or write. The linear addresses in the XSAVE area were validated when the application executed LLWPCB.

If the CPL ≠ 0, XRSTOR first validates the LWPCB and ring buffer pointers. This prevents an application from altering the XSAVE area in order to gain access to memory that it could not otherwise read or write (based on the current values in the DS segment register). Note that if a program’s DS value changes after doing a successful LLWPCB, it might be incapable of doing an XSAVE and then an XRSTOR of LWP state. The XRSTOR will fail if the new DS value no longer allows access to the linear addresses corresponding to the LWPCB or the ring buffer. Programs should avoid this behavior.

If XRSTOR is executed when the CPL ≠ 0, the system performs additional checks on the LWPCB and ring buffer addresses according to the pseudo-code below. A “Store-type Segment_check” fails if the limit check fails (address is beyond the segment limit) or if the segment is read-only.

```c
bool Check(uint64 addr, uint32 size) {  // Utility function
    if (!64bit_Mode)
        addr = truncate32(addr - DS.BASE)
    uint64 top = addr + size - 1;
    if (! Store-type Segment_check on DS:[addr] || // Check lower bound
        ! Store-type Segment_check on DS:[top])  // and upper bound
```
return false;
return true;
}

if (! Check(XSAVE.LWPCBAddress, sizeof(LWPCB)) ||
! Check(XSAVE.BufferAddress, XSAVE.BufferSize))
    Disable LWP

If any of the address checks fails, LWP is disabled. No fault is generated. A program that executes XRSTOR when the CPL ≠ 0 and DS has changed can use SLWPCB to check whether LWP is running.

As with all features that use XSAVE and XRSTOR, if bit 62 of XFEATURE_ENABLED_MASK (XCR0) is 0 but bit 62 of XSAVE.HEADER.XSTATE_BV is 1, XRSTOR will cause a #GP(0) exception.

13.4.7.5 Processor supplied values

If XRSTOR is executed when bit 62 of XFEATURE_ENABLED_MASK (XCR0) and EDX:EAX are both 1, but the corresponding bit in XSAVE.HEADER.XSTATE_BV is 0, it indicates that there is no LWP state to restore. In this case, LWP_CBADDR is set to 0 and LWP is disabled. Other processor internal state for LWP is set to 0 as necessary to avoid security issues.

13.4.8 Implementation Notes

The following subsections describe other LWP considerations.

13.4.8.1 Multiple Simultaneous Events

Multiple events are possible when an instruction retires. For instance, an indirect jump through a pointer in memory can trigger the instructions retired, branches retired, and DCache miss events simultaneously. LWP counts all events that apply to the instruction, but might not store event records for all events whose event counters became negative. It is implementation dependent as to how many event records are stored when multiple event counters simultaneously become negative. If not all events cause event records to be stored, the choice of which event(s) to report is implementation dependent and may vary from run to run on the same processor.

13.4.8.2 Processor State for Context Switch, SVM, and SMM

Implementations of LWP have internal state to hold information such as the current values of the counters for the various events, a pointer into the event ring buffer, and a copy of the tail pointer for quick detection of threshold and overflow states.

There are times when the system must preserve the volatile LWP state. When the operating system context switches from one user thread to another, the old user state must be saved with the thread’s context and the new state must be loaded. When a hypervisor decides to switch from one guest OS to another, the same must be done for the guest systems’ states. Finally, state must be stored and reloaded when the system enters and exits SMM, since the SMM code may decide to shut off power to the core.
Hardware does not maintain the LWP state in the active LWPCB. This is because the counters change with every event (not just every reported event), so keeping them in memory would generate a large amount of unnecessary memory traffic. Also, the LWPCB is in user memory and may be paged out to disk at any time, so the memory may not be available when needed.

**Saving State at Thread Context Switches**

LWP requires that an operating system use the XSAVE and XRSTOR instructions to save and restore LWP state across context switches.

XRSTOR restores the LWP volatile state when restoring other system state. Some additional LWP state will be restored from the LWPCB when operations in ring 3 require that information.

LWP does not support the "lazy" state save and restore that is possible for floating point and SSE state. It does not interact with the CR0[TS] bit. Operating systems that support LWP must always do an XSAVE to preserve the old thread’s LWP context and an XRSTOR to set up the new LWP context. The OS can continue to do a lazy switch of the FP and SSE state by ensuring that the corresponding bits in EDX:EAX are clear when it executes the XSAVE and XRSTOR to handle the LWP context.

**Saving State at SVM Worldswitch to a Different Guest**

Hypervisors that allow guests to use LWP must save and restore LWP state when the guest OS changes. In addition to the usual information in the VMCB, the hypervisor must use XSAVE/XRSTOR to maintain the volatile LWP state and must also save and restore LWP_CFG. When switching between a guest that uses LWP and one that does not, the hypervisor changes the value of XFEATURE_ENABLED_MASK (XCR0), which ensures that LWP is only enabled in the appropriate guest(s).

A hypervisor need not modify the LWP state if the guest OS is not changed.

**Enabling SVM Live Migration**

Some hypervisors support live migration of a guest virtual machine. Live migration is when a hypervisor preserves the entire state of the guest running on one physical machine, copies that state to another physical machine, and then resumes execution of the guest on the new hardware.

To allow live migration among machines which may have different internal implementations of LWP, the hypervisor must present the common subset of features among all the hosts in the pool of machines that can be used. Furthermore, since the hypervisor may XSAVE LWP state on one machine and XRSTOR it on another machine, the contents of the XSAVE area must be consistent across all implementations.

This means that an implementation of LWP keeps all event counters internally, not in the LWPCB. If implementations were permitted to differ in this detail, a counter might not get properly restored after migrating the guest machine.
Saving State at SMM Entry and Exit

SMM entry and exit must save and restore LWP state when the processor is going to change power state. SMM must use XSAVE/XRSTOR and must also save and restore LWP_CFG. Since LWP is ring 3 only and is inactive in System Management Mode, its state should not need to be saved and restored otherwise.

Notes on Restoring LWP State

The LWPCB may not be in memory at all times. Therefore, the LWP hardware does not attempt to access it while still in the OS kernel/VMM/SMM, since that access might fault. Some LWP state is restored once the processor is in ring 3 and can take a #PF exception without crashing. This usually happens the next time LWP needs to store an event record into the ring buffer.

13.4.8.3 LWPCB Access

Several LWPCB fields are written asynchronously by the LWP hardware and by the user software. This section discusses techniques for reducing the associated memory traffic. This is interesting to software because it influences what state is kept internally in LWP, and it explains the protocol between the hardware filling the event ring buffer and the software emptying it.

The hardware keeps an internal copy of the event ring buffer head pointer. It need not flush the head pointer to the LWPCB every time it stores an event record. The flush can be done periodically or it can be deferred until a threshold or buffer full condition happens or until the application executes LLWPCB or SLWPCB. Exceeding the buffer threshold always forces the head pointer to memory so that the interrupt handler emptying the ring buffer sees the threshold condition.

The hardware may keep an internal copy of the event ring buffer tail pointer. It need not read the software-maintained tail pointer unless it detects a threshold or buffer full condition. At that point, it rereads the tail pointer to see if software has emptied some records from the ring buffer. If so, it recomputes the condition and acts accordingly. This implies that software polling the ring buffer should begin processing event records when it detects a threshold condition itself. To avoid a race condition with software, the hardware rereads the tail pointer every time it stores an event record while the threshold condition appears to be true. (An implementation can relax this to “every n\textsuperscript{th} time” for some small value of n.) It also rereads it whenever the ring buffer appears to be full.

The interval values used to reset the counters can be cached in the hardware when the LLWPCB instruction is executed, or they can be read from the LWPCB each time the counter overflows.

The ring buffer base and size are cached in the hardware.

The MissedEvents value is a counter for an exceptional condition and is kept in memory.

The cached LWP state is refreshed from the LWPCB when LWP is enabled either explicitly via LLWPCB or implicitly when needed in ring 3 after LWP state is restored via XRSTOR.

Caching implies that software cannot reliably change sampling intervals or other cached state by modifying the LWPCB. The change might not be noticed by the LWP hardware. On the other hand, changing state in the LWPCB while LWP is running may change the operation at an unpredictable
moment in the future if LWP context is saved and restored due to context switching. Software must stop and restart LWP to ensure that any changes reliably take effect.

13.4.8.4 Security

The operating system must ensure that information does not leak from one process to another or from the kernel to a user process. Hence, if it supports LWP at all, the operating system must ensure that the state of the LWP hardware is set appropriately when a context switch occurs and when a new process or thread is created. LWP state for a new thread can be initialized by executing XRSTOR with bit 62 of XSAVE.HEADER.XSTATE_BV set to 0 and the corresponding bit in EDX:EAX set to 1.

13.4.8.5 Interrupts

The LWP threshold interrupt vector number is specified in the LWP_CFG MSR. The operating system must assign a vector for LWP threshold interrupts and fill in the corresponding entry in the interrupt-descriptor table. Note that the LWP interrupt is not shared with the performance counter interrupt, since the system allows concurrent and independent use of those two mechanisms.

13.4.8.6 Memory Access During LWP Operation

When LWP needs to save an event record in the event ring buffer, it accesses the user memory containing the ring buffer and sometimes the memory containing the LWPCB. This causes a Page Fault (#PF) exception if those pages are not in memory.

A particular implementation of LWP has several ways to deal with page faults when storing an event record. These may include saving the event record in the XSAVE area and retrying the store later, reexecuting the instruction, or discarding the event and reporting the next event of the appropriate type.

Note that this reinforces the notion that LWP is a sampling mechanism. Programs cannot rely on it to precisely capture every n\textsuperscript{th} instance of an event. It captures approximately every n\textsuperscript{th} instance.

13.4.8.7 Guidelines for Operating Systems

To support LWP, an operating system should follow the following guidelines. Most of these operations should be done on each core of a multi-core system.

System initialization

1. Use CPUID Fn0000_0000 to ensure that the system is running on an “Authentic AMD” processor, and then check CPUID Fn8000_0001_ECX[LWP] to ensure that the processor supports LWP.

   Alternatively, check CPUID Fn0000_000D_EDX_x0[30] to ensure that the system supports the LWP XSAVE area, indicating that the processor supports LWP.

2. Enable XSAVE operations by setting CR4[OSXSAVE].

3. Enable LWP by executing XSETBV to set bit 62 of XCR0.
4. Assign a unique interrupt vector number for LWP threshold interrupts and load the corresponding entry in the interrupt-descriptor table with the address of the interrupt handler. This handler should use some system-specific method to forward any threshold interrupts to the application.

5. Make LWP available by setting LWP_CFG. To enable all supported LWP features, set LWP_CFG[31:0] to the value returned by CPUID Fn8000_001C_EDX. Set LWP_CFG[COREID] to the APIC core number (or some other value unique to the core) and LWP_CFG[VECTOR] to the assigned interrupt vector number.

Thread support
- For each thread, allocate an XSAVE area that is at least as big as the XFeatureEnabledSizeMax value returned by CPUID Fn0000_000D_EBX_x0 (ECX=0). This is good practice for any system that supports XSAVE.
- When creating a new process or thread, execute XRSTOR with bit 62 of EDX:EAX set to 1 and bit 62 of XSAVE.HEADER.XSTATE_BV set to 0. This ensures that LWP is turned off for any new thread. Alternatively, use WRMSR to write 0 into LWP_CBADDR before starting the thread.
- When saving a running thread’s context, execute XSAVE with bit 62 of EDX:EAX set to 1 to save the thread’s LWP state. It takes almost no time or resources if the thread is not using LWP.
- When restoring a thread’s context, execute XRSTOR with bit 62 of EDX:EAX set to 1. This restores the LWP state for the thread or disables LWP if the thread is not using it.
- When a thread exits or aborts, use WRMSR to store 0 into LWP_CBADDR. This ensures that LWP is turned off.

13.4.8.8 Summary of LWP State
LWP adds the following visible state to the AMD64 architecture:
- CPUID Fn8000_0001_ECX[LWP] (bit 15) to indicate LWP support.
- CPUID Fn8000_001C to indicate LWP features.
- Two new MSRs: LWP_CFG, LWP_CBADDR,
- Four new instructions: LLWPCB, SLWPCB, LWPINS, and LWPVAL.
- Bit 62 in XCR0 (XFEATURE_ENABLED_MASK)
- A new XSAVE area for LWP state.
- New fields for LWP state in the SVM and SMM context, whether in the VMCB and SMM save area or elsewhere.

See Section 3.3, “Processor Feature Identification,” on page 64 for information on using the CPUID instruction to obtain information about processor capabilities.
14 Processor Initialization and Long Mode Activation

This chapter describes the hardware actions taken following a processor reset and the steps that must be taken to initialize processor resources and activate long mode. In some cases the actions required are implementation-specific with references made to the appropriate implementation-specific documentation.

14.1 Processor Initialization

System logic can initialize the processor in either of two ways. One method, called RESET, is usually initiated by the assertion of an external signal (typically designated RESET#). The other method, called INIT, is typically initiated by another processor by means of an INIT interprocessor interrupt (IPI). See “Interprocessor Interrupts (IPI)” on page 581 for more information.

Both initialization techniques place the processor in real mode and initialize processor resources to a known, consistent state from which software can begin execution. The processor begins execution when the RESET# pin is deasserted or the INIT state is exited.

The RESET method places the processor in a known state and prepares it to begin execution in real mode. The INIT method is similar except it does not modify the state of certain registers. See Section 14.1.3 on page 436 for a comparison of these initialization methods.

System logic ensures that the processor transitions through the RESET state whenever power is reapplied after a planned or unplanned interruption. A RESET can also be performed when power is stable. An INIT can be performed at any time after the processor is powered up.

14.1.1 Built-In Self Test (BIST)

An optional built-in self-test can be performed after the processor is reset. The mechanism for triggering the BIST is implementation-specific, and can be found in the hardware documentation for the implementation. The number of processor cycles BIST can consume before completing is also implementation-specific but typically consumes several million cycles.

BIST can be used by system implementations to assist in verifying system integrity, thereby improving system reliability, availability, and serviceability. The internal BIST hardware generally tests all internal array structures for errors. These structures can include (but are not limited to):

- All internal caches, including the tag arrays as well as the data arrays.
- All TLBs.
- Internal ROMs, such as the microcode ROM and floating-point constant ROM.
- Branch-prediction structures.
EAX is loaded with zero if BIST completes without detecting errors. If any hardware faults are detected during BIST, a non-zero value is loaded into EAX.

### 14.1.2 Clock Multiplier Selection

The internal processor clock runs at some multiple of the system clock. The processor-to-system clock multiple does not have to be fixed by a processor implementation but instead can be programmable through hardware or software, or some combination of the two. For information on selecting the processor-clock multiplier, see the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product.

### 14.1.3 Processor Initialization State

Table 14-1 shows the initial processor state following either RESET or INIT. Except as indicated, processor resources generally are set to the same value after either RESET or INIT.

#### Table 14-1. Initial Processor State

<table>
<thead>
<tr>
<th>Processor Resource</th>
<th>Value After RESET</th>
<th>Value After INIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR0</td>
<td>0000_0000_6000_0010h</td>
<td>CD and NW are unchanged Bit 4 (reserved) = 1 All others = 0</td>
</tr>
<tr>
<td>CR2, CR3, CR4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>CR8</td>
<td>0</td>
<td>Not modified</td>
</tr>
<tr>
<td>RFLAGS</td>
<td>0000_0000_0000_00h</td>
<td></td>
</tr>
<tr>
<td>EFER</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>RIP</td>
<td>0000_0000_0000_00h</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td>Selector = F000h Base = 0000_0000_FFFF_0000h Limit = FFFFh Attributes = See Table 14-2 on page 438</td>
<td></td>
</tr>
<tr>
<td>DS, ES, FS, GS, SS</td>
<td>Selector = 0000h Base = 0 Limit = FFFFh Attributes = See Table 14-2 on page 438</td>
<td></td>
</tr>
<tr>
<td>GDTR, IDTR</td>
<td>Base = 0 Limit = FFFFh</td>
<td></td>
</tr>
<tr>
<td>LDTR, TR</td>
<td>Selector = 0000h Base = 0 Limit = FFFFh Attributes = See Table 14-2 on page 438</td>
<td></td>
</tr>
<tr>
<td>RAX</td>
<td>0 (non-zero if BIST is run and fails)</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 14-1. Initial Processor State (continued)

<table>
<thead>
<tr>
<th>Processor Resource</th>
<th>Value After RESET</th>
<th>Value After INIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDX</td>
<td>Family/Model/Stepping, including extended family and extended model—see “Processor Implementation Information” on page 439</td>
<td></td>
</tr>
<tr>
<td>RBX, RCX, RBP, RSP, RDI, RSI, R8, R9, R10, R11, R12, R13, R14, R15</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
| x87 Floating-Point State | FPR0–FPR7 = 0  
Control Word = 0040h  
Status Word = 0000h  
Tag Word = 5555h  
Instruction CS = 0000h  
Instruction Offset = 0  
x87 Instruction Opcode = 0  
Data-Operand DS = 0000h  
Data-Operand Offset = 0 | Not modified |
| 64-Bit Media State | MMX0–MMX7 = 0 | Not modified |
| SSE State | XMM0–XMM15 = 0  
MXCSR = 1F80h | Not modified |
| Memory-Type Range Registers | See “Memory-Typing MSRs” on page 609 | Not modified |
| Machine-Check Registers | See “Machine-Check MSRs” on page 611 | Not modified |
| DR0, DR1, DR2, DR3 | 0 |
| DR6 | 0000_0000_FFFF_0FF0h |
| DR7 | 0000_0000_0000_0400h |
| Time-Stamp Counter | 0 | Not modified |
| Performance-Monitor Resources | See “Performance-Monitoring MSRs” on page 613 | Not modified |
| Other Model-Specific Registers | See “MSR Cross-Reference” on page 603 | Not modified |
| Instruction and Data Caches | Invalidated | Not modified |
| Instruction and Data TLBs | Enabled | Not modified |
| APIC | Not modified |
| SMRAM Base Address (SMBASE) | 0003_0000h | Not modified |
| XCR0 | 0000_0000_0000_0001h | Not modified |
| PKRU | 0000_0000h | Not modified |

Table 14-2 on page 438 shows the initial state of the segment-register attributes (located in the hidden portion of the segment registers) following either RESET or INIT.
14.1.4 Multiple Processor Initialization

Following reset in multiprocessor configurations, the processors use a multiple-processor initialization protocol to negotiate which processor becomes the bootstrap processor. This bootstrap processor then executes the system initialization code while the remaining processors wait for software initialization to complete. For further information, see the documentation for particular implementations of the architecture.

14.1.5 Fetching the First Instruction

After a RESET or INIT, the processor is operating in 16-bit real mode. Normally within real mode, the code-segment base-address is formed by shifting the CS-selector value left four bits. The base address is then added to the value in EIP to form the physical address into memory. As a result, the processor can only address the first 1 Mbyte of memory when in real mode.

However, immediately following RESET or INIT, the CS-selector register is loaded with F000h, but the CS base-address is not formed by left-shifting the selector. Instead, the CS base-address is initialized to FFFF_0000h. EIP is initialized to FFF0h. Therefore, the first instruction fetched from memory is located at physical-address FFFF_FFF0h (FFFF_0000h + 0000_FFF0h).

The CS base-address remains at this initial value until the CS-selector register is loaded by software. This can occur as a result of executing a far jump instruction or call instruction, for example. When CS is loaded by software, the new base-address value is established as defined for real mode (by left shifting the selector value four bits).
14.2 Hardware Configuration

14.2.1 Processor Implementation Information

Software can read processor-identification information from the EDX register immediately following RESET or INIT. This information can be used to initialize software to perform processor-specific functions. The information stored in EDX is defined as follows:

- **Stepping ID** (bits 3:0)—This field identifies the processor-revision level.
- **Extended Model** (bits 19:16) and **Model** (bits 7:4)—These fields combine to differentiate processor models within a instruction family. For example, two processors may share the same microarchitecture but differ in their feature set. Such processors are considered different models within the same instruction family. This is a split field, comprising an extended-model portion in bits 19:16 with a legacy portion in bits 7:4.
- **Extended Family** (bits 27:20) and **Family** (bits 11:8)—These fields combine to differentiate processors by their microarchitecture.

The CPUID instruction can be used to obtain the same information. This is done by executing CPUID with either function 1 or function 8000_0001h. Additional information about the processor and the features supported can be gathered using CPUID with other feature codes. See Section 3.3, “Processor Feature Identification,” on page 64 for additional information.

14.2.2 Enabling Internal Caches

Following a RESET (but not an INIT), all instruction and data caches are disabled, and their contents are invalidated (the MOESI state is set to the invalid state). Software can enable these caches by clearing the cache-disable bit (CR0.CD) to zero (RESET sets this bit to 1). Software can further refine caching based on individual pages and memory regions. Refer to “Cache Control Mechanisms” on page 188 for more information on cache control.

**Memory-Type Range Registers (MTRRs).** Following a RESET (but not an INIT), the MTRRdefType register is cleared to 0, which disables the MTRR mechanism. The variable-range and fixed-range MTRR registers are not initialized and are therefore in an undefined state. Before enabling the MTRR mechanism, the initialization software (usually platform firmware) must load these registers with a known value to prevent unexpected results. Clearing these registers, for example, sets memory to the uncacheable (UC) type.

14.2.3 Initializing Media and x87 Processor State

Some resources used by x87 floating-point instructions and media instructions must be initialized by software before being used. Initialization software can use the CPUID instruction to determine whether the processor supports these instructions, and then initialize their resources as appropriate.

**x87 Floating-Point State Initialization.** Table 14-3 on page 440 shows the differences between the initial x87 floating-point state following a RESET and the state established by the FINIT/FNINIT instruction. An INIT does not modify the x87 floating-point state. The initialization software can
execute an FINIT or FNINIT instruction to prepare the x87 floating-point unit for use by application software. The FINIT and FNINIT instructions have no effect on the 64-bit media state.

Table 14-3. x87 Floating-Point State Initialization

<table>
<thead>
<tr>
<th>x87 Floating-Point Resource</th>
<th>RESET</th>
<th>FINIT/FNINIT Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPR0–FPR7</td>
<td>0</td>
<td>Not modified</td>
</tr>
<tr>
<td>Control Word</td>
<td>0040h</td>
<td>• Round to nearest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Single precision</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Unmask all exceptions</td>
</tr>
<tr>
<td>Status Word</td>
<td>0000h</td>
<td>037Fh</td>
</tr>
<tr>
<td>Tag Word</td>
<td>5555h</td>
<td>• Round to nearest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Extended precision</td>
</tr>
<tr>
<td>Instruction CS</td>
<td>0000h</td>
<td>• Mask all exceptions</td>
</tr>
<tr>
<td>Instruction Offset</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>x87 Instruction Opcode</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Data-Operand DS</td>
<td>0000h</td>
<td></td>
</tr>
<tr>
<td>Data-Operand Offset</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Initialization software should also load the MP, EM, and NE bits in the CR0 register as appropriate for the operating system. The recommended settings are:

- **MP=1**—Setting MP to 1 causes a device-not-available exception (#NM) to occur when the FWAIT/WAIT instruction is executed and the task-switched bit (CR0.TS) is set to 1. This supports operating systems that perform lazy context-switching of x87 floating-point state.
- **EM=0**—Clearing EM to 0 allows the x87 floating-point unit to execute instructions rather than causing a #NM exception (CR0.EM=1). System software sets EM to 1 only when software emulation of x87 instructions is desired.
- **NE=1**—Setting NE to 1 causes x87 floating-point exceptions to be handled by the floating-point exception-pending exception (#MF) handler. Clearing this bit causes the processor to externally indicate the exception occurred, and an external device can then cause an external interrupt to occur in response.

Refer to “CR0 Register” on page 42 for additional information on these control bits.

**64-Bit Media State Initialization.** There are no special requirements placed on software to initialize the processor state used by 64-bit media instructions. This state is initialized completely by the processor following a RESET. System software should leave CR0.EM cleared to 0 to allow execution of the 64-bit media instructions. If CR0.EM is set to 1, attempted execution of the 64-bit media instructions causes an invalid-opcode exception (#UD).

The 64-bit media state is not modified by an INIT.
SSE State Initialization. Platform firmware or system software must also prepare the processor to allow execution of SSE instructions. The required preparations include:

- Leaving CR0.EM cleared to 0 to allow execution of the SSE instructions. If CR0.EM is set to 1, attempted execution of the SSE instructions except FXSAVE/FXRSTOR causes an invalid-opcode exception (#UD). An attempt to execute either of these instructions when CR0.EM is set results in a #NM exception.

- Enabling the SSE instructions by setting CR4.OSFXSR to 1. Software cannot execute the SSE instructions unless this bit is set. Setting this bit also indicates that system software uses the FXSAVE and FXRSTOR instructions to save and restore, respectively, the SSE state. These instructions also save and restore the 64-bit media state and x87 floating-point state.

- Indicating that system software uses the SIMD floating-point exception (#XF) for handling SSE floating-point exceptions. This is done by setting CR4.OSXMMEXCPT to 1.

- Setting (optionally) the MXCSR mask bits to mask or unmask SSE floating-point exceptions as desired. Because this register can be read and written by application software, it is not absolutely necessary for system software to initialize it.

Refer to “CR4 Register” on page 47 for additional information on these CR4 control bits.

14.2.4 Model-Specific Initialization

Implementations of the AMD64 architecture can contain model-specific features and registers that are not initialized by the processor and therefore require system-software initialization. System software must use the CPUID instruction to determine which features are supported. Model-specific features are generally configured using model-specific registers (MSRs), which can be read and written using the RDMSR and WRMSR instructions, respectively.

Some of the model-specific features are pervasive across many processor implementations of the AMD64 architecture and are therefore described within this volume. These include:

- System-call extensions, which must be enabled in the EFER register before using the SYSCALL and SYSRET instructions. See “System-Call Extension (SCE) Bit” on page 56 for information on enabling these instructions.

- Memory-typing MSRs. See “Memory-Type Range Registers (MTRRs)” on page 439 for information on initializing and using these registers.

- The machine-check mechanism. See “Initializing the Machine-Check Mechanism” on page 285 for information on enabling and using this capability.

- Extensions to the debug mechanism. See “Software-Debug Resources” on page 356 for information on initializing and using these extensions.

- The performance-monitoring resources. See “Performance Monitoring Counters” on page 370 for information on initializing and using these resources.

Initialization of other model-specific features used by the page-translation mechanism and long mode are described throughout the remainder of this section.
Some model-specific features are not pervasive across processor implementations and are therefore not described in this volume. For more information on these features and their initialization requirements, see the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product.

### 14.3 Initializing Real Mode

A basic real-mode (real-address-mode) operating environment must be initialized so that system software can initialize the protected-mode operating environment. This real-mode environment must include:

- A real-mode IDT for vectoring interrupts and exceptions to the appropriate handlers while in real mode. The IDT base-address value in the IDTR initialized by the processor can be used, or system software can relocate the IDT by loading a new base-address into the IDTR.
- The real-mode interrupt and exception handlers. These must be loaded before enabling external interrupts.

Because the processor can always accept a non-maskable interrupt (NMI), it is possible an NMI can occur before initializing the IDT or the NMI handler. System hardware must provide a mechanism for disabling NMIs to allow time for the IDT and NMI handler to be properly initialized. Alternatively, the IDT and NMI handler can be stored in non-volatile memory that is referenced by the initial values loaded into the IDTR.

Maskable interrupts can be enabled by setting EFLAGS.IF after the real-mode IDT and interrupt handlers are initialized.

- A valid stack pointer (SS:SP) to be used by the interrupt mechanism should interrupts or exceptions occur. The values of SS:SP initialized by the processor can be used.
- One or more data-segment selectors for storing the protected-mode data structures that are created in real mode.

Once the real-mode environment is established, software can begin initializing the protected-mode environment.

### 14.4 Initializing Protected Mode

Protected mode must be entered before activating long mode. A minimal protected-mode environment must be established to allow long-mode initialization to take place. This environment must include the following:

- A protected-mode IDT for vectoring interrupts and exceptions to the appropriate handlers while in protected mode.
- The protected-mode interrupt and exception handlers referenced by the IDT. Gate descriptors for each handler must be loaded in the IDT.
- A GDT which contains:
- A code descriptor for the code segment that is executed in protected mode.
- A read/write data segment that can be used as a protected-mode stack. This stack can be used by the interrupt mechanism if interrupts or exceptions occur.

Software can optionally load the GDT with one or more data segment descriptors, a TSS descriptor, and an LDT descriptor for use by long-mode initialization software.

After the protected-mode data structures are initialized, system software must load the IDTR and GDTR with pointers to those data structures. Once these registers are initialized, protected mode can be enabled by setting CR0.PE to 1.

If legacy paging is used during the long-mode initialization process, the page-translation tables must be initialized before enabling paging. At a minimum, one page directory and one page table are required to support page translation. The CR3 register must be loaded with the starting physical address of the highest-level table supported in the page-translation hierarchy. After these structures are initialized and protected mode is enabled, paging can be enabled by setting CR0.PG to 1.

14.5 Initializing Long Mode

From protected mode, system software can initialize the data structures required by long mode and store them anywhere in the first 4 Gbytes of physical memory. These data structures can be relocated above 4 Gbytes once long mode is activated. The data structures required by long mode include the following:

- An IDT with 64-bit interrupt-gate descriptors. Long-mode interrupts are always taken in 64-bit mode, and the 64-bit gate descriptors are used to transfer control to interrupt handlers running in 64-bit mode. See “Long-Mode Interrupt Control Transfers” on page 255 for more information.
- The 64-bit mode interrupt and exception handlers to be used in 64-bit mode. Gate descriptors for each handler must be loaded in the 64-bit IDT.
- A GDT containing segment descriptors for software running in 64-bit mode and compatibility mode, including:
  - Any LDT descriptors required by the operating system or application software.
  - A TSS descriptor for the single 64-bit TSS required by long mode.
  - Code descriptors for the code segments that are executed in long mode. The code-segment descriptors are used to specify whether the processor is operating in 64-bit mode or compatibility mode. See “Code-Segment Descriptors” on page 90, “Long (L) Attribute Bit” on page 91, and “CS Register” on page 73 for more information.
  - Data-segment descriptors for software running in compatibility mode. The DS, ES, and SS segments are ignored in 64-bit mode. See “Data-Segment Descriptors” on page 91 for more information.
  - FS and GS data-segment descriptors for 64-bit mode, if required by the operating system. If these segments are used in 64-bit mode, system software can also initialize the full 64-bit base...
addresses using the WRMSR instruction. See “FS and GS Registers in 64-Bit Mode” on page 74 for more information.

The existing protected-mode GDT can be used to hold the long-mode descriptors described above.

• A single 64-bit TSS for holding the privilege-level 0, 1, and 2 stack pointers, the interrupt-stack-table pointers, and the I/O-redirection-bitmap base address (if required). This is the only TSS required, because hardware task-switching is not supported in long mode. See “64-Bit Task State Segment” on page 345 for more information.

• The 4-level page-translation tables required by long mode. Long mode also requires the use of physical-address extensions (PAE) to support physical-address sizes greater than 32 bits. See “Long-Mode Page Translation” on page 132 for more information.

If paging is enabled during the initialization process, it must be disabled before enabling long mode. After the long-mode data structures are initialized, and paging is disabled, software can enable and activate long mode.

### 14.6 Enabling and Activating Long Mode

Long mode is enabled by setting the long-mode enable control bit (EFER.LME) to 1. However, long mode is not activated until software also enables paging. When software enables paging while long mode is enabled, the processor activates long mode, which the processor indicates by setting the long-mode-active status bit (EFER.LMA) to 1. The processor behaves as a 32-bit x86 processor in all respects until long mode is activated, even if long mode is enabled. None of the new 64-bit data sizes, addressing, or system aspects available in long mode can be used until EFER.LMA=1.

Table 14-4 shows the control-bit settings for enabling and activating the various operating modes of the AMD64 architecture. The default address and data sizes are shown for each mode. For the methods of overriding these default address and data sizes, see “Instruction Prefixes” in Volume 1.
Long mode uses two code-segment-descriptor bits, CS.L and CS.D, to control the operating submodes. If long mode is active, CS.L = 1, and CS.D = 0, the processor is running in 64-bit mode, as shown in Table 14-4 on page 445. With this encoding (CS.L=1, CS.D=0), default operand size is 32 bits and default address size is 64 bits. Using instruction prefixes, the default operand size can be overridden to 64 bits or 16 bits, and the default address size can be overridden to 32 bits.

The final encoding of CS.L and CS.D in long mode (CS.L=1, CS.D=1) is reserved for future use.

When long mode is active and CS.L is cleared to 0, the processor is in compatibility mode, as shown in Table 14-4 on page 445. In compatibility mode, CS.D controls default operand and address sizes exactly as it does in the legacy x86 architecture. Setting CS.D to 1 specifies default operand and address sizes as 32 bits. Clearing CS.D to 0 specifies default operand and address sizes as 16 bits.

### 14.6.1 Activating Long Mode

Switching the processor to long mode requires several steps. In general, the sequence involves disabling paging (CR0.PG=0), enabling physical-address extensions (CR4.PAE=1), loading CR3, enabling long mode (EFER.LME=1), and finally enabling paging (CR0.PG=1).

Specifically, software must follow this sequence to activate long mode:

1. If starting from page-enabled protected mode, disable paging by clearing CR0.PG to 0. This requires that the MOV CR0 instruction used to disable paging be located in an identity-mapped page (virtual address equals physical address).
2. In any order:

---

**Table 14-4. Processor Operating Modes**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Encoding</th>
<th>Default Address Size (bits)</th>
<th>Default Data Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EFER.LMA1</td>
<td>CS.L</td>
<td>CS.D</td>
</tr>
<tr>
<td>Long Mode</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>64-Bit Mode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compatibility Mode</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Legacy Mode</td>
<td>0</td>
<td>x</td>
<td>1</td>
</tr>
</tbody>
</table>

**Note:**

1. EFER.LMA is set by the processor when software sets EFER.LME and CR0.PG according to the sequence described in “Activating Long Mode” on page 445.
2. See “Instruction Prefixes” in Volume 1 for overrides to default sizes.
- Enable physical-address extensions by setting CR4.PAE to 1. Long mode requires the use of physical-address extensions (PAE) in order to support physical-address sizes greater than 32 bits. Physical-address extensions must be enabled before enabling paging.
- Load CR3 with the physical base-address of the level-4 page-map-table (PML4). See “Long-Mode Page Translation” on page 132 for details on creating the 4-level page translation tables required by long mode.
- Enable long mode by setting EFER.LME to 1.

3. Enable paging by setting CR0.PG to 1. This causes the processor to set the EFER.LMA bit to 1. The instruction following the MOV CR0 that enables paging must be a branch, and both the MOV CR0 and the following branch instruction must be located in an identity-mapped page.

### 14.6.2 Consistency Checks

The processor performs long-mode consistency checks whenever software attempts to modify any of the control bits directly involved in activating long mode (EFER.LME, CR0.PG, and CR4.PAE). A general-protection exception (#GP) occurs when a consistency check fails. Long-mode consistency checks ensure that the processor does not enter an undefined mode or state that results in unpredictable behavior.

Long-mode consistency checks cause a general-protection exception (#GP) to occur if:

- An attempt is made to enable or disable long mode while paging is enabled.
- Long mode is enabled, and an attempt is made to enable paging before enabling physical-address extensions (PAE).
- Long mode is enabled, and an attempt is made to enable paging while CS.L=1.
- Long mode is active and an attempt is made to disable physical-address extensions (PAE).

Table 14-5 summarizes the long-mode consistency checks made during control-bit transitions.

<table>
<thead>
<tr>
<th>Control Bit</th>
<th>Transition</th>
<th>Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFER.LME</td>
<td>0 → 1</td>
<td>If (CR0.PG=1) then #GP(0)</td>
</tr>
<tr>
<td></td>
<td>1 → 0</td>
<td>If (CR0.PG=1) then #GP(0)</td>
</tr>
<tr>
<td>CR0.PG</td>
<td>0 → 1</td>
<td>If ((EFER.LME=1) &amp; (CR4.PAE=0) then #GP(0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If ((EFER.LME=1) &amp; (CS.L=1)) then #GP(0)</td>
</tr>
<tr>
<td>CR4.PAE</td>
<td>1 → 0</td>
<td>If (EFER.LMA=1) then #GP(0)</td>
</tr>
</tbody>
</table>

### 14.6.3 Updating System Descriptor Table References

Immediately after activating long mode, the system-descriptor-table registers (GDTR, LDTR, IDTR, TR) continue to reference legacy descriptor tables. The tables referenced by these descriptors all reside in the lower 4 Gbytes of virtual-address space. After activating long mode, 64-bit operating-system software should use the LGDT, LLDT, LIDT, and LTR instructions to load the system descriptor-table.
processors with references to the 64-bit versions of the descriptor tables. See “Descriptor Tables” on page 75 for details on descriptor tables in long mode.

Long mode requires 64-bit interrupt-gate descriptors to be stored in the interrupt-descriptor table (IDT). Software must not allow exceptions or interrupts to occur between the time long mode is activated and the subsequent update of the interrupt-descriptor-table register (IDTR) that establishes a reference to the 64-bit IDT. This is because the IDTR continues to reference a 32-bit IDT immediately after long mode is activated. If an interrupt or exception occurred before updating the IDTR, a legacy 32-bit interrupt gate would be referenced and interpreted as a 64-bit interrupt gate, with unpredictable results.

External interrupts can be disabled using the CLI instruction. Non-maskable interrupts (NMI) and system-management interrupts (SMI) must be disabled using external hardware. See “Long-Mode Interrupt Control Transfers” on page 255 for more information on long mode interrupts.

### 14.6.4 Relocating Page-Translation Tables

The long-mode page-translation tables must be located in the first 4 Gbytes of physical-address space before activating long mode. This is necessary because the MOV CR3 instruction used to initialize the page-map level-4 base address must be executed in legacy mode before activating long mode. Because the MOV CR3 is executed in legacy mode, only the low 32 bits of the register are written, which limits the location of the page-map level-4 translation table to the low 4 Gbytes of memory. Software can relocate the page tables anywhere in physical memory, and re-initialize the CR3 register, after long mode is activated.

### 14.7 Leaving Long Mode

To return from long mode to legacy protected mode with paging enabled, software must deactivate and disable long mode using the following sequence:

1. Switch to compatibility mode and place the processor at the highest privilege level (CPL=0).
2. Deactivate long mode by clearing CR0.PG to 0. This causes the processor to clear the LMA bit to 0. The MOV CR0 instruction used to disable paging must be located in an identity-mapped page. Once paging is disabled, the processor behaves as a standard 32-bit x86 processor.
3. Load CR3 with the physical base-address of the legacy page tables.
4. Disable long mode by clearing EFER.LME to 0.
5. Enable legacy page-translation by setting CR0.PG to 1. The instruction following the MOV CR0 that enables paging must be a branch, and both the MOV CR0 and the following branch instruction must be located in an identity-mapped page.

### 14.8 Long-Mode Initialization Example

Following is sample code that outlines the steps required to place the processor in long mode.

---

Processor Initialization and Long Mode Activation
mydata segment para
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; This generic data-segment holds pseudo-descriptors used
; by the LGDT and LIDT instructions.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Establish a temporary 32-bit GDT and IDT.
;
pGDT32 label fword ; Used by LGDT.
  dw gdt32_limit ; GDT limit ...
  dd gdt32_base ; and 32-bit GDT base
pIDT32 label fword ; Used by LIDT.
  dw idt32_limit ; IDT limit ...
  dd idt32_base ; and 32-bit IDT base

; Establish a 64-bit GDT and IDT (64-bit linear base-
; address)
;
pGDT64 label tbyte ; Used by LGDT.
  dw gdt64_limit ; GDT limit ...
  dq gdt64_base ; and 64-bit GDT base
pIDT64 label tbyte ; Used by LIDT.
  dw idt64_limit ; IDT limit ...
  dq idt64_base ; and 64-bit IDT base
mydata ends ; end of data segment
code16 segment para use16 ; 16-bit code segment
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; 16-bit code, real mode
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Initialize DS to point to the data segment containing
; pGDT32 and PIDT32. Set up a real-mode stack pointer, SS:SP,
; in case of interrupts and exceptions.
;
ci
  mov ax, seg mydata
  mov ds, ax
  mov ax, seg mystack
  mov ss, ax
  mov sp, esp0

; Use CPUID to determine if the processor supports long mode.

  mov eax, 80000000h ; Extended-function 80000000h.
cpuid ; Is largest extended function
cmp eax, 80000000h ; any function > 80000000h?
jbe no_long_mode ; If not, no long mode.
  mov eax, 80000001h ; Extended-function 80000001h.
cpuid                ; Now EDX = extended-features flags.
btx edx, 29         ; Test if long mode is supported.
jnc no_long_mode    ; Exit if not supported.

; Load the 32-bit GDT before entering protected mode.
; This GDT must contain, at a minimum, the following
descriptors:
1) a CPL=0 16-bit code descriptor for this code segment.
2) a CPL=0 32/64-bit code descriptor for the 64-bit code.
3) a CPL=0 read/write data segment, usable as a stack
   (referenced by SS).

; Load the 32-bit IDT, in case any interrupts or exceptions
occur after entering protected mode, but before enabling
long mode).

; Initialize the GDTR and IDTR to point to the temporary
32-bit GDT and IDT, respectively.
lgdt ds:[pGDT32]
lidt ds:[pIDT32]

; Enable protected mode (CR0.PE=1).
mov eax, 000000011h
mov cr0, eax

; Execute a far jump to turn protected mode on.
codel6_sel must point to the previously-established 16-bit
code descriptor located in the GDT (for the code currently
being executed).

  db 0eah  ;Far jump...
dw offset now_in_prot;to offset...
dw codel6_sel;in current code segment.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
At this point we are in 16-bit protected mode, but long
mode is still disabled.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
now_in_prot:

; Set up the protected-mode stack pointer, SS:ESP.
Stack_sel must point to the previously-established stack
descriptor (read/write data segment), located in the GDT.
Skip setting DS/ES/FS/GS, because we are jumping right to
64-bit code.
mov ax, stack_sel
mov ss, ax
mov esp, esp0
Enable the 64-bit page-translation-table entries by setting CR4.PAE=1 (this is _required_ before activating long mode). Paging is not enabled until after long mode is enabled.

```
mov eax, cr4
bts eax, 5
mov cr4, eax
```

Create the long-mode page tables, and initialize the 64-bit CR3 (page-table base address) to point to the base of the PML4 page table. The PML4 page table must be located below 4 Gbytes because only 32 bits of CR3 are loaded when the processor is not in 64-bit mode.

```
mov eax, pml4_base ; Pointer to PML4 table (<4GB).
mov cr3, eax ; Initialize CR3 with PML4 base.
```

Enable long mode (set EFER.LME=1).

```
mov ecx, 0c0000080h ; EFER MSR number.
rdsr ; Read EFER.
bts eax, 8 ; Set LME=1.
wrmsr ; Write EFER.
```

Enable paging to activate long mode (set CR0.PG=1)

```
mov eax, cr0 ; Read CR0.
bts eax, 31 ; Set PE=1.
mov cr0, eax ; Write CR0.
```

At this point, we are in 16-bit compatibility mode ( LMA=1, CS.L=0, CS.D=0 ).

Now, jump to the 64-bit code segment. The offset must be equal to the linear address of the 64-bit entry point, because 64-bit code is in an unsegmented address space. The selector points to the 32/64-bit code selector in the current GDT.

```
db 066h
db 0eah
dd start64_linear
dw code64_sel
codel6ends ; End of the 16-bit code segment
```

```
;;;; Start of 64-bit code
;;;;
```

```
```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

```
```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
code64  para use64
start64:  ; At this point, we're using 64-bit code
;
; Point the 64-bit RSP register to the stack’s _linear_
; address. There is no need to set SS here, because the SS
; register is not used in 64-bit mode.
;
    mov    rsp, stack0_linear
;
; This LGDT is only needed if the long-mode GDT is to be
; located at a linear address above 4 Gbytes. If the long
; mode GDT is located at a 32-bit linear address, putting
; 64-bit descriptors in the GDT pointed to by [pGDT32] is
; just fine. pGDT64_linear is the _linear_ address of the
; 10-byte GDT pseudo-descriptor.
;
    lgdt   [pGDT64_linear]
;
; The new GDT should have a valid CPL0 64-bit code segment
; descriptor at the entry-point corresponding to the current
; CS selector. Alternatively, a far transfer to a valid CPL0
; 64-bit code segment descriptor in the new GDT must be done
; before enabling interrupts.
;
    lidt   [pIDT64_linear]
;
; Load the 64-bit IDT. This is _required_, because the 64-bit
; IDT uses 64-bit interrupt descriptors, while the 32-bit
; IDT used 32-bit interrupt descriptors. pIDT64_linear is
; the _linear_ address of the 10-byte IDT pseudo-descriptor.
;
    mov    ax, tss_sel
    ltr    ax
;
; Set the current TSS. tss_sel should point to a 64-bit TSS
; descriptor in the current GDT. The TSS is used for
; inner-level stack pointers and the IO bit-map.
;
    mov    ax, ldt_sel
    lldt   ax
;
; Set the current LDT. ldt_sel should point to a 64-bit LDT
; descriptor in the current GDT.
;
    mov    ax, fs_base
    ltr    ax
;
; Using fs: and gs: prefixes on memory accesses still uses
; the 32-bit fs.base and gs.base. Reload these 2 registers
; before using the fs: and gs: prefixes. FS and GS can be
; loaded from the GDT using a normal “mov fs,foo” type
; instructions, which loads a 32-bit base into FS or GS.
; Alternatively, use WRMSR to assign 64-bit base values to
; MSR_FS_base or MSR_GS_base.
;  mov    ecx, MSR_FS_base
    mov    eax, FsbaseLow
    mov    edx, FsbaseHi
    wrmsr

    ; Reload CR3 if long-mode page tables are to be located above
    ; 4 Gbytes. Because the original CR3 load was done in 32-bit
    ; legacy mode, it could only load 32 bits into CR3. Thus, the
    ; current page tables are located in the lower 4 Gbytes of
    ; physical memory. This MOV to CR3 is only needed if the
    ; actual long-mode page tables should be located at a linear
    ; address above 4 Gbytes.

    mov    rax, final_pml4_base  ; Point to PML4
    mov    cr3, rax              ; Load 64-bit CR3

    ; Enable interrupts.

    sti ; Enabled INTR

<insert 64-bit code here>
15 Secure Virtual Machine

The AMD Virtualization™ (AMD-V™) architecture is designed to support enterprise-class server virtualization software technology and facilitate virtualization development and deployment on any type of system, through the Secure Virtual Machine (SVM) extension. An SVM-enabled virtual machine architecture provides hardware resources that allow a single physical machine to run multiple operating systems efficiently, while maintaining secure, hardware-enforced isolation.

