# AMD64 Virtualization Technology

# Secure Virtual Machine Architecture Reference Manual


| Publication No. | Revision | Date |
| --- | --- | --- |
| **33047** | **3.02** | **December 2005** |


*Advanced Micro Devices*

© 2005 Advanced Micro Devices, Inc. All rights reserved.

The contents of this document are provided in connection with Advanced Micro Devices, Inc. ("AMD") products. AMD makes no representations or warranties with respect to the accuracy or completeness of the contents of this publication and reserves the right to make changes to specifications and product descriptions at any time without notice. No license, whether express, implied, arising by estoppel or otherwise, to any intellectual property rights is granted by this publication. Except as set forth in AMD's Standard Terms and Conditions of Sale, AMD assumes no liability whatsoever, and disclaims any express or implied warranty, relating to its products including, but not limited to, the implied warranty of merchantability, fitness for a particular purpose, or infringement of any intellectual property right.

AMD's products are not designed, intended, authorized or warranted for use as components in systems intended for surgical implant into the body, or in other applications intended to support or sustain life, or in any other application in which the failure of AMD's product could create a situation where personal injury, death, or severe property or environmental damage may occur. AMD reserves the right to discontinue or make changes to its products at any time without notice.

### Trademarks

AMD, the AMD Arrow logo, AMD Athlon, AMD Opteron, and combinations thereof, are trademarks, and AMD-K6 is a registered trademark of Advanced Micro Devices, Inc.

HyperTransport is a licensed trademark of the HyperTransport Technology Consortium.

Pentium is a registered trademark of Intel Corporation.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

# Contents

**Revision History** . . . . . . . . . . xiii

**Preface** . . . . . . . . . . xv

**1 Introduction** . . . . . . . . . . 1

1.1 The Virtual Machine Monitor . . . . . . . . . . 1
1.2 SVM Hardware Overview . . . . . . . . . . 1
Virtualization Support . . . . . . . . . . 1
Guest Mode . . . . . . . . . . 2
External Access Protection . . . . . . . . . . 2
Tagged TLB . . . . . . . . . . 2
Interrupt Support . . . . . . . . . . 2
Intercepting physical interrupt delivery . . . . . . . . . . 2
Virtual interrupts . . . . . . . . . . 2
Sharing a physical APIC . . . . . . . . . . 2
Restartable Instructions . . . . . . . . . . 2
Security Support . . . . . . . . . . 2
Attestation. . . . . . . . . . . 3
Memory Clear. . . . . . . . . . . 3

**2 SVM Processor and Platform Extensions** . . . . . . . . . . 5

2.1 Enabling SVM . . . . . . . . . . 5
2.2 VMRUN Instruction . . . . . . . . . . 5
Basic Operation . . . . . . . . . . 6
Saving Host State . . . . . . . . . . 7
Loading Guest State . . . . . . . . . . 7
Control Bits . . . . . . . . . . 8
Segment State in the VMCB . . . . . . . . . . 9
Canonicalization and Consistency Checks . . . . . . . . . . 10
VMRUN and TF/RF bits in EFLAGS . . . . . . . . . . 11
2.3 #VMEXIT . . . . . . . . . . 12
2.4 Intercept Operation . . . . . . . . . . 13
Exception intercepts . . . . . . . . . . 13
Instruction intercepts . . . . . . . . . . 14
State Saved on Exit . . . . . . . . . . 14
Intercepts During IDT Interrupt Delivery . . . . . . . . . . 14
EXITINTINFO Pseudo-Code . . . . . . . . . . 16
2.5 Instruction Intercepts . . . . . . . . . . 17
Read/Write of CR0 . . . . . . . . . . 17
Read/Write of CR3 (excluding task switch) . . . . . . . . . . 17
Read/Write of other CRs . . . . . . . . . . 17
Read/Write of Debug Registers, DRn . . . . . . . . . . 17
Selective CR0 Write Intercept . . . . . . . . . . 18
Reading/Writing of IDTR, GDTR, LDTR, TR . . . . . . . . . . 18

RDTSC Instruction Intercept . . . . . 18
RDPMC Instruction Intercept . . . . . 18
PUSHF Instruction Intercept . . . . . 18
POPF Instruction Intercept . . . . . 18
CPUID Instruction Intercept . . . . . 18
RSM Instruction Intercept . . . . . 19
IRET Instruction Intercept . . . . . 19
Software Interrupt Intercept . . . . . 19
INVD Instruction Intercept . . . . . 19
PAUSE Instruction Intercept . . . . . 19
HLT Instruction Intercept . . . . . 19
INVLPG Instruction Intercept . . . . . 19
INVLPGA Instruction Intercept . . . . . 19
VMRUN Instruction Intercept . . . . . 19
VMLOAD Instruction Intercept . . . . . 19
VMSAVE Instruction Intercept . . . . . 20
VMMCALL Instruction Intercept . . . . . 20
STGI Instruction Intercept . . . . . 20
CLGI Instruction Intercept . . . . . 20
SKINIT Instruction Intercept . . . . . 20
RDTSCP Instruction Intercept . . . . . 20
ICEBP Instruction Intercept . . . . . 20
WBINVD Instruction Intercept . . . . . 20
2.6 IOIO Intercepts . . . . . 20
I/O Permissions Map . . . . . 20
IN and OUT Behavior . . . . . 21
I/O Intercept Information . . . . . 21
2.7 MSR Intercepts . . . . . 22
MSR Permissions Map . . . . . 22
RDMSR and WRMSR Behavior . . . . . 23
MSR Intercept Information . . . . . 23
2.8 Exception Intercepts . . . . . 23
Example: . . . . . 24
#DE (Divide By Zero) . . . . . 24
#DB (Debug) . . . . . 24
Vector 2 (Reserved) . . . . . 25
#BP (Breakpoint) . . . . . 25
#OF (Overflow) . . . . . 25
#BR (Bound-Range) . . . . . 25
#UD (Invalid Opcode) . . . . . 25
#NM (Device-Not-Available) . . . . . 25
#DF (Double Fault) . . . . . 25
Vector 9 (Reserved) . . . . . 25
#TS (Invalid TSS) . . . . . 25
#NP (Segment Not Present) . . . . . 25
#SS (Stack Fault) . . . . . 25
#GP (General Protection) . . . . . 26

#PF (Page Fault) . . . . . 26
#MF (X87 Floating Point) . . . . . 26
#AC (Alignment Check) . . . . . 26
#MC (Machine Check) . . . . . 26
#XF (SIMD Floating Point) . . . . . 26
2.9 Interrupt Intercepts . . . . . 26
INTR Intercept . . . . . 27
NMI Intercept . . . . . 27
SMI Intercept . . . . . 27
INIT Intercept . . . . . 27
Virtual Interrupt Intercept . . . . . 27
2.10 Miscellaneous Intercepts . . . . . 27
Task Switch Intercept . . . . . 27
Ferr_Freeze Intercept . . . . . 28
Shutdown Intercept . . . . . 28
2.11 VMSAVE and VMLOAD Instructions . . . . . 28
2.12 TLB Control . . . . . 29
Software Rule . . . . . 29
TLB Flush . . . . . 30
Invalidate Page, Alternate ASID . . . . . 30
2.13 Global Interrupt Flag, STGI and CLGI Instructions . . . . . 30
2.14 VMMCALL Instruction . . . . . 32
2.15 Paged Real Mode . . . . . 32
2.16 Event Injection . . . . . 32
2.17 Interrupt and localAPIC Support . . . . . 34
Physical (INTR) Interrupt Masking in EFLAGS . . . . . 34
Virtualizing APIC.TPR . . . . . 34
TPR Access in 32-bit Mode . . . . . 34
Injecting Virtual (INTR) Interrupts . . . . . 35
Interrupt Shadows . . . . . 36
Virtual Interrupt Intercept . . . . . 36
Interrupt Masking in LocalAPIC . . . . . 37
INIT Support . . . . . 38
NMI Support . . . . . 39
2.18 SMM Support . . . . . 39
Sources of SMI . . . . . 39
Response to SMI . . . . . 39
Containerizing Platform SMM . . . . . 40
Advanced Support . . . . . 41
2.19 Last Branch Record Virtualization . . . . . 41
Enabling LBR Virualization . . . . . 42
Host and Guest LBR Virtualization . . . . . 42
LBR Virtualization CPUID Feature Detection . . . . . 42
2.20 External Access Protection . . . . . 42
Device IDs and Protection Domains . . . . . 43
Device Exclusion Vector (DEV) . . . . . 43
Host Bridge and Processor DEV Caching . . . . . 43

Multiprocessor Issues . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
Access Checking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
Memory Space Accesses . . . . . . . . . . . . . . . . . . . . . . . . . 44
I/O Space Accesses . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
Config Space Accesses . . . . . . . . . . . . . . . . . . . . . . . . . . 45
DEV Capability Block . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
DEV Capability Header . . . . . . . . . . . . . . . . . . . . . . . . . 46
DEV Register Access Mechanism . . . . . . . . . . . . . . . . . . . . . 46
DEV Control and Status Registers . . . . . . . . . . . . . . . . . . . . 47
DEV_CAP Register . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
DEV_CR Register . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
DEV_BASE Address/Limit Registers . . . . . . . . . . . . . . . 48
DEV_MAP Registers . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
Unauthorized Access Logging . . . . . . . . . . . . . . . . . . . . . . . . 50
Secure Initialization Support . . . . . . . . . . . . . . . . . . . . . . . . 50
2.21 Nested Paging Facility . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
Traditional Paging versus Nested Paging . . . . . . . . . . . . . . . . 51
Replicated State . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
Enabling Nested Paging . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
Nested Paging and VMRUN/#VMEXIT . . . . . . . . . . . . . . . . . 53
Nested Table Walk . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
Host versus Guest Page Faults, Fault Ordering . . . . . . . . . . . 54
Combining Host and Guest Attributes . . . . . . . . . . . . . . . . . . 55
Combining Memory Types, MTRRs . . . . . . . . . . . . . . . . . . . . 56
Memory Consistency Issues. . . . . . . . . . . . . . . . . . . . . . . . 56
Page Splintering . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
Legacy PAE Mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
A20 Masking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
Detecting Nested Paging Support . . . . . . . . . . . . . . . . . . . . . 59

**3 Security . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61**

SKINIT Instruction.. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
Automatic Memory Clearing.. . . . . . . . . . . . . . . . . . . . . . . 61
Security Exception.. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
3.1 Secure Startup with SKINIT . . . . . . . . . . . . . . . . . . . . . . . . 61
Secure Loader . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
Secure Loader Image . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
Secure Loader Block . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
Trusted Platform Module . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
System Interface, Memory Controller and I/O Hub Logic. . . 65
SKINIT Operation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
Pending interrupts. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
Debug considerations . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
SL Abort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
Secure Multiprocessor Initialization . . . . . . . . . . . . . . . . . . . 67
Software requirements for Secure MP initialization . . 67
AP Startup Sequence . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
Pending interrupts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

Aborting MP initialization . . . . . . . . . . . . . . . . . . . . . . . 68
3.2 Automatic Memory Clear . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
3.3 Security Exception (#SX) . . . . . . . . . . . . . . . . . . . . . . . . . . . 70

**4 SVM Instruction Set Reference . . . . . . . . . . . . . . . . . . . . . . . 71**

4.1 Changes to RSM Instruction . . . . . . . . . . . . . . . . . . . . . . . . 71
4.2 New Instructions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
CLGI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
INVLPGA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
MOV (CRn). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
SKINIT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
STGI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
VMLOAD . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
VMMCALL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
VMRUN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
VMSAVE. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87

**Appendix A Reset Values and INIT . . . . . . . . . . . . . . . . . . . . . . . . . 89**

**Appendix B Processor Feature Identification. . . . . . . . . . . . . . . . . . 91**

**Appendix C Layout of VMCB . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93**

**Appendix D Intercept Exit Codes . . . . . . . . . . . . . . . . . . . . . . . . . . 103**

**Appendix E New and Changed MSRs. . . . . . . . . . . . . . . . . . . . . . . 107**

# List of Figures

Figure 2-1. EXITINTINFO for All Intercepts . . . . . . . . . . . . . . . . . . . . . . . . 15
Figure 2-2. EXITINFO1 for IOIO Intercept. . . . . . . . . . . . . . . . . . . . . . . . . . 21
Figure 2-3. EVENTINJ Field in the VMCB . . . . . . . . . . . . . . . . . . . . . . . . . . 33
Figure 2-4. Format of SEOI register (in localAPIC). . . . . . . . . . . . . . . . . . . . 37
Figure 2-5. Host Bridge DMA Checking . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
Figure 2-7. Format of DEV_CAP Register (in PCI Config Space) . . . . . . . 48
Figure 2-8. Format of DEV_BASE_HI[n] Registers . . . . . . . . . . . . . . . . . . . 49
Figure 2-9. Format of DEV_BASE_LO[n] Registers. . . . . . . . . . . . . . . . . . . 49
Figure 2-10. Format of DEV_MAP[n] Registers . . . . . . . . . . . . . . . . . . . . . . . 49
Figure 2-11. Address Translation with Traditional Paging . . . . . . . . . . . . . . 51
Figure 2-12. Address Translation with Nested Paging. . . . . . . . . . . . . . . . . . 52
Figure 3-1. SLB Example Layout . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
Figure B-1. SVM Revision and Feature Identification in EAX, Extended Function 8000_000Ah . . . . . . . . . . . . . . . . . . . . . . . . . 91
Figure B-2. SVM Revision and Feature Identification in EBX, Extended Function 8000_000Ah . . . . . . . . . . . . . . . . . . . . . . . . . 91
Figure B-3. SVM Revision and Feature Identification in EDX, Extended Function 8000_000Ah . . . . . . . . . . . . . . . . . . . . . . . . . 91
Figure E-1. Layout of VM_CR MSR (C001_0114h) . . . . . . . . . . . . . . . . . . 107
Figure E-2. Layout of SMM_CTL MSR (C001_0116h) . . . . . . . . . . . . . . . . 108
Figure E-3. Extended APIC feature register. . . . . . . . . . . . . . . . . . . . . . . . 110
Figure E-4. Extended APIC control register. . . . . . . . . . . . . . . . . . . . . . . . 110

# List of Tables

Table 2-1. Guest Exception or Interrupt Types. . . . . . . . . . . . . . . . . . . . . . 15
Table 2-2. Ranges of MSR Permissions Map . . . . . . . . . . . . . . . . . . . . . . . . 23
Table 2-3. Effect of the GIF on Interrupt Handling . . . . . . . . . . . . . . . . . . 31
Table 2-4. Guest Exception or Interrupt Types. . . . . . . . . . . . . . . . . . . . . . 33
Table 2-5. INIT Handling in Different Operating Modes. . . . . . . . . . . . . . . 38
Table 2-6. NMI Handling in Different Operating Modes . . . . . . . . . . . . . . 39
Table 2-7. SMI Handling in Different Operating Modes . . . . . . . . . . . . . . 40
Table 2-8. DEV Capability Block, Overall Layout . . . . . . . . . . . . . . . . . . . 46
Table 2-9. DEV Capability Header (DEV_HDR) (in PCI Config Space) . 46
Table 2-10. Encoding of function field in DEV_OP register . . . . . . . . . . . . 47
Table 2-11. DEV_CR Control Register. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
Table 2-12. Combining Guest and Host PAT Types . . . . . . . . . . . . . . . . . . . 58
Table 2-13. Combining PAT and MTRR Types . . . . . . . . . . . . . . . . . . . . . . . 58
Table C-1. VMCB Layout, Control Area . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
Table C-2. VMCB Layout, State Save Area . . . . . . . . . . . . . . . . . . . . . . . . . 99
Table D-1. SVM Intercept Codes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
Table E-1. SVM New MSRs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
Table E-2. Secure-VM localAPIC Registers . . . . . . . . . . . . . . . . . . . . . . . 110

# Revision History

| Date | Revision | Description |
|---|---|---|
| December 2005 | 3.02 | Added documentation of LBR virtualization in Section 2.19 and the VMCB layout tables in Appendix B; Updated documentation of nested paging in Section 2.21; Converted SVM intercept codes in Table D-1 to hexadecimal. |
| May 2005 | 3.01 | Corrected factual errors in Section 2.21.13, “Other Guest Attributes,” on page 61. |
| April 2005 | 3.00 | First Public Release. |

# Preface

## About This Book

This book describes the AMD64 technology Security and Virtual Machine (SVM) architecture, software requirements, instruction set extensions, changes to existing instructions, and new bit settings in system registers.