15.1 The Virtual Machine Monitor

A virtual machine monitor (VMM), also known as a hypervisor, consists of software that controls the execution of multiple guest operating systems on a single physical machine. The VMM provides each guest the appearance of full control over a complete computer system (memory, CPU, and all peripheral devices). The use of the term host refers to the execution context of the VMM. World switch refers to the operation of switching between the host and guest. A guest may have one or more virtual CPUs (vCPUs) managed by the guest OS, just as on a non-virtualized system, and a VMM may run any mix of vCPUs from the same or different guests on different logical processors simultaneously with no hardware-imposed constraints.

Fundamentally, VMMs work by intercepting and emulating in a safe manner sensitive operations in the guest (such as changing the page tables, which could give a guest access to memory it is not allowed to access, or accessing peripheral devices that are shared among multiple guests). The AMD SVM architecture provides hardware assists to improve performance and facilitate implementation of virtualization.

15.2 SVM Hardware Overview

SVM processor support provides a set of hardware extensions designed to enable economical and efficient implementation of virtual machine systems. Generally speaking, hardware support falls into two complementary categories: virtualization support and security support.

15.2.1 Virtualization Support

The AMD virtual machine architecture is designed to provide:

- A guest/host tagged TLB to reduce virtualization overhead
- External (DMA) access protection for memory
- Assists for interrupt handling, virtual interrupt support, and enhanced pause filter
- The ability to intercept selected instructions or events in the guest
- Mechanisms for fast world switch between VMM and guest
15.2.2 Guest Mode

This new processor mode is entered through the VMRUN instruction. When in guest mode, the behavior of some x86 instructions changes to facilitate virtualization.

The CPUID function numbers 4000_0000h–4000_00FFh have been reserved for software use. Hypervisors can use these function numbers to provide an interface to pass information from the hypervisor to the guest. This is similar to extracting information about a physical CPU by using CPUID. Hypervisors use the CPUID Fn 40000000h:00 bit to denote a virtual platform.

Feature bit CPUID Fn0000_0001_ECX[31] has been reserved for use by hypervisors to indicate the presence of a hypervisor. Hypervisors set this bit to 1 and physical CPUs set this bit to zero. This bit can be probed by the guest software to detect whether they are running inside a virtual machine.

15.2.3 External Access Protection

Guests may be granted direct access to selected I/O devices. Hardware support is designed to prevent devices owned by one guest from accessing memory owned by another guest (or the VMM).

15.2.4 Interrupt Support

To facilitate efficient virtualization of interrupts, the following support is provided under control of VMCB flags:

**Intercepting physical interrupt delivery.** The VMM can request that physical interrupts cause a running guest to exit, allowing the VMM to process the interrupt.

**Virtual interrupts.** The VMM can inject virtual interrupts into the guest. Under control of the VMM, a virtual copy of the EFLAGS.IF interrupt mask bit, and a virtual copy of the APIC's task priority register are used transparently by the guest instead of the physical resources.

**Sharing a physical APIC.** SVM allows multiple guests to share a physical APIC with isolation of each guest's manipulation of APIC state from the other guests' views of their own APIC state, so that no guest can interfere with delivery of interrupts to another guest.

**Direct interrupt delivery.** On models that support it, the Advanced Virtual Interrupt Controller (AVIC) extension virtualizes the APIC's interrupt delivery functions. This provides for delivery of device or inter-processor interrupts directly to a target vCPU or vCPUs, which avoids the overhead of having the VMM to determine interrupt routing and speeds up interrupt delivery. (see section 15.29).

15.2.5 Restartable Instructions

SVM is designed to safely restart, with the exception of task switches, any intercepted instruction (either atomic or idempotent) after the intercept.
15.2.6 Security Support

To further support secure initialization and execution, SVM provides additional system support through a variety of extensions.

**Attestation.** The SKINIT instruction and associated system support (the Trusted Platform Module, or TPM) allow for verifiable startup of trusted software (such as a hypervisor, or a native operating system), based on secure hash comparison. (section 15.27).

**Encrypted memory.** On models that support it, the Secure Encrypted Virtualization (SEV) and SEV Encrypted State (SEV-ES) extensions guard against inspection of guest memory and (for SEV-ES) guest register state by malicious hypervisor code, memory bus tracing or memory device removal through encryption of guest memory and register contents (section 15.34 and section 15.35).

**Secure Nested Paging.** On models that support it, the SEV-SNP extension provides additional protection for guest memory against malicious manipulation of address translation mechanisms by hypervisor code. (section 15.36).

15.3 SVM Processor and Platform Extensions

SVM hardware extensions can be grouped into the following categories:

- State switch—VMRUN, VMSAVE, VMLOAD instructions, global interrupt flag (GIF), and instructions to manipulate the latter (STGI, CLGI). (section 15.5, section 15.5.2, section 15.17)
- Intercepts—allow the VMM to intercept sensitive operations in the guest. (section 15.7 through section 15.14)
- Interrupt and APIC assists—physical interrupt intercepts, virtual interrupt support, APIC.TPR virtualization. (section 15.17 and section 15.21)
- SMM intercepts and assists (section 15.22)
- External (DMA) access protection (section 15.24)
- Nested paging support for two levels of address translation. (section 15.25)
- Security—SKINIT instruction. (section 15.27)

15.4 Enabling SVM

The VMRUN, VMLOAD, VMSAVE, CLGI, VMMCALL, and INVLPGA instructions can be used when the EFER.SVME is set to 1; otherwise, these instructions generate a #UD exception. The SKINIT and STGI instructions can be used when either the EFER.SVME bit is set to 1 or the feature flag CPUID Fn8000_0001_ECX[SKINIT] is set to 1; otherwise, these instructions generate a #UD exception.

Before enabling SVM, software should detect whether SVM can be enabled using the following algorithm:
if (CPUID Fn8000_0001_ECX[SVM] == 0)
    return SVM_NOT_AVAIL;

if (VM_CR.SVMDIS == 0)
    return SVM_ALLOWED;

if (CPUID Fn8000_000A_EDX[SVML]==0)
    return SVM_DISABLED_AT_BIOS_NOT_UNLOCKABLE
    // the user must change a platform firmware setting to enable SVM
else return SVM_DISABLED_WITH_KEY;
    // SVMLock may be unlockable; consult platform firmware or TPM to obtain the key.

For more information on using the CPUID instruction to obtain processor capability information, see
Section 3.3, “Processor Feature Identification,” on page 64.

15.5 VMRUN Instruction

The VMRUN instruction is the cornerstone of SVM. VMRUN takes, as a single argument, the
physical address of a 4KB-aligned page, the virtual machine control block (VMCB), which describes a
virtual machine (guest) to be executed. The VMCB contains:

- a list of instructions or events in the guest (e.g., write to CR3) to intercept,
- various control bits that specify the execution environment of the guest or that indicate special
  actions to be taken before running guest code, and
- guest processor state (such as control registers, etc.).

Note that VMRUN is not supported inside the SMM handler and the behavior is undefined.

15.5.1 Basic Operation

The VMRUN instruction has an implicit addressing mode of [rAX]. Software must load RAX (EAX
in 32-bit mode) with the physical address of the VMCB, a 4-Kbyte-aligned page that describes a
virtual machine to be executed. The portion of RAX used in forming the address is determined by the
current effective address size.

The VMCB is accessed by physical address and should be mapped as writeback (WB) memory.

VMRUN is available only at CPL 0. A #GP(0) exception is raised if the CPL is greater than 0.
Furthermore, the processor must be in protected mode and EFER.SVME must be set to 1, otherwise, a
#UD exception is raised.

The VMRUN instruction saves some host processor state information in the host state-save area in
main memory at the physical address specified in the VM_HSAVE_PA MSR; it then loads
corresponding guest state from the VMCB state-save area. VMRUN also reads additional control bits
from the VMCB that allow the VMM to flush the guest TLB, inject virtual interrupts into the guest,
etc.
The VMRUN instruction then checks the guest state just loaded. If an illegal state has been loaded, the processor exits back to the host (section 15.6).

Otherwise, the processor now runs the guest code until an intercept event occurs, at which point the processor suspends guest execution and resumes host execution at the instruction following the VMRUN. This is called a #VMEXIT and is described in detail in (section 15.6).

VMRUN saves or restores a minimal amount of state information to allow the VMM to resume execution after a guest has exited. This allows the VMM to handle simple intercept conditions quickly. If additional guest state information must be saved or restored (e.g., to handle more complex intercepts or to switch to a different guest), the VMM must use the VMLOAD and VMSAVE instructions to handle the additional guest state. (see section 15.5.2).

**Saving Host State.** To ensure that the host can resume operation after #VMEXIT, VMRUN saves at least the following host state information:

- CS.SEL, NEXT RIP—The CS selector and rIP of the instruction following the VMRUN. On #VMEXIT the host resumes running at this address.
- RFLAGS, RAX—Host processor mode and the register used by VMRUN to address the VMCB.
- SS.SEL, RSP—Stack pointer for host.
- CR0, CR3, CR4, EFER—Paging/operating mode for host.
- IDTR, GDTR—The pseudo-descriptors. VMRUN does not save or restore the host LDTR.
- ES.SEL and DS.SEL.

Processor implementations may store only part or none of host state in the memory area pointed to by VM_HSAVE_PA MSR and may store some or all host state in hidden on-chip memory. Different implementations may choose to save the hidden parts of the host’s segment registers as well as the selectors. For these reasons, software must not rely on the format or contents of the host state save area, nor attempt to change host state by modifying the contents of the host save area.

**Loading Guest State.** After saving host state, VMRUN loads the following guest state from the VMCB:

- CS, rIP—Guest begins execution at this address. The hidden state of the CS segment register is also loaded from the VMCB.
- RFLAGS, RAX.
- SS, RSP—Includes the hidden state of the SS segment register.
- CR0, CR2, CR3, CR4, EFER—Guest paging mode. Writing paging-related control registers with VMRUN does not flush the TLB since address spaces are switched. (section 15.16.)
- INTERRUPT_SHADOW—This flag indicates whether the guest is currently in an interrupt lockout shadow; (section 15.21.5).
- IDTR, GDTR.
- ES and DS—Includes the hidden state of the segment registers.
- DR6 and DR7—The guest’s breakpoint state.
- V_TPR—The guest’s virtual TPR.
- V_IRQ—The flag indicating whether a virtual interrupt is pending in the guest.
- CPL—If the guest is in real mode, the CPL is forced to 0; if the guest is in v86 mode, the CPL is forced to 3. Otherwise, the CPL saved in the VMCB is used.

The processor checks the loaded guest state for consistency. If a consistency check fails while loading guest state, the processor performs a #VMEXIT. For additional information, see “Canonicalization and Consistency Checks” on page 459.

If the guest is in PAE paging mode according to the registers just loaded and nested paging is not enabled, the processor will also read the four PDPEs pointed to by the newly loaded CR3 value; setting any reserved bits in the PDPEs also causes a #VMEXIT.

It is possible for the VMRUN instruction to load a guest rIP that is outside the limit of the guest code segment or that is non-canonical (if running in long mode). If this occurs, a #GP fault is delivered inside the guest; the rIP falling outside the limit of the guest code segment is not considered illegal guest state.

After all guest state is loaded, and intercepts and other control bits are set up, the processor reenables interrupts by setting GIF to 1. It is assumed that VMM software cleared GIF some time before executing the VMRUN instruction, to ensure an atomic state switch.

Some processor models allow the VMM to designate certain guest VMCB fields as “clean,” meaning that they haven’t been modified relative to the current state of hardware. This allows the hardware to optimize execution of VMRUN. See section 15.15 for details on which fields may be affected by this. The descriptions below assume all fields are loaded.

**Control Bits.** Besides loading guest state, the VMRUN instruction reads various control fields from the VMCB; most of these fields are not written back to the VMCB on #VMEXIT, since they cannot change during guest execution:

- TSC_OFFSET—an offset to add when the guest reads the TSC (time stamp counter). Guest writes to the TSC can be intercepted and emulated by changing the offset (without writing the physical TSC). This offset is cleared when the guest exits back to the host.
- V_INTR_PRIO, V_INTR_VECTOR, V_IGN_TPR—fields used to describe a virtual interrupt for the guest (see “Injecting Virtual (INTR) Interrupts” on page 488).
- V_INTR_MASKING—controls whether masking of interrupts (in EFLAGS.IF and TPR) is to be virtualized (section 15.21).
- The address space ID (ASID) to use while running the guest.
- A field to control flushing of the TLB during a VMRUN (see Section 15.16).
- The intercept vectors describing the active intercepts for the guest. On exit from the guest, the internal intercept registers are cleared so no host operations will be intercepted.
The maximum ASID value supported by a processor is implementation specific. The value returned in EBX after executing CPUID F8000_000A is the number of ASIDs supported by the processor.

See Section 3.3, “Processor Feature Identification,” on page 64 for more information on using the CPUID instruction.

**Segment State in the VMCB.** The segment registers are stored in the VMCB in a format similar to that for SMM: both base and limit are fully expanded; segment attributes are stored as 12-bit values formed by the concatenation of bits 55:52 and 47:40 from the original 64-bit (in-memory) segment descriptors; the descriptor “P” bit is used to signal NULL segments (P=0) where permissible and/or relevant. The loading of segment attributes from the VMCB (which may have been overwritten by software) may result in attribute bit values that are otherwise not allowed. However, only some of the attribute bits are actually observed by hardware, depending on the segment register in question:

- CS—D, L, and R.
- SS—B, P, E, W, and Code/Data
- LDTR—Only the P bit is observed.
- TR—Only TSS type (32 or 16 bit) is relevant because a null TSS is not allowed.

NOTE: For the Stack Segment attributes, P is observed in legacy and compatibility mode. In 64-bit mode, P is ignored because all stack segments are treated as present.

The VMM should follow these rules when storing segment attributes into the VMCB:

- For NULL segments, set all attribute bits to zero; otherwise, write the concatenation of bits 55:52 and 47:40 from the original 64-bit (in-memory) segment descriptors.
- The processor reads the current privilege level from the CPL field in the VMCB. The CS.DPL will match the CPL field.
- When in virtual x86 or real mode, the processor ignores the CPL field in the VMCB and forces the values of 3 and 0, respectively.

When examining segment attributes after a #VMEXIT:

- Test the Present (P) bit to check whether a segment is NULL; note that CS and TR never contain NULL segments and so their P bit is ignored;
- Retrieve the CPL from the CPL field in the VMCB, not from any segment DPL.

**Canonicalization and Consistency Checks.** The VMRUN instruction performs consistency checks on guest state and #VMEXIT performs the appropriate subset of these consistency checks on host state. Illegal guest state combinations cause a #VMEXIT with error code VMEXIT_INVAlsALID. The following conditions are considered illegal state combinations:

- EFER.SVME is zero.
- CR0.CD is zero and CR0.NW is set.
- CR0[63:32] are not zero.
- Any MBZ bit of CR3 is set.
- Any MBZ bit of CR4 is set.
- DR6[63:32] are not zero.
- DR7[63:32] are not zero.
- Any MBZ bit of EFER is set.
- EFER.LMA or EFER.LME is non-zero and this processor does not support long mode.
- EFER.LME and CR0.PG are both set and CR4.PAE is zero.
- EFER.LME and CR0.PG are both non-zero and CR0.PE is zero.
- EFER.LME, CR0.PG, CR4.PAE, CS.L, and CS.D are all non-zero.
- The VMRUN intercept bit is clear.
- The MSR or IOIO intercept tables extend to a physical address that is greater than or equal to the maximum supported physical address.
- Illegal event injection (section 15.20).
- ASID is equal to zero.

VMRUN can load a guest value of CR0 with PE = 0 but PG = 1, a combination that is otherwise illegal (see Section 15.19).

In addition to consistency checks, VMRUN and #VMEXIT canonicalize (i.e., sign-extend to 63 bits) all base addresses in the segment registers that have been loaded.

On processor models that support designation of clean fields, the final merged hardware state is used for consistency checks; this may include state from fields marked as clean, if the processor choose to ignore the indication.

**VMRUN and TF/RF Bits in EFLAGS.** When considering interactions of VMRUN with the TF and RF bits in EFLAGS, one must distinguish between the behavior of host as opposed to that of the guest.

From the host point of view, VMRUN acts like a single instruction, even though an arbitrary number of guest instructions may execute before a #VMEXIT effectively completes the VMRUN. As a single host instruction, VMRUN interacts with EFLAGS.RF and EFLAGS.TF like ordinary instructions. EFLAGS.RF suppresses any potential instruction breakpoint match on the VMRUN, and EFLAGS.TF causes a #DB trap after the VMRUN completes on the host side (i.e., after the #VMEXIT from the guest). As with any normal instruction, completion of the VMRUN instruction clears the host EFLAGS.RF bit.

The value of EFLAGS.RF from the VMCB affects the first guest instruction. When VMRUN loads a guest value of 1 for EFLAGS.RF, that value takes effect and suppresses any potential (guest) instruction breakpoint on the first guest instruction. When VMRUN loads a guest value of 1 in EFLAGS.TF, that value does not cause a trace trap between the VMRUN and the first guest instruction, but rather after completion of the first guest instruction.
Host values of EFLAGS have no effect on the guest and guest values of EFLAGS have no effect on the host.

See also section 15.7.1 regarding the value of EFLAGS.RF saved on #VMEXIT.

15.5.2 VMSAVE and VMLOAD Instructions

These instructions transfer additional guest register context, including hidden context that is not otherwise accessible, between the processor and a guest's VMCB for a more complete context switch than VMRUN and #VMEXIT perform. The system physical address of the VMCB is specified in rAX. When these operations are needed, VMLOAD would be executed as desired prior to executing a VMRUN, and VMSAVE at any desired point after a #VMEXIT.

The VMSAVE and VMLOAD instructions take the physical address of a VMCB in rAX. These instructions complement the state save/restore abilities of VMRUN instruction and #VMEXIT. They provide access to hidden processor state that software cannot otherwise access, as well as additional privileged state.

These instructions handle the following register state:

- FS, GS, TR, LDTR (including all hidden state)
- KernelGsBase
- STAR, LSTAR, CSTAR, SFMASK
- SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP

Like VMRUN, these instructions are only available at CPL0 (otherwise causing a #GP(0) exception) and are only valid in protected mode with SVM enabled via EFER.SVME (otherwise causing a #UD exception).

15.6 #VMEXIT

When an intercept triggers, the processor performs a #VMEXIT (i.e., an exit from the guest to the host context).

On #VMEXIT, the processor:

- Disables interrupts by clearing the GIF, so that after the #VMEXIT, VMM software can complete the state switch atomically.
- Writes back to the VMCB the current guest state—the same subset of processor state as is loaded by the VMRUN instruction, including the V_IRQ, V_TPR, and the INTERRUPT_SHADOW bits.
- Saves the reason for exiting the guest in the VMCB’s EXITCODE field; additional information may be saved in the EXITINFO1 or EXITINFO2 fields, depending on the intercept. Note that the contents of the EXITINFO1 and EXITINFO2 fields are undefined for intercepts where their use is not indicated.
• Clears all intercepts.
• Resets the current ASID register to zero (host ASID).
• Clears the V_IRQ and V_INTR_MASKING bits inside the processor.
• Clears the TSC_OFFSET inside the processor.
• Reloads the host state previously saved by the VMRUN instruction. The processor reloads the host’s CS, SS, DS, and ES segment registers and, if required, re-reads the descriptors from the host’s segment descriptor tables, depending on the implementation. The segment descriptor tables must be mapped as present and writable by the host's page tables. Software should keep the host’s segment descriptor tables consistent with the segment registers when executing VMRUN instructions. Immediately after #VMEXIT, the processor still contains the guest value for LDTR. So for CS, SS, DS, and ES, the VMM must only use segment descriptors from the global descriptor table. (The VMSAVE instruction can be used for a more complete context switch, allowing the VMM to then load LDTR and other registers not saved by #VMEXIT with desired values; see section 15.5.2 for details.) Any exception encountered while reloading the host segments causes a shutdown.
• If the host is in PAE mode, the processor reloads the host's PDPEs from the page table indicated by the host's CR3. If the PDPEs contain illegal state, the processor causes a shutdown.
• Forces CR0.PE = 1, RFLAGS.VM = 0.
• Sets the host CPL to zero.
• Disables all breakpoints in the host DR7 register.
• Checks the reloaded host state for consistency; any error causes the processor to shutdown. If the host’s rIP reloaded by #VMEXIT is outside the limit of the host’s code segment or non-canonical (in the case of long mode), a #GP fault is delivered inside the host.

15.7 Intercept Operation

Various instructions and events (such as exceptions) in the guest can be intercepted by means of control bits in the VMCB. The two primary classes of intercepts supported by SVM are instruction and exception intercepts.

Exception intercepts. Exception intercepts are checked when normal instruction processing must raise an exception before resolving possible double-fault conditions and before attempting delivery of the exception (which includes pushing an exception frame, accessing the IDT, etc.).

For some exceptions, the processor still writes certain exception-specific registers even if the exception is intercepted. (See the descriptions in section 15.12 and following for details.) When an external or virtual interrupt is intercepted, the interrupt is left pending.

When an intercept occurs while the guest is in the process of delivering a non-intercepted interrupt or exception using the IDT, SVM provides additional information on #VMEXIT (See section 15.7.2).
**Instruction intercepts.** These occur at well-defined points in instruction execution—before the results of the instruction are committed, but ordered in an intercept-specific priority relative to the instruction’s exception checks. Generally, instruction intercepts are checked after simple exceptions (such as #GP—when CPL is incorrect—or #UD) have been checked, but before exceptions related to memory accesses (such as page faults) and exceptions based on specific operand values. There are several exceptions to this guideline, e.g., the RSM instruction. Instruction breakpoints for the current instruction and pending data breakpoint traps from the previous instruction are designed to be checked before instruction intercepts.

### 15.7.1 State Saved on Exit

When triggered, intercepts write an EXITCODE into the VMCB identifying the cause of the intercept. The EXITINTINFO field signals whether the intercept occurred while the guest was attempting to deliver an interrupt or exception through the IDT; a VMM can use this information to transparently complete the delivery (section 15.20). Some intercepts provide additional information in the EXITINFO1 and EXITINFO2 fields in the VMCB; see the individual intercept descriptions for details.

The guest state saved in the VMCB is the processor state as of the moment the intercept triggers. In the x86 architecture, traps (as opposed to faults) are detected and delivered after the instruction that triggered them has completed execution. Accordingly, a trap intercept takes place after the execution of the instruction that triggered the trap in the first place. The saved guest state thus includes the effects of executing that instruction.

**Example:** Assume a guest instruction triggers a data breakpoint (#DB) trap which is in turn intercepted. The VMCB records the guest state after execution of that instruction, so that the saved CS:rIP points to the following instruction, and the saved DR7 includes the effects of matching the data breakpoint.

The next sequential instruction pointer (nRIP) is saved in the guest VMCB control area at location C8h on all #VMEXITs that are due to instruction intercepts, as defined in section 15.9, as well as MSR and IOIO intercepts and exceptions caused by the INT3, INTO, and BOUND instructions. For all other intercepts, nRIP is reset to zero.

The nRIP is the RIP that would be pushed on the stack if the current instruction were subject to a trap-style debug exception, if the intercepted instruction were to cause no change in control flow. If the intercepted instruction would have caused a change in control flow, the nRIP points to the next sequential instruction rather than the target instruction.

Some exceptions write special registers even when they are intercepted; see the individual descriptions in section 15.12 for details.

Support for the NRIP save on #VMEXIT is indicated by CPUID Fn8000_000A_EDX[NRIPS]. See Section 3.3, “Processor Feature Identification,” on page 64 for more information on using the CPUID instruction.
15.7.2 Intercepts During IDT Interrupt Delivery

It is possible for an intercept to occur while the guest is attempting to deliver an exception or interrupt through the IDT (e.g., #PF because the VMM has paged out the guest’s exception stack). In some cases, such an intercept can result in the loss of information necessary for transparent resumption of the guest. In the case of an external interrupt, for example, the processor will already have performed an interrupt acknowledge cycle with the PIC or APIC to obtain the interrupt type and vector, and the interrupt is thus no longer pending.

To recover from such situations, all intercepts indicate (in the EXITINTINFO field in the VMCB) whether they occurred during exception or interrupt delivery through the IDT. This mechanism allows the VMM to complete the intercepted interrupt delivery, even when it is no longer possible to recreate the event in question.

Despite the instruction name, the events raised by the INT1 (also known as ICEBP), INT3 and INTO instructions (opcodes F1h, CCh and CEh) are considered exceptions for the purposes of EXITINTINFO, not software interrupts. Only events raised by the INTn instruction (opcode CDh) are considered software interrupts.
• Error Code Valid—Bit 11. Set to 1 if the guest exception would have pushed an error code; otherwise cleared to zero.
• Valid—Bit 31. Set to 1 if the intercept occurred while the guest attempted to deliver an exception through the IDT; otherwise cleared to zero.
• Errorcode—Bits 63:32. If EV is set to 1, holds the error code that the guest exception would have pushed; otherwise is undefined.

In the case of multiple exceptions, EXITINTINFO records the aggregate information on all exceptions but the last (intercepted) one.

**Example:** A guest raises a #GP during delivery of which a #NP is raised (a scenario that, according to x86 rules, resolves to a #DF), and an intercepted #PF occurs during the attempt to deliver the #DF. Upon intercept of the #PF, EXITINTINFO indicates that the guest was in the process of delivering a #DF when the #PF occurred. The information about the intercepted page fault itself is encoded in the EXITCODE, EXITINFO1 and EXITINFO2 fields. If the VMM decides to repair and dismiss the #PF, it can resume guest execution by re-injecting (see section 15.20) the fault recorded in EXITINTINFO. If the VMM decides that the #PF should be reflected back to the guest, it must combine the event in EXITINTINFO with the intercepted exception according to x86 rules. In this case, a #DF plus a #PF would result in a triple fault or shutdown.

### 15.7.3 EXITINTINFO Pseudo-Code

When delivering exceptions or interrupts in a guest, the processor checks for exception intercepts and updates the value of EXITINTINFO should an intercept occur during exception delivery. The following pseudo-code outlines how the processor delivers an event (exception or interrupt) E.

```plaintext
if E is an exception and is intercepted:
    #VMEXIT(E)
E = (result of combining E with any prior events)

if (result was #DF and #DF is intercepted):
    #VMEXIT(#DF)
if (result was shutdown and shutdown is intercepted):
    #VMEXIT(#shutdown)
EXITINTINFO = E // Record the event the guest is delivering.

Attempt delivery of E through the IDT
Note that this may cause secondary exceptions

Once an exception has been successfully taken in the guest:

EXITINTINFO.V = 0 // Delivery succeeded; no #VMEXIT.
Dispatch to first instruction of handler

When an exception triggers an intercept, the EXITCODE, and optionally EXITINFO1 and EXITINFO2, fields always reflect the intercepted exception, while EXITINTINFO, if marked valid, indicates the prior exception the guest was attempting to deliver when the intercept occurred.
```
15.8 Decode Assists

Decode assists are provided to allow hypervisors to decode guest instructions more efficiently. CPUID Fn8000_000A_EDX[DecodeAssists] = 1 indicates support for this feature. See Section 3.3, “Processor Feature Identification,” on page 64 for more information on using the CPUID instruction.

15.8.1 MOV CRx/DRx Intercepts

The EXITINFO1 field holds a flag indicating whether the instruction was a MOV CRx and the number of the GPR operand. MOV-to-CR instructions always set bit 63 and provide the GPR number, except for CR0 as specified below.

Table 15-2. EXITINFO1 for MOV CRx

<table>
<thead>
<tr>
<th>Bit Offsets</th>
<th>Field Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:0</td>
<td>GPR number</td>
</tr>
<tr>
<td>62:4</td>
<td>0</td>
</tr>
<tr>
<td>63</td>
<td>Instruction was MOV CRx—set to 1 if the instruction was a MOV CRx instruction; cleared to 0 otherwise.</td>
</tr>
</tbody>
</table>

Table 15-3. EXITINFO1 for MOV DRx

<table>
<thead>
<tr>
<th>Bit Offsets</th>
<th>Field Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:0</td>
<td>GPR number</td>
</tr>
<tr>
<td>63:4</td>
<td>0</td>
</tr>
</tbody>
</table>

MOV-to-CR0 Special Case. If the instruction is MOV-to-CR, the GPR number is provided; if the instruction is LMSW or CLTS, no additional information is provided and bit 63 is not set.

MOV-from-CR0 Special Case. If the instruction is MOV-from-CR, the GPR number is provided and bit 63 is set; if the instruction is SMSW, no information is provided and bit 63 is not set.

15.8.2 INTn Intercepts

EXITINFO1 records the immediate value of the interrupt number for INT n instructions. See Table 15-4.

Table 15-4. EXITINFO1 for INTn

<table>
<thead>
<tr>
<th>Bit Offsets</th>
<th>Field Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>Software interrupt number</td>
</tr>
</tbody>
</table>
15.8.3 INVLPG and INVLPGA Intercepts

For an INVLPG intercept, EXITINFO1 provides the linear address after segment base addition and address size masking produce the effective address size. See Table 15-5. For an INVLPGA intercept, the linear address is available directly from the guest rAX register and is not provided in EXITINFO1.

15.8.4 Nested and intercepted #PF

In the case of a Nested Page Fault or intercepted #PF, guest instruction bytes at guest CS:RIP are stored into the 16-byte wide field Guest Instruction Bytes located at offset 0D0h in the VMCB. The format of this field is summarized in Table 15-6 below. Up to 15 bytes are recorded, read from guest CS:RIP. If a faulting condition occurs, such as not-present page or exceeding the CS limit, then the Guest Instruction Bytes field records as many bytes as could be fetched. The number of bytes fetched is put into the first byte of this field. Zero indicates that no bytes were fetched. The default number of bytes is always 15. Fewer bytes are returned only if a fault occurs while fetching.

This field is filled in only during data page faults. Instruction-fetch page faults provide no additional information.

All other intercepts clear bits 0:7 in this field to zero (to indicate an invalid condition); implementations may leave the other bytes untouched.

<table>
<thead>
<tr>
<th>Bit Offsets</th>
<th>Field Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:8</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 15-5. EXITINFO1 for INVLPG

<table>
<thead>
<tr>
<th>Bit Offsets</th>
<th>Field Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:0</td>
<td>Linear address</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit Offsets</th>
<th>Field Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:0</td>
<td>Number of bytes fetched</td>
</tr>
<tr>
<td>4:7</td>
<td>0</td>
</tr>
<tr>
<td>127:8</td>
<td>Instruction bytes</td>
</tr>
</tbody>
</table>

Table 15-6. Guest Instruction Bytes
## 15.9 Instruction Intercepts

Table 15-7 specifies the instructions that check a given intercept and, where relevant, how the intercept is prioritized relative to exceptions.

### Table 15-7. Instruction Intercepts

<table>
<thead>
<tr>
<th>Instruction Intercept</th>
<th>Checked By</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write of CR0</td>
<td>MOV TO/FROM CR0, LMSW, SMSW, CLTS</td>
<td>Checks non-memory exceptions (CPL, illegal bit combinations, etc.) before the intercept. For LMSW and SMSW, checks SVM intercepts before checking memory exceptions.</td>
</tr>
<tr>
<td>Read/Write of CR3 (excluding task switch)</td>
<td>MOV TO/FROM CR3 (not checked by task switch operations)</td>
<td>Checks non-memory exceptions first, then the intercept. If the intercept triggers on a write, the intercept happens before the TLB is flushed. If PAE is enabled, the loading of the four PDPEs can cause a #GP; that exception is checked after the intercept check, so the VMM handling a CR3 intercept cannot rely on the PDPEs being legal; it must examine them in software if necessary. The reads and writes of CR3 that occur in VMRUN, #VMEXIT or task switches are not subject to this intercept check.</td>
</tr>
<tr>
<td>Read/Write of other CRs</td>
<td>MOV TO/FROM CRn</td>
<td>All normal exception checks take precedence over the SVM intercepts.</td>
</tr>
<tr>
<td>Read/Write of Debug Registers, DRn</td>
<td>MOV TO/FROM DRn. (Not checked by implicit DR6/DR7 writes.)</td>
<td>All normal exception checks take precedence over the SVM intercepts.</td>
</tr>
<tr>
<td>Selective CR0 Write Intercept</td>
<td>MOV TO CR0, LMSW</td>
<td>Checks non-memory exceptions (CPL, illegal bit combinations, etc.) before the intercept. For LMSW and SMSW, checks SVM intercepts before checking memory exceptions. The selective write intercept on CR0 triggers only if a bit other than CR0.TS or CR0.MP is being changed by the write. In particular, this means that CLTS does not check this intercept. When both selective and non-selective CR0-write intercepts are active at the same time, the non-selective intercept takes priority. With respect to exceptions, the priority of this intercept is the same as the generic CR0-write intercept. The LMSW instruction treats the selective CR0-write intercept as a non-selective intercept (i.e., it intercepts regardless of the value being written).</td>
</tr>
<tr>
<td>Reading or Writing IDTR, GDTR, LDTR, TR</td>
<td>LIDT, SIDT, LGDT, SGDT, LLDT, SLDT, LTR, STR</td>
<td>The SVM intercept is checked after #UD and #GP exception checks, but before any memory access is performed.</td>
</tr>
</tbody>
</table>
Table 15-7. Instruction Intercepts (continued)

<table>
<thead>
<tr>
<th>Instruction Intercept</th>
<th>Checked By</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDTSC</td>
<td>RDTSC</td>
<td>Checks all exceptions before the SVM intercept.</td>
</tr>
<tr>
<td>RDPMC</td>
<td>RDPMC</td>
<td>Checks all exceptions before the SVM intercept.</td>
</tr>
<tr>
<td>PUSHF</td>
<td>PUSHF</td>
<td>Takes priority over any exceptions.</td>
</tr>
<tr>
<td>POPF</td>
<td>POPF</td>
<td>Takes priority over any exceptions.</td>
</tr>
<tr>
<td>CPUID</td>
<td>CPUID</td>
<td>No exceptions to check.</td>
</tr>
<tr>
<td>RSM</td>
<td>RSM</td>
<td>The intercept takes priority over any exceptions.</td>
</tr>
<tr>
<td>IRET</td>
<td>IRET</td>
<td>The intercept takes priority over any exceptions.</td>
</tr>
<tr>
<td><strong>Software Interrupt</strong></td>
<td><strong>INTn</strong></td>
<td>The intercept occurs before any exceptions are checked. The CS:rIIP reported on #VMEXIT are those of the intercepted INTn instruction. Though the INTn instruction may dispatch through IDT vectors in the range of 0–31, those events cannot be intercepted by means of exception intercepts (see &quot;Exception Intercepts&quot; on page 474).</td>
</tr>
<tr>
<td>INVD</td>
<td>INVD</td>
<td>Exceptions (#GP) are checked before the intercept.</td>
</tr>
<tr>
<td><strong>PAUSE</strong></td>
<td><strong>PAUSE</strong></td>
<td>No exceptions to check. VMRUN copies the VMCB.PauseFilterCount into an internal counter. Each PAUSE instruction decrements the counter, and the PAUSE intercept only occurs if the counter goes below zero while the PAUSE intercept is enabled. The VMCB.PauseFilterCount field is not written by the processor. Certain events, including SMI, can cause the internal count to be reloaded from the VMCB. VMCB.PauseFilterCount support is indicated by EDX[10] as returned by CPUID extended function 8000_000A. If this feature is not supported or VMCB.PauseFilterCount = 0, then the first PAUSE instruction can be intercepted.</td>
</tr>
<tr>
<td>HLT</td>
<td>HLT</td>
<td>Checks all exceptions before checking for this intercept.</td>
</tr>
<tr>
<td>INVLPD</td>
<td>INVLPD</td>
<td>Checks all exceptions (#GP) before the intercept.</td>
</tr>
<tr>
<td>INVLPDGA</td>
<td>INVLPDGA</td>
<td>Checks all exceptions (#GP) before the intercept.</td>
</tr>
<tr>
<td>VMRUN</td>
<td>VMRUN</td>
<td>Checks exceptions (#GP) before the intercept. The current implementation requires that the VMRUN intercept always be set in the VMCB.</td>
</tr>
<tr>
<td>VMLOAD</td>
<td>VMLOAD</td>
<td>Checks exceptions (#GP) before the intercept.</td>
</tr>
<tr>
<td>VMSAVE</td>
<td>VMSAVE</td>
<td>Checks exceptions (#GP) before the intercept.</td>
</tr>
</tbody>
</table>
The VMM can intercept IOIO instructions (IN, OUT, INS, OUTS) on a port-by-port basis by means of the SVM I/O permissions map.

### 15.10 IOIO Intercepts

The VMM can intercept IOIO instructions (IN, OUT, INS, OUTS) on a port-by-port basis by means of the SVM I/O permissions map.
15.10.1 I/O Permissions Map

The I/O Permissions Map (IOPM) occupies 12 Kbytes of contiguous physical memory. The map is structured as a linear array of 64K+3 bits (two 4-Kbyte pages, and the first three bits of a third 4-Kbyte page) and must be aligned on a 4-Kbyte boundary; the physical base address of the IOPM is specified in the IOPM_BASE_PA field in the VMCB and loaded into the processor by the VMRUN instruction. The VMRUN instruction ignores the lower 12 bits of the address specified in the VMCB. If the address of the last byte in the IOPM is greater than or equal to the maximum supported physical address, this is treated as illegal VMCB state and causes a #VMEXIT(VMEXIT_INVALID).

Each bit in the IOPM corresponds to an 8-bit I/O port. Bit 0 in the table corresponds to I/O port 0, bit 1 to I/O port 1 and so on. A bit set to 1 indicates that accesses to the corresponding port should be intercepted. The IOPM is accessed by physical address, and should reside in memory that is mapped as writeback (WB).

15.10.2 IN and OUT Behavior

If the IOIO_PROT intercept bit is set, the IOPM controls port access. For IN/OUT instructions that access more than a single byte, the permission bits for all bytes are checked; if any bit is set to 1, the I/O operation is intercepted.

Exceptions related to virtual x86 mode, IOPL, or the TSS-bitmap are checked before the SVM intercept check. All other exceptions are checked after the SVM intercept check.

I/O Intercept Information. When an IOIO intercept triggers, the following information (describing the intercepted operation in order to facilitate emulation) is saved in the VMCB’s EXITINFO1 field:
The rIP of the instruction following the IN/OUT is saved in EXITINFO2, so that the VMM can easily resume the guest after I/O emulation.

### 15.10.3 (REP) OUTS and INS

Bits 12:10 of the EXITINFO1 field provide the effective segment number (the default segment is DS). (For segment register encodings, see Table A-32, “16-Bit Register and Memory References” on page 478, in *AMD64 Architecture Programmer’s Manual Volume 3: General-Purpose and System Instructions*.)

INS provides the effective segment (always ES, encoded as 0).

On intercepted SMI-on-I/O, bits 12:10 of EXITINFO1 encode the segment. For definitions of the remaining bits of this field, (section 15.13.3).

### 15.11 MSR Intercepts

The VMM can intercept RDMSR and WRMSR instructions by means of the *SVM MSR permissions map* (MSRPM) on a per-MSR basis.
**MSR Permissions Map.** The MSR permissions bitmap consists of four separate bit vectors of 16 Kbits (2 Kbytes) each. Each 16 Kbit vector controls guest access to a defined range of 8K MSRs. Each MSR is covered by two bits defining the guest read and write access permissions. The lsb of the two bits controls read access to the MSR and the msb controls write access. A value of 1 indicates that the operation is intercepted. The four separate bit vectors must be packed together and located in two contiguous physical pages of memory. If the MSR_PROT intercept is active, any attempt to read or write an MSR not covered by the MSRPM will automatically cause an intercept.

The following table defines the ranges of MSRs covered by the MSR permissions map. Note that the MSR ranges are not contiguous.

<table>
<thead>
<tr>
<th>MSRPM Byte Offset</th>
<th>MSR Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>000h–7FFh</td>
<td>0000_0000h–0000_1FFFh</td>
</tr>
<tr>
<td>800h–FFFh</td>
<td>C000_0000h–C000_1FFFh</td>
</tr>
<tr>
<td>1000h–17FFh</td>
<td>C001_0000h–C001_1FFFh</td>
</tr>
<tr>
<td>1800h–1FFFh</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The MSRPM is accessed by physical address and should reside in memory that is mapped as writeback (WB). The MSRPM must be aligned on a 4KB boundary. The physical base address of the MSRPM is specified in MSRPM_BASE_PA field in the VMCB and is loaded into the processor by the VMRUN instruction. The VMRUN instruction ignores the lower 12 bits of the address specified in the VMCB, and if the address of the last byte in the table is greater than or equal to the maximum supported physical address, this is treated as illegal VMCB state and causes a #VMEXIT(VMEXIT_INVALID).

**RDMSR and WRMSR Behavior.** If the MSR_PROT bit in the VMCB’s intercept vector is clear, RDMSR/WRMSR instructions are not intercepted.

RDMSR and WRMSR instructions check for exceptions and intercepts in the following order:

- Exceptions common to all MSRs (e.g., #GP if not at CPL 0)
- Check SVM intercepts in the MSR permission map, if the MSR_PROT intercept is requested.
- Exceptions specific to a given MSR (including password protection, unimplemented MSRs, reserved bits, etc.)

**MSR Intercept Information.** On #VMEXIT, the processor indicates in the VMCB’s EXITINFO1 whether a RDMSR (EXITINFO1 = 0) or WRMSR (EXITINFO1 = 1) was intercepted.
15.12 Exception Intercepts

When intercepting exceptions that define an error code (normally pushed onto the exception stack), the SVM hardware delivers that error code in the VMCB’s EXITINFO1 field; the exception vector number can be derived from the EXITCODE. The CS.SEL and rIP saved in the VMCB on an exception-intercept match those that would otherwise have been pushed onto the exception stack frame, except that when an interrupt-based instruction causes an intercept, the rIP of the instruction is stored in the VMCB, rather than the rIP of the next instruction. The interrupt-based instructions are INT3 (opcode CC), INTO, and BOUND.

Unless otherwise noted below, no special registers are written before an exception is intercepted. For details on guest state saved in the VMCB, see section 15.7.1.

External interrupts and software interrupts (INTn instruction) do not check the exception intercepts, even when they use a vector in the range 0 to 31.

Exceptions that occur during the handling of a prior exception are checked for intercepts before being combined with the prior exception (e.g., into a double-fault). If the result of combining exceptions is a double-fault or shutdown, the processor checks whether those are intercepted before attempting delivery.

Example: Assume that the VMM intercepts #GP and #DF exceptions, and the guest raises a (non-intercepted) #NP, during the delivery of which it also gets a #GP (e.g., due to an illegal IDT entry)—a situation that, according to x86 semantics, results in a #DF. In this case, #VMEXIT signals an intercepted #GP, not an intercepted #DF and fills EXITINTINFO with the #NP fault. On the other hand, if only the #DF intercept were active in this scenario, #VMEXIT would signal an intercepted #DF.

The following subsections detail the individual intercepts.

15.12.1 #DE (Divide By Zero)

The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.2 #DB (Debug)

The #DB exception can have fault-type (e.g., instruction breakpoint) or trap-type (e.g., data breakpoint) behavior; accordingly the intercept differs in what state is saved in the VMCB (see section 15.7.1). In either case, however, the value saved for DR6 and DR7 matches what would be visible to a #DB exception handler (i.e., both #DB faults and traps are permitted to write DR6 and DR7 before the intercept). The EXITINFO1 and EXITINFO2 fields are undefined.