## Audience

This volume is intended for programmers writing virtual machine monitor software and other SVM applications or system utilities. It assumes an understanding of AMD64 architecture application-level and system-level programming as described in Volumes 1 and 2 of the AMD64 Architecture Programmer's Manual (order# 24592 and order# 24593).

This volume describes SVM architecture resources and functions that are managed by system software, including operating-mode control, memory management, intercepts, interrupts and exceptions, state-change management, system-management mode, and processor initialization, as well as extensions to the AMD64 instruction set that are used to operate on SVM data structures.

## Organization

This volume begins with an overview of SVM, followed by chapters that describe the following details of system programming:

- *System Resources*—The data structures, system registers, software responsibilities, and hardware support to implement SVM systems.
- *SVM Instruction Set*—The extensions to the AMD64 instruction set used to control SVM operations.

The appendices describe details of model-specific registers (MSRs) and data strucure layout. Definitions assumed throughout this volume are listed below. The index at the end of this volume cross-references topics within the volume. For other topics relating to the AMD64 architecture, see the tables of contents and indices of the references given in "Related Documents" on page xxv.

## Definitions

Some of the following definitions assume a knowledge of the legacy x86 architecture. See "Related Documents" on page xxv for descriptions of the legacy x86 architecture.

### Terms and Notation

*1011b*

A binary value—in this example, a 4-bit value.

*F0EAh*

A hexadecimal value—in this example a 2-byte value.

*[1,2)*

A range that includes the left-most value (in this case, 1) but excludes the right-most value (in this case, 2).

*7:4*

A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first.

*32-bit mode*

Legacy mode or compatibility mode in which a 32-bit address size is active. See *legacy mode* and *compatibility mode.*

*64-bit mode*

A submode of *long mode*. In 64-bit mode, the default address size is 64 bits and new features, such as register extensions, are supported for system and application software.

*#GP(0)*

Notation indicating a general-protection exception (#GP) with error code of 0.

*absolute*

A displacement that references the base of a code segment rather than an instruction pointer. Contrast with *relative*.

*ASID*

Address space identifier.

*byte*

Eight bits.

*clear*

To write a bit value of 0. Compare *set*.

*compatibility mode*

A submode of *long mode*. In compatibility mode, the default address size is 32 bits, and legacy 16-bit and 32-bit applications run without modification.

*CPL*

Current privilege level.

*CR0–CR4*

A register range, from register CR0 through CR4, inclusive, with the low-order register first.

*CR0.PE = 1*

Notation indicating that the PE bit of the CR0 register has a value of 1.

*displacement*

A signed value that is added to the base of a segment (absolute addressing) or an instruction pointer (relative addressing). Same as *offset*.

*doubleword*

Two words, or four bytes, or 32 bits.

*double quadword*

Eight words, or 16 bytes, or 128 bits. Also called *octword*.

*DS:rSI*

The contents of a memory location whose segment address is in the DS register and whose offset relative to that segment is in the rSI register.

*EFER.LME = 0*

Notation indicating that the LME bit of the EFER register has a value of 0.

*effective address size*

The address size for the current instruction after accounting for the default address size and any address-size override prefix.

*effective operand size*

The operand size for the current instruction after accounting for the default operand size and any operand-size override prefix.

*element*

See *vector*.

*exception*

An abnormal condition that occurs as the result of executing an instruction. The processor's response to an exception depends on the type of the exception. Control is transferred to the handler (or service routine) for that exception, as defined by the exception's vector. When unmasked, the exception handler is called, and when masked, a default response is provided instead of calling the handler.

*FF /0*

Notation indicating that FF is the first byte of an opcode, and a subopcode in the ModR/M byte has a value of 0.

*flush*

An often ambiguous term meaning (1) writeback, if modified, and invalidate, as in "flush the cache line," or (2) invalidate, as in "flush the pipeline," or (3) change a value, as in "flush to zero."

*GIF*
Global interrupt flag.

*GDT*
Global descriptor table.

*IDT*
Interrupt descriptor table.

*IGN*
Ignore. Field is ignored.

*IVT*
The real-address mode interrupt-vector table.

*LDT*
Local descriptor table.

*long mode*
An operating mode unique to the AMD64 architecture. A processor implementation of the AMD64 architecture can run in either *long mode* or *legacy mode*. Long mode has two submodes, *64-bit mode* and *compatibility mode*.

*lsb*
Least-significant bit.

*LSB*
Least-significant byte.

*main memory*
Physical memory, such as RAM and ROM (but not cache memory) that is installed in a particular computer system.

*mask*
A field of bits used for a control purpose.

*MBZ*
Must be zero. If software attempts to set an MBZ bit to 1, a general-protection exception (#GP) occurs.

*memory*
Unless otherwise specified, *main memory*.

*msb*
Most-significant bit.

*MSB*
Most-significant byte.

*octword*
Same as *double quadword.*

*offset*
Same as *displacement*.

*PAE*
Physical-address extensions.

*physical memory*
Actual memory, consisting of *main memory* and cache.

*probe*
A check for an address in a processor's caches or internal buffers. *External probes* originate outside the processor, and *internal probes* originate within the processor.

*protected mode*
A submode of *legacy mode*.

*quadword*
Four words, or eight bytes, or 64 bits.

*RAZ*
Read as zero (0), regardless of what is written.

*real-address mode*
See *real mode*.

*real mode*
A short name for *real-address mode,* a submode of *legacy mode*.

*relative*
A displacement (also called offset) from an instruction pointer rather than the base of a code segment. Contrast with *absolute*.

*reserved*
Fields marked as reserved may be used at some future time.

To preserve compatibility with future processors, reserved fields require special handling when read or written by software.

Reserved fields may be further qualified as MBZ, RAZ, SBZ or IGN (see definitions).

Software must not depend on the state of a reserved field, nor upon the ability of such fields to return to a previously written state.

If a reserved field is not marked with one of the previous qualifiers, software must not change the state of that field; it must reload that field with the same values returned from a prior read.

*REX*

An instruction prefix that specifies a 64-bit operand size and provides access to additional registers.

*SBZ*

Should be zero. It is the responsibility of software to set SBZ bits to zero. The result of setting an SBZ bit to 1 may be unpredictable.

*set*

To write a bit value of 1. Compare *clear*.

*sticky bit*

A bit that is set or cleared by hardware and that remains in that state until explicitly changed by software.

*TSS*

Task-state segment.

*vector*

An index into an interrupt descriptor table (IDT), used to access exception handlers. Compare *exception*.

*virtual-8086 mode*

A submode of *legacy mode*.

*VMCB*

Virtual machine control block.

*VMM*

Virtual machine monitor.

*word*

Two bytes, or 16 bits.

*x86*

See *legacy x86*.

**Registers**

In the following list of registers, the names are used to refer either to a given register or to the contents of that register:

*AH–DH*

The high 8-bit AH, BH, CH, and DH registers. Compare *AL–DL.*

*AL–DL*

The low 8-bit AL, BL, CL, and DL registers. Compare *AH–DH.*

*AL–r15B*

The low 8-bit AL, BL, CL, DL, SIL, DIL, BPL, SPL, and R8B–R15B registers, available in 64-bit mode.

*BP*

Base pointer register.

*CRn*

Control register number *n*.

*CS*

Code segment register.

*eAX–eSP*

The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers. Compare *rAX–rSP.*

*EBP*

Extended base pointer register.

*EFER*

Extended features enable register.

*eFLAGS*

16-bit or 32-bit flags register. Compare *rFLAGS*.

*EFLAGS*

32-bit (extended) flags register.

*eIP*

16-bit or 32-bit instruction-pointer register. Compare *rIP*.

*EIP*

32-bit (extended) instruction-pointer register.

*FLAGS*

16-bit flags register.

*GDTR*

Global descriptor table register.

*GPRs*

General-purpose registers. For the 16-bit data size, these are AX, BX, CX, DX, DI, SI, BP, and SP. For the 32-bit data size, these are EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP. For the 64-bit data size, these include RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, and R8–R15.

*IDTR*

Interrupt descriptor table register.

*IP*

16-bit instruction-pointer register.

*LDTR*

Local descriptor table register.

*MSR*

Model-specific register.

*r8–r15*

The 8-bit R8B–R15B registers, or the 16-bit R8W–R15W registers, or the 32-bit R8D–R15D registers, or the 64-bit R8–R15 registers.

*rAX–rSP*

The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers, or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers, or the 64-bit RAX, RBX, RCX, RDX, RDI, RSI, RBP, and RSP registers. Replace the placeholder *r* with nothing for 16-bit size, “E” for 32-bit size, or “R” for 64-bit size.

*RAX*
64-bit version of the EAX register.

*RAZ*
Read as zero (0), regardless of what is written.

*RBP*
64-bit version of the EBP register.

*RBX*
64-bit version of the EBX register.

*RCX*
64-bit version of the ECX register.

*RDI*
64-bit version of the EDI register.

*RDX*
64-bit version of the EDX register.

*rFLAGS*
16-bit, 32-bit, or 64-bit flags register. Compare *RFLAGS*.

*RFLAGS*
64-bit flags register. Compare *rFLAGS*.

*rIP*
16-bit, 32-bit, or 64-bit instruction-pointer register. Compare *RIP*.

*RIP*
64-bit instruction-pointer register.

*RSI*
64-bit version of the ESI register.

*RSP*
64-bit version of the ESP register.

*SP*
Stack pointer register.

*SS*
Stack segment register.

*TPR*

Task priority register (CR8), a new register introduced in the AMD64 architecture to speed interrupt management.

*TR*

Task register.

**Endian Order**

The x86 and AMD64 architectures address memory using little-endian byte-ordering. Multibyte values are stored with their least-significant byte at the lowest byte address, and they are illustrated with their least significant byte at the right side. Strings are illustrated in reverse order, because the addresses of their bytes increase from right to left.

## Related Documents

- *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*, order# 24592.
- *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, order# 24593.
- *AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions*, order# 24594.

# 1 Introduction

AMD security and virtual machine (SVM) architecture is designed to provide enterprise-class server virtualization software technology that facilitates virtualization development and deployment. An SVM enabled virtual machine architecture should provide hardware resources that allow a single machine to run multiple operating systems efficiently, while maintaining secure, resource-guaranteed isolation.

## 1.1 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.

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). AMD's SVM provides hardware assists to improve performance and facilitate implementation of virtualization.

## 1.2 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.

### 1.2.1 Virtualization Support

The AMD virtual machine architecture is designed to provide:

- Mechanisms for fast world switch between VMM and guest
- The ability to intercept selected instructions or events in the guest
- External (DMA) access protection for memory.

- Assists for interrupt handling and virtual interrupt support
- A guest/host tagged TLB to reduce virtualization overhead.

### 1.2.2 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.

### 1.2.3 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).

### 1.2.4 Tagged TLB

In the SVM usage model, the VMM is mapped in a different address space than the guest. To reduce the cost of world switches, the TLB is tagged with an address space identifier (ASID) distinguishing host-space entries from guest-space entries.

### 1.2.5 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 while guarding against malicious or defective guests that might leave high-priority interrupts unacknowledged forever (and thus shut out other guest's interrupts).

### 1.2.6 Restartable Instructions

SVM is designed to safely restart, with the exception of task switches, any intercepted instruction after the intercept. Instructions are either atomic or idempotent.

### 1.2.7 Security Support

To further enable secure initialization SVM provides additional System support.

**Attestation.** The SKINIT instruction and associated system support (the Trusted Platform Module, or TPM) allow for verifiable startup of trusted software (such as a VMM), based on secure hash comparison.

**Memory Clear.** Automatic memory clear erases the contents of system memory on reset to prevent simple reset-based attacks on secrets stored in memory.

# 2 SVM Processor and Platform Extensions

This chapter describes the operation of the SVM hardware extensions. These 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). (“VMRUN Instruction” on page 5, “VMSAVE and VMLOAD Instructions” on page 29, “Global Interrupt Flag, STGI and CLGI Instructions” on page 31)
- Intercepts—allow the VMM to intercept sensitive operations in the guest. (“Intercept Operation” on page 13 through “Miscellaneous Intercepts” on page 28)
- Interrupt and APIC assists—physical interrupt intercepts, virtual interrupt support, APIC.TPR virtualization. (“Global Interrupt Flag, STGI and CLGI Instructions” on page 31 and “Interrupt and localAPIC Support” on page 35)
- SMM intercepts and assists (“SMM Support” on page 40)
- External (DMA) access protection (“External Access Protection” on page 43)
- Nested paging support for two levels of address translation. (“Nested Paging Facility” on page 52)
- Security—SKINIT instruction, automatic memory clear. (“Secure Startup with SKINIT” on page 61)

## 2.1 Enabling SVM

Before any SVM instruction (VMRUN, VMLOAD, VMSAVE, VMMCALL, STGI, CLGI, SKINIT, INVLPGA) can be used, EFER.SVME (bit 12 of the EFER MSR register) must be set to 1. While EFER.SVME is zero (the default after reset), SVM instructions cause #UD faults.

## 2.2 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 which 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.).

### 2.2.1 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 exception is raised if the CPL is greater than 0). Furthermore, the processor must be in protected mode and SVME.EFER 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_AREA 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 illegal state has been loaded, the processor exits back to the host (see “#VMEXIT” on page 12).

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 “#VMEXIT” on page 12.

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

can employ the VMSAVE and VMLOAD instructions (see “VMSAVE and VMLOAD Instructions” on page 29).

**Saving Host State.** To assure that the host can resume operation after #VMEXIT, VMRUN saves at least the following host state information at the physical address specified in the new MSR, VM_HSAVE_PA:

- 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—Host’s stack pointer.
- CR0, CR3, CR4, EFER—Host’s paging/operating mode.
- 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_AREA 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).
- INTERRUPT_SHADOW—This flag indicates whether the guest is currently in an interrupt lockout shadow; see “Interrupt Shadows” on page 36.

- IDTR, GDTR.
- ES and DS—Includes the hidden state of the segment registers.
- DR7 and DR6—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 an illegal mode is detected or an exception was encountered while loading guest state, the processor performs a #VMEXIT immediately and stores VMEXIT_INVALID as an error indication in the VMCB EXITCODE field.

If the guest is in PAE paging mode according to the registers just loaded, 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's 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's 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).

**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 36).
- V_INTR_MASKING—controls whether masking of interrupts (in EFLAGS.IF and TPR) is to be virtualized (see Section 2.17 on page 35).
- The address space ID (ASID) to use while running the guest. (See Appendix B, “Processor Feature Identification,” on page 91 for feature identification, including how many ASIDs are implemented.)
- A field to control flushing of the TLB during a VMRUN (see Section 2.12).
- The intercept vector 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.

**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. When loaded from the VMCB, only some of the attribute bits are observed by hardware, depending on the segment register in question:

- CS—D, L, R (null code segment are not allowed).
- SS—B, P, DPL, E, W (null stack segments allowed in 64-bit mode only).
- DS, ES, FS, GS —D, P, DPL, E, W, Code/Data.
- LDTR—Only the P bit is observed.
- TR—Only TSS type (32 or 16 bit) is relevant, since a null TSS is not allowed.