Fault-type #DB exceptions, whether indicated in EXITCODE or EXITINTINFO, cause the CS:rIP saved in the VMCB to indicate the instruction that caused the #DB exception. Trap-type #DB exceptions cause the VMCB’s CS:rIP to indicate the instruction following the instruction that caused the exception. A vector 1 exception generated by the single byte INT1 instruction (also known as
ICEBP) does not trigger the #DB intercept. Software should use the dedicated ICEBP intercept to intercept ICEBP (see section 15.9).

15.12.3 Vector 2 (Reserved)
This intercept bit is not implemented; use the NMI intercept (section 15.13.2) instead. The effect of setting this bit is undefined.

15.12.4 #BP (Breakpoint)
This intercept applies to the trap raised by the single byte INT3 (opcode CCh) instruction. The EXITINFO1 and EXITINFO2 fields are undefined. The CS:rIP reported on #VMEXIT are those of the INT3 instruction.

15.12.5 #OF (Overflow)
This intercept applies to the trap raised by the INTO (opcode CEh) instruction. The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.6 #BR (Bound-Range)
This intercept applies to the fault raised by the BOUND instruction. The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.7 #UD (Invalid Opcode)
The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.8 #NM (Device-Not-Available)
The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.9 #DF (Double Fault)
The EXITINFO1 and EXITINFO2 fields are undefined. The rIP value saved in the VMCB is undefined (as is the case for the rIP value pushed on the stack for #DF exceptions). If a double fault is intercepted, the exceptions leading up to the double fault will have written any status registers normally written by those exceptions.

15.12.10 Vector 9 (Reserved)
This intercept is not implemented. The effect of setting this bit is undefined.

15.12.11 #TS (Invalid TSS)
The EXITINFO1 and EXITINFO2 fields are undefined. The rIP value saved in the VMCB may point to either the instruction causing the task switch, or to the first instruction of the incoming task. See section 15.14.1 for information on the EXITINFO1 and EXITINFO2 fields.
15.12.12  #NP (Segment Not Present)

The EXITINFO1 field contains the error code that would be pushed on the stack by a #NP exception. The EXITINFO2 field is undefined.

15.12.13  #SS (Stack Fault)

The EXITINFO1 field contains the error code that would be pushed on the stack by a #SS exception. The EXITINFO2 field is undefined.

15.12.14  #GP (General Protection)

The EXITINFO1 field contains the error code that would be pushed on the stack by a #GP exception.

15.12.15  #PF (Page Fault)

This intercept is tested before CR2 is written by the exception. The error code saved in EXITINFO1 is the same as would be pushed onto the stack by a non-intercepted #PF exception in protected mode. The faulting address is saved in the EXITINFO2 field in the VMCB. Even when the guest is running in paged real mode, the processor will deliver the (protected-mode) page-fault error code in EXITINFO1, for the VMM to use in analyzing the intercepted #PF. The processor may provide additional instruction decode assist information. (See section 15.8.4.)

15.12.16  #MF (X87 Floating Point)

This intercept is tested after the floating point status word has been written, as is the case for a normal FP exception. The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.17  #AC (Alignment Check)

The EXITINFO1 field contains the error code that would be pushed on the stack by an #AC exception. The EXITINFO2 field is undefined.

15.12.18  #MC (Machine Check)

The SVM intercept is checked after all #MC-specific registers have been written, but before other guest state is modified. When #MC is being intercepted, a machine-check exits to the VMM, whenever possible, and shuts down the processor only when this is not a reasonable option. The EXITINFO1 and EXITINFO2 fields are undefined.

15.12.19  #XF (SIMD Floating Point)

This intercept is tested after the SIMD status word (MXCSR) has been written, as is the case for a normal FP exception. The EXITINFO1 and EXITINFO2 fields are undefined.
15.13 Interrupt Intercepts

External interrupts, when intercepted, cause a #VMEXIT; the interrupt is held pending so that the interrupt can eventually be taken in the VMM. Exception intercepts do not apply to external or software interrupts, so it is not possible to intercept an interrupt by means of the exception intercepts, even if the interrupt should happen to use a vector in the range from 0 to 31.

15.13.1 INTR Intercept

This intercept affects physical, as opposed to virtual, maskable interrupts. See “Virtual Interrupt Intercept” on page 489 for virtualization of maskable interrupts.

15.13.2 NMI Intercept

This intercept affects non-maskable interrupts. NMI interrupts (and SMIs) may be blocked for one instruction following an STI.

15.13.3 SMI Intercept

This intercept affects System Management Mode Interrupts (SMIs); see “SMM Support” on page 490 for details on SMI handling.

When this intercept triggers, bit 0 of the EXITINFO1 field distinguishes whether the SMI was caused internally by I/O Trapping (bit 0 = 0), or asserted externally (bit 0 = 1).

If the SMI was asserted while the guest was executing an I/O instruction, extra information (describing the I/O instruction) is saved in the upper 32 bits of EXITINFO1, and the rIP of the I/O instruction is saved in EXITINFO2. EXITINFO1 indicates that SMI was asserted during an I/O instruction when the VALID bit is set.

If the SMI wasn't asserted during an I/O instruction, the extra EXITINFO1 and EXITINFO2 bits are undefined.

The SMI intercept is ignored when HWCR[SMMLOCK] is set.
15.13.4 INIT Intercept
The INIT intercept allows the VMM to intercept the assertion of INIT while a guest is running; see section 15.21.8 for a discussion of the INIT-redirection feature.

15.13.5 Virtual Interrupt Intercept
This intercept is taken just before a guest takes a virtual interrupt. When the intercept triggers, the virtual interrupt has not been taken, and remains pending in the guest's VMCB V_IRQ field. This intercept is not required for handling fixed local APIC interrupts, but may be used for emulating ExtINT interrupt delivery mode (which is not masked by the TPR), or legacy PICs in auto-EOI mode.

15.14 Miscellaneous Intercepts
The SVM architecture includes intercepts to handle task switches, processor freezes due to FERR, and shutdown operations.

15.14.1 Task Switch Intercept

Checked by—Any instruction or event that causes a task switch (e.g., JMP, CALL, exceptions, interrupts, software interrupts).

Priority—The intercept is checked before the task switch takes place but after the incoming TSS and task gate (if one was involved) have been checked for correctness.
Task switches can modify several resources that a VMM may want to protect (CR3, EFLAGS, LDT). However, instead of checking various intercepts (e.g., CR3 Write, LDTR Write) individually, task switches check only a single intercept bit.

On #VMEXIT, the following information is delivered in the VMCB:

- EXITINFO1[15:0] holds the segment selector identifying the incoming TSS.
- EXITINFO2[31:0] holds the error code to push in the new task, if applicable; otherwise, this field is undefined.
- EXITINFO2[63:32] holds auxiliary information for the VMM:
  - EXITINFO2[36]—Set to 1 if the task switch was caused by an IRET; else cleared to 0.
  - EXITINFO2[38]—Set to 1 if the task switch was caused by a far jump; else cleared to 0.
  - EXITINFO2[44]—Set to 1 if the task switch has an error code; else cleared to 0.
  - EXITINFO2[48]—The value of EFLAGS.RF that would be saved in the outgoing TSS if the task switch were not intercepted.

15.14.2 Ferr_Freeze Intercept

Checked when the processor freezes due to assertion of FERR (while IGNNE is deasserted, and legacy handling of FERR is selected in CR0.NE), i.e., while the processor is waiting to be unfrozen by an external interrupt.

15.14.3 Shutdown Intercept

When this intercept occurs, any condition that normally causes a shutdown causes a #VMEXIT to the VMM instead. After an intercepted shutdown, the state saved in the VMCB is undefined.

15.14.4 Pause Intercept Filtering

On processors that support Pause filtering (indicated by CPUID Fn8000_000A_EDX[PauseFilter] = 1), the VMCB provides a 16 bit PAUSE Filter Count value. On VMRUN this value is loaded into an internal counter. Each time a PAUSE instruction is executed, this counter is decremented until it reaches zero at which time a #VMEXIT is generated if PAUSE intercept is enabled. If the PAUSE Filter Count is set to zero and PAUSE Intercept is enabled, every PAUSE instruction will cause a #VMEXIT.

In addition, some processor families support Advanced Pause Filtering (indicated by CPUID Fn8000_000A_EDX[PauseFilterThreshold] = 1). In this mode, a 16-bit PAUSE Filter Threshold field is added in the VMCB. The threshold value is a cycle count that is used to reset the pause counter.

As with simple Pause filtering, VMRUN loads the PAUSE count VMCB value into an internal counter. Then, on each PAUSE instruction the hardware checks the elapsed number of cycles since the most recent PAUSE instruction against the PAUSE Filter Threshold. If the elapsed cycle count is greater than the PAUSE Filter Threshold, then the internal pause count is reloaded from the VMCB and execution continues. If the elapsed cycle count is less than the PAUSE Filter Threshold, then the
internal pause count is decremented. If the count value is less than zero and PAUSE intercept is enabled, a #VMEXIT is triggered.

If Advanced Pause Filtering is supported and PAUSE Filter Threshold field is set to zero, the filter will operate in the simpler, count only mode.

See Section 3.3, “Processor Feature Identification,” on page 64 for more information on using the CPUID instruction.

15.15 VMCB State Caching

VMCB state caching allows the processor to cache certain guest register values in hardware between a #VMEXIT and subsequent VMRUN instructions and use the cached values to improve context-switch performance. Depending on the particular processor implementation, VMRUN loads each guest register value either from the VMCB or from the VMCB state cache, as specified by the value of the VMCB Clean field in the VMCB. Support for VMCB state caching is indicated by CPUID Fn8000_000A_EDX[VmcbClean] = 1.

The SVM architecture uses the physical address of the VMCB as a unique identifier for the guest virtual CPU for the purposes of deciding whether the cached copy belongs to the guest. For the purposes of VMCB state caching, the ASID is not a unique identifier for a guest virtual CPU.

15.15.1 VMCB Clean Bits

The VMCB Clean field (VMCB offset 0C0h, bits 31:0) controls which guest register values are loaded from the VMCB state cache on VMRUN. Each set bit in the VMCB Clean field allows the processor to load one guest register or group of registers from the hardware cache; each clear bit requires that the processor load the guest register from the VMCB. The clean bits are a hint, since any given processor implementation may ignore bits that are set to 1 on any given VMRUN, unconditionally loading the associated register value(s) from the VMCB. Clean bits that are set to zero are always honored.

This field is backward-compatible to CPUs that do not support VMCB state caching; older CPUs neither cache VMCB state nor read the VMCB Clean field.

Older hypervisors that are not aware of VMCB state caching and obey the SBZ property of undefined VMCB fields will not enable VMCB state caching.

15.15.2 Guidelines for Clearing VMCB Clean Bits

The hypervisor must clear specific bits in the VMCB Clean field every time it explicitly modifies the associated guest state in the VMCB. The guest's execution can cause cached state to be updated, but the hypervisor is not responsible for setting VMCB Clean bits corresponding to any state changes caused by guest execution.

The hypervisor must clear the entire VMCB field to 0 for a guest, under the following circumstances:
- This is the first time a particular guest is run.
- The hypervisor executes the guest on a different CPU core than one used the last time that guest was executed.
- The hypervisor has moved the guest's VMCB to a different physical page since the last time that guest was executed.

Failure to clear the VMCB Clean bits to zero in these cases may result in undefined behavior.

The CPU automatically treats the VMCB Clean field as zero on the current VMRUN when the hypervisor executes a guest that is not currently cached. The CPU compares the VMCB physical address against all cached VMCB physical addresses and treats the VMCB Clean field as zero, if no cached VMCB address matches.

SMM software (or any other agent external to the hypervisor that has access to VMCBs) that changes the contents of a VMCB needs to comprehend the clean bits and adjust them accordingly; otherwise the guest may not operate as intended.

### 15.15.3 VMCB Clean Field

The layout of the VMCB Clean field is illustrated in Figure 15-4 below.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:12</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>AVIC</td>
<td>AVIC_APIC_BAR; AVIC_APIC_BACKING_PAGE, AVIC_PHYSICAL_TABLE and AVIC_LOGICAL_TABLE Pointers</td>
</tr>
<tr>
<td>10</td>
<td>LBR</td>
<td>DbgCtlMsr, br_from/to, lastint_from/to</td>
</tr>
<tr>
<td>9</td>
<td>CR2</td>
<td>CR2</td>
</tr>
<tr>
<td>8</td>
<td>SEG</td>
<td>CS/DS/SS/ES Sel/Base/Limit/Attr, CPL</td>
</tr>
<tr>
<td>7</td>
<td>DT</td>
<td>GDT/IDT Limit and Base</td>
</tr>
<tr>
<td>6</td>
<td>DRx</td>
<td>DR6, DR7</td>
</tr>
<tr>
<td>5</td>
<td>CRx</td>
<td>CR0, CR3, CR4, EFER</td>
</tr>
<tr>
<td>4</td>
<td>NP</td>
<td>Nested Paging: NCR3, G_PAT</td>
</tr>
<tr>
<td>3</td>
<td>TPR</td>
<td>V_TPR, V_IRQ, V_INTR_PRIO, V_IPG_TPR, V_INTR_MASKING, V_INTR_VECTOR (Offset 60h–67h)</td>
</tr>
<tr>
<td>2</td>
<td>ASID</td>
<td>ASID</td>
</tr>
<tr>
<td>1</td>
<td>IOPM</td>
<td>IOMSRPM: IOPM_BASE, MSRRM_BASE</td>
</tr>
<tr>
<td>0</td>
<td>I</td>
<td>Intercepts: all the intercept vectors, TSC offset, Pause Filter Count</td>
</tr>
</tbody>
</table>

**Figure 15-4. Layout of VMCB Clean Field**

Bits 31:12 are reserved for future implementations. For forward compatibility, if the hypervisor has not modified the VMCB, the hypervisor may write FFFF_FFFFh to the VMCB Clean Field to indicate
that it has not changed any VMCB contents other than the fields described below as explicitly uncached. The hypervisor should write 0h to indicate that the VMCB is new or potentially inconsistent with the CPU's cached copy, as occurs when the hypervisor has allocated a new location for an existing VMCB from a list of free pages and does not track whether that page had recently been used as a VMCB for another guest. If any VMCB fields (excluding explicitly uncached fields) have been modified, all clean bits that are undefined (within the scope of the hypervisor) must be cleared to zero.

The following are explicitly not cached and not represented by Clean bits:

- TLB_Control
- Interrupt shadow
- VMCB status fields (Exitcode, EXITINFO1, EXITINFO2, EXITINTINFO, Decode Assist, etc.)
- Event injection
- RFLAGS, RIP, RSP, RAX

15.16 TLB Control

TLB entries are tagged with Address Space Identifier (ASID) bits to distinguish different guest virtual address spaces when shadow page tables are used, or different guest physical address spaces when nested page tables are used. The VMM can choose a software strategy in which it keeps multiple shadow page tables, and/or multiple nested page tables in processors that support nested paging, up-to-date; the VMM can allocate a different ASID for each shadow or nested page table. This allows switching to a new process in a guest under shadow paging (changing CR3 contents), or to a new guest under nested paging (changing nCR3 contents), without flushing the TLBs. (See section 15.25 for a complete explanation of nested paging operation.)

With shadow paging, the VMM is responsible for setting up a shadow page table for each guest linear address space that maps it to system physical addresses. These are used as the active page tables in place of the guest OS's page tables. The VMM sets the CR3 field in the guest VMCB to point to the system physical address of the desired shadow page table. The VMM is responsible for updating the shadow page table when the guest changes its page table or paging control state, and the VMM updates the access and dirty bits of the guest page table.

The VMRUN instruction and #VMEXIT write the CR0, CR3, CR4 and EFER registers, but these writes do not flush the TLB. The VMM is responsible for explicitly invalidating any guest translations that may be affected by its actions. There are two mechanisms available for this described in the next two sections.

When running with SVM enabled, global page table entries (PTEs) are global only within an ASID, not across ASIDs.

**Software Rule.** When the VMM changes a guest’s paging mode by changing entries in the guest’s VMCB, the VMM must ensure that the guest’s TLB entries are flushed from the TLB. The relevant VMCB state includes:
• CR0—PG, WP, CD, NW.
• CR3—Any bit.
• CR4—PGE, PAE, PSE.
• EFER—NXE, LMA, LME.

15.16.1 TLB Flush

TLB flush operations function identically whether or not SVM is enabled (e.g., MOV-TO-CR3 flushes non-global mappings, whereas MOV-TO-CR4 flushes global and non-global mappings). TLB flush operations must not be assumed to affect all ASIDs. If a VMM sets the intercept bit for any guest action that would have flushed the TLB, the #VMEXIT intercept occurs and the TLB is not flushed; it is the VMM's responsibility to flush the TLB appropriately. In implementations that do not provide a way to selectively flush all translations of a single specified ASID, software may effectively flush the guest's TLB entries by allocating a new ASID for the guest and not reusing the old ASID until the entire TLB has been flushed at least once.

The TLB_CONTROL field in the VMCB provides the commands specified by the control byte encodings shown in Table 15-9. The first two commands are available on all processors that support SVM; support for the other commands is optional and is indicated by CPUID Fn8000_000A_EDX[FlushByAsid] = 1.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Function Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>Do not flush</td>
</tr>
<tr>
<td>01h</td>
<td>Flush entire TLB (Should be used only on legacy hardware.)</td>
</tr>
<tr>
<td>03h</td>
<td>Flush this guest's TLB entries</td>
</tr>
<tr>
<td>07h</td>
<td>Flush this guest's non-global TLB entries</td>
</tr>
</tbody>
</table>

Note: All encodings not defined in this table are reserved.

When the VMM sets the TLB_CONTROL field to 1, the VMRUN instruction flushes the TLB for all ASIDs, for both global and non-global pages. The VMRUN instruction reads, but does not change, the value of the TLB_CONTROL field.

A MOV-to-CR3, a task switch that changes CR3, or clearing or setting CR0.PG or bits PGE, PAE, PSE of CR4 affects only the TLB entries belonging to the current ASID, regardless of whether the operation occurred in host or guest mode. The current ASID is 0 when the CPU is not inside a guest context.

All TLB entries belonging to all ASIDs are flushed by SMI, RSM, MTRR modifications, IORR modifications, and access to other system MSRs that affect address translation.
If a hypervisor modifies a nested page table by decreasing permission levels, clearing present bits, or changing address translations and intends to return to the same ASID, it should use either TLB command 011b or 001b.

### 15.16.2 Invalidate Page, Alternate ASID

The INVLPGA instruction allows the VMM to selectively invalidate the TLB mapping for a given guest virtual page within a given ASID. The linear address is specified in the implicit register operand rAX; the ASID is specified in ECX. The input address is always interpreted as a guest virtual address, so INVLPGA is typically meaningful only when used with shadow page tables; it does not provide a means to invalidate a nested translation by guest physical address.

### 15.17 Global Interrupt Flag, STGI and CLGI Instructions

The global interrupt flag (GIF) is a bit that controls whether interrupts and other events can be taken by the processor. The STGI and CLGI instructions set and clear, respectively, the GIF. Table 15-10 shows how the value of the GIF affects how interrupts and exceptions are handled. Implementations may provide hardware support for virtualizing the GIF in nested virtualization scenarios; see section 15.33, for details.

**Table 15-10. Effect of the GIF on Interrupt Handling**

<table>
<thead>
<tr>
<th>Interrupt source</th>
<th>GIF==0</th>
<th>GIF ==1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug exception or trap, due to breakpoint register match</td>
<td>Ignored and discarded</td>
<td>Normal operation</td>
</tr>
<tr>
<td>Debug trace trap due to EFLAGS.TF</td>
<td>Normal operation</td>
<td>Normal operation</td>
</tr>
<tr>
<td>RESET</td>
<td>Normal operation</td>
<td>Normal operation</td>
</tr>
<tr>
<td>INIT</td>
<td>Held pending until GIF==1</td>
<td>Normal operation, see Table 15-12</td>
</tr>
<tr>
<td>NMI</td>
<td>Held pending until GIF==1</td>
<td>Normal operation, see Table 15-13</td>
</tr>
<tr>
<td>External SMI</td>
<td>Held pending until GIF==1</td>
<td>Normal operation, see Table 15-14</td>
</tr>
<tr>
<td>Internal SMI (I/O Trapping)</td>
<td>Ignored and discarded</td>
<td>Normal operation, see Table 15-14</td>
</tr>
<tr>
<td>INTR and vINTR</td>
<td>Held pending until GIF==1</td>
<td>Normal operation</td>
</tr>
<tr>
<td>#SX (Security Exception)</td>
<td>n/a(^1)</td>
<td>Normal operation</td>
</tr>
<tr>
<td>Machine Check</td>
<td>If possible (implementation-dependent), held pending until GIF==1, otherwise shutdown.</td>
<td>Normal operation</td>
</tr>
<tr>
<td>DBREQ# (enter HDT)</td>
<td>Normal operation</td>
<td>Normal operation</td>
</tr>
</tbody>
</table>

\(^1\) VM_CR.DPD always controls DBREQ
Table 15-10. Effect of the GIF on Interrupt Handling (continued)

<table>
<thead>
<tr>
<th>Interrupt source</th>
<th>GIF==0</th>
<th>GIF==1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A20M</td>
<td>Normal operation</td>
<td>Normal operation</td>
</tr>
<tr>
<td></td>
<td>(VM_CR.DIS_A20M controls A20 masking)</td>
<td></td>
</tr>
<tr>
<td>Other implementation-specific but non-architecturally-visible interrupts (STPCLK, IGNNE toggle, ECC scrub)</td>
<td>Normal operation</td>
<td>Normal operation</td>
</tr>
</tbody>
</table>

**Note:**
1. #SX is caused only by an INIT signal that has been “redirected” (i.e., converted to an #SX; see section 15.28); the conversion only happens when GIF==1, as the INIT is simply held pending otherwise.

### 15.18 VMMCALL Instruction

This instruction is meant as a way for a guest to explicitly call the VMM. No CPL checks are performed, so the VMM can decide whether to make this instruction legal at the user-level or not.

If VMMCALL instruction is not intercepted, the instruction raises a #UD exception.

### 15.19 Paged Real Mode

To facilitate virtualization of real mode, the VMRUN instruction may legally load a guest CR0 value with PE = 0 but PG = 1. Likewise, the RSM instruction is permitted to return to paged real mode. This processor mode behaves in every way like real mode, with the exception that paging is applied. The intent is that the VMM run the guest in paged-real mode at CPL0, and with page faults intercepted. The VMM is responsible for setting up a shadow page table that maps guest physical memory to the appropriate system physical addresses.

The behavior of running a guest in paged real mode without intercepting page faults to the VMM is undefined.

### 15.20 Event Injection

The VMM can inject exceptions or interrupts (collectively referred to as events) into the guest by setting bits in the VMCB’s EVENTINJ field prior to executing the VMRUN instruction. The format of the field is shown in Table 15-5. The encoding matches that of the EXITINTINFO field. When an event is injected by means of this mechanism, the VMRUN instruction causes the guest to take the specified exception or interrupt unconditionally before executing the first guest instruction.

Injected events are treated in every way as though they had occurred normally in the guest (in particular, they are recorded in EXITINTINFO) with the following exceptions:

- Injected events are not subject to intercept checks. (Note, however, that if secondary exceptions occur during delivery of an injected event, those exceptions are subject to exception intercepts.)
An injected NMI does not block delivery of further NMIs.

If the VMM attempts to inject an event that is impossible for the guest mode (e.g., a #BR exception when the guest is in 64-bit mode), the event injection will fail and no guest state instructions will be executed; VMRUN will immediately exit with an error code of VMEXIT_INVALID.

Injecting an exception (TYPE = 3) with vectors 3 or 4 behaves like a trap raised by INT3 and INTO instructions, respectively, in which case the processor checks the DPL of the IDT descriptor before dispatching to the handler.

Software interrupts cannot be properly injected if the processor does not support the NextRIP field. Support is indicated by CPUID F8000_000A_EDX[NRIPS] = 1. Hypervisor software should emulate the event injection of software interrupts if NextRIP is not supported.

Event injection does not support injection of intercepted #DB faults that are the result of a guest ICEBP instruction. ICEBP does not perform DPL checks, as does INTn injection. Hypervisor software should emulate the injection of ICEBP.

The fields in EVENTINJ are as follows:

- **VECTOR**—Bits 7:0. The 8-bit IDT vector of the interrupt or exception. If TYPE is 2 (NMI), the VECTOR field is ignored.
- **TYPE**—Bits 10:8. Qualifies the guest exception or interrupt to generate. Table 15-11 shows possible values and their corresponding interrupt or exception types. Values not indicated are unused and reserved.
- **EV (Error Code Valid)**—Bit 11. Set to 1 if the exception should push an error code onto the stack; clear to 0 otherwise.
- **V (Valid)**—Bit 31. Set to 1 if an event is to be injected into the guest; clear to 0 otherwise.
- **ERRORCODE**—Bits 63:32. If EV is set to 1, the error code to be pushed onto the stack, ignored otherwise.

![Figure 15-5. EVENTINJ Field in the VMCB](image)

### Table 15-11. Guest Exception or Interrupt Types

<table>
<thead>
<tr>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>External or virtual interrupt (INTR)</td>
</tr>
<tr>
<td>2</td>
<td>NMI</td>
</tr>
<tr>
<td>3</td>
<td>Exception (fault or trap)</td>
</tr>
<tr>
<td>4</td>
<td>Software interrupt (INTn instruction)</td>
</tr>
</tbody>
</table>

![Image showing the fields in EVENTINJ](image)
VMRUN exits with VMEXIT_INVALID error code if either:

- Reserved values of TYPE have been specified, or
- TYPE = 3 (exception) has been specified with a vector that does not correspond to an exception (this includes vector 2, which is an NMI, not an exception).

# 15.21 Interrupt and Local APIC Support

SVM hardware support is designed to ensure efficient virtualization of interrupts.

## 15.21.1 Physical (INTR) Interrupt Masking in EFLAGS

To prevent the guest from blocking maskable interrupts (INTR), SVM provides a VMCB control bit, V_INTR_MASKING, which changes the operation of EFLAGS.IF and accesses to the TPR by means of the CR8 register. While running a guest with V_INTR_MASKING cleared to zero:

- EFLAGS.IF controls both virtual and physical interrupts.

While running a guest with V_INTR_MASKING set to 1:

- The host EFLAGS.IF at the time of the VMRUN is saved and controls physical interrupts while the guest is running.
- The guest value of EFLAGS.IF controls virtual interrupts only.

## 15.21.2 Virtualizing APIC.TPR

SVM provides a virtual TPR register, V_TPR, for use by the guest; its value is loaded from the VMCB by VMRUN and written back to the VMCB by #VMEXIT. The APIC's TPR always controls the task priority for physical interrupts, and the V_TPR always controls virtual interrupts.

While running a guest with V_INTR_MASKING cleared to 0:

- Writes to CR8 affect both the APIC's TPR and the V_TPR register.
- Reads from CR8 operate as they would without SVM.

While running a guest with V_INTR_MASKING set to 1:

- Writes to CR8 affect only the V_TPR register.
- Reads from CR8 return V_TPR.

## 15.21.3 TPR Access in 32-Bit Mode

The mechanism for TPR virtualization described in section 15.21.2 applies only to accesses that are performed using the CR8 register. However, in 32-bit mode, the TPR is traditionally accessible only by using a memory-mapped register. Typically, a VMM virtualizes such TPR accesses by not mapping the APIC page addresses in the guest. A guest access to that region then causes a #PF intercept to the VMM, which inspects the guest page tables to determine the physical address and, after recognizing the physical address as belonging to the APIC, finally invokes software emulation code.
To improve the efficiency of TPR accesses in 32-bit mode, SVM makes CR8 available to 32-bit code by means of an alternate encoding of MOV TO/FROM CR8 (namely, MOV TO/FROM CR0 with a LOCK prefix). To achieve better performance, 32-bit guests should be modified to use this access method, instead of the memory-mapped TPR. (For details, see “MOV CRn” on page 377 of the AMD64 Programmer’s Reference Volume 3: General Purpose and System Instructions, order# 24594.)

The alternate encodings of the MOV TO/FROM CR8 instructions are available even if SVM is disabled in EFER.SVME. They are available in both 64-bit and 32-bit mode.

15.21.4 Injecting Virtual (INTR) Interrupts

Virtual Interrupts allow the host to pass an interrupt (#INTR) to a guest. While inside a guest, the virtual interrupt follows the same rules that a real interrupt follows (virtual #INTR is not taken until EFLAGS.IF is 1, the guest's TPR has enabled interrupts at the same priority as that of the pending virtual interrupt).

SVM provides an efficient mechanism by which the VMM can inject virtual interrupts into a guest:

- As described in Section 15.13.1, the VMM can intercept physical interrupts that arrive while a guest is running, by activating the INTR intercept in the VMCB.
- As described in Section 15.21.4, the VMM can virtualize the interrupt masking logic by setting the V_INTR_MASKING bit in the VMCB.
- The three VMCB fields V_IRQ, V_INTR_PRIO, and V_INTR_VECTOR indicate whether there is a virtual interrupt pending, and, if so, what its vector number and priority are. The VMRUN instruction loads this information into corresponding on-chip registers.
- The processor takes a virtual INTR interrupt if
  - V_IRQ and V_INTR_PRIO indicate that there is a virtual interrupt pending whose priority is greater than the value in V_TPR,
  - interrupts are enabled in EFLAGS.IF,
  - interrupts are enabled using GIF, and
  - the processor is not in an interrupt shadow (see Section 15.21.5).

The only other difference between virtual INTR handling and normal interrupt handling is that, in the latter case, the interrupt vector is obtained from the V_INTR_VECTOR register (as opposed to running an INTACK cycle to the local APIC).

- The V_IGN_TPR field in the VMCB can be set to indicate that the currently pending virtual interrupt is not subject to masking by TPR. The priority comparison against V_TPR is omitted in this case. This mechanism can be used to inject ExtINT-type interrupts into the guest.
- When the processor dispatches a virtual interrupt (through the IDT), V_IRQ is cleared after checking for intercepts of virtual interrupts and before the IDT is accessed.
- On #VMEXIT, V_IRQ is written back to the VMCB, allowing the VMM to track whether a virtual interrupt has been taken.
• Physical interrupts take priority over virtual interrupts, whether they are taken directly or through a #VMEXIT.
• On #VMEXIT, the processor clears its internal copies of V_IRQ and V_INTR_MASKING, so virtual interrupts do not remain pending in the VMM, and interrupt control reverts to normal.

15.21.5 Interrupt Shadows

The x86 architecture defines the notion of an interrupt shadow—a single-instruction window during which interrupts are not recognized. For example, the instruction after an STI instruction that sets EFLAGS.IF (from zero to one) does not recognize interrupts or certain debug traps. The VMCB INTERRUPT_SHADOW field indicates whether the guest is currently in an interrupt shadow. This information is saved on #VMEXIT and loaded on VMRUN.

15.21.6 Virtual Interrupt Intercept

When virtualizing interrupt handling, a VMM typically needs only gain control when new interrupts for a guest arrive or are generated, and when the guest issues an EOI (end-of-interrupt). In some circumstances, it may also be necessary for the VMM to gain control at the moment interrupts become enabled in the guest (i.e., just before the guest takes a virtual interrupt). The VMM can do so by enabling the VINTR intercept.

15.21.7 Interrupt Masking in Local APIC

When guests have direct access to devices, interrupts arriving at the local APIC can usually be dismissed only by the guest that owns the device causing the interrupt. To prevent one guest from blocking other guests’ interrupts (by never processing their own), the VMM can mask pending interrupts in the local APIC, so they do not participate in the prioritization of other interrupts.

SVM introduces the following APIC features:
• A 256-bit IER (interrupt enable) register is added to the local APIC. This register resets to all ones (enabling all 256 vectors). Software can read and write the IER by means of the memory-mapped APIC page.
• Only vectors that are enabled in the IER participate in the APIC computation of the highest-priority pending interrupt.
• The VMM can issue specific end-of-interrupt (EOI) commands to the local APIC, allowing the VMM to clear pending interrupts in any order, rather than always targeting the interrupt with highest-priority.

15.21.8 INIT Support

The INIT signal interrupts the processor at the next instruction boundary and causes an unconditional control transfer. INIT reinitializes the control registers, segment registers and GP registers in a manner similar to RESET, but does not alter the contents of most MSRs, caches or numeric coprocessor (x87 or SSE) state, and then transfers control to the same instruction address as RESET (physical address.
Unlike RESET, INIT is not expected to be visible to the memory controller, and hence will not trigger automatic clearing of trusted memory pages by memory controller hardware.

To maintain the security of such pages, the VMM can request that INITs be redirected and turned into #SX exceptions by setting the R_INIT bit in the VM_CR MSR (see Section 15.30.1). This allows the VMM to gain control when an INIT is requested. The VMM may thus disable the redirection of INIT and then cause the platform to reassert INIT, at which point the processor will respond in the normal manner. The actions initiated by the INIT pin may also be initiated by an incoming APIC INIT interrupt; the mechanisms described here apply in either case. Table 15-12 summarizes the handling of INITs.

<table>
<thead>
<tr>
<th>GIF</th>
<th>INIT Intercept</th>
<th>INIT Redirect</th>
<th>Processor Response to INIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>x</td>
<td>Hold pending until GIF = 1.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>x</td>
<td>#VMEXIT(INIT), INIT is still pending.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>Taken normally.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>#SX, INIT is no longer pending.</td>
</tr>
</tbody>
</table>

**15.21.9 NMI Support**

The VMM can intercept non-maskable interrupts (NMI) using a VMCB control bit (see Table 15-13). When intercepted, NMIs cause an exit from the guest and are held pending.

<table>
<thead>
<tr>
<th>GIF</th>
<th>NMI Intercept</th>
<th>Processor Response to NMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>Hold pending until GIF=1.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>#VMEXIT(NMI), NMI is still pending.</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Taken normally.</td>
</tr>
</tbody>
</table>

**15.22 SMM Support**

This section describes SVM support for virtualization of System Management Mode (SMM).

**15.22.1 Sources of SMI**

Various events can cause an assertion of a system management interrupt (SMI); these are classified into three categories:

- Internal, synchronous (also known as I/O Trapping)—implementation-specific IOIO or config space trapping in the CPU itself; always synchronous in response to an IN or OUT instruction. I/O Trapping is set up by means of MSRs and can be brought under the control of the VMM by intercepting guest access to those MSRs.
• External, synchronous—IOIO trapping in response to (and synchronous with) IN or OUT instructions, but generated by an external agent (typically the Southbridge).
• External, asynchronous—generated externally in response to an external, physical event, e.g., closing a laptop lid, temperature sensor triggering, etc.

15.22.2 Response to SMI

How hardware responds to SMIs is a function of whether SMM interrupts are being intercepted and whether interrupts are enabled globally, as shown in Table 15-14.

Table 15-14. SMI Handling in Different Operating Modes

<table>
<thead>
<tr>
<th>GIF</th>
<th>Intercept SMI</th>
<th>Internal SMI</th>
<th>External SMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>Lost.</td>
<td>Hold pending until GIF=1.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Exit guest,</td>
<td>#VMEXIT(SMI), SMI is still pending.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>code #VMEXIT(SMI), SMI is not pending.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Taken normally.</td>
<td></td>
<td>Taken normally.</td>
</tr>
</tbody>
</table>

By intercepting SMIs, the VMM can gain control before the processor enters SMM.

15.22.3 Containerizing Platform SMM

In some usage scenarios, the VMM may not trust the existing platform SMM code, or may otherwise want to ensure that the SMM does not operate in the context of certain guests or the hypervisor. To address these cases, SVM provides the ability to containerize SMM code, i.e., run it inside a guest, with the full protection mechanisms of the VMM in place. In other scenarios, the VMM may not want to exert control over SMM.

There are three solutions for the VMM to control SMM handlers:

• The simplest solution is to not intercept SMI signals. SMIs encountered while in a guest context are taken from within the guest context. In this case, the SMM handler is not subject to any intercepts set up by the VMM and consequently runs outside of the virtualization controls. The state saved in the SMM State-Save area as seen by the SMM handler reflects the state of the guest that had been running at the time the SMI was encountered. When the SMM handler executes the RSM instruction, the processor returns to executing in the guest context, and any modifications to the SMM State-Save area made by the SMM handler are reflected in the guest state.

• A hypervisor may want to emulate all SMI-based I/O interceptions for a guest and to take SMI signals only in the hypervisor context. The hypervisor should set all IOIO intercept bits and the SMI intercept bit for the guest to ensure that there is no possibility of encountering synchronous (internal or external) SMI signals while running the guest. Any #VMEXIT(SMI) encountered is then known to be due to an external, asynchronous SMI. The hypervisor may respond to the #VMEXIT(SMI) by executing the STGI instruction, which causes the pending SMI to be taken immediately. When an SMI due to an I/O instruction is pending, the effect of executing STGI in
the hypervisor is undefined. To handle a pending SMI due to an I/O instruction, the hypervisor must either containerize SMM or not intercept SMI.

- The most involved solution is to containerize SMM by placing it in a guest. Containerizing gives the VMM full control over the state that the SMM handler can access.

**Containerizing Platform SMM.** A VMM can containerize SMM by creating its own trusted SMM hypervisor and use that handler to run the platform SMM code in a container. The SMM hypervisor may be the same code as the VMM itself, or may be an entirely different set of code. The trusted SMM hypervisor sets up a guest context to run the platform SMM as a guest. The guest context consists of a VMCB and related state and the guest's (real or virtual) SMM save area. The SMM hypervisor emulates SMM entry, including setup of the SMM save area, and emulates RSM at the end of SMM operation. The guest executes the platform SMM code in paged real mode with appropriate SVM intercepts in place, thus ensuring security.

For this approach to work, the VMM may need to write the SMM_BASE MSR, as well as related SMM control registers. As part of the emulation of SMM entry and RSM, the VMM needs to access the SMM_CTL MSR (see Section 15.30.3). However, these actions conflict with any platform firmware that locks SMM control registers.

A VMM can determine if it is running with a compatible firmware setup by checking the SMMLOCK bit in the HWCR MSR (described in the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product). If the bit is 1, firmware has locked the SMM control registers and the VMM is unable to move them or insert its own SMM hypervisor.

As the processor physically enters SMM, the SMRAM regions are remapped. The VMM design must ensure that none of its code or data disappears when the SMRAM areas are mapped or unmapped. Also note that the ASEG region of the SMRAM overlaps with a portion of video memory, so the SMM hypervisor should not attempt to write diagnostic messages to the screen. Any attempt by guests to relocate any of the SMRAM areas (by means of certain MSR writes) must also be intercepted to prevent malicious SMM code from interfering with VMM operation.

Writes to the SMM_CTL MSR cause a #GP if firmware has locked the SMM control registers.

### 15.23 Last Branch Record Virtualization

The debug control MSR (DebugCtl) provides control of control-transfer recording and other debug facilities. (See Chapter 13, “Software Debug and Performance Resources,” on page 355, for more information on using the debug control MSR.) Software sets the last-branch record (DebugCtl[LBR]) bit to 1 to cause the processor to record the source and target addresses of the last control transfer taken before a debug exception. These control transfers include branch instructions, interrupts, and exceptions. Recorded information is stored in four MSRs:

- LastBranchFromIP
- LastBranchToIP
- LastIntFromIP
Secure Virtual Machine

15.23.1 Hardware Acceleration for LBR Virtualization

Processors optionally support hardware acceleration for LBR virtualization. The following fields are allocated in the VMCB state save area to hold the contents of the DebugCTL and control-transfer recording MSRs:

- **DBGCTL**—Holds the guest value of the DebugCTL MSR.
- **BR_FROM**—Holds the guest value of the LastBranchFromIP MSR.
- **BR_TO**—Holds the guest value of the LastBranchToIP MSR.
- **LASTEXCPFROM**—Holds the guest value of the LastIntFromIP MSR.
- **LASTEXCPTO**—Holds the guest value of the LastIntToIP MSR.

When VMCB.LBR_VIRTUALIZATION_ENABLE is set, VMRUN saves all five host control-transfer MSRs in the host save area, and then loads the same five MSRs for the guest from the VMCB save area. Similarly, #VMEXIT saves the guest's MSRs and loads the host's MSRs to and from their respective save areas.

15.23.2 LBR Virtualization CPUID Feature Detection

CPUID Fn8000_000A_EDX[LbrVirt] = 1 indicates support for the LBR virtualization acceleration feature on AMD64 processors. See Section 3.3, “Processor Feature Identification,” on page 64 for more information on using the CPUID instruction.

15.24 External Access Protection

By securing the virtual address translation mechanism, the VMM can restrict guest CPU accesses to memory. However, should the guest have direct access to DMA-capable devices, an additional protection mechanism is required. SVM provides multiple protection domains which can restrict device access to physical memory on a per-page basis. This is accomplished via control logic in the northbridge’s host bridge which governs any external access port (e.g., PCI or HyperTransport™ technology interfaces).

15.24.1 Device IDs and Protection Domains

The northbridge’s host bridge provides a number of protection domains. Each protection domain has associated with it a device exclusion vector (DEV) that specifies the per-page access rights of devices in that domain. Devices are identified by a HyperTransport™ bus/unitID (device ID) and the host bridge contains a lookup table of fixed size that maps device IDs to a protection domain.
15.24.2 Device Exclusion Vector (DEV)

A DEV is a contiguous array of bits in physical memory; each bit in the DEV (in little-endian order) corresponds to one 4-Kbyte page in physical memory.

The physical address of the base of a DEV must be 4-Kbyte-aligned and stored in one of the DEVBASE registers, which are accessed through an indirection mechanism in the DEVCTL PCI Configuration Space function block in the host bridge (see “DEV Control and Status Registers” on page 497). The DEV protection hardware is not operational until enabled by setting a control bit in the DEV Control Register, also in the DEVCTL function block.

The DEV may have to cover part of MMIO space beyond the DRAM. Especially in 64-bit systems, the operating system should map MMIO space starting immediately after the DRAM area and building up, as opposed to starting down from the maximum physical address.

**Host Bridge and Processor DEV Caching.** For improved performance, the host bridge may cache portions of the DEV. Any such cached information can be invalidated by setting the DEV_FLUSH flag in the DEV control register to 1. Software must set this flag after modifying DEV contents to ensure that the protection logic uses the updated values. The host bridge automatically clears this flag when the flush operation completes. After setting this flag, software should monitor it until it has cleared, in order to synchronize DEV updates with subsequent activity.

By default, the host bridge probes the processor caches for the latest data when it accesses the DEV in DRAM. However, it is possible to disable probing by means of the DEV_CR register (“DEV_CR Register” on page 497); this is recommended in the case of unified memory architecture (UMA) graphics systems. If cache probing is disabled, host bridge reads of the DEV will not check processor caches for more recent copies. This requires software on the CPU to map the memory containing the DEV as uncacheable (UC) or write-through (WT). Alternatively, software must perform a CLFLUSH before it can expect a change to the DEV to be visible by the northbridge (and before software flushes the DEV cache in the host controller).

**Multiprocessor Issues.** Device-originated memory requests are checked against the DEV at the point of entry to the system—the northbridge to which the device is physically attached. Each northbridge can have its own set of domains, device-to-domain mappings, and DEV tables (e.g., domain #2 on one node can encompass different devices, and can have different access rights than domain #2 on another node). Thus, the number of protection domains available to software can scale with the number of northbridges in the system.

15.24.3 Access Checking

**Memory Space Accesses.** When a memory-space read or write request is received on an external host bridge port, the host bridge maps the HyperTransport bus device ID to a protection domain number, which in turn selects the DEV defining the access permissions for the device (see Figure 15-6). The host bridge then checks the memory address against the DEV contents by indexing into the DEV with the PFN portion of the address (bits 39:12). The PFN is used as a bit index within the DEV. If the bit read from the DEV is set to 1, the host bridge inhibits the access by returning all
ones for the data for a read request, or suppressing the store operation on a write request. A Master Abort error response will be returned to the requesting device.

Peer-to-peer memory accesses routed up to the host bridge are also subjected to checks against the DEV. Peer-to-peer transfers that may be occurring behind bridges are not checked.

DEV checks are applied before addresses are translated by the GART. The DEV table is never consulted by accesses originating in the CPU.

**I/O Space Accesses.** The host bridge can be configured to reject all I/O space accesses from devices, by setting the IOSPE bit in the DEV_CR control register (see “DEV_CR Register” on page 497). I/O space peer-to-peer transfers behind bridges are not checked.

**Config Space Accesses.** Major aspects of host bridge functionality are configured by means of control registers that are accessed through PCI configuration space. Because this is potentially accessible by means of device peer-to-peer transfers, the host bridge always blocks access to this space from anything other than the CPU.

![Diagram of Host Bridge DMA Checking](image)

**Figure 15-6. Host Bridge DMA Checking**
15.24.4 DEV Capability Block

The presence of DEV support is indicated through a new PCI capability block. The capability block also provides access to the registers that control operation of the DEV feature.

The DEV capability block in PCI space contains three 32-bit words: the capability header (DEV_HDR), and two registers (DEV_OP and DEV_DATA) which serve as an indirection mechanism for accessing the actual DEV control and status registers.

Table 15-15. DEV Capability Block, Overall Layout

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Register</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DEV_HDR</td>
<td>Capability block header</td>
</tr>
<tr>
<td>4</td>
<td>DEV_OP</td>
<td>Selects control/status register to access</td>
</tr>
<tr>
<td>8</td>
<td>DEV_DATA</td>
<td>Read/write to access register selected in DEV_OP</td>
</tr>
</tbody>
</table>

DEV Capability Header. The DEV capability header (DEV_HDR) is defined in Table 15-16.

Table 15-16. DEV Capability Header (DEV_HDR) (in PCI Config Space)

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:22</td>
<td>Reserved, MBZ</td>
</tr>
<tr>
<td>21</td>
<td>Interrupt Reporting Capability</td>
</tr>
<tr>
<td>20</td>
<td>Machine Check Exception Reporting Capability</td>
</tr>
<tr>
<td>19</td>
<td>Reserved, MBZ</td>
</tr>
<tr>
<td>18:16</td>
<td>DEV Capability Block Type; hardwired to 000b.</td>
</tr>
<tr>
<td>15:8</td>
<td>PCI Capability pointer; points to next capability in list</td>
</tr>
<tr>
<td>7:0</td>
<td>PCI Capability ID; hardwired to 0x0F</td>
</tr>
</tbody>
</table>

15.24.5 DEV Register Access Mechanism

The northbridge’s DEV control and status registers are accessed through an indirection mechanism: writing the DEV_OP register selects which internal register is to be accessed, and the DEV_DATA register can be read or written to access the selected register.

Figure 15-7 shows the format of the DEV_OP register. The DEV_DATA register reflects the format of the DEV register selected in DEV_OP.

Figure 15-7. Format of DEV_OP Register (in PCI Config Space)
The FUNCTION field in the DEV_OP register selects the function/register to read or write according to the encoding in Table 15-17; for blocks of registers that have multiple instances (e.g., multiple DEV_BASE_HI/LO registers), the INDEX field selects the instance; otherwise it is ignored.

### Table 15-17. Encoding of Function Field in DEV_OP Register

<table>
<thead>
<tr>
<th>Function Code</th>
<th>Register Type</th>
<th>Number of Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DEV_BASE_LO</td>
<td>multiple</td>
</tr>
<tr>
<td>1</td>
<td>DEV_BASE_HI</td>
<td>multiple</td>
</tr>
<tr>
<td>2</td>
<td>DEV_MAP</td>
<td>multiple</td>
</tr>
<tr>
<td>3</td>
<td>DEV_CAP</td>
<td>single</td>
</tr>
<tr>
<td>4</td>
<td>DEV_CR</td>
<td>single</td>
</tr>
<tr>
<td>5</td>
<td>DEV_ERR_STATUS</td>
<td>single</td>
</tr>
<tr>
<td>6</td>
<td>DEV_ERR_ADDR_LO</td>
<td>single</td>
</tr>
<tr>
<td>7</td>
<td>DEV_ERR_ADDR_HI</td>
<td>single</td>
</tr>
</tbody>
</table>

For example, to write the DEV_BASE_HI register for protection domain number 2, software sets DEV_OP.FUNCTION to 1, and DEV_OP.INDEX to 2, and then writes the desired 32-bit value into DEV_DATA. As the DEV_OP and DEV_DATA registers are accessed through PCI config space (ports 0CF8h–0CFFh), they may be secured from unauthorized access by software executing on the processor by appropriate settings in the SVM I/O protection bitmap. These registers are also protected by the host bridge from external access as described in “Config Space Accesses” on page 495.

#### 15.24.6 DEV Control and Status Registers

The DEV control and status registers are accessible by means of the indirection mechanism; these registers are *not* directly visible in PCI config space.