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, not from SS.DPL. However, SS.DPL should 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 meaningless;
- 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 host and guest state, very much like RSM performs checks on the new state. Illegal guest state combinations cause a #VMEXIT with error code of VMEXIT_INVALID. 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 bits of CR3 are set.
- CR4[63–11] are not zero.
- DR6[63–32] are not zero.
- DR7[63–32] are not zero.
- EFER[63–15] are not zero.
- 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 or SMI intercept bits are clear.
- The SMI intercept bit is zero.
- (Other MBZ bits exist in various registers stored in the VMCB.)

- The MSR or IOIO intercept tables extend to a physical address ≥ the maximum supported physical address
- Illegal event injection (see Section 2.16 on page 33).

VMRUN can load a guest value of CR0 with PE = 0 but PG = 1, a combination that is otherwise illegal (see Section 2.15).

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.

**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 first guest instruction obeys the value of EFLAGS.RF from the VMCB. 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 2.4.1 on page 14 regarding the value of EFLAGS.RF saved on #VMEXIT.

## 2.3 #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.
- 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.

*Note: 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. 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. 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 shuts down.
- 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.

*Note: When loading segment bases from the VMCB or the host-save area (on VMRUN or #VMEXIT), segment bases are* canonicalized *(i.e., sign-extended from the highest implemented address bit to bit 63); see the AMD64 Architecture Programmer's Manual, Volume 2: System Programming, order# 24593.*

Any illegal state or exception encountered while reloading host segment state in the VMCB state will cause a processor shutdown.

## 2.4 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 according to table 8-3 in Volume 2 of the *AMD64 Architecture Programmer's Manual*, order# 24593, 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 2.8 on page 23 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 2.4.2 on page 14).

**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.

### 2.4.1 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 (see "Event Injection" on page 33). 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 at the following instruction, and the saved DR7 includes the effects of hitting the data breakpoint.

Some exceptions write special registers even when they are intercepted; see the individual descriptions in "Exception Intercepts" on page 23 for details.

### 2.4.2 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 though 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.

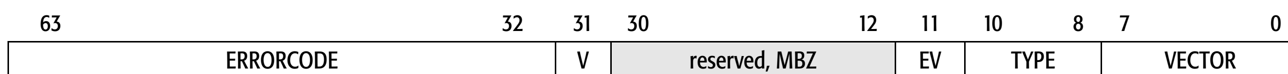


**Figure 2-1. EXITINTINFO for All Intercepts**

The fields in EXITINTINFO are as follows:

- VECTOR—Bits 7–0. The 8-bit IDT vector of the interrupt.
- TYPE—Bits 10–8. Qualifies the guest exception or interrupt. Table 2-1 shows possible values returned and their corresponding interrupt or exception types. Values not indicated are unused and reserved.

**Table 2-1. Guest Exception or Interrupt Types**

| Value | Type |
|---|---|
| 0 | External or virtual interrupt (INTR) |
| 2 | NMI |
| 3 | Exception (fault or trap) |
| 4 | Software interrupt (caused by INT*n* instruction) |

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, not software interrupts. Only events raised by the INT*n* instruction (opcode CDh) are considered software interrupts.

- EV (error code valid)—Bit 11. Set to 1 if the guest exception would have pushed an error code; cleared to zero otherwise.
- V (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 (and 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 "Event Injection" on page 33) 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 (see table 8-3 in Volume 2 of the AMD64 Architecture Programmer's Manual, order# 24593). In this case, a #DF plus a #PF would result in a triple fault or shutdown.