**DEV_CAP Register.** Read-only register; holds implementation specific information: the number of protection domains supported, the number of DEV_MAP registers (which map device/unit IDs to domain numbers), and the revision ID.

```
<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved, RAZ</td>
<td>N_MAPS</td>
<td>N_DOMAINS</td>
<td>REVISION</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Figure 15-8. Format of DEV_CAP Register (in PCI Config Space)**

The initial implementation provide four domains and three map registers.

**DEV_CR Register.** This is the main control register for the DEV mechanism; it is cleared to zero by RESET.
### DEV_BASE Address/Limit Registers

The DEV base address registers (one set per domain) each point to the physical address of a DEV table corresponding to a protection domain. The address and size are encoded in a pair (high/low) of 32-bit registers. The N_DOMAINS field in DEV_CAP indicates how many (pairs of) DEV_BASE registers are implemented. The register format is as shown in Figures 15-9 and 15-10.

#### Table 15-18. DEV_CR Control Register

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:7</td>
<td>Reserved, MBZ</td>
</tr>
</tbody>
</table>
| 6      | DEV table walk probe disable.  
|        | 0 = Use probe on DEV walk; 1 = Do not use probe |
| 5      | SL_DEV_EN. Enable bit for limited memory protection, see Section 15.24.8 on page 499. Set to “1” by SKINIT instruction, can be cleared by software. |
| 4      | Invalidate DEV cache. Software must set this bit to 1 to invalidate the DEV cache; cleared by hardware when invalidation is complete. |
| 3      | Enable MCE reporting.  
|        | 0 = Do not generate MCE; 1 = Generate MCE on errors. |
| 2      | I/O space protection enable (IOSPEN)  
|        | 0 = Allow upstream I/O cycles; 1 = Block. |
| 1      | Memory clear disable. If non-zero, memory-clearing on reset is disabled.  
|        | This bit is not writable until the memory is enabled. |
| 0      | DEV global enable bit. If zero, DEV protection is turned off. |

#### Figure 15-9. Format of DEV_BASE_HI[n] Registers

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved, MBZ</td>
<td>BASEADDRESS[39:32]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Figure 15-10. Format of DEV_BASE_LO[n]Registers

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASEADDRESS[31:12]</td>
<td>Reserved, MBZ</td>
<td>SIZE</td>
<td>P</td>
<td>V</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fields of the DEV_BASE_HI and DEV_BASE_LO registers are defined as follows:

- **Valid (V)**—Bit 0. Indicates whether a DEV table has been defined for the given protection domain; if this bit is clear, software can leave the other fields undefined, and no protection checks are performed for memory references in this domain.
Secure Virtual Machine

- **Protect (P)**—Bit 1. Indicates whether accesses to addresses beyond the address range covered by the DEV are legal (P=0) or illegal (P=1).
- **SIZE**—Bits 6:2. Specifies how much memory the DEV covers, expressed increments of \(4\text{GB} \times 2^{\text{size}}\). In other words, a DEV table covers a minimum of 4GB, and can expand by powers of two.

**DEV_MAP Registers.** The DEV_MAP registers assign protection domain numbers to device-originated requests by matching the device ID (HT bus and unit number) associated with the request against bus and unit numbers in the registers. If no match is found in any of the registers, a domain number of zero is returned. The number of DEV_MAP registers implemented by the chip is indicated by the N_MAPS field in DEV_CAP.

The format of the DEV_MAP registers is shown in Figure 15-11.

![Figure 15-11. Format of DEV_MAP[n] Registers](image)

The fields of the DEV_MAP[n] registers are defined as follows:

- **UNIT0**—Bits 4:0. Specifies the first of two HyperTransport link unit numbers on the bus number specified by the BUSNO field.
- **V0**—Bit 5. Indicates whether UNIT0 is valid (no matches occur on invalid entries).
- **UNIT1**—Bits 10:6. Specifies the second of two HyperTransport link unit numbers on the bus number specified by the BUSNO field.
- **V1**—Bit 11. Indicates whether UNIT1 is valid (no matches occur on invalid entries).
- **BUSNO**—Bits 19:12. Specifies a HyperTransport link bus number.
- **DOM0**—Bits 25:20. Specifies the protection domain for the first HyperTransport link unit.
- **DOM1**—Bits 31:26. Specifies the protection domain for the second HyperTransport link unit.

**15.24.7 Unauthorized Access Logging**

Any attempted unauthorized access by devices to DEV-protected memory is logged by the host bridge in the DEV_Error_Status and DEV_Error_Address registers for possible inspection by the VMM.

**15.24.8 Secure Initialization Support**

The host bridge contains additional logic that operates in conjunction with the SKINIT instruction to provide a limited form of memory protection during the secure startup protocol. This provides protection for a Secure Loader image in memory, allowing it to, among other things, set up full DEV protection. (See Section 15.27 for detailed operation of SKINIT.)
The host bridge logic includes a hidden (not accessible to software) SL_DEV_BASE address register. SL_DEV_BASE points to a 64KB-aligned 64KB region of physical memory. When SL_DEV_EN is 1, the 64KB region defined by SL_DEV_BASE is protected from external access (as if it were protected by the DEV), as well as from any access (both CPU and external accesses) via GART-translated addresses. Additionally, the SL_DEV mechanism, when enabled, blocks all device accesses to PCI Configuration space.

15.25 Nested Paging

The optional SVM nested paging feature provides for two levels of address translation, thus eliminating the need for the VMM to maintain shadow page tables.

15.25.1 Traditional Paging versus Nested Paging

Figure 15-12 shows how a page in the linear address space is mapped to a page in the physical address space in traditional (single-level) address translation. Control register CR3 contains the physical address of the base of the page tables (PT, represented by the shaded box in the figure), which governs the address translation.

![Figure 15-12. Address Translation with Traditional Paging](image)

With nested paging enabled, two levels of address translation are applied; refer to Figure 15-13 below.

- Both guest and host levels have their own copy of CR3, referred to as gCR3 and nCR3, respectively.
- Guest page tables (gPT) map guest linear addresses to guest physical addresses. The guest page tables are in guest physical memory, and are pointed to by gCR3.
- Nested page tables (nPT) map guest physical addresses to system physical addresses. The nested page tables are in system physical memory, and are pointed to by nCR3.
- The most-recently used translations from guest linear to system physical address are cached in the TLB and used on subsequent guest accesses.
It is important to note that gCR3 and the guest page table entries contain guest physical addresses, not system physical addresses. Hence, before accessing a guest page table entry, the table walker first translates that entry’s guest physical address into a system physical address.

Figure 15-13. Address Translation with Nested Paging

The VMM can give each guest a different ASID, so that TLB entries from different guests can coexist in the TLB. The ASID value of zero is reserved for the host; if the VMM attempts to execute VMRUN with a guest ASID of zero, the result is #VMEXIT(VMEXIT_INVALID). Note that because an ASID is associated with the guest's physical address space, it is common across all of the guest's virtual address spaces within a processor. This differs from shadow page tables where ASIDs tag individual guest virtual address spaces. Note also that the same ASID may or may not be associated with the same address space across all processors in a multiprocessor system, for either nested tables or shadow tables; this depends on how the VMM manages ASID assignment.

15.25.2 Replicated State

Most processor state affecting paging is replicated for host and guest. This includes the paging registers CR0, CR3, CR4, EFER and PAT. CR2 is not replicated but is loaded by VMRUN. The MTRRs are not replicated.

While nested paging is enabled, all (guest) references to the state of the paging registers by x86 code (MOV to/from CRn, etc.) read and write the guest copy of the registers; the VMM's versions of the
registers are untouched and continue to control the second level translations from guest physical to system physical addresses. In contrast, when nested paging is disabled, the VMM’s paging control registers are stored in the host state save area and the paging control registers from the guest VMCB are the only active versions of those registers.

15.25.3 Enabling Nested Paging

The VMRUN instruction enables nested paging when the NP_ENABLE bit in the VMCB is set to 1. The VMCB contains the hCR3 value for the page tables for the extra translation. The extra translation uses the same paging mode as the VMM used when it executed the most recent VMRUN.

Nested paging is automatically disabled by #VMEXIT.

Nested paging is allowed only if the host has paging enabled. Support for nested paging is indicated by CPUID Fn8000_000A_EDX[NP] = 1. If VMRUN is executed with hCR0.PG cleared to zero and NP_ENABLE set to 1, VMRUN terminates with #VMEXIT(VMEXIT_INVALID). See Section 3.3, “Processor Feature Identification,” on page 64 for more information on using the CPUID instruction.

15.25.4 Nested Paging and VMRUN/#VMEXIT

When VMRUN is executed with nested paging enabled (NP_ENABLE = 1), the paging registers are affected as follows:

- VMRUN saves the VMM’s CR3 in the host save area.
- VMRUN loads the guest paging state from the guest VMCB into the guest registers (i.e., VMRUN loads CR3 with the VMCB CR3 field, etc.). The guest PAT register is loaded from G_PAT field in the VMCB.
- VMRUN loads nCR3, the version of CR3 to be used while the nested-paging guest is running, from the N_CR3 field in the VMCB. The other host paging-control bits (hCR4.PAE, etc.) remain the same as they were in the VMM at the time VMRUN was executed.

When VMRUN is executed with nested paging enabled (NP_ENABLE = 1), the following conditions are considered illegal state combinations, in addition to those mentioned in “Canonicalization and Consistency Checks” on page 459:

- Any MBZ bit of nCR3 is set.
- Any G_PAT.PA field has an unsupported type encoding or any reserved field in G_PAT has a non-zero value. (See Section 7.8.1, “PAT Register,” on page 204.)

When #VMEXIT occurs with nested paging enabled:

- #VMEXIT writes the guest paging state (gCR3, gCR0, etc.) back into the VMCB. nCR3 is not saved back into the VMCB.
- #VMEXIT need not reload any host paging state other than CR3 from the host save area, though an implementation is free to do so.
15.25.5 Nested Table Walk

When the guest is running with nested paging enabled, a TLB miss causes several nested table walks:

- **Guest Page Tables**—the gCR3 register specifies a guest physical address, as do the entries in the guest's page tables. These guest physical addresses must be translated to system physical addresses using the nested page tables. Nested page table level faults can occur on these accesses, including write faults due to setting of accessed and dirty bits in the guest page table.

- **Final Guest-Physical Page**—once a guest linear to guest physical mapping is known, guest permissions can be checked. If the guest page tables allow the access, the guest physical address is walked in the nested page tables to find the system physical address.

Table walks for guest page tables are always treated as user writes at the nested page table level. For this reason,

- the page must be writable by user at the nested page table level, or else a #VMEXIT(NPF) is raised, and
- the dirty and accessed bits are always set in the nested page table entries that were touched during nested page table walks for guest page table entries.

A table walk for the guest page itself is always treated as a user access at the nested page table level, but is treated as a data read, data write, or code read, depending on the guest access.

If the guest has paging disabled (gCR0.PG = 0), there are no guest page table entries to be translated in the nested page tables. In this case, the final guest-physical address is equal to the guest-linear address, and is still translated in the nested page tables.

15.25.6 Nested versus Guest Page Faults, Fault Ordering

In nested paging, page faults can be raised at either the guest or nested page table level. Nested walks proceed in the following order; faults are generated in the same order:

1. Walk the guest page table entries in the nested page table. Dirty/Accessed bits are set as needed in the nested page table. Any nested page table faults result in #VMEXIT(NPF).

2. As the guest page table walk proceeds from the top of the page table to the last entry, any not-present entries or reserved bits in the guest page table entries at each level of the guest walk cause #PF in the guest. Guest dirty and accessed bits are set as needed in the guest page tables during the walk. Steps 1 and 2 are repeated for each level of the guest page table that is traversed.

3. Once the guest physical address for the guest access has been determined, check the guest permissions; any fault at this point causes a #PF in the guest.

4. Perform the final translation from guest physical to system physical using the nested page table; any fault during this translation results in a #VMEXIT(NPF).

Nested page faults are entirely a function of the nested page table and VMM processor mode. Nested faults cause a #VMEXIT(NPF) to the VMM. The faulting guest physical address is saved in the VMCB's EXITINFO2 field; EXITINFO1 delivers an error code similar to a #PF error code:
• Bit 0 (P)—cleared to 0 if the nested page was not present, 1 otherwise
• Bit 1 (RW)—set to 1 if the nested page table level access was a write. Note that host table walks for guest page tables are always treated as data writes.
• Bit 2 (US)—set to 1 if the nested page table level access was a user access. Note that nested page table accesses performed by the MMU are treated as user accesses unless there are features enabled that override this.
• Bit 3 (RSV)—set to 1 if reserved bits were set in the corresponding nested page table entry
• Bit 4 (ID)—set to 1 if the nested page table level access was a code read. Note that nested table walks for guest page tables are always treated as data writes, even if the access itself is a code read

In addition, the VMCB contents for nested page faults indicate whether the page fault was encountered during the nested page table walk for a guest page TLB entry, or for the final nested walk for the guest physical address, as indicated by EXITINFO1[33:32]:
• Bit 32—set to 1 if nested page fault occurred while translating the guest’s final physical address
• Bit 33—set to 1 if nested page fault occurred while translating the guest page tables

Guest faults are entirely a function of the guest page tables and processor mode; they are delivered to the guest as normal #PF exceptions without any VMM intervention, unless the VMM is intercepting guest #PF exceptions. Bits 32 and 33 of EXITINFO1 are written during nested page faults to indicate whether the page fault was encountered during the nested page table walk for a guest page table's table entries, or if the fault was encountered during the nested page table walk for the translation of the final guest physical address.

The processor may provide additional instruction decode assist information. (See section 15.10.)

15.25.7 Combining Nested and Guest Attributes

Any access to guest physical memory is subjected to a permission check by examining the mapping of the guest physical address in the nested page table.

A page is considered writable by the guest only if it is marked writable at both the guest and nested page table levels. Note that the guest’s gCR0.WP affects only the interpretation of the guest page table entry; setting gCR0.WP cannot make a page writable at any CPL in the guest, if the page is marked read-only in the nested page table. The host hCR0.WP bit is ignored under nested paging.

A page is considered executable by the guest only if it is marked executable at both the guest and nested page table levels. If the EFER.NXE bit is cleared for the guest, all guest pages are executable at the guest level. Similarly, if the EFER.NXE bit is cleared for the host, all nested page table mappings are executable at the underlying nested level.

Some attributes are taken from the guest page tables and operating modes only. A page is considered global within the guest only if is marked global in the guest page tables; the nested page table entry and host hCR4.PGE are irrelevant. Global pages are only global within their ASID.
A page is considered user in the guest only if it is marked as user at the guest level. The page must be marked user in the nested page table to allow any guest access at all.

15.25.8 Combining Memory Types, MTRRs

When nested paging is disabled, the processor behaves as though there is no gPAT register.

The host PAT MSR determines memory type attributes for the current VM, and guest writes to the PAT MSR that aren't intercepted by the VMM will alter the host PAT MSR. The hypervisor is responsible for context-switching the PAT MSR contents on world switches between VM's.

When nested paging is enabled, the processor combines guest and nested page table memory types. Registers that affect memory types include:

- The PCD/PWT/PATi bits in the nested and guest page table entries.
- The PCD/PWT bits in the nested CR3 and guest CR3 registers.
- The guest PAT type (obtained by appropriately indexing the gPAT register).
- The host PAT type (obtained by appropriately indexing the host’s PAT register).
- The MTRRs (which are referenced based only on system physical address).
- gCR0.CD and hCR0.CD.

Note that there is no hardware support for guest MTRRs; the VMM can simulate their effect by altering the memory types in the nested page tables. Note that the MTRRs are only applied to system physical addresses.

The rules for combining memory types when constructing a guest TLB entry are:

- Nested and guest PAT types are combined according to Table 15-19, producing a “combined PAT type.”
- Combined PAT type is further combined with the MTRR type according to Table 15-20, where the relevant MTRRs are determined by the system physical address.
- Either gCR0.CD or hCR0.CD can disable caching.

Memory Consistency Issues. Because the guest uses extra fields to determine the memory type, the VMM may use a different memory type to access a given piece of memory than does the guest. If one access is cacheable and the other is not, the VMM and guest could observe different memory images, which is undesirable. (MP systems are particularly sensitive to this problem when the VMM desires to migrate a virtual processor from one physical processor to another.)

To address this issue, the following mechanisms are provided:

- VMRUN and #VMEXIT flush the write combiners. This ensures that all writes to WC memory by the guest are visible to the host (or vice-versa) regardless of memory type. (It does not ensure that cacheable writes by one agent are properly observed by WC reads or writes by the other agent.)
- A new memory type WC+ is introduced. WC+ is an uncachable memory type, and combines writes in write-combining buffers like WC. Unlike WC (but like the CD memory type), accesses to
WC+ memory also snoop the caches on all processors (including self-snooping the caches of the processor issuing the request) to maintain coherency. This ensures that cacheable writes are observed by WC+ accesses.

- When combining nested and guest memory types that are incompatible with respect to caching, the WC+ memory type is used instead of WC (and Table 15-20 ensures that the snooping behavior is retained regardless of the host MTRR settings). Refer to Table 15-19 or details.

Table 15-19 shows how guest and host PAT types are combined into an effective PAT type. When interpreting this table, recall (a) that guest and host PAT types are not combined when nested paging is disabled and (b) that the intent is for the VMM to use its PAT type to simulate guest MTRRs.

### Table 15-19. Combining Guest and Host PAT Types

<table>
<thead>
<tr>
<th>Guest PAT Type</th>
<th>Host PAT Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC</td>
<td>UC UC UC UC UC UC</td>
</tr>
<tr>
<td>UC–</td>
<td>UC UC UC UC UC UC</td>
</tr>
<tr>
<td>WC</td>
<td>WC WC WC WC+ WC+ WC+</td>
</tr>
<tr>
<td>WP</td>
<td>UC UC UC WP UC WP</td>
</tr>
<tr>
<td>WT</td>
<td>UC UC UC UC WT WT</td>
</tr>
<tr>
<td>WB</td>
<td>UC UC WC WP WT WB</td>
</tr>
</tbody>
</table>

The existing AMD64 table that defines how PAT types are combined with the physical MTRRs is extended to handle CD and WC+ PAT types as shown in Table 15-20.

### Table 15-20. Combining PAT and MTRR Types

<table>
<thead>
<tr>
<th>Effective PAT Type</th>
<th>MTRR Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC</td>
<td>UC WC WP WT WB</td>
</tr>
<tr>
<td>UC–</td>
<td>UC WC CD CD CD</td>
</tr>
<tr>
<td>WC</td>
<td>WC WC WC WC WC</td>
</tr>
<tr>
<td>WC+</td>
<td>WC WC+ WC+ WC+</td>
</tr>
<tr>
<td>WP</td>
<td>UC CD WP CD WP</td>
</tr>
<tr>
<td>WT</td>
<td>UC CD CD WT WT</td>
</tr>
<tr>
<td>WB</td>
<td>UC WC WP WT WB</td>
</tr>
</tbody>
</table>

15.25.9 Page Splintering

When an address is mapped by guest and nested page table entries with different page sizes, the TLB entry that is created matches the size of the smaller page.
15.25.10 Legacy PAE Mode

The behavior of PAE mode in a nested-paging guest differs slightly from the behavior of (host-only) legacy PAE mode, in that the guest’s four PDPEs are not loaded into the processor at the time CR3 is written. Instead, the PDPEs are accessed on demand as part of a table walk. This has the side-effect that illegal bit combinations in the PDPEs are not signaled at the time that CR3 is written, but instead when the faulty PDPE is accessed as part of a table walk.

This means that an operating system cannot rely on the behavior when the in-memory PDPEs are different than the in-processor copy.

15.25.11 A20 Masking

There is no provision for applying A20 masking to guest physical addresses; the VMM can emulate A20 masking by changing the nested page mappings accordingly.

15.25.12 Detecting Nested Paging Support

Nested Paging is an optional feature of SVM and is not available in all implementations of SVM-capable processors. The CPUID instruction should be used to detect nested paging support on a particular processor. See Section 3.3, “Processor Feature Identification,” on page 64 for more information on using the CPUID instruction.

15.25.13 Guest Mode Execute Trap Extension

The Guest Mode Execute Trap (GMET) extension allows a hypervisor to cause nested page faults on attempts by a guest to execute code at CPL0, 1 or 2 from pages designated by the hypervisor. The presence of the GMET extension is indicated by CPUID Fn8000_000A EDX[17]=1. The GMET mode is selected for a targeted guest by setting bit 3 of VMCB offset 090h to 1. For processors that don’t support GMET this bit is ignored.

On GMET capable processors, when this bit is set to 1 on a VMRUN, the processor changes how the U/S bit in the nested page table is interpreted. The NX bit still prohibits execution of code at any privilege level when set to 1. However, with GMET enabled and the effective NX bit =0, if the effective U/S bit =1 and the page is being accessed for execution at CPL0, 1 or 2, a nested page fault #VMEXIT(NPF) is generated. If the effective NX bit =0 and the effective U/S bit =0 then the translation is allowed for the code page. The following table summarizes the behavior when GMET is enabled.

<table>
<thead>
<tr>
<th>nPT NX Bit</th>
<th>nPT U/S Bit</th>
<th>Guest User-Mode Code</th>
<th>Guest Supervisor-Mode Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>No Execute</td>
<td>No Execute</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Execute</td>
<td>No Execute</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Execute</td>
<td>Execute</td>
</tr>
</tbody>
</table>
The EXITINFO1 field for the nested page fault contains the page fault error code describing attributes of the attempted translation that caused the fault. A GMET violation is not explicitly indicated with a separate bit. It is up to software to determine if it was NX based or GMET based by inspecting this error code along with the faulting page’s effective NX and U/S settings.\(^1\)

15.26 Security

SVM provides additional hardware support that is designed to facilitate the construction of trusted software systems. While the security features described in this section are orthogonal to SVM’s virtualization support (and are not required for processor virtualization), the two form building blocks for trusted systems.

SKINIT Instruction. The SKINIT instruction and associated system support (the Trusted Platform Module or TPM) are designed to allow for verifiable startup of trusted software (such as a VMM), based on secure hash comparison.

Security Exception. A security exception (#SX) is used to signal certain security-critical events.

15.27 Secure Startup with SKINIT

The SKINIT instruction is one of the keys to creating a “root of trust” starting with an initially untrusted operating mode. SKINIT reinitializes the processor to establish a secure execution environment for a software component called the secure loader (SL) and starts execution of the SL in a way that cannot be tampered with. SKINIT also copies the secure loader executable image to an external device, such as a Trusted Platform Module (TPM) for verification using unique bus transactions that preclude SKINIT operation from being emulated by software in a way that the TPM could not readily detect. (Detailed operation is described in Section 15.27.4.)

15.27.1 Secure Loader

A secure loader (SL) typically initializes SVM hardware mechanisms and related data structures, and initiates execution of a trusted piece of software such as a VMM (referred to as a Security Kernel, or SK, in this document), after first having validated the identity of that software.

SKINIT allows SVM protections to be reliably enabled after the system is already up and running in a non-trusted mode — there is no requirement to change the typical x86 platform boot process.

Exact details of the handoff from the SL to an SK are dependent on characteristics of the SL, SK and the initial untrusted operating environment. However, there are specific requirements for the SL image, as described in Section 15.27.2.

\(^1\) The guest user/supervisor indication is normally provided in ExitInfo1, however on some implementations a GMET erratum may require CPL to be read from the guest VMCB.
15.27.2 Secure Loader Image

The secure loader (SL) image contains all code and initialized data sections of a secure loader. This code and initial data are used to initialize and start a security kernel in a completely safe manner, including setting up DEV protection for memory allocated for use by SL and SK. The SL image is loaded into a region of memory called the secure loader block (SLB) and can be no larger than 64Kbyte (see Section 15.27.3). The SL image is defined to start at byte offset 0 in the SLB.

The first word (16 bits) of the SL image must specify the SL entry point as an unsigned offset into the SL image. The second word must contain the length of the image in bytes; the maximum length allowed is 65535 bytes. These two values are used by the SKINIT instruction. The layout of the rest of the image is determined by software conventions. The image typically includes a digital signature for validation purposes. The digital signature hash must include the entry point and length fields. SKINIT transfers the SL image to the TPM for validation prior to starting SL execution (see Section 15.27.6 for further details of this transfer). The SL image for which the hash is computed must be ready to execute without prior manipulation.

15.27.3 Secure Loader Block

The secure loader block is a 64Kbyte range of physical memory which may be located at any 64Kbyte-aligned address below 4Gbyte. The SL image must have been loaded into the SLB starting at offset 0 before executing SKINIT. The physical address of the SLB is provided as an input operand (in the EAX register) to SKINIT, which sets up special protection for the SLB against device accesses (i.e., the DEV need not be activated yet).

The SL must be written to execute initially in flat 32-bit protected mode with paging disabled. A base address can be derived from the value in EAX to access data areas within the SL image using base+displacement addressing, to make the SL code position-independent.

Memory between the end of the SL image and the end of the SLB may be used immediately upon entry by the SL as secure scratch space, such as for an initial stack, before DEV protections are set up for the rest of memory. The amount of space required for this will limit the maximum size of the SL image, and will depend on SL implementation. SKINIT sets the ESP register to the appropriate top-of-stack value (EAX + 10000h).

Figure 15-14 illustrates the layout of the SLB, showing where EAX and ESP point after SKINIT execution. Labels in italics indicate suggested uses; other labels reflect required items.
The trusted platform module, or TPM, is an essential part of full trusted system initialization. This device is attached to an LPC link off the system I/O hub. It recognizes special SKINIT transactions, receives the SL image sent by SKINIT and verifies the signature. Based on the outcome, the device decides whether or not to cooperate with the SL or subsequent SK. The TPM typically contains sealed storage containing cryptographic keys and other high-security information that may be specific to the platform.
15.27.5 System Interface, Memory Controller and I/O Hub Logic

SKINIT uses special support logic in the processor’s system interface unit, the internal controller and the I/O hub to which the TPM is attached. SKINIT uses special transactions that are unique to SKINIT, along with this support logic, designed to securely transmit the SL Image to the TPM for validation.

The use of this special protocol is intended to allow the TPM to detect true execution, as opposed to emulation, of a trusted Secure Loader, which in turn provides a means for verifying the subsequent loading and startup of a trusted Security Kernel.

15.27.6 SKINIT Operation

The SKINIT instruction is intended to be used primarily in normal mode prior to the VMM taking control.

SKINIT takes the physical base address of the SLB as its only input operand in EAX, and performs the following steps:

1. Reinitialize processor state in the same manner as for the INIT signal, then enter flat 32-bit protected mode with paging off. The CS selector is set to 8h and CS is read only. The SS selector is set to 10h and SS is read/write and expand-up. The CS and SS bases are cleared to 0 and limits are set to 4G. DS, ES, FS and GS are left as 16-bit real mode segments and the SL must reload these with protected mode selectors having appropriate GDT entries before using them. Initialized data in the SLB may be referenced using the SS segment override prefix until DS is reloaded. The general purpose registers are cleared except for EAX, which points to the start of the secure loader, EDX, which contains model, family and stepping information, and ESP, which contains the initial stack pointer for the secure loader. Cache contents remain intact, as do the x87 and SSE control registers. Most MSRs also retain their values, except those which might compromise SVM protections. The EFER MSR, however, is cleared. The DPD, R_INIT and DIS_A20M flags in the VM_CR register are unconditionally set to 1.

2. Form the SLB base address by clearing bits 15:0 of EAX (EAX is updated), and enable the SL_DEV protection mechanism (see Section 15.24.8) to protect the 64-Kbyte region of physical memory starting at the SLB base address from any device access.

3. In multiprocessor operation, perform an interprocessor handshake as described in section 15.27.8.

4. Read the SL image from memory and transmit it to the TPM in a manner that cannot be emulated by software.

5. Signal the TPM to complete the hash and verify the signature. If any failures have occurred along the way, the TPM will conclude that no valid SL was started.

6. Clear the Global Interrupt Flag. This disables all interrupts, including NMI, SMI and INIT and ensures that the subsequent code can execute atomically. If the processor enters the shutdown state (due to a triple fault for instance) while GIF is clear, it can only be restarted by means of a RESET.

7. Update the ESP register to point to the first byte beyond the end of the SLB (SLB base + 65536), so that the first item pushed onto the stack by the SL will be at the top of the SLB.
8. Add the unsigned 16-bit entry point offset value from the SLB to the SLB base address to form
the SL entry point address, and jump to it.

The validation of the SL image by the TPM is a one-way transaction as far as SKINIT is concerned. It
does not depend on any response from the TPM after transferring the SL image before jumping to the
SL entry point, and initiates execution of the Secure Loader unconditionally. Because of the processor
initialization performed, SKINIT does not honor instruction or data breakpoint traps, or trace traps due
to EFLAGS.TF.

Pending interrupts. Device interrupts that may be pending prior to SKINIT execution due to
EFLAGS.IF being clear, or that assert during the execution of SKINIT, will be held pending until
software subsequently sets GIF to 1. Similarly, SMI, INIT and NMI interrupts that assert after the start
of SKINIT execution will also be held pending until GIF is set to 1.

Debug Considerations. SKINIT automatically disables various implementation-specific hardware
debug features. A debug version of the SL can reenable those features by clearing the VM_CR.DPD
flag immediately upon entry.

15.27.7 SL Abort

If the SL determines that it cannot properly initialize a valid SK, it must cause GIF to be set to 1 and
clear the VM_CR MSR to re-enable normal processor operation.

15.27.8 Secure Multiprocessor Initialization

The following standard APIC features are used for secure MP initialization:

- The concept of a single Bootstrap Processor (BSP) and multiple Application Processors (APs).
- The INIT interprocessor interrupt (IPI), which puts the target processors into a halted state (INIT
  state) which is responsive only to a subsequent Startup IPI.
- The Startup IPI causes target processors to begin execution at a location in memory that is
  specified by the Boot Processor and conveyed along with the Startup IPI. The operation of the
  processor in response to a Startup IPI is slightly modified to support secure initialization, as
  described below.

A Startup IPI normally causes an AP to start execution at a location provided by the IPI. To support
secure MP startup, each AP responds to a startup IPI by additionally clearing its GIF and setting the
DPD, R_INIT and DIS_A20M flags in the VM_CR register if, and only if, the BSP has indicated that
it has executed an SKINIT. All other aspects of Startup IPI behavior remain unchanged.

Software Requirements for Secure MP initialization. The driver that starts the SL must execute on
the BSP. Prior to executing the SKINIT instruction, the driver must save any processor-specific system
register contents to memory for restoration after reinitialization of the APs. The driver should also put
all APs in an idle state. The driver must first confirmed that all APs are idle and then it must issue an
INIT IPI to all APs and wait for its local APIC busy indication to clear. This places the APs into a
halted state which is responsive only to a subsequent Startup IPI. APs will still respond to snoops for
cache coherency. The driver may execute SKINIT at any time after this point. Depending on processor
implementation, a fixed delay of no more than 1000 processor cycles may be necessary before executing SKINIT to ensure reliable sensing of APIC INIT state by the SKINIT.

**AP Startup Sequence.** While the SL starts executing on the BSP, the APs remain halted in APIC INIT state. Either the SL or the SK may issue the Startup IPI for the APs at whatever point is deemed appropriate. The Startup IPI conveys an 8-bit vector specified by the software that issues the IPI to the APs. This vector provides the upper 8 bits of a 20-bit physical address. Therefore, the AP startup code must reside in the lower 1Mbyte of physical memory—with the entry point at offset 0 on that particular page.

In response to the Startup IPI, the APs start executing at the specified location in 16-bit real mode. This AP startup code must set up protections on each processor as determined by the SL or SK. It must also set GIF to re-enable interrupts, and restore the pre-SKINIT system context (as directed by the SL or SK executing on the BSP), before resuming normal system operation.

The SL must guarantee the integrity of the AP startup sequence, for example by including the startup code in the hashed SL image and setting up DEV protection for it before copying it to the desired area. The AP startup code does not need to (and should not) execute SKINIT. Care must also be taken to avoid issuing another INIT IPI from any processor after the BSP executes SKINIT and before all APs have received a Startup IPI, as this could compromise the integrity of AP initialization.

**Pending interrupts.** Device interrupts that may be pending on an AP prior to the APIC INIT IPI due to EFLAGS.IF being clear, or that assert any time after the processor has accepted the INIT IPI, will be held pending through the subsequent Startup IPI, and remain pending until software sets GIF to 1 on that AP. Similarly, SMI, INIT, and NMI interrupts that assert after the processor has accepted the INIT IPI will also be held pending until GIF is set to 1.

**Aborting MP initialization.** In the event that the SL or SK on the BSP decides to abort SVM system initialization for any reason, the following clean-up actions must be performed by SL code executing on each processor before returning control to the original operating environment:

- The BSP and all APs that responded to the Startup IPI must restore GIF and clear VM_CR on each processor for normal operation.
- For each processor that has a distinct memory controller associated with it, the SL_DEV_EN flag in the DEV control register must be cleared in order to restore normal device accessibility to the 64KB SL memory range.

Any secure context created by the SL that should not be exposed to untrusted code should be cleaned up as appropriate before these steps are taken.

### 15.28 Security Exception (#SX)

The Security Exception fault signals security-sensitive events that occur while executing the VMM, in the form of an exception so that the VMM may take appropriate action. (A VMM would typically intercept comparable sensitive events in the guest.) In the current implementation, the only use of the #SX is to redirect external INITs into an exception so that the VMM may — among other possibilities
— destroy sensitive information before re-issuing the INIT, this time without redirection. The INIT redirection is controlled by the VM.CR.R_INIT bit.

The #SX exception dispatches to vector 30, and behaves like other fault-class exceptions such as General Protection Fault (#GP). The #SX exception pushes an error code. The only error code currently defined is 1, and indicates redirection of INIT has occurred.

The #SX exception is a contributory fault.

## 15.29 Advanced Virtual Interrupt Controller

The AMD Advanced Virtual Interrupt Controller (AVIC) is an important enhancement to AMD Virtualization™ Technology (AMD-V). In a virtualized environment, AVIC presents to each guest a virtual interrupt controller that is compliant with the local Advanced Programmable Interrupt Controller (APIC) architecture. See Chapter 16, “Advanced Programmable Interrupt Controller (APIC),” on page 567 for a detailed description of APIC.

### 15.29.1 Introduction

In a virtualized computer system, each guest operating system needs access to an interrupt controller to send and receive device and interprocessor interrupts. When there is no hardware acceleration, it falls to the virtual machine monitor (VMM) to intercept guest-initiated attempts to access the interrupt controller registers and provide direct emulation of the controller system programming interface allowing the guest to initiate and process interrupts. The VMM uses the underlying physical and virtual interrupt delivery mechanisms of the system to deliver interrupts from I/O devices and virtual processors to the target guest virtual processor and to handle any required end of interrupt processing.

Given the high rate of device and interprocessor interrupt generation in modern computer systems, the emulation of a local APIC is a significant burden for the VMM.

AVIC architecture addresses the overhead of guest interrupt processing in a virtualized environment by applying hardware acceleration to the following components of interrupt processing:

- Providing a guest operating system access to performance-critical interrupt controller registers
- Initiating intra- and inter-processor interrupts (IPIs) in and between virtual processors in a guest

Acceleration of the delivery of virtual interrupts from I/O devices to virtual processors is not addressed directly by AVIC hardware. This acceleration would be provided by an I/O memory management unit (IOMMU). The AVIC architecture is compatible with the AMD I/O Memory Management Unit (IOMMU). For more information on the IOMMU architecture, see *AMD I/O Virtualization Technology (IOMMU) Specification* (order #48882).

### 15.29.1.1 Local APIC Register Access

The system programming interface for the local APIC comprises a set of memory-mapped registers. In a non-virtualized environment, system software directly reads and writes these registers to configure the interrupt controller and initiate and process interrupts. In a virtualized environment, each guest
operating system still requires access to this system programming interface but does not own the underlying interrupt processing hardware. To provide this facility to the guest operating system, VMM-level software emulates the local APIC for each guest virtual processor.

The AVIC architecture provides an image of the local APIC called the guest virtual APIC (guest vAPIC) in the guest physical address (GPA) space of each virtual processor when the virtual machine for the guest is instantiated. This image is backed by a page in the system physical address (SPA) space called a vAPIC backing page. The backing page remains pinned in system memory as long as the virtual machine persists, even when the specific virtual processor associated with the backing page is not running. Accesses to the memory-mapped register set by the guest are redirected by AVIC hardware to this backing page.

The VMM reads configuration, control, and command information written by the guest from the backing page and writes status information to this page for the guest to read. The guest is allowed to read most registers directly without the need for VMM intervention. Most writes are intercepted allowing the VMM to process and act on the configuration, control, and command data from the guest. However, for certain frequently used command and control operations, specific hardware support allows the guest to directly initiate interrupts and complete end of interrupt processing eliminating the need for VMM intervention in the execution of performance-critical operations.

**Software-initiated Interrupts.** Modern operating systems use software interrupts (self-IPIs) to implement software event signalling, inter-process communication and the scheduling of deferred processing. System software sets up and initiates these interrupts by writing to control registers of the local APIC. AVIC hardware reduces VMM overhead by providing hardware assist for many of these operations.

**Inter-processor Interrupts.** Inter-processor interrupts (IPIs) are used extensively by modern operating systems to handle communication between processor cores within a machine (or, in a virtualized environment, between virtual processors within a virtual machine). IPIs are also employed to provide signaling and synchronization for operations such as cross-processor TLB invalidations (also known as TLB shootdowns). AVIC provides hardware mechanisms that deliver the interrupt to the virtual interrupt controller of the target virtual processor without VMM intervention.

### 15.29.2 Architectural Definition

The following sections describe the AVIC architecture. Specific implementations of AVIC may deviate from this description as long as the observed behavior of the hardware complies with this description.

#### 15.29.2.1 Virtualizing the Local APIC

The guest virtual processor accesses the facilities of its local APIC by reading and writing a set of registers located in a 4-Kbyte page in its guest physical address space. AVIC hardware virtualizes this access by redirecting attempted accesses by the guest to a vAPIC backing page located in system physical address (SPA) space.
AVIC hardware detects attempted accesses by the guest to its local APIC register set and redirects these accesses to the vAPIC backing page. This is illustrated in the figure below.

**Figure 15-15. vAPIC Backing Page Access**

To correctly redirect guest accesses of the guest vAPIC registers to the vAPIC backing page, the hardware needs two addresses. These are:

- vAPIC backing page address in the SPA space
- Guest vAPIC base address (APIC BAR) in the GPA space

System software is responsible for setting up a translation in the nested page table granting guest read and write permissions for accesses to the vAPIC Backing Page in SPA space. AVIC hardware walks the nested page table to check permissions, but does not use the SPA address specified in the leaf page table entry. Instead, AVIC hardware finds this address in the AVIC_BACKING_PAGE pointer field of the VMCB.
The VMM initializes the backing page with appropriate default APIC register values including items such as APIC version number. The vAPIC backing page address and the guest vAPIC base address are stored in the VMCB fields AVIC_BACKING_PAGE pointer and V_APIC_BAR respectively.

System firmware initializes the value of guest vAPIC base address (and VMCB.V_APIC_BAR) to FEE0_0000h. This is the address where the guest operating system expects to find the local APIC register set when it boots. If the guest attempts to relocate the local APIC register base address in GPA space by writing to the APIC Base Address Register (MSR 0000_001Bh), the VMM should intercept the write to update the V_APIC_BAR field of the guest’s VMCB(s) and the GPA part of translation in the virtual machine’s nested page tables.

The vAPIC backing page must be present in system physical memory for the life of the guest VM because some fields are updated even when the guest is not running.

**Virtual APIC Register Accesses.** AVIC hardware detects attempted guest accesses to the vAPIC registers in the backing page. These attempted accesses are handled by the register-level permissions filter in one of three ways:

- **Allow**—The access to the backing page is allowed to complete. Writes update the backing page value, while reads return the current value. In certain cases, a write results in specific hardware-based acceleration actions (summarized in Table 15-22 and described below).
- **Fault**—The processor performs an SVM intercept before the access. Causes a #VMEXIT.
- **Trap**—The processor performs an SVM intercept immediately after the access completes. Causes a #VMEXIT.

The details of this behavior for each of these registers are summarized in the following table:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register Name</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>20h</td>
<td>APIC ID Register</td>
<td>Read: Allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write: #VMEXIT (trap)</td>
</tr>
<tr>
<td>30h</td>
<td>APIC Version Register</td>
<td>Read: Allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write: #VMEXIT (fault)</td>
</tr>
<tr>
<td>80h</td>
<td>Task Priority Register (TPR)</td>
<td>Read: Allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write: Accelerated by AVIC</td>
</tr>
<tr>
<td>90h</td>
<td>Arbitration Priority Register  (APR)</td>
<td>Read: #VMEXIT (fault)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write: #VMEXIT (fault)</td>
</tr>
<tr>
<td>A0h</td>
<td>Processor Priority Register (PPR)</td>
<td>Read: Allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write: #VMEXIT (fault)</td>
</tr>
<tr>
<td>B0h</td>
<td>End of Interrupt Register (EOI)</td>
<td>Read: Allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write: Accelerated by AVIC for edge-triggered interrupts or #VMEXIT (trap) for level triggered interrupts</td>
</tr>
<tr>
<td>C0h</td>
<td>Remote Read Register</td>
<td>Read: Allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write: #VMEXIT (trap)</td>
</tr>
</tbody>
</table>
Table 15-22. Guest vAPIC Register Access Behavior (continued)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register Name</th>
<th>Read: Allowed</th>
<th>Write: #VMEXIT (trap) or #VMEXIT (fault)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0h</td>
<td>Logical Destination Register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E0h</td>
<td>Destination Format Register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0h</td>
<td>Spurious Interrupt Vector Register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100h–170h</td>
<td>In-Service Register (ISR)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>180h–1F0h</td>
<td>Trigger Mode Register (TMR)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200h–270h</td>
<td>Interrupt Request Register (IRR)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>280h</td>
<td>Error Status Register (ESR)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>300h</td>
<td>Interrupt Command Register Low (bits 31:0)</td>
<td></td>
<td>Accelerated by AVIC or #VMEXIT (trap) for advanced functions.</td>
</tr>
<tr>
<td>310h</td>
<td>Interrupt Command Register High (bits 63:32)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>320h</td>
<td>Timer Local Vector Table Entry</td>
<td></td>
<td>#VMEXIT (trap)</td>
</tr>
<tr>
<td>330h</td>
<td>Thermal Local Vector Table Entry</td>
<td></td>
<td>#VMEXIT (trap)</td>
</tr>
<tr>
<td>340h</td>
<td>Performance Counter Local Vector Table Entry</td>
<td></td>
<td>#VMEXIT (trap)</td>
</tr>
<tr>
<td>350h</td>
<td>Local Interrupt 0 Vector Table Entry</td>
<td></td>
<td>#VMEXIT (trap)</td>
</tr>
<tr>
<td>360h</td>
<td>Local Interrupt 1 Vector Table Entry</td>
<td></td>
<td>#VMEXIT (trap)</td>
</tr>
<tr>
<td>370h</td>
<td>Error Vector Table Entry</td>
<td></td>
<td>#VMEXIT (trap)</td>
</tr>
<tr>
<td>380h</td>
<td>Timer Initial Count Register</td>
<td></td>
<td>#VMEXIT (trap)</td>
</tr>
<tr>
<td>390h</td>
<td>Timer Current Count Register</td>
<td></td>
<td>#VMEXIT (fault)</td>
</tr>
<tr>
<td>3E0h</td>
<td>Timer Divide Configuration Register</td>
<td></td>
<td>#VMEXIT (trap)</td>
</tr>
<tr>
<td>400h–FFFh</td>
<td>Extended Registers</td>
<td></td>
<td>#VMEXIT (fault)</td>
</tr>
</tbody>
</table>
Accesses to any other register locations not explicitly defined in this table are allowed to read and write the backing page.

All vAPIC registers are 32-bits wide and are located at 16-byte aligned offsets. The results of an attempted read or write of any bytes in the range [register_offset + 4:register_offset + 15] are undefined.

Guest writes to the Task Priority Register (TPR) and specific usage cases of writes to the End of Interrupt (EOI) Register and the Interrupt Command Register Low (ICRL) cause specific hardware actions. AVIC hardware allows guest writes to the Interrupt Command Register High (ICRH) since the writing of this register has no immediate hardware side-effect. AVIC hardware maintains and uses the value in the Processor Priority Register (PPR) to control the delivery of interrupts to guest virtual processors. The following sections discuss the handling of accesses by the guest to these registers in the vAPIC backing page.

**Task Priority Register (TPR).** When the guest operating system writes to the TPR, the value is updated in the backing page and the upper 4 bits of the value are automatically copied by the hardware to the V_TPR value in the VMCB. All reads from the TPR location return the value from the vAPIC backing page. Also, any TPR accesses using the MOV CR8 semantics update the backing page and V_TPR values.

The priority value stored in CR8 and V_TPR are not the same format as the APIC TPR register. Only the Task Priority bits of are maintained in the lower 4 bits of CR8 and V_TPR. The Task Priority Subclass value is not stored. Writes to the memory-mapped TPR register update bits 3:0 of CR8 and V_TPR and writes to CR8 update the TPR backing page value bits 7:4 while bits 3:0 are set to zero.

![Figure 15-16. Virtual APIC Task Priority Register Synchronization](v2_TPR_figure.eps)

The synchronization between the Task Priority field of the TPR and the Task Priority field of CR8 is normal local APIC behavior which is emulated by AVIC. For more information on the APIC, see Chapter 16, “Advanced Programmable Interrupt Controller (APIC),” on page 567.

**Processor Priority Register (PPR).** Writes to the processor priority register by the guest cause a #VMEXIT without updating the value in the backing page. AVIC hardware maintains the PPR value in
the backing page. AVIC hardware updates the PPR value in the backing page when either the TPR value or the highest in-service interrupt changes. This value is used to control the delivery of virtual interrupts to the guest. PPR reads by the guest are allowed.

**End of Interrupt (EOI) Register.** When the guest writes to the EOI register address, AVIC hardware clears the highest priority in-service interrupt (ISR) bit in the backing page and re-evaluates the interrupt state to determine if another pending interrupt should be delivered. If the highest priority in-service interrupt is set to level mode (in the corresponding TMR bit), the EOI write causes a #VMEXIT to allow the VMM to emulate the level-triggered behavior.

**Interrupt Control Register Low (ICRL).** Writes to the ICRL register have the side-effect of initiating the generation of an interprocessor interrupt (IPI) based on the values written to the fields in both the ICRL and ICRH registers. AVIC hardware handles the generation of IPIs when the specified Message Type is Fixed (also known as fixed delivery mode) and the Trigger Mode is edge-triggered. The hardware also supports self and broadcast delivery modes specified via the Destination Shorthand (DSH) field of the ICRL. Logical and physical APIC ID formats are supported. All other IPI types cause a #VMEXIT. For more information on AVIC’s handling of IPI commands, see “Inter-processor Interrupts” on page 515.

### 15.29.2.2 VMCB Changes in Support of AVIC

The following paragraphs provide an overview of new VMCB fields defined as part of the AVIC architecture.

**VMCB Control Word.** AVIC adds the AVIC Enable bit to the VMCB control word at offset 60h:
AVIC Enable—Virtual Interrupt Control, Bit 31. The AVIC hardware support may be enabled on a per virtual processor basis. This bit determines whether or not AVIC is enabled for a particular virtual processor. Any guest configured to use AVIC must also enable RVI (nested paging). Enabling AVIC implicitly disables the V_IRQ, V_INTR_PRIO, V_IGN_TPR, and V_INTR_VECTOR fields in the VMCB Control Word.

Newly Defined VMCB Fields. AVIC utilizes a number of formerly reserved locations in the VMCB. Table 15-24 below lists the new fields defined by the architecture:

Table 15-24. New VMCB Fields Defined by AVIC

<table>
<thead>
<tr>
<th>VMCB Offset</th>
<th>Bit(s)</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>098h</td>
<td>63:52</td>
<td>Reserved, SBZ</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>51:12</td>
<td>V_APIC_BAR</td>
<td>Bits 51:12 of the GPA of the guest vAPIC register bank</td>
</tr>
<tr>
<td></td>
<td>11:0</td>
<td>Reserved, SBZ</td>
<td>—</td>
</tr>
<tr>
<td>0E0h</td>
<td>63:52</td>
<td>Reserved, SBZ</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>51:12</td>
<td>AVIC_BACKING_PAGE Pointer</td>
<td>Bits 51:12 of HPA of the vAPIC backing page</td>
</tr>
<tr>
<td></td>
<td>11:0</td>
<td>Reserved, SBZ</td>
<td>—</td>
</tr>
</tbody>
</table>
These fields are discussed further in the following paragraphs:

**V_APIC_BAR**—VMCB, Offset 098h. This entry is used to hold a copy of guest physical base address of its local APIC register block. The guest can change the GPA of its local APIC register block by writing to the guest version of the APIC Base Address Register (MSR 0000_001Bh). Writes to this MSR are intercepted by the VMM and the value is used to update the GPA in the nested page table entry for the vAPIC backing page and the value to be saved in this field of the VMCB.

**APIC_BACKING_Page Pointer**—VMCB, Offset 0E0h. This is a 52-bit HPA pointer to the vAPIC backing page for this virtual processor. The vAPIC backing page is described in more detail in the following section.

**Logical APIC Table Pointer**—VMCB, Offset 0F0h. This is a 52-bit HPA pointer to the Logical APIC ID Table for the virtual machine containing this virtual processor. This table is described in more detail in the following section.

**Physical APIC Table Pointer**—VMCB, Offset 0F8h. This is a 52-bit HPA pointer to the Physical APIC ID Table for the virtual machine containing this virtual processor. This table is described in more detail in the following section.

**AVIC_PHYSICAL_MAX_INDEX**—VMCB, Offset 0F8h. Bits [7:0]. This 8-bit value provides the index of the last guest physical core ID for this guest.

**Restrictions on Physical Address Pointers.** All of the physical addresses in the previous sections must point to legal, implementation-supported physical address ranges. These pointers are evaluated on VMRUN and cause a #VMEXIT if they are outside of the legal range. These memory ranges must be mapped as write-back cacheable memory type.

All the addresses point to 4-Kbyte aligned data structures. Bits 11:0 are reserved (except for offset 0F8h) and should be set to zero. The lower 8 bits of offset 0F8h are used for the field AVIC_PHYSICAL_MAX_INDEX.
Multiprocessor VM requirements. When running a VM which has multiple virtual CPUs, and the VMM runs a virtual CPU on a core which had last run a different virtual CPU from the same VM, regardless of the respective ASID values, care must be taken to flush the TLB on the VMRUN using a TLB_CONTROL value of 3h. Failure to do so may result in stale mappings misdirecting virtual APIC accesses to the previous virtual CPU's APIC backing page.

15.29.2.3 AVIC Memory Data Structures
The AVIC architecture defines three new memory-resident data structures. Each of these structures is defined to fit exactly in one 4-Kbyte page. Future implementations may expand the size.

Virtual APIC Backing Page. Each virtual processor in the system is assigned a virtual APIC backing page (vAPIC backing page). Accesses by the guest to the local APIC register block in the guest physical address space are redirected to the vAPIC backing page in system memory. The vAPIC backing page is used by AVIC hardware and the VMM to emulate the local APIC. See “Virtual APIC Register Accesses” on page 517 for a detailed description.

Physical APIC ID Table. The physical APIC ID table is set up and maintained by the VMM and is used by the hardware to locate the proper vAPIC backing page to be used to deliver interrupts based on the guest physical APIC ID. One physical APIC ID table must be provided per virtual machine.

The guest physical APIC ID is used as an index into this table. Each entry contains a pointer to the virtual processor’s vAPIC backing page, a bit to indicate whether the virtual processor is currently scheduled on a physical core, and if so, the physical APIC ID of that core.

The length of this table is fixed at 4 Kbytes allowing a maximum of 512 virtual processors per virtual machine. However, in this version of the architecture the maximum number of virtual processors per guest is limited to 256. The physical ID table can be populated in a sparse manner using the valid bit to indicate assigned IDs. The index of the last valid entry is stored in the VMCB AVIC_PHYSICAL_MAX_INDEX field.

A pointer to this table is maintained in the VMCB. Because there is a single Physical APIC ID Table per virtual machine, the value of this pointer is the same for every virtual processor within the virtual machine.

Each entry in the table has the following format:
Figure 15-17. Physical APIC ID Table Entry

Table 15-25. Physical APIC ID Table Entry Fields

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>V</td>
<td>Valid bit. When set, indicates that this entry contains a valid vAPIC backing page pointer. If cleared, this table entry contains no information.</td>
</tr>
<tr>
<td>62</td>
<td>IR</td>
<td>IsRunning. This bit indicates that the corresponding guest virtual processor is currently scheduled by the VMM to run on a physical core.</td>
</tr>
<tr>
<td>61:52</td>
<td>—</td>
<td>Reserved, SBZ. Should always be set to zero.</td>
</tr>
<tr>
<td>51:12</td>
<td>Backing Page Pointer</td>
<td>4-Kbyte aligned HPA of the vAPIC backing page for this virtual processor.</td>
</tr>
<tr>
<td>11:8</td>
<td>—</td>
<td>Reserved, SBZ. Should always be set to zero.</td>
</tr>
<tr>
<td>7:0</td>
<td>Host Physical APIC ID</td>
<td>Physical APIC ID of the physical core allocated by the VMM to host the guest virtual processor. This field is not valid unless the IsRunning bit is set.</td>
</tr>
</tbody>
</table>

Note that the IR bit, when set, indicates that the VMM has assigned a physical core to host this virtual processor. The bit does not differentiate between a physical processor running in guest mode (actively executing guest software) or in hypervisor mode (having suspended the execution of guest software).
The Physical APIC ID Table occupies the lower half of a single 4-Kbyte memory page, formatted as follows:

Since a destination of FFh is used to specify a broadcast, physical APIC ID FFh is reserved. The upper 2048 bytes of the table are reserved and should be set to zero.

**Logical APIC ID Table.** In addition to the Physical APIC ID Table, each guest VM is assigned a Logical APIC ID Table. This table is used to lookup the guest physical APIC ID for logically addressed interrupt requests. Each entry of this table provides the guest physical APIC ID corresponding to a single logically addressed APIC. Note that this implies that the logical ID of each vAPIC must be unique. The entries of this table are selected using the logical ID and interpreted differently depending upon logical APIC addressing mode of the guest. logical destination modes are supported: flat clustered.

If the guest attempts to change the logical ID of its APIC, the VMM must reflect this change in the Logical APIC ID Table. AVIC hardware supports the fixed interrupt message type targeting one or more logical destinations. The hardware also supports self and broadcast delivery modes specified via the Destination Shorthand (DSH) field of the ICRL. Any other message types must be supported through emulation by the VMM.

**Figure 15-18. Physical APIC Table in Memory.**
A pointer to this table is maintained in the VMCB. Because there is a single Logical APIC ID Table per virtual machine, the value of this pointer is the same for every virtual processor within the virtual machine.

For all logical destination modes, the table entries have the following format:

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>V</td>
<td>Valid Bit. When set, indicates that this table entry contains a valid physical APIC ID. If cleared, this table entry contains no information.</td>
</tr>
<tr>
<td>30:8</td>
<td>—</td>
<td>Reserved, SBZ. Should always be set to zero.</td>
</tr>
<tr>
<td>7:0</td>
<td>Guest Physical APIC ID</td>
<td>Guest physical APIC ID corresponding to the local APIC selected when logically addressed.</td>
</tr>
</tbody>
</table>

**Figure 15-19. Logical APIC ID Table Entry**

**Logical APIC ID Table Format for Flat Mode.** When running in flat mode, AVIC expects the logical APIC ID table to be formatted as shown in Figure 15-20 below. This mode uses only the first 8 entries of the table. Although the logical APIC ID is an eight bit value, supported encodings must be of the form $2^i$, where $i = 0$ to 7. In the figure the value $i$ is used and represents the index into the table. The actual byte offset into the table for a given logical APIC ID $l_{\text{apic_id}}$ is $4 \times \log_2(l_{\text{apic_id}})$. 

**Table 15-26. Logical APIC ID Table Entry Fields**
Logical APIC ID Table Format for Cluster Mode. In cluster mode, bits [7:4] of the logical APIC ID represent the cluster number and bits [3:0] represent the APIC index (bit encoded). The cluster number Fh (15) is reserved. Since the APIC index field is four bits, four encodings are supported for the APIC index value.

The actual byte offset into the table for a given cluster $c$ and an APIC index $apic\_ix$ is $(16 \times c) + 4 \times \log_2(apic\_ix)$

When running in cluster mode, AVIC expects the logical APIC ID table to be formatted as shown in Figure 15-21 below.
15.29.2.4 Interrupt Delivery

There are two fundamental types of virtual interrupts—interprocessor interrupts (IPIs) and I/O device interrupts (device interrupts). An IPI is initiated when guest system software writes the ICRL register. A device interrupt is initiated by a I/O device that has been programmed by guest system software (usually a device driver) to send a message signaling an event to a specific guest physical processor. This message usually includes an interrupt vector number indicating the nature of the event.

The following sections discuss the actions taken by AVIC hardware when a virtual processor signals an IPI and the actions taken by I/O virtualization hardware when a device signals a virtual interrupt.

Interprocessor Interrupts. To process an IPI, AVIC hardware executes the following steps:

1. If the destination-shorthand coded in the command is 01b (i.e. self), update the IRR in the backing page, signal doorbell to self and skip remaining steps.
2. If destination-shorthand is non-zero, or if the destination field is FFh (i.e. broadcast), jump to step 4.
3. If the destination(s) is (are) logically addressed, lookup the guest physical APIC IDs for each logical ID using the Logical APIC ID table.
   If the entry is not valid (V bit is cleared), cause a #VMEXIT.
   If the entry is valid, but contains an invalid backing page pointer, cause a #VMEXIT.

Figure 15-21. Logical APIC ID Table Format, Cluster Mode.
4. Lookup the vAPIC backing page address in the Physical APIC table using the guest physical APIC ID as an index into the table.
   For directed interrupts, if the selected table entry is not valid, cause a #VMEXIT. For broadcast IPIs, invalid entries are ignored.

5. For every valid destination:
   - Atomically set the appropriate IRR bit in each of the destinations’ vAPIC backing page.
   - Check the IsRunning status of each destination.
   - If the destination IsRunning bit is set, send a doorbell message using the host physical core number from the Physical APIC ID table.

6. If any destinations are identified as not currently scheduled on a physical core (i.e., the IsRunning bit for that virtual processor is not set), cause a #VMEXIT.

Refer to Section , “AVIC IPI Delivery Not Completed,” on page 531 for new exitcodes associated with the #VMEXIT exceptions listed above.

Device Interrupts. The delivery of a I/O device interrupt to a virtual processor is handled by an IOMMU with virtual interrupt capability. To deliver a virtual interrupt, I/O virtualization hardware executes the following steps:

1. An interrupt message arrives from the I/O device identifying the source device and interrupt vector number.

2. I/O virtualization hardware uses the device ID to determine the guest physical APIC ID of the core that is the target of the device interrupt.

3. I/O virtualization hardware uses the guest physical APIC ID to index into the Physical APIC ID Table to find the SPA of the vAPIC backing page. If the I/O virtualization hardware accesses an entry in the Physical APIC ID Table that is not valid (V bit is cleared), the I/O virtualization hardware aborts the virtual interrupt delivery and logs an error.