### 2.4.3 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.

```
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 (raw) intercepted exception, while EXITINTINFO (if marked valid) indicates the prior exception the guest was attempting to deliver when the intercept occurred.

## 2.5 Instruction Intercepts

This section specifies which instructions check a given intercept and, where relevant, how the intercept is prioritized relative to exceptions.

### 2.5.1 Read/Write of CR0

**Checked by—**MOV TO/FROM CR0, LMSW, SMSW, CLTS.

**Priority—**Checks non-memory exceptions (CPL, illegal bit combinations, etc.) before the intercept. For LMSW and SMSW, checks SVM intercepts before checking memory exceptions.

### 2.5.2 Read/Write of CR3 (excluding task switch)

**Checked by—**MOV TO/FROM CR3 (not checked by task switch operations).

**Priority—**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.

### 2.5.3 Read/Write of other CRs

**Checked by—**MOV TO/FROM CR*n*.

**Priority—**All normal exception checks take precedence over the SVM intercepts.

### 2.5.4 Read/Write of Debug Registers, DR*n*

**Checked by—**MOV TO/FROM DR*n*. (Not checked by implicit DR6/DR7 writes.)

**Priority—**All normal exception checks take precedence over the SVM intercepts.

### 2.5.5 Selective CR0 Write Intercept

**Checked by—**MOV TO CR0, LMSW

**Priority—**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).

### 2.5.6 Reading/Writing of IDTR, GDTR, LDTR, TR

**Checked by—**LIDT, SIDT, LGDT, SGDT, LLDT, SLDT, LTR, STR instructions, respectively.

**Priority—**The SVM intercept is checked after #UD and #GP exception checks, but before any memory access is performed.

### 2.5.7 RDTSC Instruction Intercept

**Checked by—**RDTSC instruction

**Priority—**Checks all exceptions before the SVM intercept.

### 2.5.8 RDPMC Instruction Intercept

**Checked by—**RDPMC instruction

**Priority—**Checks all exceptions before the SVM intercept.

### 2.5.9 PUSHF Instruction Intercept

**Checked by—**PUSHF instruction.

**Priority—**The intercept takes priority over any exceptions.

### 2.5.10 POPF Instruction Intercept

**Checked by—**POPF instruction.

**Priority—**The intercept takes priority over any exceptions.

### 2.5.11 CPUID Instruction Intercept

**Checked by—**CPUID instruction.

**Priority—**No exceptions to check.

### 2.5.12 RSM Instruction Intercept

**Checked by—**RSM instruction.

**Priority—**The intercept takes priority over any exceptions.

### 2.5.13 IRET Instruction Intercept

**Checked by—**IRET instruction.

**Priority—**The intercept takes priority over any exceptions.

### 2.5.14 Software Interrupt Intercept

**Checked by—**INT*n* instruction.

**Priority—**The intercept occurs before any exceptions are checked. The CS:RIP reported on #VMEXIT are those of the intercepted INT*n* instruction.

Though the INT*n* instruction may dispatch through IDT vectors in the range of 0–31, those events cannot be intercepted by means of exception intercepts (“Exception Intercepts” on page 23).

### 2.5.15 INVD Instruction Intercept

**Checked by—**INVD instruction.

**Priority—**Exceptions (#GP) are checked before the intercept.

### 2.5.16 PAUSE Instruction Intercept

**Checked by—**PAUSE instruction (opcode F3 90).

**Priority—**No exceptions to check.

### 2.5.17 HLT Instruction Intercept

**Checked by—**HLT instruction.

**Priority—**Checks all exceptions before checking for this intercept.

### 2.5.18 INVLPG Instruction Intercept

**Checked by—**INVLPG instruction.

**Priority—**Checks all exceptions (#GP) before the intercept.

### 2.5.19 INVLPGA Instruction Intercept

**Checked by—**INVLPGA instruction.

**Priority—**Checks all exceptions (#GP) before the intercept.

### 2.5.20 VMRUN Instruction Intercept

**Checked by—**VMRUN instruction.

**Priority—**Checks exceptions (#GP) before the intercept.

*Note: The VMRUN intercept must always be set in the VMCB.*

### 2.5.21 VMLOAD Instruction Intercept

**Checked by—**VMLOAD instruction.

**Priority—**Checks exceptions (#GP) before the intercept.

#### 2.5.22 VMSAVE Instruction Intercept

**Checked by—**VMSAVE instruction.

**Priority—**Checks exceptions (#GP) before the intercept.

#### 2.5.23 VMMCALL Instruction Intercept

**Checked by—**VMMCALL instruction.

**Priority—**The intercept takes priority over exceptions. VMMCALL causes #UD in the guest if it is not intercepted.

#### 2.5.24 STGI Instruction Intercept

**Checked by—**STGI instruction.

**Priority—**Checks exceptions (#GP) before the intercept.

#### 2.5.25 CLGI Instruction Intercept

**Checked by—**CLGI instruction.

**Priority—**Checks exceptions (#GP) before the intercept.

#### 2.5.26 SKINIT Instruction Intercept

**Checked by—**SKINIT instruction.

**Priority—**Checks exceptions (#GP) before the intercept.

#### 2.5.27 RDTSCP Instruction Intercept

**Checked by—**RDTSCP instruction.

**Priority—**Checks all exceptions before the SVM intercept.

#### 2.5.28 ICEBP Instruction Intercept

**Checked by—**ICEBP instruction (opcode F1h).

*Note: Although the ICEBP instruction dispatches through IDT vector 1, that event is not interceptable by means of the #DB exception intercept.*

#### 2.5.29 WBINVD Instruction Intercept

**Checked by—**WBINVD instructions.

**Priority—**Checks exceptions (#GP) before the intercept.

## 2.6 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.*

**I/O Permissions Map.** The I/O Permissions Map (IOPM) occupies 12 Kbytes of *contiguous physical* memory. The table 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.

*Note: The VMRUN instruction ignores the lower 12 bits of the address specified in the VMCB. 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).*

Each bit in the table 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).

**IN and OUT Behavior.** If the IOIO_PROT intercept bit is set, the IOPM table 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:

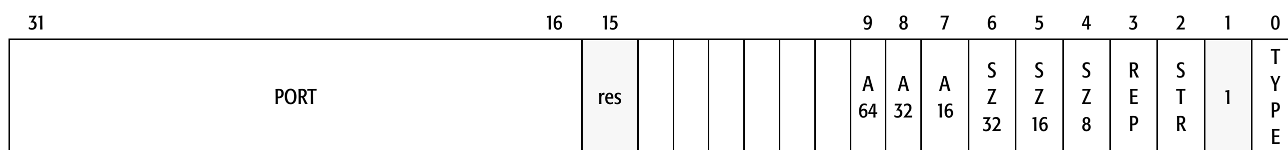


**Figure 2-2. EXITINFO1 for IOIO Intercept**

The fields are as follows:

- PORT—Intercepted I/O port
- A64—64-bit address size
- A32—32-bit address size
- A16—16-bit address size
- SZ32—32-bit operand size
- SZ16—16-bit operand size
- SZ8—8-bit operand size

- REP—Repeated port access
- STR—String based port access (INS, OUTS)
- TYPE—Access type (0 = OUT instruction, 1 = IN instruction)

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.

## 2.7 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 a number of smaller separate bitmaps of 2K bytes each covering a defined range of 8K MSRs. Four of these smaller bitmaps reside in two physical pages (8KB, covering 32K MSRs). One 8Kbyte range is used for the Pentium® compatible MSRs, the next 8K range is used for the AMD sixth generation x86 processor (AMD-K6®) MSRs, and the third 8K range for the AMD seventh and eighth generation x86 processors (e.g., the AMD Athlon™ and AMD Opteron™) MSRs. If the MSR_PROT intercept is active, any attempt to read or write an MSR not covered by the bitmap will automatically cause an intercept.

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 loaded into the processor by the VMRUN instruction.

*Note: 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).*

**Table 2-2. Ranges of MSR Permissions Map**

| Byte Offset | MSR Range | Current Usage |
|---|---|---|
| 000h–7FFh | 0000_0000h–0000_1FFFh | Pentium®-compatible MSRs |
| 800h–FFFh | C000_0000h–C000_1FFFh | AMD Sixth Generation x86 Processor MSRs and SYSCALL |
| 1000h–17FFh | C001_0000h–C001_1FFFh | AMD Seventh and Eighth Generation Processor MSRs |
| 1800h–1FFFh | XXXX_XXXX–XXXX_XXXX | reserved |

Table 2-2 defines the ranges of the MSR permissions map. For each MSR mapped by the table, two bits are allocated—the lower order of the two bits controls read access to the MSR, and the higher order of the two bits controls write access. A bit value of 1 indicates that the operation is intercepted.

**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.

## 2.8 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 inferred from the EXITCODE.

The CS.SEL and RIP saved in the VMCB on an exception-intercept always match those that would otherwise have been pushed onto the exception stack frame. 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 2.4.1.

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.

### 2.8.1 #DE (Divide By Zero)

The EXITINFO1 and EXITINFO2 fields are undefined.

### 2.8.2 #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 “State Saved on Exit” on page 14). 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.

*Note: 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 “ICEBP Instruction Intercept” on page 20).*

### 2.8.3 Vector 2 (Reserved)

This intercept bit is not implemented; use the NMI intercept (Section 2.9.2) instead. The effect of setting this bit is undefined.

### 2.8.4 #BP (Breakpoint)

This intercept applies to the trap raised by the single byte INT3 (opcode CCh) instruction. The EXITINFO1 and EXITINFO2 fields are undefined.

### 2.8.5 #OF (Overflow)

This intercept applies to the trap raised by the INTO (opcode CEh) instruction. The EXITINFO1 and EXITINFO2 fields are undefined.

### 2.8.6 #BR (Bound-Range)

This intercept applies to the fault raised by the BOUND instruction. The EXITINFO1 and EXITINFO2 fields are undefined.

### 2.8.7 #UD (Invalid Opcode)

The EXITINFO1 and EXITINFO2 fields are undefined.

### 2.8.8 #NM (Device-Not-Available)

The EXITINFO1 and EXITINFO2 fields are undefined.

### 2.8.9 #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).

*Note: 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.*

### 2.8.10 Vector 9 (Reserved)

This intercept is not implemented. The effect of setting this bit is undefined.

### 2.8.11 #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.

### 2.8.12 #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.

### 2.8.13 #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.

#### 2.8.14 **#GP (General Protection)**

The EXITINFO1 field contains the error code that would be pushed on the stack by a #GP exception.

#### 2.8.15 **#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.

*Note: 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.*

#### 2.8.16 **#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.

#### 2.8.17 **#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.

#### 2.8.18 **#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 wherever possible, and shuts down the processor only where this is not a reasonable option. The EXITINFO1 and EXITINFO2 fields are undefined.

#### 2.8.19 **#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.

## 2.9 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.

### 2.9.1 INTR Intercept

This intercept affects physical, as opposed to virtual, maskable interrupts. See “Virtual Interrupt Intercept” on page 37 for virtualization of maskable interrupts.

### 2.9.2 NMI Intercept

This intercept affects non-maskable interrupts.

### 2.9.3 SMI Intercept

This intercept affects System Management Mode Interrupts (SMIs); see “SMM Support” on page 40 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 = 1), 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.

If the SMI wasn't asserted during an I/O instruction, the extra EXITINFO1 and EXITINFO2 bits are undefined.

*Note: The current implementation requires that the SMI intercept always be set in the VMCB.*

**Figure 2-3. EXITINFO1 for SMI Intercept**

The fields are as follows:

- PORT—Intercepted I/O port
- A64—64-bit address size
- A32—32-bit address size
- A16—16-bit address size
- SZ32—32-bit operand size
- SZ16—16-bit operand size

- SZ8—8-bit operand size
- REP—Repeated port access
- STR—String based port access (INS, OUTS)
- VAL—Valid
- TYPE—Access type (0 = OUT instruction, 1 = IN instruction).
- SMISRC—SMI source (0 = internal, 1 = external).

#### 2.9.4 INIT Intercept

This allows the VMM to intercept the assertion of INIT while a guest is running; see “INIT Support” on page 39 for a discussion of the INIT-redirection feature.

#### 2.9.5 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.

*Note: This intercept is not required for handling fixed localAPIC interrupts, but may be used for emulating ExtINT interrupt delivery mode (which does not obey the TPR), or legacy PICs in auto-EOI mode.*

## 2.10 Miscellaneous Intercepts

The SVM architecture includes intercepts to handle task switches, processor freezes due to FERR, and shutdown operations.

#### 2.10.1 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 had been involved) have been checked for correctness.

Task switches can modify several resources that a VMM may need to protect (CR3, EFLAGS, LDT). However, instead of checking various intercepts (e.g., CR3 Write, LDTR Write) individually, task switches check only this 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 (undefined if not applicable).
- EXITINFO2[63–32] holds auxiliary information:
  - 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 errorcode; 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.

#### 2.10.2 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.

#### 2.10.3 Shutdown Intercept

When this intercept occurs, any condition that normally causes a shutdown causes a #VMEXIT to the VMM instead.

*Note: After an intercepted shutdown, the state saved in the VMCB is undefined.*

## 2.11 VMSAVE and VMLOAD Instructions

The VMSAVE and VMLOAD instructions take the physical address of a VMCB in the (implicit) rAX operand. The instructions are intended to complement the state save/restore abilities of the VMRUN instruction. They provide access to hidden processor state that software cannot otherwise access, as well as additional privileged state.

VMSAVE saves the following state to the VMCB indicated by rAX:

- FS, GS, TR, LDTR (including all hidden state)
- KernelGsBase
- STAR, LSTAR, CSTAR, SFMASK
- SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP

VMLOAD loads the corresponding state from the VMCB. VMLOAD and VMSAVE are available only at CPL-0 (#GP

otherwise), and in protected mode with SVM enabled in EFER.SVME (#UD otherwise).

## 2.12 TLB Control

TLB entries are tagged with *Address Space Identifier* (ASID) bits to distinguish different host and/or guest address spaces. The VMM can choose a software strategy in which it keeps multiple shadow page tables (SPTs) and/or multiple host page tables (in processors that support nested paging) up-to-date; the VMM can allocate a different ASID for each SPT or host page table. This allows switching to a new process in a guest (i.e., a new CR3 value, which means a new SPT) without flushing the TLBs.

For each guest address space, the VMM is responsible for setting up a shadow page table or host page table that maps guest virtual addresses to host physical addresses. In shadow paging, the VMM should set the CR3 field in the guest VMCB to point to the host physical address of this shadow page table. The VMM is responsible for updating the shadow page table when the guest changes the page table or paging control state, and the VMM should update the access and dirty bits of the guest page table.

The VMRUN instruction and #VMEXIT write the CR0, CR3, CR4 and EFER registers — 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, as 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.

### 2.12.1 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), and affect all ASIDs. However, 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 currently has one command implemented. 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.

### 2.12.2 Invalidate Page, Alternate ASID

A new instruction, INVLPGA, allows the VMM to selectively invalidate the TLB mapping for a given virtual page and a given ASID. The virtual address is specified in the implicit register operand rAX; the ASID is specified in ECX.

## 2.13 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 2-3 shows how the value of the GIF affects how interrupts and exceptions are handled.

**Table 2-3. Effect of the GIF on Interrupt Handling**

| Interrupt source | GIF==0 | GIF ==1 |
|---|---|---|
| Debug exception or trap, due to breakpoint register match | Ignored and discarded | Normal operation |
| Debug trace trap due to EFLAGS.TF | Normal operation | Normal operation |
| RESET# | Normal operation | Normal operation |
| INIT | Held pending until GIF==1 | Normal operation, see Table 2-5 on page 39 |

**Table 2-3. Effect of the GIF on Interrupt Handling** (continued)

| Interrupt source | GIF==0 | GIF ==1 |
|---|---|---|
| NMI | Held pending until GIF==1 | Normal operation, see Table 2-6 on page 40 |
| External SMI | Held pending until GIF==1 | Normal operation, see Table 2-7 on page 41 |
| Internal SMI (I/O Trapping) | Ignored and discarded | Normal operation, see Table 2-7 on page 41 |
| INTR and vINTR | Held pending until GIF==1 | Normal operation |
| #SX (Security Exception) | n/a[1] | Normal operation |
| Machine Check | If possible (implementation-dependent), held pending until GIF==1, otherwise shutdown. | Normal operation |
| DBREQ# (enter HDT) | Normal operation | Normal operation |
| | (VM_CR.DPD always controls DBREQ) | |
| A20M | Normal operation | Normal operation |
| | (VM_CR.DIS_A20M controls A20 masking) | |
| Other implementation-specific but non-architecturally-visible interrupts (STPCLK, IGNNE toggle, ECC scrub) | Normal operation | Normal operation |

***Note:***

*1. #SX is caused only by an INIT signal that has been "redirected" (i.e., converted to an #SX; see Section 3.3); the conversion only happens when GIF==1, as the INIT is simply held pending otherwise.*

## 2.14 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.

## 2.15 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 host physical addresses.

The behavior of running a guest in paged real mode without also intercepting page faults to the VMM is undefined.

## 2.16 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 2-4 on page 34. 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 unconditionally take the specified exception or interrupt 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 two 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 errorcode of VMEXIT_INVALID.

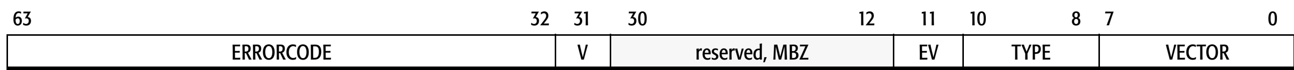


**Figure 2-4. EVENTINJ Field in the VMCB**

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 2-4 shows possible values and their corresponding interrupt or exception types. Values not indicated are unused and reserved.

**Table 2-4. Guest Exception or Interrupt Types**

| Value | Type |
|---|---|
| 0 | External or virtual interrupt (INTR) |
| 2 | NMI |
| 3 | Exception (fault or trap) |
| 4 | Software interrupt (INT*n* instruction) |

- 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.

*Note: 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.*

VMRUN exits with VMEXIT_INVALID 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).

## 2.17 Interrupt and localAPIC Support

SVM hardware support is designed to ensure efficient virtualization of interrupts.

### 2.17.1 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 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.

### 2.17.2 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.

### 2.17.3 TPR Access in 32-bit Mode

The mechanism for TPR virtualization described in Section 2.17.2 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 74.)

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.

### 2.17.4 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 2.9.1, 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 2.17.4, 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 2.17.5).

  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 INTAK cycle to the localAPIC).

- 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.

#### 2.17.5 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.

#### 2.17.6 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.

#### 2.17.7 Interrupt Masking in LocalAPIC

When guests have direct access to devices, interrupts arriving at the localAPIC 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

localAPIC, so they do not participate in the prioritization of other interrupts.

SVM introduces the following new APIC features:

- A 256-bit IER (interrupt enable) register is added to the localAPIC. 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's computation of the highest-priority pending interrupt.
- The VMM can issue specific end-of-interrupt (EOI) commands to the localAPIC, allowing the VMM to clear pending interrupts in any order, rather than always targeting the interrupt with highest-priority.

Software issues a specific EOI (SEOI) by writing the vector number of the interrupt to the new SEOI register in the localAPIC. The SEOI register is located at offset 420h in the APIC space. The SEOI register format is shown in Figure 2-5 below.

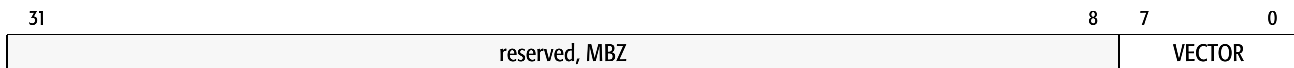


**Figure 2-5. Format of SEOI register (in localAPIC)**

The IER is made available to software by means of eight 32-bit registers in the localAPIC; bit *i* of the 256-bit IER is located at bit position (*i* mod 32) in the localAPIC register IER[*i* / 32]. The eight IER registers are located at offsets 480h, 490h, ...,4F0h in APIC space.

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).

### 2.17.8 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 FFFFFFF0h). 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. (See “Automatic Memory Clear” on page 69.)

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 E.1 on page 107). 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 2-5 on page 39 summarizes the handling of INITs.

**Table 2-5. INIT Handling in Different Operating Modes**

| GIF | INIT Intercept | INIT Redirect | Processor Response to INIT |
|---|---|---|---|
| 0 | x | x | Hold pending until GIF = 1. |
| 1 | 1 | x | #VMEXIT(INIT), INIT is still pending. |
| | 0 | 0 | Taken normally. |
| | | 1 | #SX, INIT is no longer pending. |

### 2.17.9 NMI Support

The VMM can intercept non-maskable interrupts (NMI) using a VMCB control bit (see Table 2-6). When intercepted, NMIs cause an exit from the guest and are held pending.

**Table 2-6. NMI Handling in Different Operating Modes**

| GIF | NMI Intercept | Processor Response to NMI |
|---|---|---|
| 0 | X | Hold pending until GIF=1. |
| 1 | 1 | #VMEXIT(NMI), NMI is still pending. |
| | 0 | Taken normally. |

## 2.18 SMM Support

This section describes SVM support for virtualization of System Management Mode (SMM).

### 2.18.1 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.

### 2.18.2 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 2-7.

**Table 2-7. SMI Handling in Different Operating Modes**

| GIF | Intercept SMI | Internal SMI | External SMI |
|---|---|---|---|
| 0 | x | Lost. | Hold pending until GIF=1. |
| 1 | 1 | Exit guest, code #VMEXIT(SMI), SMI is not pending. | #VMEXIT(SMI), SMI is still pending. |
| | 0 | Taken normally. | Taken normally. |

By intercepting SMIs, the VMM can gain control before the processor enters SMM.

### 2.18.3 Containerizing Platform SMM

In some usage scenarios, the VMM may not trust the existing platform SMM code. To address this case, 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.

A simple solution is for the VMM to create its own trusted SMM handler and to use the handler as a trampoline to invoke the platform SMM code inside a container. The main function of the trampoline code is to set up a guest and associated VMCB, and copy relevant state between the trampoline's SMM save area, and the guest's (virtual) SMM save area. 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 must be able to write the SMM_BASE MSR, as well as related SMM control registers. However, this action conflicts with any BIOS that attempts to lock SMM control registers.

A VMM can determine if it is running with a compatible BIOS setup by checking the SMMLOCK bit in the HWCR MSR (descibed in the appropriate BIOS and kernel developer's guide for your processor). If the bit is 1, the BIOS has locked the SMM control registers and the VMM will be unable to move them or insert its own SMM trampoline.

***Warning:*** *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. 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.*

**Advanced Support.** For more efficient and flexible operation, the new SMM_CTL MSR (described in more detail in Section E.3 on page 108) is designed to allow the VMM to control explicitly:

- when SMI is acknowledged or deasserted to the chipset,
- when SMM is considered active (i.e., SMRAM areas are mapped, NMIs and various other interrupts are blocked), and
- when the SMI-pending flag is cleared in the processor.

With this hardware support, the VMM can enter and exit SMM at will and the VMM code should be simplified.

*Note: Writes to the SMM_CTL MSR cause a #GP if the BIOS has locked the SMM control registers. Otherwise, SMM_CTL can be used to inspect the SMRAM areas at will, which risks revealing secrets that the BIOS might intend to hide.*

## 2.19 Last Branch Record Virtualization

The AMD64 debug control MSR (DebugCtl) provides the processor control of control-transfer recording and other debug tasks. (See "Debug and Performance Resources" in the *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*, order #24593, for more detailed information on these subjects.) 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—Holds the segment offset of the source instruction pointer (rIP).
- LastBranchToIP—Holds the segment offset of the target rIP.