4. I/O virtualization hardware performs any required vector number translation.

5. I/O virtualization hardware atomically sets the bit in the IRR in the vAPIC backing page that corresponds to the vector.

6. If the virtual processor that is the target of the interrupt is not currently running on its assigned physical core, the virtual interrupt will be presented when the virtual processor is made active again. I/O virtualization hardware may provide additional information to the VMM about the device interrupt to aid in virtual processor scheduling decisions.
   If the virtual processor that is the target of the interrupt is scheduled on a physical processor (indicated by the IsRunning bit of the Physical APIC ID table entry being set), I/O virtualization hardware uses the host physical APIC ID in the table entry to send a doorbell signal to the corresponding processor core to signal that an interrupt needs to be processed.
15.29.2.5 CPUID Feature Bits

A CPUID feature bit is used indicate support for AVIC on a specific hardware implementation. CPUID Fn8000_000A_EDX[AVIC] is designated for this purpose and is returned in bit 13 of EDX. If EDX[13] is set, the AVIC architecture is supported on that hardware.

See Section 3.3, “Processor Feature Identification,” on page 64 for more information on using the CPUID instruction.

15.29.2.6 New Processor Mechanisms

In order to support the direct injection of interrupts into the guest and to accelerate critical vAPIC functions, new hardware mechanisms are implemented in the processor.

Special Trap/Fault Handling for vAPIC Accesses. To virtualize the local APIC utilized by the guest to generate and process interrupts, all read and write accesses by the guest virtual processor to its local APIC registers are redirected to the vAPIC backing page. Most reads and many writes to this guest physical address range read or write the contents of memory locations within the vAPIC backing page at the corresponding offset.

To support proper handling and emulation of the guest local APIC, the processor provides permissions filtering hardware (Refer to Figure 15-15.) that detects and intercepts accesses to specific offsets (representing APIC registers) within the vAPIC backing page. This hardware either allows the access, blocks the access and causes a #VMEXIT (fault behavior), or allows the access and then causes a #VMEXIT (trap behavior).

Hardware directly handles the side effects of guest writes to the TPR and EOI registers. Writes to the ICRL register with simple functional side effects such as the generation of a directed IPI or a self-IPI request are handled directly. Values written to the ICRL defined to initiate more complex behavior cause a #VMEXIT to allow the VMM to emulate the function. A guest write to the ICRH register has no immediate hardware side effect and is allowed.

Most other write access attempts within the vAPIC register bank address range cause a #VMEXIT with trap or fault behavior allowing the VMM to emulate the function of that register. See Table 15-22 for more detail.

Reads and writes to locations within the vAPIC backing page, but outside the offset range of defined vAPIC registers are allowed to complete.

Doorbell Mechanism. Each core provides a doorbell mechanism that is used by other cores (for IPIs) and the IOMMU (for device interrupts) to signal to the VMM of the target physical core that a virtual interrupt requires processing. The exact mechanism is implementation-specific, but must be protected from access from non-privileged software running on other cores and from direct access by an external device.

When the doorbell is received in guest mode, hardware on the receiving core evaluates the vAPIC state in the vAPIC backing page for the currently running virtual processor and injects the interrupt into the guest as appropriate.
Doorbell Register. The system programming interface to the doorbell mechanism is provided via an MSR. Sending a doorbell signal to another core is initiated by writing the physical APIC ID corresponding to that core to the Doorbell Register (MSR C001_011Bh). The format of this register is shown in Figure 15-22 below.

![Figure 15-22. Doorbell Register, MSR C001_011Bh](image)

Writing to this register causes a doorbell signal to be sent to the specified physical core. Any attempt to read from this register results in a #GP.

Processing of Doorbell Signals. A doorbell signal delivered to a running guest is recognized by the hardware regardless of whether it can be immediately injected into the guest as a virtual interrupt. On the next VMRUN, the virtual interrupt delivery mechanism evaluates the state of the IRR register of the guest’s vAPIC backing page to find the highest priority pending interrupt and injects it if interrupt masking and priority allow.

Additional VMRUN Handling. In addition to the normal VMRUN operations, the core re-evaluates the APIC state in the vAPIC backing page upon entry into the guest and processes pending interrupts as necessary. Specifically:

- On VMRUN the interrupt state is evaluated and the highest priority pending interrupt indicated in the IRR is delivered if interrupt masking and priority allow
- Any doorbell signals received during VMRUN processing are recognized immediately after entering the guest
- When AVIC mode is enabled for a virtual processor, the V_IRQ, V_INTR_PRIO, V_INTR_VECTOR, and V_IGN_TPR fields in the VMCB are ignored.

15.29.2.7 New Exit Codes for AVIC

The AVIC architecture defines two new AVIC-related #VMEXIT events. These cases are described in the following sections. Assigned EXITCODE values are given in Table C-1 on page 631.

AVIC IPI Delivery Not Completed. An IPI could not be delivered to all targeted guest virtual processors because at least one guest virtual processor was not allocated to a physical core at the time. This results in a #VMEXIT with an exit code of AVIC_INCOMPLETE_IPI. Additional data associated with this #VMEXIT event is returned in the EXITINFO1 and EXITINFO2 fields.

EXITINFO1. This field contains the values written to the vAPIC ICRH and ICRL registers.
EXITINFO1. This field contains information describing the specific reason for the IPI delivery failure.

**Table 15-27. EXITINFO1 Fields**

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>ICRH</td>
<td>Value written to the vAPIC ICRH register.</td>
</tr>
<tr>
<td>31:0</td>
<td>ICRL</td>
<td>Value written to the vAPIC ICRL register.</td>
</tr>
</tbody>
</table>

EXITINFO2. This field contains information describing the specific reason for the IPI delivery failure.

**Table 15-28. EXITINFO2 Fields**

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:32</td>
<td>ID</td>
<td>Specific reason for the delivery failure. See Table 15-29 for defined values.</td>
</tr>
<tr>
<td>31:8</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>7:0</td>
<td>Index</td>
<td>For ID = 1 – 3, this field provides the index of a logical or physical table entry. Reserved for all other ID values.</td>
</tr>
</tbody>
</table>

The ID field identifies the reason for the IPI delivery failure:
**Table 15-29. ID Field—IPI Delivery Failure Cause**

<table>
<thead>
<tr>
<th>ID</th>
<th>Cause</th>
<th>Description</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Invalid Interrupt type</td>
<td>The trigger mode for the specified IPI was set to level or the destination type is unsupported.</td>
<td>Reserved.</td>
</tr>
<tr>
<td>1</td>
<td>IPI Target Not Running</td>
<td>IsRunning bit of the target for a Singlecast/Broadcast/Multicast IPI is not set in the physical APIC ID table.</td>
<td>Index of the physical or logical APIC ID table entry for the target virtual processor that was not scheduled on a physical core.</td>
</tr>
<tr>
<td>2</td>
<td>Invalid Target in IPI</td>
<td>Target ID invalid. This is due to one the following reasons:</td>
<td>Index of the physical or logical table entry for the invalid target.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• In logical mode: cluster &gt; max_cluster (64)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• In physical mode: target &gt; max_physical (512)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• address is not present in the physical or logical ID tables</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Invalid Backing Page Pointer</td>
<td>The vAPIC Backing Page Pointer field of the Physical APIC ID Table contained an invalid physical address.</td>
<td>For shorthand or broadcast delivery modes, index of the physical APIC ID Table containing the invalid address. For directed IPIs, index of the logical or physical APIC ID table depending on the destination mode.</td>
</tr>
<tr>
<td>&gt;3</td>
<td>Reserved</td>
<td>—</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

**AVIC Access to un-accelerated vAPIC register.** A guest access to an APIC register that is not accelerated by AVIC results in a #VMEXIT with the exit code of AVIC_NOACCEL. This fault is also generated if an EOI is attempted when the highest priority in-service interrupt is set for level-triggered mode. Additional data associated with this #VMEXIT event is returned in the EXITINFO1 and EXITINFO2 fields.

**EXITINFO1.** This field contains the offset of the un-accelerated virtual APIC register and a bit indicating whether a read or write operation was attempted.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:33</td>
<td>—</td>
<td>Reserved.</td>
</tr>
<tr>
<td>32</td>
<td>R/W</td>
<td>If set, write was attempted. If clear, read was attempted.</td>
</tr>
</tbody>
</table>

**Table 15-30. EXTINFO1 Fields**

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:33</td>
<td>—</td>
<td>Reserved.</td>
</tr>
<tr>
<td>32</td>
<td>R/W</td>
<td>If set, write was attempted. If clear, read was attempted.</td>
</tr>
</tbody>
</table>
EXITINFO2. This field contains extra information for the un-accelerated operation. If the EXITINFO1 fields indicate a write to the vAPIC EOI register (offset = B0h), bits 7:0 of this value contain the number of the highest in-service vector found in the virtual APIC ISR.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:12</td>
<td>—</td>
<td>Reserved.</td>
</tr>
<tr>
<td>11:4</td>
<td>APIC_Offset[11:4]</td>
<td>Offset within virtual vAPIC backing page at which read or write was attempted. APIC_Offset[3:0] = 0, since all registers are aligned on 16-byte boundaries.</td>
</tr>
<tr>
<td>3:0</td>
<td>—</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

### 15.30 SVM Related MSRs

SVM uses the following MSRs for various control purposes. These MSRs are available regardless of whether SVM is enabled in EFER.SVME. For details on implementation-specific features, see the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual applicable to your product.

#### 15.30.1 VM_CR MSR (C001_0114h)

The VM_CR MSR controls certain global aspects of SVM. The layout of the MSR is shown in Figure 15-25.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>—</td>
<td>Reserved.</td>
</tr>
<tr>
<td>31:0</td>
<td>Vector</td>
<td>Vector for attempted EOI; otherwise undefined.</td>
</tr>
</tbody>
</table>

![Figure 15-25. Layout of VM_CR MSR (C001_0114h)](image)
• DIS_A20M—Bit 2. If set, disables A20 masking.

• LOCK—Bit 3. When this bit is set, writes to LOCK and SVMDIS are silently ignored. When this bit is clear, VM_CR bits 3 and 4 can be written. Once set, LOCK can only be cleared using the SVM_KEY MSR (See Section 15.31.) This bit is not affected by INIT or SKINIT.

• SVMDIS—Bit 4. When this bit is set, writes to EFER treat the SVME bit as MBZ. When this bit is clear, EFER.SVME can be written normally. This bit does not prevent CPUID from reporting that SVM is available. Setting SVMDIS while EFER.SVME is 1 generates a #GP fault, regardless of the current state of VM_CR.LOCK. This bit is not affected by SKINIT. It is cleared by INIT when LOCK is cleared to 0; otherwise, it is not affected.

15.30.2 IGNNE MSR (C001_0115h)

The read/write IGNNE MSR is used to set the state of the processor-internal IGNNE signal directly. This is only useful if IGNNE emulation has been enabled in the HW_CR MSR (and thus the external signal is being ignored). Bit 0 specifies the current value of IGNNE; all other bits are MBZ.

15.30.3 SMM_CTL MSR (C001_0116h)

The write-only SMM_CTL MSR provides software control over SMM signals.

<table>
<thead>
<tr>
<th>63</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved, MBZ</td>
<td>RSM_CYCLE</td>
<td>EXIT</td>
<td>SMI_CYCLE</td>
<td>ENTER</td>
<td>DISMISS</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 15-26. Layout of SMM_CTL MSR (C001_0116h)**

Writing individual bits causes the following actions:

• DISMISS—Bit 0. Clear the processor-internal “SMI pending” flag.

• ENTER—Bit 1. Enter SMM: map the SMRAM memory areas, record whether NMI was currently blocked and block further NMI and SMI interrupts.

• SMI_CYCLE—Bit 2. Send SMI special cycle.

• EXIT—Bit 3. Exit SMM: unmap the SMRAM memory areas, restore the previous masking status of NMI and unconditionally reenable SMI.

• RSM_CYCLE—Bit 4. Send RSM special cycle.

Writes to the SMM_CTL MSR cause a #GP if platform firmware has locked the SMM control registers by setting HWCR[SMMLOCK].

Conceptually, the bits are processed in the order of ENTER, SMI_CYCLE, DISMISS, RSM_CYCLE, EXIT, though only the following bit combinations may be set together in a single write (for all other combinations of more than one bit, behavior is undefined):

• ENTER + SMI_CYCLE

• DISMISS + ENTER
• DISMISS + ENTER + SMI_CYCLE
• EXIT + RSM_CYCLE

The VMM must ensure that ENTER and EXIT operations are properly matched, and not nested, otherwise processor behavior is undefined. Also undefined are ENTER when the processor is already in SMM, and EXIT when the processor is not in SMM.

15.30.4 VM_HSAVE_PA MSR (C001_0117h)

The 64-bit read/write VM_HSAVE_PA MSR holds the physical address of a 4KB block of memory where VMRUN saves host state, and from which #VMEXIT reloads host state. The VMM software is expected to set up this register before issuing the first VMRUN instruction. Software must not attempt to read or write the host save-state area directly.

Writing this MSR causes a #GP if:

• any of the low 12 bits of the address written are nonzero, or
• the address written is greater than or equal to the maximum supported physical address for this implementation.

15.30.5 TSC Ratio MSR (C000_0104h)

Writing to the TSC Ratio MSR allows the hypervisor to control the guest's view of the Time Stamp Counter. The contents of TSC Ratio MSR sets the value of the TSCRatio. This constant scales the timestamp value returned when the TSC is read by a guest via the RDTSC or RDTSCP instructions or when the TSC, MPERF, or MPerfReadOnly MSRs are read via the RDMSR instruction by a guest running under virtualization.

This facility allows the hypervisor to provide a consistent TSC, MPERF, and MPerfReadOnly rate for a guest process when moving that process between cores that have a differing P0 rate. The TSCRatio does not affect the value read from the TSC, MPERF, and MPerfReadOnly MSRs when in host mode or when virtualization is disabled. System Management Mode (SMM) code sees unscaled TSC, MPERF and MPerfReadOnly values unless the SMM code is executed within a guest container. The TSCRatio value does not affect the rate of the underlying TSC, MPERF, and MPerfReadOnly counters, nor the value that gets written to the TSC, MPERF, and MPerfReadOnly MSRs counters on a write by either the host or the guest.

The TSC Ratio MSR specifies the TSCRatio value as a fixed-point binary number in 8.32 format, which is composed of 8 bits of integer and 32 bits of fraction. This number is the ratio of the desired P0 frequency to be presented to the guest relative to the P0 frequency of the core (See Section 17.1, “P-State Control,” on page 595). The reset value of the TSCRatio is 1.0, which sets the guest P0 frequency to match the core P0 frequency.

Note that:

\[ TSCFreq = \text{Core P0 frequency} \times \text{TSCRatio}, \text{ so } TSCRatio = \frac{(\text{Desired TSCFreq})}{\text{Core P0 frequency}}. \]
The TSC value read by the guest is computed using the TSC Ratio MSR along with the TSC_OFFSET field from the VMCB so that the actual value returned is:

\[
\text{TSC Value (in guest)} = (P0 \text{ frequency} \times \text{TSCRatio} \times t) + \text{VMCB.TSC OFFSET} + (\text{Last Value Written to TSC}) \times \text{TSCRatio}
\]

Where \( t \) is time since the TSC was last written via the TSC MSR (or since reset if not written).

The layout of the TSC Ratio MSR is illustrated in figure below.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Access Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:40</td>
<td>—</td>
<td>Reserved</td>
<td>Reserved, MBZ</td>
</tr>
<tr>
<td>39:32</td>
<td>INT</td>
<td>Integer Part</td>
<td>R/W</td>
</tr>
<tr>
<td>31:0</td>
<td>FRAC</td>
<td>Fractional Part</td>
<td>R/W</td>
</tr>
</tbody>
</table>

**Figure 15-27.** TSC Ratio MSR (C000_0104h)

**INT.** Integer Part. Bits 39:32. Integer part of TSCRatio.

**FRAC.** Fractional Part. Bits 39:32. Fractional part of TSCRatio.

\[
\text{TSCRatio} = \text{INT} + \text{FRAC} \times 2^{-32}
\]

CPUID Fn8000_000A_EDX[TscRateMsr] = 1 indicates support for the TSC Ratio MSR. See Section 3.3, “Processor Feature Identification,” on page 64 for more information on using the CPUID instruction.

### 15.31 SVM-Lock

The SVM-Lock feature allows software to prevent EFER.SVME from being set, either unconditionally or with a 64-bit key to re-enable SVM functionality.

Support for SVM-Lock is indicated by CPUID Fn8000_000A_EDX[SVML] = 1. On processors that support the SVM-Lock feature, SKINIT and STGI can be executed even if EFER.SVME=0. See descriptions of LOCK and SVMDIS bits in Section 15.30.1. When the SVM-Lock feature is not available, hypervisors can use the read-only VM_CR.SVMDIS bit to detect SVM (see Section 15.4).

#### 15.31.1 SVM_KEY MSR (C001_0118h)

The write-only SVM_KEY MSR is used to create a password-protected mechanism to clear VM_CR.LOCK.

When VM_CR.LOCK is zero, writes to SVM_KEY MSR set the 64-bit SVM Key value.
When VM_CR.LOCK is one, writes to SVM_KEY MSR compare the written value to the SVM Key value; if the values match and are non-zero, the VM_CR.LOCK bit is cleared. If the values mismatch or the SVM Key value is zero, the write to SVM_KEY is ignored, and VM_CR.LOCK is unmodified. Software should read VM_CR.LOCK after writing SVM_KEY to determine whether the unlock succeeded.

If SVM Key is zero when VM_CR.LOCK is one, VM_CR.LOCK can only be cleared by a processor reset.

To preserve the security of the SVM key, reading the SVM_KEY MSR always returns zero.

15.32 SMM-Lock

The SMM-Lock feature allows software to prevent System Management Interrupts (SMI) from being intercepted in SVM. The SmmLock bit is located in the HWCR MSR register.

15.32.1 SmmLock Bit — HWCR[0]

The SmmLock bit (bit 0) is located in the HWCR MSR (C001_0015h). When SmmLock is clear, it can be set to one. Once set, the bit cannot be cleared by software and writes to it are ignored. SmmLock can only be cleared using the SMM_KEY MSR (see Section 15.32.2), or by a processor reset. This bit is not affected by INIT or SKINIT. When SmmLock is set, other SMM configuration registers cannot be written. For complete information on the HWCR register, see the BIOS and Kernel Developer's Guide (BKDG) or Processor Programming Reference Manual applicable to your product.

15.32.2 SMM_KEY MSR (C001_0119h)

The write-only SMM_KEY MSR is used to create a password-protected mechanism to clear SmmLock.

When SmmLock is zero, writes to SMM_KEY MSR set the 64-bit SMM Key value.

When SmmLock is one, writes to SMM_KEY MSR compare the written value to the SMM Key value; if the values match and are non-zero, the SmmLock bit is cleared. If the values mismatch or the SMM Key value is zero, the write to SMM_KEY is ignored, and SmmLock is unmodified. Software should read SmmLock after writing SMM_KEY to determine whether the unlock succeeded.

If SMM_KEY MSR is equal to zero when SmmLock is one, SmmLock can only be cleared by a processor reset.

To preserve the security of the SMM key, reading SMM_KEY MSR always returns zero.

15.33 Nested Virtualization

Hardware support for improved performance of nested virtualization, which is the act of running a hypervisor as a guest under a higher-level hypervisor, is provided through the features described here.
These relieve the top-level hypervisor from performing certain common, high-overhead operations that can occur with nested virtualization.

15.33.1 VMSAVE and VMLOAD Virtualization

This feature allows the VMSAVE and VMLOAD instructions to execute in guest mode without causing a #VMEXIT. The VMCSR address in RAX is treated as a guest physical address and is translated to a host physical address. Any page fault in attempting that translation will result in a normal #VMEXIT with a nested page fault exit code. If the translation is successful, the register state transfer to or from the VMCSR will then be performed.

Support for virtualized VMSAVE and VMLOAD is indicated by CPUID Fn8000_000A_EDX[15]=1. When this feature is available, it must be explicitly enabled by setting bit 1 of VMCSR offset 0B8h to 1. This enable bit is only recognized when the hypervisor is in 64 bit mode, nested paging is enabled and Secure Encrypted Virtualization is disabled, otherwise attempted execution of a VMLOAD or VMSAVE in the guest will result in a #VMEXIT with a VMSAVE/VMLOAD exit code.

15.33.2 Virtual GIF (VGIF)

This feature allows STGI and CLGI to execute in guest mode and control virtual interrupts in guest mode while still allowing physical interrupts to be intercepted by the hypervisor. The presence of the VGIF feature is indicated by CPUID Fn8000_000A_EDX[16]=1. In order to provide this ability, two new bits are added to the VMCSR field at offset 60h:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>60h</td>
<td>9</td>
<td>VGIF value (0 – Virtual interrupts are masked, 1 – Virtual Interrupts are unmasked)</td>
</tr>
<tr>
<td>60h</td>
<td>25</td>
<td>Virtual GIF enable for this guest (0 - Disabled, 1 - Enabled)</td>
</tr>
</tbody>
</table>

When a VMRUN is executed and VGIF is enabled, the processor uses bit 9 as the starting value of the virtual GIF. It then provides masking capability for when virtual interrupts are taken. STGI executed in the guest sets bit 9 of VMCSR offset 60h and allows a virtual interrupt to be taken. CLGI executed in the guest clears bit 9 of VMCSR offset 60h and causes the virtual interrupt to be masked. Bit 9 in the VMCSR is also writeable by the hypervisor, and loaded on VMRUN and is saved on #VMEXIT.

The hypervisor can still use the STGI and CLGI intercept controls in the VMCSR to intercept execution of these in the guest regardless of VGIF enablement.

15.34 Secure Encrypted Virtualization

Secure Encrypted Virtualization (SEV) is available when the CPU is running in guest mode utilizing AMD-V virtualization features. SEV enables running encrypted virtual machines (VMs) in which the code and data of the virtual machine are secured so that the decrypted version is available only within
the VM itself. Each virtual machine may be associated with a unique encryption key so if data is accessed by a different entity using a different key, the SEV encrypted VM's data will be decrypted with an incorrect key, leading to unintelligible data.

It is important to note that SEV mode therefore represents a departure from the standard x86 virtualization security model, as the hypervisor is no longer able to inspect or alter all guest code or data. The guest page tables, managed by the guest, may mark data memory pages as either private or shared, thus allowing selected pages to be shared outside the guest. Private memory is encrypted using a guest-specific key, while shared memory is accessible to the hypervisor.

15.34.1 Determining Support for SEV

Support for memory encryption features is reported in CPUID 8000_001F[EAX] as described in Section 7.10.1, “Determining Support for Secure Memory Encryption,” on page 214. Bit 1 indicates support for Secure Encrypted Virtualization.

When memory encryption features are present, CPUID 8000_001F[EBX] and 8000_001F[ECX] supply additional information regarding the use of memory encryption, such as the number of keys supported simultaneously and which page table bit is used to mark pages as encrypted. Additionally, in some implementations, the physical address size of the processor may be reduced when memory encryption features are enabled, for example from 48 to 43 bits. In this example, physical address bits 47:43 would be treated as reserved except where otherwise indicated. When memory encryption is supported in an implementation, CPUID 8000_001F[EBX] reports any physical address size reduction present. Bits reserved in this mode are treated the same as other page table reserved bits, and will generate a page fault if found to be non-zero when used for address translation.

Full CPUID details for memory encryption features may be found in Volume 3, section E.4.17.

15.34.2 Key Management

Under the memory encryption extensions defined here, each SEV-enabled guest virtual machine is associated with a memory encryption key, and the SME mode (if used, see Section 7.10 on page 214) is associated with a separate key. Key management for the SEV feature is not handled by the CPU but rather by a separate processor known as the AMD Secure Processor (AMD-SP) which is present on AMD SOCs. A detailed discussion of AMD-SP operation is beyond the scope of this manual.

CPU software is not aware of the values of these keys but the hypervisor should coordinate the loading of virtual machine keys through the AMD-SP driver. This coordination will also determine which ASID the hypervisor should use for a particular guest. Under SEV, the ASID is used as the key index that identifies which encryption key is used to encrypt/decrypt memory traffic associated with that SEV-enabled guest. Encryption keys themselves are never visible to CPU software and are never stored off-chip in the clear.
15.34.3 Enabling SEV

Prior to starting an encrypted VM, software must enable MemEncryptionModEn through MSR C001_0010 (SYSCFG) as described in Section 7.10.2, “Enabling Memory Encryption Extensions,” on page 215. SEV may then be enabled on a specific virtual machine during the VMRUN instruction if the hypervisor sets the SEV enable bit in VMCB offset 090h.

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Bit[s]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>090h</td>
<td>0</td>
<td>Enable nested paging</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Enable Secure Encrypted Virtualization</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Enable Encrypted State for Secure Encrypted Virtualization</td>
</tr>
<tr>
<td></td>
<td>63-3</td>
<td>Reserved, SBZ</td>
</tr>
</tbody>
</table>

When SEV is enabled in a guest, the following additional consistency checks are performed during VMRUN:

- Nested paging (VMCB offset 090h, bit 0) must be enabled
- MSR C001_0015 (HWCR) [SmmLock] must be set
- ASID (VMCB offset 058h) must be within the allowed range for SEV

The allowed ASIDs for SEV operation may be a subset of the overall number of hardware supported ASIDs. In this scenario, SEV-enabled guests must use ASIDs in the defined subset, while non-SEV enabled guests can use the remaining ASID range. The range of ASIDs allowed for SEV-enabled guests is from 1 to a maximum value defined via CPUID 8000_001F[ECX].

Note that on systems where CPUID Fn8000_001F_EAX[11] is set to 1, the hypervisor must be in 64-bit mode in order to execute a VMRUN to an SEV-enabled guest. If not the VMRUN fails with a VMEXIT_INVALID error code.

If any of the above consistency checks fail when SEV is enabled on a guest, the VMRUN instruction will terminate with a VMEXIT_INVALID error code. If MemEncryptionModEn is 0, SEV cannot be enabled and the VMCB control bit for SEV is ignored.

15.34.4 Supported Operating Modes

Secure Encrypted Virtualization may be enabled on guests running in any operating mode. However the guest is only able to control memory encryption when operating in long mode or legacy PAE mode. In all other modes, all guest memory accesses are unconditionally considered private and are encrypted with the guest-specific key.

15.34.5 SEV Encryption Behavior

When a guest is executed with SEV enabled, the guest page tables are used to determine the C-bit for a memory page and hence the encryption status of that memory page. This allows a guest to determine which pages are private or shared, but this control is available only for data pages. Memory accesses on behalf of instruction fetches and guest page table walks are always treated as private, regardless of
the software value of the C-bit. This behavior ensures non-guest entities (such as the hypervisor) cannot inject their own code or data into an SEV-enabled guest. If a guest does wish to make data in instruction pages or page tables accessible to code outside of the guest, this data must be explicitly copied into a shared data page.

Note that while the guest may choose to set the C-bit explicitly on instruction pages and page table addresses, the value of this bit is a don't-care in such situations as hardware always performs these as private accesses.

15.34.6 Page Table Support

An SEV-enabled guest controls encryption in its own guest page tables using the C-bit defined by CPUID 8000_001F[EBX]. This location is the same C-bit location as defined under SME (Section 7.10, “Secure Memory Encryption,” on page 214) in non-virtualized mode. If the C-bit is an address bit, this bit is masked from the guest physical address when it is translated through the nested page tables. Consequently, the hypervisor does not need to be aware of which pages the guest has chosen to mark private.

For example if the C-bit is address bit 47, when a guest accesses virtual address 0x54321, it might be translated to guest physical address 0x8000_00AB_C321, indicating the page should be encrypted with the private guest key. When this guest physical address is translated through the nested page tables, host virtual address 0xAB_C321 is used for translation. The C-bit value from the guest physical address is saved and used on the final system physical address after the nested table translation as shown in Figure 15-28.

Note that because guest physical addresses are always translated through the nested page tables, the size of the guest physical address space is not impacted by any physical address space reduction indicated in CPUID 8000_001F[EBX]. If the C-bit is a physical address bit however, the guest physical address space is effectively reduced by 1 bit.
15.34.7 Restrictions

As with SME, some hardware implementations may not enforce coherency between mappings of the same physical page with different encryption enablement or keys. In such a system, when the encryption enablement or key for a particular memory page is to be changed, software must first ensure the page is flushed from all CPU caches. Certain conventional cache flushing techniques may not work however; see Section 15.34.9 for further details.

Note that if the hardware implementation enforces coherency across encryption domains as indicated by CPUID Fn8000_001F_EAX[10] then this flush is not required.

15.34.8 SEV Interaction with SME

SEV may be used in conjunction with SME mode. In this scenario, the guest page tables control encryption for guest memory, and the host (nested) page tables control encryption for shared memory. This behavior is summarized in Table 15-32. SEV is considered active when the CPU is in guest mode and the guest has SEV enabled in the VMCB.

Figure 15-28. Guest Data Request
Note that during a nested page table walk, it is possible for both the guest page tables to be encrypted and the nested page tables to be encrypted. In this scenario, the guest page tables are decrypted using the guest private encryption key, and the nested page tables are decrypted using the host (SME) encryption key.

Guest data accesses that are marked shared (C=0) by the guest may still be optionally encrypted using the host (SME) key if the pages are marked encrypted in the nested tables. If a page is marked encrypted in both the guest and nested tables, the guest tables have priority and the page will be encrypted using the guest key. This behavior is summarized in Table 15-33.

Table 15-32.  Encryption Control

<table>
<thead>
<tr>
<th>Type of Access</th>
<th>MemEncryptionModEn</th>
<th>Guest Mode</th>
<th>SEV Mode Active</th>
<th>Encrypted</th>
<th>Encryption Key</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Legacy Mode (memory encryption disabled)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>No</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Secure Memory Encryption Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>Optional</td>
<td>Host Key</td>
<td>Determined by page tables (CR3)</td>
</tr>
<tr>
<td>All</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Optional</td>
<td>Host Key</td>
<td>Determined by nested page tables (hCR3)</td>
</tr>
<tr>
<td>Secure Encrypted Virtualization Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction Fetch</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Yes</td>
<td>Guest Key</td>
<td></td>
</tr>
<tr>
<td>Guest Page Table Access</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Yes</td>
<td>Guest Key</td>
<td></td>
</tr>
<tr>
<td>Nested Page Table Access</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Optional</td>
<td>Host Key</td>
<td>Determined by nested page tables (hCR3)</td>
</tr>
<tr>
<td>Data Access</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Optional1</td>
<td>See Table 15-33: SEV/SME Interaction</td>
<td>Determined by guest page tables (gCR3) and nested page tables (hCR3)</td>
</tr>
</tbody>
</table>

Note:  
1. Encryption is guest-controlled in long mode and legacy PAE mode only. In all other modes, these accesses are always considered private and are encrypted with the guest key

Note that during a nested page table walk, it is possible for both the guest page tables to be encrypted and the nested page tables to be encrypted. In this scenario, the guest page tables are decrypted using the guest private encryption key, and the nested page tables are decrypted using the host (SME) encryption key.

Guest data accesses that are marked shared (C=0) by the guest may still be optionally encrypted using the host (SME) key if the pages are marked encrypted in the nested tables. If a page is marked encrypted in both the guest and nested tables, the guest tables have priority and the page will be encrypted using the guest key. This behavior is summarized in Table 15-33.
In the event the hypervisor wishes to read an encrypted page, it must first flush the guest view of that page from all CPU caches to ensure it is able to view the most recent copy of that data. This may be accomplished by issuing a WBINVD instruction, or by using the VMPAGE_FLUSH MSR (C001_011E). Either operation must be performed on all cores on which the guest has run. Support for the VMPAGE_FLUSH MSR is indicated in CPUID 8000_001F[EAX] bit 2.

The VMPAGE_FLUSH MSR is a write-only register that may be used to flush 4KB of data on behalf of a guest. The hypervisor writes the host linear address of the page and guest ASID to the MSR, and hardware will then perform a write-back invalidation of the page causing any dirty data to be encrypted and written to DRAM. Note that the VMPAGE_FLUSH MSR uses the standard host page tables to perform the page translation. The Page Flush MSR operation will hit on and evict guest-cached instances of the memory, whereas CLFLUSH instructions using this same translation will not.

<table>
<thead>
<tr>
<th>Bit[s]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:12</td>
<td><strong>VirtualAddr:</strong> Write-only. Host virtual address of page to flush</td>
</tr>
<tr>
<td>11:0</td>
<td><strong>ASID:</strong> Write-only. Guest ASID to use for the flush</td>
</tr>
</tbody>
</table>

The VMPAGE_FLUSH MSR will only flush memory pages marked private by the guest. If the hypervisor does not know if the memory page was marked private but wishes to evict the page from the cache, it should perform a standard CLFLUSH in addition to using the VMPAGE_FLUSH MSR.

Attempts to flush a host virtual address that is not mapped into a physical address or use of an ASID=0 will cause a #GP(0) fault.

**15.34.10 SEV_STATUS MSR**

Guests can determine what SEV features are currently active by reading the SEV_STATUS MSR (C001_0131). This MSR indicates which SEV features (if any) were enabled by the hypervisor in the last VMRUN for that guest as shown in Table 15-34. The SEV_STATUS MSR can only be read in guest context and is read-only. Additionally, accesses to the SEV_STATUS MSR cannot be intercepted by the hypervisor. The SEV_STATUS MSR is available on all platforms that support SEV. Bits 9:2 of SEV_STATUS reflect the enablement of various SEV-SNP security features as described in section 15.36.
15.34.11 Virtual Transparent Encryption (VTE)

The Virtual Transparent Encryption feature can be enabled to force all memory accesses within an SEV guest to be encrypted with the guest’s key. Support for this feature is indicated in CPUID Fn8000_001F[EAX] bit 16.

To enable this feature, the hypervisor must set VMCB offset 90h bit 5. Bit 5 is only observed when SEV (bit 1) is also set to 1 and SEV-ES (bit 2) is cleared to 0. In all other configurations of these bits (namely SEV disabled or SEV-ES enabled), bit 5 is ignored by hardware.

When this feature is enabled, CPU hardware treats the guest C-bit as 1 for all guest memory references. The actual C-bit in the guest page tables is ignored by hardware.

Guest address translation is unchanged, so the guest physical address (without the C-bit) is used for translation in the nested page tables.

15.35 Encrypted State (SEV-ES)

Encrypted VMs that use the SEV feature described in Section 15.34 may additionally use the SEV-ES feature to protect guest register state from the hypervisor. An SEV-ES VM's CPU register state is encrypted during world switches and cannot be directly accessed or modified by the hypervisor. This is designed to protect against attacks such as exfiltration (unauthorized reading of VM state) and control flow attacks (modifying VM state) including rollback attacks (restoring an earlier VM register state).

Table 15-34. SEV_STATUS MSR Fields

<table>
<thead>
<tr>
<th>Bit[s]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:10</td>
<td>Reserved</td>
</tr>
<tr>
<td>9</td>
<td>SNPBTBIsolation_Enabled: The guest was run with the BTB isolation feature enabled in SEV_FEATURES[7]</td>
</tr>
<tr>
<td>8</td>
<td>PreventHostIBS_Enabled: This guest was run with the PreventHostIBS feature enabled in SEV_FEATURES[6].</td>
</tr>
<tr>
<td>7</td>
<td>DebugSwap_Enabled: This guest was run with debug register swapping enabled in SEV_FEATURES[5].</td>
</tr>
<tr>
<td>6</td>
<td>AlternateInjection_Enabled: The guest was run with the Alternate Injection feature enabled in SEV_FEATURES[4]</td>
</tr>
<tr>
<td>5</td>
<td>RestrictedInjection_Enabled: The guest was run with the Restricted Injection feature enabled in SEV_FEATURES[3]</td>
</tr>
<tr>
<td>4</td>
<td>ReflectVC_Enabled: The guest was run with the ReflectVC feature enabled in SEV_FEATURES[2]</td>
</tr>
<tr>
<td>3</td>
<td>vTOM_Enabled: The guest was run with the Virtual TOM feature enabled in SEV_FEATURES[1]</td>
</tr>
<tr>
<td>2</td>
<td>SNP_Active: The guest was run in SNP-Active mode as selected by SEV_FEATURES[0]</td>
</tr>
<tr>
<td>1</td>
<td>SEV_ES_Enabled: The guest was run with the SEV-ES feature enabled in VMCB offset 90h</td>
</tr>
<tr>
<td>0</td>
<td>SEV_Enabled: The guest was run with the SEV feature enabled in VMCB offset 90h</td>
</tr>
</tbody>
</table>
SEV-ES includes architectural support for notifying a VM's operating system when certain types of world switches are about to occur, allowing the VM to selectively share information with the hypervisor when needed for functionality.

15.35.1 Determining Support for SEV-ES

SEV-ES support can be determined by reading CPUID Fn8000_001F[EAX] as described in Section 15.34.1. Bit 3 of EAX indicates support for SEV-ES.

15.35.2 Enabling SEV-ES

SEV-ES may be enabled on a per-VM basis by setting bit 2 in offset 90h of the VMCB. When enabling SEV-ES, the hypervisor must also enable SEV (offset 90h bit 1) and LBR virtualization (offset B8h bit 0). Additionally, all other programming requirements related to enabling SEV (see Section 15.34.3) must be satisfied when running an SEV-ES guest.

On some systems, there is a limitation on which ASID values can be used on SEV guests that are run with SEV-ES disabled. While SEV-ES may be enabled on any valid SEV ASID (as defined by CPUID Fn8000_001F[ECX]), there are restrictions on which ASIDs may be used for SEV guests with SEV-ES disabled. CPUID Fn8000_001F[EDX] indicates the minimum ASID value that must be used for an SEV-enabled, SEV-ES-disabled guest. For example, if CPUID Fn8000_001F[EDX] returns the value 5, then any VMs which use ASIDs 1-4 and which enable SEV must also enable SEV-ES.

Note that prior to running an SEV-ES VM for the first time, the hypervisor must coordinate with the AMD Secure Processor to create the initial encrypted state image for the guest VM.

15.35.3 SEV-ES Overview

The SEV-ES architecture is designed to protect guest VM register state by default, and only allow the guest VM itself to grant selective access as required. This additional security protection functionality is accomplished in two ways. First, all VM register state is saved and encrypted when a VM exit event (#VMEXIT) occurs. This state is decrypted and restored on a VMRUN only. Second, certain types of #VMEXIT events cause a new exception to be taken within the guest VM. This new exception (#VC, see Section 15.35.5) indicates that the guest VM performed some action which requires hypervisor involvement, an example of which would be an IO access by the VM. The guest #VC handler is responsible for determining what register state is necessary to expose to the hypervisor for the purpose of emulating this operation. The #VC handler also inspects the returned values from the hypervisor and updates the guest state if the output is deemed acceptable.

Register state that needs to be exposed utilizes a new structure called the Guest-Hypervisor Communication Block (GHCB). The GHCB location is chosen by the guest who maps the page as a shared memory page, thus allowing direct hypervisor access. Only state located in the GHCB can be read by the hypervisor as all state stored in the traditional VMCB save state structure is encrypted using the guest memory encryption key and integrity protected.
In the #VC handler, the guest may utilize a new instruction (Section 15.35.6) to perform a world switch and invoke the hypervisor. In response to this, the hypervisor can inspect the GHCB and determine the services requested by the guest.

### 15.35.4 Types of Exits

When SEV-ES is enabled, all #VMEXIT events are classified as either Automatic Exits (AE) or Non-Automatic Exits (NAE). AE events are generally events that occur asynchronously with respect to the guest execution (e.g. interrupts) or events that need not involve exposing any guest register state. All other #VMEXIT events are classified as NAE events, and with NAE events the guest is allowed to determine what register state (if any) to expose in the GHCB. During guest execution, #VMEXIT events (both AE and NAE) are only taken if the corresponding intercept bit in the VMCB control area is set.

The hypervisor is informed of specific AE events exclusively via the #VMEXIT codes within the EXITCODE field of the VMCB control area. NAE events result in a #VC exception which is handled by the guest. Table 15-35 lists the possible AE events, all other events are considered NAE events.

#### Table 15-35. AE Exitcodes

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Notes</th>
<th>HW Advances RIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>52h</td>
<td>VMEXIT_MC</td>
<td>Machine check exception</td>
<td>No</td>
</tr>
<tr>
<td>60h</td>
<td>VMEXIT_INTR</td>
<td>Physical INTR</td>
<td>No</td>
</tr>
<tr>
<td>61h</td>
<td>VMEXIT_NMI</td>
<td>Physical NMI</td>
<td>No</td>
</tr>
<tr>
<td>63h</td>
<td>VMEXIT_INIT</td>
<td>Physical INIT</td>
<td>No</td>
</tr>
<tr>
<td>64h</td>
<td>VMEXIT_VINTR</td>
<td>Virtual INTR</td>
<td>No</td>
</tr>
<tr>
<td>77h</td>
<td>VMEXIT_PAUSE</td>
<td>PAUSE instruction</td>
<td>Yes</td>
</tr>
<tr>
<td>78h</td>
<td>VMEXIT_HLT</td>
<td>HLT instruction</td>
<td>Yes</td>
</tr>
<tr>
<td>7Fh</td>
<td>VMEXIT_SHUTDOWN</td>
<td>Shutdown</td>
<td>No</td>
</tr>
<tr>
<td>8Fh</td>
<td>VMEXIT_EFER_WRITE_TRAP</td>
<td>See section 15.35.10</td>
<td>Yes</td>
</tr>
<tr>
<td>90h -9Fh</td>
<td>VMEXIT_CR[0-15]_WRITE TRAP</td>
<td>See section 15.35.10</td>
<td>Yes</td>
</tr>
<tr>
<td>400h</td>
<td>VMEXIT_NPF</td>
<td>Only if PFCODE[3]=0 (no reserved bit error)</td>
<td>No</td>
</tr>
<tr>
<td>403h</td>
<td>VMEXIT_VMEXIT</td>
<td>VMEXIT instruction</td>
<td>Yes</td>
</tr>
<tr>
<td>-1</td>
<td>VMEXIT_INVALID</td>
<td>Invalid guest state</td>
<td>–</td>
</tr>
<tr>
<td>-2</td>
<td>VMEXIT_BUSY</td>
<td>Busy bit was set in guest state (see Section 15.36.16)</td>
<td>–</td>
</tr>
</tbody>
</table>

In the case of exits due to specific instructions, the CPU will automatically advance the guest RIP in response to the AE so that execution will resume at the next instruction on a subsequent VMRUN.

In the case of nested page faults, these are treated as AEs only if there was no reserved bit error. This is intended to be used to help distinguish nested page faults due to demand misses (hypervisor needs to
allocate a page) vs MMIO emulation (hypervisor needs to emulate a device). Consequently, the hypervisor should set a reserved page table bit, such as a reserved address bit, on all MMIO pages that it intends to emulate. (This can include address bits that may become reserved when SEV is enabled; see Section 15.34.1.) This will ensure that MMIO page faults become NAE events, which is critical so the guest #VC handler can be invoked to assist in the MMIO emulation. Nested page faults that are AE events do not invoke any guest handler and the hypervisor is intended to allocate memory as needed and then resume the guest.

Note that when a guest is running with SEV-ES enabled, instruction bytes (VMCB offset D0h) are never saved to the VMCB on a nested page fault.

15.35.5 #VC Exception

The VMM Communication Exception (#VC) is always generated by hardware when an SEV-ES enabled guest is running and an NAE event occurs. The #VC exception is a precise, contributory, fault-type exception utilizing exception vector 29. This exception cannot be masked. The error code of the #VC exception is equal to the #VMEXIT code (see Appendix C) of the event that caused the NAE.

In response to a #VC exception, a typical flow would involve the guest handler inspecting the error code to determine the cause of the exception and deciding what register state must be copied to the GHCB for the event to be handled. The handler should then execute the VMGEXIT instruction to create an AE and invoke the hypervisor. After a later VMRUN, guest execution will resume after the VMGEXIT instruction where the handler can view the results from the hypervisor and copy state from the GHCB back to its internal state as needed. This flow is shown in Figure 15-29.

Note that it is inadvisable for the hypervisor to set the VMCB intercept bit for the #VC exception as this would prevent proper handling of NAEs by the guest. Similarly, the hypervisor should avoid setting intercept bits for events that would occur in the #VC handler (such as IRET).
Figure 15-29.  EXAMPLE #VC FLOW
15.35.6 VMGExit

The VMGEXIT instruction creates an AE and is intended to allow a guest #VC handler to invoke the hypervisor when needed. The opcode for VMGEXIT is the same as VMMCALL (0F 01 D9) but with a REP prefix (F3/F2). VMGEXIT causes an AE with the VMEXIT_VMGEXIT code and behaves like a trap so that upon a subsequent VMRUN, execution resumes following the VMGEXIT. There is no hypervisor intercept bit for VMGEXIT as the instruction unconditionally causes an AE when executed in an SEV-ES guest.

The VMGEXIT opcode is only valid within a guest when run with SEV-ES mode active. If the guest is not run with SEV-ES mode active, the VMGEXIT opcode will be treated as a VMMCALL opcode and will behave exactly like a VMMCALL.

15.35.7 GHCB

The GHCB is an unencrypted memory page used to communicate register state between the SEV-ES guest and the hypervisor. The guest VM is able to set the location of the GHCB via the GHCB MSR (C001_0130). This value is also included in the VMCB and is saved/restored on VMRUN/#VMEXIT respectively.

The GHCB MSR is used to set up the location of the GHCB memory page. The format of this MSR is defined below:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:0</td>
<td>Guest physical address of GHCB</td>
</tr>
</tbody>
</table>

The value of this MSR is saved/restored from the VMCB offset 0A0h. It is recommended software write this MSR with a page-aligned address. The GHCB MSR can be read/written only in guest mode, attempts to access this MSR in host mode will result in a #GP.

Hardware never accesses the GHCB directly, and as a result the format of the GHCB is not fixed.

15.35.8 VMRUN

When SEV-ES is enabled, the VM save state area does not reside at offset 400h in the VMCB page. Instead it resides starting at offset 0h in a separate page called the VM Save Area (VMSA) as indicated by the VMSA Pointer at offset 108h. The VMSA Pointer value is stored as a host physical address.

Hardware always accesses the VMSA save state area using encrypted memory accesses utilizing the guest's memory encryption key.

When hardware executes a VMRUN instruction and the VMCB indicates SEV-ES is enabled for the guest, the hardware loads guest state from the encrypted save state area indicated by the VMSA Pointer. Also, the VMRUN instruction will perform the following actions in addition to the standard VMRUN behavior:

- Calculate a checksum over guest state to verify integrity
- Perform a VMLOAD to load additional guest register state
• Load guest GPR state
• Load guest FPU state

When a guest has SEV-ES enabled, the encrypted VM state save area definition is expanded to include all GPR and FPU state (see Appendix B). If any part of the VMRUN flow faults or if the integrity checksum fails to match, a #VMEXIT(VMEXIT_INVALID) is generated.

Note that if SEV-ES is enabled, the VMRUN instruction ignores bits 10:5 of the VMCB clean bits and always reloads the full guest state.

Also note that for SEV-ES guests, while the full guest state is loaded on VMRUN only the minimal hypervisor state defined by the legacy VMRUN instruction (see Section 15.5.1) is saved to the host save area. The hypervisor itself should save its desired additional segment state and GPR values to the host save area since these values will be restored by hardware on a subsequent VMEXIT. Hardware does not automatically save host state such as FS, STAR, or GPR values from the hypervisor on a VMRUN. See Appendix B for a detailed breakdown of each piece of VMCB state.

Finally, note that event injection for SEV-ES guests is restricted. Software interrupts and exception vectors 3 and 4 may not be injected. If this is attempted, the VMRUN will fail with a VMEXIT_INVALID error code.

15.35.9 Automatic Exits

When an automatic exit event occurs while an SEV-ES enabled guest is executing, hardware automatically saves guest state to the encrypted save state area and restores hypervisor state from the host save area. Specifically, in addition to the standard state saved/restored by the VMEXIT flow, hardware will also perform the following steps:
• Perform a VMSAVE to save additional guest register state
• Save guest GPR state
• Save guest FPU state
• Calculate and store a checksum over the guest state for use in a subsequent VMRUN
• Perform a VMLOAD to load additional host register state
• Load host GPR state
• Re-initialize FPU state to their reset values

The loading of host GPR state from the host save area is done using the format of the expanded VMCB described in Appendix B. All register state is either loaded from this location or re-initialized to default values so no guest register state is visible to the hypervisor.

15.35.10 Control Register Write Traps

The use of CR[0-15]_WRITE intercepts are discouraged for guests that are run with SEV-ES. These intercepts occur prior to the control register being modified, and the hypervisor is not able to modify the control register itself since the register is located in the encrypted state image. Hypervisors are
encouraged to use the new CR[0-15]_WRITE_TRAP and EFER_WRITE_TRAP intercept bits instead which cause an AE after a control register has been modified. These intercepts enable the hypervisor to track the guest mode and verify if desired features are being enabled. When these traps are taken, the new value of the control register is saved in EXITINFO1. CR write traps are only supported for SEV-ES guests.

Note that writes by SEV-ES guests to EFER.SVME are always ignored by hardware.

15.36 Secure Nested Paging (SEV-SNP)

The SEV-SNP features enable additional protection for encrypted VMs designed to achieve stronger isolation from the hypervisor. SEV-SNP is used with the SEV and SEV-ES features described in Section 15.34 and Section 15.35 respectively and requires the enablement and use of these features.

Primarily, SEV-SNP provides integrity protection of VM memory to help prevent hypervisor-based attacks that rely on guest data corruption, aliasing, replay, and various other attack vectors. To achieve this, a new system-wide data structure called the Reverse Map Table (RMP) is used to perform additional security checks on memory access as described in Section 15.36.3.

In addition to memory protection, SEV-SNP also includes several security features including a new Virtual Machine Privilege Level (VMPL) architecture, interrupt injection restrictions, and side-channel protection. These features are designed to enable additional use models and enhanced security protections.

While this chapter describes the CPU hardware behavior of SEV-SNP, the technology also requires the use of the AMD Secure Processor (AMD-SP) SEV-SNP Application Binary Interface (ABI) to manage the lifecycle events of SEV-SNP VMs. See the SEV-SNP ABI specification on AMD’s website for more details.

15.36.1 Determining Support for SEV-SNP

Support for SEV-SNP can be determined by reading CPUID Fn8000_001F[EAX] as described in Section 15.34.1. Bit 4 indicates support for SEV-SNP, while bit 5 indicates support for VMPLs. The number of VMPLs available in an implementation is indicated in bits [15:12] of CPUID Fn8000_001F[EBX].

CPUID Fn8000_001F[EAX] also indicates support for additional security features used with SEV-SNP guests, which are described in the following sections.
15.36.2 Enabling SEV-SNP

SEV-SNP depends on SEV for confidentiality protection. Before enabling SEV-SNP, the MemEncryptionModEn bit in MSR C001_0010 (SYSCFG) must be set, and all programming requirements described in Section 15.34.3 must be satisfied.

Enabling SEV-SNP requires a two-step initialization procedure:

1. Construct the Reverse Map Table (RMP) as described in Section 15.36.4.
2. Set VMPLEn and SecureNestedPagingEn in MSR C001_0010 (SYSCFG) on every core in the system.

After the SEV-SNP feature has been globally enabled, SEV-SNP can be activated on a per-VM basis by setting bit 0 of the SEV_FEATURES field at offset 3B0h of the VMSA during VM creation. SEV-SNP activated VMs must also enable SEV-ES as described in Section 15.35.2 and SEV as described in Section 15.34.3.

In this chapter, the term SNP-enabled indicates that SEV-SNP is globally enabled in the SYSCFG MSR. The term SNP-active indicates that SEV-SNP is enabled for a specific VM in the SEV_FEATURES field of its VMSA. While SNP-enabled systems support both SNP-active and non-SNP-active VMs, SNP-active VMs can only run on SNP-enabled systems.

15.36.3 Reverse Map Table

The Reverse Map Table (RMP) is a structure shared globally by all logical processors that resides in system memory and is used to ensure a one-to-one mapping between system physical addresses and guest physical addresses. Each page of physical memory that is potentially assignable to guests has one entry within the RMP. RMP entries contain the security attributes of the system physical page as described in Table 15-36.

Table 15-36. Fields of an RMP Entry

<table>
<thead>
<tr>
<th>Name</th>
<th>Notes</th>
</tr>
</thead>
</table>
| Assigned              | Flag indicating that the system physical page is assigned to a guest or to the AMD-SP.  
  0: Owned by the hypervisor  
  1: Owned by a guest or the AMD-SP |
| Page_Size             | Encoding of the page size.  
  0: 4kB page  
  1: 2MB page |
| Immutable             | Flag indicating that software can alter the entry via x86 RMP manipulation instructions.  
  0: RMP entry can be altered by software  
  1: RMP entry cannot be altered by software |
| Guest_Physical_Address| Guest physical address associated with the page |
The integrity of the RMP is maintained by restricting software manipulation of it to the following special-purpose instructions:

- **RMPUPDATE**: Available to the hypervisor to alter the Guest_Physical_Address, Assigned, Page_Size, Immutable, and ASID fields of an RMP entry. See Section 15.36.5 for details.

- **PSMASH**: Allows the hypervisor to split a 2MB entry in the RMP into 512 4KB entries in the RMP. See Section 15.36.11 for details.

- **RMPADJUST**: Allows a guest to alter the VMPL permission masks of the RMP entry. See Section 15.36.7 for details.

- **PVALIDATE**: Allows a guest to write to the Validated flag in the RMP entry. See Section 15.36.6 for details.

When SEV-SNP is globally enabled, it adds more restrictions to page access controls. The hypervisor and the guests use the above instructions to enforce these restrictions on memory accesses. A violation of memory access restrictions indicated by the RMP will result in an exception. See Section 15.36.10 for details.

### 15.36.4 Initializing the RMP

MSR C001_0132 (RMP_BASE) defines the system physical address of the first byte of the RMP. The MSR C001_0133 (RMP_END) defines the system physical address of last byte of the RMP. Software must program RMP_BASE and RMP_END identically for each core in the system and before enabling SEV-SNP globally.

RMP_BASE and (RMP_END+1) must be 8KB aligned. The AMD-SP may place further alignment requirements on these registers. Refer to the latest AMD-SP specifications to determine the required alignment.
The region of memory between RMP_BASE and RMP_END contains a 16KB region used for processor bookkeeping followed by the RMP entries, which are each 16B in size. The size of the RMP determines the range of physical memory that the hypervisor can assign to SNP-active virtual machines at runtime. The RMP covers the system physical address space from address 0h to the address calculated by:

$$((\text{RMP}\_\text{END} + 1 - \text{RMP}\_\text{BASE} - 16\text{KB}) / 16\text{B}) \times 4\text{KB}$$

For example, if the RMP_BASE is equal to 10_0000h, then to cover the first 4GB of physical memory, RMP_END must be set to 110_3FFFh, which makes the RMP just over 16MB.

Once SEV-SNP is globally enabled, memory accesses are restricted by RMP checks. To ensure that the RMP starts in a known and non-restrictive state, software should write zeros to all memory from RMP_BASE to RMP_END before setting the SecureNestedPagingEn bit in the SYSCFG MSR. The hypervisor then requests the AMD-SP to finalize the initialization of the RMP. The AMD-SP initializes the RMP to prevent all software from directly writing to the memory between RMP_BASE and RMP_END. All subsequent RMP entry manipulation must occur either via the x86 RMP manipulation instructions or through interactions with the AMD-SP.

### 15.36.5 Hypervisor RMP Management

The hypervisor manages the SEV-SNP security attributes of pages assigned to SNP-active guests by altering the RMP entries of those pages. Because the RMP is initialized by the AMD-SP to prevent direct access to the RMP, the hypervisor must use the RMPUPDATE instruction to alter the entries of the RMP. RMPUPDATE allows the hypervisor to alter the Guest_Physical_Address, Assigned, Page_Size, Immutable, and ASID fields of an RMP entry.

SEV-SNP associates an owner with each system physical page through settings of the Assigned, ASID, and Immutable fields of the page’s RMP entry according to Table 15-37. A page can be owned by the hypervisor, a guest, or the AMD-SP.

**Table 15-37. RMP Page Assignment Settings**

<table>
<thead>
<tr>
<th>Owner</th>
<th>Assigned</th>
<th>ASID</th>
<th>Immutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hypervisor</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Guest</td>
<td>1</td>
<td>ASID of the guest</td>
<td>-</td>
</tr>
<tr>
<td>AMD-SP</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

When the hypervisor assigns a page to a guest, it must also set the Guest_Physical_Address and Page_Size to match the nested page table mapping for the guest. If not, access to the page by the guest will result in a fault. See Section 15.36.10 for details on the RMP access checks.

The hypervisor may transition any page that has Immutable set to 0 into a hypervisor-owned page by using RMPUPDATE to set Assigned to 0 and ASID to 0. To transition a page that has Immutable set to 1, the hypervisor must request the AMD-SP to transition the page.
The RMP initialization requirement to write zeros to the RMP (see Section 15.36.4) results in all pages in the system initially belonging to the hypervisor. Any memory pages which are not covered by the RMP are considered permanent hypervisor pages. For example, if the RMP is configured to only cover the first 4GB of memory then all memory above 4GB is considered hypervisor memory for the purpose of RMP access checks.

15.36.6 Page Validation

Each page assigned to a VM is either validated or unvalidated, as indicated by the Validated flag in the page’s RMP entry. Memory accesses by the VM to private pages that are unvalidated generate a #VC. All pages are initially assigned as unvalidated.

The VM may use the PVALIDATE instruction to either set or clear the Validated flag of a page. It is expected that VMs would use PVALIDATE to set the Validated flag during VM startup to gain access to the memory the hypervisor has assigned. The VM may later use PVALIDATE to clear the Validated flag if its memory space is being reduced, such as after a memory hot-plug event.

Page validation allows a VM to detect an unexpected remapping of its pages by the hypervisor. Before accessing a page, the VM must validate the page. Once validated, any use of RMPUPDATE by the hypervisor to unassign, reassign, or remap the page will cause the page to become unvalidated. The VM can then detect tampering with the page mapping via the #VC that occurs from accessing unvalidated pages.

PVALIDATE takes a page size as an input parameter indicating that either a 4KB or 2MB page should be validated. If the VM attempts to use PVALIDATE on a 4KB page that is mapped to a 2MB or 1GB page in the nested page table, PVALIDATE generates an #VMEXIT(NPF). In this case, the hypervisor can smash the larger page into 4KB pages using the PSMASH instruction as described in Section 15.36.11. If the VM attempts to use PVALIDATE on a 2MB guest page that is mapped to 4KB nested pages, PVALIDATE returns an error indication to the VM. The VM can instead attempt to execute PVALIDATE for each of the 4KB pages individually.

15.36.7 Virtual Machine Privilege Levels

SEV-SNP allows each virtual CPU (vCPU) of a VM to be assigned a Virtual Machine Privilege Level (VMPL), indicated in the VMPL field of a vCPU’s VMSA.

VMPLs are identified numerically starting at 0 with VMPL0 being the most privileged. The number of VMPLs available in an implementation is indicated in bits [15:12] of CPUID Fn8000_001F[EBX]. The VMPL feature enables a guest to sub-divide its address space and implement vCPU-specific access controls on a page-by-page basis.

The processor restricts guest memory accesses based on VMPL permission masks in RMP entries. Each RMP entry contains a set of permission masks, one mask for each implemented VMPL. On memory accesses, the processor checks the current VMPL permission mask of the page to determine whether the access is allowed. The permission mask bits are defined in Table 15-38.
When a guest access results in a #VMEXIT(NPF) due to a VMPL permission violation, an error code bit in EXITINFO1 is set as described in Section 15.36.10.

When the hypervisor assigns a page to a guest using RMPUPDATE, full permissions are enabled for VMPL0 and are disabled for all other VMPLs. A VM can then use the RMPADJUST instruction to modify the permissions of VMPLs numerically higher than its own. For example, a vCPU executing at VMPL0 could use RMPADJUST to restrict a page of memory to be only read-write but not executable at VMPL1. However, the vCPU executing at VMPL1 could not alter its own permissions or the permissions of VMPL0.

Further, RMPADJUST cannot be used to grant greater permissions than what is allowed by the permission mask for the current VMPL. For example, if VMPL1 attempts to grant write permission to a page to VMPL2, but VMPL1 does not have write permission to the page, RMPADJUST will fail.

### 15.36.8 Virtual Top-of-Memory

In the VMSA of an SNP-active guest, the VIRTUAL_TOM field designates a 2MB aligned guest physical address called the virtual top of memory. When bit 1 (vTOM) of SEV_FEATURES is set in the VMSA of an SNP-active VM, the VIRTUAL_TOM field is used to determine the C-bit for data accesses instead of the guest page table contents. All data accesses below VIRTUAL_TOM are accessed with an effective C-bit of 1 and all addresses at or above VIRTUAL_TOM are accessed with an effective C-bit of 0. Note that page table accesses and instruction fetches always have an effective C-bit of 1, regardless of the value of VIRTUAL_TOM or whether the feature is enabled.

When virtual top of memory is enabled in SEV_FEATURES, the C-bit in the guest page table entries must be zero for all accesses. Any guest memory accesses with the C-bit set to 1 in the guest page tables will result in a #PF due to a reserved bit error.

### 15.36.9 Reflect #VC

When running an SEV-SNP VM, the CPU generates #VC exceptions in response to events that may require hypervisor interaction. #VC exceptions and the events that may lead to them are discussed in

---

Table 15-38. VMPL Permission Mask Definition

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Read</td>
<td>0: Reads cause #VMEXIT(NPF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Reads are allowed</td>
</tr>
<tr>
<td>1</td>
<td>Write</td>
<td>0: Writes cause #VMEXIT(NPF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Writes are allowed</td>
</tr>
<tr>
<td>2</td>
<td>Execute-User</td>
<td>0: Execution at CPL 3 causes #VMEXIT(NPF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Execution at CPL 3 is allowed</td>
</tr>
<tr>
<td>3</td>
<td>Execute-Supervisor</td>
<td>0: Execution at CPL &lt; 3 causes #VMEXIT(NPF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Execution at CPL &lt; 3 is allowed</td>
</tr>
<tr>
<td>4-7</td>
<td>Reserved</td>
<td>SBZ</td>
</tr>
</tbody>
</table>
Section 15.35.5. SEV-SNP VMs may either choose to handle #VC exceptions directly in their current guest context or turn #VC exceptions into Automatic Exits. This behavior is controlled by bit 2 (ReflectVC) of SEV_FEATURES. If this bit is set to 1, then any event that would otherwise lead to a #VC exception is instead turned into an Automatic Exit.

When a #VC is turned into an Automatic Exit, the guest VM terminates with an exit code of VMEXIT_VC. The error code for the #VC, which reflects the event which led to the #VC (e.g., VMEXIT_CPUID), is saved to the GUEST_EXITCODE field in the VMSA. Additional information about the event that caused the #VC is saved to the GUEST_EXITINFO1, GUEST_EXITINFO2, GUEST_EXITINTINFO, and GUEST_NRIP fields in the VMSA. The information saved to these fields is the same as the standard exit information provided for the event that occurred. For example, if the VM performs a port I/O instruction which is marked for interception, the GUEST_EXITCODE field will be set to VMEXIT_IOIO and the GUEST_EXITINFO1 field will contain information about the I/O port access, as defined in Section 15.10.2.

The Reflect #VC feature enables #VC events to be handled by a vCPU other than the one that initiated them. For example, a guest may contain one vCPU running at VMPL0 and another one running at VMPL3 with ReflectVC enabled. When the vCPU running at VMPL3 encounters a #VC condition, the information is saved to its VMSA and control is returned to the hypervisor. The hypervisor may then run the VMPL0 vCPU which can read the exit information saved to the VMPL3 vCPU’s VMSA, interact with the hypervisor as required, write appropriate response data back into the VMPL3 vCPU VMSA, and instruct the hypervisor to resume execution of the faulting vCPU.

If the #VC event occurred during the processing of an interrupt or exception, the GUEST_EXITINTINFO.V bit will be set. If the Alternate Injection feature is enabled (see Section 15.36.15), hardware will automatically set the VINTR_CTL[BUSY] bit in the VMSA. This enables a higher privileged VMPL to re-inject the event that caused the #VC.

The GUEST_EXITCODE, GUEST_EXITINFO1, GUEST_EXITINFO2, GUEST_EXITINTINFO, and GUEST_NRIP fields are populated by hardware on every Automatic Exit, regardless of the ReflectVC feature. For Automatic Exits other than reflected #VC’s, these fields are set to the same values that are set in the unencrypted VMCB.

15.36.10 RMP and VMPL Access Checks

When SEV-SNP is enabled globally, the processor places restrictions on all memory accesses based on the contents of the RMP, whether the accesses are performed by the hypervisor, a legacy guest VM, a non-SNP guest VM or an SNP-active guest VM. The processor may perform one or more of the following checks depending on the context of the access:

- RMP-Covered: Checks that the target page is covered by the RMP. A page is covered by the RMP if its corresponding RMP entry is below RMP_END. Any page not covered by the RMP is considered a Hypervisor-Owned page.
- Hypervisor-Owned: Checks that if the target page is covered by the RMP then the Assigned bit of the target page is 0. If the page table entry that specifies the sPA indicates that the target page size is 2MB, then all RMP entries for the 4KB constituent pages of the target page must have the
Assigned bit set to 0. Accesses to 1GB pages only install 2MB TLB entries when SEV-SNP is enabled, therefore this check treats 1GB accesses as 2MB accesses for purposes of this check.

- **Guest-Owned**: Checks that the ASID field of the RMP entry of the target page matches the ASID of the current VM.
- **Reverse-Map**: Checks that the Guest_Physical_Address of the RMP entry of the target page matches the guest physical address of the translation.
- **Validated**: Checks that the Validated field of the RMP entry of the target page is 1.
- **Mutable**: Checks that the Immutable field of the RMP entry of the target page is 0.
- **Page-Size**: Checks that the following conditions are met:
  - If the nested page table indicates a 2MB or 1GB page size, the Page_Size field of the RMP entry of the target page is 1.
  - If the nested page table indicates a 4KB page size, the Page_Size field of the RMP entry of the target page is 0.
- **VMPL**: Checks that the VMPL permission mask allows access. See Section 15.36.7 for details.

Table 15-39 describes under which conditions each check is performed and what fault is produced on failure.

### Table 15-39. RMP Memory Access Checks

<table>
<thead>
<tr>
<th>Host/Guest</th>
<th>SNP-Active</th>
<th>Type of Access</th>
<th>C-Bit</th>
<th>Check</th>
<th>Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>-</td>
<td>Date Write Page Table Access</td>
<td>-</td>
<td>Hypervisor-Owned</td>
<td>#PF</td>
</tr>
<tr>
<td>Guest</td>
<td>No</td>
<td>Date Write Page Table Access</td>
<td>-</td>
<td>Hypervisor-Owned</td>
<td>#VMEXIT(NPF)</td>
</tr>
<tr>
<td>Guest</td>
<td>Yes</td>
<td>Instruction Fetch Page Table Access</td>
<td>-</td>
<td>RMP-Covered, Guest-Owned, Reverse-Map, Mutable, Page-Size</td>
<td>#VMEXIT(NPF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Validated</td>
<td>#VC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VMPL</td>
<td>#VMEXIT(NPF)</td>
</tr>
<tr>
<td>Guest</td>
<td>Yes</td>
<td>Date Write 0</td>
<td></td>
<td>Hypervisor-Owned</td>
<td>#VMEXIT(NPF)</td>
</tr>
<tr>
<td>Guest</td>
<td>Yes</td>
<td>Date Write 1</td>
<td></td>
<td>RMP-Covered, Guest-Owned, Reverse-Map, Mutable, Page-Size</td>
<td>#VMEXIT(NPF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Validated</td>
<td>#VC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VMPL</td>
<td>#VMEXIT(NPF)</td>
</tr>
</tbody>
</table>
In addition, any memory access that results in an RMP check may result in an RMP violation (#PF or #VMEXIT(NPF)) if the accessed RMP entries are in use by other logical processors. In this case, software should retry the access.

If a memory access results in a modification of the Accessed or Dirty bits in a page table entry, this page table modification is treated similarly to data write accesses by SEV-SNP. For any such page table modification access, the page size of the access is inherently 4KB.

If the virtual TOM feature (see Section 15.36.8) is enabled, then the Virtual TOM setting is used to determine the C-bit for a given guest access. Guest physical addresses below Virtual TOM are considered to have a C-bit set to 1.

The following page-fault error bits are set on an RMP check related #PF:

- Bit 31 (RMP): Set to 1 if the fault was caused due to an RMP check or a VMPL check failure, 0 otherwise. All RMP violations described in this section will set this bit to 1.

Additionally, the following page-fault error bits may be set on a #VMEXIT(NPF) in EXITINFO1:

- Bit 34 (ENC): Set to 1 if the guest’s effective C-bit was 1, 0 otherwise.
- Bit 35 (SIZEM): Set to 1 if the fault was caused by a size mismatch between PVVALIDATE or RMPADJUST and the RMP, 0 otherwise.
- Bit 36 (VMPL): Set to 1 if the fault was caused by a VMPL permission check failure, 0 otherwise.

The effective C-bit is always a 1 on any guest instruction fetch, page table access, or data write to private (C=1) memory.

All RMP checks described in this section occur after page table and nested page table access checks and have lower priority than existing paging checks. Table 15-39 reflects the relative priority of RMP checks. Namely, VMPL checks have the lowest priority, preceded by page validation checks. For example, if a guest access fails the Page-Size check and the Validated check, a #VMEXIT(NPF) will occur instead of a #VC since the Page-Size check has priority over the page validation check.

A failure of the page validation check results in a #VC with error code PAGE_NOT_VALIDATED (0x404). The faulting guest virtual address is saved to CR2 when this error occurs.

15.36.11 Large Page Management

The hypervisor may need to convert a 2MB page assigned to a guest into 4KB pages. This conversion is called page smashing and requires the hypervisor to alter the RMP. The hypervisor can use RMPUPDATE to alter the size of the page in the RMP, but this will clear the validated bit.

To convert a 2MB page into 4KB pages without altering the validated status of the region, the hypervisor may use the PSMASH instruction. PSMASH takes a 2MB aligned system physical address and smashes the page while preserving the Validated bit in the RMP. After PSMASH successfully completes, the RMP entries of the resulting 4KB pages have the following contents:
• Consecutive values in the Guest_Physical_Address fields
• Page_Size set to 0 indicating 4KB pages
• All other RMP fields copied from the original 2MB page RMP entry

One reason the hypervisor may need to smash a 2MB page is if the guest executes PVALIDATE or RMPADJUST on a 4KB page that is backed by a 2MB page. In that case, the instructions generate a #VMEXIT(NPF) with the SIZEM bit set in EXITINFO1. To resolve this, the hypervisor can smash the page and then have the guest restart the instruction.

If the guest wishes to validate a 2MB aligned region, the guest should first attempt to execute PVALIDATE with a size of 2MB. If the page is backed by 4KB pages, PVALIDATE terminates with a FAIL_SIZEMISMATCH error. In this case, the guest should then execute PVALIDATE on each 4KB page individually. This allows the guest to take advantage of the more efficient 2MB mappings and avoid having the hypervisor unnecessarily smash the page.

Table 15-40 summarizes the potential page size mismatches and how to resolve them.

<table>
<thead>
<tr>
<th>Requested Page Size</th>
<th>Page Size in RMP</th>
<th>Error Condition</th>
<th>Recommended Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB</td>
<td>2MB</td>
<td>#VMEXIT(NPF)</td>
<td>PSMASH</td>
</tr>
<tr>
<td>2MB</td>
<td>4KB</td>
<td>FAIL_SIZEMISMATCH</td>
<td>Guest retries on each 4KB constituent page</td>
</tr>
</tbody>
</table>

The reverse operation of converting a set of consecutive 4KB pages into a single 2MB page requires assistance from either the guest or the AMD-SP to ensure that the operation is safe to perform.

15.36.12 Running SNP-Active Virtual Machines

As with SEV-ES guests, SNP-active guests are described by a hypervisor controlled VMCB and a guest encrypted VMSA. The initial VMSA for an SNP-active guest must be set up through coordination with the AMD-SP, the details of which are beyond the scope of this manual. This includes the initial configuration of the SEV_FEATURES field in the VMSA which indicates which guest security features are enabled for that particular VM instance. VMRUN to an SNP-active guest will fail with a VMEXIT_INVALID error code if SEV-SNP is not globally enabled.

VMRUN Checks. When SEV-SNP is globally enabled on a system, the VMRUN instruction performs additional security checks on various memory pages. These checks are similar to the ones described in Section 15.36.10. Note that where a check depends on page size, a page size of 4KB is used. In addition to the checks described in that section, an additional check exists:
• VMSA: Checks that the VMSA field in the RMP entry equals 1.
The VMSA field in an RMP entry may be set by the AMD-SP, or by a vCPU running at VMPL0 using the RMPADJUST instruction.

The checks performed on VMRUN are as follows:

**Table 15-41. VMRUN Page Checks**

<table>
<thead>
<tr>
<th>Page Type</th>
<th>SNP-Active</th>
<th>Check</th>
<th>Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMCB</td>
<td>-</td>
<td>Hypervisor-Owned</td>
<td>#GP(0)</td>
</tr>
<tr>
<td>AVIC Backing Page</td>
<td>-</td>
<td>Hypervisor-Owned</td>
<td>#VMEXIT(VMEXIT_INVALID)</td>
</tr>
<tr>
<td>VMSA</td>
<td>No</td>
<td>Hypervisor-Owned</td>
<td>#VMEXIT(VMEXIT_INVALID)</td>
</tr>
<tr>
<td>VMSA</td>
<td>Yes</td>
<td>RMP-Covered Guest-Owned Reverse-Map Mutable VMSA</td>
<td>#VMEXIT(VMEXIT_INVALID)</td>
</tr>
</tbody>
</table>

The AVIC Logical Table, AVIC Physical Table, IOPM_BASE_PA, MSRPM_BASE_PA, and nCR3 are not checked by VMRUN as these structures are only read by the hardware.

After a successful VMRUN, the VMCB page, as well as any AVIC Backing Page and VMSA Page are marked as in-use by hardware, and any attempt to modify the RMP entries for these pages via instructions like RMPUPDATE will result in a FAIL_INUSE response. The in-use marking is automatically cleared by hardware after a #VMEXIT event.

**Other Checks.** In addition to the RMP checks performed by VMRUN, a few other VM-related operations perform special RMP checks.

The address written to the VM_HSAVE_PA MSR, which holds the address of the page used to save the host state on a VMRUN, must point to a hypervisor-owned page. If this check fails, the WRMSR will fail with a #GP(0) exception. Note that a value of 0 is not considered valid for the VM_HSAVE_PA MSR and a VMRUN that is attempted while the HSAVE_PA is 0 will fail with a #GP(0) exception.

The VMSAVE instruction also performs checks to ensure that the target page is hypervisor-owned. The VMSAVE instruction is not expected to be used with SEV-ES and SNP-active guests, as described in Section 15.36.8, but may be used with other guests.

If VMSAVE is executed in host mode and the target page fails the RMP check, a #GP(0) exception is generated. If VMSAVE is executed in a guest when the VMSAVE instruction is virtualized (see Section 15.33.1) and the target page fails, the RMP check then a #VMEXIT(NPF) is generated indicating an RMP permission error. In processors that support SEV-SNP, the execution of the VMSAVE instruction inside an SEV-ES or SNP-active guest is not supported and will result in a #VMEXIT(VMSAVE).
15.36.13 Debug Registers

SEV-ES and SNP-active guests may choose to enable full virtualization of CPU debug registers through SEV_FEATURES bit 5 (DebugSwap).

When enabled, the DR[0-3] registers and DR[0-3]_ADDR_MASK registers are swapped as type ‘B’ state (see Appendix B). Note that the DR6 and DR7 registers are always swapped as type ‘A’ state for any SEV guest.

15.36.14 Memory Types

When an SNP-active guest accesses memory, the hardware forces the use of coherent memory types. This prevents the hypervisor from attempting to corrupt guest memory by the use of non-coherent memory types for accesses by the guest.

If a guest memory access is determined to be non-coherent after the memory type determination logic described in Section 15.25.8, the hardware forces a coherent type as described in Table 15-42.

Table 15-42. Non-Coherent Memory Type Conversion

<table>
<thead>
<tr>
<th>Non-Coherent Memory Type</th>
<th>Forced Coherent Memory Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC</td>
<td>CD</td>
</tr>
<tr>
<td>WC</td>
<td>WC+</td>
</tr>
</tbody>
</table>

15.36.15 TLB management

For non-SNP-active guests, when a hypervisor moves a vCPU to a new logical processor it must ensure that the vCPU cannot use any stale (incorrect) TLB translations to prevent corruption of the guest. For SNP-active guests, to avoid any dependency on the hypervisor for correctly managing guest TLB contents, the hardware detects when the vCPU is moved and manages the TLB for that guest automatically. The hardware uses two VMSA fields to track this information: the TLB_ID (byte offset 3D0h) and the PCPU_ID (byte offset 3D8h).

During guest creation, software should initialize the TLB_ID and PCPU_ID by setting both to zero. The hardware subsequently manages the values in both fields throughout the lifetime of that vCPU. During operation, software may explicitly write PCPU_ID to 0 to force a TLB flush on the next VMRUN to that VMSA if desired. If this occurs, the hardware will set the PCPU_ID field to a non-zero value when it flushes the TLB.

For example, when guest software performs an RMPADJUST to alter the permissions of a VMPL, it may need to ensure that the existing TLB entries of all vCPUs executing at the targeted VMPL are not used anymore. The guest software can do this by writing zero to PCPU_ID of the affected vCPUs. When these vCPUs are re-entered with VMRUN, the hardware will ensure existing TLB entries are no longer used and set PCPU_ID to a non-zero value. This value can be checked for non-zero by the guest software to ensure the operation has completed before proceeding.
As with any guest, the hypervisor may use the TLB_CONTROL field in the VMCB to force TLB flushes when desired. When the hypervisor writes 3h or 7h to TLB_CONTROL, both global and non-global TLB entries of the guest are invalidated.

15.36.16 Interrupt Injection Restrictions

SNP-active guests may choose to enable the Restricted Injection or Alternate Injection features through SEV_FEATURES bits 3 and 4 respectively. These features enforce additional interrupt and event injection security protections designed to help protect against malicious injection attacks. The two are mutually exclusive for a specific vCPU and an attempt to enable both will result in a #VMEXIT(VMEXIT_INVALID) when the vCPU is run.

**Restricted Injection Operation.** This feature disables all hypervisor-based interrupt queuing and event injection of all vectors except a new exception vector, #HV (28), which is reserved for SNP guest use, but never generated by hardware. #HV is only allowed to be injected into vCPUs that execute with Restricted Injection. #HV is a benign exception and can only be injected as an exception (VMCB.EVENTINJ[Type]=3) and without an error code. Guests running with Restricted Injection are expected to communicate with the hypervisor about events via a software-managed para-virtualization interface. This interface can use #HV injection as a doorbell to inform the guest that new events have been added.

The VMRUN instruction with Restricted Injection enabled will fail with a VMEXIT_INVALID error code if the hypervisor attempts the injection of any unsupported event or attempts to run the guest with AVIC enabled.

**Alternate Injection Operation.** This feature replaces all hypervisor-based interrupt queuing and event injection with guest-controlled queuing and injection. When this is enabled on a vCPU, event injection information on VMRUN is read from the EventInjCtrl field in the VMSA (offset 3E0h) and interrupt queuing information is read from the VIntrCtrl field in the VMSA (offset 3B8h). This feature is intended to be used in a multi-VMPL architecture where a high privilege vCPU injects events and interrupts directly into a low privilege vCPU by writing to its encrypted VMSA.

When Alternate Injection is enabled, the EventInjCtrl field in the unencrypted VMCB (offset A8h) is ignored on VMRUN. The VIntrCtrl field in the unencrypted VMCB (offset 70h) is processed, but only the V_INTR_MASKING, Virtual GIF Mode, and AVIC Enable bits are used. The AVIC Enable bit must be 0 if the guest is running with Alternate Injection enabled, otherwise the VMRUN will fail with a VMEXIT_INVALID error code.

The remaining fields of VIntrCtrl (V_TPR, V_IRQ, VGIF, V_INTR_PRIO, V_IGN_TPR, V_INTR_VECTOR) are read exclusively from the encrypted version in the VMSA. Additionally, bit 10 of the encrypted VIntrCtrl field is defined as the INT_SHADOW bit and the unencrypted INT_SHADOW bit in VMCB offset 68h bit 0 is ignored. On a VMEXIT, the V_TPR, V_IRQ, and INT_SHADOW values are written back to the encrypted VIntrCtrl only.

In guests that run with Alternate Injection, bit 63 of the encrypted VIntrCtrl field is defined as a BUSY bit. On VMRUN, if VIntrCtrl[BUSY] is set to 1, then the VMRUN fails with a VMEXIT_BUSY error.
code. The BUSY bit enables a VMSA to be temporarily marked non-runnable while software modifications are in progress.

**Additional Intercept Behavior.** Additional hardware-forced intercept behavior exists in guests that run with either of these features enabled:

- For either feature, hardware treats physical INTR, NMI, INIT, and #MC events as intercepted regardless of the intercept bit set in the VMCB.

- Under Alternate Injection, any MSR access to the x2APIC MSR range (MSR 0x800-0x8FF) by the guest is intercepted regardless of the MSR_PROT intercept and MSR protection bitmap. In this case, the interception behavior is the same as what would occur if the MSR bitmap indicated an interception of the corresponding MSR.

### 15.36.17 Side-Channel Protection

SEV-SNP provides optional protections against certain side channel attacks.

**Branch Target Buffer Isolation**

SNP-active guests may choose to enable the Branch Target Buffer Isolation mode through SEV_FEATURES bit 7 (BTBIsolation). The Branch Target Buffer (BTB) is an internal CPU structure that is used when predicting indirect branches, and SNP-active guests may choose to impose additional restrictions on it in order to help prevent certain types of speculative execution-based side channels.

When executing an SNP-active guest when BTB Isolation is enabled, CPU hardware will ensure that no code outside of that guest context is able to influence the BTB-based predictions performed by hardware within the guest. Hardware tracks the source of prediction information in the BTB and may flush BTB contents when required to maintain this isolation.

In hardware that supports BTB Isolation, new BTB prediction information is never written if SPEC_CTL[IBRS] is enabled in the current context. Therefore, it is recommended that non-guest software that executes temporarily (e.g., hypervisor exit handling code) run with SPEC_CTL[IBRS] set to 1. This ensures that indirect branch information from that context is not stored in the BTB and may avoid the need for a BTB flush when guest execution is resumed.

**Instruction Based Sampling**

SEV-ES and SNP-active guests may choose to disallow the use of Instruction Based Sampling (IBS) by the hypervisor in order to limit the information that may be gathered about their execution. Guests may enable this restriction through SEV_FEATURES bit 6 (PreventHostIBS). When a VMRUN is executed on a guest that has enabled this protection, the IbsFetchCtl[IbsFetchEn] and IbsOpCtl[IbsOpEn] MSR bits must be 0. If either of these bits are not 0 then the VMRUN will fail with a VMEXIT_INVALID error code.
The Advanced Programmable Interrupt Controller (APIC) provides interrupt support on AMD64 architecture processors. The local APIC accepts interrupts from the system and delivers them to the local CPU core interrupt handler.

Support for APIC is indicated by CPUID Fn0000_0001_EDX[APIC] = 1. For information on using the CPUID instruction to obtain processor implementation information, see Section 3.3, “Processor Feature Identification,” on page 64.

The APIC block diagram is provided in Figure 16-1.

---

**Figure 16-1. Block Diagram of a Typical APIC Implementation**
16.1 Sources of Interrupts to the Local APIC

Each CPU core has an associated local APIC which receives interrupts from the following sources:

- I/O interrupts from the IOAPIC interrupt controller (including LINT0 and LINT1)
- Legacy interrupts (INTR and NMI) from the legacy interrupt controller
- Message Signalled Interrupts
- Interprocessor Interrupts (IPIs) from other local APICs. Interprocessor Interrupts are used to send interrupts or to execute system wide functions between CPU cores in the system, including the originating CPU core (self-interrupt).
- Locally generated interrupts within the local APIC. The local APIC receives local interrupts from the APIC timer, Performance Monitor Counters, thermal sensors, APIC errors and extended interrupts from implementation specific sources.

The sources of interrupts for the local APIC are provided in Table 16-1.

<table>
<thead>
<tr>
<th>Source</th>
<th>Description</th>
<th>Message Type to Local APIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O interrupts</td>
<td>System interrupts from I/O devices or system hardware received through the I/O APIC and sent to the local APIC as interrupt messages. They may be edge-triggered or level-sensitive.</td>
<td>Fixed, Lowest Priority, SMI, NMI, INIT, Restart, External interrupt, LINT0, LINT1</td>
</tr>
<tr>
<td>Legacy Interrupts</td>
<td>Legacy interrupts (INT and NMI) from the PIC and sent to the local APIC as interrupt messages.</td>
<td>NMI, INT</td>
</tr>
<tr>
<td>Interprocessor (IPI)</td>
<td>Interprocessor interrupts. Used for interrupt forwarding, system-wide functions, or software self-interrupts.</td>
<td>Fixed, lowest priority, SMI, read request, NMI, INIT, Restart, External interrupt</td>
</tr>
<tr>
<td>APIC Timer</td>
<td>Local interrupt from the programmed APIC timer reaches zero, under control of TIMER_LVT.</td>
<td>Fixed</td>
</tr>
<tr>
<td>Performance Monitor Counter</td>
<td>Local interrupt from the performance monitoring counter when it overflows, under control of PERF_CNT_LVT.</td>
<td>Fixed, SMI, or NMI</td>
</tr>
<tr>
<td>Thermal Sensor</td>
<td>Local interrupt from internal thermal sensors when it has tripped, under control of THERMAL_LVT.</td>
<td>Fixed, SMI, or NMI</td>
</tr>
<tr>
<td>Extended Interrupt[3:0]</td>
<td>Local Interrupts from programmable internal CPU core sources, under the control of the EXTENDED_INTERRUPT[3:0]_LVT.</td>
<td>Fixed, SMI, NMI, or External interrupt</td>
</tr>
<tr>
<td>APIC Internal Error</td>
<td>Local interrupt when an error is detected within the local APIC, under control of ERROR_LVT.</td>
<td>Fixed, SMI, or NMI</td>
</tr>
</tbody>
</table>
16.2 Interrupt Control

I/O, legacy, and interprocessor interrupts are sent via interrupt messages. The interrupt messages contain the following information:

- Destination address of the local APIC.
- VECTOR[7:0] indicating interrupt priority of up to 256 interrupt vectors. This information is captured in the IRR register for Fixed and Lowest Priority interrupt message types.
- Trigger Mode indicating edge triggered or level-sensitive (which requires an EOI response to the source).
- Message Type[3:0] indicating the type of interrupt to be presented to the local APIC. For Fixed and Lowest Priority message types, the interrupt is processed through the target local APIC. For all other message types, the interrupt is sent directly to the destination CPU core. There is a 5-line interrupt interface to the CPU core for INTR, SMI, NMI, INIT and STARTUP interrupts. For locally-generated interrupts, control is provided by local vector tables or LVTs. Separate LVTs are provided for each interrupt source, allowing for a unique entry point for each source. The LVT contains the VECTOR[7:0], trigger mode and message type as well as other fields associated with the specific interrupt. The message type may be Fixed, SMI, NMI, or External interrupt. A Mask bit is also provided to mask the interrupt.

16.3 Local APIC

16.3.1 Local APIC Enable

The local APIC is controlled by the APIC enable bit (AE) in the APIC Base Address Register (MSR 0000_001Bh). See Figure 16-2 on page 570.

When AE is set to 1, the local APIC is enabled and all interrupt types are accepted. When AE is cleared to 0, the local APIC is disabled, including all local vector table interrupts.

Software can disable the local APIC, using the APIC_SW_EN bit in the Spurious Interrupt Vector Register (APIC_F0). When this bit is cleared to zero, the local APIC is temporarily disabled:

- SMI, NMI, INIT, Startup, and Remote Read interrupts may be accepted.
- Pending interrupts in the ISR and IRR are held.
- Further fixed, lowest-priority, and ExtInt interrupts are not accepted.
- All LVT entry mask bits are set and cannot be cleared.
The fields within the APIC Base Address register are as follows:

- **Boot Strap CPU Core (BSC)**—Bit 8. The BSC bit indicates that this CPU core is the boot core of the BSP. Each CPU core that is not the boot core of the boot processor is an AP (Application Processor).

- **APIC Enable (AE)**—Bit 11. This is the APIC enable bit. The local APIC is enabled and all interruption types are accepted when AE is set to 1. Clearing AE to 0 disables the local APIC, and no local vector table interrupts are supported.

- **APIC Base Address (ABA)**—Bits 51:12. Specifies the base physical address for the APIC register set. The address is extended by 12 bits at the least-significant end to form the 52-bit physical base address. The reset value of the APIC base address is 0_0000_FEE0_0000h.

Note that a given processor may implement a physical address less than 52 bits in length.

16.3.2 APIC Registers

The system programming interface of the local APIC is made up of the registers listed in Table 16-2 below. All APIC registers are memory-mapped into the 4-Kbyte APIC register space, and are accessed with memory reads and writes. The memory address is indicated as:

\[
\text{APIC Register address} = \text{APIC Base Address + Offset}
\]

where the APIC Base Address must point to an uncacheable memory region, and is located in APIC Base Address Register, MSR 0000_001Bh. See Figure 16-2.

APIC registers are aligned to 16-byte offsets and must be accessed using naturally-aligned DWORD size read and writes. All other accesses cause undefined behavior.

The table includes the value of each register after reset.
### 16.3.3 Local APIC ID

Unique local APIC IDs are assigned to each CPU core in the system. The value is determined by hardware, based on the number of CPU cores on the processor and the node ID of the processor.

#### Table 16-2. APIC Registers

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>20h</td>
<td>APIC ID Register</td>
<td>??000000h</td>
</tr>
<tr>
<td>30h</td>
<td>APIC Version Register</td>
<td>80??0010h</td>
</tr>
<tr>
<td>80h</td>
<td>Task Priority Register (TPR)</td>
<td>00000000h</td>
</tr>
<tr>
<td>90h</td>
<td>Arbitration Priority Register (APR)</td>
<td>00000000h</td>
</tr>
<tr>
<td>A0h</td>
<td>Processor Priority Register (PPR)</td>
<td>00000000h</td>
</tr>
<tr>
<td>B0h</td>
<td>End of Interrupt Register (EOI)</td>
<td>–</td>
</tr>
<tr>
<td>C0h</td>
<td>Remote Read Register</td>
<td>00000000h</td>
</tr>
<tr>
<td>D0h</td>
<td>Logical Destination Register (LDR)</td>
<td>00000000h</td>
</tr>
<tr>
<td>E0h</td>
<td>Destination Format Register (DFR)</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>F0h</td>
<td>Spurious Interrupt Vector Register</td>
<td>000000FFh</td>
</tr>
<tr>
<td>100-170h</td>
<td>In-Service Register (ISR)</td>
<td>00000000h</td>
</tr>
<tr>
<td>180-1F0h</td>
<td>Trigger Mode Register (TMR)</td>
<td>00000000h</td>
</tr>
<tr>
<td>200-270h</td>
<td>Interrupt Request Register (IRR)</td>
<td>00000000h</td>
</tr>
<tr>
<td>280h</td>
<td>Error Status Register (ESR)</td>
<td>00000000h</td>
</tr>
<tr>
<td>300h</td>
<td>Interrupt Command Register Low (bits 31:0)</td>
<td>00000000h</td>
</tr>
<tr>
<td>310h</td>
<td>Interrupt Command Register High (bits 63:32)</td>
<td>00000000h</td>
</tr>
<tr>
<td>320h</td>
<td>Timer Local Vector Table Entry</td>
<td>00010000h</td>
</tr>
<tr>
<td>330h</td>
<td>Thermal Local Vector Table Entry</td>
<td>00010000h</td>
</tr>
<tr>
<td>340h</td>
<td>Performance Counter Local Vector Table Entry</td>
<td>00010000h</td>
</tr>
<tr>
<td>350h</td>
<td>Local Interrupt 0 Vector Table Entry</td>
<td>00010000h</td>
</tr>
<tr>
<td>360h</td>
<td>Local Interrupt 1 Vector Table Entry</td>
<td>00010000h</td>
</tr>
<tr>
<td>370h</td>
<td>Error Vector Table Entry</td>
<td>00010000h</td>
</tr>
<tr>
<td>380h</td>
<td>Timer Initial Count Register</td>
<td>00000000h</td>
</tr>
<tr>
<td>390h</td>
<td>Timer Current Count Register</td>
<td>00000000h</td>
</tr>
<tr>
<td>3E0h</td>
<td>Timer Divide Configuration Register</td>
<td>00000000h</td>
</tr>
<tr>
<td>400h</td>
<td>Extended APIC Feature Register</td>
<td>00040007h</td>
</tr>
<tr>
<td>410h</td>
<td>Extended APIC Control Register</td>
<td>00000000h</td>
</tr>
<tr>
<td>420h</td>
<td>Specific End of Interrupt Register (SEOI)</td>
<td>–</td>
</tr>
<tr>
<td>480-4F0h</td>
<td>Interrupt Enable Registers (IER)</td>
<td>FFFFFFFFh</td>
</tr>
<tr>
<td>500-530h</td>
<td>Extended Interrupt [3:0] Local Vector Table Registers</td>
<td>00000000h</td>
</tr>
</tbody>
</table>
The APIC ID is located in the APIC ID register at APIC offset 20h. See Figure 16-3. It is model dependent, whether software can modify the APIC ID Register. The initial value of the APIC ID (after a reset) is the value returned in CPUID function 0000_0001h_EBX[31:24].

![Figure 16-3. APIC ID Register (APIC Offset 20h)](image)

- **APIC ID (AID)**—Bits 31:24. The APIC ID field contains the unique APIC ID value assigned to this specific CPU core. A given implementation may use some bits to represent the CPU core and other bits represent the processor.

### 16.3.4 APIC Version Register

A version register is provided to allow software to identify which APIC version is used. Bits 7:0 of the APIC Version Register indicate the version number of the APIC implementation.

The number of entries in the local vector table are specified in bits 23:16 of the register as the maximum number minus one.

Bit 31 indicates the presence of extended APIC registers which have an offset starting at 400h.

![Figure 16-4. APIC Version Register (APIC Offset 30h)](image)

The fields within the APIC Version register are as follows:
• **Version (VER)**—Bits 7:0. The VER field indicates the version number of the APIC implementation. The local APIC implementation is identified with a value=1Xh (20h-FFh are reserved).

• **Max LVT Entries (MLE)**—Bits 23:16. The MLE field specifies the number of entries in the local vector table minus one.

• **Extended APIC Register Space Present (EAS)**—Bit 31. The EAS bit when set to 1 indicates the presence of an extended APIC register space, starting at offset 400h.

### 16.3.5 Extended APIC Feature Register

The Extended APIC Feature Register indicates the number of extended Local Vector Table registers in the local APIC, whether the Interrupt Enable Registers are present, and whether the 8-bit Extended APIC ID and Specific End Of Interrupt (SEOI) Register are supported.

![Figure 16-5. Extended APIC Feature Register (APIC Offset 400h)](image)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:24</td>
<td>Reserved</td>
<td>Reserved, Must be Zero</td>
<td>RO</td>
</tr>
<tr>
<td>23:16</td>
<td>XLC</td>
<td>Extended LVT Count</td>
<td>RO</td>
</tr>
<tr>
<td>15:3</td>
<td>Reserved</td>
<td>Reserved, Must be Zero</td>
<td>RO</td>
</tr>
<tr>
<td>2</td>
<td>XAIDC</td>
<td>Extended APIC ID Capable</td>
<td>RO</td>
</tr>
<tr>
<td>1</td>
<td>SNIC</td>
<td>Specific End of Interrupt Capable</td>
<td>RO</td>
</tr>
<tr>
<td>0</td>
<td>INC</td>
<td>Interrupt Enable Register Capable</td>
<td>RO</td>
</tr>
</tbody>
</table>

### 16.3.6 Extended APIC Control Register

This bit enables writes to the interrupt enable registers.
Extended APIC ID Enable (XAIDN)—Bit 2. Setting XAIDN to 1 enables the upper four bits of the APIC ID field described in “APIC ID Register (APIC Offset 20h)” on page 572. Clearing this bit, specifies a 4-bit APIC ID using only the lower four bits of the APIC ID field of the APIC ID register.

Enable SEOI Generation (SN)—Bit 1. Read-write. This bit enables Specific End of Interrupt (SEOI) generation when a write to the specific end of interrupt register is received.

Enable Interrupt Enable Registers (IERN)—Bit 0. This bit enables writes to the interrupt enable registers.

16.4 Local Interrupts

The local APIC handles the following local interrupts:

- APIC Timer
- Local Interrupt 0 (LINT0)
- Local Interrupt 1 (LINT1)
- Performance Monitor Counters
- Thermal Sensors
- APIC internal error
- Extended (Implementation dependent)

A separate entry in the local vector table is provided for each interrupt to allow software to specify:

- Whether the interrupt is masked or not.
- The delivery status of the interrupt.
- The message type.
- The unique address vector.
- For LINT0 and LINT1 interrupts, the trigger mode, remote IRR, and input pin polarity.
• For the APIC timer interrupt, the timer mode.

The general format of a Local Vector Table Register is shown in Figure 16-7.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:18</td>
<td>—</td>
<td>Reserved, MBZ</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>TMM</td>
<td>Timer Mode</td>
<td>R/W</td>
</tr>
<tr>
<td>16</td>
<td>M</td>
<td>Mask</td>
<td>R/W</td>
</tr>
<tr>
<td>15</td>
<td>TGM</td>
<td>Trigger Mode</td>
<td>R/W</td>
</tr>
<tr>
<td>14</td>
<td>RIR</td>
<td>Remote IRR</td>
<td>RO</td>
</tr>
<tr>
<td>13</td>
<td>—</td>
<td>Reserved, MBZ</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>DS</td>
<td>Delivery Status</td>
<td>RO</td>
</tr>
<tr>
<td>11</td>
<td>—</td>
<td>Reserved, MBZ</td>
<td></td>
</tr>
<tr>
<td>10:8</td>
<td>MT</td>
<td>Message Type</td>
<td>R/W</td>
</tr>
<tr>
<td>7:0</td>
<td>VEC</td>
<td>Vector</td>
<td>R/W</td>
</tr>
</tbody>
</table>

**Figure 16-7. General Local Vector Table Register Format**

The fields within the General Local Vector Table register are as follows:

• **Vector (VEC)**—Bits 7:0. The VEC field contains the vector that is sent for this interrupt source when the message type is fixed. It is ignored when the message type is NMI and is set to 00h when the message type is SMI. Valid values for the vector field are from 16 to 255. A value of 0 to 15 when the message type is fixed results in an illegal vector APIC error.

• **Message Type (MT)**—Bits 10:8. The MT field specifies the delivery mode sent to the CPU core interrupt handler. The legal values are:
  - 000b = Fixed - The vector field specifies the interrupt delivered.
  - 010b = SMI - An SMI interrupt is delivered. In this case, the vector field should be set to 00h.
  - 100b = NMI - A NMI interrupt is delivered with the vector field being ignored.
  - 111b = External interrupt is delivered.

• **Delivery Status (DS)**—Bit 12. The DS bit indicates the interrupt delivery status. The DS bit is set to 1 when the interrupt is pending at the CPU core interrupt handler. After a successful delivery of the interrupt, the associated bit in the IRR is set and this bit is cleared to zero. See Section 16.6.2, “Lowest Priority Messages and Arbitration,” on page 586 for details. The bit is cleared to 0 when the interrupt is idle.

• **Remote IRR (RIR)**—Bit 14. The RIR bit is set to 1 when the local APIC accepts an LINT0 or LINT1 interrupt with the trigger mode=1 (level sensitive). The bit is cleared to 0 when the interrupt completes, as indicated when an EOI is received.
• **Trigger Mode (TGM)**—Bit 15. Specifies how interrupts to the local APIC are triggered. The TGM bit is set to 1 when the interrupt is level-sensitive. It is cleared to 0 when the interrupt is edge-triggered. When the message type is SMI or NMI, the trigger mode is edge triggered.

• **Mask (M)**—Bit 16. When the M bit is set to 1, reception of the interrupt is disabled. When the M bit is cleared to 0, reception of the interrupt is enabled.

• **Timer Mode (TMM)**—Bit 17. Specifies the timer mode for the APIC Timer interrupt. The TMM bit set to 1 indicates periodic timer interrupts. The TMM bit cleared to 0 indicates one-shot operation.

### 16.4.1 APIC Timer Interrupt

The APIC timer is a programmable 32-bit counter used by software to time operations or events. The timer can operate in two modes, periodic and one-shot, under the control of bit 17 (Timer Mode) in APIC Timer Local Vector Table Register (see Figure 16-8). In one-shot mode, the APIC timer is set to a programmable initial value and decrements at a programmable clock rate. When the timer value reaches zero, an APIC timer interrupt is generated under the control of bit 16 (Mask) in the APIC Timer Local Vector Table Register. In periodic mode, the APIC timer is initialized again when it reaches zero, and it starts to decrement again. Another APIC timer interrupt is generated when the timer value reaches zero.

![Figure 16-8. APIC Timer Local Vector Table Register (APIC Offset 320h)](image)

Three APIC registers are defined for the APIC timer function:

• **Current Count Register (CCR)** is the actual APIC timer. It is initialized to a start count loaded from the ICR and then decrements. The APIC timer interrupt is generated when the CCR value reaches zero. The counting rate is controlled by the DCR. See Figure 16-9.

• **Initial Count Register (ICR)** contains the start count value for the APIC timer. See Table 16-10.

• **Divide Configuration Register (DCR)** controls the counting rate of the APIC timer by dividing the CPU core clock by a programmable amount. See Figure 16-11. For the specific details on the implementation of the APIC timer base clock rate, see the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product.
• **APIC Timer Current Count (APICTCC)**—Bits 31:0. The APICTCC field contains the current value of the APIC timer.

• **APIC Timer Initial Count (APICTIC)**—Bits 31:0. The APICTIC field contains the value that is loaded into the APIC Timer Current Count Register when the APIC timer is initialized.
• *Divide Value (DV)—* Bits 3, 1:0. The DV field specifies the value of the CPU core clock divisor. Table 16-3 lists the allowable values.

**Table 16-3. Divide Values**

<table>
<thead>
<tr>
<th>Bits 3, 1:0</th>
<th>Resulting Timer Divide</th>
</tr>
</thead>
<tbody>
<tr>
<td>000b</td>
<td>Divide by 2</td>
</tr>
<tr>
<td>001b</td>
<td>Divide by 4</td>
</tr>
<tr>
<td>010b</td>
<td>Divide by 8</td>
</tr>
<tr>
<td>011b</td>
<td>Divide by 16</td>
</tr>
<tr>
<td>100b</td>
<td>Divide by 32</td>
</tr>
<tr>
<td>101b</td>
<td>Divide by 64</td>
</tr>
<tr>
<td>110b</td>
<td>Divide by 128</td>
</tr>
<tr>
<td>111b</td>
<td>Divide by 1</td>
</tr>
</tbody>
</table>

**16.4.2 Local Interrupts LINT0 and LINT1**

When the target local APIC receives an interrupt message from an IOAPIC with the LINT0 or LINT1 message type, the appropriate local interrupt is generated under the control of bit 16 (Mask) in the APIC LINT0 or LINT1 Local Vector Table Register. See Figure 16-12.

**Figure 16-12. Local Interrupt 0/1 (LINT0/1) Local Vector Table Register (APIC Offset 350h/360h)**

In addition to the normal LVT control bits (mask, delivery status and vector offset), the LINT0/LINT1 interrupts provide the following controls:

• Trigger Mode - indicates whether the interrupt pin is edge triggered or level sensitive when the message type is fixed.

• Remote IRR - When the trigger mode indicates level, this flag is set when the local APIC accepts the interrupt, and is reset when the local APIC receives an EOI. When the flag is set, no additional local interrupt requests are sent to the local APIC, and they remain pending.

**16.4.3 Performance Monitor Counter Interrupts**

When a performance monitor counter overflows, an APIC interrupt is generated under the control of bit 16 (Mask) in the APIC Performance Monitor Counter Local Vector Table Register. See Figure 16-13 on page 579.
16.4.4 Thermal Sensor Interrupts

When a thermal event occurs, an APIC interrupt is generated under the control of bit 16 (Mask) in the APIC Thermal Sensor Local Vector Table Register. See Figure 16-14. See the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual applicable to your product for more information on thermal events. This interrupt may not be supported in all implementations.

16.4.5 Extended Interrupts

The local interrupts are extended to include more LVT registers, to allow additional interrupt sources. The additional sources are model dependent and can include:

- Counter overflow from the Machine Check Miscellaneous Threshold Register. See “Machine-Check Miscellaneous-Error Information Register 0(MCi_MISC0)” on page 282 for details.
- ECC Error Count Threshold in memory system.
- Instruction Sampling.

The LVT register used for each interrupt source is specified by the control register associated with the source.

The Extended LVT Count field (bits 23:16) of the Extended APIC Feature Register specifies the number of extended LVT registers. Currently there are four additional LVT registers defined, Extended Interrupt [3:0], Local Vector Table Register, located at APIC offsets 500h–530h. (See Section 16.7.1, “Specific End of Interrupt Register,” on page 593 and Figure 16-5 on page 573.)

16.4.6 APIC Error Interrupts

Errors that are detected while handling interrupts cause an APIC error interrupt to be generated under the control of bit 16 (Mask) in the APIC Error Local Vector Table Register. See Figure 16-15 on page 580.
The error information is recorded in the APIC Error Status Registers. The APIC Error Status Register is a read-write register. Writes to the register cause the internal error state to be recorded in the register, clearing the original error. See Figure 16-16.

The fields within the APIC Error Status register are as follows:

- **Sent Accept Error (SAE)**—Bit 2. The SAE bit when set to 1 indicates that a message sent by the local APIC was not accepted by any other APIC.
- **Receive Accept Error (RAE)**—Bit 3. The RAE bit when set to 1 indicates that a message received by the local APIC was not accepted by this or any other APIC.
- **Sent Illegal Vector (SIV)**—Bit 5. The SIV bit when set to 1 indicates that the local APIC attempted to send a message with an illegal vector value.
- **Receive Illegal Vector (RIV)**—Bit 6. The RIV bit when set to 1 indicates that the local APIC has received a message with an illegal vector value.
- **Illegal Register Address (IRA)**—Bit 7. The IRA bit when set to 1 indicates that an access to an unimplemented register location within the local APIC register range (APIC Base Address + 4 Kbytes) was attempted.
16.4.7 Spurious Interrupts

A timing issue exists between software and hardware that, though rare, results in spurious interrupts. In the event that the task priority is set to or above the level of the interrupt to be serviced while the interrupt is being acknowledged, the local APIC delivers a spurious interrupt to the CPU core instead, with the vector number specified by the Vector field of the Spurious Interrupt Register. The ISR is unaffected by the spurious interrupt, so the interrupt handler completes without sending an EOI back to the issuing local APIC.

![Figure 16-17. Spurious Interrupt Register (APIC Offset F0h)](image)

The fields within the Spurious Interrupt register are as follows:

- **Vector (VEC)**—Bits 7:0. The VEC field contains the vector that is sent to the CPU core in the event of a spurious interrupt.
- **APIC Software Enable (ASE)**—Bit 8. The ASE bit when set to 0 disables the local APIC temporarily. When the local APIC is disabled, SMI, NMI, INIT, Startup, and Remote Read may be accepted; pending interrupts in the ISR and IRR are held, but further fixed, lowest-priority, and ExtInt interrupts are not accepted. All LVT entry mask bits are set and cannot be cleared. Setting the ASE bit to 1, enables the local APIC.
- **Focus CPU Core Checking (FCC)**—Bit 9. The FCC bit when set to 1 disables focus CPU core checking when the lowest-priority message type is used. A CPU core is the focus of an interrupt if it is already servicing that interrupt (ISR=1) or if it has a pending request for that interrupt (IRR=1). Clearing the FCC bit to 0 disables focus CPU core checking.

16.5 Interprocessor Interrupts (IPI)

A local APIC can send interrupts to other local APICs (or itself) using software-initiated Interprocessor Interrupts (IPIs) using the Interrupt Command Register (ICR). Writing into the low order doubleword of the ICR causes the IPI to be sent.

The ICR can issue the following types of interrupt messages:
- basic interrupt message to another local APIC, including forwarding an interrupt that was received but not serviced
- basic interrupt message to the same local APIC (self-interrupt)
- system management interrupt (SMI)
- remote read message to another local APIC to read one of its APIC registers.
- non-maskable interrupt (NMI) delivered to another local APIC
- initialization message (INIT) to a target local APIC to be reset to their INIT state and await a STARTUP IPI.
- startup message (SIPI) to the target local APICs, pointing to a start-up routine.

The format of the Interrupt Command Register is shown in Figure 16-18.

![Figure 16-18. Interrupt Command Register (APIC Offset 300h–3010h)](image)

The fields within the Interrupt Command register are as follows:

- **Vector (VEC)**—Bits 7:0. The function of this field varies with the Message Type field. The VEC field contains the vector that is sent for this interrupt source for fixed and lowest priority message types.
- **Message Type (MT)**—Bits 10:8. The MT field specifies the message type sent to the CPU core interrupt handler. The legal values are:
  - 000b = Fixed - The IPI delivers an interrupt to the target local APIC specified in Destination field.
- 001b = Lowest Priority - The IPI delivers an interrupt to the local APIC executing at the lowest priority of all local APICs that match the destination logical ID specified in the Destination field. See Section 16.6.1, “Receiving System and IPI Interrupts,” on page 585.
- 010b = SMI - The IPI delivers an SMI interrupt to target local APIC(s). The trigger mode is edge-triggered and the Vector field must = 00h.
- 011b = Remote read - The IPI delivers a read request to read an APIC register in the target local APIC specified in Destination field. The trigger mode is edge triggered and the Vector field specifies the APIC offset of the APIC register to be read. The Remote Status field provides the current status of the remote read access after it has been issued. Data is returned from the target local APIC and captured in the Remote Read Register of the issuing local APIC. See Figure 16-19 on page 584.
- 100b = NMI - The IPI delivers a non-maskable interrupt to the target local APIC specified in the Destination field. The Vector field is ignored.
- 101b = INIT - The IPI delivers an INIT request to the target local APIC(s) specified in the Destination field, causing the CPU core to assume the INIT state. The trigger mode is edge-triggered, and the Vector field must =00h. In the INIT state, the target APIC is responsive only to the STARTUP IPI. All other interrupts (including SMI and NMI) are held pending until the STARTUP IPI has been accepted.
- 110b = STARTUP - The IPI delivers a start-up request (SIPI) to the target local APIC(s) specified in Destination field, causing the CPU core to start processing the platform firmware boot-strap routine whose address is specified by the Vector field.
- 111b = External interrupt - The IPI delivers an external interrupt to the target local APIC specified in Destination field. The interrupt can be delivered even if the APIC is disabled.

- **Destination Mode (DM)**—Bit 11. The DM bit when set to 1 specifies a logical destination which may be one or more local APICs with a common destination logical ID. When cleared to 0, the DM bit specifies a physical destination which indicates a single local APIC ID.

- **Delivery Status (DS)**—Bit 12. The DS bit indicates the interrupt delivery status. The DS bit is set to 1 when the local APIC has sent the IPI and is waiting for it to be accepted by another local APIC (the ICR is not idle). Clearing the DS bit indicates that the target local APIC is idle. Code may repeatedly write ICRL without polling the DS bit; all requested IPIs will be delivered.

- **Level (L)**—Bit 14. The L bit when set to 1 indicates assert. Clearing the L bit to 0 indicates deassert.

- **Trigger Mode (TGM)**—Bit 15. Specifies how IPIs to the local APIC are triggered. The TGM bit is set to 1 when the interrupt is level-sensitive. It is cleared to 0 when the interrupt is edge-triggered.

- **Remote Read Status (RRS)**—Bits 17:16. The RRS field indicates the current read status of a Remote Read from another local APIC. The encoding for this field is as follows:
- 00b = Read was invalid
- 01b = Delivery pending
- 10b = Delivery done and access was valid. Data available in Remote Read Register.
- 11b = Reserved

- **Destination Shorthand (DSH)**—Bits 19:18. The DSH field indicates whether a shorthand notation is used, and provides a quick way to specify a destination for a message. It replaces the Destination field, when the destination field is not required (DSH > 00b), allowing software to use a single write to the low order ICR. The encoding are as follows:
  - 00b = Destination - The Destination field is required to specify the destination.
  - 01b = Self - The issuing APIC is the only destination.
  - 10b = All including self - The IPI is sent to all local APICs including itself (destination field=FFh).
  - 11b = All excluding self - The IPI is sent to all local APICs except itself (destination field=FFh).

  Note that if the lowest priority is used, the message could end up being reflected back to this local APIC. If DS=1xb, the destination mode is ignored and physical is automatically used.

- **Destination (DES)**—Bits 63:56. The DES field identifies the target local APIC(s) for the IPI and contains the destination encoding used when the Destination Shorthand field=00b. The field indicates the target local APIC when the destination mode=0 (physical), and the destination logical ID (as indicated by LDR and DFR) when the destination mode=1 (logical).

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>RRD</td>
<td>Remote Read Data</td>
<td>RO</td>
</tr>
</tbody>
</table>

**Figure 16-19. Remote Read Register (APIC Offset C0h)**

- **Remote Read Data (RRD)**—Bits 31:0. The RRD field contains the data resulting from a valid completion of a remote read interprocessor interrupt.

Not all combinations of ICR fields are valid. Only the combinations indicated in Table 16-4 are valid.

**Table 16-4. Valid ICR Field Combinations**

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Trigger Mode</th>
<th>Level</th>
<th>Destination Shorthand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed</td>
<td>Edge</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>Level</td>
<td>Assert</td>
<td>x</td>
</tr>
<tr>
<td>Lowest Priority, SMI, NMI, INIT</td>
<td>Edge</td>
<td>x</td>
<td>Destination or all excluding self.</td>
</tr>
<tr>
<td></td>
<td>Level</td>
<td>Assert</td>
<td>Destination or all excluding self</td>
</tr>
<tr>
<td>Startup</td>
<td>x</td>
<td>x</td>
<td>Destination or all excluding self</td>
</tr>
</tbody>
</table>

**Note:** x indicates a don’t care.
16.6 Local APIC Handling of Interrupts

16.6.1 Receiving System and IPI Interrupts

Each local APIC verifies the destination ID, the destination mode and the message type of an APIC interrupt to determine if it is the target of the interrupt.

The destination mode is either physical or logical. In physical destination mode, the value of the interrupt message destination field is compared with the unique APIC ID value of each local APIC to select the target local APIC. If the destination field of the Interrupt Command Register is set to FFh, the interrupt is broadcasted and accepted by all local APICs. In physical destination mode, the lowest priority message type is not supported.

In logical destination mode, all local APICs use the Logical Destination Register and the Destination Format Register to determine if the interrupt is directed to them. The value of the interrupt message destination field is compared with the value in the Logical Destination Register (see Figure 16-20) of all local APICs.

The logical APIC ID must be unique. Since the comparison with the interrupt message destination field is on a bit-basis, there are only 8 unique logical IDs (01h, 02h, 04h, 08h, 10h, 20h, 40h, and 80h). For flat mode, the logical ID must be one of these values (for a total of eight local APICs supported). In cluster mode, the value of the logical ID is constrained to be \( xy \)h, where \( 0 \leq x \leq Eh \) and \( y = 1, 2, 4, \) or 8, for a total of \((15 \times 4)\) possible unique logical IDs.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:24</td>
<td>DLID</td>
<td>Destination Logical ID</td>
<td>R/W</td>
</tr>
<tr>
<td>23:0</td>
<td>—</td>
<td>Reserved, MBZ</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 16-20. Logical Destination Register (APIC Offset D0h)**

- *Destination Logical ID (DLID)—Bits 31:24.* The DLID field contains the logical APIC ID assigned to this specific CPU core. The logical APIC ID must be unique.

Two interrupt models are defined for the logical destination mode, the flat model and the cluster model, under the control of the Destination Format Register. See Figure 16-21.
• **Model (MOD)**—Bits 31:28. The MOD field controls which format to use when accepting interrupts in logical destination mode. The allowable values are 0h = cluster model and Fh = flat model.

With the flat model, up to eight unique logical APIC ID values can be provided by software by setting a different bit in the LDR. When the logical ID of the destination is compared with the LDR, if any bit position is set in both fields, this local APIC is a valid destination. A broadcast to all local APICs occurs when the LDR is set to all ones.

In the cluster model, bits 31:28 of the logical ID of the destination are compared with bits 31:28 of the LDR. If there is a match, then bits 27:24 are tested for matching ones, similar to the flat model. If bits 31:28 match, and any of bits 27:24 are set in both fields, this local APIC is a valid destination. The cluster model allows for 15 unique clusters to be defined, with each cluster having four unique logical APIC values to be addressed. In cluster logical destination mode, lowest priority message type is not supported.

In both the flat model and the cluster model, if the destination field = FFh, the interrupt is accepted by all local APICs.

### 16.6.2 Lowest Priority Messages and Arbitration

In the case where the interrupt is valid for several local APICs in logical destination mode with a lowest priority message type, the interrupt is accepted by the local APIC with the lowest arbitration priority, as indicated by the *Arbitration Priority* field in the Arbitration Priority Register (APR). The value in the *Arbitration Priority* field indicates the current priority for a pending interrupt or task, or an interrupt being serviced by the CPU core. See Figure 16-22.
The fields within the Arbitration Priority register are as follows:

- **Arbitration Priority Sub-class (APS)**—Bits 3:0. The APS field indicates the current sub-priority to handle arbitrated interrupts to be serviced by the CPU core.
- **Arbitration Priority (AP)**—Bits 7:4. The AP field indicates the current priority to handle arbitrated interrupts to be serviced by the CPU core. The priority is used to arbitrate between CPU cores to determine which core accepts a lowest-priority interrupt request.

The value in the Arbitration Priority field is equal to the highest priority of the Task Priority field of the Task Priority Register (TPR), the highest bit set in the In-Service Register (ISR) vector, or the highest bit set in the Interrupt Request Register (IRR) vector. The value in the Arbitration Priority Sub-class field is equal to the Task Priority Sub-class if the APR is equal to the TPR, and zero otherwise.

If focus CPU core checking is enabled (Spurious Interrupt Register bit 9=0), the focus CPU core for an interrupt can always accept the interrupt. A CPU core is the focus of an interrupt if it is already servicing that interrupt (corresponding ISR bit is set) or if it already has a pending request for that interrupt (corresponding IRR bit is set). If there is no focus CPU core for an interrupt or if focus CPU core checking is disabled (Spurious Interrupt Register bit 9=1), all target local APICs identified as candidates for the interrupt arbitrate to determine which is executing with the lowest arbitration priority. If there is a tie for lowest priority, the local APIC with the highest APIC ID is selected.

### 16.6.3 Accepting System and IPI Interrupts

If the local APIC accepting the interrupt determines that the message type for the interrupt request indicates SMI, NMI, INIT, STARTUP or ExtINT, it sends the interrupt directly to the CPU core for handling. If the message type is fixed or lowest priority, the accepting local APIC places the interrupt into an open slot in either the IRR or ISR registers. If there is no free slot, the interrupt is rejected and sent back to the sender with a retry request.

Three 256-bit acceptance registers support interrupts accepted by the local APIC. Bits 255:16 correspond to interrupt vectors 255:16 with 255 being the highest priority; bits 15:0 are reserved.

- **Interrupt Request Register (IRR)**, which contains interrupt requests that have been accepted but have not been sent to the CPU core for interrupt handling. When a system interrupt is accepted, the associated bit corresponding to the interrupt vector is set in the IRR. When the CPU core requests a
new interrupt, the local APIC selects the highest priority IRR interrupt and sends it to the CPU core. The local APIC then sets the corresponding bit in the ISR and resets the associated IRR bit. See Figure 16-23 on page 588.

- **In-Service Register (ISR)** contains the bit map of the interrupts that have been sent to the CPU core and are still being serviced. When the CPU core writes to the EOI register indicating completion of the interrupt processing, the associated ISR bit is reset and a new interrupt is selected from the IRR register. If a higher priority interrupt is accepted by the local APIC while the CPU core is servicing another interrupt, the higher priority interrupt is sent directly to the CPU core (before the current interrupt finishes processing) and the associated IRR bit is set. The CPU core interrupts the current interrupt handler to service the higher priority interrupt. When the interrupt handler for the higher priority interrupt completes, the associated IRR bit is reset and the interrupt handler returns to complete the previous interrupt handler routine. If a second interrupt with the same interrupt vector number is received by the local APIC while the ISR bit is set, the local APIC sets the IRR bit. No more than two interrupts can be pending for the same interrupt vector number. Subsequent interrupt requests to the same interrupt vector number will be rejected. See Figure 16-24 on page 589.

- **Trigger Mode Register (TMR)** indicates the trigger mode of the interrupt and determines whether an EOI message is sent to the I/O APIC for level-sensitive interrupts. When the interrupt is accepted by the local APIC and the IRR bit is set, the associated TMR bit is set for level-sensitive interrupts or reset for edge-triggered interrupts. At the end of the interrupt handler routine, when the EOI is received at the local APIC, an EOI message is sent to the I/O APIC if the associated TMR bit is set for a system interrupt. See Figure 16-25 on page 590.

<table>
<thead>
<tr>
<th>255</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td></td>
<td></td>
<td>Res, MBZ</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>255:16</td>
<td>IR</td>
<td>Interrupt Request bits</td>
<td>RO</td>
</tr>
<tr>
<td>15:0</td>
<td>—</td>
<td>Reserved, MBZ</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 16-23. Interrupt Request Register (APIC Offset 200h–270h)**
• **Interrupt Request bits (IR)**—Bits 255:16. The corresponding request bit is set when an interrupt is accepted by the local APIC. The interrupt request registers provide a bit per interrupt to indicate that the corresponding interrupt has been accepted by the local APIC. Interrupts are mapped as follows:

<table>
<thead>
<tr>
<th>Register</th>
<th>Interrupt Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRR (APIC offset 200h)</td>
<td>31–16</td>
</tr>
<tr>
<td>IRR (APIC offset 210h)</td>
<td>63–32</td>
</tr>
<tr>
<td>IRR (APIC offset 220h)</td>
<td>95–64</td>
</tr>
<tr>
<td>IRR (APIC offset 230h)</td>
<td>127–96</td>
</tr>
<tr>
<td>IRR (APIC offset 240h)</td>
<td>159–128</td>
</tr>
<tr>
<td>IRR (APIC offset 250h)</td>
<td>191–160</td>
</tr>
<tr>
<td>IRR (APIC offset 260h)</td>
<td>223–192</td>
</tr>
<tr>
<td>IRR (APIC offset 270h)</td>
<td>255–224</td>
</tr>
</tbody>
</table>

**Figure 16-24. In Service Register (APIC Offset 100h–170h)**

• **In Service bits (IS)**—Bits 255:16. These bits are set when the corresponding interrupt is being serviced by the CPU core. The in-service registers provide a bit per interrupt to indicate that the corresponding interrupt is being serviced by the CPU core. Interrupts are mapped as follows:

<table>
<thead>
<tr>
<th>Register</th>
<th>Interrupt Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISR (APIC offset 100h)</td>
<td>31–16</td>
</tr>
<tr>
<td>ISR (APIC offset 110h)</td>
<td>63–32</td>
</tr>
<tr>
<td>ISR (APIC offset 120h)</td>
<td>95–64</td>
</tr>
<tr>
<td>ISR (APIC offset 130h)</td>
<td>127–96</td>
</tr>
<tr>
<td>ISR (APIC offset 140h)</td>
<td>159–128</td>
</tr>
<tr>
<td>ISR (APIC offset 150h)</td>
<td>191–160</td>
</tr>
<tr>
<td>ISR (APIC offset 160h)</td>
<td>223–192</td>
</tr>
<tr>
<td>ISR (APIC offset 170h)</td>
<td>255–224</td>
</tr>
</tbody>
</table>
• Trigger Mode bits (TM)—Bits 255:16. These bits provide a bit per interrupt to indicate the assertion mode of each interrupt. Interrupts are mapped as follows:

<table>
<thead>
<tr>
<th>Register</th>
<th>Interrupt Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMR (APIC offset 180h)</td>
<td>31–16</td>
</tr>
<tr>
<td>TMR (APIC offset 190h)</td>
<td>63–32</td>
</tr>
<tr>
<td>TMR (APIC offset 1A0h)</td>
<td>95–64</td>
</tr>
<tr>
<td>TMR (APIC offset 1B0h)</td>
<td>127–96</td>
</tr>
<tr>
<td>TMR (APIC offset 1C0h)</td>
<td>159–128</td>
</tr>
<tr>
<td>TMR (APIC offset 1D0h)</td>
<td>191–160</td>
</tr>
<tr>
<td>TMR (APIC offset 1E0h)</td>
<td>223–192</td>
</tr>
<tr>
<td>TMR (APIC offset 1F0h)</td>
<td>255–224</td>
</tr>
</tbody>
</table>

16.6.4 Selecting and Handling Interrupts

Interrupts are selected by the local APIC for delivery to the CPU core interrupt handler on a priority determined by the interrupt vector number. Of the 15 priority levels, 15 is the highest and 1 is the lowest. The priority level for an interrupt is equal to the interrupt vector number divided by 16, rounded down to the nearest integer, with vectors 0Fh–00h reserved. Therefore, interrupt vectors 79h and 70h have the same priority level. The high-order hex digit indicates the priority level while the low-order hex digit indicates the priority within the same priority level.

Two registers are used to determine the priority threshold for selecting interrupts to be delivered to the CPU core, the Task Priority Register (TPR) and the Processor Priority Register (PPR). Software uses the TPR to set a priority threshold for interrupts to the CPU core, allowing the OS to block specific interrupts. See Figure 16-26 on page 591 for more details on the TPR.

The value in the Task Priority field is set by software to set a threshold priority at which the processor is to be interrupted. The value varies from 0 (all interrupts are allowed) to 15 (all interrupts with fixed delivery mode are inhibited). See Figure 16-26.
The fields within the Task Priority register are as follows:

- **Task Priority Sub-class (TPS)**—Bits 3:0. The TPS field indicates the current sub-priority to be used when arbitrating lowest-priority messages. This field is written with zero when TPR is written using the architectural CR8 register.

- **Task Priority (TP)**—Bits 7:4. The TP field indicates the current priority to be used when a core is deciding when to handle interrupts. A value of zero allows all interrupts; a value of Fh disables all interrupts. TP is also used to arbitrate between CPU cores to determine which core accepts a lowest-priority interrupt request. This field can also be written using the architectural CR8 register.

The PPR is set by the CPU core and represents the current priority level at which the CPU core is executing. The PPR determines whether a pending interrupt in the local APIC can be selected for interrupt handling in the CPU core. The value set by hardware is either the interrupt priority level of the highest priority ISR bit set or the value in the TPR, whichever is higher. The PPR is equal to the TPR when the CPU core is not servicing a higher priority interrupt. See Figure 16-27 on page 591.

### Figure 16-26. Task Priority Register (APIC Offset 80h)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:8</td>
<td>—</td>
<td>Reserved, Must be Zero</td>
<td></td>
</tr>
<tr>
<td>7:4</td>
<td>TP</td>
<td>Task Priority</td>
<td>R/W</td>
</tr>
<tr>
<td>3:0</td>
<td>TPS</td>
<td>Task Priority Sub-class</td>
<td>R/W</td>
</tr>
</tbody>
</table>

### Figure 16-27. Processor Priority Register (APIC Offset A0h)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:8</td>
<td>—</td>
<td>Reserved, MBZ</td>
<td></td>
</tr>
<tr>
<td>7:4</td>
<td>PP</td>
<td>Processor Priority</td>
<td>RO</td>
</tr>
<tr>
<td>3:0</td>
<td>PPS</td>
<td>Processor Priority Sub-class</td>
<td>RO</td>
</tr>
</tbody>
</table>

The fields within the Processor Priority register are as follows:

- **Processor Priority Sub-class (PPS)**—Bits 3:0. The PPS field is set to the Task Priority sub-class field of the Task Priority Register (TPR) if the PP field is equal to the Task Priority field of the TPR.

- **Processor Priority (PP)**—Bits 7:4. The PP field indicates the CPU core’s current priority for servicing a task or interrupt, and is used to determine if any pending interrupts should be serviced.
It is the higher value of either the interrupt priority level of the highest priority ISR bit set or the value in the TPR.

Pending interrupts must have a higher priority level than the value in the PPR to be selected by the local APIC for interrupt handling in the core; otherwise, they remain pending in the IRR until the PPR is lowered below the pending interrupt priority level. No pending interrupts are selected by the local APIC when the TPR=15.

The local APIC selects the highest priority pending interrupt (highest priority IRR) when the CPU core is ready, and sends the interrupt (with the IRR vector) to the CPU core. The local APIC resets the highest priority IRR bit and sets the associated ISR bit.

As part of the completion of the interrupt handling routine, software writes a value of zero to the End-of-Interrupt Register (EOI) in the local APIC, which causes the local APIC to reset the associated ISR bit. The EOI register is a write-only register.

If a higher priority interrupt is accepted by the local APIC while the CPU core is servicing another interrupt, the higher priority interrupt is sent directly to the CPU core (before the current interrupt finishes processing) and the associated ISR bit is set. The CPU core interrupts the current interrupt handler to service the higher priority interrupt. When the interrupt handler for the higher priority interrupt completes, the associated ISR bit is reset and the interrupt handler returns to complete the previous interrupt handler routine.