- LastExceptionFromIP—Updated with the previous value of LastBranchFromIP during interrupts and exceptions (except #DB exceptions caused by debug breakpoint and ICEBP).

- LastExceptionToIP—Updated with the previous value of LastBranchToIP during interrupts and exceptions (except #DB exceptions caused by debug breakpoint and ICEBP).

Under SVM, the contents of the control-transfer recording MSRs must be exchanged between values tracked by host and and guest. This is done by activating LBR virtualization in the guest VMCB control area.

#### 2.19.1 Enabling LBR Virualization

Setting the LBR_VIRTUALIZATION_ENABLE bit to 1 in the VMCB control area enables LBR virtualization. When LBR virtualization is enabled, the VMM stores an image of the DebugCtl MSR and of each of the pointers stored in the control-transfer recording MSRs in four fields in the VMCB state save area.

- 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 LastExceptionToIP MSR.
- LASTEXCPTO—Holds the guest value of the LastExceptionFromIP MSR.

#### 2.19.2 Host and Guest LBR Virtualization

When VMCB.LBR_VIRTUALIZATION_ENABLE[0] 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.

#### 2.19.3 LBR Virtualization CPUID Feature Detection

EDX bit 1 as returned by CPUID extended function 8000_000Ah reports the LBR virtualization feature on AMD64 processors.

## 2.20 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).

### 2.20.1 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.

### 2.20.2 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 48). 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.

*Note: 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 (see "DEV_CR Register" on page 49); 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.

### 2.20.3 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 2-6 on page 46). 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 49). 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.

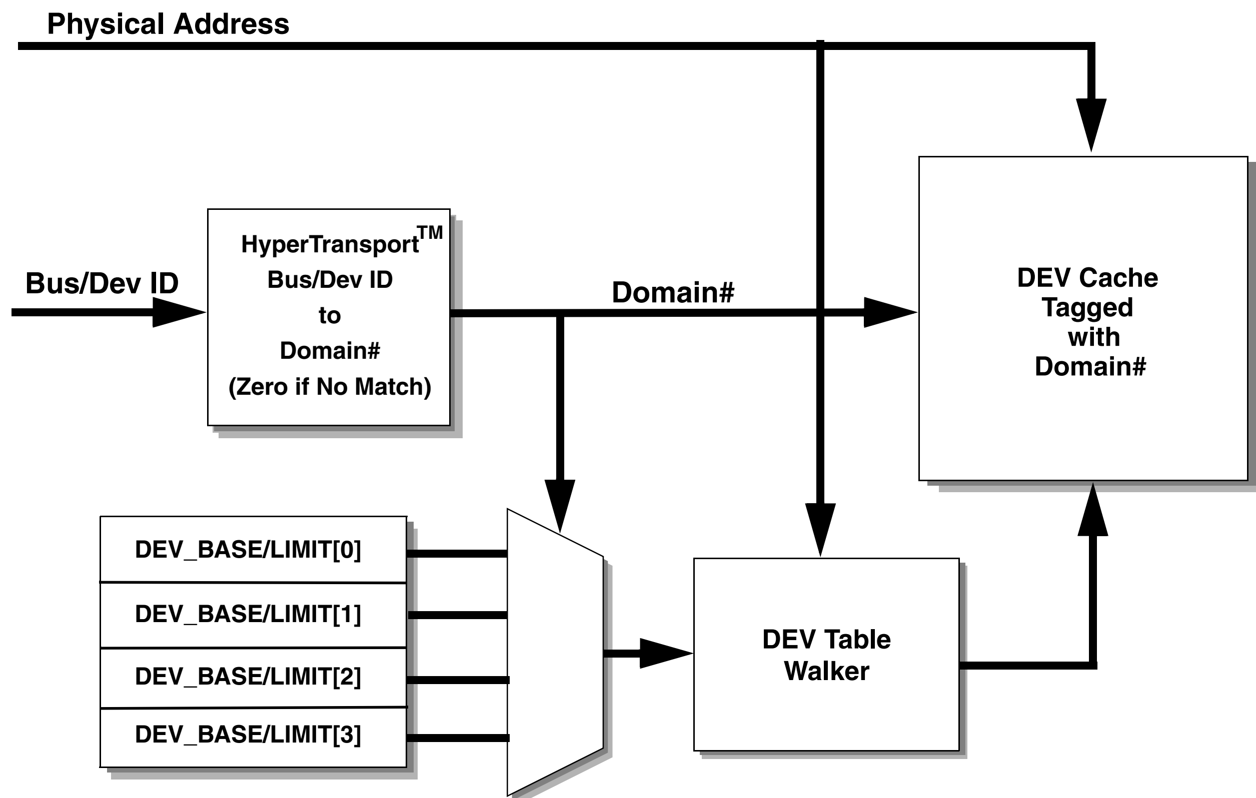


**Figure 2-6. Host Bridge DMA Checking**

### 2.20.4 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 facility.

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 2-8. DEV Capability Block, Overall Layout**

| Byte Offset | Register | Comments |
|---|---|---|
| 0 | DEV_HDR | Capability block header |
| 4 | DEV_OP | Selects control/status register to access |
| 8 | DEV_DATA | Read/write to access register selected in DEV_OP |

**DEV Capability Header.** The DEV capability header (DEV_HDR) is defined as follows

**Table 2-9. DEV Capability Header (DEV_HDR) (in PCI Config Space)**

| Bit(s) | Definition |
|---|---|
| 31–22 | Reserved, MBZ |
| 21 | Interrupt Reporting Capability |
| 20 | Machine Check Exception Reporting Capability |
| 19 | Reserved, MBZ |
| 18–16 | DEV Capability Block Type; hardwired to 010b. |
| 15–8 | PCI Capability pointer; points to next capability in list |
| 7–0 | PCI Capability ID; hardwired to 0x0F |

### 2.20.5 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 2-7 shows the format of the DEV_OP register. The DEV_DATA register reflects the format of the DEV register selected in DEV_OP.

**Figure 2-7. 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 2-10; 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 2-10. Encoding of function field in DEV_OP register**

| Function Code | RegisterType | Number of Instances |
|---|---|---|
| 0 | DEV_BASE_LO | multiple |
| 1 | DEV_BASE_HI | multiple |
| 2 | DEV_MAP | multiple |
| 3 | DEV_CAP | single |
| 4 | DEV_CR | single |
| 5 | DEV_ERR_STATUS | single |
| 6 | DEV_ERR_ADDR_LO | single |
| 7 | DEV_ERR_ADDR_HI | single |

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 46.

#### 2.20.6 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.

**Figure 2-8. Format of DEV_CAP Register (in PCI Config Space)**

**DEV_CR Register.** This is the main control register for the DEV mechanism; it is cleared to zero by RESET.

**Table 2-11. DEV_CR Control Register**

| Bit(s) | Definition |
|---|---|
| 31-7 | reserved, MBZ |
| 6 | DEV table walk probe disable.<br>0 = Use probe on DEV walk; 1 = Do not use probe |
| 5 | SL_DEV_EN. Enable bit for limited memory protection, see Section 2.20.8 on page 51. 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.<br>0 = Do not generate MCE; 1 = Generate MCE on errors. |
| 2 | I/O space protection enable (IOSPEN)<br>0 = Allow upstream I/O cycles; 1 = Block. |
| 1 | Memory clear disable. If non-zero, memory-clearing on reset is disabled.<br>This bit is not writable until the memory is enabled. |
| 0 | DEV global enable bit. If zero, DEV protection is turned off. |

**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 2-9 and 2-10 on page 50.

**Figure 2-9. Format of DEV_BASE_HI[n] Registers**

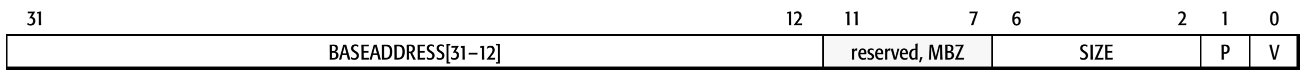


**Figure 2-10. Format of DEV_BASE_LO[n] Registers**

Fields of the DEV_BASE_HI and DEV_BASE_LO registers are defined as follows:

- V (valid)—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.
- P (protect)—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 4GB * $2^{size}$. 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 2-11.

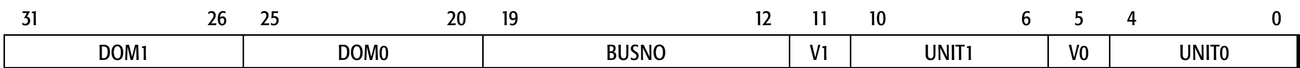


**Figure 2-11. Format of DEV_MAP[n] Registers**

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.

#### 2.20.7 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.

#### 2.20.8 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 3.1.6 on page 65 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.

## 2.21 Nested Paging Facility

The optional SVM nested paging facility provides for two levels of address translation, thus eliminating the need for the VMM to maintain shadow page tables.

### 2.21.1 Traditional Paging versus Nested Paging

Figure 2-12 shows how a page in the virtual 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.

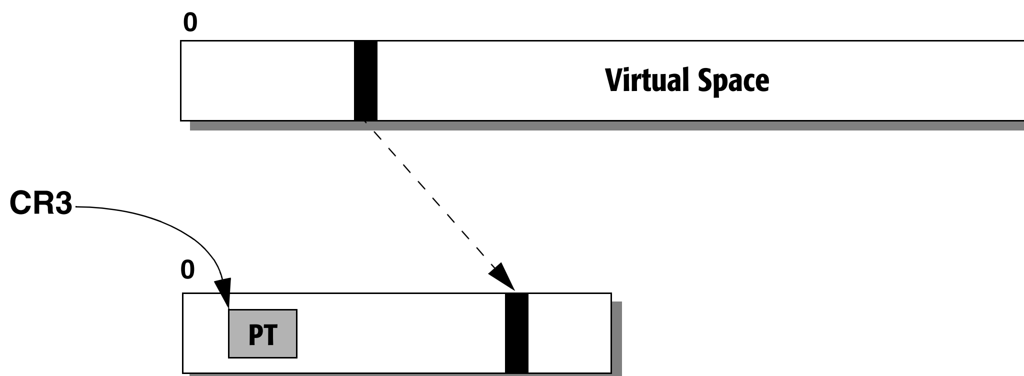


**Figure 2-12. Address Translation with Traditional Paging**

With nested paging enabled, *two* levels of address translation are applied; refer to Figure 2-13 below.

- Both guest and host levels have their own copy of CR3, referred to as gCR3 and hCR3, respectively.
- Guest page tables (gPT) map guest virtual addresses to guest physical addresses. The guest page tables are in guest physical memory, and are pointed to by gCR3.
- Host page tables (hPT) map guest physical addresses to host physical addresses. The host page tables are in host physical memory, and are pointed to by hCR3.
- The most-recently used translations from guest virtual to host 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 host physical

addresses. Hence, before accessing a guest page table entry, the table walker first translates that entry's guest physical address into a host physical address.

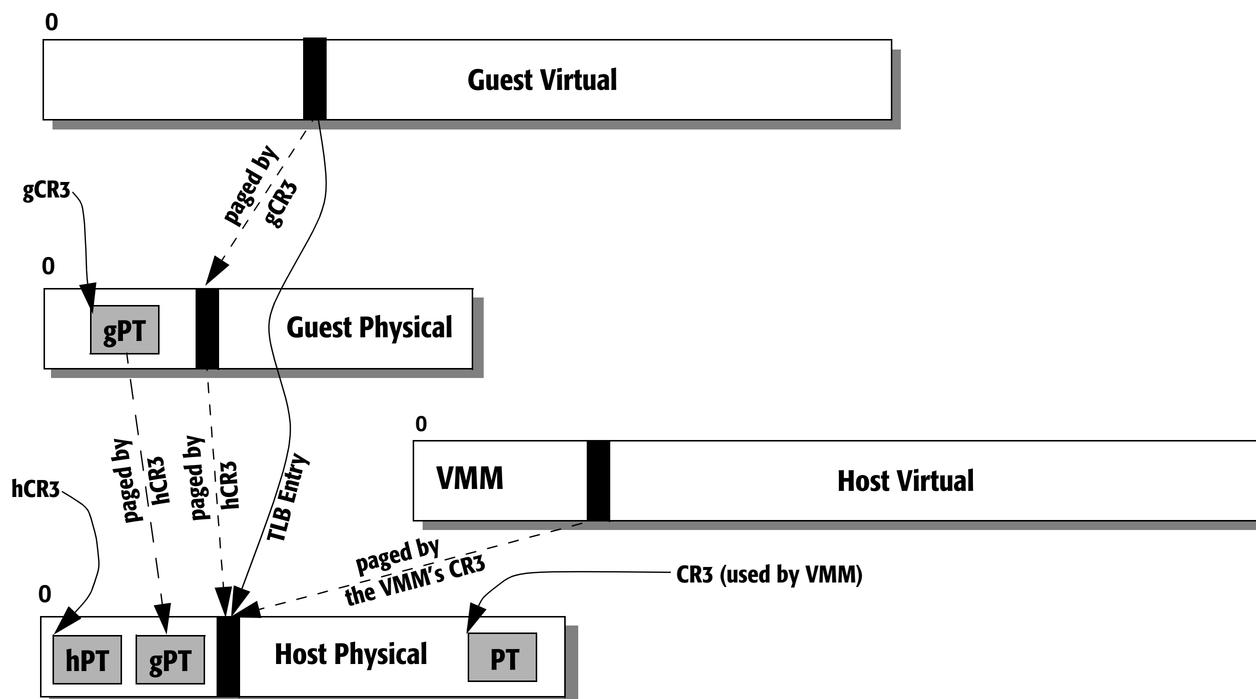


**Figure 2-13. Address Translation with Nested Paging**

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).

### 2.21.2 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 CR*n*, 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 host 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.

#### 2.21.3 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. If VMRUN is executed with hCR0.PG cleared to zero and NP_ENABLE set to 1, VMRUN terminates with #VMEXIT(VMEXIT_INVALID).

#### 2.21.4 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 hCR3, the version of CR3 to be used while the nested-paging guest is running, from the H_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 #VMEXIT occurs with nested paging enabled:

- #VMEXIT writes the guest paging state (gCR3, gCR0, etc) back into the VMCB. hCR3 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.

#### 2.21.5 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 the guest's page tables. These guest physical addresses must be translated to host physical addresses using the host page tables. Host-

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 virtual 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 host page tables to find the host physical address.

Table walks for guest page tables are always treated as user writes at the host level. For this reason,

- the page must be writable by user at the host level, or else a #VMEXIT(NPF) is raised, and
- the dirty and accessed bits are always set in the host 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 host 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 host page tables. In this case, the final guest-physical address is equal to the guest-virtual address, and is still translated in the host page tables.

#### 2.21.6 **Host versus Guest Page Faults, Fault Ordering**

In nested paging, page faults can be raised at either the guest or host level. Nested walks proceed in the following order; faults are generated in the same order:

1. Walk the guest page table entries in the host page tables. Host dirty/accessed bits are set as needed in the host page tables. Any host page 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 host physical using the host page table; any fault during this translation results in a #VMEXIT(NPF).

Host faults are entirely a function of the host page tables and VMM processor mode. Host 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 errorcode:

- Bit 0 (P)—cleared to 0 if the host page was not present, 1 otherwise
- Bit 1 (RW)—set to 1 if the host-level access was a write. Note that host table walks for guest page tables are always treated as data writes.
- Bit 2 (US)—always 1, since all guest accesses are treated as user accesses at the host level
- Bit 3 (RSV)—set to 1 if reserved bits were set in the corresponding host page table entry
- Bit 4 (ID)—set to 1 if the host-level access was a code read. Note that host table walks for guest page tables are always treated as data writes, even if the access itself is a code read

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.

### 2.21.7 Combining Host and Guest Attributes

Any access to guest physical memory is subjected to a host permission check by examining the host's mapping of the host virtual address corresponding to the guest physical address under consideration.

A page is considered *writable* by the guest only if it is marked writable at both the guest and host 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 host page tables. 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 host 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 host pages are executable at the host 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 host 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 user at the guest level. The page must be marked user in the host to allow any guest access at all.

#### 2.21.8 **Combining Memory Types, MTRRs**

The processor combines guest and host memory types; registers that affect memory types include:

- The PCD/PWT/PAT*i* bits in the host and guest page table entries.
- The PCD/PWT bits in the host and guest CR3 registers.
- The guest PAT type (obtained by appropriately indexing the gPAT register).
- The host PAT type (obtained by appropriately indexing the hPAT register).
- The MTRRs (which are referenced based only on host 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 host page tables. Note that the MTRRs are only applied to host physical addresses.

The rules for combining memory types when constructing a guest TLB entry are:

- Host and guest PAT types are combined according to Table 2-12 on page 59, producing a “combined PAT type”
- the combined PAT type is further combined with the MTRR type according to Table 2-13 on page 59, where the relevant MTRRs are determined by the host 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 uncacheable 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 host and guest memory types that are incompatible with respect to caching, the WC+ memory type is used instead of WC (and Table 2-13 on page 59 ensures that the snooping behavior is retained regardless of the host MTRR settings). Refer to Table 2-12 (below) for details.

Table 2-12 (below) shows how guest and host PAT types are combined into an effective PAT type. When interpreting this table, recall that the intent is for the VMM to use its PAT type to simulate guest MTRRs.

**Table 2-12. Combining Guest and Host PAT Types**

| Guest PAT Type | | Host PAT Type | | | | | |
|---|---|---|---|---|---|---|---|
| | | **UC** | **UC–** | **WC** | **WP** | **WT** | **WB** |
| | **UC** | UC | UC | UC | UC | UC | UC |
| | **UC–** | UC | UC | WC | UC | UC | UC |
| | **WC** | WC | WC | WC | WC+ | WC+ | WC+ |
| | **WP** | UC | UC | UC | WP | UC | WP |
| | **WT** | UC | UC | UC | UC | WT | WT |
| | **WB** | UC | UC | WC | WP | WT | WB |

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 2.

**Table 2-13. Combining PAT and MTRR Types**

| Effective PAT Type | | MTRR Type | | | | |
|---|---|---|---|---|---|---|
| | | **UC** | **WC** | **WP** | **WT** | **WB** |
| | **UC** | UC | CD | CD | CD | CD |
| | **UC–** | UC | WC | CD | CD | CD |
| | **WC** | WC | WC | WC | WC | WC |
| | **WC+** | WC | WC | WC+ | WC+ | WC+ |
| | **WP** | UC | CD | WP | CD | WP |
| | **WT** | UC | CD | CD | WT | WT |
| | **WB** | UC | WC | WP | WT | WB |

### 2.21.9 Page Splintering

When an address is mapped by guest and host page table entries with different different page sizes, the TLB entry that is created matches the size of the smaller page.

### 2.21.10 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.

#### 2.21.11 A20 Masking

There is no provision for applying A20 masking to guest physical addresses; the VMM can emulate A20 masking by changing the host-level page mappings accordingly.

#### 2.21.12 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 Appendix B on page 91 for the details of processor feature identification and support).

# 3 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.

**Automatic Memory Clearing.** Automatic clearing of memory upon reset protects secrets stored in system memory from simple reset-based attacks.

**Security Exception.** A new security exception (#SX) is used to signal certain security-critical events.

## 3.1 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 3.1.4.)

### 3.1.1 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 or VMM (referred to as a Security Kernel, or SK, in this document), after first having validated the identity of that software.

One of the main features of 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 3.1.2.

### 3.1.2 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 “Secure Loader Block” on page 62). 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 “SKINIT Operation” on page 65 for further details of this transfer). The SL image for which the hash is computed must be ready to execute without prior manipulation.

### 3.1.3 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 3-1 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.

t

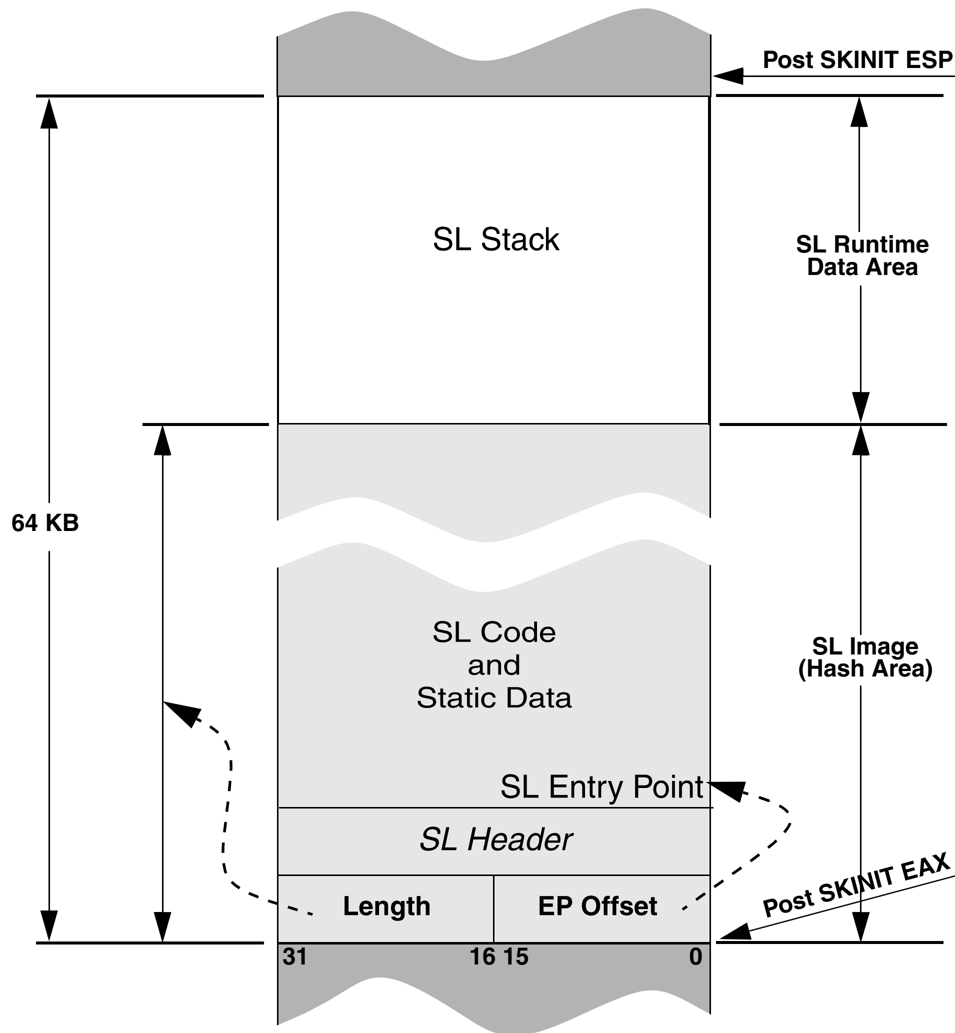


**Figure 3-1. SLB Example Layout**

#### 3.1.4 Trusted Platform Module

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.

#### 3.1.5 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 should allow the TPM to reliably detect true execution, as opposed to emulation, of a trusted Secure Loader, which in turn provides a reliable means for verifying the subsequent loading and startup of a trusted Security Kernel.

#### 3.1.6 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 and SS selectors are set to 0008h and 0010h respectively, and CS and SS base, limit and attribute registers are set to (base = 0, limit = 4G, CS:read-only, SS:read/write, expand-up). 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 “Secure Initialization Support” on page 51) 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 inter-processor handshake as described in Section 3.1.8 on page 67.
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 such as HDT that could subvert security. A debug version of the SL can reenable those features by clearing the VM_CR.DPD flag immediately upon entry.

### 3.1.7 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.

### 3.1.8 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 inter-processor interrupt (IPI), which puts the target processors into a halted 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 arrange for any processor-specific system register contents to be saved to memory (to be restored after the APs undergo hardware re-initialization), and for all APs to be idled using whatever software means is appropriate (for example, by means of an OS kernel function or driver threads running on the other processors). Once the driver has confirmed that all APs are idle, it must issue an INIT IPI to all APs and wait for its localAPIC Busy indication to clear. This places the APs into a halted state which is responsive only to a subsequent Startup IPI (although the APs will still respond

to snoops for cache coherency). The driver may execute SKINIT 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.

**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.

## 3.2 Automatic Memory Clear

Automatic memory clear (AMC) erases the contents of sytem memory after the processor is subjected to a cold reset, and under controlled circumstances after a warm reset.

The processor shadows the AMC Check registers (the northbridge registers that configure the DRAM size and configuration), for use after the next warm reset. The shadow copies are updated each time the DRAM controller completes initialization.

The memory clear operates as follows:

- Memory is cleared after warm reset, when DRAM access is first enabled, if either of these conditions is true
  - AMC was not disabled in the northbridge (MemClrDis = 0), or
  - the new value of the DRAM configuration registers do not match the shadowed AMC Check registers.
- Once the memory clear starts, it continues through completion (unless interrupted by a reset).
- The range of DRAM cleared is the entire memory that was enabled the *previous time* DRAM was enabled. This configuration can be determined from the shadow registers.
- After the memory clear ends, the new AMC Check register values are shadowed, for use after the next warm reset.

After trusted software has taken steps to ensure that any secrets in system memory have been removed or encrypted, trusted software is expected to set MemClrDis before entering the ACPI-defined S3 state (suspend to RAM).

Refer to the AMD *BIOS and Kernel Developer's Guide* for your processor for details on the relevant register definitions.

## 3.3 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.

# 4 SVM Instruction Set Reference

AMD virtualization technology introduces several new instructions and modifies several existing instructions to facilitate the implementation of VMM systems.

The SVM instruction set includes instructions to:

- Start execution of a guest (VMRUN)
- Save and restore subsets of processor state (VMSAVE, VMLOAD)
- Allow guests to explicitly communicate with the VMM (VMMCALL)
- Set and clear the global interrupt flag (STGI, CLGI)
- Invalidate TLB entries in a specified ASID (INVLPGA)
- Read and write CR8 in all processor modes
- Secure init and control transfer with attestation (SKINIT)

Enabling SVM also affects the behavior of existing AMD64 instructions.

## 4.1 Changes to RSM Instruction

RSM is not allowed to change EFER.SVME. Attempts to do so are ignored.

When EFER.SVME is 1, RSM reloads the four PDPEs (through the incoming CR3) when returning to a mode that has legacy PAE mode paging enabled.

When EFER.SVME is 1, the RSM instruction is permitted to return to paged real mode (i.e., CR0.PE=0 and CR0.PG=1).

## 4.2 New Instructions

The basic operation of each SVM instruction is given in the pages that follow.

# CLGI Clear Global Interrupt Flag

Clears the global interrupt flag (GIF). While GIF is zero, all external interrupts are disabled.

| Mnemonic | Opcode | Description |
|---|---|---|
| CLGI | 0F 01 DD | Clears the global interupt flag (GIF). |

### Related Instructions

STGI

### rFLAGS Affected

None.

### Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---|---|---|---|---|
| Invalid opcode, #UD | | | X | The SVM instructions are not supported as indicated by ECX bit 2 as returned by CPUID extended function 8000_0001h. |
| | | | X | EFER.SVME was zero. |
| | X | X | | This instruction is only recognized in protected mode. |
| General protection, #GP | | | X | CPL was not zero. |

# INVLPGA Invalidate TLB Entry in a Specified ASID

Invalidates the TLB mapping for a given virtual page and a given ASID. The virtual address is specified in the implicit register operand rAX. The portion of RAX used to form the address is determined by the effective address size. The ASID is taken from ECX.

The INVLPGA instruction may invalidate any number of additional TLB entries, in addition to the targeted entry.

The INVLPGA instruction is a serializing instruction and a privileged instruction. The current privilege level must be 0 to execute this instruction.

| Mnemonic | Opcode | Description |
|---|---|---|
| INVLPGA rAX, ECX | 0F 01 DF | Invalidates the TLB mapping for the virtual page specified in rAX and the ASID specified in ECX. |

## Related Instructions

INVLPG.

## rFLAGS Affected

None.

## Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---|---|---|---|---|
| Invalid opcode, #UD | | | X | The SVM instructions are not supported as indicated by ECX bit 2 as returned by CPUID extended function 8000_0001h. |
| | | | X | EFER.SVME was zero. |
| | X | X | | This instruction is only recognized in protected mode. |
| General protection, #GP | | | X | CPL was not zero. |

## MOV (CR*n*) Move to/from Control Registers

Moves the contents of a 32-bit or 64-bit general-purpose register to a control register or vice versa.

In 64-bit mode, the operand size is fixed at 64 bits without the need for a REX prefix. In non-64-bit mode, the operand size is fixed at 32 bits and the upper 32 bits of the destination are forced to 0.

CR0 maintains the state of various control bits. CR2 and CR3 are used for page translation. CR4 holds various feature enable bits. CR8 is used to prioritize external interrupts. CR1, CR5, CR6, CR7, and CR9 through CR15 are all reserved and raise an undefined opcode exception (#UD) if referenced.

CR8 can also be read and modified using the task priority register described in "System-Control Registers" in Volume 2.

CR8 can be read and written in 64-bit mode, using a REX prefix. CR8 can be read and written in legacy mode using the MOV (CR*n*) opcode, using a LOCK prefix instead of a REX prefix to specify the additional opcode bit. To verify whether the LOCK prefix can be used in this way, check the status of ECX bit 4 returned by CPUID standard function 80000001h.

This instruction is always treated as a register-to-register (MOD = 11) instruction, regardless of the encoding of the MOD field in the MODR/M byte.

MOV(CRn) is a privileged instruction and must always be executed at CPL = 0.

MOV (CRn) is a serializing instruction.

| Mnemonic | Opcode | Description |
|---|---|---|
| MOV CR*n*, *reg32* | 0F 22 /r | Move the contents of a 32-bit register to CR*n* |
| MOV CR*n*, *reg64* | 0F 22 /r | Move the contents of a 64-bit register to CR*n* |
| MOV *reg32*,CR*n* | 0F 20 /r | Move the contents of CR*n* to a 32-bit register. |
| MOV *reg64*,CR*n* | 0F 20 /r | Move the contents of CR*n* to a 64-bit register. |
| MOV CR8, *reg32* | F0 0F 22/r | Move the contents of a 32-bit register to CR8. |
| MOV CR8, *reg64* | F0 0F 22/r | Move the contents of a 64-bit register to CR8. |

| | | |
|---|---|---|
| MOV reg32, CR8 | F0 0F 20/r | Move the contents of CR8 into a 32-bit register. |
| MOV reg64, CR8 | F0 0F 20/r | Mov the contents of CR8 into a 64-bit register. |

### Related Instructions

CLTS, LMSW, SMSW

### rFLAGS Affected

None

### Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---|---|---|---|---|
| Invalid Instruction, #UD | X | X | X | An illegal control register was referenced (CR1, CR5–CR7, CR9–CR15). |
| | X | X | X | The use of the LOCK prefix to read CR8 in legacy mode is not supported, as indicated by ECX bit 4 as returned by CPUID standard function 8000_0001h. |
| General protection, #GP | | X | X | CPL was not 0. |
| | X | | X | An attempt was made to set CR0.PG = 1 and CR0.PE = 0. |
| | X | | X | An attempt was made to set CR0.CD = 0 and CR0.NW = 1. |
| | X | | X | Reserved bits were set in the page-directory pointers table (used in the legacy extended physical addressing mode) and the instruction modified CR0, CR3, or CR4. |
| | X | | X | An attempt was made to write 1 to any reserved bit in CR0, CR3, CR4 or CR8. |
| | X | | X | An attempt was made to set CR0.PG while long mode was enabled (EFER.LME = 1), but paging address extensions were disabled (CR4.PAE = 0). |
| | | | X | An attempt was made to clear CR4.PAE while long mode was active (EFER.LMA = 1). |

## SKINIT Secure Init and Jump with Attestation

Securely reinitializes the cpu, allowing for the startup of trusted software (such as a VMM). The code to be executed after reinitialization can be verified based on a secure hash comparison. SKINIT takes the physical base address of the SLB as its only input operand, in EAX. The SLB must be structured as described in "Secure Loader Block" on page 62, and is assumed to contain the code for a Secure Loader (SL).

| Mnemonic | Opcode | Description |
|---|---|---|
| SKINIT EAX | 0F 01 DE | Secure initialization and jump, with attestation. |

### Action