```
31 0
EOI
```

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:0</td>
<td>EOI</td>
<td>End of Interrupt</td>
<td>WO</td>
</tr>
</tbody>
</table>

**Figure 16-28.  End of Interrupt (APIC Offset B0h)**

- *End of Interrupt (EOI)—Bits 31:0. Write-only operation signals end of interrupt processing to source of interrupt.*

## 16.7 SVM Support for Interrupts and the Local APIC

The SVM hypervisor uses the Extended APIC Feature Register, Extended APIC Control Register, Specific End of Interrupt Register (SEOI), and Interrupt Enable Register (IER) to control virtualized interrupts. When guests have direct access to devices, interrupts arriving at the local APIC can usually be dismissed only by the guest that owns the device causing the interrupt. To prevent one guest from blocking other guests’ interrupts (by never processing their own), the VMM can mask pending interrupts in the local APIC, so they do not participate in the prioritization of other interrupts.
16.7.1 Specific End of Interrupt Register

Software issues a specific EOI (SEOI) by writing the vector number of the interrupt to the SIEOI register in the local APIC. The SIEOI register is located at offset 420h in the APIC space. The SIEOI register format is shown in Figure 16-29.

![Figure 16-29. Specific End of Interrupt (APIC Offset 420h)](image)

16.7.2 Interrupt Enable Register

The IER is made available to software by means of eight 32-bit registers in the local APIC; bit \( i \) of the 256-bit IER is located at bit position \((i \mod 32)\) in the local APIC register \( IER[i/32] \). The eight IER registers are located at offsets 480h, 490h, ...,4F0h in APIC space. The IER format is shown in Figure 16-30.

![Figure 16-30. Interrupt Enable Register (APIC Offset 480h–4F0h)](image)

- **Interrupt Enable (IE)**—Bits 255:16. Interrupts are mapped as follows:

<table>
<thead>
<tr>
<th>Register</th>
<th>Interrupt Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>IER (APIC offset 480h)</td>
<td>31–16</td>
</tr>
<tr>
<td>IER (APIC offset 490h)</td>
<td>63–32</td>
</tr>
<tr>
<td>IER (APIC offset 4A0h)</td>
<td>95–64</td>
</tr>
<tr>
<td>IER (APIC offset 4B0h)</td>
<td>127–96</td>
</tr>
<tr>
<td>IER (APIC offset 4C0h)</td>
<td>159–128</td>
</tr>
<tr>
<td>IER (APIC offset 4D0h)</td>
<td>191–160</td>
</tr>
</tbody>
</table>
The IER and SEOI registers are located in the APIC Extended Space area. The presence of the APIC Extended Space area is indicated by bit 31 of the APIC Version Register (at offset 30h in APIC space).

The presence of the IER and SEOI functionality is identified by bits 0 and 1, respectively, of the APIC Extended Feature Register (located at offset 400h in APIC space). IER and SEOI are enabled by setting bits 0 and 1, respectively, of the APIC Extended Control Register (located at offset 410h).

Only vectors that are enabled in IER participate in APIC's computation of the highest-priority pending interrupt. The reset value of IER is all ones.

<table>
<thead>
<tr>
<th>Register</th>
<th>Interrupt Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>IER (APIC offset 4E0h)</td>
<td>223–192</td>
</tr>
<tr>
<td>IER (APIC offset 4F0h)</td>
<td>255–224</td>
</tr>
</tbody>
</table>
17 Hardware Performance Monitoring and Control

The AMD64 architecture provides several mechanisms by which software can monitor and control processor performance to optimize power use. The following lists the facilities that are described in the sections that follow:

- The P-state control interface allows dynamic control of performance states. See Section 17.1 which follows immediately below.
- Core performance boost (CPB) dynamically increases core clock rate beyond that defined for the P0 power state to achieve higher performance while maintaining power consumption below a preset level. See Section 17.2 on page 597.
- The effective frequency interface provides a measure of the actual core clock rate over a specified period of time. See Section 17.3 on page 598.
- The processor power reporting interface allows system software to measure average processor core power over a given time period. See Section 17.5 on page 600.

17.1 P-State Control

P-states are operational performance states (states in which the processor is executing instructions, that is, running software) characterized by a unique frequency of operation for a CPU core. The P-state control interface supports dynamic P-state changes in up to 16 P-states called P-states 0 through 15 or P0 though P15. P0 is the highest power, highest performance P-state; each ascending P-state number represents a lower-power, lower-performance state.

Core P-states are controlled by software. Each CPU core contains one set of P-state control registers. Software controls the P-states of each CPU core independently; however, hardware may include interdependencies that affect the P-state achieved by each core.

Hardware provides the highest P-state value in the PstateMaxVal field of the P-State Current Limit Register. P-states may be limited to a lower performance value under certain conditions. The current P-state limit is dynamic and is specified in the CurPstateLimit field of the P-State Current Limit Register.

Software requests a core P-state change by writing a 4-bit index corresponding to the desired core P-state number to the P-State Control Register of the appropriate core. For example, to request the P3 state for core 0, software writes 3h to the core 0’s PstateCmd field in MSR C001_0062h. If the P-state value is greater than the value in PstateMaxVal, the value written is clipped to that value.

As the current P-state limit changes, the P-state for the CPU core is either set to the software-requested P-state value or the new current P-state limit, whichever is the higher P-state value.
The current P-state value can be read using the P-State Status Register. The P-State Current Limit Register and the P-State Status Register are read-only registers. Writes to these registers cause a #GP exception. Support for hardware P-state control is indicated by CPUID Fn8000_0007_EDX[HwPstate] = 1. Figure 17-1 below shows the format of the P-State Current Limit register.

**Figure 17-1. P-State Current Limit Register (MSR C001_0061h)**

The fields within the P-State Current Limit register are:

- **Current P-State Limit (CurPstateLimit)—**Bits 3:0. Provides the current P-state limit, which is the lowest P-state value (highest-performance state) that is currently supported by the hardware. This is a dynamic value controlled by hardware. Reset value is implementation specific.

- **P-State Maximum Value (PstateMaxVal)—**Bits 7:4. Specifies the highest P-state value (lowest performance state) supported by the hardware. Attempts to change the current P-state number to a higher value by writes to the P-State Control Register are clipped to the value of this field. Reset value is implementation specific.

**Figure 17-2. P-State Control Register (MSR C001_0062h)**

*P-State Change Command (PstateCmd)*—Bits 3:0. Writes to this field cause the CPU core to change to the indicated P-state number, which may be clipped by the PstateMaxVal field of the P-State Current Limit Register. Reset value is implementation specific.
Current P-State (CurPstate)—Bits 3:0. This field provides the current P-state of the CPU core regardless of the source of the P-state change, including writes to the P-State Control Register: 0 = P-state 0, 1 = P-state 1, etc. The value of this field is updated when the frequency transitions to a new value associated with the P-state. Reset value is implementation specific.

17.2 Core Performance Boost

Core performance boost (CPB) dynamically monitors processor activity to create an estimate of power consumption. If the estimated processor consumption is below an internally defined power limit and software has requested P0 on a given core, hardware may transition the core to a frequency and voltage beyond those defined for P0. If the estimated power consumption exceeds the defined power limit, some or all cores are limited to the frequency and voltage defined by P0. CPB ensures that average power consumption over a thermally significant time period remains at or below the defined power limit.

CPB can be disabled using the CPBDis field of the Hardware Configuration Register (HWCR MSR) on the appropriate core. When CPB is disabled, hardware limits the frequency and voltage of the core to those defined by P0.

Support for core performance boost is indicated by CPUID Fn8000_0007_EDX[CPB] = 1. See Section 3.3, “Processor Feature Identification,” on page 64 for more information on using the CPUID instruction.
17.3 Determining Processor Effective Frequency

The Max Performance Frequency Clock Count (MPERF) and the Actual Performance Frequency Clock Count (APERF) registers constitute the effective frequency interface. This interface provides a means for software to calculate an average, or effective, frequency of a core over a known window of time. This provides software a measure of actual performance rather than forcing software to assume that the current frequency of the core is the frequency of the last P-state requested.

To calculate an effective clock frequency of a given processor core, on that processor do the following:

1. Read both MPERF and APERF and save their initial values.
   - \[
   \text{MPERF\_INIT} = \text{MPERF} \quad \text{and} \quad \text{APERF\_INIT} = \text{APERF}
   \]
2. Wait an appropriate amount of time.
3. Read both MPERF and APERF again.
4. Effective frequency = \[
   \frac{(\text{APERF} - \text{APERF\_INIT})}{(\text{MPERF} - \text{MPERF\_INIT})} \times \text{P0 frequency}.
   \]

The amount of time that elapses between steps 1 and 3 is determined by software. This allows software to define the time window over which the processor frequency is averaged. Software should disable interrupts or any other events that may occur between the read of MPERF and the read of APERF in step 1 and again when the two MSRs are read in step 3. Step 4 provides the equation for the calculation of the effective frequency value. Software determines the P0 frequency using ACPI defined data structures.

The effective frequency interface only counts clock cycles while the core is in the ACPI defined C0 state.

Only the ratio between MPERF and APERF is architecturally defined. Software should not assume any specific definition of the MPERF or APERF registers. If an overflow of either the MPERF or
APERF register occurs between the read of MPERF in step 1 and the read of APERF in step 3, the effective frequency calculated in step 4 is invalid.

Hardware support for the effective frequency interface is indicated by CPUID Fn0000_0006_ECX[EffFreq]. See Section 3.3, “Processor Feature Identification,” on page 64 for more information on using the CPUID instruction.

### 17.3.1 Actual Performance Frequency Clock Count (APERF)

Specifies the numerator of the effective frequency ratio.

![Figure 17-5. Actual Performance Frequency Count (MSR0000_00E8h)](image)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Access Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:0</td>
<td>APERF</td>
<td>Actual Performance Frequency Count</td>
<td>R/W</td>
</tr>
</tbody>
</table>

### 17.3.2 Maximum Performance Frequency Clock Count (MPERF)

Specifies the denominator of the effective frequency ratio. The value read is scaled by the TSCRatio value (MSR C000_0104h) for guest reads, but the underlying counters are not affected. Reads in host mode or writes to MPERF are not affected.

![Figure 17-6. Max Performance Frequency Count (MSR0000_00E7h)](image)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Access Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:0</td>
<td>MPERF</td>
<td>Max Performance Frequency Count</td>
<td>R/W</td>
</tr>
</tbody>
</table>
17.3.3 MPERF Read-only (MperfReadOnly)

Read-only version of MPERF. The value read is scaled by the TSCRatio value (MSR C000_0104h) for guest reads.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Access Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:0</td>
<td>MPERF_RD_ONLY</td>
<td>MPERF Read Only</td>
<td>RO</td>
</tr>
</tbody>
</table>

Figure 17-7. MPERF Read Only (MSR C000_00E7h)

17.4 Processor Feedback Interface

The Processor Feedback Interface is deprecated. Some processor products may support this feature. To determine support on a given processor, software can test the feature bit CPUID Fn8000_0007_EDX[ProcFeedbackInterface]. For more information, consult the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual applicable to your product.

17.5 Processor Core Power Reporting

The processor power reporting interface allows system software to estimate the average power consumed by a processor core over a software-determined time period. Computing the average power involves reading a “core power accumulator” register at the beginning and end of the measurement interval, taking the difference and then dividing by the length of the time interval.

Support for the processor power reporting interface is indicated by CPUID Fn8000_0007_EDX[ProcPowerReporting] = 1.

17.5.1 Processor Facilities

Estimating core average power involves the use of several processor facilities. Processors that support the processor power reporting interface define the following three facilities:

- CpuSwPwrAcc MSR
- MaxCpuSwPwrAcc MSR
- CpuPwrSampleTimeRatio (CPUID Fn8000_0007 ECX)
A fourth facility, available on all processors, is the time-stamp counter (TSC). The TSC is a free-running counter that increments on every processor clock cycle. The current value of this counter is read using the RDTSC instruction.

The contents of the CpuSwPwrAcc register represents the cumulative energy consumed by the core. Each hardware-determined sample period (Tsample) a value that represents the energy consumed since the previous sample is added to the contents of this register. Tsample is on the order of a few microseconds. The exact value is immaterial because the CpuPwrSampleTimeRatio register provides the ratio of Tsample to the TSC period.

CpuSwPwrAcc is cleared to zero at power-on and is never reset. Therefore, it is possible for this counter to overflow and roll over to zero. To account for this, the interface provides the MaxCpuSwPwrAcc register. When read, this register provides a value that represents the maximum energy that the CpuSwPwrAcc register can report.

### 17.5.2 Software Algorithm

The following algorithm should be used to calculate the average power consumed by a processor core during the measurement interval $T_M$. To obtain a stable average power value, $T_M$ should be on the order of several milliseconds.

- Determine the value of the ratio of Tsample to the TSC period (CpuPwrSampleTimeRatio) by executing CPUID Fn8000_0007. Call this value $N$.
  \[ N = \text{CPUID Fn8000_0007_ECX}[31:0]. \]
- Read the full range of the cumulative energy value from the register MaxCpuSwPwrAcc.
  \[ J_{\text{max}} = \text{value returned from RDMSR MaxCpuSwPwrAcc}. \]
- At time $x$, read CpuSwPwrAcc and the TSC
  \[ J_x = \text{value returned by RDMSR CpuSwPwrAcc} \]
  \[ T_x = \text{value returned by RDTSC} \]
- At time $y$, read CpuSwPwrAcc and the TSC again
  \[ J_y = \text{value returned by RDMSR CpuSwPwrAcc} \]
  \[ T_y = \text{value returned by RDTSC} \]

Calculate the average power consumption for the processor core over the measurement interval $T_M = (T_y - T_x)$.

- If ($J_y < J_x$), rollover has occurred; set $J_{\text{delta}} = (J_y + J_{\text{max}}) - J_x$
  else $J_{\text{delta}} = J_y - J_x$
- $\text{PwrCPUave} = N \times J_{\text{delta}} / (T_y - T_x)$

Units of result is milliwatts.
Appendix A  MSR Cross-Reference

This appendix lists the MSRs that are defined in the AMD64 architecture. The AMD64 architecture supports some of the same MSRs as previous versions of the x86 architecture and implementations thereof. Where possible, the AMD64 architecture supports the same MSRs, for the same functions, as these previous architectures and implementations.

The first section lists the MSRs according to their MSR address, and it gives a cross reference for additional information. The remaining sections list the MSRs by their functional group. Those sections also give a brief description of the register and specify the register reset value.

Some MSRs are implementation-specific For information about these MSRs, see the documentation for specific implementations of the AMD64 architecture.

A.1 MSR Cross-Reference by MSR Address

Table A-1 lists the MSRs in the AMD64 architecture in order of MSR address.

Table A-1.  MSRs of the AMD64 Architecture

<table>
<thead>
<tr>
<th>MSR Address</th>
<th>MSR Name</th>
<th>Functional Group</th>
<th>Cross-Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010h</td>
<td>TSC</td>
<td>Performance</td>
<td>“Time-Stamp Counter” on page 377</td>
</tr>
<tr>
<td>001Bh</td>
<td>APIC_BASE</td>
<td>System Software</td>
<td>“Local APIC Enable” on page 569</td>
</tr>
<tr>
<td>00E7h</td>
<td>MPERF</td>
<td>Performance</td>
<td>“Determining Processor Effective Frequency” on page 598</td>
</tr>
<tr>
<td>00E8h</td>
<td>APERF</td>
<td>Performance</td>
<td>“Determining Processor Effective Frequency” on page 598</td>
</tr>
<tr>
<td>00FEh</td>
<td>MTRRcap</td>
<td>Memory Typing</td>
<td>“Identifying MTRR Features” on page 201</td>
</tr>
<tr>
<td>0174h</td>
<td>SYSENDER_CS</td>
<td>System Software</td>
<td>“SYSENDER and SYSEXIT MSRs” on page 160</td>
</tr>
<tr>
<td>0175h</td>
<td>SYSENDER_ESP</td>
<td>System Software</td>
<td></td>
</tr>
<tr>
<td>0176h</td>
<td>SYSENDER_EIP</td>
<td>System Software</td>
<td></td>
</tr>
<tr>
<td>0179h</td>
<td>MCG_CAP</td>
<td>Machine Check</td>
<td>“Machine-Check Global-Capabilities Register” on page 274</td>
</tr>
<tr>
<td>017Ah</td>
<td>MCG_STATUS</td>
<td></td>
<td>“Machine-Check Global-Status Register” on page 275</td>
</tr>
<tr>
<td>017Bh</td>
<td>MCG_CTL</td>
<td></td>
<td>“Machine-Check Global-Control Register” on page 276</td>
</tr>
<tr>
<td>01D9h</td>
<td>DebugCtl</td>
<td>Software Debug</td>
<td>“Debug-Control MSR (DebugCtl)” on page 361</td>
</tr>
</tbody>
</table>
### Table A-1. MSRs of the AMD64 Architecture (continued)

<table>
<thead>
<tr>
<th>MSR Address</th>
<th>MSR Name</th>
<th>Functional Group</th>
<th>Cross-Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>01DBh</td>
<td>LastBranchFromIP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01DCh</td>
<td>LastBranchToIP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01DDh</td>
<td>LastIntFromIP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01DEh</td>
<td>LastIntToIP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0200h</td>
<td>MTRRphysBase0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0201h</td>
<td>MTRRphysMask0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0202h</td>
<td>MTRRphysBase1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0203h</td>
<td>MTRRphysMask1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0204h</td>
<td>MTRRphysBase2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0205h</td>
<td>MTRRphysMask2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0206h</td>
<td>MTRRphysBase3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0207h</td>
<td>MTRRphysMask3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0208h</td>
<td>MTRRphysBase4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0209h</td>
<td>MTRRphysMask4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>020Ah</td>
<td>MTRRphysBase5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>020Bh</td>
<td>MTRRphysMask5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>020Ch</td>
<td>MTRRphysBase6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>020Dh</td>
<td>MTRRphysMask6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>020Eh</td>
<td>MTRRphysBase7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>020Fh</td>
<td>MTRRphysMask7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0250h</td>
<td>MTRRfix64K_00000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0258h</td>
<td>MTRRfix16K_80000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0259h</td>
<td>MTRRfix16K_A00000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0268h</td>
<td>MTRRfix4K_C00000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0269h</td>
<td>MTRRfix4K_C80000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>026Ah</td>
<td>MTRRfix4K_D00000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>026Bh</td>
<td>MTRRfix4K_D80000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>026Ch</td>
<td>MTRRfix4K_E00000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>026Dh</td>
<td>MTRRfix4K_E80000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>026Eh</td>
<td>MTRRfix4K_F00000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>026Fh</td>
<td>MTRRfix4K_F80000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0277h</td>
<td>PAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>02FFh</td>
<td>MTRRdefType</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Memory Typing**

- "Variable-Range MTRRs" on page 198
- "Fixed-Range MTRRs" on page 196
- "PAT Register" on page 204
- "Default-Range MTRRs" on page 200

**Software Debug**

- "Control-Transfer Recording MSRs" on page 363
Table A-1. MSRs of the AMD64 Architecture (continued)

<table>
<thead>
<tr>
<th>MSR Address</th>
<th>MSR Name</th>
<th>Functional Group</th>
<th>Cross-Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0400h</td>
<td>MC0_CTL</td>
<td></td>
<td>See the documentation for particular implementations of the architecture.</td>
</tr>
<tr>
<td>0404h</td>
<td>MC1_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0408h</td>
<td>MC2_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>040Ch</td>
<td>MC3_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0410h</td>
<td>MC4_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0414h</td>
<td>MC5_CTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0401h</td>
<td>MC0_STATUS</td>
<td>Machine Check</td>
<td>“Machine-Check Status Registers” on page 279</td>
</tr>
<tr>
<td>0405h</td>
<td>MC1_STATUS</td>
<td>Machine Check</td>
<td></td>
</tr>
<tr>
<td>0409h</td>
<td>MC2_STATUS</td>
<td>Machine Check</td>
<td></td>
</tr>
<tr>
<td>040Dh</td>
<td>MC3_STATUS</td>
<td>Machine Check</td>
<td></td>
</tr>
<tr>
<td>0411h</td>
<td>MC4_STATUS</td>
<td>Machine Check</td>
<td></td>
</tr>
<tr>
<td>0415h</td>
<td>MC5_STATUS</td>
<td>Machine Check</td>
<td></td>
</tr>
<tr>
<td>0402h</td>
<td>MC0_ADDR</td>
<td>Machine Check</td>
<td>“Machine-Check Address Registers” on page 282</td>
</tr>
<tr>
<td>0406h</td>
<td>MC1_ADDR</td>
<td>Machine Check</td>
<td></td>
</tr>
<tr>
<td>040Ah</td>
<td>MC2_ADDR</td>
<td>Machine Check</td>
<td></td>
</tr>
<tr>
<td>040Eh</td>
<td>MC3_ADDR</td>
<td>Machine Check</td>
<td></td>
</tr>
<tr>
<td>0412h</td>
<td>MC4_ADDR</td>
<td>Machine Check</td>
<td></td>
</tr>
<tr>
<td>0416h</td>
<td>MC5_ADDR</td>
<td>Machine Check</td>
<td></td>
</tr>
<tr>
<td>0403h</td>
<td>MC0_MISC</td>
<td>Machine Check</td>
<td>“Machine-Check Miscellaneous-Error Information Register 0(MCI_MISC0)” on page 282</td>
</tr>
<tr>
<td>0407h</td>
<td>MC1_MISC</td>
<td>Machine Check</td>
<td></td>
</tr>
<tr>
<td>040Bh</td>
<td>MC2_MISC</td>
<td>Machine Check</td>
<td></td>
</tr>
<tr>
<td>040Fh</td>
<td>MC3_MISC</td>
<td>Machine Check</td>
<td></td>
</tr>
<tr>
<td>0413h</td>
<td>MC4_MISC</td>
<td>Machine Check</td>
<td></td>
</tr>
<tr>
<td>0417h</td>
<td>MC5_MISC</td>
<td>Machine Check</td>
<td></td>
</tr>
<tr>
<td>C000_0080h</td>
<td>EFER</td>
<td>System Software</td>
<td>“Extended Feature Enable Register (EFER)” on page 55</td>
</tr>
<tr>
<td>C000_0081h</td>
<td>STAR</td>
<td>System Software</td>
<td></td>
</tr>
<tr>
<td>C000_0082h</td>
<td>LSTAR</td>
<td>System Software</td>
<td>“SYSCALL and SYSRET MSRs” on page 159</td>
</tr>
<tr>
<td>C000_0083h</td>
<td>CSTAR</td>
<td>System Software</td>
<td></td>
</tr>
<tr>
<td>C000_0084h</td>
<td>SF_MASK</td>
<td>System Software</td>
<td></td>
</tr>
<tr>
<td>C000_0100h</td>
<td>FS.Base</td>
<td>System Software</td>
<td>“FS and GS Registers in 64-Bit Mode” on page 74</td>
</tr>
<tr>
<td>C000_0101h</td>
<td>GS.Base</td>
<td>System Software</td>
<td></td>
</tr>
<tr>
<td>C000_0102h</td>
<td>KernelGSbase</td>
<td>System Software</td>
<td>“SWAPGS Instruction” on page 161</td>
</tr>
<tr>
<td>C000_0103h</td>
<td>TSC_AUX</td>
<td>System Software</td>
<td>“RDTSCP Instruction” on page 163</td>
</tr>
</tbody>
</table>

MSR Cross-Reference
Table A-1. MSRs of the AMD64 Architecture (continued)

<table>
<thead>
<tr>
<th>MSR Address</th>
<th>MSR Name</th>
<th>Functional Group</th>
<th>Cross-Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>C000_0408h</td>
<td>MC4_MISC1</td>
<td>Machine Check</td>
<td>“Machine-Check Miscellaneous-Error Information Register 0(MC4_MISC0)” on page 282</td>
</tr>
<tr>
<td>C000_0409h</td>
<td>MC4_MISC2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C000_040Ah</td>
<td>MC4_MISC3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C001_0000h</td>
<td>PerfEvtSel0</td>
<td>Performance</td>
<td>“Core Performance Event-Select Registers” on page 372</td>
</tr>
<tr>
<td>C001_0001h</td>
<td>PerfEvtSel1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C001_0002h</td>
<td>PerfEvtSel2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C001_0003h</td>
<td>PerfEvtSel3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C001_0004h</td>
<td>PerfCtr0</td>
<td>Performance</td>
<td>“Performance Counter MSRs” on page 371</td>
</tr>
<tr>
<td>C001_0005h</td>
<td>PerfCtr1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C001_0006h</td>
<td>PerfCtr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C001_0007h</td>
<td>PerfCtr3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C001_0010h</td>
<td>SYSCFG</td>
<td>Memory Typing</td>
<td>“System Configuration Register (SYSCFG)” on page 59</td>
</tr>
<tr>
<td>C001_0016h</td>
<td>IORRBase0</td>
<td>Memory Typing</td>
<td>“IORRs” on page 210</td>
</tr>
<tr>
<td>C001_0017h</td>
<td>IORRMask0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C001_0018h</td>
<td>IORRBase1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C001_0019h</td>
<td>IORRMask1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C001_001Ah</td>
<td>TOP_MEM</td>
<td>Memory Typing</td>
<td>“Top of Memory” on page 212</td>
</tr>
<tr>
<td>C001_001Dh</td>
<td>TOP_MEM2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C001_0030h</td>
<td>Processor_Name_String</td>
<td>CPUID Name</td>
<td>See appropriate BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual for details.</td>
</tr>
<tr>
<td>C001_0031h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C001_0032h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C001_0033h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C001_0034h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C001_0035h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C001_0061h</td>
<td>P-State Current Limit</td>
<td>SMM</td>
<td>“Hardware Performance Monitoring and Control” on page 595</td>
</tr>
<tr>
<td>C001_0062h</td>
<td>P-State Control</td>
<td>SMM</td>
<td></td>
</tr>
<tr>
<td>C001_0063h</td>
<td>P-State Status</td>
<td>SMM</td>
<td></td>
</tr>
<tr>
<td>C001_0074h</td>
<td>CPU_Watchdog_Timer</td>
<td>Machine Check</td>
<td>“CPU Watchdog Timer Register” on page 276</td>
</tr>
<tr>
<td>C001_0104h</td>
<td>TSC Ratio</td>
<td>SVM</td>
<td>“TSC Ratio MSR (C000_0104h)” on page 536</td>
</tr>
<tr>
<td>C001_0111h</td>
<td>SMBASE</td>
<td>SMM</td>
<td>“SMBASE Register” on page 293</td>
</tr>
<tr>
<td>C001_0112h</td>
<td>SMM_ADDR</td>
<td>SMM</td>
<td>“SMRAM Protected Areas” on page 299</td>
</tr>
<tr>
<td>C001_0113h</td>
<td>SMM_MASK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table A-1. MSRs of the AMD64 Architecture (continued)

<table>
<thead>
<tr>
<th>MSR Address</th>
<th>MSR Name</th>
<th>Functional Group</th>
<th>Cross-Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>C001_0114h</td>
<td>VM_CR</td>
<td>SVM</td>
<td>“SVM Related MSRs” on page 534</td>
</tr>
<tr>
<td>C001_0115h</td>
<td>IGNNE</td>
<td>SVM</td>
<td>“SVM Related MSRs” on page 534</td>
</tr>
<tr>
<td>C001_0116h</td>
<td>SMM_CTL</td>
<td>SVM</td>
<td>“SVM Related MSRs” on page 534</td>
</tr>
<tr>
<td>C001_0117h</td>
<td>VM_HSAVE_PA</td>
<td>SVM</td>
<td>“SVM Related MSRs” on page 534</td>
</tr>
<tr>
<td>C001_0118h</td>
<td>SVM_KEY_MSR</td>
<td>SVM</td>
<td>“SVM-Lock” on page 537</td>
</tr>
<tr>
<td>C001_0119h</td>
<td>SMM_KEY_MSR</td>
<td>SMM</td>
<td>“SMM-Lock” on page 538</td>
</tr>
<tr>
<td>C001_011Bh</td>
<td>Doorbell Register</td>
<td>SVM</td>
<td>“Doorbell Register” on page 531</td>
</tr>
<tr>
<td>C001_011E</td>
<td>VMPAGE_FLUSH</td>
<td>SVM</td>
<td>“Secure Encrypted Virtualization” on page 540</td>
</tr>
<tr>
<td>C001_0130</td>
<td>GHCB</td>
<td>SVM</td>
<td>“Guest-HV Communication Block&quot; (see “GHCB” on page 551)</td>
</tr>
<tr>
<td>C001_0131</td>
<td>SEV_STATUS</td>
<td>SVM</td>
<td>“SEV_STATUS MSR” (see “SEV_STATUS MSR” on page 545)</td>
</tr>
<tr>
<td>C001_0132</td>
<td>RMP_BASE</td>
<td>SVM</td>
<td>”Initializing the RMP“ (see ”Initializing the RMP“ on page 555)</td>
</tr>
<tr>
<td>C001_0133</td>
<td>RMP_END</td>
<td>SVM</td>
<td>”Initializing the RMP“ (see ”Initializing the RMP“ on page 555)</td>
</tr>
<tr>
<td>C001_0140h</td>
<td>OSVW_ID_Length</td>
<td>OSVW</td>
<td>“OS-Visible Workarounds” on page 641</td>
</tr>
<tr>
<td>C001_0141h</td>
<td>OSVW Status</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C000_1019h</td>
<td>DR1_ADDR_MASK</td>
<td>Software Debug</td>
<td>“Debug Breakpoint Address Masking” on page 370</td>
</tr>
<tr>
<td>C000_101Ah</td>
<td>DR2_ADDR_MASK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C000_101Bh</td>
<td>DR3_ADDR_MASK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C000_1027h</td>
<td>DR0_ADDR_MASK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### A.2 System-Software MSRs

Table A-2 lists the MSRs defined for general use by system software in controlling long mode and in allowing fast control transfers between applications and the operating system.
## Table A-2. System-Software MSR Cross-Reference

<table>
<thead>
<tr>
<th>MSR Address</th>
<th>MSR Name</th>
<th>Description</th>
<th>Reset Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000_001Bh</td>
<td>APIC_BASE</td>
<td>See appropriate BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual for details.</td>
<td>0000_0000_FEE0_0x00h</td>
</tr>
<tr>
<td>C000_0080h</td>
<td>EFER</td>
<td>Contains control bits that enable extended features supported by the processor, including long mode.</td>
<td>0000_0000_0000_0000h</td>
</tr>
<tr>
<td>C000_0081h</td>
<td>STAR</td>
<td>In legacy mode, used to specify the target address of a SYSCALL instruction, as well as the CS and SS selectors of the called and returned procedures.</td>
<td>undefined</td>
</tr>
<tr>
<td>C000_0082h</td>
<td>LSTAR</td>
<td>In 64-bit mode, used to specify the target RIP of a SYSCALL instruction.</td>
<td>undefined</td>
</tr>
<tr>
<td>C000_0083h</td>
<td>CSTAR</td>
<td>In compatibility mode, used to specify the target RIP of a SYSCALL instruction.</td>
<td>undefined</td>
</tr>
<tr>
<td>C000_0084h</td>
<td>SF_MASK</td>
<td>SYSCALL Flags Mask</td>
<td>undefined</td>
</tr>
<tr>
<td>C000_0100h</td>
<td>FS.Base</td>
<td>Contains the 64-bit base address in the hidden portion of the FS register (the base address from the FS descriptor).</td>
<td>0000_0000_0000_0000h</td>
</tr>
<tr>
<td>C000_0101h</td>
<td>GS.Base</td>
<td>Contains the 64-bit base address in the hidden portion of the GS register (the base address from the GS descriptor).</td>
<td>0000_0000_0000_0000h</td>
</tr>
<tr>
<td>C000_0102h</td>
<td>KernelGSbase</td>
<td>The SWAPGS instruction exchanges the value in KernelGSbase with the value in GS.base, providing a fast method for system software to load a pointer to system data-structures.</td>
<td>undefined</td>
</tr>
<tr>
<td>C000_0103h</td>
<td>TSC_AUX</td>
<td>The RDTSCP instruction copies the value of this MSR into the ECX register.</td>
<td>0000_0000_0000_0000h</td>
</tr>
<tr>
<td>C000_0104h</td>
<td>TSC_RATIO</td>
<td>Specifies the TSCRatio value which is used to scale the TSC value read by a Guest.</td>
<td>0000_0001_0000_0000h</td>
</tr>
<tr>
<td>0174h</td>
<td>SYSENTER_CS</td>
<td>In legacy mode, used to specify the CS selector of the procedure called by SYSENTER.</td>
<td>undefined</td>
</tr>
<tr>
<td>0175h</td>
<td>SYSENTER_ESP</td>
<td>In legacy mode, used to specify the stack pointer for the procedure called by SYSENTER.</td>
<td>undefined</td>
</tr>
<tr>
<td>0176h</td>
<td>SYSENTER_EIP</td>
<td>In legacy mode, used to specify the EIP of the procedure called by SYSENTER.</td>
<td>undefined</td>
</tr>
</tbody>
</table>