```
Initialize processor state as for an INIT signal
CR0.PE = 1

CS.sel = 0x0008
CS.attr = 32-bit code, read/execute
CS.base = 0
CS.limit = 0xFFFFFFFF

SS.sel = 0x0010
SS.attr = 32-bit stack, read/write, expand up
SS.base = 0
SS.limit = 0xFFFFFFFF

EAX = EAX & 0xFFFF0000 // Form SLB base address.
EDX = family/model/stepping
ESP = EAX + 0x00010000  // Initial SL stack.
Clear GPRs other than EAX, EDX, ESP

EFER = 0
VM_CR.DPD = 1
VM_CR.R_INIT = 1
VM_CR.DIS_A20M = 1

Enable SL_DEV, to protect 64Kbyte of physical memory starting at
the physical address in EAX

GIF = 0

Read the SL length from offset 0x0002 in the SLB
Copy the SL image to the TPM for attestation

Read the SL entrypoint offset from offset 0x0000 in the SLB
Jump to the SL entrypoint, at EIP = EAX+entrypoint offset
```

### Related Instructions

None.

### rFLAGS Affected

| ID | VIP | VIF | AC | VM | RF | NT | IOPL | OF | DF | IF | TF | SF | ZF | AF | PF | CF |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 21 | 20 | 19 | 18 | 17 | 16 | 14 | 13–12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 | 2 | 0 |

*Note: Bits 31–22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.*

### Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---|---|---|---|---|
| Invalid opcode, #UD | | | X | The SVM instructions are not supported as indicated by ECX bit 2 as returned by CPUID extended function 8000_0001h. |
| | | | X | EFER.SVME was zero. |
| | X | X | | This instruction is only recognized in protected mode. |
| General protection, #GP | | | X | CPL was not zero. |

# STGI Set Global Interrupt Flag

Sets the global interrupt flag (GIF) to 1. While GIF is zero, all external interrupts are disabled.

| Mnemonic | Opcode | Description |
|---|---|---|
| STGI | 0F 01 DC | Sets the global interupt flag (GIF). |

## Related Instructions

CLGI

## rFLAGS Affected

None.

## Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---|---|---|---|---|
| Invalid opcode, #UD | | | X | The SVM instructions are not supported as indicated by ECX bit 2 as returned by CPUID extended function 8000_0001h. |
| | | | X | EFER.SVME was zero. |
| | X | X | | This instruction is only recognized in protected mode. |
| General protection, #GP | | | X | CPL was not zero. |

# VMLOAD Load State from VMCB

Loads a subset of processor state from the VMCB specified by the physical address in the rAX register. The portion of RAX used to form the address is determined by the effective address size.

The VMSAVE and VMLOAD instructions complement the state save/restore abilities of VMRUN and #VMEXIT, providing access to hidden state that software is otherwise unable to access, plus some additional commonly-used state.

| Mnemonic | Opcode | Description |
|---|---|---|
| VMLOAD rAX | 0F 01 DA | Load additional state from VMCB. |

## Action

```
Load from a VMCB at physical address rAX:
      FS, GS, TR, LDTR (including all hidden state)
      KernelGsBase
      STAR, LSTAR, CSTAR, SFMASK
      SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP
```

## Related Instructions

VMSAVE

## rFLAGS Affected

None.

## Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---|---|---|---|---|
| Invalid opcode, #UD | | | X | The SVM instructions are not supported as indicated by ECX bit 2 as returned by CPUID extended function 8000_0001h. |
| | | | X | EFER.SVME was zero. |
| | X | X | | This instruction is only recognized in protected mode. |
| General protection, #GP | | | X | CPL was not zero. |
| | | | X | rAX referenced a physical address above the maximum supported physical address. |
| | | | X | The address in rAX was not aligned on a 4Kbyte boundary. |

# VMMCALL Call VMM

Provides a mechanism for a guest to explicitly communicate with the VMM by generating a #VMEXIT.

A non-intercepted VMMCALL unconditionally raises a #UD exception.

VMMCALL is not restricted to either protected mode or CPL zero.

| Mnemonic | Opcode | Description |
|---|---|---|
| VMMCALL | 0F 01 D9 | Explicit communication with the VMM. |

### Related Instructions

None.

### rFLAGS Affected

None.

### Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---|---|---|---|---|
| Invalid opcode, #UD | X | X | X | The SVM instructions are not supported as indicated by ECX bit 2 as returned by CPUID extended function 8000_0001h. |
| | X | X | X | EFER.SVME was zero. |
| | X | X | X | VMMCALL was not intercepted. |

# VMRUN Run Virtual Machine

Starts execution of a guest instruction stream. The physical address of the *virtual machine control block* (VMCB) describing the guest is taken from the rAX register (the portion of RAX used to form the address is determined by the effective address size).

VMRUN saves a subset of host processor state to the host state-save area specified by the physical address in the VM_HSAVE_PA MSR. VMRUN then loads guest processor state (and control information) from the VMCB at the physical address specified in rAX. The processor then executes guest instructions until one of several *intercept* events (specified in the VMCB) is triggered. When an intercept event occurs, the processor stores a snapshot of the guest state back into the VMCB, reloads the host state, and continues execution of host code at the instruction following the VMRUN instruction.

| Mnemonic | Opcode | Description |
|---|---|---|
| VMRUN rAX | 0F 01 D8 | Performs a world-switch to guest. |

## Action

```
if (intercepted(VMRUN))
    #VMEXIT (VMRUN)
remember VMCB address (delivered in rAX) for next #VMEXIT
save host state to physical memory indicated in the VM_HSAVE_PA MSR:
      ES.sel
      CS.sel
      SS.sel
      DS.sel
      GDTR.{base,limit}
      IDTR.{base,limit}
      EFER
      CR0
      CR4
      CR3
      // host CR2 is not saved
      RFLAGS
      RIP
      RSP
      RAX

from the VMCB at physical address rAX, load control information:
      intercept vector
      TSC_OFFSET
      interrupt control (v_irq, v_intr_*, v_tpr)
      EVENTINJ field
if (nested paging supported)
```

```
    NP_ENABLE
    if (NP_ENABLE = 1)
       hCR3
ASID

from the VMCB at physical address rAX, load guest state:
      ES.{base,limit,attr,sel}
      CS.{base,limit,attr,sel}
      SS.{base,limit,attr,sel}
      DS.{base,limit,attr,sel}
      GDTR.{base,limit}
      IDTR.{base,limit}
      EFER
      CR0
      CR4
      CR3
      CR2
      if (NP_ENABLE = 1)
          gPAT          // leaves host hPAT register unchanged
      RFLAGS
      RIP
      RSP
      RAX
      DR7
      DR6
      CPL              // 0 for real mode, 3 for v86 mode, else as loaded
      INTERRUPT_SHADOW

if (LBR virtualization supported)
    LBR_VIRTUALIZATION_ENABLE
    if (LBR_VIRTUALIZATION_ENABLE=1)
        save LBR state to the host save area
             DBGCTL
             BR_FROM
             BR_TO
             LASTEXCP_FROM
             LASTEXCP_TO
        load LBR state from the VMCB
             DBGCTL
             BR_FROM
             BR_TO
             LASTEXCP_FROM
             LASTEXCP_TO

if (guest state consistency checks fail)
    #VMEXIT(INVALID)

Execute command stored in TLB_CONTROL.

GIF = 1           // allow interrupts in the guest
if (EVENTINJ.V)
```

```
      cause exception/interrupt in guest
else
      jump to first guest instruction
```

**Upon #VMEXIT, the processor performs the following actions in order to return to the host execution context:**

```
GIF = 0
save guest state to VMCB:
      ES.{base,limit,attr,sel}
      CS.{base,limit,attr,sel}
      SS.{base,limit,attr,sel}
      DS.{base,limit,attr,sel}
      GDTR.{base,limit}
      IDTR.{base,limit}
      EFER
      CR4
      CR3
      CR2
      CR0
      if (nested paging enabled)
          gPAT
      RFLAGS
      RIP
      RSP
      RAX
      DR7
      DR6
      CPL
      INTERRUPT_SHADOW
save additional state and intercept information:
      V_IRQ, V_TPR
      EXITCODE
      EXITINFO1
      EXITINFO2
      EXITINTINFO
clear EVENTINJ field in VMCB

prepare for host mode by clearing internal processor state bits:
      clear intercepts
      clear v_irq
      clear v_intr_masking
      clear tsc_offset
      disable nested paging
      clear ASID to zero

reload host state
      GDTR.{base,limit}
      IDTR.{base,limit}
      EFER
      CR0
      CR0.PE = 1  // saved copy of CR0.PE is ignored
      CR4
      CR3
      if (host is in PAE paging mode)
           reloaded host PDPEs
```

```
      // Do not reload host CR2 or PAT
      RFLAGS
      RIP
      RSP
      RAX
      DR7 = "all disabled"
      CPL = 0
      ES.sel; reload segment descriptor from GDT
      CS.sel; reload segment descriptor from GDT
      SS.sel; reload segment descriptor from GDT
      DS.sel; reload segment descriptor from GDT

if (LBR virtualization supported)
    LBR_VIRTUALIZATION_ENABLE
    if (LBR_VIRTUALIZATION_ENABLE=1)
        save LBR state to the VMCB:
            DBGCTL
            BR_FROM
            BR_TO
            LASTEXCP_FROM
            LASTEXCP_TO
        load LBR state from the host save area:
            DBGCTL
            BR_FROM
            BR_TO
            LASTEXCP_FROM
            LASTEXCP_TO

if (illegal host state loaded, or exception while loading host state)
      shutdown
else
      execute first host instruction following the VMRUN
```

### Related Instructions

VMLOAD, VMSAVE.

### rFLAGS Affected

None.

### Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---|---|---|---|---|
| Invalid opcode, #UD | X | X | X | The SVM instructions are not supported as indicated by ECX bit 2 as returned by CPUID extended function 8000_0001h. |
| | | | X | EFER.SVME was zero. |
| | X | X | | This instruction is only recognized in protected mode. |
| General protection, #GP | | | X | CPL was not zero. |
| | | | X | rAX referenced a physical address above the maximum supported physical address. |
| | | | X | The address in rAX was not aligned on a 4Kbyte boundary. |

# VMSAVE Save State to VMCB

Stores a subset of the processor state into the VMCB specified by the physical address in the rAX register (the portion of RAX used to form the address is determined by the effective address size).

The VMSAVE and VMLOAD instructions complement the state save/restore abilities of VMRUN and #VMEXIT, providing access to hidden state that software is otherwise unable to access, plus some additional commonly-used state.

| Mnemonic | Opcode | Description |
|---|---|---|
| VMSAVE rAX | 0F 01 DB | Save additional guest state to VMCB. |

## Action

```
Store to a VMCB at physical address rAX:
      FS, GS, TR, LDTR (including all hidden state)
      KernelGsBase
      STAR, LSTAR, CSTAR, SFMASK
      SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP
```

## Related Instructions

VMLOAD

## rFLAGS Affected

None.

## Exceptions

| Exception | Real | Virtual 8086 | Protected | Cause of Exception |
|---|---|---|---|---|
| Invalid opcode, #UD | | | X | The SVM instructions are not supported as indicated by ECX bit 2 as returned by CPUID extended function 8000_0001h. |
| | | | X | EFER.SVME was zero. |
| | X | X | | This instruction is only recognized in protected mode. |
| General protection, #GP | | | X | CPL was not zero. |
| | | | X | rAX referenced a physical address above the maximum supported physical address. |
| | | | X | The address in rAX was not aligned on a 4Kbyte boundary. |

# Appendix A Reset Values and INIT

This appendix provides data on reset values of SVM-related data structures and features, and the reinitialization of state as a result of INIT.

## A.1 Reset Values

The SVM-related processor state resets as follows:

- EFER.SVME is cleared to 0 (SVM extensions disabled).
- GIF is set to 1 (interrupts enabled globally).
- SVM intercepts are cleared to 0 (no intercepts active).
- The “current ASID” register is cleared to 0.
- VM_CR is cleared to 0 (debug, INIT and A20M function as usual).
- V_IRQ and V_INTR_MASKING are cleared to 0 (no virtual interrupt pending, interrupt masking not virtualized).
- TSC_OFFSET is cleared to 0 (RDTSC delivers “raw” value).
- Nested paging is disabled.

SVM-related Northbridge state resets as follows:

- DEV table features are disabled.
- Interrupt Enable Register (IER) is set to “all vectors enabled”.

## A.2 Action of INIT

INIT can be intercepted when inside a guest (in which case it causes a #VMEXIT and INIT is held pending) and can be redirected inside the host context, in which case it causes INIT to be dropped and raises an #SX exception. In either case, the INIT has no effect on hardware state. Only if the INIT is neither intercepted nor redirected does it reinitialize state as follows:

- EFER.SVME is cleared to 0 (SVM extensions disabled).
- GIF is set to 1 (interrupts enabled globally).
- SVM intercepts are cleared to 0 (no intercepts active).
- The “current ASID” register is cleared to 0.

- VM_CR is cleared to 0 (debug, INIT and A20M function as usual).
- V_IRQ and V_INTR_MASKING are cleared to 0 (no virtual interrupt pending, interrupt masking not virtualized).
- TSC_OFFSET is cleared to 0 (RDTSC delivers “raw” value).
- Nested paging is disabled.

SVM-related Northbridge state is initialized as follows:

- DEV table features are disabled.
- Interrupt Enable Register (IER) is set to “all vectors enabled”.

# Appendix B Processor Feature Identification

The presence of the SVM extensions is indicated by the SVM feature flag in the extended feature flags returned by extended CPUID function 8000_0001h, in bit 2 of ECX.

On processors that support SVM, CPUID function 8000_000Ah returns the SVM revision and feature flags in EAX, and the number of supported ASIDs in EBX, as shown in Table B-1. EDX is used to report feature flags, and ECX is currently reserved and set to zero.

**Figure B-1. SVM Revision and Feature Identification in EAX, Extended Function 8000_000Ah**

The fields returned in EAX are defined as follows:

- REVISION—Bits 7–0. An 8-bit ordinal representing the SVM REVISION number.
- Available—Bit 8. EAX bit 0 reads as zero. A VMM may use this bit to signal its presence by intercepting and emulating CPUID.

**Figure B-2. SVM Revision and Feature Identification in EBX, Extended Function 8000_000Ah**

The N_ASIDS field returned in EBX specifies the number of address space IDs supported by the implementation. The N_ASIDS value reported is one larger than the largest supported ASID value. The number of supported ASIDS need not be a power of two.

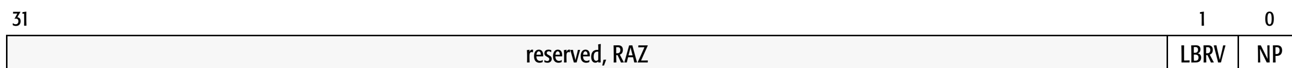


**Figure B-3. SVM Revision and Feature Identification in EDX, Extended Function 8000_000Ah**

The NP field in EDX indicates whether the nested paging facility is implemented.

The LBRV field in EDX indicates that LBR virtualization is implemented.

Future SVM features will be identified by a combination of revision number and feature flags in the currently reserved bits.

# Appendix C Layout of VMCB

## C.1 Layout of VMCB

The VMCB is divided into two areas—the first one contains various control bits including the intercept vector and the second one contains saved guest state.

Table C-1 describes the layout of the control area of the VMCB, which starts at offset zero within the VMCB page. The control area is padded to a size of 1024 bytes. All unused bytes *must* be zero, as they are reserved for future expansion. It is required that software initialize all bits to zero in any newly allocated VMCB.

**Table C-1. VMCB Layout, Control Area**

| Byte Offset | Bit(s) | Function |
|---|---|---|
| 000h | 0–15 | Intercept reads of CR0–15, respectively. |
| | 16–31 | Intercept writes of CR0–15, respectively. |
| 004h | 0–15 | Intercept reads of DR0–15, respectively. |
| | 16–31 | Intercept writes of DR0–15, respectively. |
| 008h | 0–31 | Intercept exception vectors 0–31, respectively. |

**Table C-1. VMCB Layout, Control Area** (continued)

| Byte Offset | Bit(s) | Function |
|---|---|---|
| 00Ch | 0 | Intercept INTR (physical maskable interrupt). |
| | 1 | Intercept NMI. |
| | 2 | Intercept SMI. |
| | 3 | Intercept INIT. |
| | 4 | Intercept VINTR (virtual maskable interrupt). |
| | 5 | Intercept CR0 writes that change bits other than CR0.TS or CR0.MP. |
| | 6 | Intercept reads of IDTR. |
| | 7 | Intercept reads of GDTR. |
| | 8 | Intercept reads of LDTR. |
| | 9 | Intercept reads of TR. |
| | 10 | Intercept writes of IDTR. |
| | 11 | Intercept writes of GDTR. |
| | 12 | Intercept writes of LDTR. |
| | 13 | Intercept writes of TR. |
| | 14 | Intercept RDTSC instruction. |
| | 15 | Intercept RDPMC instruction. |

**Table C-1. VMCB Layout, Control Area** (continued)

| Byte Offset | Bit(s) | Function |
|---|---|---|
| 00Ch (continued) | 16 | Intercept PUSHF instruction. |
| | 17 | Intercept POPF instruction. |
| | 18 | Intercept CPUID instruction. |
| | 19 | Intercept RSM instruction. |
| | 20 | Intercept IRET instruction. |
| | 21 | Intercept INT*n* instruction. |
| | 22 | Intercept INVD instruction. |
| | 23 | Intercept PAUSE instruction. |
| | 24 | Intercept HLT instruction. |
| | 25 | Intercept INVLPG instruction. |
| | 26 | Intercept INVLPGA instruction. |
| | 27 | IOIO_PROT—Intercept IN/OUT accesses to selected ports. |
| | 28 | MSR_PROT—intercept RDMSR or WRMSR accesses to selected MSRs. |
| | 29 | Intercept task switches. |
| | 30 | FERR_FREEZE: intercept processor "freezing" during legacy FERR handling. |
| | 31 | Intercept shutdown events. |

**Table C-1. VMCB Layout, Control Area** (continued)

| Byte Offset | Bit(s) | Function |
|---|---|---|
| 010h | 0 | Intercept VMRUN instruction. |
| | 1 | Intercept VMMCALL instruction. |
| | 2 | Intercept VMLOAD instruction. |
| | 3 | Intercept VMSAVE instruction. |
| | 4 | Intercept STGI instruction. |
| | 5 | Intercept CLGI instruction. |
| | 6 | Intercept SKINIT instruction. |
| | 7 | Intercept RDTSCP instruction. |
| | 8 | Intercept ICEBP instruction. |
| | 9 | Intercept WBINVD instruction. |
| | 10–31 | RESERVED, MBZ |
| 014h–03Fh | RESERVED, MBZ | |
| 040h | 0–63 | IOPM_BASE_PA—Physical base address of IOPM (bits 11:0 are ignored). |
| 048h | 0–63 | MSRPM_BASE_PA—Physical base address of MSRPM (bits 11:0 are ignored). |
| 050h | 0–63 | TSC_OFFSET—To be added in RDTSC and RDTSCP. |
| 058h | 0–31 | Guest ASID. |
| | 32–39 | TLB_CONTROL—Only two values are currently defined:<br>0—Do nothing<br>1—Flush TLB on VMRUN (all entries, all ASIDs) |
| | 40–63 | RESERVED, MBZ |

**Table C-1. VMCB Layout, Control Area** (continued)

| Byte Offset | Bit(s) | Function |
|---|---|---|
| 060h | 0–7 | V_TPR—The virtual TPR for the guest; currently bits 3:0 are used for a 4-bit virtual TPR value; bits 7:4 are MBZ.<br>*NOTE: This value is written back to the VMCB at #VMEXIT.* |
| | 8 | V_IRQ—If nonzero, virtual INTR is pending.<br>*NOTE: This value is written back to the VMCB at #VMEXIT.* |
| | 9–15 | RESERVED, MBZ |
| | 16–19 | V_INTR_PRIO—Priority for virtual interrupt. |
| | 20 | V_IGN_TPR—If nonzero, the current virtual interrupts ignores the (virtual) TPR. |
| | 21–23 | RESERVED, MBZ |
| | 24 | V_INTR_MASKING—Virtualize masking of INTR interrupts. See Section 2.17.1. |
| | 25–31 | RESERVED, MBZ |
| | 32–39 | V_INTR_VECTOR—Vector to use for this interrupt. |
| | 40–63 | RESERVED, MBZ |
| 068h | 0 | INTERRUPT_SHADOW—Guest is in an interrupt shadow; see Section 2.17.5.<br>*Note: This value is written back to the VMCB at #VMEXIT.* |
| | 1–63 | RESERVED, MBZ |
| 070h | 0–63 | EXITCODE |
| 078h | 0–63 | EXITINFO1 |
| 080h | 0–63 | EXITINFO2 |
| 088h | 0–63 | EXITINTINFO |
| 090h | 0 | NP_ENABLE—Enable nested paging. |
| | 1–63 | RESERVED, MBZ |
| 098h–0A7h | RESERVED. MBZ | |
| 0A8h | 0–63 | EVENTINJ—Event injection. (See Section 2.16, “Event Injection,” on page 33 for details.) |
| 0B0h | 0–63 | H_CR3—Host-level CR3 to use for nested paging. |

**Table C-1. VMCB Layout, Control Area** (continued)

| Byte Offset | Bit(s) | Function |
|---|---|---|
| 0B8h | 0 | LBR_VIRTUALIZATION_ENABLE<br>0—Do nothing.<br>1—Enable LBR Virtualization. |
| | 1—63 | Ignored. |
| All other fields up to 3FFh | RESERVED, MBZ | |

The state-save area within the VMCB starts at offset 400h into the VMCB page; Table C-2 below describes the fields within the state-save area; note that the table lists offsets *relative to the state-save area* (not the VMCB as a whole).

**Table C-2. VMCB Layout, State Save Area**

| Offset | Size | Contents | | Notes |
|---|---|---|---|---|
| 000h | word | ES | selector | |
| 002h | word | | attrib | |
| 004h | dword | | limit | |
| 008h | qword | | base | Only lower 32 bits are implemented; other bits are reserved. |
| 010h | word | CS | selector | |
| 012h | word | | attrib | |
| 014h | dword | | limit | |
| 018h | qword | | base | Only lower 32 bits are implemented; other bits are reserved. |
| 020h | word | SS | selector | |
| 022h | word | | attrib | |
| 024h | dword | | limit | |
| 028h | qword | | base | Only lower 32 bits are implemented; other bits are reserved. |
| 030h | word | DS | selector | |
| 032h | word | | attrib | |
| 034h | dword | | limit | |
| 038h | qword | | base | Only lower 32 bits are implemented; other bits are reserved. |
| 040h | word | FS | selector | |
| 042h | word | | attrib | |
| 044h | dword | | limit | |
| 048h | qword | | base | |

**Table C-2. VMCB Layout, State Save Area** (continued)

| Offset | Size | Contents | | Notes |
|---|---|---|---|---|
| 050h | word | GS | selector | |
| 052h | word | | attrib | |
| 054h | dword | | limit | |
| 058h | qword | | base | |
| 060h | word | GDTR | selector | reserved |
| 062h | word | | attrib | reserved |
| 064h | dword | | limit | Only lower 16 bits are implemented; other bits are reserved. |
| 068h | qword | | base | |
| 070h | word | LDTR | selector | |
| 072h | word | | attrib | |
| 074h | dword | | limit | |
| 078h | qword | | base | |
| 080h | word | IDTR | selector | reserved |
| 082h | word | | attrib | reserved |
| 084h | dword | | limit | Only lower 16 bits are implemented; other bits are reserved. |
| 088h | qword | | base | |
| 090h | word | TR | selector | |
| 092h | word | | attrib | |
| 094h | dword | | limit | |
| 098h | qword | | base | |
| 0A0h - 0CAh | | RESERVED | | |
| 0CBh | byte | CPL | | If the guest is real-mode then the CPL is forced to 0; if the guest is virtual-mode then the CPL is forced to 3. |
| 0CCh | dword | RESERVED | | |
| 0D0h | qword | EFER | | |
| 0D8h - 147h | | RESERVED | | |

**Table C-2. VMCB Layout, State Save Area** (continued)

| Offset | Size | Contents | Notes |
|---|---|---|---|
| 148h | qword | CR4 | |
| 150h | qword | CR3 | |
| 158h | qword | CR0 | |
| 160h | qword | DR7 | |
| 168h | qword | DR6 | |
| 170h | qword | RFLAGS | |
| 178h | qword | RIP | |
| 180h - 1D7h | | RESERVED | |
| 1D8h | qword | RSP | |
| 1E0h - 1F7h | | RESERVED | |
| 1F8h | qword | RAX | |
| 200h | qword | STAR | |
| 208h | qword | LSTAR | |
| 210h | qword | CSTAR | |
| 218h | qword | SFMASK | |
| 220h | qword | KernelGsBase | |
| 228h | qword | SYSENTER_CS | |
| 230h | qword | SYSENTER_ESP | |
| 238h | qword | SYSENTER_EIP | |
| 240h | qword | CR2 | |
| 248h - 267h | | RESERVED | |
| 268h | qword | G_PAT | Guest PAT—only used if nested paging enabled. |
| 270h | qword | DBGCTL | Guest DBGCTL MSR—only used if the LBR registers are virtualized. |
| 278h | qword | BR_FROM | Guest LastBranchFromIP MSR—only used if the LBR registers are virtualized. |
| 280h | qword | BR_TO | Guest LastBranchToIP MSR—only used if the LBR registers are virtualized. |

**Table C-2. VMCB Layout, State Save Area** (continued)

| Offset | Size | Contents | Notes |
|---|---|---|---|
| 288h | qword | LASTEXCPFROM | Guest LastExceptionFromIP MSR—Only used if the LBR registers are virtualized. |
| 290h | qword | LASTEXCPTO | Guest LastExceptionToIP MSR—Only used if the LBR registers are virtualized. |
| 298h to end of VMCB | | RESERVED | |

# Appendix D Intercept Exit Codes

When the VMRUN instruction exits (back to the host), an exit/reason code is stored in the EXITCODE field in the VMCB. The exit codes are defined in Table D-1. Intercept exit codes 0h–89h equal the bit position of the corresponding flag in the VMCB's intercept vector.

:

**Table D-1. SVM Intercept Codes**

| Code | Name | Cause |
|---|---|---|
| 0h–Fh | VMEXIT_CR[0–15]_READ | read of CR 0 through 15, respectively |
| 10h–1Fh | VMEXIT_CR[0–15]_WRITE | write of CR 0 through 15, respectively |
| 20h–2Fh | VMEXIT_DR[0–15]_READ | read of DR 0 through 15, respectively |
| 30h–3Fh | VMEXIT_DR[0–15]_WRITE | write of DR 0 through 15, respectively |
| 40h–5Fh | VMEXIT_EXCP[0–31] | exception vector 0–31, respectively |
| 60h | VMEXIT_INTR | physical INTR (maskable interrupt) |
| 61h | VMEXIT_NMI | physical NMI |
| 62h | VMEXIT_SMI | physical SMI (the EXITINFO1 field provides more information) |
| 63h | VMEXIT_INIT | physical INIT |
| 64h | VMEXIT_VINTR | virtual INTR |
| 65h | VMEXIT_CR0_SEL_WRITE | write of CR0 that changed any bits other than CR0.TS or CR0.MP |
| 66h | VMEXIT_IDTR_READ | read of IDTR |
| 67h | VMEXIT_GDTR_READ | read of GDTR |
| 68h | VMEXIT_LDTR_READ | read of LDTR |
| 69h | VMEXIT_TR_READ | read of TR |
| 6Ah | VMEXIT_IDTR_WRITE | write of IDTR |
| 6Bh | VMEXIT_GDTR_WRITE | write of GDTR |
| 6Ch | VMEXIT_LDTR_WRITE | write of LDTR |
| 6Dh | VMEXIT_TR_WRITE | write of TR |

**Table D-1. SVM Intercept Codes** (continued)

| Code | Name | Cause |
|---|---|---|
| 6Eh | VMEXIT_RDTSC | RDTSC instruction |
| 6Fh | VMEXIT_RDPMC | RDPMC instruction |
| 70h | VMEXIT_PUSHF | PUSHF instruction |
| 71h | VMEXIT_POPF | POPF instruction |
| 72h | VMEXIT_CPUID | CPUID instruction |
| 73h | VMEXIT_RSM | RSM instruction |
| 74h | VMEXIT_IRET | IRET instruction |
| 75h | VMEXIT_SWINT | software interrupt (INT*n* instructions) |
| 76h | VMEXIT_INVD | INVD instruction |
| 77h | VMEXIT_PAUSE | PAUSE instruction |
| 78h | VMEXIT_HLT | HLT instruction |
| 79h | VMEXIT_INVLPG | INVLPG instructions |
| 7Ah | VMEXIT_INVLPGA | INVLPGA instruction |
| 7Bh | VMEXIT_IOIO | IN or OUT accessing protected port (the EXITINFO1 field provides more information) |
| 7Ch | VMEXIT_MSR | RDMSR or WRMSR access to protected MSR |
| 7Dh | VMEXIT_TASK_SWITCH | task switch |
| 7Eh | VMEXIT_FERR_FREEZE | FP legacy handling enabled, and processor is frozen in an x87/mmx instruction waiting for an interrupt |
| 7Fh | VMEXIT_SHUTDOWN | Shutdown |
| 80h | VMEXIT_VMRUN | VMRUN instruction |
| 81h | VMEXIT_VMMCALL | VMMCALL instruction |
| 82h | VMEXIT_VMLOAD | VMLOAD instruction |
| 83h | VMEXIT_VMSAVE | VMSAVE instruction |
| 84h | VMEXIT_STGI | STGI instruction |
| 85h | VMEXIT_CLGI | CLGI instruction |
| 86h | VMEXIT_SKINIT | SKINIT instruction |
| 87h | VMEXIT_RDTSCP | RDTSCP instruction |

**Table D-1. SVM Intercept Codes** (continued)

| Code | Name | Cause |
|---|---|---|
| 88h | VMEXIT_ICEBP | ICEBP instruction |
| 89h | WBINVD | WBINVD instruction |
| 400h | VMEXIT_NPF | Nested paging: host-level page fault occurred (EXITINFO1 contains fault errorcode; EXITINFO2 contains the guest physical address causing the fault.) |
| –1 | VMEXIT_INVALID | Invalid guest state in VMCB |

# Appendix E New and Changed MSRs

SVM introduces new MSRs and adds new fields to existing MSRs as summarized in Table E-1. These new MSRs and fields are available regardless of whether SVM is enabled in EFER.SVME.

**Table E-1. SVM New MSRs**

| Name | Address | Access | Description |
|---|---|---|---|
| VM_CR | C001_0114 | r/w | Security-related control bits. |
| IGNNE | C001_0115 | | Set the processor-internal IGNNE state. |
| SMM_CTL | C001_0116 | w/o | Explicit control over SMM state and signals. |
| VM_HSAVE_PA | C001_0117 | r/w | Physical address of host state-save area. |

## E.1 VM_CR MSR (C001_0114h)

The read/write VM_CR MSR controls certain “global” aspects of SVM. The layout of the MSR is shown in Figure E-1.

**Figure E-1. Layout of VM_CR MSR (C001_0114h)**

The individual fields are as follows:

- DPD—Bit 0. If set, disables HDT and certain internal debug features.
- R_INIT—Bit 1. If set, non-intercepted INIT signals are converted (“redirected”) into an #SX exception.
- DIS_A20M—Bit 2. If set, disables A20 masking.

## E.2 IGNNE MSR (C001_0115h)

The read/write IGNNE MSR is used to directly set the state of the processor-internal IGNNE signal. 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.

## E.3 SMM_CTL MSR (C001_0116h)

The write-only SMM_CTL MSR provides software control over SMM signals.

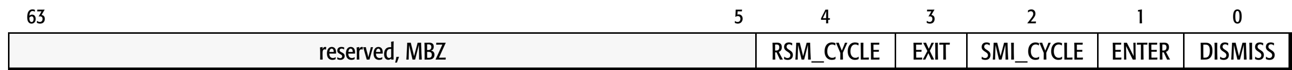


**Figure E-2. Layout of SMM_CTL MSR (C001_0116h)**

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 the BIOS has locked the SMM control registers.

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.

## E.4 VM_HSAVE_PA MSR (C001_0117h)

The 64-bit read/write VM_SAVE_PA MSR holds the physical address of a 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.

## E.5 Changes to Existing MSR

The following existing MSR has been changed:

### E.5.1 EFER

- SVME—Bit 12. Enables the SVM extensions. When this bit is zero, the new SVM instructions cause #UD exceptions. EFER.SVME defaults to a reset value of zero. The effect of turning off EFER.SVME while a guest is running is undefined; therefore, the VMM should always prevent guests from writing EFER.

## E.6 LocalAPIC Registers

The 256-bit IER and the SEOI command register are made available via new registers in the second APIC page, at the offsets defined in Table E-2, “Secure-VM localAPIC Registers,” on page 110 below.

**Table E-2. Secure-VM localAPIC Registers**

<table>
<tr><th>Name</th><th>Address</th><th>Description</th></tr>
<tr><td>APIC Feature Register</td><td>0x400</td><td>Extended APIC feature register (read only); see Section E.7.</td></tr>
<tr><td>APIC Control Register</td><td>0x410</td><td>Extended APIC control register (read/write); see Section E.7.</td></tr>
<tr><td>SEOI</td><td>0x420</td><td>Specific End-of-Interrupt register (write only).<br>Software writes this register with an 8-bit vector number in the low 8 bits to cause an end-of-interrupt cycle to be performed for the specified vector. If no interrupt is in service for the specified vector, the behavior is undefined.</td></tr>
<tr><td>IER0</td><td>0x480</td><td rowspan="4">The 256-bit IER (Interrupt Enable Register) is made available as eight 32-bit APIC registers; the layout is little-endian (IER0 contains IER bits 0–31, IER1 contains bits 32–63, and so on).</td></tr>
<tr><td>IER1</td><td>0x490</td></tr>
<tr><td colspan="2">...</td></tr>
<tr><td>IER7</td><td>0x4F0</td></tr>
</table>

## E.7 APIC Feature Identification, and Enabling

Secure virtualization also depends on new APIC features. These are identified in the new extended APIC feature register and must be enabled via the new extended APIC control register. Bit 31 in the existing APIC version register (offset 30h) indicates whether the extended APIC register space is present.

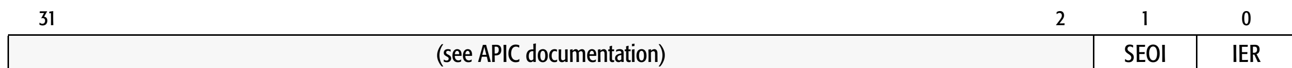


**Figure E-3. Extended APIC feature register.**

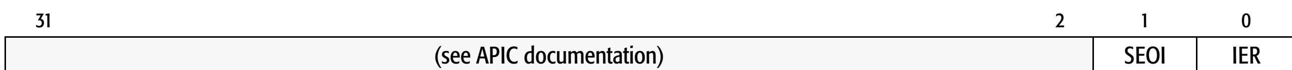


**Figure E-4. Extended APIC control register.**

The IER and SEOI fields in these two registers indicate the presence of, and enable, the new APIC SEOI and IER registers, respectively.